R Programming for Mass Spectrometry

Effective and Reproducible Data Analysis

Randall K. Julian

WILEY

Table of Contents

List of Illustrations

List of Tables

R Programming for Mass Spectrometry

Effective and Reproducible Data Analysis

Randall K. Julian

Indigo BioAutomation, Inc.
Carmel, IN, USA

WILEY

Copyright © 2025 by John Wiley & Sons, Inc. All rights reserved, including rights for text and data mining and training of artificial intelligence technologies or similar technologies.

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.

Published simultaneously in Canada.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750–8400, fax (978) 750–4470, or on the web at www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748–6011, fax (201) 748–6008, or online at http://www.wiley.com/go/permission.

Trademarks

Wiley and the Wiley logo are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates in the United States and other countries and may not be used without written permission. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc. is not associated with any product or vendor mentioned in this book.

Limit of Liability/Disclaimer of Warranty

While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Further, readers should be aware that websites listed in this work may have changed or disappeared between when this work was written and when it is read. Neither the publisher nor authors shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services or for technical support, please contact our Customer Care Department within the United States at (800) 762–2974, outside the United States at (317) 572–3993 or fax (317) 572–4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic formats. For more information about Wiley products, visit our web site at www.wiley.com.

Library of Congress Cataloging-in-Publication Data has been applied for.

Print ISBN: 9781119872351
ePDF ISBN: 9781119872368
ePub ISBN: 9781119872399
oBook ISBN: 9781119872405
Cover Design: Wiley
Cover Image: © Valery Rybakow/Shutterstock

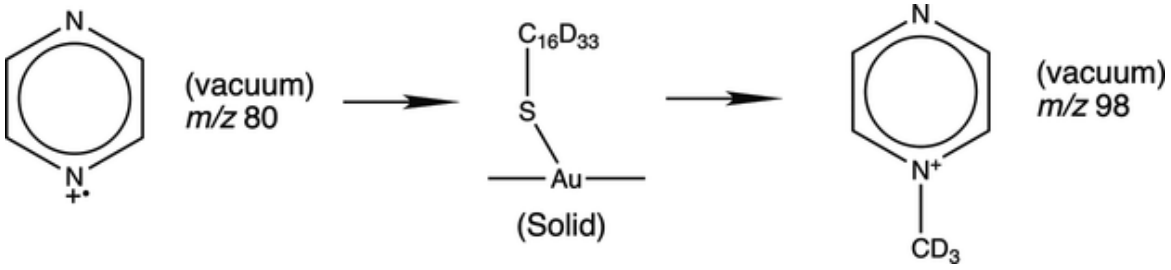
To Lauren:

You made it possible for me to start this book. Thank you for all your help while I took time away from you to finish it.

Foreword

This is not only a book about R coding, rather it is an indicator of where we are in the journey from an analog real world to one ruled by digital data. We stand at an inflection point on the science/technology pathway. Behind us lies the golden ages of empirical discovery science, more attractive as it recedes into the distant past...perhaps to the steppes of Central Asia, where the night skies whispered their thousand questions and imaginative answers – supported by primitive observations – were provided in response. Now, many mysteries have been removed and replaced by high-quality data. This is not an unalloyed good, and there must be some regret, even among those who do not march under the “Stop the Technology Madness” banner.

Author Randy Julian’s own life in science has covered the before and after of this inflection timepoint, between the sparse-data era and the current rich-data hegemony. He started his PhD in Chemistry in 1990 with measurements of reactive collisions of polyatomic ions impacting monolayer surfaces in a vacuum, as shown by the simple bond-formation reaction illustrated in *Surface reactions and surface-induced dissociation of polyatomic ions at self-assembled organic monolayer surfaces* [1] / American Chemical Society.



Randy Julian was working in a basement lab where for decades the time taken for data to emerge from instruments was so slow that it was possible to think about it as it arrived, to adjust the planned experiment, and even to draft an outline of a planned scientific publication. A perfect resonance existed between the instrumentation and the experimenter. The dramatic nature of the changes then underway is seen in work from Randy Julian's second year at Purdue, when he coded a multiparticle numerical simulation of the trajectories of realistic numbers of ions under the actual electrical fields of a mass spectrometer and matched it to the experimental mass spectrum, including peak shapes. Then, in a tour de force experiment, *Large Scale Simulation of Mass Spectra Recorded with a Quadrupole Ion Trap Mass Spectrometer* [2], likely one of the earliest studies in analytical chemistry to use supercomputers, he simulated the individual ion dynamics of a system of thousands of ions in an electric field and subjected to collisions by harnessing computers ranging from the Connection Machine at Los Alamos National Laboratory to the Cray to solve the classical equations of motion in sub-microsecond increments using parallel and vector processing.

Three years later, as a fresh employee at Eli Lilly & Co. in Indianapolis, Dr. Julian persuaded management to provide him with a factory of mass spectrometers so that he could examine millions of chemical constituents generated by Lilly's massive collection of natural product extracts. Many current drugs come from such sources, making it a worthwhile challenge. It took over two years to record the mass spectra of all these samples. Dr. Julian's subsequent career has been along similar lines, only faster and wider in scope. His company (Indigo BioAutomation, Inc.) currently processes and validates some 300 million mass spectrometry/liquid chromatography clinical tests a month.

They perform data quality assurance for every major diagnostics laboratory in the United States – it is likely that your physician has received data analyzed by Indigo. Had Randy lived in California instead of Indiana, a career like his would have been emblazoned on T-shirts and funded rock concerts. What does one say of a scientist whose command of experimental data is such that the difference, at six sigma, between Gaussian and Lorentzian curves can form the basis for solid medical decisions.

Randy Julian’s career has been centered on analytical chemistry, a topic with wide societal applications and now often referred to as measurement science to avoid the use of offensive terms. The subject has two components; the instrumentation used to make the measurement and the digital processing used to maximize the quality of the information output. Data science provides access to the information produced by instrumentation acting on chemicals. Dr. Julian’s work and this volume focus on the latter, but a major impact on the design and utilization of novel instrumentation is likely too. Instruments like mass spectrometers record spectra on an entire world of chemicals, in atomic, ionic, or molecular form, as pure materials or organized into biopolymers, or in the form of neurons or biofluids, or whole organisms. The rapid growth in data science and technology, which is central to this book, has allowed the extraction of detailed information, often biomedical in nature, from this data. This development transformed the way science is done and how it is reported. The author himself played a significant role in persuading the editors of leading journals to require that the published data on which published conclusions were to be based be archived and widely available. This has allowed quality checks on data to be performed post facto to ensure the validity of the conclusions. Not surprisingly, a “crises of reproducibility” is being experienced as some important

studies fail to withstand the withering examination now possible.

Unarguably, the landscape of science and technology has undergone highly significant changes in the past three decades, but what fueled this daemon? Perhaps the beast is omnivorous, requiring physics and microdevices and systems of knowledge and algorithms to be combined and harnessed to the instrumentation that provides the analytical chemical data. The traditional tools of academia – books and lectures – were successful in driving this transformation. Books like Diefendorfer's ground-breaking *Principles of Electronic Instrumentation* addressed the marriage of physics and EE to analytical measurements. A required graduate course in the Analytical Chemistry program at Purdue University has also made a remarkable contribution to the combination of data handling and chemical measurement. Now in its fifth decade, CHM 621 was initiated and taught for many years by Prof. Fred Lytle, and it has served as the foundation upon which hundreds of PhD students built their experiments and data interpretations. Randy Julian was enrolled in this course in 1990, and during the next two years, Randy taught a course on how to write programs based on the principles taught in Fred's course. The course has had a wide impact over the years in samizdat lecture notes. After Randy left Eli Lilly & Co. and started Indigo BioAutomation, Fred retired from Purdue to join Randy at Indigo. In this book, Randy has updated his Purdue programming course notes and combined them with his experience over the past three decades. Meanwhile, Fred Lytle has also been reworking his lecture notes and will soon complete his book. Over the years, they have made a formidable team.

This book will help create big-data scientists, but it is likely also to stimulate readers to improve the next generation of analytical instruments. Data science has already allowed

mass spectrometry to tame the ‘wildness’ of biology and produce reliable, actionable information. Randy Julian’s graduate work concerned a single bond-forming reaction occurring on a nearly perfectly characterized surface, and the readout consisted of intensity measurements in two channels of mass using essentially time-invariant signals. The data processing tools in this book accommodate millions of analytes from millions of patients. As Brison Shira, a current PhD student, commented, “Randy once applied MS to single analytes, and he now applies it to populations of patients.” There is power in this book.

R. Graham Cooks
West Lafayette, June 2024

Preface

This book will teach you how to analyze data generated by mass spectrometers using R [3]. The modern mass spectrometer is a marvel of science and engineering. What was once an imposing instrumental method of analysis with limited application has now become a workhorse in research and industry. At the same time, the boundaries of measurement capability are rapidly expanding with each new generation of analyzer, detector, and ionization method. At the outer limits are instruments that still act like temperamental thoroughbreds, which, on any given day, deliver extraordinary results or confusing noise. Workhorse instruments, on the other hand, often operate in a factory-like mode, producing data that is changing how we discover and develop drugs, diagnose and treat diseases, and understand our drinking water, food, the oceans, and the atmosphere. Mass spectrometers are used for measurements in such large numbers that ensuring problems with data analysis do not corrupt results is critical. Well below the limits of performance, mass spectrometers can generate such huge volumes of complex data that the analysis is beyond simple statistics and enters the domain of data science. For nearly all of the uses of mass spectrometry, there is a need for more advanced and more reproducible data analysis than can be done in spreadsheets.

The Main Goal of this Book

The main goal of this book is to show how to analyze mass spectrometry data effectively and reproducibly using the R programming language. Any mass spectrometrist can learn to go beyond spreadsheets and build data analysis solutions using R in a reasonable amount of time. My approach will be

like climbing a ladder. Through the lens of mass spectrometry, I will start by introducing native features of the R language. On the next rung are the packages that simplify data storage and retrieval, data manipulation, statistics, and visualization. The next step uses modules originally created to help with molecular biology tasks that also work with data from mass spectrometers. Further up the ladder are mass spectrometry-specific modules used to perform data manipulation and analysis for data generated specifically by mass spectrometers. Beyond that, the ladder goes on, but this book will end on the machine learning rung, far from the top.

Because the intended audience for this book is relatively broad, different sections will be of more value to some readers than others, so hopefully, familiar parts can be skipped. The example code is intended to show techniques and methods for analyzing mass spectrometry data that are effective and reproducible. However, within the example code, I hope you will find solutions to common problems that repeatedly appear in the analysis of mass spectrometry data. A word of warning: this is a code-heavy book, and the code is meant to be read. If some of the syntax is unfamiliar, please refer to some of the amazing books on R data analysis available. Along the way, I will provide pointers on where to find more information outside the scope of this book. I hope that some of the references will provide additional reading in areas of interest.

What You Will Learn

You will learn to analyze mass spectrometry data using R in a way that is widely accepted and supported by the data science community. In addition, you will learn to use various packages beyond the main R program to organize data, programs, and reports. Using examples from mass

spectrometry research, you will learn how to understand your data, wrangle it into easy-to-manage structures, perform exploratory data analysis, visualize, and then analyze it to produce reproducible findings. You will also learn how to integrate description and discussion with data and code so you can build web pages and manuscripts about your analysis that other researchers can reproduce.

Conventions

This book contains three types of text: regular text, examples of R code, and output from code. In the body of the book, references to elements of the R programming language will appear in a `monospaced text` typeface. Code that you can type in and execute will be in a gray box and look like this:

```
a <- 1 + 1
```

Also, R implicitly calls `print()` when a single variable name is given. But sometimes I will explicitly call print.

```
a
```

```
## [1] 2
```

When generating output, console output is shown with two hash symbols, ##. These symbols allow text to be distinguished from other text in this book and copied to into R, where the ## symbols are treated as comments. If the output is a series, there will be a leading number like `[1]` indicating the starting index of the series printed on that line.

Getting Started

To get started, download the latest version of R for your machine. R is an open-source project and is free. Installation packages are available for macOS, Windows, and Linux. The R project webpage is *https://r-project.com*, and you can find installation packages there.

I worked on the example for this book using the RStudio integrated development environment [4]. It is also free and also runs on macOS, Windows, and Linux. R can be used with its own user interface or the command line. It can also be used from other environments and editors. The examples should work with whatever you are comfortable with.

Once you have R installed, there are several add-on packages that you will need to run the examples. RStudio makes it easy to install and update packages from the R project repository called CRAN. In some chapters, I will give directions for the installation of select packages, but the following are the basic packages needed for most of the examples in this book.

```
list.of.packages <- c("tidyverse", "tidymodels",
"rmarkdown")
install.packages(list.of.packages)
```

For mass spectrometry-specific packages, I will mostly rely on the Bioconductor project repository [5]. Unlike CRAN, Bioconductor packages are built around a core set of packages, and the entire collection is designed to be as interoperable as possible. In addition, the Bioconductor project is versioned as a whole and operates on its own release schedule to help improve interoperability and consistency. The `BiocManager` package [6, 7] is a specialized package management system used to install packages from Bioconductor. To install Bioconductor packages, install its package manager first:

```
install.packages("BiocManager")
```

Several packages are used for reading raw mass spectrometry data: `mzR` [8–13], used for fast, low-level reading of open format XML files. `MSnbase` [14, 15] and `Spectra` [16] are both higher level packages that can use `mzR`, `MsBackendMgf` [17], `MsBackendMsp` [18], and other *backends* (format-specific file interfaces) to read data. Most packages used in this book are from either Bioconductor or CRAN; however, in Section 5.2.4, I will show how to install packages from other repositories, specifically from the source code repository system called GitHub.

To install Bioconductor packages, you just use the `install()` function from the `BiocManager` package:

```
bioc.packages <- c("MSnbase", "Spectra", "mzR",
"MsBackendMgf", "MsBackendMsp")
BiocManager::install(bioc.packages, update=FALSE)
```

After you have these packages installed and can run RStudio, you are ready to start.

About the Code and Examples in this Book

The code and information in this textbook have been carefully reviewed and tested to the best of the author's ability. However, as with any programming resource, errors or inaccuracies may occur. The author and publisher make no warranties or representations regarding the accuracy, completeness, or suitability of the code and information presented.

Readers who use or implement any code from this book do so at their own risk. Neither the author nor the publisher shall

be held liable for any damages or consequences arising from the use of the information or code contained herein.

It is recommended that readers thoroughly test and validate any code before using it in production environments.

The examples, citations, and references to external work or products in this book are used for instructional purposes only and do not constitute an endorsement by the author or publisher. Such references are provided solely to illustrate concepts and techniques discussed in the text.

Acknowledgments

I have many people to thank for being able to share this book with you. First, I would like to thank everyone at Indigo BioAutomation. I am fortunate to work on such an incredible team. I want to thank Prof. Fred Lytle, who mentored me as a graduate student and became an even more significant influence and friend after retiring from Purdue and coming to Indigo. I'd also like to thank Rick Higgs, who got me started with R in the 1990s and introduced me to machine learning before it was a buzzword. None of this work would have been possible without my PhD adviser, Prof. R. Graham Cooks, who, besides teaching me mass spectrometry, allowed me to teach a programming class to chemistry graduate students before finishing my thesis and then arranged for me to teach a graduate-level data science course at Purdue. I am forever thankful for my experiences at Purdue. Thank you to Russ Grant, Nigel Clarke, Brian Rappold, Patrick Mathias, and Shannon Haymond, with whom I've worked and taught and are now dear friends. You've all greatly impacted my development as a scientist and a person. I would especially like to thank Stephen Master who provided valuable comments and suggestions to an early draft of the book. I'd also like to thank the Harrold family: Dave, Chris, and Amber, for running the *Mass Spectrometry & Advances in the Clinical Laboratory* (MSACL) conference and giving me the opportunity to teach short courses and help expand the data science program. So many people have helped in my development that I cannot thank them all here, but no one helped me more than my parents, who supported my early addiction to programming computers. Thanks, Dad, for teaching me about computer hardware, and Mom, for putting up with

me staying up all night with my brother Mike writing computer games.

About the Companion Website

This book is accompanied by a companion website:

www.wiley.com/go/julianrprogramming

The website includes:

- Figures
- Codes
- Data

Chapter 1
Data Analysis with R

This chapter will give an overview of R, the base R libraries, the Tidyverse packages, the Bioconductor project, and RMarkdown. I will also describe R scripting and the RStudio integrated development environment (IDE). If you are familiar with these topics, feel free to skip this introduction. The goal is for you to have a working R development environment, understand the basic ideas behind the tidyverse and the Bioconductor projects, and be able to use libraries and packages from both Comprehensive R Archive Network (CRAN) and Bioconductor.

1.1 Introduction

The R programming language [19] is an open-source project inspired by both the S language [20] and Scheme [21]. Over the decades since its initial development, the data science community has embraced R to an extraordinary level. While you can use almost any programming language for data science, R was one of the first freely accessible languages to make statistics its primary focus. Statistics is one of those subjects in which experts are practically necessary. For a nonstatistician, having highly reliable statistical functions improves the quality of analysis, especially compared to writing statistical algorithms from scratch. R is an interpreted language, and a community of dedicated experts continually updates it. Some of the best computational statisticians in the world actively support the statistical functions available in R. On top of these incredible contributions, the applied statistical community has created a fantastic array of add-in packages to handle specific analysis requirements. The core components of R and its vast library

of packages allow for a wide range of statistical and visual analyses.

So why learn a programming language like R instead of just using a spreadsheet program like Excel? That's a good question, which has a good answer. Excel has become very powerful over the years but has significant drawbacks for demanding data analysis tasks. First, each cell in a spreadsheet can be any data type; you can't tell what it is by looking. A cell might look like a date, but it might also be a string. Or, it could have a formula that produces the content. The equation likely references other cells and is often created by cutting and pasting. Performing calculations this way makes all but the most trivial spreadsheets challenging to test and debug. Despite the limitations of spreadsheets, we almost all use spreadsheets for some tasks. But we have all experienced some errors when working with spreadsheets. This lack of robustness keeps most people working in data science away from spreadsheets. The one thing spreadsheets seem particularly good at is creating and editing text files (usually saved and loaded as comma-separated value or "CSV" files), but even here, trouble is just waiting to strike. CSV files often have a header that gives the names of the columns. When loaded into a spreadsheet, this row becomes another row in the sheet. When a spreadsheet has no header row in the data, a text file created from it will also have no header. At first, this may seem trivial, but since the top of a spreadsheet shows the names of the columns assigned by the program, the application-specific column names need to appear as text in the first data row. If someone reads the resulting text file assuming that a header is present and it's not, then the first row of numeric data can be consumed as the header, and all of the data will then be loaded as if the read function skipped the first row. Again, while it sounds trivial, but mishandling header rows in spreadsheets has done tremendous damage to data analysis over the years. If

you use a spreadsheet to help edit data, be careful in later analysis steps.

Another famous problem with spreadsheets is that some information will be interpreted by programs like Excel as dates when they are strings that look like dates. Excel will quietly change your data without warning, and if you don't catch it, then when you save your file, some of the values may be corrupted by the string-to-date conversion. You can see a concrete example of this error: load a file that contains chemical abstract service (CAS) registry numbers. If you load the CAS number `6538-02-9` into Excel, for example, it will convert it into the date `2-9-6538`, and then when you convert it to a number, you will get `1694036` (this is from an actual Microsoft support case from 2017 which I reproduced at the time of writing). People doing data science use spreadsheets all the time, but you have to be very careful and look for at least these two big problems.

You can perform data analysis in any computer programming language. While I will not cover them, Python and Julia are first-rate languages and good choices for any data analysis project. Python, in particular, has been the go-to language for the exploding machine-learning community. Like R, Python is an interpreted language with excellent community support. Many data analysts learn R and Python and switch between them depending on the project. The main difference is that the central focus of statistical analysis in R, whereas Python is a general programming language with good statistical libraries. Julia is different. Its community motto is: "Walk like Python; Run like C." Julia is faster than Python and R in most cases, depending on the libraries you use. I encourage everyone working in data analysis to become familiar with Python and R. It will also pay to be aware of Julia. All three languages will run as automated scripts, and all three have development environments for writing more complex programs. Recently, there has been a trend toward using a notebook environment for programming, especially for

Python with its almost addictive Jupyter Notebook system. Notebook environments allow mixing code with text by putting each in different types of cells. Opening a notebook and typing in natural language in some cells and code in others is a very agile way to work with code and data. However, working in a notebook can sometimes produce a mindset that you are not actually developing a program but just a document with some code mixed in. That mindset can lead to a lot of cut-and-paste programming, and other programming practices can make for messy and hard-to-reproduce analysis. It's not a defect of the notebook concept but something to guard against when using them. Some people will start in a notebook environment, and if the program becomes complex, they will switch to an IDE. The method of mixing natural language text and code is so powerful that the approach can be used directly in the RStudio IDE for R. With RStudio, you don't have to choose between working in an IDE or a notebook since both practices are supported.

R supports mixing natural language and code using the `knitr` package to implement *literate programs* [22], introduced below. One of my main objectives here is to show analysts how to improve the reproducibility of mass spectrometry data analysis. I will return to using R combined with `knitr` and RMarkdown to create literate programs throughout the book.

1.2 Modern R Programming

This section will teach you how to use R as a scripting language for batch processing and from within the IDE RStudio. Further, you will learn about the base packages of R and the modern approaches to data management and analysis introduced by the `tidyverse` collection of packages, including the plotting system provided by the `ggplot2` package.

1.2.1 R as a Scripting Language

As described earlier, R belongs to the family of interpreted languages. In UNIX-type systems, languages like Perl, Shell-scripts, Ruby, and Python can be run as scripts by the OS. Any R program can be typed into a text editor and run from the command line as a script.

Take this trivial program:

```
# This program should be saved in a file called
"hello.R"

print("Hello, R")
```

To run this example and have the output display on in the console, you can use the `Rscript` program:

```
Rscript hello.R
```

The output to the console will be:

```
[1] "Hello, R"
```

When you want to run an R program as part of a noninteractive, automated process, you can use *batch mode*. Running in batch mode allows you to pass arguments to the program and have the output go to a file rather than the console. Starting the R interpreter with the options `CMD BATCH` puts the program into batch mode. The R interpreter will assume that the working directory is the current directory, which you may need to change depending on how your system runs automated scripts.

```
# leading './' is for the macOS, change this for your
OS

R CMD BATCH ./hello.R
```

This will send all of the output of the program to a file called `hello.Rout` In this case, it is the output:

```
R version 4.3.1 (2023-06-16) -- "Beagle Scouts"
Copyright (C) 2023 The R Foundation for Statistical
Computing
Platform: aarch64-apple-darwin20 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain
conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English
locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in
publications.

Type 'demo()' for some demos, 'help()' for on-line help,
or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> print("Hello, R")
[1] "Hello, R"
>
> proc.time()
   user  system elapsed
  0.130   0.037   0.150
```

While running R programs as scripts from the command line is helpful, it is much more typical to write and run programs

in an IDE. IDEs have been available for languages like C/C++ and other compiled languages for decades. Various R installation packages also come with an IDE called the R GUI. It is sufficiently powerful to allow anyone to get started, while missing many convenient features of RStudio. As data science has matured, additional tools are now available. In this book, I will focus on using R via the powerful and popular IDE for R called RStudio.

1.2.2 RStudio

Software development and engineering tools have matured over the years. With the arrival of high-speed hardware, there has been a revival of interpreted languages, like R. Interpreted languages allow development environments extra flexibility by using real-time interpretation to assist the programmer while writing a program. Since R is intended primarily as a statistical analysis language, a good IDE makes it easy to write and test code, see plots, and examine data. RStudio extends the concept of the IDE by integrating with the powerful report-generating packages used for reproducible research. RStudio is free (as in *open* and *beer*), and the RStudio team has shown itself to be a dedicated contributor to R, supporting some of the most important packages, including `tidyverse` and many others.

There are versions of RStudio for Windows, Mac, and many Linux distributions, and since it is open source, you can build it yourself from the source code if there is no binary distribution for your OS. To get started, go to the RStudio website (rstudio.com) and select the download for your machine. Each binary has an installer with instructions. Once you have RStudio installed, you can run it, and you should see something like Figure 1.1.

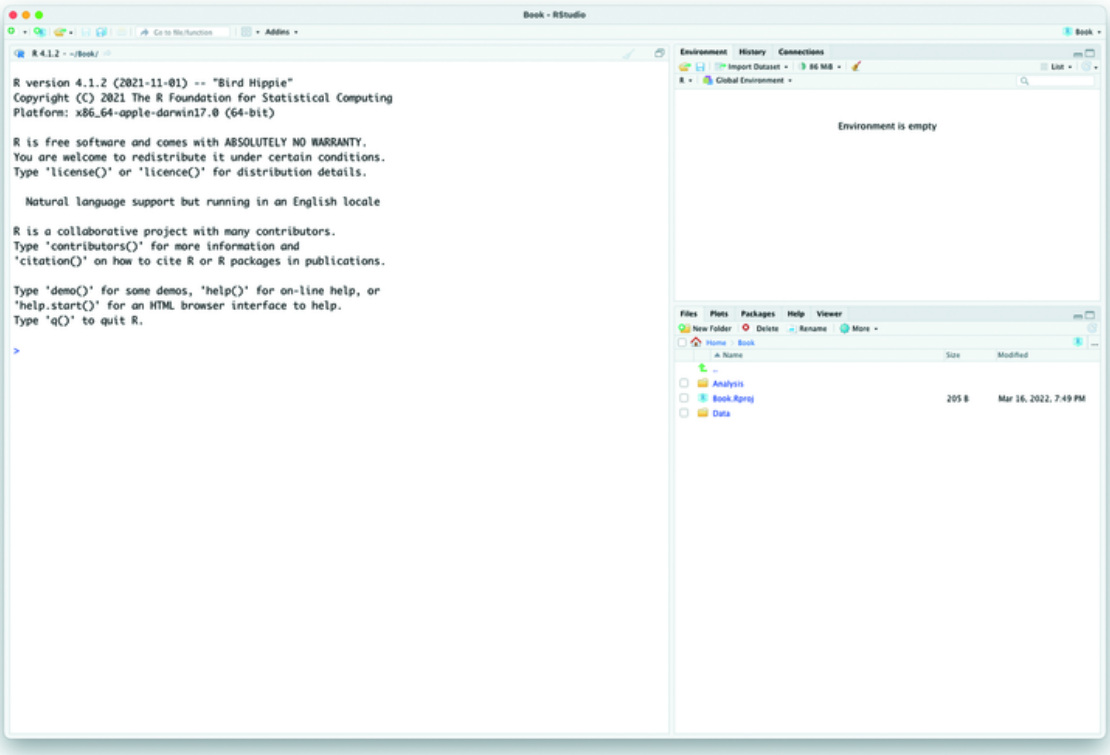


Figure 1.1 RStudio startup interface.

1.2.3 Base R

Much of the power of R comes from the large collection of base libraries developed by the R community. Currently, there are 14 *base packages* and 15 *recommended packages* [23]. These packages allow users to perform various statistical analyses and data visualization. The various distributions of R incorporate the base and recommended packages. Over time, as new packages are developed, they are usually shared using a repository called the CRAN, which was created to make them available to the R community. The recommended packages come from CRAN and are installed with most distributions of R. Together, the base and recommended packages have become known as Base R, which is usually sufficient for most statistical analysis and data plotting tasks.

Base R provides the mechanisms for essential data manipulation on several fundamental data types. Beyond scalar variables, base R allows you to manipulate vectors, sequences, matrices, lists, and strings. Probably the most significant data type provided in base R is the `data.frame`. A `data.frame` is a rectangular table where each column is assigned a data type and can have a name. Each row can also be named, even if the name is simply a row number. What makes Data Frames (and other newer data types derived from the Data Frame) powerful is that data can be manipulated and selected with conditional statements. The syntax of Data Frame operations can be slightly confusing, but learning it allows you to work with data in R in ways that are much easier than most programming languages.

1.2.4 Basics of Data Frames

To demonstrate how to use the `data.frame`, I've extracted a part of the Human Metabolite Database [24–28] into a CSV file. CSV files are simple text files that usually contain the column names in the first row. The base R function to read a CSV file is conveniently named `read.csv()`.

```
hmdb_df <-
read.csv(file.path("data","hmdb_urine_metabolites.csv")
)

str(hmdb_df, width=72, strict.width="cut")
```

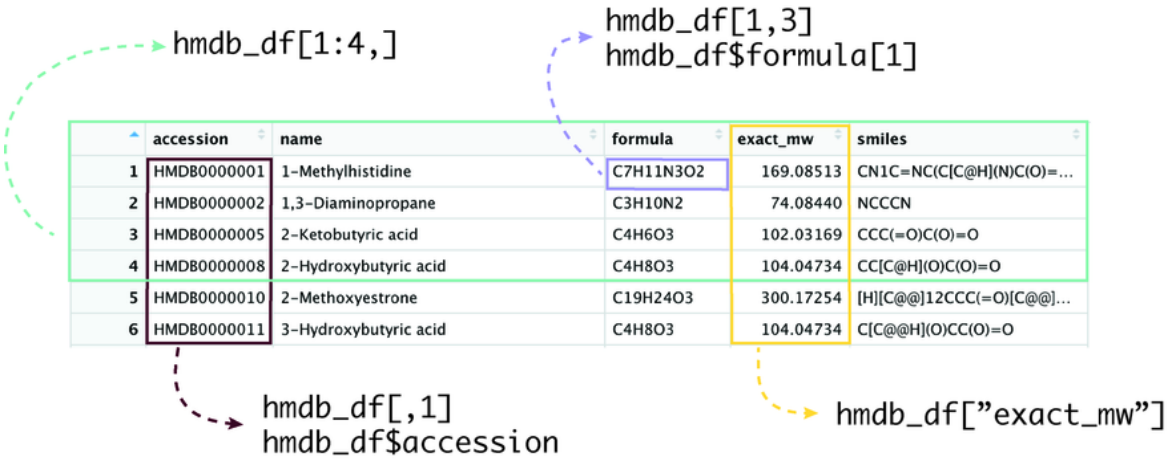
```
## 'data.frame':    4692 obs. of  6 variables:
##  $ accession: chr  "HMDB0000001"  "HMDB0000002"
"HMDB0000005" "HMDB000"..
##  $ name     : chr  "1-Methylhistidine"  "1,3-
Diaminopropane" "2-Ketobu"..
##  $ formula  : chr  "C7H11N3O2"  "C3H10N2" "C4H6O3"
"C4H8O3" ...
##  $ exact_mw : num  169.1 74.1 102 104 300.2 ...
```

```
##  $ smiles   : chr  "CN1C=NC(C[C@H](N)C(O)=O)=C1"
"NCCCN" "CCC(=O)C(O)"..
##  $ status   : chr  "quantified"  "quantified"
"quantified" "quantified"..
```

The `str()` function shows the structure of any R object, and in this case, it shows that `nmdb_df` is a `data.frame` with 4692 rows (observations) and 6 variables (columns). The columns have both names and types. Here the column names are given next to the `$` symbol, followed by the data type for that row. The `str()` function also shows a sample of the data in each column.

Once data is in a `data.frame`, there are several ways to access specific elements, depending on your needs. For example, you can access data by row, column, or specify both.

Figure 1.2 shows the syntax to access elements of a `data.frame`.



	accession	name	formula	exact_mw	smiles
1	HMDB0000001	1-Methylhistidine	C7H11N3O2	169.08513	CN1C=NC(C[C@H](N)C(O)=...
2	HMDB0000002	1,3-Diaminopropane	C3H10N2	74.08440	NCCCN
3	HMDB0000005	2-Ketobutyric acid	C4H6O3	102.03169	CCC(=O)C(O)=O
4	HMDB0000008	2-Hydroxybutyric acid	C4H8O3	104.04734	CC[C@H](O)C(O)=O
5	HMDB0000010	2-Methoxyestrone	C19H24O3	300.17254	[H][C@@]12CCC(=O)[C@@]...
6	HMDB0000011	3-Hydroxybutyric acid	C4H8O3	104.04734	C[C@@H](O)CC(O)=O

Figure 1.2 Accessing elements of a `data.frame`.

A `data.frame` is a collection of rows and columns. Each column has a specific data type and can have a name. Each row can also have a name that can be used to access particular observations. Columns and rows can also be accessed by their index value, which is an integer number. In R, it's important to remember that indexing always starts at 1 rather than 0, as in many other languages.

Accessing elements of a `data.frame` uses the square bracket notation: `df[row,column]`. In 1.2, you can see that selecting the element in the first row and the third column (the value of the chemical formula in the first row) is simply `hmdb_df[1,3]`. Besides an index value, the third row has the name `formula`. R can use the `$` symbol to access a column's name. `hmdb_df$accession` returns an array of values from the first column, and `hmdb_df$formula` returns an array of values from the third column. The first elements of the array returned in case of formula can be accessed with the same `[]` notation so that `hmdb_df$formula[1]` returns the first element in the `formula` column. The ability to specify columns by name is helpful when accessing data by name, and access by index is helpful when using numeric loops to access each column. To access all the data in the first column, you just leave the row element empty: `hmdb_df[,1]`. Leaving the row or column value blank returns all the elements, so this statement returns all the rows from column 1.

In Figure 1.2, when single integers are used for rows or columns, what is returned is a vector of the column's data type with no name.

```
  head(hmdb_df[,1])
```

```
## [1] "HMDB0000001" "HMDB0000002" "HMDB0000005"
"HMDB0000008" "HMDB0000010"
## [6] "HMDB0000011"
```

```
  class(hmdb_df[,1])
```

```
## [1] "character"
```

The extract operator `$` also returns a vector for the name given:

```
  head(hmdb_df$accession)
```

```
## [1] "HMDB0000001" "HMDB0000002" "HMDB0000005"
"HMDB0000008" "HMDB0000010"
## [6] "HMDB0000011"
```

The sequence operator : can also be used in the row and column position to specify a range of rows or columns to be returned. One important thing to notice is that when using the : operator, the class returned is a `data.frame` rather than a vector.

```
  head(hmdb_df[1:4,])
```

```
##     accession                  name   formula exact_mw
## 1 HMDB0000001     1-Methylhistidine C7H11N3O2 169.0851
## 2 HMDB0000002    1,3-Diaminopropane   C3H10N2  74.0844
## 3 HMDB0000005    2-Ketobutyric acid    C4H6O3 102.0317
## 4 HMDB0000008 2-Hydroxybutyric acid    C4H8O3 104.0473
##                          smiles     status
## 1 CN1C=NC(C[C@H](N)C(O)=O)=C1 quantified
## 2                         NCCCN quantified
## 3                CCC(=O)C(O)=O quantified
## 4            CC[C@H](O)C(O)=O quantified
```

```
  class(hmdb_df[1:4,])
```

```
## [1] "data.frame"
```

Another way to return a subset of a `data.frame` as a `data.frame` is to combine the `[]` operator with a string name of the column:

```
  head(hmdb_df["exact_mw"])
```

```
##   exact_mw
```

```
## 1 169.0851
## 2  74.0844
## 3 102.0317
## 4 104.0473
## 5 300.1725
## 6 104.0473
```

```
  class(hmdb_df["exact_mw"])
```

```
## [1] "data.frame"
```

One of the most powerful aspects of the `[]` operator in R is that a boolean vector can be used in place of the sequence generated by the `:` operator. Using boolean vectors instead of numeric sequences allows subsetting based on conditional statements:

```
  hmdb_df[hmdb_df["formula"] == "C4H8O3",]
```

```
##       accession                          name formula
exact_mw            smiles
## 4   HMDB0000008        2-Hydroxybutyric acid  C4H8O3
104.0473  CC[C@H](O)C(O)=O
## 6   HMDB0000011        3-Hydroxybutyric acid  C4H8O3
104.0473 C[C@@H](O)CC(O)=O
## 14  HMDB0000023 (S)-3-Hydroxyisobutyric acid  C4H8O3
104.0473 C[C@@H](CO)C(O)=O
## 188 HMDB0000336 (R)-3-Hydroxyisobutyric acid  C4H8O3
104.0473  C[C@H](CO)C(O)=O
## 228 HMDB0000442    (S)-3-Hydroxybutyric acid  C4H8O3
104.0473  C[C@H](O)CC(O)=O
## 358 HMDB0000710        4-Hydroxybutyric acid  C4H8O3
104.0473        OCCCC(O)=O
## 367 HMDB0000729 alpha-Hydroxyisobutyric acid  C4H8O3
104.0473    CC(C)(O)C(O)=O
##         status
## 4   quantified
## 6   quantified
## 14  quantified
```

```
## 188 quantified
## 228 quantified
## 358 quantified
## 367 quantified
```

Notice that the row component of the `[row,column]` statement is a boolean vector with a value of `TRUE` for every row in which the `formula` column value is equal to the string "C4H8O3." Again by leaving the row specification empty, the statement returns all the matching rows. Since the row specification was not a single value but a sequence, a new `data.frame` is returned. Using conditional statements to subset a `data.frame` is very powerful, but complex filtering can require convoluted conditional statements, which are hard to debug. The next section introduces a more modern method for subsetting based on filtering.

There is much more to subsetting than the basics I've shown here. You can find details in the "Subsetting" chapter in Wickham's excellent text: *Advanced R* [29].

1.2.5 The Tidyverse

A relatively new approach to managing data in R introduced a notable advance in R programming called "Tidy Data." Programs that follow tidy data principles manage and access data robustly, similar to modern database techniques.

The tidyverse is a collection of R packages intended to make it easier to write readable R code and support reproducible research. It was released in 2016 and is described in *R For Data Science* [30] and is available free at https://r4ds.had.co.nz/. The tidyverse team continually updates the package, and the project enjoys broad support from the R community.

To use the tidyverse, simply load it using the `library()` function:

```
library(tidyverse)
```

Loading the `tidyverse` metapackage adds the packages `ggplot2`, `tibble`, `tidyr`, `readr`, `purrr`, `dplyr`, `stringr`, and `forcats`.

Notice that `dplyr` masks two functions from the `stats` package: `filter()` and `lag()`. The `filter()` function in `dplyr` is central to how the tidyverse performs data subsetting. The `lag()` function in `dplyr` creates a new vector offset from a given vector by one element. When functions with the same name are loaded from different packages, you can specify the specific function by naming the package and using the `::` operator. The `filter()` function is probably one of the most often used function names when using R packages useful for mass spectrometry, because the verb *filter* has many different meanings in this domain. In some cases, you want to filter rows in a table. Other meanings include filtering a range of m/z values or retention times. It can also mean applying a signal processing filter to a dataset. It is often necessary to access the specific version of `filter()` by putting the package name first, like `stats::filter()`. As more libraries are used, it is more important to specify the package name.

1.2.6 Tidy Data

The main idea behind tidy data is that in a table, each column represents a variable, and each row represents an observation. You can create objects in which every element is of a different type, but that data is difficult to deal with. In the tidyverse, if you need to represent observations with a variable that is duplicated across many observations, you could add a new variable (column), but it might be a sign that you should create a new table. Rather than duplicating observations two tables can be constructed and then related

by a unique, shared variable. In database management, this approach is called *normalization*, which aims to remove duplicate entries by using multiple tables. There are situations where you don't want highly normalized data in multiple tables. For example, systems like data warehouses or data lakes use a single table with a high degree of duplication to simplify filtering, grouping, and aggregation by avoiding joining multiple tables. The tidyverse team has explicitly declared that it is an opinionated project with clearly stated ideas about what constitutes *correct* data management. The functions in the `dplyr` package, which provides the tools to manage data, are designed with normalized, tidy data in mind. However, many application-specific packages in R do not follow the tidy data principles. In this book, I will focus on *tidying* the data returned by many application-specific functions, so your data is easier to understand and manipulate. A focus on tidy data will also have the effect of making your data more compatible with the other projects, like Tidymodels, which has grown up around the tidyverse ecosystem.

1.2.7 The `tibble`: An Improved `data.frame`

The `data.frame` is an incredibly useful data structure and is one reason data analysis in R is superior to using spreadsheets. However, when analyzing complex data, the data access and manipulation syntax can sometimes become hard to read. One of the goals of the tidyverse approach is to use functions to perform data organization and manipulation at a higher level, rather than having to resort to low-level base R syntax. Central to the tidyverse metapackage is the `dplyr` (pronounced de-plyer) that implements tidy manipulation functions. The `dplyr` package relies on a modernized version of the `data.frame` class called `tibble` from the `tibble` package. A `tibble` is a subclass of the `data.frame` class, which means `tibbles` are `data.frames`, and anything

that works with a `data.frame` also works with a `tibble`. However, a `tibble` is a simplified version of the `data.frame` class, which for example, doesn't use row names, among other internal data representation changes. The `tibble` class *overloads* many of the `data.frame` functions. The result is that the default behavior of a `tibble` is, in some cases, quite different from the `data.frame` parent class.

Data can be read directly into a `tibble` from a file, or you can create a `tibble` from an existing `data.frame`.

```
hmdb <- as_tibble(hmdb_df)
print(hmdb)
```

```
## # A tibble: 4,692 x 6
##    accession   name                  formula   exact_mw
smiles            status
##    <chr>       <chr>                 <chr>        <dbl>
<chr>             <chr>
##  1 HMDB0000001 1-Methylhistidine     C7H11N3O2    169.
CN1C=NC(C[C@H](N~ quant~
##  2 HMDB0000002 1,3-Diaminopropane    C3H10N2       74.1
NCCCN             quant~
##  3 HMDB0000005 2-Ketobutyric acid    C4H6O3       102.
CCC(=O)C(O)=O     quant~
##  4 HMDB0000008 2-Hydroxybutyric acid C4H8O3       104.
CC[C@H](O)C(O)=O  quant~
##  5 HMDB0000010 2-Methoxyestrone      C19H24O3     300.
[H][C@@]12CCC(=O~ quant~
##  6 HMDB0000011 3-Hydroxybutyric acid C4H8O3       104.
C[C@@H](O)CC(O)=O quant~
##  7 HMDB0000012 Deoxyuridine          C9H12N2O5    228.
OC[C@H]1O[C@H](C~ quant~
##  8 HMDB0000014 Deoxycytidine         C9H13N3O4    227.
NC1=NC(=O)N(C=C1~ quant~
##  9 HMDB0000015 Cortexolone           C21H30O4     346.
[H][C@@]12CC[C@]~ quant~
## 10 HMDB0000017 4-Pyridoxic acid      C8H9NO4      183.
CC1=NC=C(CO)C(C(~ quant~
## # i 4,682 more rows
```

The first observable difference between a `tibble` and a `data.frame` is the `print()`. In addition to limiting the default output to 10 rows, `print()` gives extra data about the shape, and column types. In a tibble, variables (columns) still have names, but rows (observations) do not.

The `dplyr` package provides all the subsetting and manipulation functions needed to work with a `tibble`. Idiomatic tidyverse programming using `tibble` generally avoids the `[]` selection operator but like many idioms, this convention is often ignored and many programs move between tibbles and data.frames without strict adherence to tidyverse conventions. This blending of styles allows flexibility, as tibble objects are compatible with base R data frame operations, but it can sometimes lead to confusion or unexpected behavior when functions treat tibble objects differently from traditional data frames.

To perform the selection of the first column as a vector the `pull()` function is used with the column number:

```
  head(pull(hmdb, 1))
```

```
## [1] "HMDB0000001" "HMDB0000002" "HMDB0000005"
"HMDB0000008" "HMDB0000010"
## [6] "HMDB0000011"
```

The same output can be obtained using the variable name:

```
  head(pull(hmdb, accession))
```

To extract the first column as a tibble, the `dplyr::select()` function is used:

```
  dplyr::select(hmdb, exact_mw)
```

```
## # A tibble: 4,692 x 1
```

```
##    exact_mw
##       <dbl>
##  1    169.
##  2     74.1
##  3    102.
##  4    104.
##  5    300.
##  6    104.
##  7    228.
##  8    227.
##  9    346.
## 10    183.
## # i 4,682 more rows
```

The `dplyr` package tries to make what actions and return values clear by using functions with descriptive names rather than depending on syntax.

A good example of how the tidyverse approach makes code more understandable is the `dplyr::filter()` function, which is much easier to read than the Base R example given above.

```
  dplyr::filter(hmdb, formula == "C4H8O3")
```

```
## # A tibble: 7 x 6
##   accession   name                           formula
exact_mw smiles        status
##   <chr>       <chr>                          <chr>
<dbl> <chr>         <chr>
## 1 HMDB0000008 2-Hydroxybutyric acid          C4H8O3
104. CC[C@H](O)C(~ quant~
## 2 HMDB0000011 3-Hydroxybutyric acid          C4H8O3
104. C[C@@H](O)CC~ quant~
## 3 HMDB0000023 (S)-3-Hydroxyisobutyric acid C4H8O3
104. C[C@@H](CO)C~ quant~
## 4 HMDB0000336 (R)-3-Hydroxyisobutyric acid C4H8O3
104. C[C@H](CO)C(~ quant~
## 5 HMDB0000442 (S)-3-Hydroxybutyric acid    C4H8O3
104. C[C@H](O)CC(~ quant~
## 6 HMDB0000710 4-Hydroxybutyric acid          C4H8O3
104. OCCCC(O)=O    quant~
```

```
## 7 HMDB0000729 alpha-Hydroxyisobutyric acid C4H8O3
104. CC(C)(O)C(O)~ quant~
```

You can find many more details on the `dplyr` package in *R For Data Science* [30, 31].

The `as.tibble()` function is very helpful when moving data from application-specific functions that primarily use base R types. When reading data from files the functions in the `readr` package from the `tidyverse` return a tibble. For example, using the `read_csv()` function:

```
hmdb <-
read_csv(file.path("data","hmdb_urine_metabolites.csv")
)
```

```
## Rows: 4692 Columns: 6
## -- Column specification ------------------------------
-------------------------------------
## Delimiter: ","
## chr (5): accession, name, formula, smiles, status
## dbl (1): exact_mw
##
## i Use `spec()` to retrieve the full column
specification for this data.
## i Specify the column types or set `show_col_types =
FALSE` to quiet this message.
```

The `read_csv()` function shows that there were 4692 rows and six columns. It also shows the delimiter symbol and the *guessed* column specification so you can check to ensure the function reads data the way you expected. If `read_csv()` guessed the wrong column type, you must provide the actual specifications using the `col_types` argument. Both `read.csv()` and `read_csv()` import all string (character) columns as just strings by default. In an earlier version of R, the default import type for string columns was *factor*, meaning a categorical variable where the string value was considered a *level*. Over the years, the `stringsAsFactors=TRUE` default

caused so much confusion that the core R team changed it. As of version 4.0, the default for `read.csv()` is `stringsAsFactors=FALSE` so that strings are not assumed to be factors by default.

In this example, `status` can be treated as a factor with three levels found using `unique()`:

```
unique(hmdb$status)
```

```
## [1] "quantified" "detected"   "expected"
```

Now the file can be read with the desired column specification and the `spec()` function can be used to make sure all the columns have the correct type, including cases where strings are treated like factors:

```
hmdb <-
read_csv(file.path("data","hmdb_urine_metabolites.csv")
,
            col_types = cols(
                accession = readr::col_character(),
                name      = readr::col_character(),
                formula   = readr::col_character(),
                exact_mw  = readr::col_double(),
                smiles    = readr::col_character(),
                status    = readr::col_factor(levels =
c("quantified",

"detected",

"expected")
                                          ),
            )
)

readr::spec(hmdb)
```

```
## cols(
```

```
##   accession = col_character(),
##   name = col_character(),
##   formula = col_character(),
##   exact_mw = col_double(),
##   smiles = col_character(),
##   status = col_factor(levels = c("quantified",
"detected",
     "expected"), ordered = FALSE, include_na = FALSE)
## )
```

1.2.8 Functional Programming with Pipes

Notice how all of the functions in `dplyr` take the data object as the first argument. The idea is to use this convention to allow for method chaining. Many data analysis processes can be stated clearly as a chain of transformations: selection, subsetting, manipulation, etc. Functional programming idioms represent chained operations naturally. The functional programming approach makes expressing chained operations easier. The `tidyverse` brought functional programming for chained operations to R. The `tidyverse` uses *pipes* to link sequential operations to make programs easier to read and write. The `tidyverse` first introduced the pipe using the symbol `%>%` implemented by the `magrittr` package. Functional programming is so useful in data science and `%>%` chains became such a common practice, that a native pipe operator `|>` was added to the core R language in version 4.1. It's highly recommended that you use the native pipe in your programs. You will still see many programs and examples that use the `%>%` symbol. The two operators work differently, so care must be taken when substituting the native pipe for the `magrittr` pipe. I will exclusively use the native `|>` pipe in this book.

The chained method approach to using the `pull()` function in combination with the `head()` function makes it easier to see the sequential nature of the operations, and eliminates the need for :

```
hmdb |>
    pull(exact_mw) |>
    head()
```

```
## [1] 169.0851  74.0844 102.0317 104.0473 300.1725
104.0473
```

Notice how each function (`pull()` and `head()`) is run with the left-hand side of the `|>` operator's return value as the first argument.

Before performing any numerical calculations on `exact_mw`, it's usually a good idea to check to see if there are missing values and decide what to do about them. Again the pipe approach could be used here:

```
hmdb |>
    summarise(count = sum(is.na(exact_mw)))
```

```
## # A tibble: 1 x 1
##   count
##   <int>
## 1     1
```

One of the rows in the table has an `exact_mw` which is missing. This will cause problems with calculating summary statistics, so first, let's see which one it is:

```
hmdb |>
    dplyr::filter(is.na(exact_mw))
```

```
## # A tibble: 1 x 6
##   accession   name    formula           exact_mw
smiles                  status
##   <chr>       <chr>   <chr>                <dbl> <chr>
<fct>
```

```
## 1 HMDB0001394 Heparin (C12H19NO19S3)nH2O       NA
[H]O[C@H]1O[C@H](COS(O~ quant~
```

Since the chemical formula for Heparin allows for “n” number of water molecules to be attached, the exact molecular weight cannot be computed, so it is missing. This row can be filtered out before computing statistics on the table:

```
  hmdb |>
      dplyr::filter(!is.na(exact_mw)) |>
      summarise(mean=mean(exact_mw))
```

```
## # A tibble: 1 x 1
##    mean
##   <dbl>
## 1  389.
```

The `dplyr` functions, like `summarise()`, return a `tibble`. Like Base R, the `tidyverse` has a set of printing options for `tibble`. The default is to print three significant digits. That is not ideal for a variable like `exact_mw`. There are multiple ways to control the default options, starting with changing the global options for all printing. There is a help page that can be found by typing the search command: `??tibble::numbers` into the R console. The help page for `tibble::numbers` describes how to change the display options for numbers. For a specific block of code, it is easy to convert the output of the `summarise()` to a double-precision numeric type.

```
  hmdb |>
      dplyr::filter(!is.na(exact_mw)) |>
      summarise(mean=mean(exact_mw)) |>
      as.double()
```

```
## [1] 388.9683
```

Another example of a chained operation is to collect the unique values of a variable and then use those for summarization:

```
hmdb |>
    distinct(formula) |>
    summarise(count=n())
```

```
## # A tibble: 1 x 1
##   count
##   <int>
## 1  2973
```

This indicates that of the 4692 molecules in this dataset, there are only 2973 distinct chemical formulas.

```
hmdb |>
    distinct(formula, .keep_all = TRUE) |>
    dplyr::filter(!is.na(exact_mw)) |>
    summarise(mean=mean(exact_mw)) |>
    as.double()
```

```
## [1] 374.8294
```

In the example where we just counted the distinct rows, the fact that `distinct()` drops all the other variables by default was helpful since the resulting tibble only had the variable to be counted. In this example, the option `.keep_all` was needed to produce a table that kept all the columns that had distinct values in the `formula` column.

1.2.9 Plotting with Base R and `ggplot2`

Plotting data is an essential part of statistical analysis. So, naturally, R was built with a powerful and easy-to-use plotting system. Like the flexibility of the original `data.frame`,

the ability to plot data easily is another reason people are drawn to R as a data analysis language.

Before I start plotting, I want to address an issue with colors in scientific visualization. Often, colors like red, green, and blue are used for annotation and overlaying lines. These are fine color choices for readers who can read a color version of the plot and don't have any problems with color distinction in their vision. Unfortunately, there are many people affected by color distinction issues. Using R it is easy to ensure that your plots can be more readable for either base R or using `ggplot2`. As of R 4.0.0, a new color palette was adopted to make the default colors more accessible. The new palette follows the recommendations of Okabe and Ito [32] and can seen with the `palette.colors()` function.

```
  palette.colors()
```

```
## [1] "#000000" "#E69F00" "#56B4E9" "#009E73" "#F0E442"
"#0072B2" "#D55E00"
## [8] "#CC79A7" "#999999"
```

To make these colors easier to use when specifying colors, I put the color codes and shortened names into a data.frame and used them via my own names:

```
  pal<-as_tibble(t(palette.colors()), .name_repair =
  "universal")
  colnames(pal) <- c("black", "orange",  "lightblue",
  "green",  "yellow", "blue",
                     "darkorange",  "red",  "gray")
  pal$yellow <- "#FDDA0D"
```

I kept the default colors with the exception of yellow, which I changed to a slightly darker shade since the default can be difficult for anyone to see on a white background.

A time-of-flight MS spectrum for serine collected in profile mode is stored in the file `serine-0649.csv` and I'll show how to plot this spectrum using Base R and `ggplot2`.

```
s <- read_csv(file.path("data","serine-0649.csv"),
show_col_types=FALSE)

serine_mw = 106.0498696

plot(s$mz, s$inten,
     type="b",
     xlim=c(106.0,106.1),
     ylim=c(0,60000),
     xlab="m/z",
     ylab="intensity",
     main="Serine Profile TOF Spectrum"
)

# Monoisotopic mass of C3H7NO3 [M+H]+ is 106.0498696
# add a vertical line to the plot at [M+H]+

abline(v=serine_mw, col=pal$red, lty=2, lwd=2)
```

The base plotting functions in R are based on a pencil and paper model. In his groundbreaking book *The Grammar of Graphics* [33], Wilkinson compares this to the *chart* metaphor. A function is called to make a particular kind of chart, and then editing functions are called to add or change things on the chart to better convey your intent. The call to the `abline()` function in Figure 1.3 is an example. The chart metaphor, however, limits the user by providing a limited set of chart types with an associated collection of editing functions. This makes it easy for the package designer but forces the data analyst to find the specific charts and edits to get the plot to tell the desired story. The idea of a grammar of graphics is to provide the analyst with a set of components (words) and rules (grammar) to assemble complex graphical objects much in the way languages are composed by creating

a set of reusable objects that can be combined in a variety of ways, more sophisticated and more meaningful graphics can be constructed in an iterative way.

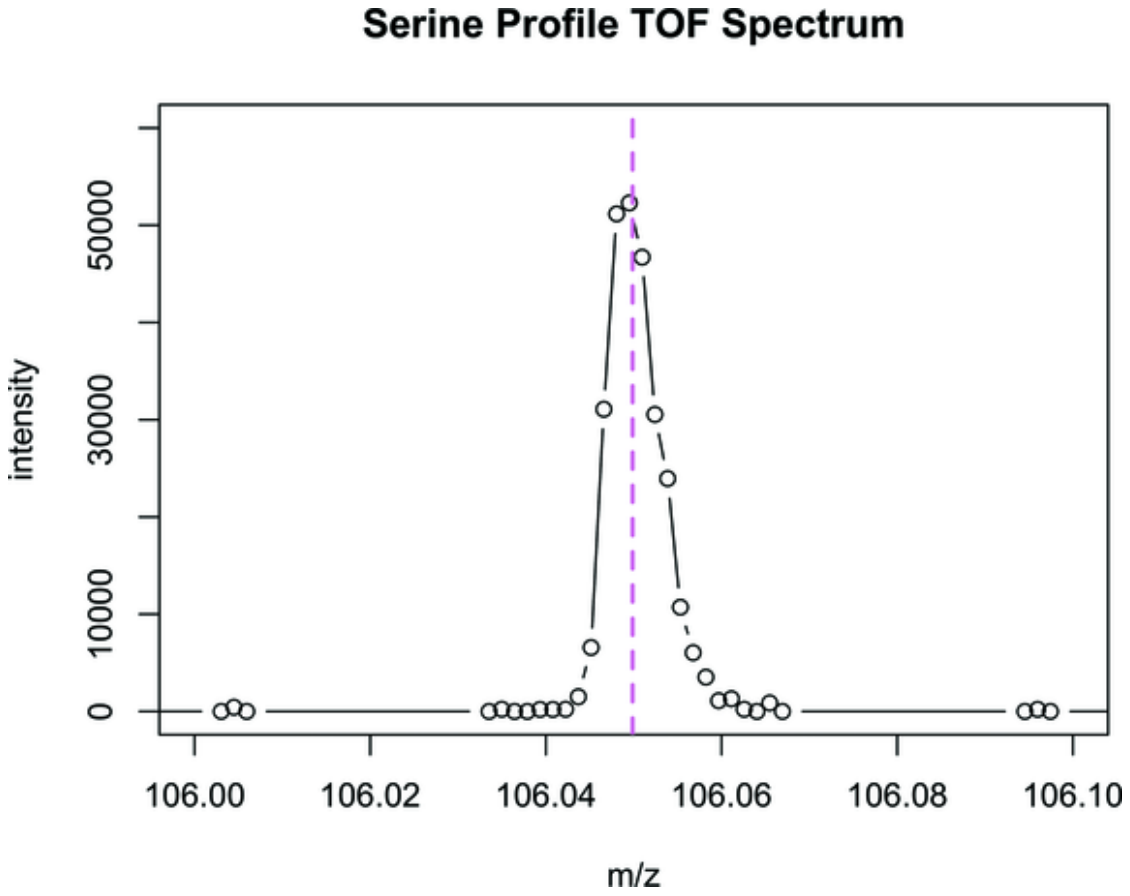


Figure 1.3 Using base R plot() to show a time-of-flight spectrum of serine ($C_3H_7NO_3$) collected in profile mode. A dashed line shows the monoisotopic mass of $[M+H]^+$.

The `ggplot2` package, described in Wickham's book *ggplot2* [34], extends the use of reusable graphical objects by adding the idea of *layers*. In `ggplot2`, layers are used to describe how to render each observation in the data selected for the graph.

```
p <- ggplot(s, aes(mz, inten)) +
        geom_line() +
        geom_point() +
        geom_vline(xintercept = serine_mw,
                   color=pal$red, linetype="dashed",
linewidth=1) +
        xlab("m/z") +
        ylab("intensity") +
        xlim(106.0,106.1) +
        ylim(0,60000) +
        ggtitle("Serine Profile TOF Spectrum")
print(p)
```

In `ggplot2`, the idea is to use the object-oriented nature of R to construct a plot object using the `+` operator to combine layers of graphical elements. The `ggplot()` function takes the full data set, which must be a type of `data.frame` (remember `tibbles` are a subclass of `data.frame`) as the first argument, and then a mapping of data to visual properties (called aesthetics) using the `aes()` function. In the ggplot example in Figure 1.4, the tibble `s` is passed into `ggplot()` and the variables `mz` and `intensity` are selected to be plotted by the `aes()` function. Once the `ggplot` object has been created, the first type of layer added is a line plot. Line plots are a type of *geom* (geometry) layer. There are many types of geoms built into ggplot2. In the example, plotting the data points over the `geom_line()` is done by adding a `geom_points()` layer to the plot object. Another layer added is the vertical line `geom_vline()`. You can add as many geom layers as you need.

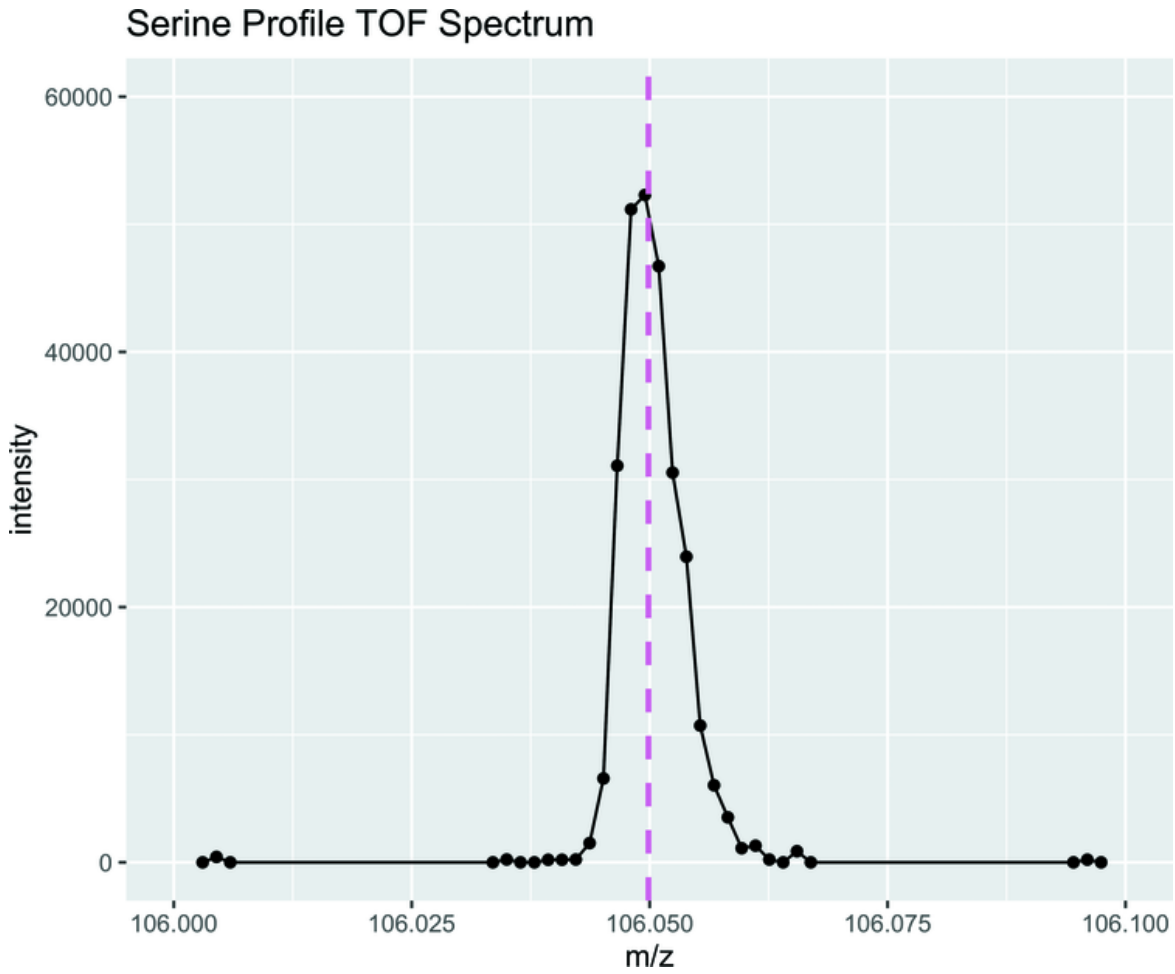


Figure 1.4 Using ggplot2 to show a time-of-flight spectrum of serine.

Next, in the example, the axes are customized with layers that override the defaults generated by the original `ggplot()` function. The `xlab()`, `ylab()`, `xlim()`, and `ylim()` are all used to specify the details for the axes. Notice that in this code, you could change any of the layers to change the plot without changing how the axes were specified. Finally, a `ggtitle` layer is added. The plot in Figure 1.4 follows the default theme for `ggplot2`. The last layer that we will add is a `theme` layer which changes the look of the plot to give Figure 1.5.

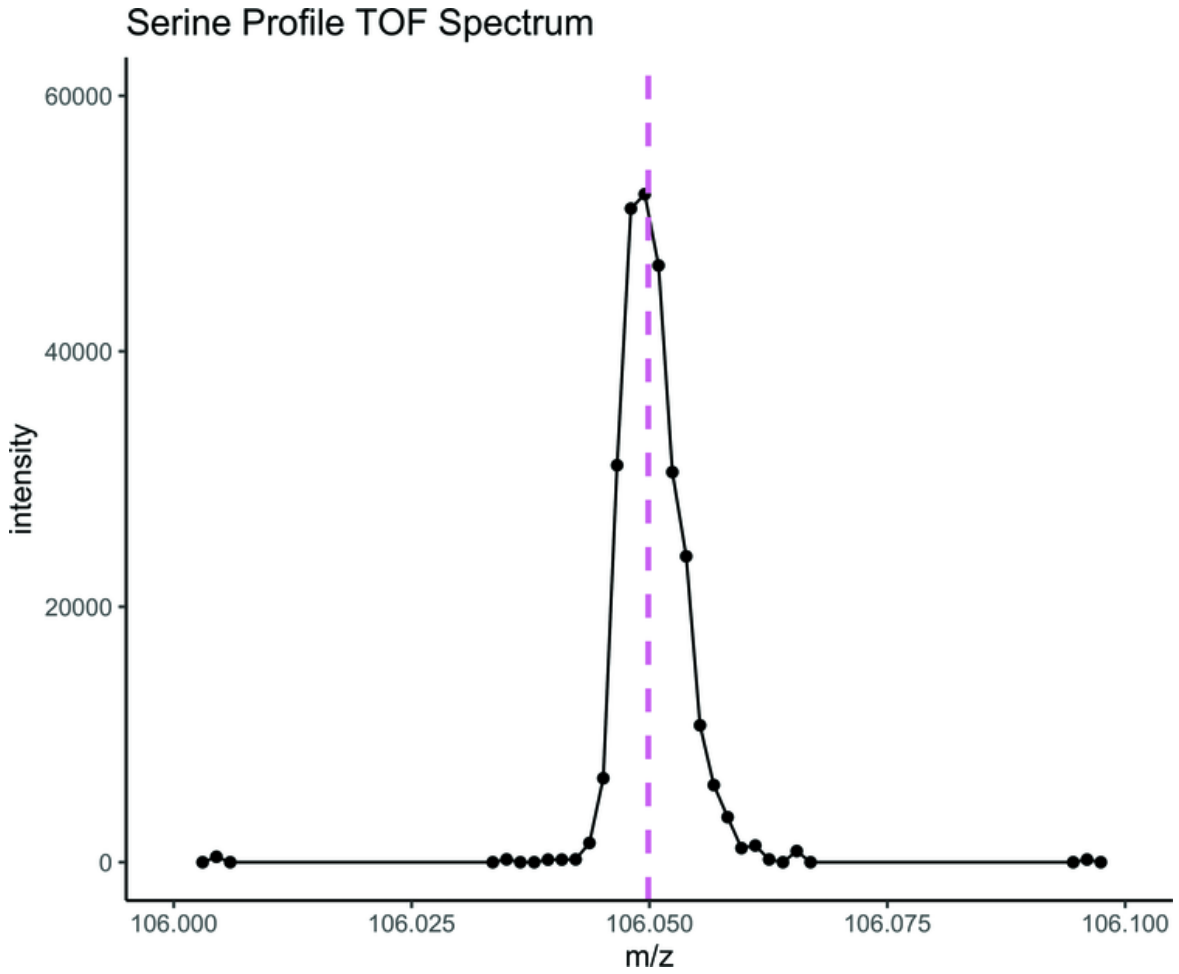


Figure 1.5 Using ggplot2 with classic theme to show a time-of-flight spectrum of serine.

```
p <- p + theme_classic()
print(p)
```

Throughout this book, you will find more examples of how to generate publication-quality graphics from mass spectrometry and related data. In addition, there are many practical resources for base R graphics, like Chang's *R Graphics Cookbook* [35] and *Data Visualization* by Healy [36], which uses `ggplot2`.

1.3 Bioconductor

Bioconductor is a package repository supported by a very active community since 2001 [37].

> *The mission of the Bioconductor project is to develop, support, and disseminate free open-source software that facilitates rigorous and reproducible analysis of data from current and emerging biological assays. We are dedicated to building a diverse, collaborative, and welcoming community of developers and data scientists.*

Bioconductor is a tightly integrated collection of packages with a narrower audience than CRAN. Like CRAN, it is a way for the community to share packages but has a stricter review process and uses a continuous integration and deployment approach. The Bioconductor project builds and tests over 2100+ packages continuously [5]. Mass spectrometry is one of many technologies used in biological research, so there are a significant number of packages available to work with mass spec data. There are more than 100 mass spectrometry-related packages in Bioconductor. Many are helpful for any mass spectrometry experiment, while others incorporate mass spectrometry data into workflows with other kinds of -*omics* experiments.

1.3.1 Essential Packages

Because Bioconductor follows a different release schedule than R and CRAN, the team has created an alternative installation process mentioned in the Preface. Bioconductor uses the `BiocManager` package `install()` function to install packages. Bioconductor uses a core set of classes automatically installed when any of the Bioconductor packages is needed. While not required, Bioconductor encourages developers to use R's S4 object paradigm in R [38]. The object-oriented system package uses usually won't

affect how you use the functions and data in Bioconductor packages. Still, S4 is a more formal system of classes and methods, and examples will generally use *accessor* functions rather than extracting data directly from objects. Using S4 means that sometimes data will not be easy to browse in R Studio. However, this book shows how to use Bioconductor and other application-specific systems by moving results into a tidy format to use the `tidyverse` and all of its packages more easily.

1.3.2 Mass Spectrometry

The Bioconductor project contains over 100 packages dedicated to analyzing mass spectrometry data using R. The approach I am recommending in this book is to use Bioconductor packages to read and process mass spectrometry data and then move from the application-specific data structures to the tidyverse and standard R programming. In some cases, you will be able to find entire workflows in Bioconductor that can efficiently complete your analysis, especially if proteomics or other biological applications are your primary use of mass spectrometry data. While many mass spectrometry applications are not directly related to biological research, there are valuable components found in Bioconductor that can save time and effort. A prime example is using the Bioconductor packages for reading various mass spectrometry data formats. To read raw data, packages such as `mzR` or `MSnbase` will cover all but the most specialized applications. Depending on your needs, other mass spectrometry packages may also be helpful. Since the Bioconductor project is so well maintained, it will always be a good idea to check the package descriptions to see if you can use one or more directly. The Bioconductor project is so large that it provides a package browser called *BiocViews* from the main project page (https://www.bioconductor.org/packages/release/BiocViews.

html) [39]. In addition to software packages for mass spectrometry, some *data packages* contain data you can use to test analysis approaches or new applications. Throughout this book, I will show the use of both the software and data packages in Bioconductor.

1.4 Reproducible Data Analysis

The objective of reproducible data analysis is to allow others to see and repeat the *computational analysis* used to arrive at a finding [40]. Minimally, this means providing access to data sets and code. The goal is to increase the reliability of published results, either within your organization or in the literature. For the larger community, reproducibility often starts with ensuring data is accessible and others can reproduce the analysis method. In descriptions of methods translated into natural language, it is too easy to leave out some details needed to make an analysis work. Ultimately button clicks inside user interface programs are not easily reproduced by others, even if they have your data and access to the program. Reports rarely describe a user interface gesture such as dragging, dropping, and clicking, making some analysis steps easy to miss. Scripts have statements for every step in the analysis, so no action in a computation is left undescribed. This completeness has led to a preference for using programs to perform *reproducible research*.

Further, anyone can run the program and reanalyze the data when the analyst writes the program using an open-source system. Programs written in closed-source software analysis tools create a barrier for anyone who does not have access to the software. The open nature of the R system has made it a good choice for improving the reproducibility of analysis.

R is constantly changing. For someone to reproduce your past results, they will have to understand the environment you used. R has a function called `sessionInfo()`, which

displays the R version and all the packages present during the analysis. Providing the session information allows everyone to see what versions of packages are needed and which ones might be masking functions from base R or other packages to understand the system behavior. If you used the latest package for a particular analysis, by the time someone needs to reanalyze your data, it is possible that the package was updated in the meantime. Even if the dependencies cannot be exactly reconstructed, knowing that you used an older package will help with future analysis.

The output of `sessionInfo()` looks like this:

```
  sessionInfo()
```

```
## R version 4.4.1 (2024-06-14)
## Platform: aarch64-apple-darwin20
## Running under: macOS Sonoma 14.6.1
##
## Matrix products: default
## BLAS:   /Library/Frameworks/R.framework/Versions/4.4-
arm64/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/4.4-
arm64/Resources/lib/libRlapack.dylib;
   LAPACK version 3.12.0
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-
8/en_US.UTF-8
##
## time zone: America/Indiana/Indianapolis
## tzcode source: internal
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets
methods   base
##
## other attached packages:
##  [1] lubridate_1.9.3     forcats_1.0.0
stringr_1.5.1
```

```
##  [4] dplyr_1.1.4         purrr_1.0.2
readr_2.1.5
##  [7] tidyr_1.3.1         tibble_3.2.1
ggplot2_3.5.1
## [10] tidyverse_2.0.0     knitr_1.48
BiocParallel_1.38.0
##
## loaded via a namespace (and not attached):
##  [1] utf8_1.2.4        generics_0.1.3    stringi_1.8.4
hms_1.1.3
##  [5] digest_0.6.36     magrittr_2.0.3
evaluate_0.24.0   grid_4.4.1
##  [9] timechange_0.3.0  bookdown_0.40     fastmap_1.2.0
tinytex_0.51
## [13] fansi_1.0.6       scales_1.3.0      codetools_0.2-
20  cli_3.6.3
## [17] rlang_1.1.4       crayon_1.5.3      bit64_4.0.5
munsell_0.5.1
## [21] withr_3.0.0       yaml_2.3.9        tools_4.4.1
parallel_4.4.1
## [25] tzdb_0.4.0        colorspace_2.1-0  vctrs_0.6.5
R6_2.5.1
## [29] lifecycle_1.0.4   bit_4.0.5         vroom_1.6.5
pkgconfig_2.0.3
## [33] pillar_1.9.0      gtable_0.3.5      glue_1.7.0
xfun_0.45
## [37] tidyselect_1.2.1  rstudioapi_0.16.0 farver_2.1.2
htmltools_0.5.8.1
## [41] rmarkdown_2.27    labeling_0.4.3    compiler_4.4.1
```

I have not used many packages at this point, so the list is somewhat shorter than it will be at the end of a typical analysis project. It is best practice to run `sessionInfo()` at the end of your work to capture everything used.

1.4.1 Project and File Organization

Another issue that can occur in reproducing analysis is hardware and operating specifics. In various operating systems, directory names and file paths have different

requirements. When working in R, setting the working directory to the location of your code allows you to use system-independent path construction using the `file.path()` function. Using `file.path()` creates a path specific to your operating system.

Another important consideration is the organization of project files and data. Today the primary environment is a local computer or server running an operating system that uses files and directory structures. There is no consensus on a specific directory structure for project management, but a central guiding principle is to keep your work organized. Roughly, it should be clear from looking at the contents of a directory where everything is. Having one directory for data and another for code is a good start.

As computing platforms evolve, the methods for providing access to data and programming environments will change drastically. With increasing demands for more storage and higher-performance computing with more complex analysis pipelines, virtual machines (VM) are becoming as important as the “desktop computer” environment. There are advantages to using a VM in a cloud computing environment, especially for large-scale projects and collaborations. First, the construction and configuration of high-complexity analysis pipelines is a separate skill from data analysis. Building a high-performance VM system is more akin to system administration than data science. Second, using a virtual machine allows complete pipelines to be assembled and made available to anyone with access to a computing environment that can run the VM. The use of cloud computing and the VM approach is outside the scope of this book, but it is an active area that includes the R ecosystem. See the *AnVIL* project [41] for an example.

1.4.2 RMarkdown and Literate Programming

Most research reports are static documents such as slide presentations, posters, journal manuscripts, and technical reports. One way to make research findings easier to reproduce is to create a dynamic document that incorporates the analysis, graphics, and text in a single document. Donald Knuth coined the term *literate programming* [22] to describe writing a natural language document that includes executable chunks of code that can be *woven* into a complete and understandable final document. Initially described in the 1980s, the *notebook* supported literate programming with the concept of *cells* which could contain natural language, code, or plots. Notebook environments are now a popular tool in data science. R supports literate programming primarily through the `knitr` package. R uses `knitr` to integrate code with text-based writing systems such as LaTex, HTML, Markdown, and others [42].

One of the tenets of reproducible research is that documents should be created and maintained in clear text formats. There are many methods for moving between text and presentation. HTML is simple text that can be written in a code editor and rendered by a browser or other program into a highly polished document. LaTex is another way to generate high-quality technical documents. LaTex documents are plain text files that follow a specific syntax that a program can typeset and render automatically. Another approach to document creation is to use a file format called Markdown [43]. Markdown is more limited than other document creation formats but, in exchange, it is much easier to write. Like HTML and LaTex, Markdown is a program that can convert the text into a high-quality final document. There are many references for how to use HTML and LaTex to create documents, and using these directly for reproducible research is outside the scope of this book. The approach recommended here is to use Markdown as the primary tool

for expressing the format of a document. Using `knitr`, however, allows RMarkdown documents to include snippets of LaTex or HTML, depending on the target for the final document. Also, you can add bits of LaTex or HTML to the RMarkdown text to achieve a specific presentation effect. Thanks to the power of `knitr`, all of these can be mixed and matched in a single RMarkdown document.

Some simple examples of the Markdown syntax used in RMarkdown:

```
# Header 1

## Header 2

*italics*

**bold**

inline equation: $C = 2*\pi*r$
```

Which renders as:

Header 1

Header 2

italics

bold

inline equation: $C = 2\pi r$

1.5 Summary

In this chapter I've given you an introduction to the ideas behind using the tidyverse as the main way to organize data

from specialized packages used for analyzing mass spectrometry data. In the next chapter, I will walk through a more detailed example using more complex mass spectrometry data to show how to use the tidyverse and Bioconductor together.

Chapter 2
Introduction to Mass Spectrometry Data Analysis

This chapter will show an example of analyzing data from a mass spectrometer. In later chapters, I will discuss mass spectrometers in more detail and dive deeper into the many different mass spectrometer designs and the type of data they produce. The data analysis methods will follow the data's complexity level, starting relatively simple and moving to more advanced analysis methods as needed.

2.1 An Example of Mass Spectrometry Data Analysis

To get oriented using the tidyverse and Bioconductor, the rest of this chapter will show a simple example of loading, processing, and plotting a mass spectrometry data set from the `MassIVE` repository hosted by the Center for Computation Mass Spectrometry, at the University of California, San Diego [44].

First, download the data set MSV000081318 from: `ftp://massive.ucsd.edu/MSV000081318/`.

This dataset contains 78 files and is 4.37 GB in size. The data comes from a high-resolution liquid chromatography with tandem mass spectrometry (LC-MS/MS) analysis of a sea sponge, *Theonella swinhoei* [45].

2.1.1 Basic Mass Spectral Data

All eXtensible markup language (XML)-based mass spectrometry formats are text files that any text editor can open. Computers can store mass spectrometry data in many different file formats. Each format has a function and some strengths and weaknesses. I will describe additional file formats as I cover the applications where they are most applicable. In general, even though these files are human-readable, you won't need to examine them in their raw format. However, before you start working with these files, it is instructional to peek inside one and see what you are dealing with and why you want specialized tools for working with them. Below are the top few lines of the data file `2017_04_08_TSW_Sponge.mzXML`:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<mzXML
    xmlns="http://sashimi.sourceforge.net/schema_revision/mzXML_3.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://sashimi.sourceforge.net/schema_revision/mzXML_3.0
                        http://sashimi.sourceforge.net/schema_revision/mzXML_3.0
                            /mzXML_idx_3.0.xsd">
    <msRun scanCount="5703"  startTime="PT60.14S"  endTime="PT1320.5S">
        <parentFile
            fileName="file://9020-DDDXS22/C/Xcalibur/data/eriche/
                            Theonellamides_Israel_3/2017_04_08_TSW_Sponge.raw"
            fileType="RAWData"
            fileSha1="4f6120724c2ca4a15e5eeddadea3404abfba4a2f"/>
        <msInstrument>
            <msManufacturer category="msManufacturer" value="Thermo Finnigan"/>
            <msModel category="msModel" value="unknown"/>
            <msIonisation category="msIonisation" value="ESI"/>
            <msMassAnalyzer category="msMassAnalyzer" value="FTMS"/>
            <msDetector category="msDetector" value="unknown"/>
            <software type="acquisition" name="Xcalibur"
                      version="2.5-204201/2.5.0.2042"/>
        </msInstrument>
        <dataProcessing centroided="1">
            <software type="conversion" name="ReAdW"
                      version="4.0.2(build Jul 1 2008 14:23:37)"/>
        </dataProcessing>
        <scan num="1"  msLevel="1" peaksCount="75" polarity="+" scanType="Full"
            filterLine="FTMS + p ESI Full ms [600.00-2000.00]"
            retentionTime="PT60.14S" lowMz="611.187" highMz="1853.32"
            basePeakMz="831.448" basePeakIntensity="6739.2"
            totIonCurrent="96513.3">
            <peaks precision="32" byteOrder="network" pairOrder="m/z-int">
                RBjL9kQvdwVEGQvURBjL6UQZnwFFKlq6RBnfOEQfr7FEGtweRajAA0Qa7uREEGFM
                RBr00EPlR9hEGxxhRICOS0QbMIBEEdhkRB7ad0TfJhlEI01BRBSRH0QjWkhEDasz
                RCSgmkUfxuhEJODlRFNid0Ql3atFHtGZRCYd+0R/kVJEKCZuRCEf10QojV5E9nQW
                RCkgnkQ6pSZEKn28RFEMrkQrTPREJAjgRCtra0QtsqVEK60nRBVGokQtL1dEG7gK
                RC1qqER1QWVEL6JSRSUUVEQv4ntESNSdRDAmjkQ1he9EMN9nRQUldEQxH5VE+CHY
                RDFjzUR8EUVEMaP6RB0trUQ03d9EJbNyRDpBhUQSjeVEOmm5RC7uU0Q6pCNEXIUW
                RDvhGkVVsABEPCExREfdUEQ8c9JENTH4RECBbUQdfk9ERuK3RFnjzERI2pNE9Ahq
                RExbnUUN+xNETfCLRWg5TERP3LJF0pmXRFAc9EUIUXtEUFsdRDJLQERQXWJE/ZmZ
                RFCdxERcCKBEU129RFXcM0RV5oxEebvHRFhTq0RvBT1EWHOpRExN1URYk5REY8kR
                RFizh0UaZThEWNOoRIpdR0RZX1BEWkWuRFqT3EQ8/EBEWssERQRt2URa091EX34D
                RFrz1URkXbtEW91vRCzkHkRdUwVFAT19RF2TVERLHf1EivUyRGAkGUSWzmxEXpo/
                RJkx2kQ8zzZEmTPQRFMqvUSdJ59EJwOjRKGi9ERKJN1EpfGlRDguZ0Sn6jVEUNF8
                RMLiQ0SAJKtE41IkRGWRPETnqiFETbIj
            </peaks>
        </scan>
```

XML elements are enclosed in <> brackets, and the file begins with the standard XML header giving the version and encoding. The next element shows that the file follows the mzXML format described by the mzXML schema `mzXML_idx_3.0.xsd`. There are multiple versions of mzXML; to read it correctly, the libraries need to know the format and version. The next element describes the run, providing metadata such as scan count and start/end time stamps as attributes. More metadata is supplied in the `<parentFile>` and `<msInstrument>` elements. The first piece of raw data is in the `<scan>` element. Here details of the first scan are given along with an element called `<peaks>`. The peaks element is the first place where the XML file becomes unreadable by humans. The binary-to-text conversion called base64 encoding is used to represent the raw data. In the XML file, the

binary arrays holding the m/z and intensity values are converted into printable text. The precision and byte order are needed to decode the text in an array, which in this case, will be 75 pairs of 32-bit floating point numbers.

From the `<scan>` element, you can also see that the data are from an MS level 1 spectrum (`msLevel="1"`). Each scan has a number starting at 1: `num=1`. Later in the run, there are MS level 2 spectra (`msLevel="2"`). Many mass spectrometry measurements combine MS levels with different MS scans and can also include chromatograms. This variety of data types combined with base64 encoding means that while general-purpose XML reading libraries will work to parse the file, the special-purpose packages in Bioconductor, specifically `MSnbase`, make working with XML data more manageable.

I'll begin by using low-level file reading functions in the `mzR` package to read the file and extract some metadata. In the preface, I installed the `MSnbase` package, which installs the `mzR` package as a dependency. The `mzR` package can be loaded independently like any other with the `library()` function.

```
library(Rcpp)
library(mzR)
```

The `mzR` package provides a virtual class, which is normally created using the `openMSfile()` function. This is the lowest level function you will need to read XML raw data files.

```
ms_file <- openMSfile(file.path("data","2017_04_08_TSW_Sponge.mzXML"))
```

The `openMSfile()` function opens the file and loads the contents into memory. In the second part of this example, I will show a way to access the data directly from the disk using the higher-level functions in `MSnbase`. The variable `ms_file` is an instance of an `mzR` object, with accessor functions used to extract metadata and data from the XML file.

First, to get information on the instrument, the `instrumentInfo()` function can be used:

```
inst_info <- instrumentInfo(ms_file)
```

This assignment produces a list called `inst_info`:

```
print(inst_info)
```

```
## $manufacturer
## [1] "Thermo Finnigan"
##
## $model
## [1] "unknown"
##
## $ionisation
## [1] "electrospray ionization"
##
## $analyzer
## [1] "fourier transform ion cyclotron resonance mass spectrometer"
##
## $detector
## [1] "unknown"
##
## $software
## [1] "Xcalibur software 2.5-204201/2.5.0.2042"
##
```

```
## $sample
## [1] ""
##
## $source
## [1] ""
```

Next, information on the spectra in the file can be obtained using the `run_info()` function:

```
run_info <- runInfo(ms_file)
print(run_info)
```

```
## $scanCount
## [1] 5703
##
## $lowMz
## [1] 50.0135
##
## $highMz
## [1] 3526.62
##
## $dStartTime
## [1] 60.14
##
## $dEndTime
## [1] 1320.5
##
## $msLevels
## [1] 1 2
##
## $startTimeStamp
## [1] NA
```

From these two lists, you can tell that this file is from a Thermo Oribitrap operating in FTMS mode, that the acquisition was collected from 60.14 seconds and ended at 1320.5 seconds, and that it contains both mass spectrometry (MS) and tandem mass spectrometry (MS/MS) data. Knowing the number of levels of MS performed is required to select how to examine the data further.

2.1.2 Exploring and Plotting

To begin plotting data, it's time to move up to the higher-level methods in `MSnbase`. First, I'll make a standard summary plot: the total ion chromatogram (TIC) for the MS level 1 data.

When the `MSnbase` package is loaded, it will print several messages to the console explaining what other packages `MSnbase` loaded as dependencies and what functions it masks from previously loaded packages. I have omitted the message output to save space. It might also vary with different versions of `MSnbase` in the future.

```
library(MSnbase)
```

Next, read the data file, using the `readMSData()` function:

```
ms_data <- readMSData(file.path("data","2017_04_08_TSW_Sponge.mzXML"),
                      mode = "onDisk")
```

The `readMSData()` function takes the argument `mode`, which allows data files to either be loaded into memory (the default) `mode = "inMemory"` or accessed directly from disk `mode = "onDisk"`. Depending on file size and memory available to R on your computer, you may want to experiment with both modes.

The object created by `readMSData()` is of the class `MSnExp` for the *inMemory* mode and the class `OnDiskMSnExp` for the *onDisk* mode. Many functions operate on `MSnExp` and `OnDiskMSnExp` objects, which help in computing a TIC. A TIC is almost always computed from MS level 1 spectra, which you can select with the `filterMsLevel()` function.

```
ms1_data <- filterMsLevel(ms_data, msLevel = 1)
```

Next, the `chromatogram()` function turns a collection of spectra into a TIC. The data in the example file were collected using a *data-dependent acquisition* (DDA) mode. In DDA, the peaks in an MS level 1 spectrum trigger MS level 2 acquisitions based on their m/z values and intensities. The aggregation function `max()` finds the highest intensity peak in each spectrum and uses it as the value for the chromatogram intensity at that time. Using `max()` creates a *base peak chromatogram* (BPC). In other situations, the TIC is more useful. To compute the TIC, set the aggregation function to `sum()`. Summing all peak intensities in a single spectrum creates the chromatogram intensity for that time point.

```
mz_tic <- chromatogram(ms1_data, aggregationFun = "sum")
```

Plot the TIC using the `plot()` method from R's base graphics package.

```
plot(mz_tic)
```

Figure 2.1 shows the TIC from the MS level 1 spectra in the example file.

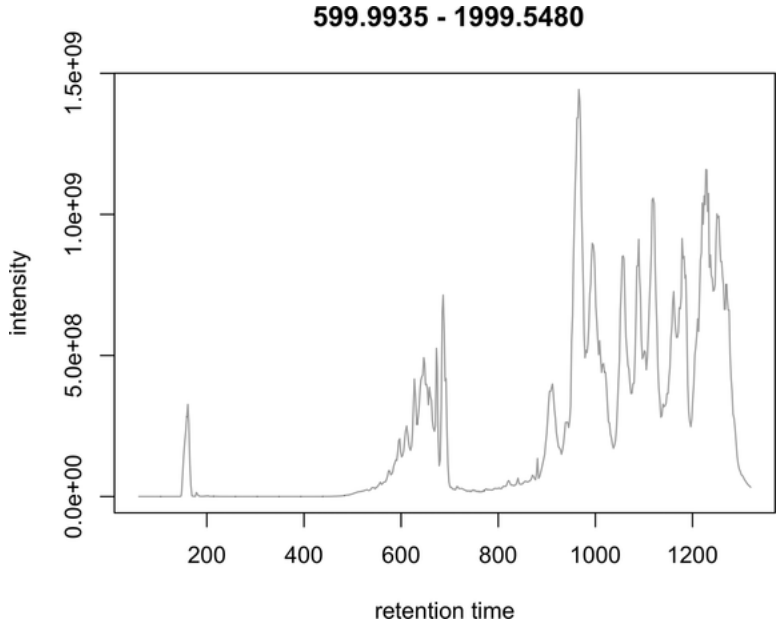


Figure 2.1 TIC plotted using R's base graphics system.

2.2 Using the Tidyverse in Mass Spectrometry

In the example code that produced Figure 2.1, each step created a variable for the following function regardless of any future need for that variable. Creating intermediate variables makes the flow of the process harder to read and forces you to come up with names for variables along the way. In addition, creating many intermediate variables throughout a program can be a source of confusion and make it difficult to understand a program. An example above is the variable `ms1_data`. It is only needed to pass into the `chromatogram()` function and contains no extra information not available in the `ms_data` variable.

A simple example of chained operations for the TIC plot, using the `tidyverse` plotting system `ggplot2`, is shown below:

```
ms_data <- readMSData(file.path("data","2017_04_08_TSW_Sponge.mzXML"),
                      mode = "onDisk")

ms1_tic <- filterMsLevel(ms_data, msLevel = 1) |>
          chromatogram(aggregationFun = "sum")

p_tic <- ms1_tic[1] |>
            (function(x) {tibble(MSnbase::rtime(x),
                                 MSnbase::intensity(x))})() |>
            setNames(c('rt', 'inten')) |>
            ggplot(aes(x=rt, y=inten)) +
                geom_line()
print(p_tic)
```

This code example uses several elements of the tidyverse. After loading the data using the `readMSData()` function as before, the next assignment uses a pipe to chain the output of a function `filterMsLevel()`, which selects the MS level 1 spectra, to the function `chromatogram()`, which then returns a TIC assigned to `ms1_tic`. You have no use for the filtered spectra except for passing them to the next step in the TIC creation process, so you don't need to name them and keep them as was done in the previous example. The `|>` (pipe) symbol for function chaining, sets the first argument of the following function in the program to the output of the previous one. In other words, it passed the filtered spectra into the first argument of `chromatogram()`. In the next example, I'll show how to add those values to the annotations of the plot, similar to how they were displayed using the base graphics `plot()` function.

Using pipes to chain steps in a process can make code easier to read. However, there are a few caveats. Pipes can make code harder to debug since there are no intermediate variables. It's considered best practice to keep chains to only a few steps and store any useful intermediate results. Creating the `ms_data` variable and the `ms1_tic` variable for later use made sense in this example.

Since the goal is to plot the chromatogram, you will ultimately need to pass a `data.frame` into the plotting function `ggplot()`. In this example, I used the `tidyverse` version of the `data.frame`: the `tibble`. The `tibble` is an updated version of the base class `data.frame`. Significant improvements make a `tibble` much easier to use than a `data.frame`, especially when working with other `tidyverse` packages. I will go into more detail in the next chapter, but one advantage is that the resulting `tibble` in the example is almost ten times smaller than the equivalent `data.frame`, which is likely to make further processing faster and use less memory.

In the code example, the line that creates the `tibble` and passes it into `ggplot()` is enclosed in an anonymous function. This is one of the differences between the `%>%` pipe and the native pipe. The native pipe expects the next item in the chain to be a function. Using an anonymous function allows you to control the `|>` operator where to insert the

output of the previous function. The %>% operator used *metaprogramming* to implement the piping operation and inserted the code needed for the operation to work. With the native pipe, the next item can be either a named or anonymous function. Without the notation `(function(x) {...})()`, the pipe would assume the next function is `tibble()` and pass the output of `ms1_tic[1]` as the first argument. This is not the behavior I need. To make a plot of the TIC, I need to make the tibble out of two vectors: the retention time (x), obtained using the `rtime()` function, and the ion intensity (y), obtained using the `intensity()` function. In R, the unnamed result created by the pipe gets assigned to the `x` symbol by the `function(x){}` statement. There is also a shortcut to reduce keystrokes for anonymous `functions: \(x)`. For most examples, I will spell out the word `function()` rather than use the `\()` just to be clear, but some examples and programs you find will use the shortcut. Even further, it is common to shortcut the variable `x` in `function(x)` with the `.` symbol so that the anonymous calls can be written as `function(.)` or `\(.)`. In the example, the output of `ms1_tic[1]` are passed into `rtime()` and `intensity()`, so the `x` variable inserts the first element of `ms1_tic` into both functions. Again, I could have used `function(.) {}` and then avoided having to name the incoming variable, which has no semantic meaning in this context, so the name is often something random like `x`. This is the biggest difference between the native pipe `|>` and the older `magrittr` pipe `%>%`. The native pipe requires the creation of a function to direct where the pipe sends the input into the next step in the chain. To finish, I want to specify the plot's x- and y-axis names, so I called the `setNames()` function, which gives names to the first and second columns in the `tibble` and then pipes the final result into `ggplot()`.

In this simple example, which produces Figure 2.2, the `tibble` containing the retention time (`rt`) and the intensity (`inten`) becomes the first argument to `ggplot()`. The key element of a graphic is what data to plot. For example, for a standard x –y plot, the x-and y-axis data are specified using the column names. In the *Grammar of Graphics* used by `ggplot2`, this is called the *aesthetic* (`aes`). To get a plot of a particular type, `ggplot()` uses the notion of adding things to the core graph. The simplest thing to add is the plot type, in this case, a line plot, which is added using the + symbol and specifying the geometry of the plot as a `geom_line()`. The `ggplot2` package is powerful and complex. In the next chapter, I will give more examples of how `ggplot()` can be used with mass spectrometry data. For now, I'd like to make one additional TIC plot to give a flavor of how much control `ggplot()` provides over the data display.

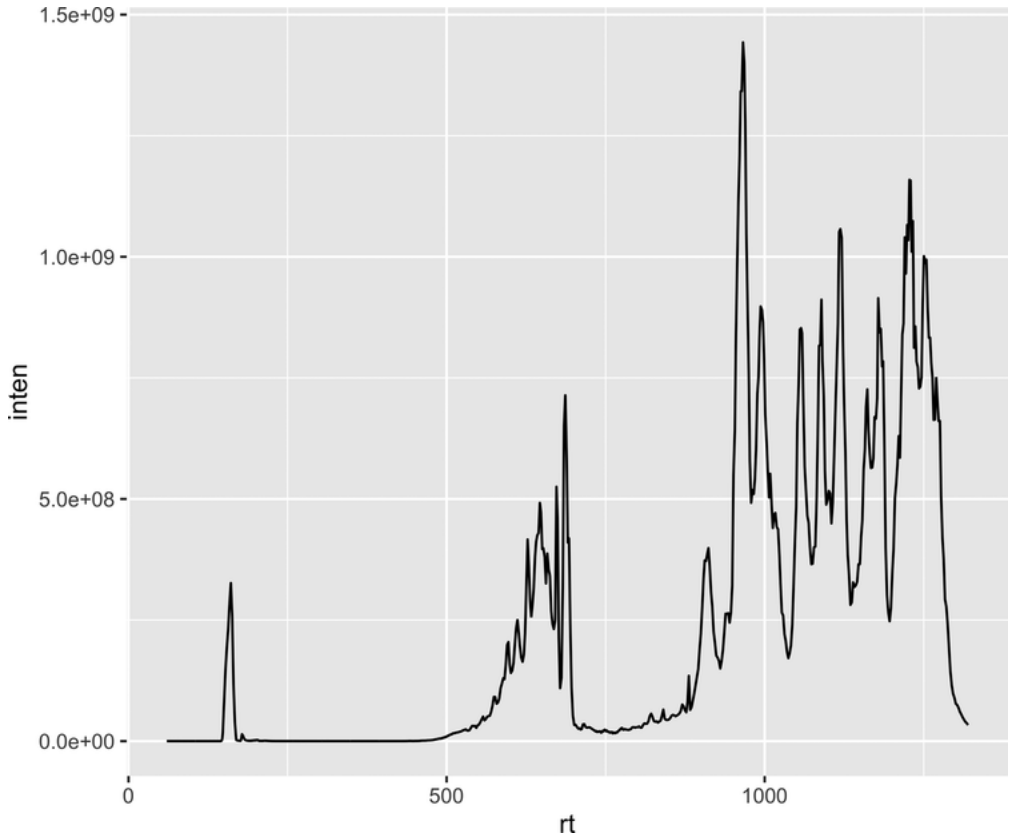


Figure 2.2 TIC plotted using ggplot2.

2.2.1 Customizing Plots Using `ggplot2`

Notice in Figure 2.1 that the title, the labels for the x- and y-axis, the tick marks, and even the background were all generated automatically because the `MChromatograms` class implements a custom `plot()` function using the base R graphics package. In Figure 2.2, some default behavior was provided, such as the background and reference lines based on the default tick mark spacing, but the axis labels, colors, and other attributes of the plot were not specialized to the content of the graphic. In this section, I will further customize the TIC plot to show how to add layers to a `ggplot2` graphic.

I have a personal preference for how I like to see mass spectra and chromatogram intensity values plotted. You might want to format your y-axis differently, or it may not matter if you are just doing something quick for exploration. Throughout this book I use the `scale_y_continuous()` function and pass into it a custom function called `inten_label()` for labeling the y-axis on many plots.

The function `inten_label()` uses the function `parse()` to generate each element in the array of y-axis labels. The one complicating factor in this method is that the equation used to convert the break values on the y-axis into scientific notation produces a nonsensical result for 0 intensity. To replace that value with the actual string "0," I used the `grepl()` function to look for `NaN`, which stands for *not a number*, which is the result when `break_value` is 0 in the line: `0/10^log10(0)`. I use the `base::grepl()` function, which, as I pointed out earlier, is needed because I want the `base` version of `grepl()`. However, the Bioconductor package `MSnbase` loads a package called `BiocGenerics`, which provides a different version of `grepl()`. It's important to use the exact function you want, even when

it's masked. In many cases, masking will not matter, but to simplify debugging, you should ensure you know which packages provide which functions.

```
inten_label <- function(break_value) {
        s <- sprintf(paste0("%.2f*x*10^%f"),
                     break_value / 10^floor(log10(abs(break_value))),
                     floor(log10(abs(break_value))))
        if (base::grepl("NaN", s[1]))
            s[1] = "0"
        return(parse(text=s))
}
```

Now, I will plot a more customized version of the TIC, which applies the specific kind of y-axis I want using `scale_y_continuous()`, as well as the background by adding the `theme_classic()` layer and a custom title and subtitle.

```
mz_range <- mz(ms1_tic)

p_clean_tic <- ms1_tic[1] |>
    (function(x) {tibble(MSnbase::rtime(x),
                         MSnbase::intensity(x))})() |>
    setNames(c('rt', 'inten')) |>
    ggplot(aes(x=rt, y=inten)) +
        geom_line() +
        scale_y_continuous(labels = inten_label) +
        xlab("Retention Time (s)") +
        ylab("Intensity (counts)") +
        theme_classic() +
        theme(plot.title = element_text(hjust = 0.5)) +
        theme(plot.subtitle = element_text(hjust = 0.5)) +
        ggtitle(label = "Base Peak Chromatogram - 2017_04_08_TSW_Sponge.mzXML",
                subtitle = sprintf("MS Level 1: mz %.2f - %.2f",
                                   mz_range[1], mz_range[2]))
print(p_clean_tic)
```

For Figure 2.3, I wanted to add the mass range information to the subtitle. You can access a spectrum's starting and ending m/z values with the `mz()` function. In this example, I stored the result of `mz()` as `mz_range`, which has two elements, the lower m/z in element 1 and the upper m/z in element 2. I used the `sprintf()` function to make a string and control the formatting of the numbers in the subtitle, and finally, I added meaningful axis labels using the `xlab()` and `ylab()` layers.

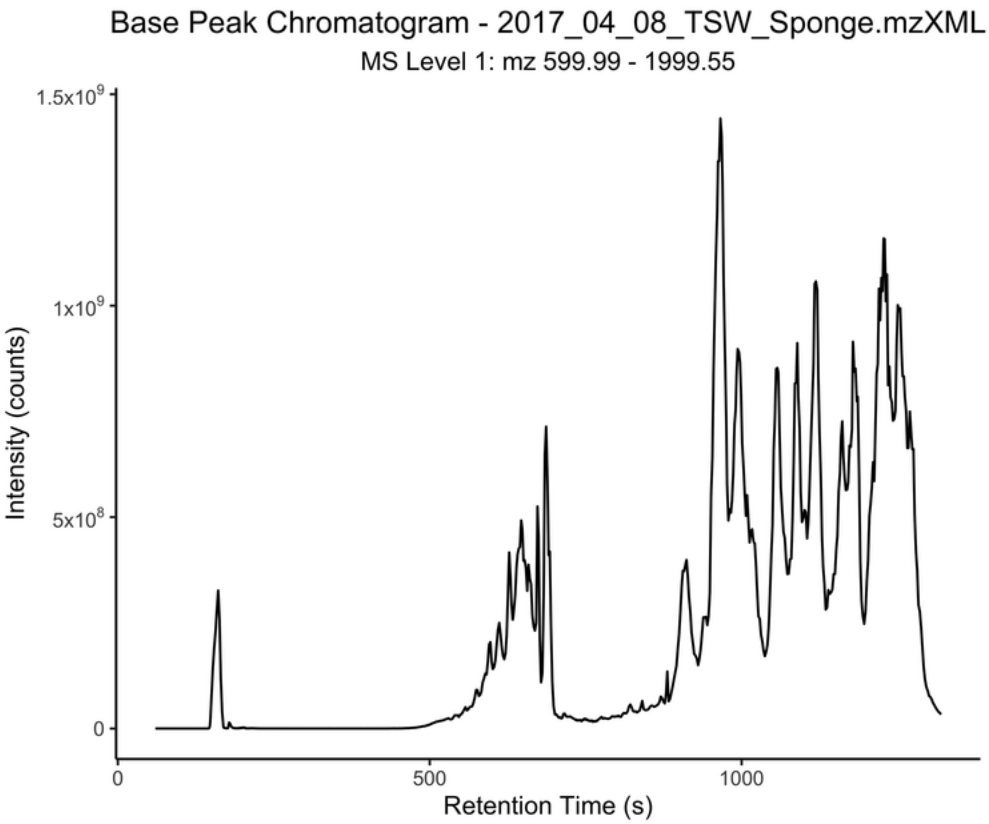


Figure 2.3 Customized TIC using the layering features of ggplot.

2.2.2 Tidy Mass Spectrometry Data

One of the central ideas expressed in `tidyverse` packages is the use of tables in a structured way – similar to the way tables are used in relational databases [46]. The `dplyr` package contains many functions to make working with data more manageable, with the notion that the first step in data analysis is to *tidy up*. To give a brief example of the main ideas, you can look at the example dataset used to create the TIC plot, dig into its structure more, and then *wrangle* some of the data into a form that is easier to use.

The `ms_data` variable holds all the run information created by `readMSData()`. I can use the `print()` function to display *some* of the metadata about the dataset:

```
  print(ms_data)
```

```
## MSn experiment data ("OnDiskMSnExp")
## Object size in memory: 2.12 Mb
## - - - Spectra data - - -
##  MS level(s): 1 2
##  Number of spectra: 5703
##  MSn retention times: 1:00 - 22:00 minutes
## - - - Processing information - - -
## Data loaded [Thu Aug 29 16:42:39 2024]
##  MSnbase version: 2.30.1
## - - - Meta data  - - -
## phenoData
##   rowNames: 2017_04_08_TSW_Sponge.mzXML
```

```
##   varLabels: sampleNames
##   varMetadata: labelDescription
## Loaded from:
##   2017_04_08_TSW_Sponge.mzXML
## protocolData: none
## featureData
##   featureNames: F1.S0001 F1.S0002 ... F1.S5703 (5703 total)
##   fvarLabels: fileIdx spIdx ... spectrum (35 total)
##   fvarMetadata: labelDescription
## experimentData: use 'experimentData(object)'
```

The output has a section `- - - Spectra data - - -` that shows that this dataset has both MS level 1 and MS level 2 data. The following example will separate the two levels and determine their relationship.

```
ms1_data <- filterMsLevel(ms_data, msLevel = 1)
ms2_data <- filterMsLevel(ms_data, msLevel = 2)
```

The `MSnExp` accessor function `fvarLabels()` shows the variables that are available from the MS level 2 scans:

```
fvarLabels(ms2_data)
```

```
##  [1] "fileIdx"                    "spIdx"
##  [3] "smoothed"                   "seqNum"
##  [5] "acquisitionNum"             "msLevel"
##  [7] "polarity"                   "originalPeaksCount"
##  [9] "totIonCurrent"              "retentionTime"
## [11] "basePeakMZ"                 "basePeakIntensity"
## [13] "collisionEnergy"            "ionisationEnergy"
## [15] "lowMZ"                      "highMZ"
## [17] "precursorScanNum"           "precursorMZ"
## [19] "precursorCharge"            "precursorIntensity"
## [21] "mergedScan"                 "mergedResultScanNum"
## [23] "mergedResultStartScanNum"   "mergedResultEndScanNum"
## [25] "injectionTime"              "filterString"
## [27] "spectrumId"                 "centroided"
## [29] "ionMobilityDriftTime"       "isolationWindowTargetMZ"
## [31] "isolationWindowLowerOffset" "isolationWindowUpperOffset"
## [33] "scanWindowLowerLimit"       "scanWindowUpperLimit"
## [35] "spectrum"
```

What is listed are 35 elements that are available for each MS level 2 scan. Of interest is the `precursorScanNum`, which indicates which scan in the MS level 1 data triggered the MS level 2 scan. I can also get the `precursorMZ`, the particular peak in the MS level 1 that the instrument fragmented to get the MS level 2 spectra.

```
fData(ms2_data)["precursorScanNum"] |>
    head(15)
```

```
##          precursorScanNum
## F1.S0251              250
## F1.S0252              250
## F1.S0253              250
## F1.S0254              250
## F1.S0255              250
## F1.S0256              250
```

```
## F1.S0257              250
## F1.S0258              250
## F1.S0259              250
## F1.S0260              250
## F1.S0262              261
## F1.S0263              261
## F1.S0264              261
## F1.S0265              261
## F1.S0266              261
```

The `data.frame`, produced by the `fData()` function, includes row names. `MSnbase` uses meaningful row names, in this case, a string describing the scan number of the MS level 2 scans. The data values are selected with the `[precursorScanNum]` syntax, since `precursorScanNum` was one of the parameters listed by the `fvarLabels()`. In this file, scans 251–260 are MS level 2 scans triggered by data in scan 250 as the precursor scan. You can see that there are 10 MS level 2 scans triggered from scan number 250 and 10 from scan 260. It could be that all MS level 2 trigger events have 10 scans, or there could be more for some and less for others, depending on how the instrument acquired the data. To explore this, let's summarize the type of scan data in this file using the base approach. The function `table()` can be used to create data summaries from a data frame. The default output of `table()` is the count of occurrences of values in a data frame. In this example, the `data.frame` has only one column: the precursor scan number, so `table()` will output a count of unique values of that column.

```
fData(ms2_data)["precursorScanNum"] |>
    table() |>
    head(15)
```

```
## precursorScanNum
## 250 261 272 283 294 305 316 327 338 349 360 387 431 438 448
##  10  10  10  10  10  10  10  10  10  10   2   5   1   1   1
```

This output shows that there must be a decision rule about how many MS level 2 scans are collected from a particular MS level 1 scan. Starting at scan 250, the next 9 MS level 1 scans triggered 10 MS level 2 scans, but no MS level 2 scans were collected for scans 363–386. Then the instrument collected two MS level 2 scans for scan 360 and five for scan 387. After that, the system collected only one MS level 2 scan for the scans up to number 448.

The `table()` function is useful but outputs a structure designed for display rather than use as data. For example, to access precursor scans, you access the names of each column in the table. To get these as strings, use the `names()` function. To use these values to get access to other information from the `ms_data` variable, they also have to be converted from strings to integers.

```
names(fData(ms2_data)["precursorScanNum"] |>
          table()) |>
          as.integer() |>
          head(15)
```

```
##  [1] 250 261 272 283 294 305 316 327 338 349 360 387 431 438 448
```

The process can be significantly simplified using the relational features of `dplyr` in the `tidyverse`. For example, rather than creating a table for display, you can create a `tibble` that holds the information you need and then use `summarise()` to perform the desired summarization.

```
ms2_parents <- tibble(fData(ms2_data)["spIdx"],
                      fData(ms2_data)["precursorScanNum"],
                      fData(ms2_data)["precursorMZ"],
                      fData(ms2_data)["precursorIntensity"],
                      fData(ms2_data)["retentionTime"])

summary <- ms2_parents |>
    group_by(precursorScanNum) |>
    summarise(scan_n = n())

head(summary, 15)
```

```
## # A tibble: 15 x 2
##    precursorScanNum scan_n
##               <int>  <int>
##  1              250     10
##  2              261     10
##  3              272     10
##  4              283     10
##  5              294     10
##  6              305     10
##  7              316     10
##  8              327     10
##  9              338     10
## 10              349     10
## 11              360      2
## 12              387      5
## 13              431      1
## 14              438      1
## 15              448      1
```

Another summary that can be useful is to determine the minimum intensity of precursor ions used to produce MS level 2 spectra. Low-intensity precursors may need to be filtered out since they might produce low-quality spectra.

```
min(ms2_parents["precursorIntensity"])
```

```
## [1] 0
```

Surprisingly, the minimum value of all precursor ion intensities is zero. Does that mean that MS/MS spectra were collected from nonexistent precursor ions? Before investigating further, knowing the type of data in the `precursorIntensity` column would help:

```
typeof(ms2_parents$precursorIntensity)
```

```
## [1] "double"
```

To perform conditional selection in base R, you pass a vector of Boolean values into `data.frame` via the selector operator `[]`. These expressions are very flexible, but they can occasionally be confusing. Rather than syntax, the `tidyverse` approach selects data using the `dplyr::filter()` function. Since the `precursorIntensity` is a variable of type `double`, comparisons need to be in terms of ranges. In this example, I want to know all the precursor intensities that *might as well be zero*, so let's select the rows where the intensity is less than 1. For an instrument using an ion counting detector, an intensity of less than 1 represents no signal. In Fourier transform and other instruments with analog

detectors, peak intensities are electrical signals that can take on any value. In practice, even electrical currents are often converted to integers, but the mzXML format used in this example stores both m/z and intensity as 32-bit floating point numbers. The `MSnExp` accessor function `fData()` returns intensity values as floating point numbers.

```
# override the default significant figures setting for printing numbers
old <- options(pillar.sigfig = 7)

dplyr::filter(ms2_parents, precursorIntensity <= 1)
```

```
## # A tibble: 3 x 5
##   spIdx precursorScanNum precursorMZ precursorIntensity retentionTime
##   <int>            <int>       <dbl>              <dbl>         <dbl>
## 1   340              338     1032.77                  0       165.504
## 2   341              338     1042.8                   0       165.694
## 3   361              360      626.63                  0       169.565
```

```
# return to the default options
options(old)
```

A quick look at the first scan returned with a zero precursor ion intensity shows no mistake in the data access as the `<precursorMz>` element clearly shows `precursorIntensity="0"` for the precursor ion at m/z `1032.77`.

```
<scan num="340" msLevel="2" peaksCount="23" polarity="+" scanType="Full"
    filterLine="FTMS + p ESI d Full ms2 1032.77@hcd35.00 [71.67-1075.00]"
    retentionTime="PT165.504S" lowMz="72.9376" highMz="309.894"
    basePeakMz="92.1098" basePeakIntensity="10183.8" totIonCurrent="50637.6"
    collisionEnergy="35">
    <precursorMz precursorIntensity="0" activationMethod="HCD">
        1032.77
    </precursorMz>
    <peaks precision="32" byteOrder="network" pairOrder="m/z-int">
        QpHgB0QvqgpCoeWiRUt8d0K19DhEdOhsQsHYNUWBfRNCxda9RXTj30LhypxFp0LcQuXJk0SA
        PtZC8AGsRE3JVEMCnXREJ/qiQxw2e0RMOO1DJ2mwRDRcvEMp/FZFVsmBQzn1kkSMAb5DTuSr
        RKK/lkNQ48ZEV+iEQ1jsO0TqU4tDXt3uRGhMNEN+0rRElPAcQ4Bo90VeS6RDhG1BRF7Y9UOF
        bKdEjGm6Q47zmkRz76FDmvJyREJdQw==
    </peaks>
</scan>
```

It is unusual for a precursor ion with zero intensity to be used to trigger an MS level 2 acquisition. One possibility is that the precursor was on an *inclusion list* from a previous MS level 1 spectrum. In later chapters, I'll go into more depth about inclusion and exclusion lists, but to check that possibility, I can write a function to pair two acquisitions and then use `facet_grid()` to plot and label the two spectra.

First, I will create a function called `get_survey()` that will return the selected scan numbers as a single `tibble`.

```
get_survey <- function(dataset, scan_num_1, scan_num_2 ) {
    bind_rows(
        tibble(
            scan = scan_num_1,
            mz = mz(spectra(dataset[scan_num_1])[[1]]),
            intensity = MSnbase::intensity(spectra(ms_data[scan_num_1])[[1]])),
        tibble(scan = scan_num_2,
            mz = mz(spectra(dataset[scan_num_2])[[1]]),
            intensity = MSnbase::intensity(spectra(ms_data[scan_num_2])[[1]])))
}
```

Now, use `get_survey()` to get the two spectra, scan 338 and the preceding MS level 1 scan, which was scan 327. Filter the result to the mass range near the precursor ion selected in scan 338 and then plot it with the preceding MS level 1 scan. What can be seen in Figure 2.4 is that the m/z value 1032.77 was present in scan 327 with reasonable abundance and somehow made it onto the inclusion list for scan 338 even though the intensity of that ion intensity in scan 338 was zero.

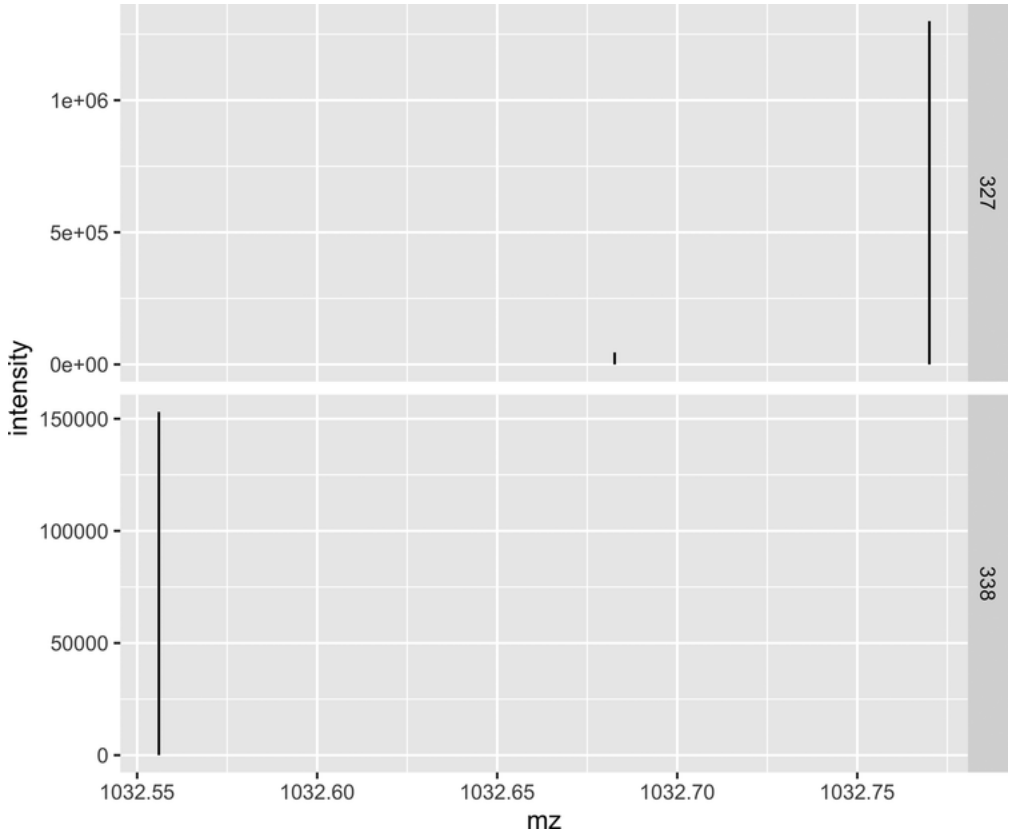


Figure 2.4 Comparison of adjacent MS level 1 scans: 327 and 338 for precursor m/z 1032.77.

In the `facet_grid()` layer, an R *formula* is used to indicate that the scan number `scan` represents columns and ~ is read as *is described by* where the `.` means all the other columns. The remaining columns that describe `scan` are, therefore, `mz` and `intensity`. The R formula syntax was designed for describing statistical models [47], but it has many other uses, including relating variables in plots.

```
p_spec_facet_a <- get_survey(ms_data, 327, 338) |>
        dplyr::filter(between(mz, 1032.5, 1033.0)) |>
        ggplot(aes(x=mz, y=intensity, ymax=intensity, ymin=0)) +
            geom_linerange() +
            facet_grid(scan ~ ., scales = "free_y")

print(p_spec_facet_a)
```

Using the same approach, and importantly, the same function `get_survey()`, the comparisons can be made for the remaining anomalous precursors. See Figures 2.5 and 2.6.

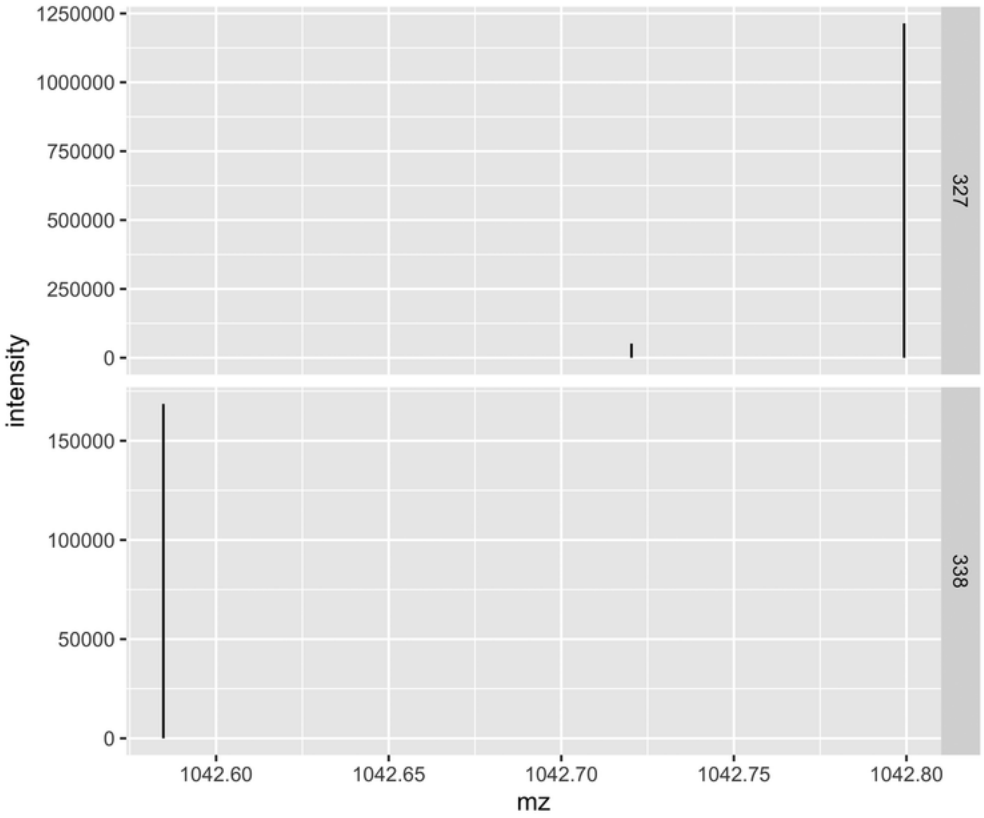


Figure 2.5 Comparison of adjacent MS level 1 scans: 327 and 338 for precursor m/z 1042.8.

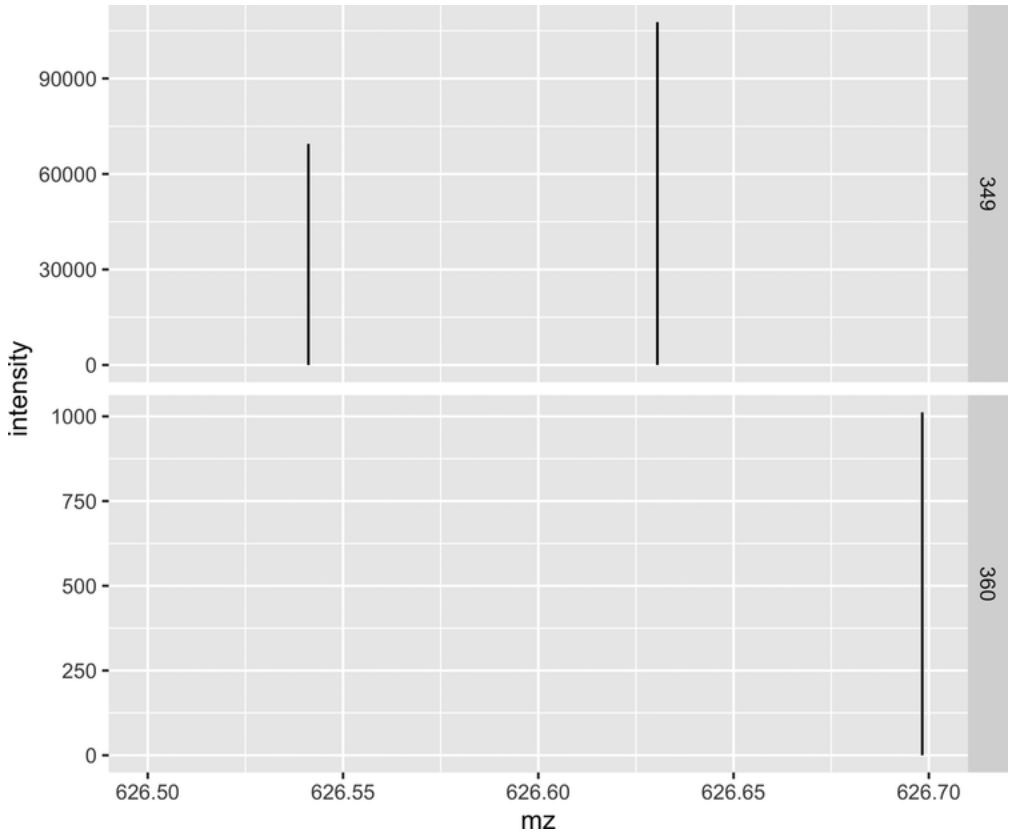


Figure 2.6 Comparison of adjacent MS level 1 scans: 349 and 360 for precursor m/z 626.63.

```
p_spec_facet_b <- get_survey(ms_data, 327, 338) |>
    dplyr::filter(between(mz, 1042.5, 1043.0)) |>
    ggplot(aes(x=mz, y=intensity, ymax=intensity, ymin=0)) +
        geom_linerange() +
        facet_grid(scan ~ ., scales = "free_y")

print(p_spec_facet_b)
```

```
p_spec_facet_c <- get_survey(ms_data, 349, 360) |>
    dplyr::filter(between(mz, 626.3, 626.7)) |>
    ggplot(aes(x=mz, y=intensity, ymax=intensity, ymin=0)) +
        geom_linerange() +
        xlim(626.5, 626.7) +
        facet_grid(scan ~ ., scales = "free_y")

print(p_spec_facet_c)
```

One compelling reason to keep data in a tidy format is that it makes data access and manipulation much more straightforward than other ways of organizing data. Tidy data can be thought of like data in traditional database tables. The `dplyr` package is organized around the idea that functions are *verbs* that operate tables. I'll use tidy data in the following example to perform logical joins between tables with a shared key. Because of what I discovered about the precursor ions in this data set, it will be helpful to analyze the

noise in the MS level 2 spectra so that noisy spectra are not passed forward in an analysis workflow.

Several publications show how to estimate noise in MS level 2 spectra of peptides for proteomics. One simple algorithm computes the "dynamic noise level" [48]. The `dnl()` function implements the algorithm.

```
dnl <- function(.data, threshold = 0.5) {

    # sort the spectrum in order of increasing intensity

    sorted_spec <- arrange(.data, intensity) |>
                   (function(x) {mutate(x, index = row_number(x$intensity))})()

    # perform a linear regression on all of the peaks above index 2

    m <- sorted_spec |>
            dplyr::filter(index > 2) |>
            (function(x) {lm(intensity ~ index, data = x)})()

    # compute the signal-to-noise ratio by dividing the measured intensity
    # by the predicted intensity from the linear model

    sorted_spec <- mutate(sorted_spec,
                          snr = sorted_spec$intensity/predict.lm(m, sorted_spec))

    # assume the lowest intensity peak is noise (SNR=1)

    sorted_spec$snr[1] = 1

    # compute the SNR of the second peak based on the intensity of the first
    # using a threshold factor that defaults to 1.5

    sorted_spec$snr[2] = sorted_spec$snr[2]/
                         (sorted_spec$intensity[1] * (1 + threshold))

    # return the number of peaks above the minimum SNR of 2 and
    # the sum of all the peak intensities below SNR of 2
    # as a tibble

    tibble(dplyr::filter(sorted_spec, snr >= 2) |> summarise(peaks = n()),
       dplyr::filter(sorted_spec, snr < 2) |> summarise(noise = sum(intensity)))

}
```

In Figure 2.3, the most prominent chromatographic peak appears at around 950 seconds. Using the dynamic noise level function `dnl()` above, you can iterate through this subset of the MS level 2 spectra and compute the number of peaks with an SNR greater than 2 and the total amount of noise in each spectrum. Next, the program stores the results in a tibble called `dnl_results`.

```
# start and end retention times (in seconds) of the region of interest
start_time = 935
end_time = 985

rt_region = filterRt(ms2_data, c(start_time, end_time))

for (spec in 1:length(rt_region)) {

    spec_dnl <- tibble(
            mz = mz(spectra(rt_region[spec])[[1]]),
            intensity = MSnbase::intensity(spectra(rt_region[spec])[[1]])) |>
        dnl() |>
        mutate(scan=fData(rt_region)["spIdx"][[1]][spec], .before=peaks)

    if(spec == 1) {
        dnl_results <- spec_dnl
    } else {
        dnl_results <- add_row(dnl_results, spec_dnl)
    }
}
```

I've characterized the highest intensity MS level 1 region of the chromatogram. Now, I can join the results of noise analysis `dnl_results` with the `tibble` I created earlier that holds the precursor scan information `ms2_parents`. Since the two tables share a common column, `dplyr` provides the `left_join()` function, which, as the name implies, performs a *left join* that matches rows in the first table (left) with the rows in the second (right) table when the values of specified columns match.

```
ms2_summary <- dnl_results |> left_join(ms2_parents, by=c("scan" = "spIdx"))
head(ms2_summary, 10)
```

```
## # A tibble: 10 x 7
##     scan peaks   noise precursorScanNum precursorMZ precursorIntensity
##    <int> <int>   <dbl>            <int>       <dbl>              <dbl>
##  1  3557     1 204702.             3554        846.            2536920
##  2  3558     2 102598.             3554        630.            1793200
##  3  3559     3 134141.             3554        626.            1706860
##  4  3560     1 202821.             3554        846.            2536920
##  5  3561     5 409428.             3554       1393.            1203720
##  6  3562     1  54268.             3554        851.            1079130
##  7  3563     5 441688.             3554       1393.            1203720
##  8  3564     4  52712.             3554        818.             953234
##  9  3566     7 147042.             3565        675.            1100390
## 10  3567     7 123841.             3565        675.            1100390
## # i 1 more variable: retentionTime <dbl>
```

How many noise values are there between `start_time` and `end_time`? The `nrow()` from base R gives the number of rows in a `data.frame` or a `tibble`:

```
nrow(dnl_results)
```

```
## [1] 260
```

In the ms2_parents there were `nrow(ms2_parents)` rows:

```
nrow(ms2_parents)
```

```
## [1] 4027
```

In the `left_join()` performed above, the 260 rows from `dnl_results` were joined with the 4027 rows in `ms2_parents` by matching the `scan` value in `dnl_results` with the `spIdx` column in `ms2_parents`. Performing the left join operation starting with `dnl_results` keeps only the matches for the `scan` column, so the length of `ms2_summary` should be 260:

```
nrow(ms2_summary)
```

```
## [1] 260
```

To conclude this section, I'd like to show how R can make highly readable tables. In Section 2.3 below, I will introduce dynamic reports using RMarkdown using the `knitr` package. The `kable()` function from the `knitr` package generates high-quality tables. I used the console output in the previous examples when displaying a `data.frame`. While this is quick and convenient, it is not attractive or easy to read. You can use the `kable()` function to make tables in a variety of formats (Table 2.1):

```
knitr::kable(head(ms2_summary, 15), format="pipe",
             caption="MS2 Spectral Quality Summary")
```

TABLE 2.1

MS2 spectral quality summary.

scan	peaks	noise	precursorScanNum	precursorMZ	precursorIntensity	retentio
3557	1	204701.90	3554	846.4405	2536920	935.0
3558	2	102598.24	3554	630.4591	1793200	935.2
3559	3	134141.33	3554	626.3974	1706860	935.4
3560	1	202820.78	3554	846.4405	2536920	935.6
3561	5	409427.90	3554	1392.9473	1203720	935.8
3562	1	54268.43	3554	851.3958	1079130	935.9
3563	5	441688.26	3554	1392.9473	1203720	936.1
3564	4	52712.04	3554	818.0127	953234	936.3
3566	7	147042.13	3565	675.4562	1100390	936.7
3567	7	123840.67	3565	675.4562	1100390	936.9
3568	0	25026.73	3565	1144.1958	369413	937.1
3569	9	435980.30	3565	1402.9143	2280310	937.3
3570	6	195635.68	3565	662.4293	2091100	937.5
3571	10	482915.21	3565	1402.9143	2280310	937.7
3572	8	550243.90	3565	1407.8700	2161240	937.9

The pattern for examples in this book uses application-specific packages, such as those from `Bioconductor` (and other collections), and then moves the results into tidy data structures. I have touched on several aspects of the `tidyverse` for plotting and keeping data clean and usable. The focus on tidy data leads to programs that are easier to

understand than those that use many different data structures from the many mass spectrometry-specific functions available. To conclude this example, I'll introduce one more topic: How to create executable reports that integrate the analysis's narrative with the code used to perform all the calculations. These dynamic reports will use RMarkdown and the `knitr` package.

2.3 Dynamic Reports with RMarkdown

Making research more reproducible is a complicated task with many different types of issues. This book focuses on *reproducible computation* to help make research results more transparent, organized, and inclusive. The approach recommended by the R community is to use dynamic documents using Markdown. The Markdown concept has significantly impacted software development and other technical fields. Because Markdown documents are flat text files, they are inherently portable and resistant to technological changes. The `knitr` package extends the Markdown concept to include executable code. Because of the excellent support of `knitr` and RMarkdown in the RStudio IDE, it is now quite simple to create reports that include the explanation of an experiment and its analysis along with the executable code used to perform the analysis, in the report. I want to show how easy it is to create a literate program based on some of the analyses performed earlier in the chapter.

All RMarkdown documents contain two main parts: metadata in YAML (yet another Markdown language) and the document's body, which is plain text interpreted using Markdown conventions and code chunks. Code chunks are blocks of executable code that can generate output that is incorporated into the final document. The `knitr` package can generate many output formats from an RMarkdown document by using the `pandoc` program [49]. Creating HTML or PDF files via LaTeX is typical, but using `pandoc`, `knitr` can generate many other formats.

The YAML header begins and ends with `---` and must at least include the `output:` type but can also include title, author, and date information. After the header, you can mix text following markdown conventions and R code.

````
---
title: "Base Peak Chromatograms from 2017_04_08_TSW_Sponge"
author: "Randall Julian"
date: "2023-10-07"
output: html_document
---

The following code will load the MSnbase and tidyverse libraries and then read
an MS data file using the readMSData() function using the "onDisk" option which
saves on memory usage. The program will compute the base peak chromatogram
(using the aggregationFun = "max" option) and finally, plot the chromatogram
using ggplot().

```{r message=FALSE, warning=FALSE}
library(MSnbase)
library(tidyverse)

ms1_tic <-  readMSData(file.path("data","2017_04_08_TSW_Sponge.mzXML"),
                       mode="onDisk") |>
            filterMsLevel(msLevel = 1) |>
            chromatogram(aggregationFun = "max")

ms1_tic[1] |>
    (function(x) {tibble(MSnbase::rtime(x), MSnbase::intensity(x))})() |>
    setNames(c('rt', 'inten')) |>
    ggplot(aes(x=rt, y=inten)) +
    geom_line()
```
````

This code generates the output shown in Figure 2.7:

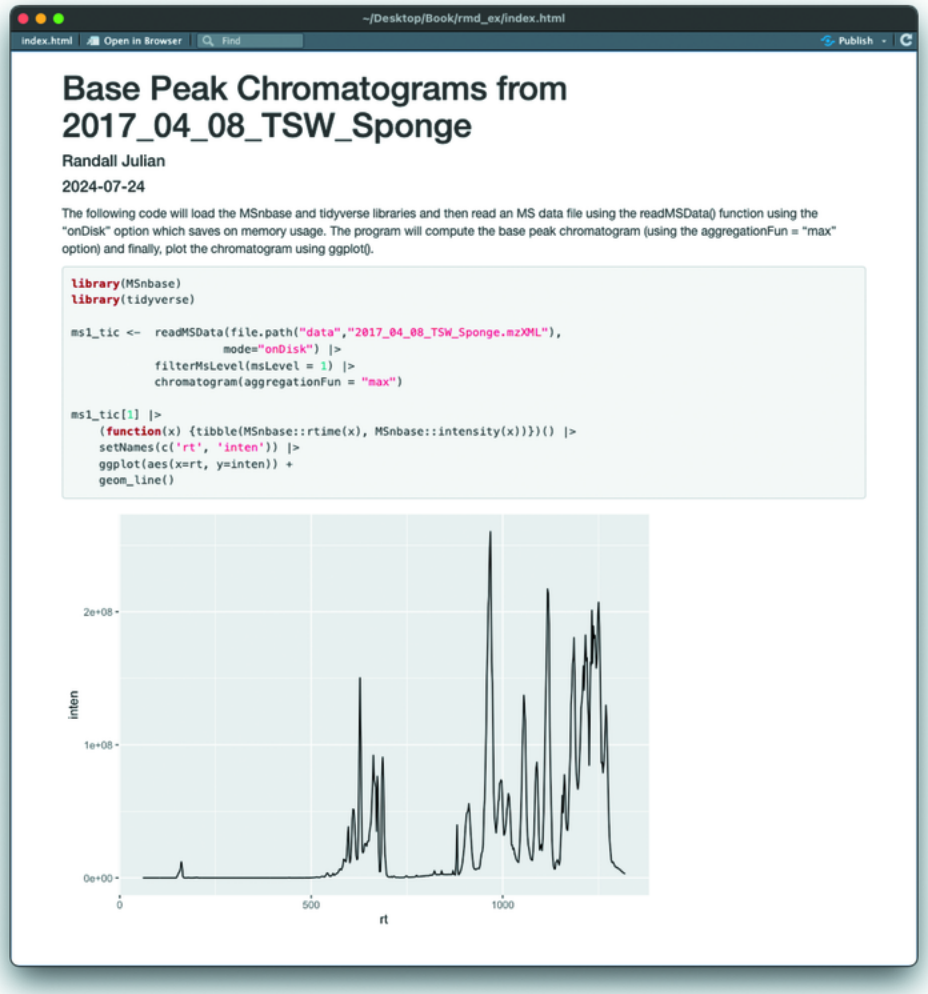


Figure 2.7 HTML output from RMarkdown.

2.4 Summary

In this chapter, I have given a *very* brief introduction to using Base R, the tidyverse, and packages in the Bioconductor repository in the form of a dynamic document. In the next chapter, I begin a more detailed discussion on each phase of data analysis: reading, wrangling, exploring, and analyzing.

Chapter 3
Wrangling Mass Spectrometry Data

3.1 Introduction

Data wrangling is the phrase the data science community adopted to describe steps needed to get data organized for analysis. While mass spectrometers are instruments which produce a raw data stream, meaning that while operating a detector is generating a signal, usually an ion current or ion counts. Mass spectrometers can be configured to make many different types of measurements, and the control system organizes the signal, digitizes it, and saves it as *raw data*. In addition to raw data; therefore, the data *describing the measurement* is required for any type of analysis. At a higher level, the organization of the experiment can create data that can be used to determine the quality of a specific measurement and determine the limits of an analysis. At the next level up, whole experiments can be saved and organized to aid in interpreting raw data. Examples of this data include libraries of spectra, libraries of structures, and other chemical information.

In this chapter, I will look at different types of data associated with mass spectrometry and show ways to manipulate them to make analysis easier. Most mass spectrometry data do not start off in tidy formats described so far. Mass spectrometry data analysis tasks can usually be described as analysis pipelines. There have been many data analysis pipelines described for various mass spectrometry experimental analysis. These are almost always linear processes that begin with reading data and end with generating a report. As described by Hadley Wickham et al. in *R For Data Science* [30, 31], data science follows an

iterative process. In some cases, this initial work will produce a linear pipeline that can be reused for specific problems. An important goal of this book is to help you perform analysis and borrow components of previous work without necessarily forcing your analysis problem into the shape of an existing pipeline.

3.2 Accessing Mass Spectrometry Data

Most of the data used for mass spectrometry analysis resides in files stored on a disk. There are many ways to access data in files using R. Simple text files, like the comma-separated values (CSV) files used in the introductory example, can be read directly using base R functions or tidyverse functions. For more complex files, such as the extensible markup language (XML) files also shown in the earlier example may be more easily read by using an application-specific function like `mzR` described in Chapter 2. Simple file readers assume a minimum structure, such as an encoding or specifying field delimiters. Complex file readers like `mzR` assume a very specific use of a particular file format, such as a published way to encode data in XML. You can also read file formats using specifications like XML, with assuming that the data follows any specific data encoding. Intermediate -level file readers can read files following standards like XML without making any further assumptions. The `read_xml()` function in the `xml2` package is a useful example of an intermediate -level file reader. With XML, the reader can at least tell if the file is well-formed before trying to parse it into a data structure. Needing to read files at all levels, from simple to complex, is common in mass spectrometry. If you find an application-specific function to read a file containing data, it is usually best to use it if it gives you access to the data you want. The goal will then be to translate that data into a tidy format. Application-specific functions are sometimes designed to be part of a particular analysis process. If that process is

different than what you want to implement, you may find it doesn't give you access to the data you want in the format you need. In that case, you will need to drop down a level and use a file reader that makes fewer assumptions about the file. For example, `mzR` assumes that the data file follows the XML syntax, and, further, that it uses XML to store data according to a supported mass spectrometry XML schema. If the data you want to read uses the XML syntax but is not one of the supported specifications, you can drop that assumption and use an XML parser. If the file does not follow the XML specification, you can still read the file using a character reader such as the `read_file()` function from the `readr` package, which is part of the `tidyverse`.

Scientific data is often stored using a binary rather than text format. Like XML, there are binary specifications that can be read with an intermediate function. Specifications such as *netCDF* [50] and *HDF5* [51] are widely used, platform-independent binary formats. Both of these formats are used in mass spectrometry and described later in this chapter. If you need to read a binary file directly, the `ReadBin()` function is available in Base R. To use these low-level functions, you will have to know much more about the binary file, but if the specification is published, with extra work, you can usually read any file.

Often a function that reads a binary file will link to an *application programming interface* (API) written in another language to make reading more efficient. For example, the `xml2` package was built using the `libxml2` library, which is written in `C`. Some of the vendor-specific file formats described in this chapter use this approach, but building on an API supplied by the vendor. Sometimes, this will require running the program on the vendor-specific CPU and OS configuration. Accessing data via APIs will be discussed below, but one of the main motivations for creating open, text-based file formats is to avoid having to read vendor formats directly unless it's necessary.

3.2.1 Open, Closed, Binary, and Text File Formats

It is common practice for software to use a proprietary, application-specific format to store data. Developers can optimize the format for the needs of the program they are writing and its users. Using this approach, the program manages the information needed to read and write files within the source code. Files of this type can be exchanged between users of the programs, as long as they have the particular program that originally wrote the file. If a file format is *closed*, then the file is not designed to be read or written by anyone other than the developer of the original software. An example of this approach is the early `xls` file format used by Microsoft Excel. It is a proprietary format, readable by anyone with Excel. For other programs to use the format, the details had to be either disclosed by Microsoft or the format had to be reverse engineered. There are several advantages to the developer of the closed format approach. These formats are usually optimized for performance, and the developer can change them when their program changes and its storage requirements change. As long as the program keeps track of how to read various versions of the file, it can remain backward compatible and gives the developer flexibility and speed. At the time of writing, the Microsoft website reports that the `xls` format has had 10 major versions since 2010, with a total of 49 different version numbers. Microsoft switched the default format for Excel and started publishing the specification of the `xls` and other formats [52], which made it easier for other programmers to read files of that type. Microsoft effectively *opened* the `xls` format while leaving it a binary format and fully under its control.

An *open* format is one that is publicly described and can be either binary or text. Excel provides another good example of this. In 2007, Microsoft switched to its Open Office Format `xlsx`, which is a set of directories and text files compressed into a single file using the `zip` compression format. The

uncompressed text files can be opened by any program that can read plain text, and the specification for the contents is publicly available [53]. Like the `xls` format, the `xlsx` specification has been revised multiple times and has 24 major versions.

In mass spectrometry, instrument vendors typically use a closed format optimized for performance. These formats are almost exclusively binary rather than text files. Further, most of the files you encounter in mass spectrometry are closed, and their binary formats are not disclosed. For various reasons, there has been no equivalent movement to open these formats, as was done by Microsoft. Instead, instrument vendors have chosen to make the data available via programming interface libraries. You can use a programming language to call a vendor library to extract some, but usually not all, of the information from a mass spectrometer. Over the years, vendors have also provided various export formats that can be generated from within their programs. They have also provided mechanisms to import some types of data, typically work instructions or instrument methods.

The desire to use more powerful computing systems which use UNIX-like operating systems rather than the one used to control the instrument (typically a version of Microsoft Windows) created the need to move beyond the closed formats to open formats. The intent of these formats was not to exchange data between instrument types (although this was briefly achieved with the ANDI/AIA standard), but to exchange data between different computing systems and allow researchers to write their own programs to perform data analysis. In order to perform data interchange, the specification had to describe both the syntax (where are data elements and what is their data type) and the semantics (what do the values of a data element mean) for the file.

Early data exchange formats for mass spectrometry were simple text files containing relatively simple data. As mass

spectrometry has evolved, experiments have become more sophisticated, and data files now need to describe more complex configurations and relationships. Fortunately, all of the science evolved with the development of computer systems, and the need to exchange data of all types has become a central issue for computer science. It's not a coincidence that hypertext markup language (HTML) and the World Wide Web were developed to exchange complex scientific information. The mass spectrometry community has been part of the push to improve data exchange since the first computers were used to control instruments. In its formats, you can see the evolution of data exchange technology.

3.2.2 Syntax, Semantics, and Controlled Vocabularies

The syntax of a file is the description of the structure of the data. The description specifies the data elements, their relationships, their data types, how types are represented, and what constitutes a complete file. Another word for this is the *schema* of the data. File formats like XML and JavaScript Object Notation (JSON), and others, can store the schema for a data file format as a schema file that can be read and used by a program to make sure the data file is *syntactically correct* before attempting to parse the file. In order to make decisions based on data in a file, a program also has to have information about what the data elements *mean*. This includes, for example, what text strings are allowed in a data element or the units of a numeric value. Like in natural language, it is possible to create a data file that is syntactically correct and readable by a program but meaningless in terms of describing an experiment. Valid files are said to be *semantically correct*. Semantic specifications define the allowed values for data elements, often in the form of *controlled vocabularies*. Like other aspects of file formats,

controlled vocabularies can either be open or closed. For a program that is maintained by a software developer, the only requirement is that their program be able to use their own files. The actual values and units of data elements do not need to follow any particular external convention. Further, if the developer publishes the specification, then it is up to any other program to interpret the values according to that specification, regardless of scientific convention. This can lead to a file format being so application-specific that its use for other programs is limited for purposes of data interchange.

Standard formats are developed to allow interchange by a scientific community, usually by working groups of scientific societies or by industry associations. The result is an agreement on the limits of the use of the format (what will and will not be covered), the minimum schema, and the minimum controlled vocabulary. The best of these standards undergo a peer review process and result in something resembling a computer-readable version of a scientific paper. The advantage of standards is that changes are made in such a way that allows developers to maintain their programs and include requests from the community for enhanced functionality or expanded scope or semantic content. Even if a format is open, when it is controlled by a single developer, changes can be made independently without regard to community expectations or requirements. So, like all things that are *built by committee*, standards change slowly and often reflect a lowest common denominator among many competing interests.

A middle ground can be achieved to make standards stable enough for reliable software development but flexible enough to keep up with a rapidly changing scientific area like mass spectrometry. The result is an increase in complexity in specifying both the syntax and the semantics of a format. Separating the schema from the controlled vocabulary is one way to allow a format to change over time; this has been done

in most recent standardization efforts in mass spectrometry, where the syntax is specified by a schema and the controlled vocabulary is specified by an ontology managed by a separate process. This way the syntactic validity of a file can be checked before attempting to read it, and once read, its semantic validity with regard to a specific application can be checked before operations using the file are performed. This allows data to be interchanged reliably, with varying levels of descriptive power and keeps changes to programs to a manageable level, ensuring that files are valid so that program behavior can be predictable.

In the next sections, I will show examples of some of the common file formats used in mass spectrometry and describe how to *wrangle* them into tidy structures that can be used to take full advantage of the `tidyverse` and other modern programming paradigms used in R.

3.3 Types of Mass Spectrometry Data

We use and produce many types of data in the analysis of mass spectrometry experiments [54]. A summary of these data types and their relationships to various applications and analysis steps is shown in Figure 3.1.

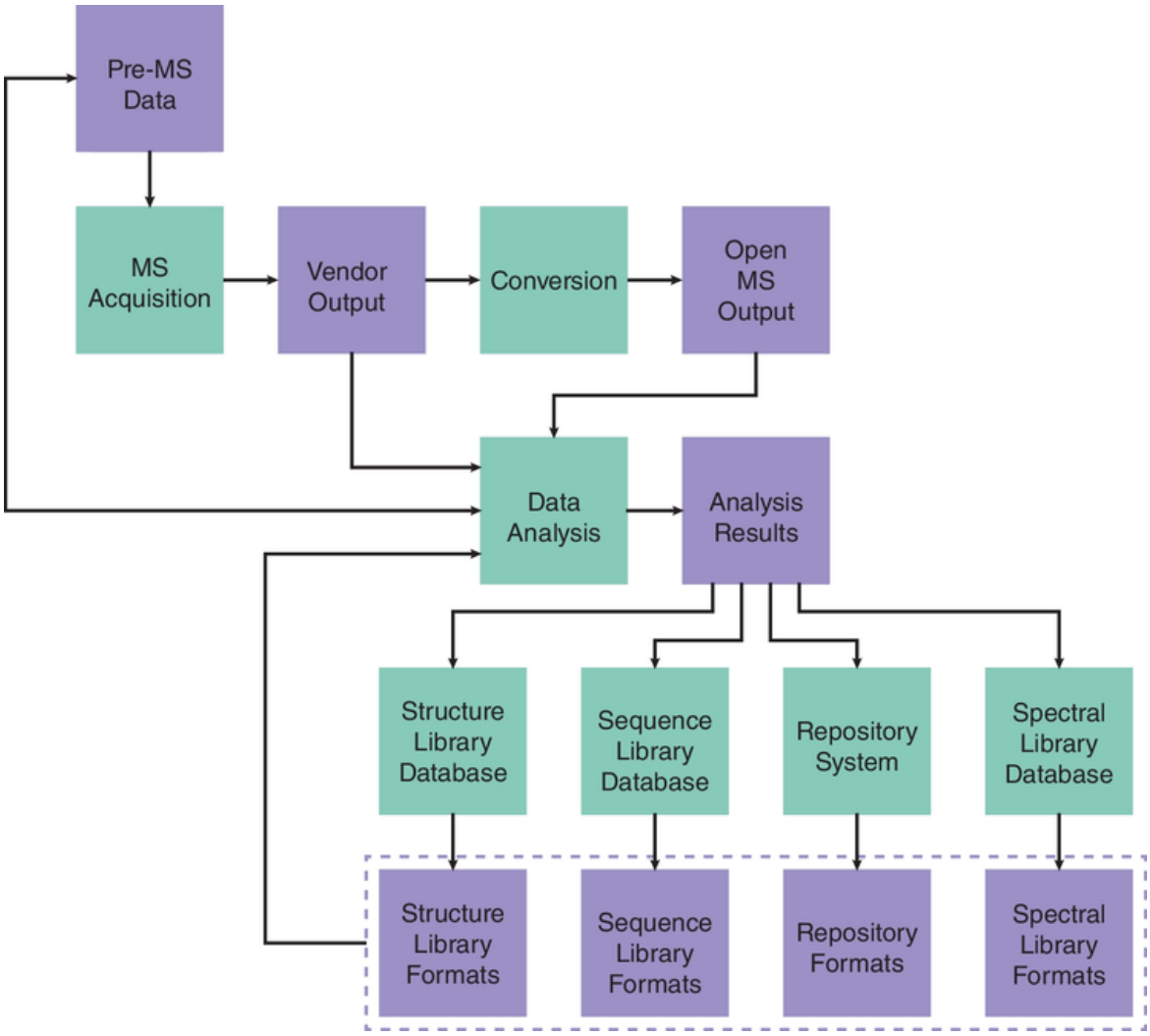


Figure 3.1 Summary of data types and applications used in mass spectrometry.

Data about the sample and the instrument configuration are used by the vendor acquisition application to label the data (sample names, concentrations for calibrations, etc.) and to set parameters on the instrument (m/z ranges, voltages, flow rates, etc.). The application that performs the acquisition creates a vendor-specific output, which can be a file or set of files. As described in Section 3.2.1, these formats are often closed binary formats. Conversion programs are used to convert these files to open formats that can more easily be read by data analysis applications. Since file conversion programs that read the vendor binary files directly use

vendor-supplied code libraries, these code libraries can also be used directly to write data analysis programs that read vendor-specific binary files.

Data analysis programs span the range from vendor-provided applications to programs written by individual researchers to answer a particular question. Data analysis programs that perform design-of-experiment calculations, for example, can write files describing pre-mass spectrometry steps describing which samples are to be processed, and in what order, and provide settings to optimize for a specific outcome. Other types of data analysis applications can create visualizations and no other output, or they can create analysis results that have a use in future analysis. The data analysis could generate predicted chemical structures, which are then stored in a structure library. Structure libraries can also be used by data analysis programs to compute the relationship between measurement data and the structures in the database. The same is true for sequence libraries. The data analysis application could also simply store the entire result in a data repository for later use or extract specific spectra from the results to create a spectral library. Again, all of these data types can be both outputs from some types of analysis and used as inputs in other analysis. Once a spectral library is created, for example, it can be searched for matches between spectra obtained in a measurement and previous measurements. Searching spectral libraries will be covered in detail in Chapter 7.

In this chapter, I will focus on data analysis programs written in R, but it is important to keep in mind that these programs might work in conjunction with other data analysis programs that have created other types of output. For example, a program may have taken information from a structure database and computed simulated spectra under a variety of possible instrument conditions. Using R, you might write a program that reads the pre-mass spectrometry data to determine which conditions were used to collect a dataset

and then compares your data to a subset of those theoretical spectra with the goal of producing a list of possible structure matches. The reason programming in R for mass spectrometry is so powerful is that the possibilities for data analysis are nearly endless.

3.3.1 Investigation Metadata

The data associated with the pre-mass spectrometry aspects of an investigation is the least standardized of all the types of data used in mass spectrometry. Data analysis is usually focused on the output of a measurement, and whatever information *about* the sample that is captured is normally in those files. However, one of the prerequisites of reproducible research is to have as complete a description of the investigation's goals and methods as possible. The specific expectations of the data analysis need to be stated *prior* to the investigation, which includes the specific studies that comprise the investigation and the assays that were used to perform the studies. The expectations of the investigation are easily stated in natural language. However, it is often the case that the description of specific studies (what samples were used and for what purpose) and the details of assays (everything from sample preparation to the specifics of the apparatus used in measurement) need to be included in the data analysis process and ultimately need to be stored in data structures rather than human language text. In mass spectrometry, this information is the context for the raw mass spectrometry data and an indication of what is expected from the measurements produced. For many years, various work has been done to model and create data structures to hold detailed study and assay information when mass spectrometry is used for large and small molecule studies. The Human Proteome Organization (HUPO) created the Proteomics Standards Initiative (HUPO-PSI), which has published a number of minimum reporting requirements for

describing proteomics experiments, “the minimum information about a proteomics experiment (MIAPE)” [55] which includes extensions for different aspects including mass spectrometry, informatics, and various sample preparation and separation techniques [56–60]. These reporting guidelines led to attempts to create specific data formats to implement the requirement. The community has been highly successful at making raw mass spectrometry data accessible [61]. For other aspects of mass spectrometry-based assays, specifications such as gelML [62] and spML, mentioned by Deutsch [54], have turned out to be difficult for the community to implement.

Since fine-grained models have not been widely adopted, the mass spectrometry community has turned to higher-level models to describe investigations. These are machine readable, and use a combination of controlled vocabularies and natural language to describe investigations. The most active project for describing investigations is organized by the ISA Commons [63]. ISA stands for Investigation-study-assay, and its goal is to promote a framework for interoperable data within the bioscience community [64].

The central component of the ISA framework is the ISA abstract model [65] which defines the data elements of the model and their relationships. The data model has been implemented as a tab-separated-value (TSV) format [66] and using the JSON format [67, 68]. There are also programming language libraries for R and other languages [69, 70].

3.3.1.1 Using the ISA Model in R

For this example, I will use a dataset from the Metabolites data repository [71], which uses the ISA specification for storing metadata and data from publications. I will show how to read the metadata from the investigation `MTBLS1572` published in “Comparison of Full-Scan, Data-Dependent, and

Data-Independent Acquisition Modes in Liquid Chromatography-Mass Spectrometry Based Untargeted Metabolomics." by Jian Guo and Tao Huan in *Analytical Chemistry* in 2020 [72].

There are a few ways to parse TSV and CSV files in R. CSV files are probably the most common file format in all of data science and R programming, but the selection of separator is arbitrary, and the *tab* character is also a good choice. Using the tab allows the comma to be used in values, which is especially useful when values contain natural human languages that use the comma.

Since the ISA specification can be represented as either TSV files or JSON files, I'll use the `Risa` package [73, 74] from Bioconductor, which can read both file formats. The example file chosen here is a TSV. To read JSON files using `Risa`, refer to the documentation and publications on the package.

I'll use the `MTBLS1572` ISA files to show how to get a single table that includes all of the important factors for using the raw data to confirm various claims in the original publication.

Using `Risa` to load the dataset is straightforward. The `readISAtab()` function simply needs the path containing the ISA files. As shown in Table 3.1, there are three types of files in the directory. The *investigation* file lists all of the *study* and *study assay* files that are expected. The `readISAtab()` function parses the investigation file, then parses each study file and finally each assay file and populates a single S4 object of type `ISATab`.

TABLE 3.1

Overall investigation description.

Data type	Name	Syntax	Usage
Investigation description	ISA	TSV/JSON	Description of a collection of studies
Study description	ISA	TSV/JSON	Description of a study
Assay description	ISA	TSV/JSON	Description of an assay

```
  library(Risa)

  investigation <-
  readISAtab(file.path("data","MTBLS1572"))

  slotNames(investigation)
```

```
##  [1] "path"
"investigation.filename"
##  [3] "investigation.file"
"investigation.identifier"
##  [5] "study.identifiers"           "study.titles"
##  [7] "study.descriptions"          "study.contacts"
##  [9] "study.contacts.affiliations" "study.filenames"
## [11] "study.files"                 "assay.filenames"
## [13] "assay.filenames.per.study"   "assay.files"
## [15] "assay.files.per.study"       "assay.names"
## [17] "assay.technology.types"
"assay.measurement.types"
## [19] "data.filenames"              "samples"
## [21] "samples.per.study"
"samples.per.assay.filename"
## [23] "assay.filenames.per.sample"
"sample.to.rawdatafile"
## [25] "sample.to.assayname"
"rawdatafile.to.sample"
## [27] "assayname.to.sample"         "factors"
```

```
## [29] "treatments"                    "groups"
## [31] "assay.tabs"
```

Since `ISATab` is an S4 object, class variables are stored in *slots*, and the names are accessible from the `slotNames()` function. For S3, object variables are accessed by name using the `$` symbol. For S4 objects, the `@` symbol is used.

A basic piece of information is "What is this investigation about?" That is stored in the `study.titles` S4 slot. In ISA, there can be more than one study in an investigation:

```
  investigation@study.titles
```

```
## [1] "Comparison of Full-Scan, Data-Dependent, and Data-
Independent Acquisition Modes
in Liquid Chromatography-Mass Spectrometry Based
Untargeted Metabolomics."
```

The longer description is in the `studies.descriptions` slot.

```
  investigation@study.descriptions
```

```
## [1] "Full-scan, data-dependent acquisition (DDA), and
data-independent acquisition (DIA)
are the three common data acquisition modes in high
resolution mass spectrometry-based
untargeted metabolomics. It is an important yet underrated
research topic on which
acquisition mode is more suitable for a given untargeted
metabolomics application.
In this work, we compared the three data acquisition
techniques using a standard mixture of
134 endogenous metabolites and a human urine sample. Both
hydrophilic interaction and
reversed-phase liquid chromatographic separation along
with positive and negative
ionization modes were tested. Both the standard mixture
and urine samples generated
```

```
consistent results. Full-scan mode is able to capture the
largest number of metabolic
features, followed by DIA and DDA (53.7% and 64.8%
respective features fewer on average
in urine than full-scan). Comparing the MS2 spectra in DIA
and DDA, spectra quality is
higher in DDA with average dot product score 83.1% higher
than DIA in Urine(H), and the
number of MS2 spectra (spectra quantity) is larger in DIA
(on average 97.8% more than DDA
in urine). Moreover, a comparison of relative standard
deviation distribution between modes
shows consistency in the quantitative precision, with the
exception of DDA showing a minor
disadvantage (on average 19.8% and 26.8% fewer features in
urine with RSD < 5% than full-scan
and DIA). In terms of data preprocessing convenience,
full-scan and DDA data can be processed
by well-established software. In contrast, several
bioinformatic issues remain to be
addressed in processing DIA data and the development of
more effective computational programs
is highly demanded."
```

Later, this string can be incorporated into any markdown document you might make, but in the meantime, it gives you enough detail to determine if the data collected will serve your analysis purpose. It is human-readable, and the specifics are not in any other field.

What can be determined from the description is that there are three data acquisition modes that were performed on both a standard containing 134 known compounds and a human urine sample. I can also tell that two chromatography methods were used and that both positive and negative ionization modes were used. One claim that can be tested is the consistency between the standard mixture and the human sample, and one area of investigation suggested is to improve the processing of data-independent acquisition (DIA) assays.

For purposes of this chapter, the goal will be to wrangle the raw data associated with this investigation into data structures which simplifies testing the stated observations and working on improving DIA processing.

The `readISAtab()` function parses all of the study and assay file content into data frames, and it is helpful first to look at what information is available from the individual assays that were performed.

```
met_assays <- investigation@assay.files
length(met_assays)
```

```
## [1] 6
```

```
names(met_assays[[1]]) |> head(12)
```

```
##  [1] "Sample Name"
##  [2] "Protocol REF"
##  [3] "Parameter Value[Post Extraction]"
##  [4] "Parameter Value[Derivatization]"
##  [5] "Extract Name"
##  [6] "Protocol REF"
##  [7] "Parameter Value[Chromatography Instrument]"
##  [8] "Term Source REF"
##  [9] "Term Accession Number"
## [10] "Parameter Value[Autosampler model]"
## [11] "Term Source REF"
## [12] "Term Accession Number"
```

From this sample of the names of the columns in the data tables extracted by `Risa` from ISA files, it's clear that the assay tables are designed for human consumption, like the `investigation@study.descriptions` variable. The use of duplicate column names will make tidy table operations impossible since a program won't be able to tell which column is being specified. Attempting to convert this `data.frame` to a `tibble` will produce an error unless a

`.name_repair` function is specified. The `universal` repair renames columns so that they are both unique and have no characters that prevent them from being valid variable names:

```
met_tbl <- as_tibble(met_assays[[1]], .name_repair =
"universal")
```

```
names(met_tbl)[1:3]
```

```
## [1] "Sample.Name"
"Protocol.REF...2"
## [3] "Parameter.Value.Post.Extraction."
```

The syntactic names provided by `.name_repair="universal"` work, but are not very readable. Below, I show how to use a custom function for repairing the names that improves readability. But first, why are there duplicate names in the first place?

In the ISA format, the duplicate names are associated with the use of a controlled vocabulary. Since any code you write will have to take the specifics of the variables into account in the code, you will have to evaluate these manually and incorporate those details into your program. A good example is when a sample is diluted; the units of concentration need to be known if the actual concentration factors into a calculation. For the example at hand, there are no variables that require units, except for `Parameter Value[Scan m/z range]` which on inspection takes the form of `start-end`, and the name `m/z` indicates the units are the standard *Dalton/charge*. The variable `Parameter Value[Ion source]` comes from the controlled vocabulary for mass spectrometry http://purl.obolibrary.org/obo/MS_1000073 which can be found at the OBO foundry [75]. The vocabulary term is formatted using the Internationalized Resource Identifier

(IRI) standard [76],which means the actual control vocabulary name is `ms` (`https://obofoundry.org/ontology/ms.html`) and the specific term has an accession number `1000073` in the `psi-ms.obo`.

```
[Term]
id: MS:1000073
name: electrospray ionization
def: "A process in which ionized species in the gas
phase are produced from an
analyte-containing solution via highly charged fine
droplets, by means of
spraying the solution from a narrow-bore needle tip at
atmospheric pressure in
the presence of a high electric field. When a
pressurized gas is used to aid in
the formation of a stable spray, the term pneumatically
assisted electrospray
ionization is used. The term ion spray is not
recommended." [PSI:MS]
synonym: "ESI" EXACT []
is_a: MS:1000008 ! ionization type
```

While the default name repair is serviceable, column names can be fixed using a custom function that transforms the name strings into the format of column names you want. There are several ways to do this. For this example, I want to replace the spaces with _, remove the `/` character, and extract the names from within the `[]` symbols for `Parameter Value[]`. I'll call the function `fix_names()`.

```
fix_names <- function(x) {

    # use a regular expression to replace spaces with
"_"
    # then replace the "/" with an empty string
    # finally extract a string from between "[]" in a
name
    # if no "[]" characters exist, the str_match()
return will be NA

    name <- gsub("\\s+", "_", x)
    name <- gsub("\\/", "", name)
    ex_name <- str_match(name, "\\[(.+[^\\]])")[,2]

    # use the extracted name list and replace any non "
[]" entries
    # which have the value NA with items from the name
string

    for(i in 1:length(ex_name)){
        if(is.na(ex_name[i])) {
            ex_name[i] <- name[i]
        }
    }

    return(ex_name)
}
```

Calling `fix_names()` on column names will correct the names according to my specific rules:

```
print(fix_names("Parameter Value[Scan m/z range]"))
```

```
## [1] "Scan_mz_range"
```

```
print(fix_names("Sample Name"))
```

```
## [1] "Sample_Name"
```

Now I am ready to use `tibble()` with all of the data collected in this investigation into a table using my custom name repair function.

```
# list the duplicate names to be removed
dup_names <- c("Protocol REF","Term Source REF","Term
Accession Number")

# start with an empty list
assay_list <- list()

# for each assay in the investigation, remove all the
duplicate columns
# and append it to the empty list
for(assay in investigation@assay.files) {
    assay <- assay[,!(names(assay) %in% dup_names) ]
    assay_list <- append(assay_list, list(assay))
}

# convert the list of data frames to a tibble using
bind_rows()
# and repair the column names using the custom
fix_names() function
assay_tbl <- bind_rows(assay_list) |>
    tibble(.name_repair = fix_names)
```

```
assay_tbl |>
    names() |>
    head(5)
```

```
## [1] "Sample_Name"                "Post_Extraction"
## [3] "Derivatization"             "Extract_Name"
## [5] "Chromatography_Instrument"
```

Later I can use the `Derived_Spectral_Data_File` to read the raw data collected in the study and use it for visualization and computation, and use the other columns to compare data collected under the same conditions specified by the other variables (`Column_type` and `Scan_polarity`, etc.).

3.3.2 Instrument Methods

Currently, there is no standard data model for storing the specifics of pre-mass spectrometry data. In single-level MS measurements, most of the instrument method information is available in the raw data file. In investigations that use multiple-stage mass spectrometry (MS/MS), it is helpful to break out the selected reaction monitoring (SRM) transitions (mz1–mz2), so they can be exchanged in a vendor-independent way. Two formats are commonly used for storing and exchanging transitions: `sky` (skyline document) and `TraML` (Transitions Markup Language) shown in Table 3.2.

TABLE 3.2

Instrument method data.

Data type	Name	Syntax	Usage
SRM transitions	sky	XML	Transitions monitored in MS analysis
	TraML	XML	Transitions monitored in MS analysis

Both the skyline `sky` file and the HUPO-PSI `TraML` file use the XML format. XML is used for a variety of mass spectrometry-related data exchange tasks, and in this section, I will describe the basic task of reading an XML file directly. It is not always required to read the XML file format directly in R, when there is a standard or *de facto* standard, there are probably already packages available to read the XML file into R. I will have more to say about this in Section 3.5.1.

3.3.3 Example of Reading Skyline Main Documents

In this example, I will extract data from a Skyline main document into several tibbles, each containing a unique identifier, allowing the tables to be joined and queried using the tidyverse functions. Since one of the key features of Skyline is to associate SRM transitions with peptide and protein analysis, the goal will be to create tables that hold the protein information, which is the same for all peptides found; another to hold the peptide information, which again is the same for all SRM transitions used; and a third for the SRM data. Along the way, I will point out the difference between XML attributes and XML elements and give some pointers for dealing with the specifics of Skyline data.

Before reading an XML file directly, it is helpful to know the schema. XML has a schema description language called XSD that can be used to understand the details of a specific XML format, as well as to validate that a file using that schema follows correct usage.

Figure 3.2 is a schematic of the Skyline main document XML schema. You can see that a Skyline document begins with a tag called `srm_setting`, which has two *required* attributes: `format_version` and `software_version`.

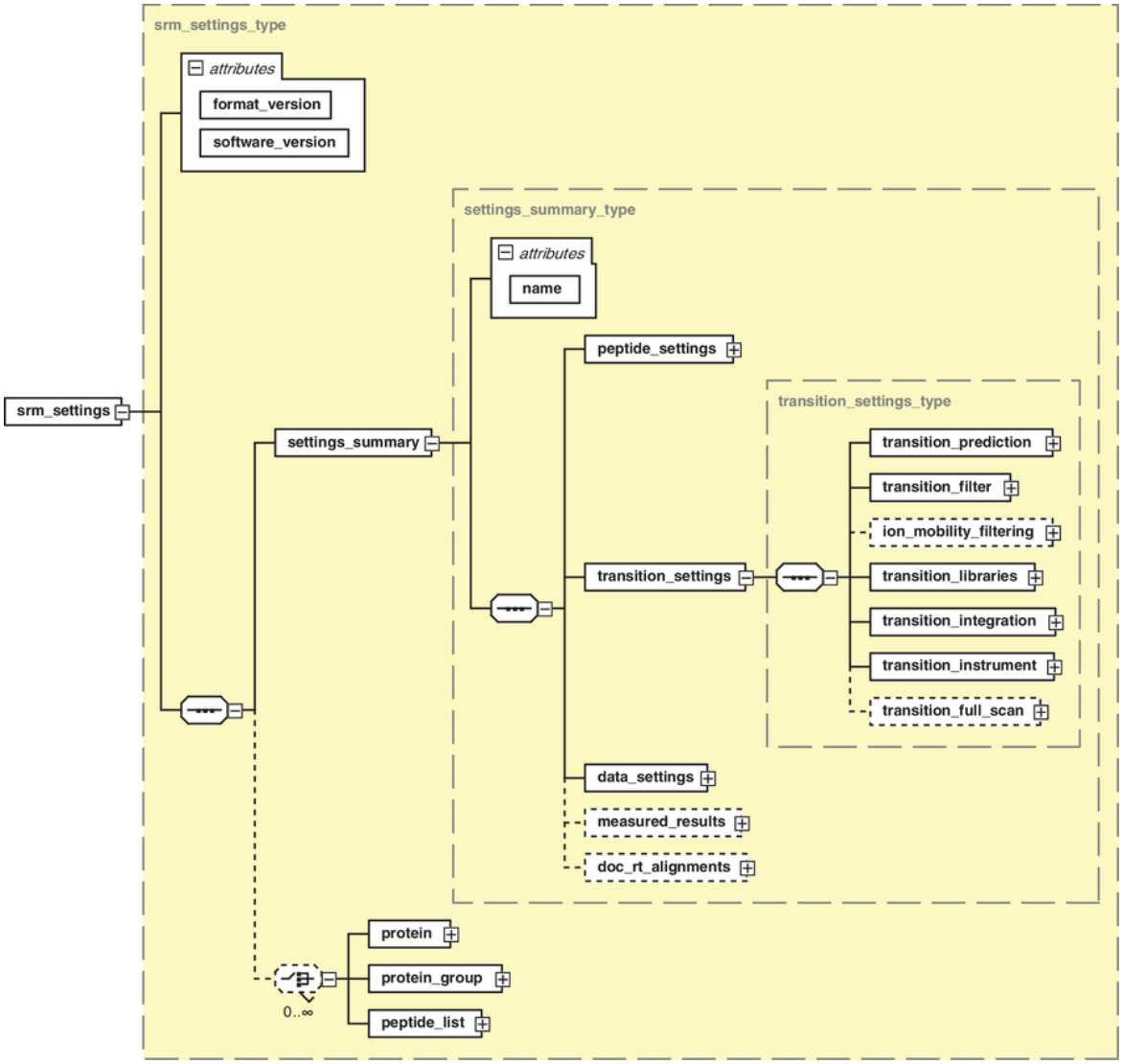


Figure 3.2 XML schema describing the Skyline main document. Required components are drawn in solid boxes while optional components are drawn in dashed-line boxes.

In plain text, like previous XML examples, it looks like this:

```
<srm_settings format_version="3.73"
              software_version="Skyline-daily (64-bit)
4.0.9.11635"
>
```

For this example, I will use data from Panorama Public [77] which is a repository for sharing targeted quantitative results contained in Skyline documents. The specific investigation I will use as an example is called "gRED – Automated QC of Targeted MS Data" [78] which is the deposited data from a paper published in 2018 by Toghi Eshghi et al. in *Clinical Proteomics*, called "Quality assessment and interference detection in targeted mass spectrometry data using machine learning" [79].

Unlike the approach used in Section 2.1.1, there is no equivalent package for reading Skyline files into R. In this situation, you can use the XML schema file to find the names and locations of data you want, and then parse (i.e., read) the XML file directly and then extract the elements you need for your analysis.

There are two main packages for parsing XML files in R: `XML` [80] and `xml2` [81]. Bioconductor packages like `MSnbase` use `XML`, while `xml2` is used by `tidyverse`. In this book, when reading XML files directly, I will use `xml2` since the goal is to wrangle data into the `tidyverse`. If you have a reason (such as contributing to Bioconductor) to use `XML`, refer to the package documentation for help.

The first line of the `.sky` file makes it clear that Skyline XML files are open vendor files with many versions similar to the Microsoft `.xlsx` format described earlier. With XML, the process is usually done in three steps: the file is parsed, validated, and then data is extracted using a query. Parsing ensures that the file is a properly formed XML document, while validation checks to make sure that the XML file follows a specific schema. For the example file shown above, the schema version is given as `3.73`. No schema location is given in the file, which means that if you want to validate the file, you will have to locate and download the correct version of the Skyline schema file from the Skyline project website. In the chapter on reproducible research, I will give more details

on how to get access to open-source project code, but for those already familiar with Skyline, the schema is maintained on GitHub in the ProteoWizard/pwiz repository [82] within the `pwiz_tools` directory where the Skyline code is stored [83]. Skyline schema files follow a naming convention: `Skyline_version.xsd`.

```
library(xml2)

csf_art_mat <- read_xml(file.path("data","gRED",

"CSF_Biomarkers_Artificial_Matrix.sky"))


sky_schema <-
read_xml(file.path("schema","Skyline_3.73.xsd"))

xml_validate(csf_art_mat, sky_schema)
```

```
## [1] TRUE
## attr(,"errors")
## character(0)
```

The output of the validation step is `TRUE`, which means that the Skyline document file follows the specifications given in the `3.73` schema. This validation step should be used when using XML files with Bioconductor library functions, not just when parsing manually. Most of the library functions that handle XML files will expect the file to meet at least some level of the specification, and some will generate errors if the file is incomplete or not valid. Depending on the library used, the error messages generated from unexpected XML input can range from simple validation failure messages to cryptic messages or even crashing the R session.

The `read_xml()` function returns the entire XML document. Both the data and the schema are XML documents and are read using the same method.

The “artificial matrix” file has been read into `art_mat`, but it is not in a very usable state yet. The XML document returned by `read_xml()` is a list of lists where the items in the lists are `xml_node` objects. The approach I recommend for navigating and extracting information from XML is called XPath, and it is the query language used by `xml2`. There are several functions in `xml2` to locate and return nodes in an XML document.

First, I want to create a tibble containing all of the information about the proteins described in the Skyline document. Looking at the text of the schema of the XML file, you can see that this document contains several attributes.

Looking at the raw XML text, you can see that the attributes of the `<protein>` element match the schema:

```
<protein
    name="P68082|MYG_HORSE"
    description="Myoglobin - Equus caballus (Horse)."
    accession="P68082"
    gene="MB"
    species="Equus caballus (Horse)"
    preferred_name="MYG_HORSE"
    websearch_status="X#UP68082"
    auto_manage_children="false">...
</protein>
```

The first protein described in the document shows that the accession number is stored as an attribute of the `<protein>` element. Figure 3.3 shows that except for the attribute `name`, almost all of the other attributes of a protein are *optional* (dashed-line boxes). That means in any particular document the only requirements are the inclusion of an attribute called `name` and a single element called `sequence`.

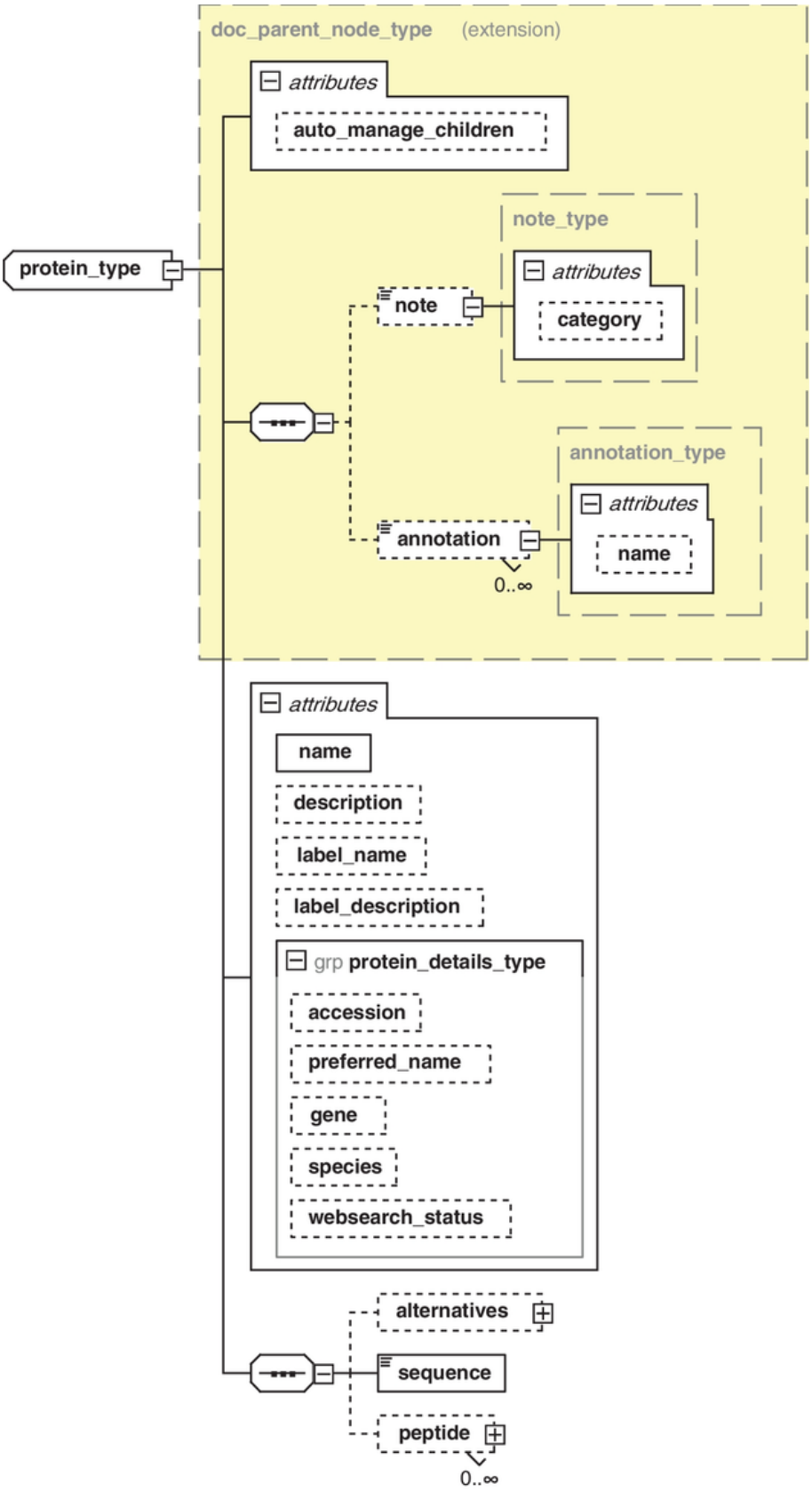

doc_parent_node_type (extension)
attributes
auto_manage_children
note_type
attributes
category
protein_type
note
annotation_type
attributes
name
annotation
0..∞
attributes
name
description
label_name
label_description
grp protein_details_type
accession
preferred_name
gene
species
websearch_status
alternatives
sequence
peptide
0..∞

Figure 3.3 XML schema describing the Skyline protein element.

My goal is to extract all of the proteins into a tibble, with each attribute being represented in a column. Now I know some will be present and some might be missing, even in a data file that is valid according to the schema.

Part of the `xml2` package includes functions that perform queries using the XPath language [84]. XPath is a powerful and complex language, and for this example, I'm only going to use a small part of what's available. In `xml2`, the function `xml_find_all()` takes as input, the XML document parsed by the `read_xml()` function and a string representing an XPath query. It will return an `xml_nodeset` for all the nodes that match the query string. For my purposes, I only need the `//node` pattern and the name of the nodes I want to match. The `//` prefix says to match all the descendants of the root node. In this case, there are multiple instances of `<protein>` that are descendants of `<srm_settings>`.

```
prot <- xml_find_all(csf_art_mat, "//protein")
```

Now the `prot` object is an `xml_nodeset` containing all of the `<protein>` nodes. The attributes of a node can be accessed using the `xml_attrs()` function. The only required child node of `<protein>` is `<sequence>`, and I'd like to include the sequence in the table along with the other metadata. Since `<sequence>` is an element, its contents are accessed using the `xml_text()` function.

Protein sequences are represented as a string with white space, newlines, and, in the case of Skyline files, some extra spaces that need to be removed. So before creating the table, it's worth it to clean up the sequence strings as much as possible.

```
  seq <- xml_find_all(csf_art_mat, "//sequence") |>
      xml_text() |>
      str_trim() |>
      str_replace_all("\\n","") |>
      str_replace_all("        ", " ")
```

Now a tibble can be created that combines the protein attributes, unnested into columns, combined with the sequence string.

```
  prot_attrs <- tibble(attrs = xml_attrs(prot), seq=seq)
  |>
      unnest_wider(attrs)

  prot_attrs |>
      head(5)
```

```
## # A tibble: 5 x 9
##   name       description accession gene  species
preferred_name websearch_status
##   <chr>      <chr>       <chr>     <chr> <chr>   <chr>
<chr>
## 1 P68082|MY~ Myoglobin ~ P68082    MB    Equus ~
MYG_HORSE      X#UP68082
## 2 sp|P02768~ Serum albu~ P02768    ALB   Homo s~
ALBU_HUMAN     X
## 3 sp|P05067~ Amyloid be~ P05067    APP   Homo s~
A4_HUMAN       X
## 4 sp|P02649~ Apolipopro~ P02649    APOE  Homo s~
APOE_HUMAN     X
## 5 sp|P36222~ Chitinase-~ P36222    CHI3~ Homo s~
CH3L1_HUMAN    X
## # i 2 more variables: auto_manage_children <chr>, seq
<chr>
```

In this code chunk, the `attrs` column of the `prot_attrs` table is a list of *named* character vectors. Since the names are the same for all the character vectors returned by `xml_attrs()`,

then the `unnest_wider()` function turns each character vector name into a new column name.

When reading XML files, you will usually have to deal with elements and attributes that are *optional*. In the schema for Skyline, the only required attribute is `name`. If the optional `accession` attribute is missing in any of the nodes, the column will be created, but the entry for that row will be `NA`. Looking at the schema for Skyline, it appears that **most** elements and attributes are optional. That means that if you want an entry for any missing elements or attributes, you have to decide on a way to replace them or declare them `NA`.

In this example, I'd like to use the `accession` value as a unique identifier that can be used to join tables together. So while the file we've read so far seems to have an entry for `accession` for every protein, you cannot be sure that will always be true for every file you read. A block of code that creates an `accession` value that cannot be in UniProt can be used if you want to use `accession` as a key for joins.

```
# accession is an optional attribute will become NA if
missing
# replace any NA values unique value that does not
match the UNIPROT
# specification, but can be used as a key for join()

for(i in 1:length(prot_attrs$accession)) {
    if(is.na(prot_attrs$accession[i])) {
        prot_attrs$accession[i] = sprintf("ZYZZY%04d",
i)
    }
}
```

Some might consider using the `for` statement to loop in R non-idiomatic. Perhaps it is, since you should try and vectorize everything you can in R. However, for a simple task like making up a value for `accession`, the for-loop is more understandable and faster than some might think. When you

are writing code that uses another language like XPath or complex formatting, simple flow control syntax is easier to understand for others and your future self.

It is up to you how robust you want to make handling the optional elements for a particular file. The schema makes it clear which elements and attributes you can count on being there if you validate the file against its schema, but some attributes and elements, being optional, maybe in one specific file and missing from another, and both are valid. Also remember that the order of attributes and elements is not ensured by the schema, so you can't count on them being in any specific order when you read them.

Up to this point, I've created lists of XML nodesets by using only the `//node` XPath expression. XPath is a rich query language, and for reading some XML files, you may have to take advantage of some of its advanced features. For this example, I'm taking the approach of minimizing the amount of non-R programming you have to do and working with slightly messier input to create tidy data. It is possible, even likely, that with all types of data, advanced techniques in XPath, SQL, or some other query language could reduce the amount of R programming you have to do. For most XML files, and in mass spectrometry, the basic approach of this book is to *depend mostly on R, not query language functionality*. The only parts of the XPath expression language I will use are shown in Table 3.3. You may find that you have to dig a little deeper into the query language you use for some types of files or data sources, but in general, it will make your code more reusable and more resilient if you let R do most of the work using only the necessary features of a secondary language, even if that leads to more lines of R code.

TABLE 3.3

Some basic XPath expression syntax.

Expression	Description	Example
//node	Any descendants that match node name	//protein
/node	Matches direct descentants of node name	//protein/peptide
@	Selects an attribute of a node	//protein/@name
[]	Selector constraint. Matches node that meet conditions	//protein[@species="Homo sapiens"]
|	Union operator. Join results	//@accession | //@sequence

```
  pattern <- paste(
      "//protein/@name",
      "| //protein/@accession",
      "| //peptide/@sequence",
      "| //peptide/precursor/@charge",
      "| //peptide/precursor/@isotope_label",
      "| //peptide/precursor/transition/@fragment_type",
      "| //peptide/precursor/transition/precursor_mz",
      "| //peptide/precursor/transition/product_mz")

  srm_list <- xml_find_all(csf_art_mat, pattern)
```

The first step is to build a query string that will match the data elements and attributes in the XML file that I want to extract. The `xml_find_all()` function will return a list of

nodes that match an XPath pattern. The XPath pattern is constructed to return all the elements and attributes specified in the order they are found. The first item in the node list will be the first `name` attribute from the first `protein` node, followed by optional attributes: the accession number, the peptide sequence, the peptide precursor by the fragment type, and then the precursor and product m/z values. The `xml_find_all()` function finds all the matches for these patterns, joins them in a list, adds them to the result list, and then moves to the next node until it hits the end of the file. The result will be a long list of nodes with the `accession` attribute beginning each section. Once this list is created in the `srm` object, it can be parsed out into a `tibble`. There are several ways to iterate over a list like this in R. For this first example, I will use a simple `repeat` loop to iterate over the node list created by the query of `csf_art_mat`. Using loops like `for`, `while` and `repeat` used to be considered slow in R, but over the years, they have been so optimized that they can be used without much performance loss. As mentioned regarding the for-loop, you should use the vectorized functionality of R if you can. Vectorized approaches are usually faster and are sometimes easier to read, but they add complexity to your programming by hiding the looping process. If you want your code to have maximum readability, sometimes standard control structures are the best way to create loops.

Next, I will loop through the node list, loading each value into a variable that will be assigned to a value in a row of a table. The goal is to end up with a table that has every transition that is described in the file in a row. That will mean repeating some column values if there is more than one transition per protein. Each section of the list begins with the protein name, which is a required value. That means when a new name is found, you know that you are parsing a new protein section.

To start, create the `tibble` that will be populated in the loop.

```
srm <- tibble(
        accession = character(),
        sequence = character(),
        precursor_charge = numeric(),
        isotope_label = character(),
        fragment_type = character(),
        precursor_mz = numeric(),
        product_mz = numeric()
)
```

Next loop through the `srm_list` and extract each SRM description.

```
i = 1
repeat {
    if(i > length(srm_list)) {
        break
    }

    node = srm_list[i]

    # Set default values for optional attributes and
elements
    # Here only the accession and isotope_label
attributes are handled
    # Expand this section if new files have missing
attributes/elements

    if(xml_name(node) == "name") {
        accession =
prot_attrs$accession[prot_attrs$name == xml_text(node)]
        isotope_label = "light"
        i = i + 1
        next
    }

    row_done = FALSE
    while(xml_name(node) != "name" ) {
        if(xml_name(node) == "accession") {
            accession = xml_text(node)
        } else if(xml_name(node) == "sequence") {
            sequence = xml_text(node)
        } else if (xml_name(node) == "charge") {
            precursor_charge = xml_integer(node)
        } else if (xml_name(node) == "isotope_label") {
            isotope_label = xml_text(node)
        } else if (xml_name(node) == "fragment_type") {
            fragment_type = xml_text(node)
        } else if (xml_name(node) == "precursor_mz") {
            precursor_mz = xml_double(node)
        } else if (xml_name(node) == "product_mz") {
            product_mz = xml_double(node)
            row_done = TRUE
        }
```

```
        if(row_done == TRUE) {
            srm <- add_row(srm,
                           accession = accession,
                           sequence = sequence,
                           precursor_charge =
precursor_charge,
                           isotope_label = isotope_label,
                           fragment_type = fragment_type,
                           precursor_mz = precursor_mz,
                           product_mz = product_mz
            )
            row_done = FALSE
            isotope_label = "light"
        }

        i = i + 1
        if(i > length(srm_list)) {
            break
        }

        node = srm_list[i]
        next
    }
}
```

In addition to the `accession` attribute, the `isotope_label` is also optional. In the case of `accession`, the missing value was replaced based on my wanting to use `accession` as a key in joining tables. The `isotope_label` attribute is also optional, but in the context of Skyline, if it is missing, the default value is `light`, and the value was assigned to the default and only reassigned if the attribute was present in the file. Again, it is up to you to decide how robustly you would like to handle missing values. When you know the default values or know you want to use a column in a certain way, you can use a range of methods for dealing with missing values.

```
str(srm)
```

```
## tibble [333 x 7] (S3: tbl_df/tbl/data.frame)
##  $ accession       : chr [1:333] "P68082" "P68082"
"P68082" "P68082" ...
##  $ sequence        : chr [1:333] "HGTVVLTALGGILK"
"HGTVVLTALGGILK" "HGTVVLTALGGILK"
                                          "HGTVVLTALGGILK"
...
##  $ precursor_charge: num [1:333] 2 2 2 2 2 2 2 2 3 3
...
##  $ isotope_label   : chr [1:333] "light" "light"
"light" "light" ...
##  $ fragment_type   : chr [1:333] "y" "y" "y" "b" ...
##  $ precursor_mz    : num [1:333] 690 690 690 690 693
...
##  $ product_mz      : num [1:333] 985 886 772 494 992
...
```

3.4 Result Data

Result data from a mass spectrometry experiment can be either qualitative or quantitative. In the Skyline example above, I've shown how to extract qualitative results (transition information on proteins and peptides found in a sample) from an XML file. Most result data can be read either as a text file with known separators or as an XML file using an XML library and XPath (Table 3.4).

TABLE 3.4

MS experimental result file types.

Data type	Name	Syntax	Usage
MS results	mzIdentML	XML	HUPO molecular identification
	mzTab	TSV	HUPO generalized result format
Metabolite ID	MAF	TSV	metaboBank metabolite assignment file

Results can also be stored in open standard formats, for which packages are available for reading that specific format. For protein and peptide *search* (identification) results, the HUPO-PSI mzIdentML XML format is available for many investigations and deposited with data files supporting publications. As mentioned earlier, there are several Bioconductor packages that can be used to parse standardized XML files. Depending on your goal, mzIdentML files can be read with several packages. In the next example,I will use the `PSMatch` package to read mzid files. Another common result format is the HUPO-PSI format called mzTab. Files in the mzTab format can be read with functions in `MSnbase` introduced earlier or directly as a TSV file. For metabolite identification, the metabolite assignment file (MAF) format [85] from the MetaboBank repository [86] is the recommended format [87]. The MAF file also uses the TSV format, which can be read directly using the `dplyr` function `read_tsv()`.

To show how to read result data as well as access raw data formats, I will use the data deposited from the publication “Unbiased proteomics, histochemistry, and mitochondrial DNA copy number reveal better mitochondrial health in muscle of high-functioning octogenarians” by Ubaida-Mohien

et al. [88]. The authors of this paper did an outstanding job of depositing a very complete record of the data they used to perform data analysis. The data was deposited in the MassIVE repository with the identifier MSV000086195 and can be downloaded from https://massive.ucsd.edu/ProteoSAFe/dataset.jsp?accession=MSV000086195.

For the examples in this chapter, I created a directory, `MSV000086195`, under the `large-data` directory in the working directory for R and downloaded the contents from MassIVE. It is sometimes useful to separate very large files in their own directory, especially if you choose to use a source code repository to manage your analysis. Most source code repositories have file size limits, so data files under the limit can be included with the code files. Files above the size limit can be placed in a folder that is marked for exclusion from source code management. This doesn't detract from the reproducibility of your work if you are using a source code management system because the data is available in the data repository and can be accessed separately. One word of caution: many studies in MassIVE contain enormous amounts of data, especially when the vendor's raw data is included, as is the case in this example. For this reason, you may want to consider which parts of the deposited data you choose to download. For the examples in this chapter and the next, I did not download the vendor raw files, which represent hundreds of gigabytes of data.

3.4.1 Molecular Identification

In this example, I will explore using tidy methods to analyze a simple aspect of a proteomics experiment based on one aspect of the Ubaida-Mohien 2022 study. The primary goal of the investigation was to measure the differential expression of proteins extracted from the muscle cells of two groups (see the manuscript for details). The simplified description of the

method is that three batches of extracts were prepared, enzymatically digested, and then mass tag labeled to allow pooling and relative between-group protein quantification.

The authors used the tandem mass tag (TMT) 10-plex [89] kit for labeling. In the TMT approach, proteins from a single specimen are enzymatically digested and then labeled with a mass tag such that peptides from a specific specimen can be identified by reporter fragments in their MS2 spectra. In this experiment, the authors combined 10 different digested extracts into a single sample for analysis. Key to the TMT approach is that each of the 10 tags has the same monoisotopic mass when reacted with the digested peptides. Thus, they modify the molecular weight of a specific peptide by the same mass shift regardless of which specimen they came from. That means that the peptide can be selected in the MS1 spectrum and then fragmented. In the MS2 spectrum, each of the 10 different tags is fragmented along with peptide to produce a unique isotopically labeled *reporter ion* at a low m/z value (126.127726–131.144499). Since the labeled peptides from different specimens have the same molecular weight modification, they show up in the MS1 spectrum as a single peptide with a modified m/z value (+229.162932). This allows data-dependent acquisition to be used to select the single m/z value for fragmentation, producing an MS2 spectrum that can be used for peptide identification using a database search and which includes the signals for the reporter ions, indicating which specimen the peptide came from, with the added benefit that the reporter ion intensities indicate the relative quantity of the peptide for each of the 10 individual specimens in the pooled sample.

Because of the multiple stages of chromatography, multiple sample-handling steps, and the very long run time of the final chromatography step (195 minutes), there is the potential for variability in many parts of the experiment. To estimate the variability due to the pre-analytical and analytical steps, the authors chose to spike each digested extract with an internal

control. In this case, they added a predigested protein standard (bacterial beta-galactase [90]). The variation in the signals generated by the control peptides reflects the expected between-subject measurement variation and is not related to biological effect. Ideally, these signals should be constant across all samples. The intensity of each of these fragments should also be consistent across all of the internal control peptides. Internal controls are critically important for experimental processes as complex as those described in the Ubaida-Mohien 2022 manuscript and its predecessor [91].

I will start with the protein search step performed by the original research team, but this time, focusing only on the internal control protein. This will allow you to see how identification (qualitative) results and quantitative results from both MS1 and MS2 data can be wrangled into tidy structures and the various techniques that can be used with multimode analysis pipelines in a wide range of mass spectrometry applications.

In the method description for the study, the authors state that the MS2 data from their experiment was searched with the programs Mascot and X! Tandem against the SwissProt Human sequences from UniProt appended with 115 contaminants. Further, the peptides matching contaminants were removed from the results. A common source of potential contaminants is the common Repository of Adventitious Proteins (cRAP) database from the Global Proteomics Machine project [92]. This database contains 115 sequences and includes the bacterial beta-galactase internal control. With the peptides from beta-galactase removed, performing analysis on the internal control is not possible using the output provided in the repository. However, since the authors deposited the extracted MS2 spectra, the internal control peptides can be found using the beta-galactase sequence and the X! Tandem search engine configured as described in the manuscript. X! Tandem can be configured to produce mzIdentML files which include only the highest confidence

peptide-spectrum matches for the internal control, allowing qualitative and quantitative analysis of the control.

3.5 Example of Wrangling Data: Identification Data

There are several ways to read `mzid` files in various packages in Bioconductor. In this example, I will use the `PSMatch` library and the `PSM()` function. `PSM()` generates a specialized data structure of the class PSM from which identification information can be extracted into a tidy format.

The data acquisition for MSV000086195 was performed using a Q Exactive HF mass spectrometer (Thermo Scientific, San Jose, CA). As shown in Table 3.5, Thermo instruments generate a proprietary data file, which must first be converted to an open format before it can be used with R and other tools.

TABLE 3.5

Vendor raw data formats.

Name	Syntax	Usage
d/	proprietary	Data from Agilent instruments
raw/	proprietary	Data from Waters instruments
RAW	proprietary	Data from Thermo instruments
wiff + wiff.scan	proprietary	Data from Sciex instruments
t2d	proprietary	Data from Sciex TOF instruments
yep,baf,flex	proprietary	Data from Bruker TOF instruments

The protein used for this example is an internal control and is known to be present in every sample prior to labeling,

fractionation, chromatography, and the data-dependent acquisition (DDA) mass spectrometer data collection. Because the internal control was predigested, what can be observed is the batch-to-batch variability in labeling, fractionation, chromatography, and mass spectrometer sensitivity and performance of the DDA step.

The objective of this example is to use a peptide identification program to find the internal control peptides in the samples provided by the authors in the MassIVE repository. By setting the threshold for identification to an extreme value and using only the internal control protein sequence for the search, the resulting output of the search is extremely small compared to the results from searching a full proteome as was done in the publication.

The method for converting the Thermo vendor files to open formats for use in search and analysis tools was the popular open-source program *MSConvert* [13]. More details on how to convert various vendor formats to an open format are given in the Global Natural Products Social Network (GNPS) [93] documentation site [94]. Notice that most vendor data file types are supported by MSConvert. If your file type is not supported, you can contact the ProteoWizard team via their GitHub site [95] and submit an issue, or join the GNPS community [96] where more help is available.

Using MSConvert, Thermo raw data files can be converted to a variety of open file formats. Table 3.6 shows some of the useful open data formats in mass spectrometry.

TABLE 3.6

Open data formats for raw mass spectrometry data.

Name	Syntax	Usage
MGF	Text	Mascot generic format for MS2 peak lists
JCAMP-DX	Text	Text representation of MS data
MSP	Text	NIST spectrum/structure library format
ANDI-MS/netCDF	netCDF	Cross-platform binary format for MS data
mzML	XML	HUPO open format for raw MS data
mzXML	XML	ISB open format for raw MS data
imzML	XML + binary	Open binary format for imaging MS data
mz5	HDF5	Open binary format for raw MS data

3.5.1 Open Raw Data Formats

One of the oldest and most difficult challenges in developing programs for mass spectrometry data analysis is how to make the data acquired by an instrument accessible and complete. The formats proposed and used have followed the development of data representation of the computers that control analytical instrumentation.

Early computer systems tended to use a record-oriented approach to data storage, and that is reflected in the early mass spectrometry formats like JCAMP-DX [97, 98]. With the advent of networked computers, file formats had to be

transmissible over networks and readable by computers with different internal data representations. Formats like netCDF were developed to accomplish portability and allowed the creation of formats for mass spectrometry using a portable binary approach. With the development of the World Wide Web, the format of the HTML led to a more formalized version for data representation called the XML. XML is the most common open format, but being a text-based format means it has limitations when data sizes get large and when random access to elements is needed. The evolution from netCDF to HDF5 means that it is now possible to store very large amounts of data in a portable binary format, and at least one format for mass spectrometry has been created using this latest approach. All of the open formats have advantages and disadvantages, and many of the formats shown in Table 3.6 have R packages to support the reading and writing of these files. Depending on your application, you may have to read any of several file formats. If you are writing files, you can choose which formats best suit your needs or the needs of your audience.

For proteomics and other applications, the *.mgf* file format [99] is extremely useful. It is a simple text file that can be read line by line. Each spectrum starts with a `BEGIN IONS` record and ends with `END IONS`. Between these records, a set of embedded search parameters can be provided, which are followed by the peak list as shown below.

Example 3.1 MGF format for a spectrum

```
BEGIN IONS
TITLE=File:"ScltlMsclsMAvsCntr_Batch1_BRPhsFr29.raw"
RTINSECONDS=0.79676478
PEPMASS=752.448056103694 367438.5
CHARGE=2+
117.4452527 760.5017700195
117.447868 0.0
127.1198491 0.0
127.125235 8669.3916015625
127.1313907 2595.0625
127.1352261 0.0
128.1252025 0.0
128.1283167 1025.9385986328
128.1321975 0.0
...
1349.632791 0.0
1349.765885 3455.4172363281
1349.924943 0.0
1499.052612 0.0
1499.177249 842.9288330078
END IONS
...
BEGIN IONS
TITLE=File:"ScltlMsclsMAvsCntr_Batch1_BRPhsFr29.raw"
RTINSECONDS=14699.6562
PEPMASS=600.3286 8715.2294921875
CHARGE=4+
111.2126462 597.8501586914
111.2150581 0.0
119.9709253 0.0
119.973747 530.4479980469
119.9763741 0.0
...
907.7844305 0.0
907.8431639 804.5353393555
END IONS
```

Another text file format that is commonly found outside of the -omics ecosystem is JCAMP-DX [97, 98]. JCAMP-DX is also a plain text format used to store spectra data for many different types of instruments besides mass spectrometers, as shown below.

Example 3.2 JCAMP-DX format for a spectrum

```
##TITLE= 2-Chlorphenol
##JCAMP-DX= 5.00   $$ ISAS JCAMP-DX program (V.1.0)
##DATA TYPE= MASS SPECTRUM
##DATA CLASS= PEAKTABLE
##ORIGIN= H. Mayer, ISAS Dortmund
##OWNER= COPYRIGHT (C) 1993 by ISAS Dortmund, FRG
##SPECTROMETER/DATA SYSTEM= Finnigan MAT Magnum
##INSTRUMENTAL PARAMETERS= LOW RESOLUTION
##.SPECTROMETER TYPE= TRAP  $$ ion trap spectrometer
##.INLET= GC                $$ gas chromatograph as
inlet
##.IONIZATION MODE= EI+     $$ electron impact
ionization with positiv polarity
##.BASE PEAK= 128
##.BASE PEAK INTENSITY= 687729 COUNTS
##.RIC= 3063043
##XUNITS= M/Z
##YUNITS= RELATIVE ABUNDANCE
##NPOINTS= 26
##PEAK TABLE= (XY..XY)
50, 5.84
51, 9.55
...
130, 32.45
131, 2.13
##END=
```

The NIST *.msp* library format is designed so that each record in the files has chemical names, structures, and other chemical information, and an associated spectrum. The

objective is to allow an unknown spectrum to be compared to a known spectrum and, if it is a close match, to suggest the chemical identity. Alternatively, given a chemical structure or name, the observed spectra for that compound can be located.

3.6 Wrangling Multiple Data Sources

In the Ubaida-Mohien 2022 study, the manuscript states that MGF files were extracted from the raw file using MSConvert and deposited in MassIVE. Each MGF file represents an MS level 2 spectrum that was generated for purposes of peptide identification. Many peptide search programs, including X! Tandem, can read MGF files directly. Others can read a variety of formats, so the input format you need for a search may range from using the instrument vendor formats directly to an XML format or the simpler text formats list MGF. In addition to the peak lists (`.mgf`) used for peptide searching, all of the vendor data (`.raw`) and the open format raw data in the mzML format were deposited along with the identification data. In the next chapter, I will use this data to introduce some useful exploratory data analysis of the data in this study. To help with that exploration, I will create a data table that combines a subset of the result data and some information from the raw data to help answer some basic questions about the data set, specifically on the subject of the internal controls.

The approach taken for all tasks is to build a tidy representation of the application-specific data and use that to guide further analysis. In this case, I want to gather and tidy information on the peptides of the internal control that were found. Of primary interest are the search scores (the quality of the match) and information on both the MS1 and MS2 spectra that were used for the identification. I will construct a simpler table from the 30+ variables returned by the `PSM()`

function from the `PSMatch` package. To do this, the program will have to extract elements from the identification data (in the `.mzid` files) along with some elements from the raw data. The goal is to end up with a data table that can be used for exploratory data analysis in the next chapter.

To start the example, take a look at what information is available in the `.mzid` file using the `PSM()` function. Since this analysis uses both searches done by X! Tandem and by Mascot, I have set the PSM parser to `mzID` which is compatible with `.mzid` files generated by both search engines. Since the two work differently, they generate controlled vocabulary terms that are specific to each engine. When moving data from the application-specific output to the tidy format, a translation between the engine-specific terms is required. In the example, I have created two functions: `extract_tandem_match()` and `top_tandem_scans()`. In Chapter 5, I will create the Mascot-specific versions of these two functions for reading the results from the Mascot searches that were deposited on the repository.

```
library(PSMatch)

mzID_schema <-
read_xml(file.path("schema","mzIdentML1.1.0.xsd"))

result_file = file.path("large-data",
"MSV000086195","tandem_result",

"ScltlMsclsMAvsCntr_Batch1_BRPhsFr29.mzid")

if(!xml_validate(read_xml(result_file), mzID_schema)) {
    print(paste0("Invalid mzIdentML:", result_file))
    break
} else {
    all_psm <- PSM(result_file, parser = "mzID")
    id <- as_tibble(all_psm)
    names(id)
}
```

```
## reading ScltlMsclsMAvsCntr_Batch1_BRPhsFr29.mzid...
DONE!

##  [1] "spectrumid"                "scan.number.s."
##  [3] "acquisitionnum"            "passthreshold"
##  [5] "rank"
"calculatedmasstocharge"
##  [7] "experimentalmasstocharge" "chargestate"
##  [9] "x..tandem.expect"          "x..tandem.hyperscore"
## [11] "isdecoy"                   "post"
## [13] "pre"                       "end"
## [15] "start"                     "accession"
## [17] "length"                    "description"
## [19] "pepseq"                    "modified"
## [21] "modification"              "idFile"
## [23] "spectrumFile"              "databaseFile"
```

In this code, I first checked to make sure the `.mzid` file matched the mzIdentML schema and then read all of the ID information into a base `data.frame` using the `PSM()` function. One of the reasons for doing this is that the XML file being read may not be valid. It is possible that the files deposited follow a different schema than what is expected by the `PSM()` function. Since no information can be extracted from an invalid file, it's better to check first than to let `PSM()` fail. For example, I set the X! Tandem threshold expectation value very low and search for only one (control) protein, which can cause the program to create an empty match section in the `.mzid` file. For most peptide search applications, this would be a failure, and it might be an indication that some search setting is incorrect. In an analysis of internal controls, which are not expected to be in every fraction for every batch, it's a realistic outcome. Depending on the search program you use, different results might occur for the situation where no spectra meet the search criteria. Checking that the file exists and that it is valid based on a specific schema is a good precaution and suits our needs well. For every project, you will have to decide how to handle missing data and how

robust you need to make your program in case of problems with formats or file corruption.

If everything is in order with the identification file, the output of the `PSM()` function can be converted directly to a `tibble` since the `PSM` class extends the `DataFrame` class, which can be directly coerced to a tibble. To see what classes an application-specific class extends, the function `getClassDef()` from the `methods` package is helpful:

```
  getClassDef("PSM")
```

```
## Class "PSM" [package "PSMatch"]
##
## Slots:
##
## Name:            rownames             nrows
elementType   elementMetadata
## Class: character_OR_NULL           integer
character DataFrame_OR_NULL
##
## Name:            metadata          listData
## Class:                list              list
##
## Extends:
## Class "DFrame", directly
## Class "DataFrame", by class "DFrame", distance 2
## Class "SimpleList", by class "DFrame", distance 2
## Class "RectangularData", by class "DFrame", distance 3
## Class "List", by class "DFrame", distance 3
## Class "DataFrame_OR_NULL", by class "DFrame", distance
3
## Class "Vector", by class "DFrame", distance 4
## Class "list_OR_List", by class "DFrame", distance 4
## Class "ListorHits", by class "DFrame", distance 4
## Class "Annotated", by class "DFrame", distance 5
## Class "vector_OR_Vector", by class "DFrame", distance 5
```

I'd like to have a single tibble that contains only unique MS level 2 scans for each peptide found. So, rows will have

unique values for `scan.number.s.`. If multiple peptide matches occur for a single MS2 scan because of multiple modification locations, I will keep the one with the highest value of `x..tandem.hyperscore`. This lets me keep every MS2 spectrum that contained a high-confidence match but eliminate duplicate entries for the same spectrum. Several other variables will be useful in exploring the entire dataset. As was used in a previous example, I use the pattern of creating an empty tibble defining the names and types of the columns (variables), then append a selected tibble to the empty one to build up a final result. This is handy when using loops to iterate over multiple files. The empty tibble is created outside of the loop, and the results of the calculation or selection are appended inside the loop.

For this example, I've decided I want the batch number, the fraction number, and the retention time of the matching spectrum. These are not isolated in the output of `PSM()` but they are embedded in the `spectrumid` string.

```
  id$spectrumid[1]
```

```
## [1] "ScltlMsclsMAvsCntr_Batch1_BRPhsFr29.49412.49412.2
File:\"ScltlMsclsMAvsCntr_Batch1_BRPhsFr29.raw\",
NativeID:\"controllerType=0 controllerNumber=1
scan=49412\" RTINSECONDS=9195.8256 "
```

To extract the values desired, R has a regular expression matching function called `str_match()`, which takes an input string and a regular expression pattern and returns all the matches. Like XPath, regular expressions are a powerful tool, and I will only scratch the surface in this book. Regular expressions are so powerful that it is worth learning how to use them well. For that I can recommend *R for Data Science* [30, 31], and for a much deeper dive: *Mastering Regular Expressions* by Friedl [100].

The batch, fraction, and retention time of an identification can be extracted using the following patterns.

```
  str_match(id$spectrumid[1], "_Batch(\\d+)_")
```

```
##      [,1]      [,2]
## [1,] "_Batch1_" "1"
```

```
  str_match(id$spectrumid[1], "_BRPhsFr(\\d+)")
```

```
##      [,1]        [,2]
## [1,] "_BRPhsFr29" "29"
```

```
  str_match(id$spectrumid[1], "RTINSECONDS=([0-9.]+)")
```

```
##      [,1]                     [,2]
## [1,] "RTINSECONDS=9195.8256" "9195.8256"
```

The `str_match()` function returns a character matrix. The first column is the complete match, followed by a column for each *capture* group. The capture groups are placed in `()` in the pattern string. The batch number is found by looking in the string for "_Batch" followed by one or more digits. The batch number is captured with the pattern `\\d+`, which matches one or more digits. The same idea and capture pattern is used for the fraction number. Since the retention time is a positive floating point number, the capture pattern is `[0-9.]+`, which matches one or more of the characters inside the `[]` symbols. In this case `0-9` means any digit between 0 and 9, and the `.` matches the decimal place symbol.

The values of batch, fraction, and retention time can then be obtained from the `[1,2]` position of the output of `str_match()`.

More robust patterns can be written, for example, to allow for scientific notation and other types of numbers and

patterns. I tend to keep patterns as simple as possible expanding complexity only when the pattern I'm matching becomes demanding. When you use regular expressions, you are writing in a second language (*regex*), not R, and while it is powerful, it is not easy to read for everyone. To make programs as easy to read as possible, my advice is to limit the complexity of secondary languages. This will especially be true for extracting data from databases using languages such as SQL.

To simplify a program like this, I usually isolate helper functions that are used in multiple places first. Again, I am going to use the `for()` loop rather than use a parallel approach to make the operation easier to understand and debug. Many people take the approach that R is simply a scripting language and often don't pay attention to some of the useful ideas from basic program design. The result can be a lot of repeated code generated by cutting and pasting. While it is usually not worth the time to ensure your R code meets professional software engineering standards, it is worth taking enough time to make sure that at least *you* can understand the program later and that others don't have too much trouble reading your programs. That is the approach you will see throughout this book. I am not trying to use all of the latest features of either Bioconductor or the `tidyverse` packages, but I do try to make programs easy to read (at least for my future self).

Based on using the regular expression method, the tibble I intend to build is:

```
  tibble(
      batch = numeric(),
      fraction = numeric(),
      seq = character(),
      charge = numeric(),
      ex_mz = numeric(),
      calc_mz = numeric(),
      match_score = numeric(),
      exp_score = numeric(),
      scan = numeric(),
      rt = numeric(),
      mods = character(),
      base_filename = character()
  )
```

```
## # A tibble: 0 x 12
## # i 12 variables: batch <dbl>, fraction <dbl>, seq
<chr>, charge <dbl>,
## #   ex_mz <dbl>, calc_mz <dbl>, match_score <dbl>,
exp_score <dbl>, scan <dbl>,
## #   rt <dbl>, mods <chr>, base_filename <chr>
```

To put the program together, I create functions to perform the actions that I will need repeatedly later. First, a function to validate the XML file against the schema file.

```
  mzID_valid <- function(mzID_filename, id_schema) {
      doc <- read_xml(mzID_filename)
      xml_validate(doc, id_schema)
  }
```

The `empty_psm()` function creates an empty tibble to hold the peptide-spectrum match data in order to implement an *accumulator* design pattern. The idea of design patterns was made popular by the book *Design patterns: Elements of reusable object-oriented software* [101]. Design patterns are simply well-established ways of getting things done in software. Design patterns, in general, are outside the scope

of this book, but the accumulator pattern is so low-level and basic that it will probably be useful in your analysis and commonly found in code you might read. The idea used here is simple: create a data structure to hold a single observation – like a row in a table – and then accumulate these into a table containing many observations.

```
empty_psm <- function() {
    tbl_item <- tibble(
        batch = numeric(),
        fraction = numeric(),
        seq = character(),
        charge = numeric(),
        ex_mz = numeric(),
        calc_mz = numeric(),
        match_score = numeric(),
        exp_score = numeric(),
        scan = numeric(),
        rt = numeric(),
        mods = character(),
        base_filename = character()
    )
    tbl_item
}
```

Create a function to extract PSM data so it can be added to a table:

```
extract_tandem_match <- function(psm) {
    m1 = str_match(psm$spectrumid, "_Batch(\\d+)_")
    m2 = str_match(psm$spectrumid, "_BRPhsFr(\\d+)")
    m3 = str_match(psm$spectrumid, "RTINSECONDS=([0-
9.-]+)")

    tbl_item <- tibble(
        batch = as.numeric(m1[1,2]),
        fraction = as.numeric(m2[1,2]),
        seq = psm$pepseq,
        charge = psm$chargestate,
        ex_mz = psm$experimentalmasstocharge,
        calc_mz = psm$calculatedmasstocharge,
        match_score = psm$x..tandem.hyperscore,
        exp_score = psm$x..tandem.expect,
        scan = psm$scan.number.s.,
        rt = as.numeric(m3[1,2]),
        mods = psm$modification,
        base_filename =
str_split_1(psm$spectrumFile,".mgf")[1]
    )
    tbl_item
}
```

A feature of the X! Tandem search program is that in its output it reports when a peptide has been modified in different locations.

```
top_tandem_scans <- function(all_id) {

    all_id <- all_id |>
        arrange(scan.number.s.)

    pep_tbl <- empty_psm()

    pep_score_best <- 0.0
    curr_scan <- all_id[1,]$scan.number.s.

    pep_tbl <- empty_psm()

    for(i in 1:length(all_id$pepseq)) {
        peptide <- all_id[i,]
        pep_scan <- peptide$scan.number.s.
        if(pep_scan == curr_scan) {
            if(peptide$x..tandem.hyperscore >
pep_score_best) {
                pep_best <-
extract_tandem_match(peptide)
                pep_score_best <-
peptide$x..tandem.hyperscore
            }
            next
        } else {
            pep_tbl <- bind_rows(pep_tbl, pep_best)
            pep_best <- extract_tandem_match(peptide)
            curr_scan <- peptide$scan.number.s.
            pep_score_best <-
peptide$x..tandem.hyperscore
        }
    }
    pep_tbl
}
```

Now you are ready to iterate through all of the output files from the study and build the tidy data analysis table. There does not seem to be a way to change the default setting of the `PSM()` function when using the `mzID` parser, which defaults to `verbose`. This might crop up in other application-specific

functions you use. If you want to silence them, the generic way is to wrap the expression in an `invisible(capture.output(...))` statement. This captures the output of hidden `print()` type statements but passes the actual value of the expression back out to the rest of the code. In Chapter 5, I'll show how to perform the same process using the `mzID()` function directly, and in the example in Section 5.3.4, I show how to silence a particular function, while the method shown here will work for any function you want to silence.

```
# get the schema for the mzML files
mzID_schema <-
read_xml(file.path("schema","mzIdentML1.1.0.xsd"))

# get a list of all the mzIdentML files generated by X!
Tandem
result_files = list.files(
    file.path("large-data",
"MSV000086195","tandem_result"),
    pattern=".mzid")

# create an empty tibble to accumulate desired matches
all_top_psm <- empty_psm()

n_iterations <- length(result_files)
pb <- progress_estimated(n_iterations)

# read and extract data from each mzIdentML file using
a loop
for(filename in result_files) {

    full_path = file.path("large-data",
"MSV000086195","tandem_result",
                          filename)

    if(!mzID_valid(full_path, mzID_schema)) {
        next
    }

    invisible(capture.output(all_psm <- PSM(full_path,
parser = "mzID")))
    id <- as_tibble(all_psm)

    # if the table made by PSM() is empty skip to the
next file
    # otherwise keep only the peptide matches which
contain at least one
    # TMT6plex modification (229.1629 Da)
    if(nrow(id) < 1) {
        next
    } else {
```

```
        id <- dplyr::filter(id,
str_detect(modification, '229.1629'))
    }

    # the top_scans() function returns the highest
scoring unique MS2
    # spectra, which are added to the table
    kept_peptides <- top_tandem_scans(id)
    all_top_psm <- bind_rows(all_top_psm,
kept_peptides)

    rm(all_psm)
    rm(id)

    update_progress(pb)
}
```

Sort the table by sequence, batch, fraction, and retention time, and then print the output:

```
all_top_psm <- arrange(all_top_psm, seq, batch,
fraction, rt)

print(all_top_psm)
```

```
## # A tibble: 220 x 12
##    batch fraction seq     charge ex_mz calc_mz
match_score exp_score  scan    rt
##    <dbl>    <dbl> <chr>    <dbl> <dbl>   <dbl>
<dbl>     <dbl> <dbl> <dbl>
##  1     2        8 AMGNSL~      2  756.    756.
59.3  4.6 e-12 25054 5107.
##  2     2        8 AMGNSL~      3  504.    504.
39.5  4.1 e- 7 25125 5119.
##  3     3        9 AMGNSL~      2  756.    756.
59    5.6 e-12 25185 5314.
##  4     3        9 AMGNSL~      3  504.    504.
38.1  9.3 e- 7 25263 5329.
##  5     1        5 APLDND~      2  844.    844.
54.5  1.80e-10 21516 4796.
```

```
##  6     2         3 APLDND~      2  844.    844.
46.7  5   e- 9 21982 4592.
##  7     2         3 APLDND~      2  844.    844.
42.3  6.5 e- 7 22560 4673.
##  8     2         9 APLDND~      2  844.    844.
53.8  2.70e-10 20725 4459.
##  9     3         4 APLDND~      2  844.    844.
46.6  5.50e- 9 23423 4859.
## 10     3         5 APLDND~      2  844.    844.
43.1  1.20e- 7 21639 4744.
## # i 210 more rows
## # i 2 more variables: mods <chr>, base_filename <chr>
```

The last step in building up this data table is to read the raw data in order to obtain the precursor scan number (MS level 1) for the MS level 2 spectra that were searched from the `.mgf` files, since the `.mgf` file format does not give the precursor scan number for the spectrum.

As mentioned in Section 2.1.2, I will use the `MSnbase` package to read raw data files based on the file names I extracted from the `.mzid` files and stored in the `base_filename` column of the `all_matches` table. In the data that were deposited for `MSV000086195`, there are `mzML` format files available in the `ccms_peak` folder. The `readMSData()` function can read many open format file types and supports multiple file types and both in-memory and on-disk storage [14].

```
library(MSnbase)

# get the raw filename from the all_top_psm table
created above
filename <-
paste0(all_top_psm$base_filename[1],".mzML")
full_path <- file.path("large-data",
"MSV000086195","ccms_peak",
                       filename)

spectra <- readMSData(full_path, mode="onDisk",
verbose=FALSE)
```

The `readMSData()` function returns an object of type `OnDiskMSnExp` when used with the `mode="onDisk"` argument:

```
  class(spectra)
```

```
## [1] "OnDiskMSnExp"
## attr(,"package")
## [1] "MSnbase"
```

Since you may want other data items from the spectrum, it can be useful to use the `methods()` function from the `utils` package to list the methods associated with the `OnDiskMSnExp` class.

```
  methods(class="OnDiskMSnExp")
```

```
##   [1] [                         [[
$
##   [4] $<-                       abstract
acquisitionNum
##   [7] addIdentificationData     all.equal
analyser
##  [10] analyserDetails           analyzer
analyzerDetails
##  [13] assayData                 bin
bpi
##  [16] centroided                centroided<-
chromatogram
##  [19] classVersion              classVersion<-
clean
##  [22] coerce                    coerce<-
collisionEnergy
##  [25] combineSpectra            compareSpectra
description
##  [28] detectorType              dim
dirname
##  [31] dirname<-                 estimateMzResolution
estimateNoise
##  [34] expemail                  experimentData
```

```
expinfo
##  [37] exptitle                extractPrecSpectra
fData
##  [40] fData<-                 featureData
featureData<-
##  [43] featureNames            featureNames<-
fileNames
##  [46] filterAcquisitionNum    filterEmptySpectra
filterFile
##  [49] filterIsolationWindow   filterMsLevel
filterMz
##  [52] filterPolarity          filterPrecursorMz
filterPrecursorScan
##  [55] filterRt                fromFile
fvarLabels
##  [58] fvarMetadata            header
idSummary
##  [61] initialize              instrumentCustomisations
instrumentManufacturer
##  [64] instrumentModel         ionCount
ionSource
##  [67] ionSourceDetails        isCentroided
isCurrent
##  [70] isolationWindow         isolationWindowLowerMz
isolationWindowTargetMz
##  [73] isolationWindowUpperMz  isVersioned
length
##  [76] msInfo                  msLevel
MSmap
##  [79] mz                      normalize
notes
##  [82] pData                   pData<-
peaksCount
##  [85] phenoData               phenoData<-
pickPeaks
##  [88] plot                    plot2d
plotDensity
##  [91] plotMzDelta             polarity
precScanNum
##  [94] precursorCharge         precursorIntensity
precursorMz
##  [97] processingData          protocolData
```

```
pubMedIds
## [100] pubMedIds<-                quantify
removeMultipleAssignment
## [103] removeNoId                 removePeaks
removeReporters
## [106] rtime                      sampleNames
sampleNames<-
## [109] scanIndex                  show
smooth
## [112] smoothed                   smoothed<-
spectra
## [115] spectrapply                splitByFile
tic
## [118] trimMz                     updateObject
varLabels
## [121] varMetadata                writeMgfData
writeMSData
## see '?methods' for accessing help and source code
```

As you can see, the `OnDiskMSnExp` class has a rich set of functions. In RStudio, you can use the `Help` feature to access the manual page for all of these functions.

```
# get the scan number from the first row of the table
created earlier
psm_scan_num <- all_top_psm$scan[1]

# get the precursor scan number for the first MS2
spectrum
prec_scan <- precScanNum(spectra[psm_scan_num])

print(paste0("The precursor scan for ", psm_scan_num, "
is ", prec_scan))
```

```
## [1] "The precursor scan for 25054 is 25053"
```

Reading each `.mzML` file can be slow, especially for high-resolution instruments as were used in the example study. One of the advantages of having data in a tidy format is the ability to control sorting and iteration beyond what might be

available in application-specific packages. To create a column of precursor scan numbers and add it to the `all_matches` table, I can sort the table of peptide matches by `base_filename` and then open the individual raw data files one at a time, extracting all the data needed, and then move to the next unique file in the table.

```
# Since it is possible that a raw (.mzML) file could be
missing or be
# corrupted in a way that makes the readMSData()
function fail.
# Here, I define two functions for the tryCatch()
function used below.
# The program will first try to perform the
normalRead() function
# if it fails for some reason then it will run the
exceptionRead()
# function. The point is that we want the program to
simply put a
# missing value for the precursor scan if it's not
available and then
# continue with the rest of the files rather than halt

normalRead <- function(full_path) {
    readMSData(full_path, mode = "onDisk", verbose =
FALSE)
}
```

```
get_precursor_scan <- function(psm_tbl) {

    # File read exception handler
    exceptionRead <- function(e) {
        print(paste0(curr_file," read error"))
        return(NULL)
    }

    psm_tbl <- arrange(psm_tbl, base_filename)
    curr_file <- ""

    precursor_scan_list <- tibble(
        base_filename = character(),
        scan = integer(),
        precursor_scan = integer()
    )

    n_iterations <- length(psm_tbl$base_filename)
    pb <- progress_estimated(n_iterations)

    for(i in 1:n_iterations) {

        update_progress(pb)

        filename <- psm_tbl$base_filename[i]

        if(filename == curr_file) {

            if(is.null(sp)) {
                precursor_scan <- NaN
            } else {
                precursor_scan <-
precScanNum(sp[psm_tbl$scan[i]])
            }

            prec_scan <- tibble(
                base_filename =
psm_tbl$base_filename[i],
                scan = psm_tbl$scan[i],
                precursor_scan = precursor_scan
            )
```

```
            precursor_scan_list <-
bind_rows(precursor_scan_list, prec_scan)
            next
        } else {
            rm(sp)
            gc()
            curr_file <- filename
            full_path <- file.path("large-data",
"MSV000086195","ccms_peak",
                                   paste0(curr_file,
".mzML"))
            id_scan <- psm_tbl$scan[i]

            # Here the program attempts to read the raw
data file
            # if there is an error, the sp variable
will get the value
            # returned by the function specified
            sp <- tryCatch(normalRead(full_path),
error=exceptionRead)

            # if the read failed, put a missing value
(NaN) in for the
            # precursor scan, otherwise, get the
precursor scan from the
            # raw data
            if(is.null(sp)) {
                precursor_scan <- NaN
            } else {
                precursor_scan <-
precScanNum(sp[id_scan])
            }

            precursor <- tibble(
                base_filename =
psm_tbl$base_filename[i],
                scan = id_scan,
                precursor_scan = precursor_scan
            )
            precursor_scan_list <-
bind_rows(precursor_scan_list,
                                              precursor)
```

```
        }
    }
    precursor_scan_list
}
```

Now, I can build the final table using the functions defined so far.

```
# get all of the MS1 scans which are precursors to the
top MS2 scans
psm_precursors <- get_precursor_scan(all_top_psm)
```

```
# join the precursor scan numbers to the peptide
matched scans
all_top_psm <- all_top_psm |>
    inner_join(psm_precursors,
by=join_by(base_filename, scan))

# show the contents of the first row (as strings)
as_tibble(t(all_top_psm[1,]), .name_repair,
rownames="Variable") |>
    dplyr::rename(Value=V1)
```

Now, the `all_top_psm` table can be used to perform exploratory data analysis. It's often convenient to save tables like this for later use by writing a `.csv` file to disk.

```
all_top_psm <- arrange(all_top_psm, seq, batch,
fraction, rt)
write_csv(all_top_psm, file.path("data",
"all_top_psm.csv"))
```

3.7 Summary

It is often the case that application-specific libraries and packages generate data that, while consistent within the

application domain, carry too many assumptions about how the data will be used. If these assumptions (or opinions) match your needs, then all of your data analysis can occur using only the data structures provided. However, it is often the case in mass spectrometry that only a part of a useful library applies to your situation. You can use the parts of a package in Bioconductor, for example, and create tidy versions of the data that have exactly what you need ready to be used in your analysis.

In the next chapter, I'll take the output of the example in Section 3.5 and use it to demonstrate several ways to explore mass spectrometry data of different types.

Chapter 4
Exploratory Data Analysis

4.1 Introduction

In this chapter, I will cover the critical step of exploring mass spectrometry data. R is particularly well suited for performing exploratory data analysis, including statistical summarization and preliminary data visualization. Because mass spectrometry data vary widely in type and content, general characterization and statistical summary are essential first steps. However, analysis cannot proceed without understanding the data set contents, organization, and available metadata. Performing exploratory data visualization is usually an early step in analyzing spectra, chronograms (magnitudes over time without a separation), and chromatograms (magnitudes over time with a separation stage).

4.2 Exploring Tabular Data

It is not surprising that tools like Excel find such wide use in analyzing mass spectrometry data given the prevalence of tabular data in all types of research. The wild success of table-based databases could have only happened if most of our data could be represented as tables and sets of related tables. It is important to be able to reproducibly analyze data in tabular form, which as described earlier in this book, is quite difficult with ad hoc tools like spreadsheets. The power of spreadsheets is that they start with a view that naturally lends itself to exploring the data you have.

I would make the argument, however, that true exploratory data analysis goes far beyond what can be done easily with spreadsheets, and that in R, a much richer exploration can be performed, leading to a more powerful analysis. During the exploratory phase of an analysis project, the goal is to understand aspects of your data which will lead to selecting specific analysis approaches and the acceptance or rejection of various statistical assumptions. For example, if parts of your data do not follow a normal distribution, or they contain outliers, many standard statistics (like mean and standard deviation) can have no meaning. It will also affect the use of various hypothesis test methods. Therefore, it is essential to understand what's going on for all the variables you might want to use for various types of analysis.

4.2.1 Statistical Summarization

Let's start by loading a file containing some drug screen data. The file is called `opioids_peaks.csv`, and you can load it as follows:

```
opioid_msdata <-
read_csv(file.path("data","opioid_peaks.csv")) |>
  mutate_if(is.character, as.factor)

opioid_msdata
```

```
## # A tibble: 555 x 9
##    injection compound sample_type quant_area qual_area
quant_rt qual_rt response
##        <dbl> <fct>    <fct>           <dbl>     <dbl>
<dbl>   <dbl>    <dbl>
##  1         1 Codeine  standard         6046.     4994.
1.29    1.29    1.45
##  2         1 Oxycodo~ standard         1736.     1026.
1.41    1.42    0.451
##  3         2 Codeine  standard        12817.    10377.
```

```
1.28    1.28    2.93
##  4         2 Oxycodo~ standard         3738.     1792.
1.41    1.41    1.02
##  5         3 Codeine  standard        58342.    47001.
1.28    1.28   14.3
##  6         3 Oxycodo~ standard        15805.     8793.
1.41    1.41    5.38
##  7         4 Codeine  standard       152514.   116663.
1.29    1.29   44.0
##  8         4 Oxycodo~ standard        35823.    19784.
1.42    1.42   13.6
##  9         5 Codeine  qc             128370.   105415.
1.28    1.28   27.8
## 10         5 Oxycodo~ qc              30576.    17761.
1.41    1.41    9.30
## # i 545 more rows
## # i 1 more variable: ion_ratio <dbl>
```

From this data, you can see that there are 555 rows with a batch ID, batch name, injection number (position in the batch), and the sample type. Also, the areas and retention times of the quantifier chromatographic peaks and qualifier peaks are given along with the names of the compounds. The computed values instrument response (quantifier area divided by the internal standard [IS] area) and the ratio of the quantifier area divided by the qualifier area are also in this table.

This data set contains liquid chromatography with tandem mass spectrometry (LC-MS/MS) data from selected reaction monitoring (SRM) measurements of human urine toxicology samples in which precursor ions were selected and a fragment ion was measured as a function of time, creating chromatograms for each compound observed. In this particular measurement, both the target compound and a stable isotopically labeled (SIL) version of the target compound were measured. As is typical for this type of measurement, one precursor fragment is selected as the *Quantifier* ion, and its area is divided by the SIL fragment ion area to generate an instrument response. To confirm that this

response can reasonably be assigned to the target compounds, a second fragment from the precursor ion is selected as a *Qualifier*. Since the quantifier and qualifier come from the same precursor, they should have a fixed area ratio and nearly identical retention times. The approach of using a SIL compound which is spiked into an unknown sample to correct for variations in preanalytical and analytical processes is very common. The use of a second fragment ion to improve the confidence that the resulting instrument response came from a specific compound is also common in a wide range of mass spectrometry measurements, especially when reporting concentrations of critical compounds (e.g. toxic compounds, controlled substances, or biomarkers of disease).

In this example, the codeine quantifier monitored a peak with a precursor ion m/z of 300.5 and a product ion m/z of 152.0. The qualifier has the same precursor (300.5) and a different product: 165.1. For oxycodone, the precursor ion was m/z 316.5 and the product ion m/z 241.0 was used for the quantifier and m/z 212.1 was used for the qualifier. Later, I will show how to explore the spectra of these compounds and related substances to determine the potential for interference.

R provides an easy way to see what a data set contains using the `summary()` function.

```
  names(opioid_msdata)
```

```
## [1] "injection"   "compound"    "sample_type"
"quant_area"  "qual_area"
## [6] "quant_rt"    "qual_rt"     "response"
"ion_ratio"
```

Based on the variables provided, there are two compounds, which can be summarized individually:

```
  opioid_msdata |>
      dplyr::select(-injection) |>
      dplyr::filter(compound=='Codeine') |>
      summary()
```

```
##       compound     sample_type    quant_area
qual_area
##  Codeine  :224   qc      : 40   Min.   :     13.5
Min.   :     16.1
##  Oxycodone:  0   standard: 80   1st Qu.:   1080.9   1st
Qu.:    935.3
##                  unknown :104   Median :  12019.6
Median :   9869.5
##                                 Mean   :  84898.8
Mean   :  69947.4
##                                 3rd Qu.:  75175.7   3rd
Qu.:  66030.5
##                                 Max.   :2392432.6
Max.   :1902945.9
##     quant_rt        qual_rt         response
ion_ratio
##  Min.   :1.267   Min.   :1.264   Min.   :  0.0018
Min.   :0.3963
##  1st Qu.:1.289   1st Qu.:1.289   1st Qu.:  0.2509   1st
Qu.:1.0996
##  Median :1.298   Median :1.298   Median :  2.8711
Median :1.1593
##  Mean   :1.299   Mean   :1.299   Mean   : 21.0970
Mean   :1.1958
##  3rd Qu.:1.308   3rd Qu.:1.308   3rd Qu.: 17.5162   3rd
Qu.:1.2413
##  Max.   :1.331   Max.   :1.327   Max.   :633.9002
Max.   :3.0942
```

```
  opioid_msdata |>
      dplyr::select(-injection) |>
      dplyr::filter(compound=='Oxycodone') |>
      summary()
```

```
##       compound     sample_type    quant_area
qual_area
##  Codeine  :  0   qc      : 40   Min.   :     11   Min.
:     10
##  Oxycodone:331   standard: 80   1st Qu.:    883   1st
Qu.:    529
##                  unknown :211   Median :  14076
Median :   8246
##                                 Mean   : 209845   Mean
: 132538
##                                 3rd Qu.:  98514   3rd
Qu.:  56138
##                                 Max.   :5450375   Max.
:4403262
##     quant_rt        qual_rt         response
ion_ratio
##  Min.   :1.397   Min.   :1.404   Min.   :   0.0013
Min.   :0.2837
##  1st Qu.:1.421   1st Qu.:1.421   1st Qu.:   0.2517
1st Qu.:1.6120
##  Median :1.431   Median :1.429   Median :   4.3614
Median :1.7116
##  Mean   :1.432   Mean   :1.431   Mean   :  72.3826
Mean   :1.6668
##  3rd Qu.:1.441   3rd Qu.:1.441   3rd Qu.:  24.5704
3rd Qu.:1.7945
##  Max.   :1.479   Max.   :1.479   Max.   :2492.4049
Max.   :4.2204
```

4.2.2 Exploring Tabular Data with Plots

From the summary, you can see that the data come from two compounds, Codeine and oxycodone, and that both are almost equally represented. These observations come from human toxicology tests, and the two compounds are consumed in a range of doses and metabolized at different rates, so you would not expect the concentration (which is proportional to `response`) to follow any particular distribution. In other sample populations (e.g. controlled compound

stability tests), there might be an expected concentration distribution, so plotting the distribution of a variable can help understand the data. The `facet_wrap()` layer takes a one-dimensional sequence of panels and wraps them into two dimensions according to the variable specified using the `vars()` function.

```
# Draw histograms of the distribution of peakRTQuant
# for the two different compounds, and overlay them
p_response <- ggplot(opioid_msdata, aes(x=response)) +
        facet_wrap(vars(compound), scales = 'free') +
        geom_histogram()

print(p_response)
```

As suggested in the `summary()` table, the mean and median of `response` are quite far apart for both compounds, and from Figure 4.1 it's clear that most of the values are at or near the noise limit (which appears to be close to zero).

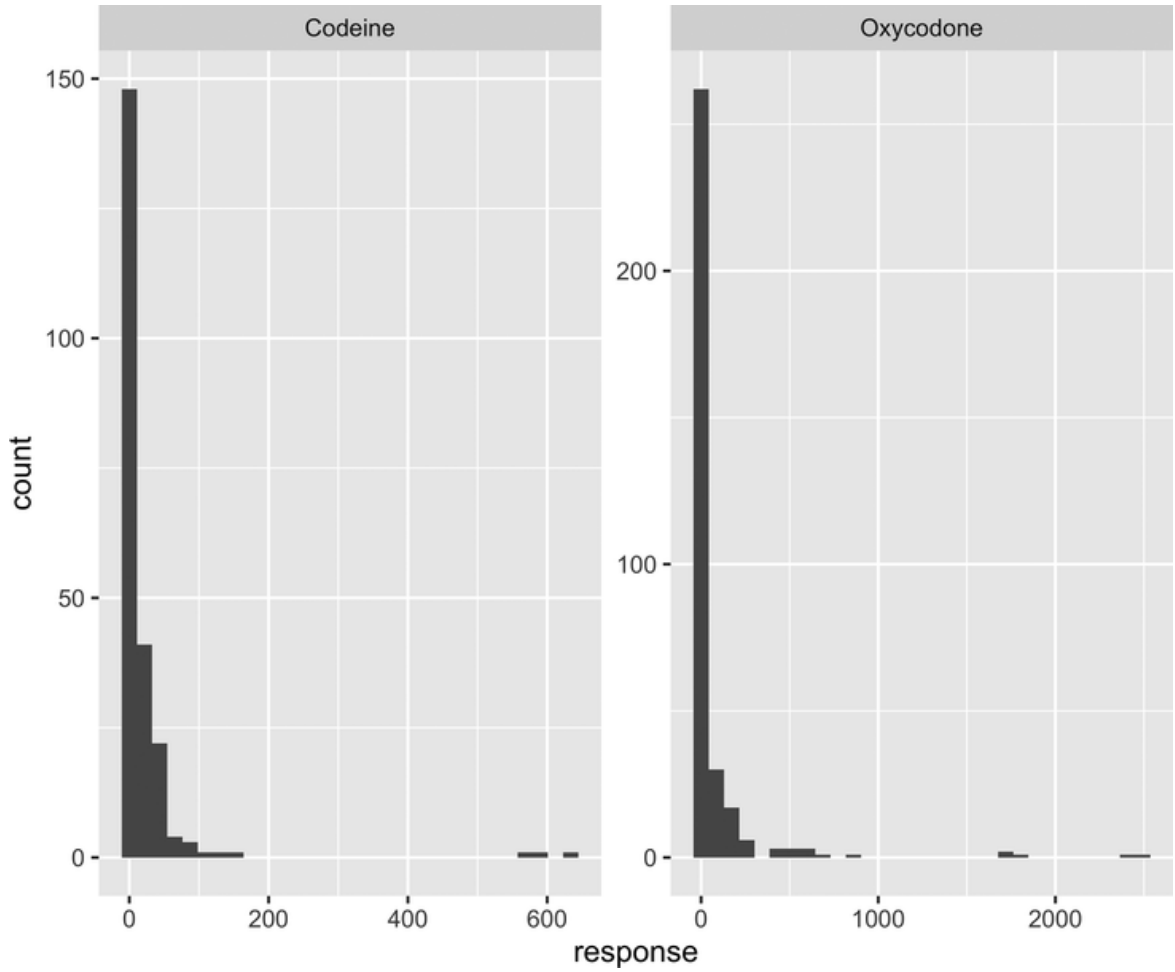


Figure 4.1 Distribution of compound responses.

Unlike the instrument response, which is related to compound concentration, the chromatographic retention time is expected to follow a well-known distribution based on the chemical separation process. In this example, a reversed-phase chromatography column was used to separate compounds from other compounds in the sample matrix. Some fluctuation from the expected retention time established during method development is expected. These are due to random changes in solvent flow control, column temperature control, solvent mixing, and even the presence of other compounds in an individual sample. However, if the process is under good control, a signal that deviates significantly from the expected retention time is possibly an entirely different compound.

```
# Draw histograms of the distribution of peakRTQuant
# for the two different compounds, and overlay them

p_quant_rt <- opioid_msdata |>
    ggplot(aes(x=quant_rt, fill=compound)) +
        geom_histogram(alpha=0.6, binwidth = 0.01,
position="identity")

print(p_quant_rt)
```

The plot in Figure 4.2 shows that the retention times for the Codeine and oxycodone quantifiers are roughly normal distributions which suggests that the `mean` value in the summary table is approximately the same as the median.

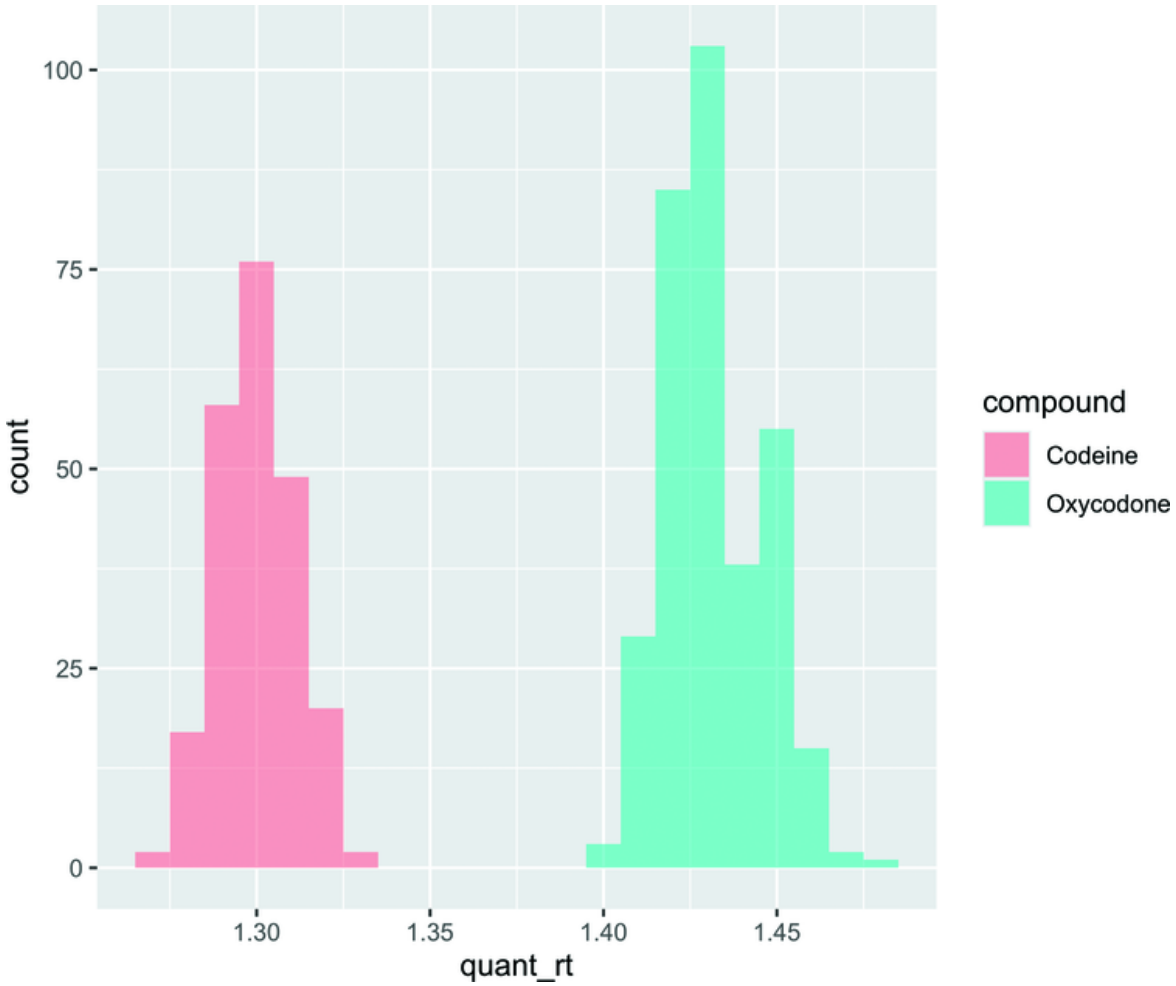


Figure 4.2 Distribution of quantifier peak retention times.

If the deviation from the expected retention time was due to a lack of selectivity in the mass spectrometer, then it might be reasonable to exclude the low retention time observations. Since the qualifier ion is supposed to have come from the same intact precursor molecule, if there is a significant deviation between the quantifier retention time and the qualifier retention time, then it is unlikely that the two signals came from the same compound and they could be outliers.

Plotting the two retention times against each other should show a fixed relationship between the two retention times if the fragments came from the same precursor ion. When

`color` is specified inside the `aes()` function of a layer, a legend is automatically created based on the colors assigned to the variables plotted.

```
p_qual_quant_rt <- opioid_msdata |>
    ggplot() +
        geom_point(aes(x=quant_rt, y=qual_rt,
color=compound))

print(p_qual_quant_rt)
```

Figure 4.3 shows no significant difference between the retention times of the quantifier ion and the qualifier ion, which indicates that they came from the same precursor molecule. However, I have not ruled out the possibility of multiple precursor ions. When a molecular ion is excited to a particular energy, it undergoes a unimolecular dissociation reaction, which creates reaction products, some neutral and some ionized. The ionized reaction products are detected as fragments, and if the energy of the reaction is fixed, then the reaction rates are constant and the yield of the various reaction products is fixed. The idea behind the qualifier ion is that the ratio of the quantifier fragment and the qualifier fragment should remain fixed for a given molecular ion assuming the reaction energy is controlled.

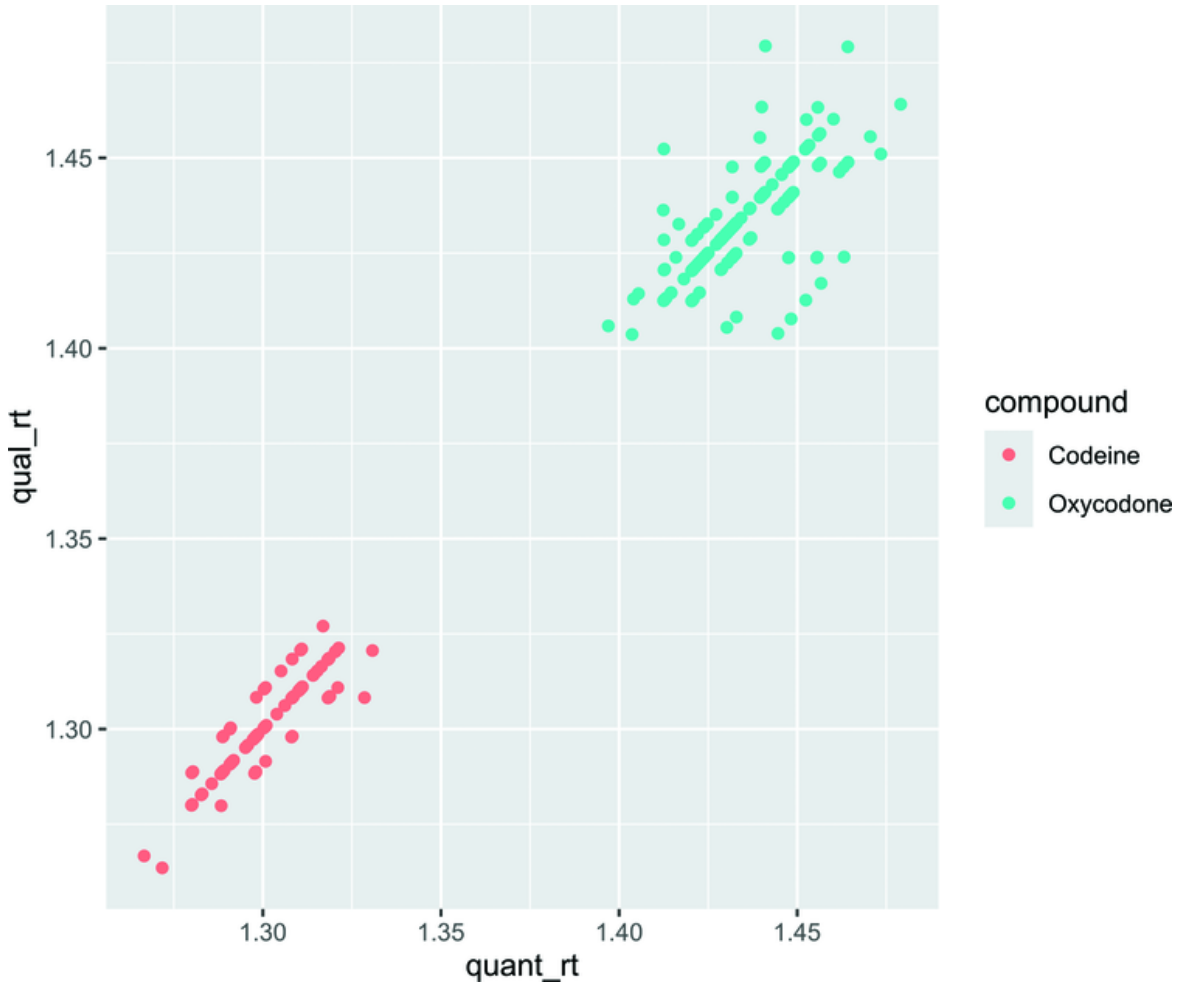


Figure 4.3 Qualifier peak retention time versus quantifier peak retention time.

Another visualization that could be helpful is to compute the ion ratio and compare it to the retention times of the quantifier and qualifier ions.

```
p_ratio_rt <- opioid_msdata |>
    ggplot() +
        geom_point(aes(y=quant_rt, x=ion_ratio,
color=compound))

print(p_ratio_rt)
```

The plot in Figure 4.4 suggests a way to analyze the data: use the spread in the ion ratio values for the high-confidence data

near the expected retention time to test the hypothesis that the lower retention time samples are likely to be drawn from the same population as the expected retention times. If they are, further investigation into the separation stability and mass spectrometer selectivity might be needed. This plot also shows why most assays of this type place limits on both the retention time tolerance and the ion ratio tolerance to prevent real retention time shifts from creating potential interferences with other compounds that might elute at an earlier time.

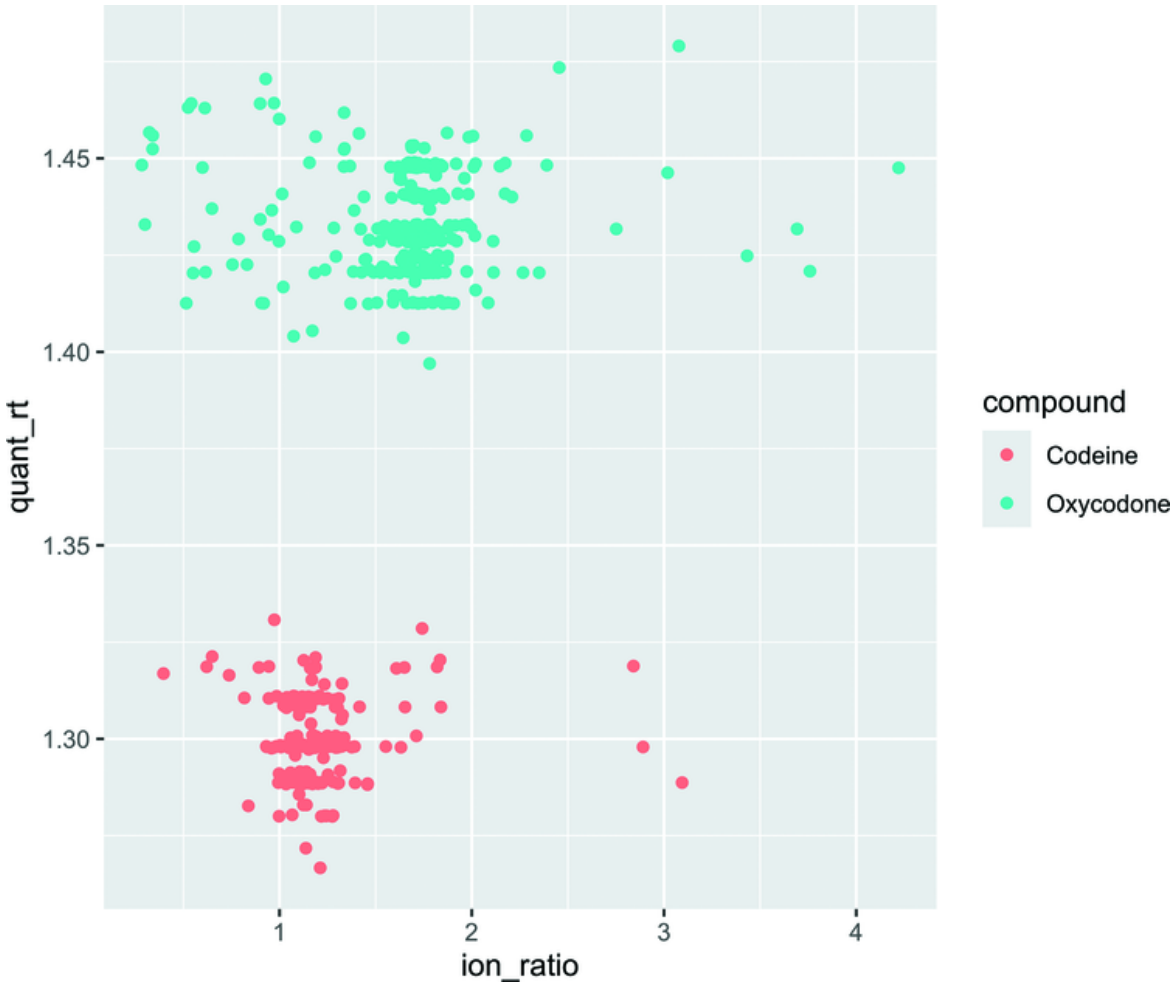


Figure 4.4 Comparison of retention times and ion ratios.

```
p_ratio_response <- opioid_msdata |>
    ggplot() +
        facet_wrap(vars(compound), scales = 'free') +
        geom_point(aes(y=quant_area, x=ion_ratio)) +
        scale_y_continuous(trans='log10',labels =
scales::comma)

print(p_ratio_response)
```

An interesting observation from the data in Figure 4.5 is that for both compounds, the ion ratio values are more dispersed when the quantifier peak area is lower. This means there is some lower limit of concentration below which the ion ratio

becomes less effective at identity confirmation. Performing hypothesis testing on observations for exploratory analysis could be the next step in the analysis of this type of data. Statistical analysis of liquid chromatography–mass spectrometry (LC-MS) and LC-MS/MS data (hybrid MS) will be discussed in more detail in Chapter 6.

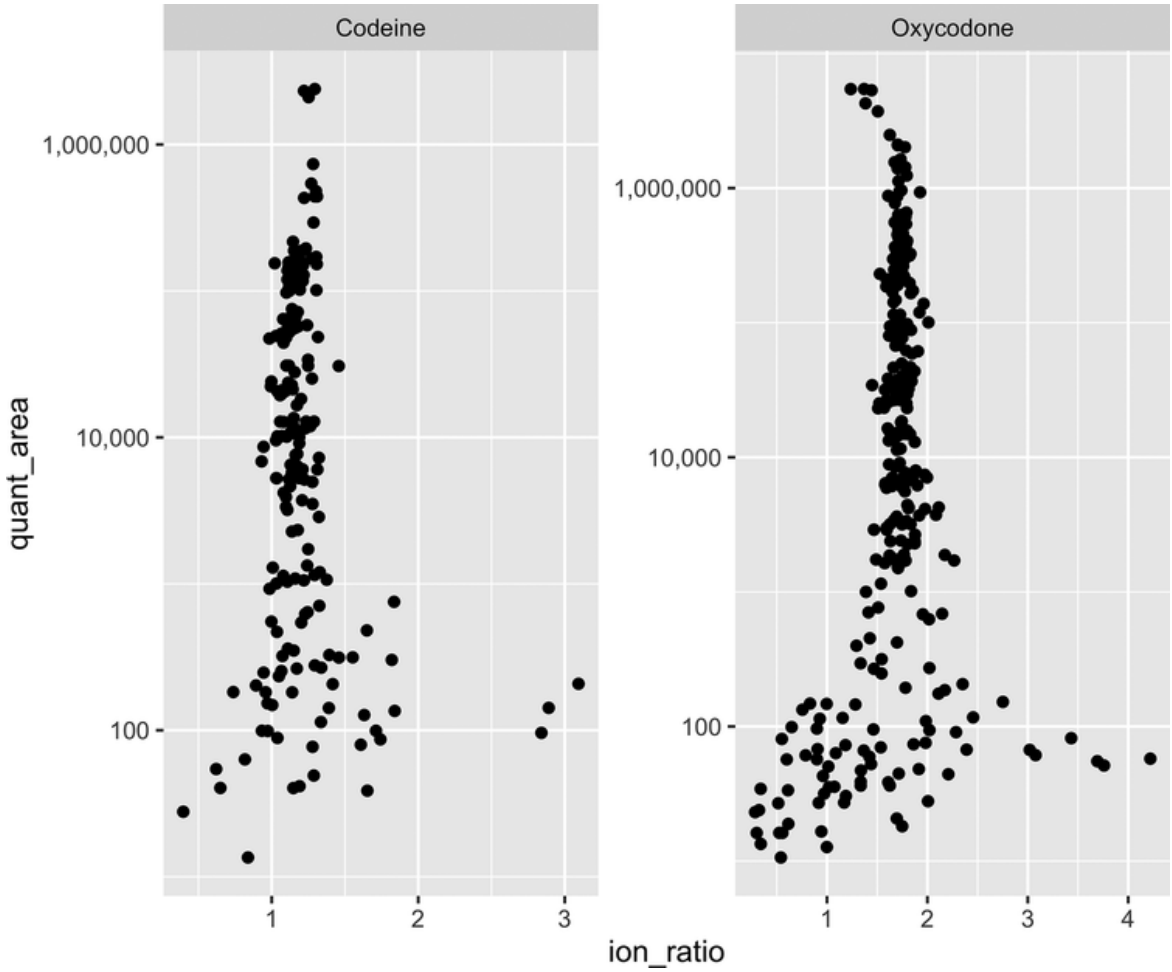


Figure 4.5 Comparison of quant peak area and ion ratios.

To explore the Ubaida-Mohien 2022 study [88] I can use the data collected in Section 3.6 to perform some summary analysis of the tibble created, which connected the spectrum matches to the original fractions and the MS level 1 spectra in those fractions.

From this data, I can tell how similar the batches were in terms of peptide-spectrum matches for the control peptides by looking at how many peptide identifications were recorded for each fraction in each batch.

Load the csv file:

```
all_top_psm <- read_csv(file.path("data",
"all_top_psm.csv"))
```

And plot the distribution of the PSMs:

```
p_top_psm <- all_top_psm |>
    ggplot() +
        ggtitle("Distributions of PSMs by Batch and
Fraction (MSV000086195)") +
        facet_wrap(vars(batch)) +
        geom_histogram(aes(x=fraction)) +
        theme(plot.title = element_text(hjust = 0.5))

print(p_top_psm)
```

From the histogram in Figure 4.6, it appears that fraction 5 contains multiple matches in all three batches. I'll start with this fraction to explore the raw MS data. This is typical of exploratory data analysis: you create some specialized summarizations which are first used for exploration, and then used in specific analysis.

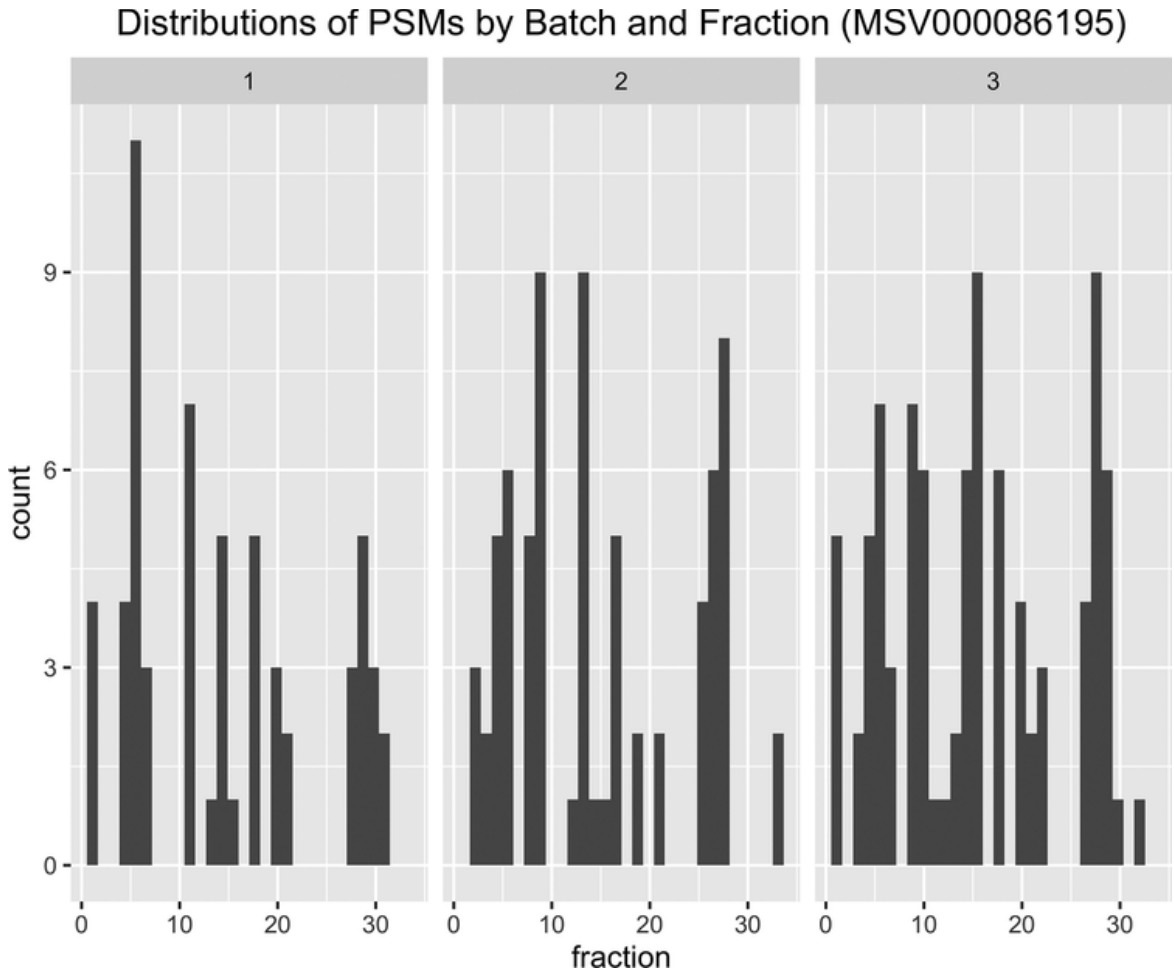


Figure 4.6 Counts of internal control PSM identifications by batch and fraction.

To examine fraction 5, the tibble can be filtered to show what was matched in batch 1.

```
psm_batch1_fraction5 <- all_top_psm |>
    dplyr::filter(batch == 1, fraction == 5) |>
    dplyr::select(-c(batch,fraction,base_filename))

head(psm_batch1_fraction5)
```

```
## # A tibble: 6 x 10
##   seq               charge ex_mz calc_mz match_score
exp_score  scan    rt mods
##   <chr>              <dbl> <dbl>   <dbl>       <dbl>
```

```
<dbl> <dbl> <dbl> <chr>
## 1 APLDNDIGVSEATR         2  844.    844.        54.5
1.80e-10 21516 4796. 229.~
## 2 DVSLLHKPTTQISDFH~      3  909.    909.        44
7.6 e- 8 22774 4974. 229.~
## 3 DVSLLHKPTTQISDFH~      3  909.    909.        49.4
1.10e- 9 22782 4975. 229.~
## 4 TADTLADAVLITTAHA~      3  937.    936.        52
3.20e-10 48271 9220. 229.~
## 5 TADTLADAVLITTAHA~      3  937.    937.        54.7
6.60e-11 48333 9230. 229.~
## 6 TMITDSLAVVLQR          3  565.    565.        43.3
4.80e- 8 41938 8142. 229.~
## # i 1 more variable: precursor_scan <dbl>
```

The first entry looks like a very good match so I can plot the spectrum from the mzML file.

4.3 Exploring Raw Mass Spectrometry Data

Similar to exploring tabular data, spectral data can be explored using both summarization methods and plotting. In addition to the `MSnbase` package, the `Spectra` package [16] is another excellent tool for exploring and processing spectra. For a quick look at an individual spectrum from the repository, I'll load the file using the `Spectra()` function and plot the first precursor (MS level 1) spectrum from the tibble above.

```
raw_file_name <- file.path("large-
data","MSV000086195","ccms_peak",

"ScltlMsclsMAvsCntr_Batch1_BRPhsFr5.mzML")
batch1_fraction05 <- Spectra(raw_file_name)
plotSpectra(batch1_fraction05[21515],
xlim=c(844.25,846.5), ylim=c(0,1.5e7))
```

Several pieces of information are clear from the plot in Figure 4.7. The X! Tandem result summary tibble indicated that the precursor peptide used for identification was experimentally observed at `844.4393` Da and was doubly charged. This plot shows the isotopic pattern with the monoisotopic ion having the highest intensity for this peptide, and the mass difference between the next two significant peaks is approximately `0.5` Da, consistent with the peptide being in charge state `2`. Also, it is apparent from this plot that the data in the repository has been centroided prior to conversion to XML. This is usually done to save space, and is an option in the MSConvert program called “peak picking.” To determine if this is the case, I converted the vendor `.raw` file to mzML *without* the “peak picking” turned on to create another version of the `.mzML` file.

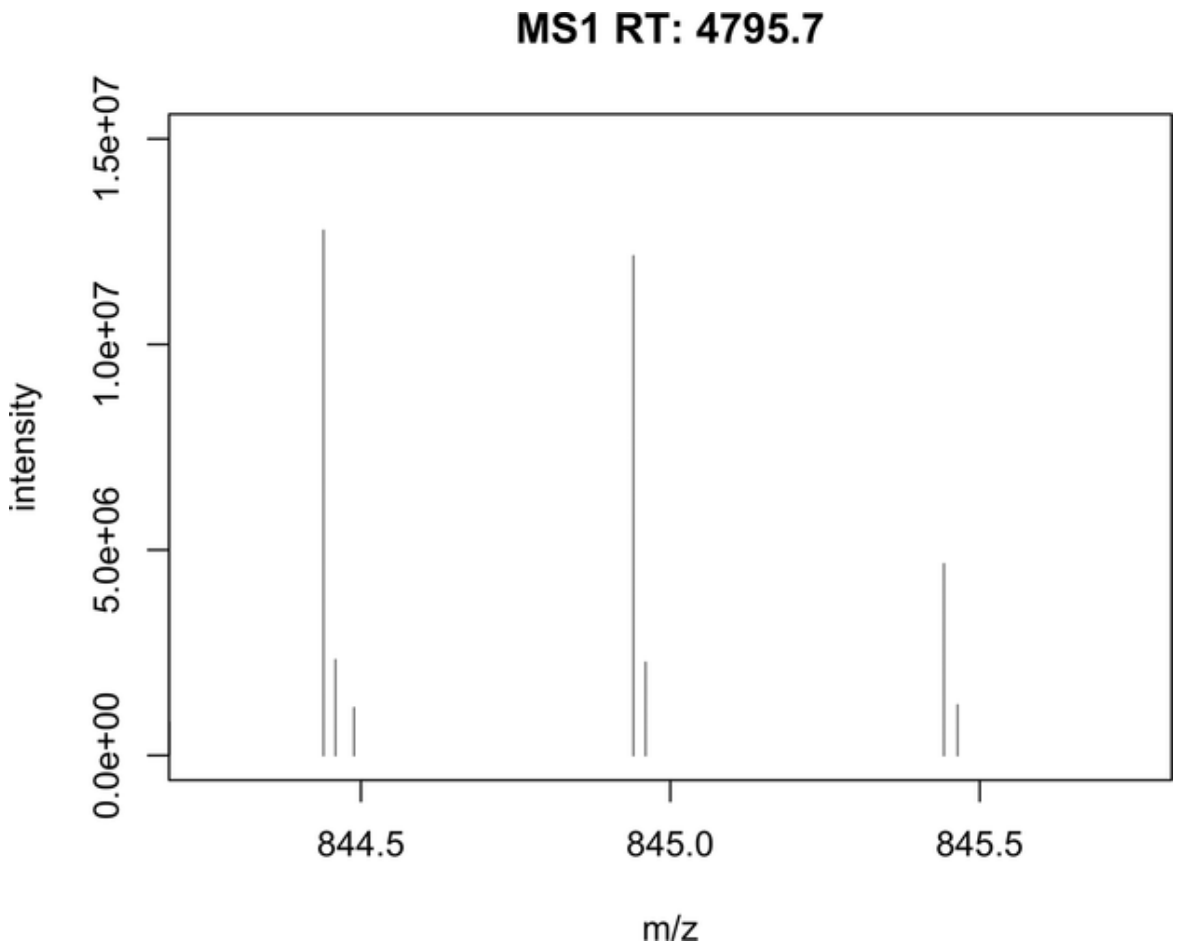


Figure 4.7 Mass spectrum of precursor for APLDNDIGVSEATR in Batch 1, Fraction 5.

4.3.1 Profile Data

The data in the raw files provided by the MSV000086195 project, were collected in profile mode, meaning that each peak is actually multiple data points and the final molecular weight was computed from those peaks. Let's take a look at the primary peak that was selected for fragmentation in this scan using a more customized plot. First, load the mz and intensity data for both the profile and centroided spectra.

```
# load the profile data from the profile raw data

profile_file_name <- file.path("large-data",

"ScltlMsclsMAvsCntr_Batch1_BRPhsFr5_prof.mzML")

batch1_fraction05_profile <- Spectra(profile_file_name)

# Based on the summary table the peptide of interest is
at scan number 21515
prof_prec_scan <- batch1_fraction05_profile[21515]
prof_x <- mz(prof_prec_scan)[[1]]
prof_y <- intensity(prof_prec_scan)[[1]]

# The centroided data from the vendor was loaded in the
previous example

cent_prec_scan <- batch1_fraction05[21515]
cent_x <- mz(cent_prec_scan)[[1]]
cent_y <- intensity(cent_prec_scan)[[1]]
```

And then make an overlay plot

```
p_overlay <- ggplot() +
    coord_cartesian(xlim=c(844.25,846.5),
ylim=c(0,1.5e7)) +
    scale_y_continuous(labels = inten_label) +
    geom_segment(aes(x=cent_x, y=cent_y,yend = 0, xend
= cent_x),
                 linewidth = 0.5, color=pal$blue) +
    geom_line(aes(x=prof_x, y=prof_y)) +
    geom_point(aes(x=prof_x, y=prof_y), shape=1 ) +
    xlab("m/z") +
    ylab("Intensity") +
    theme_classic() +
    theme(plot.title = element_text(hjust = 0.5)) +
    theme(plot.subtitle = element_text(hjust = 0.5)) +
    ggtitle(label = "MSV000086195 Batch 1 Fraction 5
(Scan 21515)",
            subtitle = "MS1 APLDNDIGVSEATR 2+
(Deaminidated N, TMT10plex)")

print(p_overlay)
```

The plot in Figure 4.8 overlays the mzML spectrum from the data deposited for MSV000086195 with the raw data without any data reduction. It is clear that multiple data points are collected for each molecular weight reported. Profile spectra contain the continuous values produced by the detector. In this particular case, the instrument is a Fourier Transform (FT) MS-type instrument, which collects a time-domain ion current representing ion frequencies in an electrostatic ion trap. The time-domain data is then converted into frequency-domain data in which each observed frequency can be calibrated to a specific m/z value. In the case of FTMS data, the conversion from the time-domain to the frequency domain produces a peak with a width corresponding to the length of time the time-domain image currents were observed. This is how FTMS instruments achieve high resolution by using the inverse relationship between

observation time and resolution in the corresponding frequency spectrum.

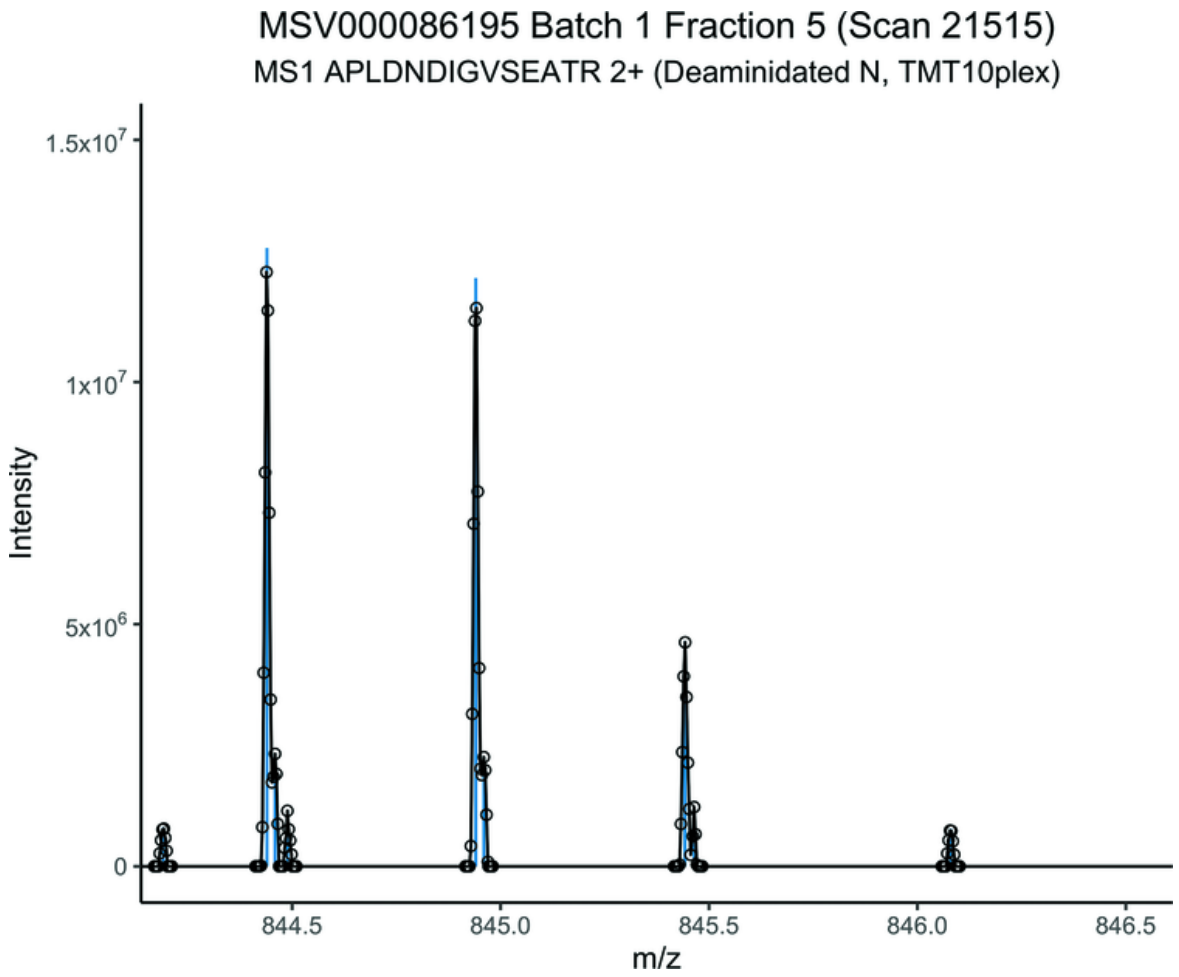


Figure 4.8 Profile spectrum of precursor for APLDNDIGVSEATR in Batch 1, Fraction 5.

For each mass analyzer type, the observed data is different. In quadrupole mass spectrometers, the voltages on the quadrupole rods relate to specific m/z values, and the raw data is obtained by changing a voltage over time, producing multiple intensity values along an x-axis which is voltage converted to m/z. In time-of-flight (TOF) instruments, it is the flight time from the source to the detector that is measured and the x-axis is converted to m/z.

```
  p_overlay_zoom <- p_overlay +
      coord_cartesian(xlim=c(844.4,844.55),
  ylim=c(0,1.5e7)) +
      annotate("text", x = 844.46, y = 1.3e7,
  color=pal$blue,
               label = "Instrument Centroid\n844.4393")
```

```
## Coordinate system already present. Adding new
coordinate system, which will
## replace the existing one.
```

```
  print(p_overlay_zoom)
```

As you can see from Figure 4.9 the profile peak has some width, unlike the centroid m/z peak which is normally shown as a "stick." The m/z peak width defines the resolution of the mass analyzer. The width at 50% of its full height, sometimes called full width at half maximum (FWHM) divided by the m/z is the definition of resolution in mass spectrometry.

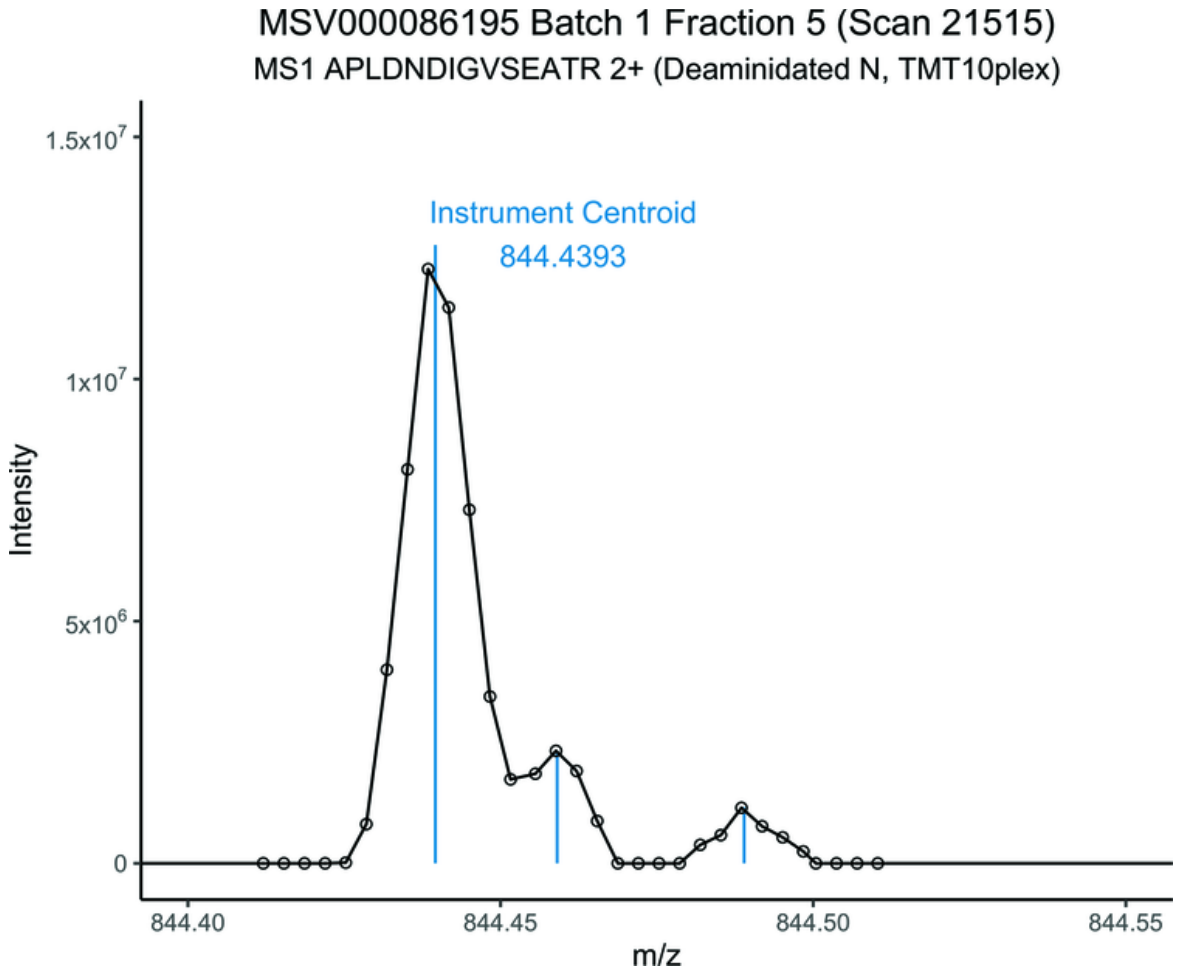


Figure 4.9 Details of m/z 844.43864 profile peak in Batch 1, Fraction 5.

The stick-like peaks seen overlaying the raw data represent an estimation of the m/z value based on computing either the maximum intensity recorded for the signal or the intensity at the "center of mass" of the profile peak– the so-called *centroid* of the peak. In Figure 4.9 the centroid spectrum was computed using the instrument vendor approach as reported by `msConvert`. Depending on the size of the expected data file, and the noise level, it may be useful to acquire data in either centroid mode (in which only the centroid values are stored), or in profile mode, when additional signal processing steps can be taken to estimate m/z values and intensities. It's worth noting that despite being collected at resolution setting of 120 000, this spectrum still has overlapping m/z

peaks. These overlaps can create problems in determining both the m/z value and its intensity. Because of overlapping peaks and noise in mass spectra, significant effort has been put into peak processing to improve mass accuracy as much as possible.

4.3.2 Centroiding and Profile Peak Processing

The `Spectra` package includes a function for picking peaks in a mass spectrum called `pickPeaks()` which has two methods for determining the centroid of a peak. For exploratory purposes, it's useful to know how close the data are to the theoretical isotope m/z values which can be computed from the chemical formula. The peptide in this example is the deamidated peptide APLDNDIGVSEATR with the N-terminus TMT-10plex adduct (`229.162932`) and two additional protons to create the +2 charged ion. The final ion has the chemical formula $C_{68}{}^{13}C_4H_{121}N_{18}{}^{15}NO_{27}{}^{2+}$ which has a theoretical monoisotopic m/z of `844.43864`. In Chapter 5, I will dig deeper into computing molecular weights of various types of molecules and describe their isotopic distributions. For now, I will use the results of the calculation for this specific formula to compare the centroided value to the theoretical value.

```
# there are many standalone, and on-line tools for
computing relative
# abundances of the isotopes of specific structures.

theoretical_x <- c(844.43864, 844.94032, 845.44200,
845.94368)
theoretical_y <- c(1.0, 0.735, 0.266, 0.0063)

scaled_y <- theoretical_y *
    max(prof_y[which(prof_x > 844)[1]:which(prof_x >
844.55)[1]])

picked_peaks <-
Spectra::pickPeaks(filterMzRange(prof_prec_scan,

mz=c(844.4,844.55)))
picked_x <- mz(picked_peaks)[[1]]
picked_y <- intensity(picked_peaks)[[1]]
```

Plotting the theoretical and the picked values shows the accuracy of the profile spectra peak picking (Figure 4.10):

```
p_picked_theory <- ggplot() +
    coord_cartesian(xlim=c(844.425,844.475),
ylim=c(0,1.5e7)) +
    geom_segment(aes(x=theoretical_x, y=scaled_y,
                     yend = 0, xend = theoretical_x),
                 linewidth = 0.5, color=pal$darkorange)
+
    annotate("text", x = 844.443, y = 2.5e6,
color=pal$darkorange,
             label = "Theoretical\n844.43864") +
    geom_segment(aes(x=picked_x, y=picked_y,
                     yend = 0, xend = picked_x),
                 linewidth = 0.5, color=pal$blue) +
    annotate("text", x = 844.434, y = 2.5e6,
color=pal$blue,
             label = "Picked\n844.43841") +
    geom_line(aes(x=prof_x, y=prof_y)) +
    geom_point(aes(x=prof_x, y=prof_y), shape=1 ) +
    xlab("m/z") +
    ylab("Intensity") +
    theme_classic() +
    theme(plot.title = element_text(hjust = 0.5)) +
    theme(plot.subtitle = element_text(hjust = 0.5)) +
    ggtitle(label = "MSV000086195 Batch 1 Fraction 5
(Scan 21515)",
            subtitle = "MS1 APLDNDIGVSEATR 2+
(Deaminidated N, TMT10plex)")

print(p_picked_theory)
```

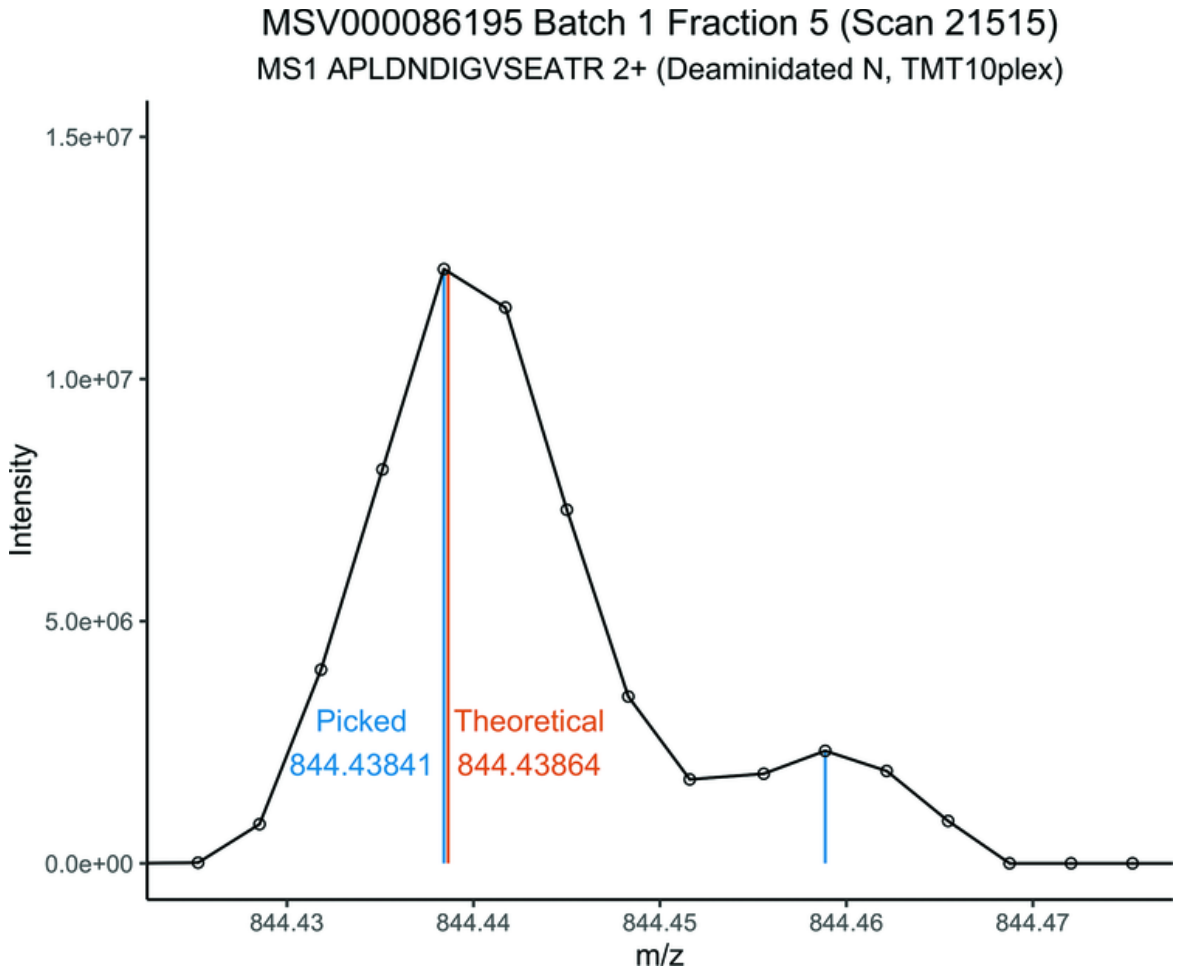


Figure 4.10 Comparing picked m/z values with theoretical monoistopic values.

4.3.3 Binning and Regularization Techniques

The plot shown in Figure 4.10 is for a single scan (21 515) which according to the summary tibble `psm_batch1_fraction5` has real values for both retention time and molecular weights. Here I use the `t()` (transpose) function from the `Matrix` package to turn a row into a column.

```
Matrix::t(psm_batch1_fraction5[1,])
```

```
##                  [,1]
```

```
## seq             "APLDNDIGVSEATR"
## charge          "2"
## ex_mz           "844.4393"
## calc_mz         "844.4386"
## match_score     "54.5"
## exp_score       "1.8e-10"
## scan            "21516"
## rt              "4795.983"
## mods            "229.1629 (1), 0.984 (5)"
## precursor_scan "21515"
```

The retention time of this scan is `4795.983` seconds and the experimentally observed m/z value is `844.4393`. While these exact values are useful for calculations, it is often useful to bin and regularize multidimensional data as part of exploratory data analysis.

The EDA technique that I will use next is a type of heatmap. For some data sets, a basic heatmap plot supplied by `MSnbase` is sufficient. Like many of the plots shown so far, it is sometimes necessary to do a little more work to get a better view of the data.

I'll start with the `MSmap` function from the `MSnbase` package following the example from the package vignette [102] on Bioconductor. The code below generates Figure 4.11.

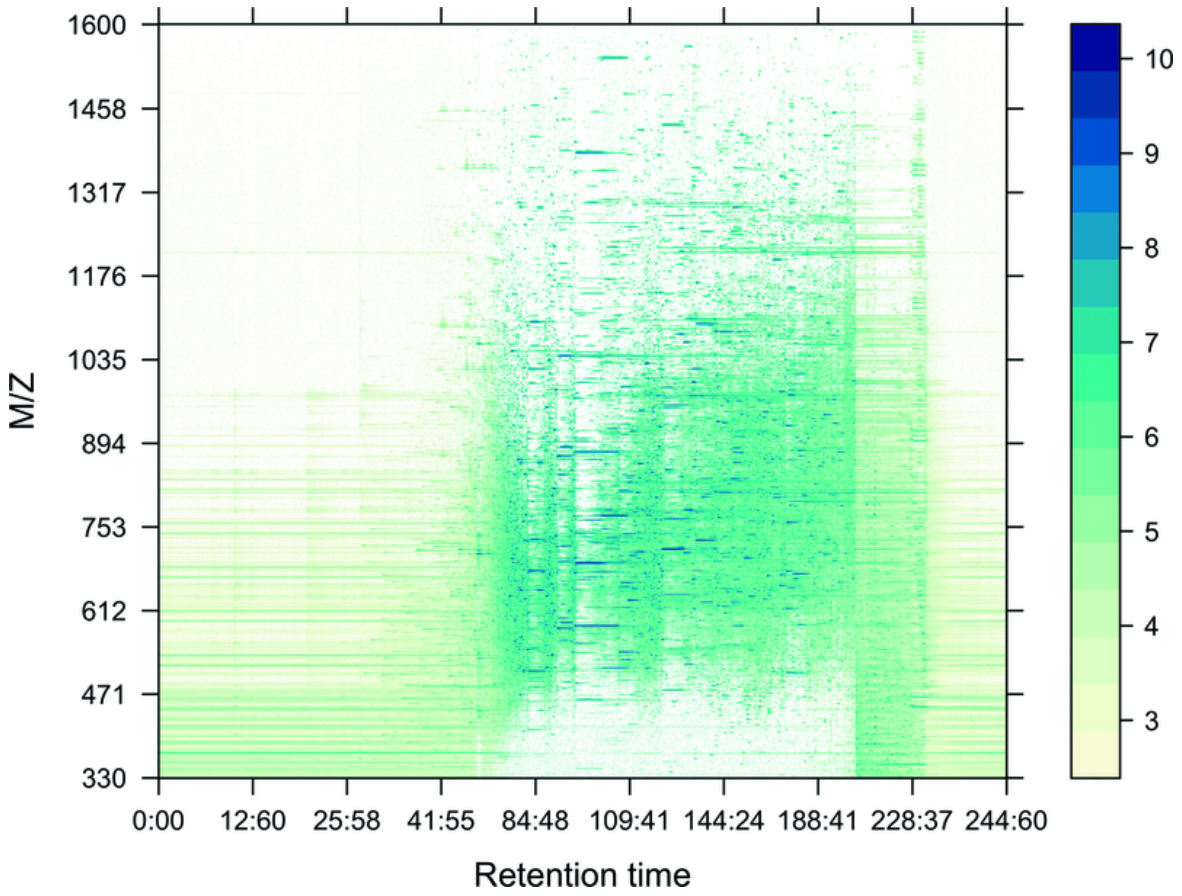


Figure 4.11 Heatmap from MSmap using 1 Da binning.

```
file_name <- file.path("large-
data","MSV000086195","ccms_peak",

"ScltlMsclsMAvsCntr_Batch1_BRPhsFr5.mzML")

# read the mzML file and extract the header
ms <- openMSfile(file_name)
hd <- header(ms)

ms1 <- which(hd$msLevel == 1)

rtsel <- hd$retentionTime[ms1] > 0 &
    hd$retentionTime[ms1] < 14700

## make a map from 330 to 1600 m/z using a 1 Da bin
size
```

```
M <- MSmap(ms, ms1[rtsel], 330, 1600, 1, hd, zeroIsNA =
TRUE)

plot(M, aspect = "fill", allTicks = FALSE)
```

This quick plot, while likely useful in some situations, suffers from two problems. First, the intensities of peaks seem to range from 0 to about 10^{10} which hides most of the details of the run. Second, while some customization can be done to this plot it uses the older R graphics system called `lattice` [103], which has been largely replaced by `ggplot2` and so I'm not going to cover it in this book. Instead, I will show you how to generate a heatmap that exactly meets your needs and can be customized using the `ggplot2` system.

Instead of using `MSnbase` directly, I will use the `Spectra` package to load, filter, and bin the spectral data. I will then use a custom function to bin the time axis to create a three-dimensional data set where the value for each bin is the ion intensity. Once I have a grid, I can use the `ggplot2` geometry layer `geom_raster()` to plot it. Binning can be thought of as turning the continuous values of the time and m/z axis into categorical variables. The axes can still be labeled with continuous values, but in reality, these are simply ordinal, rather than continuous values. The ion intensity is still handled as a continuous variable. By using the scaling functions of `ggplot2` I can control the dynamic range of the plot to give the best overview.

The following three issues will need to be decided for this type of data visualization:

1. What is the desired bin width for m/z?
2. What is the bin width for retention time?
3. What dynamic range for the intensity gives the most information about the data set?

The first step in this analysis will be to get the m/z and retention time ranges.

```
ms1_spectra <- file.path("large-
data","MSV000086195","ccms_peak",

"ScltlMsclsMAvsCntr_Batch1_BRPhsFr5.mzML") |>
    Spectra() |>
    filterMsLevel(msLevel=1)

ms1_spectra_count <- length(ms1_spectra)
print(paste("File contains", ms1_spectra_count,
"spectra"))
```

```
## [1] "File contains 19081 spectra"
```

There are 19 081 MS level 1 spectra in the batch 1, fraction 5 run, and they have been loaded from the mzML file filtered into `Spectra` object. Now the start and end m/z for these spectra can be found using the `mz()` accessor function for the `Spectra` class. The `mz()` function returns a list of all the m/z values for all of the spectra in a list of numeric arrays. Calling `min()` on the result of `mz()` produces an array holding the minimum value in each of the arrays in the original list. To find the lowest m/z value in the entire run, `min()` is called twice on is array. The same is done to find the single highest value using two calls to `max()`.

```
start_mz <- mz(ms1_spectra) |>
            min() |>    # array of minimum m/z for each
spectrum
            min()       # lowest m/z for all spectra

end_mz <- mz(ms1_spectra) |>
            max() |>  # array of maximum m/z for each
spectrum
            max()     # lowest m/z for all spectra

print(paste0("Start m/z: ", format(start_mz, digits=1,
nsmall=1),
             " End m/z: ", format(end_mz, digits=1,
nsmall=1)))
```

```
## [1] "Start m/z: 330.0 End m/z: 1600.0"
```

The `format()` function cleans up the floating point values and the m/z range is 330–1600 as indicated in the method section of the manuscript. Next, I'll get the retention time range in the same way.

```
start_rt <- rtime(ms1_spectra) |>
            min() |>    # array of minimum m/z for each
spectrum
            min()       # lowest m/z for all spectra

end_rt <- rtime(ms1_spectra) |>
            max() |>    # array of maximum m/z for each
spectrum
            max()       # lowest m/z for all spectra

print(paste0("Start RT (s): ", format(start_rt,
digits=1, nsmall=1),
             " End RT (s): ", format(end_rt, digits=1,
nsmall=1)))
```

```
## [1] "Start RT (s): 0.3 End RT (s): 14700.3"
```

The manuscript indicated that the gradient was run for 195 minutes (11 700 seconds), and a rough calculation of the time for the sample to get to the instrument is approximately 50–60 minutes. Based on the longest acquisition time, it appears that 50 minutes (3000 seconds) were added to the run, which would mean that the acquisition was started as soon as the sample was loaded and the first 3000 seconds should not contain any of the internal control peptides. When customizing the plot to look at different regions in more detail, the `mz_range` and `rt_range` variables will be used to zoom in on the desired ranges.

```
# full ranges from examination of raw data

mz_range <- c(330,1600)
rt_range <- c(0,14700)

ms1_spectra_filtered <- ms1_spectra |>
    filterMzRange(mz_range) |>
    filterRt(rt_range)
```

The `Spectra` package has a `bin()` function that can be used to bin the m/z spectra into equally spaced bins. I'll use that function to bin the mass spectra. Here is where you can control the degree of binning. First, I'll look at the entire run. Binning the m/z values into 1 Da bins like in Figure 4.11 is done by setting the `binSize=1` argument to `Spectra::bin()`.

```
  # start with a m/z bin width of 1 Da
  mz_bin_size <- 1

  # Bin the mass spectra into mz_bin_size bin widths.
  # Keep the zero entries (zero.rm=FALSE)
  # The default function for the combination of spectra
  is sum().
  # Other function could be used with the argument
  FUN=max()

  binned_intesities_list <-
  Spectra::intensity(Spectra::bin(ms1_spectra_filtered,

  binSize = mz_bin_size,

  zero.rm=FALSE))

  print(paste0("Minimum Intensity: ",

  format(min(unlist(binned_intesities_list@listData)),
                      scientific = TRUE)))
```

```
## [1] "Minimum Intensity: 0e+00"
```

```
  print(paste0("Maximum Intensity: ",

  format(max(unlist(binned_intesities_list@listData)),
                      scientific = TRUE)))
```

```
## [1] "Maximum Intensity: 1.037875e+10"
```

This confirms the problem with the simple plot, 10^{10} is an extremely wide range of intensity values to display on one plot. Especially, given that the maximum intensity observed in Figure 4.8 is below 1.5×10^7.

The next step is to convert the binned intensity lists into a `tibble` that can be used by `ggplot2`. The easiest way to do this

is to create a named matrix, which is just a matrix that has row and column names. In this case, the column names will be the m/z values of the binned m/z axis, and the row names will just be an index.

```
binned_spectra_matrix <-
matrix(unlist(binned_intesities_list),
            nrow = length(binned_intesities_list),
            byrow=TRUE)

# column names will be character type
# they will have to be converted to numeric for
plotting

colnames(binned_spectra_matrix) <-
    head(seq(mz_range[1],mz_range[2],mz_bin_size), -1)

rownames(binned_spectra_matrix) <-
    c(1:(length(binned_spectra_matrix[,1])))
```

To bin the retention times of the MS level 1 spectra, I have to compare all of the binned m/z vectors (rows) in the retention time window. This could be done with two `for` loops, one for each time window, and another for each element in the spectrum arrays. It is common for this kind of nested loop to run quite slowly. However, before adding a parallel processing tool, which will increase the difficulty in understanding and debugging, it is worth it to do some timing tests to see if the process you are guessing is actually worth optimizing.

4.3.3.1 Binning Retention Time: Evaluating Algorithms

I'll walk through the process of evaluating the speed and resource consumption of algorithms–called *code profiling*–to show the speed differences between parallel and single step comparison.

```
spectra_tibble <- as_tibble(binned_spectra_matrix) |>
    mutate(rt=rtime(ms1_spectra_filtered),
.before=as.character(mz_range[1]))
```

One approach to binning the retention time axis is to first assign all of the spectra in each time window a retention time for that time bin.

```
rt_bin <- 10    # For this test use a rt bin of 10
seconds
current_rt <- spectra_tibble$rt[1]
next_bin <- current_rt +   rt_bin     # initialize
next_bin

for(i in 1:length(spectra_tibble$rt)) {
    if(spectra_tibble$rt[i] < next_bin) {
        spectra_tibble$rt[i] <- current_rt
    } else {
        current_rt <- next_bin
        next_bin <- current_rt + rt_bin
        spectra_tibble$rt[i] <- current_rt
    }
}

# make a smaller tibble of just the first time window
spectra_tibble <- spectra_tibble[spectra_tibble$rt ==
spectra_tibble$rt[1],]
print(paste0("There are ",length(spectra_tibble$rt),
             " spectra in the first time bin."))
```

```
## [1] "There are 25 spectra in the first time bin."
```

Next, I'll use the `Sys.time()` function to get the time at the start of the code I'm interested in timing. I'll then get the time at the end of the section and subtract it to get the run time. That time will be the approximate time to bin the first-time bin which has 25 spectra according to the analysis above.

```
current_spectrum <- spectra_tibble[1,]
n_iterations <- length(spectra_tibble$rt)

start_time <- Sys.time()

for(i in 1:n_iterations) {
    for(j in 2:length(spectra_tibble)) {
        if(spectra_tibble[i,j] > current_spectrum[1,j])
{
            current_spectrum[1,j] <-
spectra_tibble[i,j]
        }
    }
}

end_time <- Sys.time()
print(end_time - start_time)
```

```
## Time difference of 1.699873 secs
```

Instead of a second loop, I can use the `pmax()` function to produce a single numeric vector containing the largest values in each position of the vector. This cannot be directly converted to a tibble without providing column names, so a single row matrix is created, column names are added and the named matrix is converted to a tibble.

```
current_spectrum <- spectra_tibble[1,]
n_iterations <- length(spectra_tibble$rt)

start_time <- Sys.time()

for(i in 1:n_iterations) {
    acquisition <- spectra_tibble[i,]

    highest_values <-
pmax(as.numeric(current_spectrum),

as.numeric(acquisition))

    matrix_row <- matrix(t(highest_values), nrow = 1)
    colnames(matrix_row)<-c('rt',

head(seq(mz_range[1],mz_range[2],mz_bin_size), -1))

    current_spectrum <- as_tibble(matrix_row)
}

end_time <- Sys.time()
print(end_time - start_time)
```

```
## Time difference of 0.1073031 secs
```

This analysis confirms that the `pmax()` approach is about 10 times faster than the nested loop. An estimated run time for the entire data set is at most 5 minutes, compared to approximately 51 minutes.

4.3.3.2 Plotting a Heatmap with `ggplot2` Using Binned Data

Now, I will define a function `bin_rt()` which I'll use to bin the retention time axis in a pipeline. It will take a tibble with the retention time in a column and the binned m/z values as each of the other columns. The basic idea is to loop through the

tibble and gather a single spectrum for each time bin. That spectrum will contain the largest intensity values for each m/z value.

```
bin_rt <- function(unbinned_spectra, rt_binsize=1) {

    current_rt <- unbinned_spectra$rt[1]
    rt_bin <- rt_binsize  # width of retention time bin
(s)
    next_bin <- current_rt + rt_bin # initialize
next_bin

    # bin the retention time first
    for(i in 1:length(unbinned_spectra$rt)) {
        if(unbinned_spectra$rt[i] < next_bin) {
            unbinned_spectra$rt[i] <- current_rt
        } else {
            current_rt <- next_bin
            next_bin <- current_rt + rt_bin
            unbinned_spectra$rt[i] <- current_rt
        }
    }

    # pick out the highest signal for each mass channel
for each retention time

    rt_binned_spectra <- unbinned_spectra[0,] # empty
accumulator
    current_spectrum <- unbinned_spectra[1,] # start
with the first row
    current_rt = unbinned_spectra$rt[1]

    # this could be a long running function - use a
progress bar (pb)

    n_iterations <- length(unbinned_spectra$rt)
    pb <- progress_estimated(n_iterations)

    for(i in 1:n_iterations) {

        update_progress(pb)

        acquisition <- unbinned_spectra[i,]
        rt <- acquisition$rt
        if(rt %==% current_rt) {
```

```
            highest_values <-
pmax(as.numeric(current_spectrum),

as.numeric(acquisition))

            # convert the numeric vector from pmax()
into a tibble
            # via a matrix with named columns

            matrix_row <- matrix(t(highest_values),
nrow = 1)
            colnames(matrix_row)<-c('rt',

head(seq(mz_range[1],mz_range[2],mz_bin_size), -1))

            current_spectrum <- as_tibble(matrix_row)

        } else {
            rt_binned_spectra <-
bind_rows(rt_binned_spectra, current_spectrum)
            current_spectrum <- unbinned_spectra[i,]
            current_rt <- unbinned_spectra$rt[i]
        }
    }
    rt_binned_spectra
}
```

Now the spectral table can be constructed.

```
rt_bin_size <- 10

spectra_table <- as_tibble(binned_spectra_matrix) |>
        mutate(rt=rtime(ms1_spectra_filtered),
               .before=as.character(mz_range[1])) |>
        bin_rt(rt_binsize = rt_bin_size) |>
        pivot_longer(cols=!"rt", names_to="mz",
values_to = "intensity") |>
        mutate(mz=as.numeric(mz))
```

With the `spectra_table` tibble constructed, I can use it to plot a `ggplot2` based heatmap. The `geom_raster()` layer expects data to contain only the three variables to be plotted, so the `pivot_longer()` function was used in the pipeline after binning, to leave only `rt`, `mz`, and `intensity`. Since the `mz` values are derived from the column names, they have to be converted to `numeric` using the `mutate()` function.

Along with using the `geom_raster()` layer, I will customize the plot further by choosing a vivid and accessible color scheme, and limiting the `intensity` scale based on the range of the data observed. Since a high-quality spectrum for a peptide was observed just below 2×10^7, I'll set that for the highest intensity to be plotted. This will set any intensity above the limit to `NA`. The `scale_fill_viridis()` layer uses the vivid *viridis* color palette, which like the Okabe-Ito palette as mentioned in Chapter 1, is a highly accessible color palette. In the heatmap plot, I want any `NA` value to appear `white`.

```
p_heatmap <- spectra_table |>
    ggplot() +
    geom_raster(aes(x = rt, y = mz, fill = intensity))
+
    scale_fill_viridis_c(limits=c(0.0, 2e7), na.value =
"white",
                         option = "plasma") +
    coord_cartesian(expand = FALSE) +
    scale_x_continuous(breaks = seq(0,14700,1000)) +
    scale_y_continuous(breaks = seq(300,1600,100)) +
    theme_classic() +
    theme(plot.title = element_text(hjust = 0.5)) +
    theme(plot.subtitle = element_text(hjust = 0.5)) +
    ggtitle(label = "MSV000086195 Batch 1 Fraction 5",
            subtitle = "Binning: m/z=1 rt=10s")

print(p_heatmap)
```

Figure 4.12 gives a good overview of the data you are working with in this project. It shows the roughly 50-minute

delay between the acquisition start and the end of the gradient. It also shows that some compounds show up as long streaks, meaning they are not real chromatographic signals, but probably contaminates that *bleed* over a long period of time.

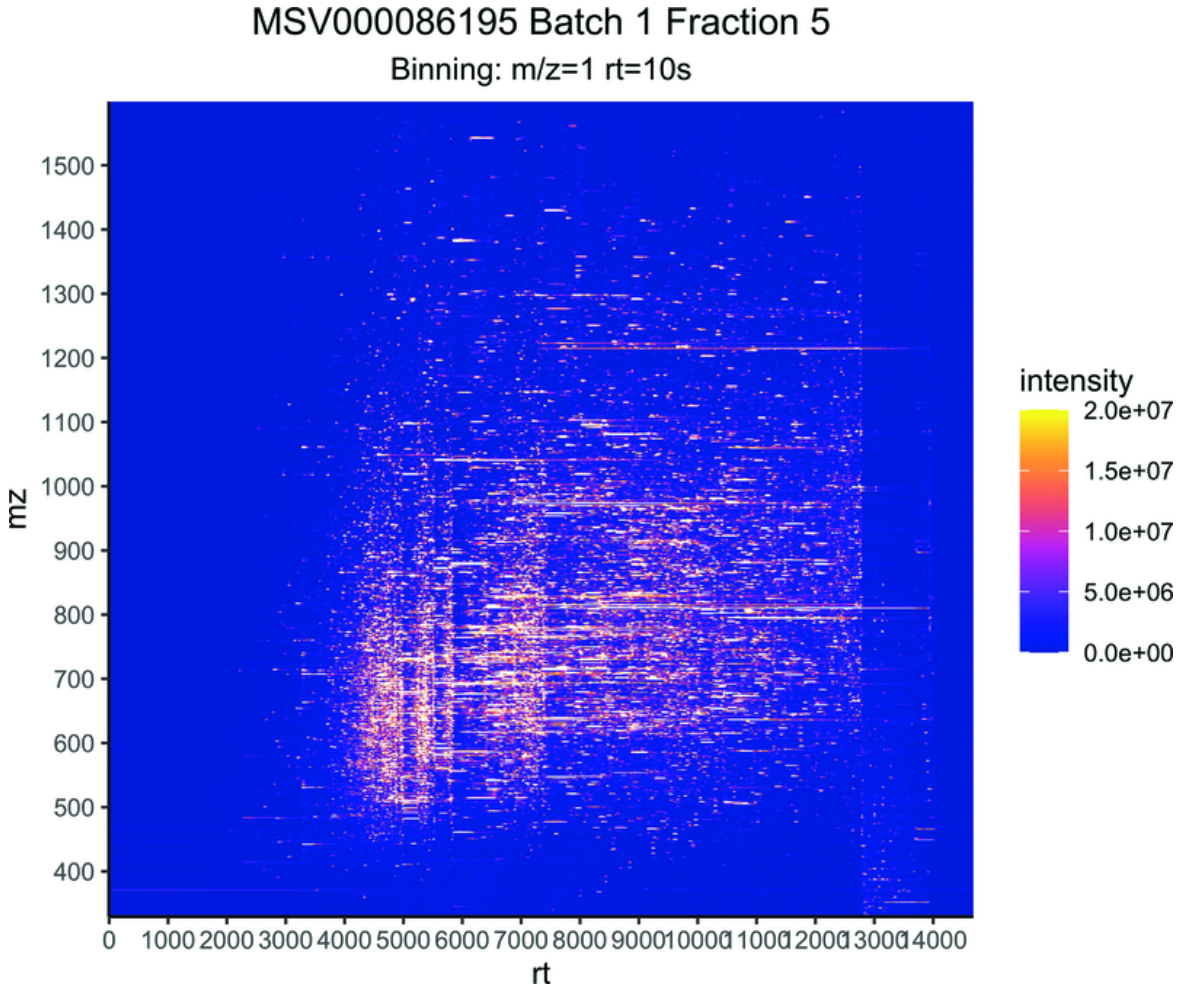


Figure 4.12 Heatmap for Batch 1 Fraction 5.

To take a closer look at the region around the peptide from Figure 4.8, I can zoom in on that region with smaller bins. Before repeating the code for the heatmap, it's worth pulling a big, repeated chunk into a function. Since every different zoom level will call the same code to get the spectra_table tibble, I simply moved all this code into a function called `get_spectra_table()`. This is called *refactoring*. As refactoring goes, this is a pretty simple step. It could be improved by

providing reasonable defaults for ranges, and bin sizes, but since those are critical, I would rather have the function fail than do something odd.

```
get_spectra_table <- function(ms1_spectra, mz_range,
rt_range,
                              mz_bin_size, rt_bin_size)
{

    ms1_spectra_filtered <- ms1_spectra |>
        filterMzRange(mz_range) |>
        filterRt(rt_range)

    binned_intesities_list <-
intensity(Spectra::bin(ms1_spectra_filtered,

binSize = mz_bin_size,

zero.rm = FALSE))

    binned_spectra_matrix <-
matrix(unlist(binned_intesities_list),
           nrow = length(binned_intesities_list),
           byrow=TRUE)

    colnames(binned_spectra_matrix) <-
        head(seq(mz_range[1],mz_range[2],mz_bin_size),
-1)

    rownames(binned_spectra_matrix) <-
        c(1:(length(binned_spectra_matrix[,1])))

    #return the spectra_table
    as_tibble(binned_spectra_matrix) |>
        mutate(rt=rtime(ms1_spectra_filtered),
               .before=as.character(mz_range[1])) |>
        bin_rt(rt_binsize = rt_bin_size) |>
        pivot_longer(cols=!"rt", names_to="mz",
values_to = "intensity") |>
        mutate(mz=as.numeric(mz))
}
```

Now, this “helper function” can be used at each different zoom level such that only the ranges and the specifics of the

plot are changed from one region or zoom level to another.

```
mz_bin_size <- 0.1
rt_bin_size <- 2

mz_range <- c(840,850)
rt_range <- c(4700,4900)

p_map_zoom_1 <- get_spectra_table(ms1_spectra,
mz_range, rt_range,
                                  mz_bin_size,
rt_bin_size) |>
    ggplot() +
    geom_raster(aes(x = rt, y = mz, fill = intensity))
+
    scale_fill_viridis_c(limits=c(0.0, 1e7),
                         na.value = "white", option =
"plasma") +
    coord_cartesian(ylim=c(840,850), expand = FALSE) +
    scale_x_continuous(breaks = seq(4700,4900,25)) +
    scale_y_continuous(breaks = seq(840,850,1)) +
    annotate("segment", x=4795.983, y=850, yend = 0,
xend = 4795.983,
                 linewidth = 0.25, linetype="dotted",
color="white") +
    theme_classic() +
    theme(plot.title = element_text(hjust = 0.5)) +
    theme(plot.subtitle = element_text(hjust = 0.5)) +
    ggtitle(label = "MSV000086195 Batch 1 Fraction 5",
            subtitle = "Binning: m/z=0.1 rt=2s")
```

```
print(p_map_zoom_1)
```

From the zoomed-in view shown in Figure 4.13, it appears that there is more going on around m/z 844 than just the isotopes. There is a larger peak which appears to elute slightly earlier with the same charge state but higher intensity. Repeating the process for a tighter zoom requires

not just zooming in on the coordinates, but also changing the binning resolution.

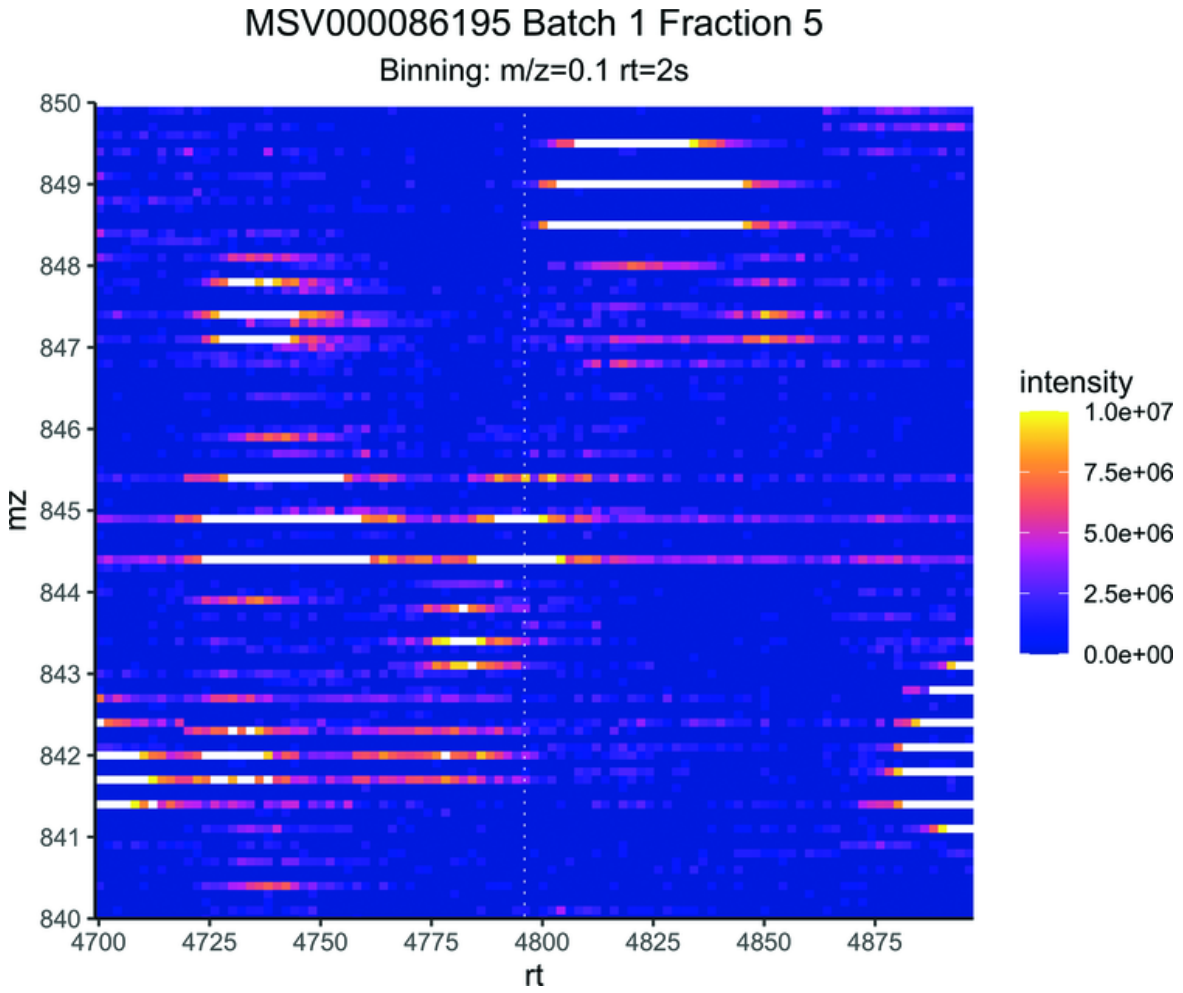


Figure 4.13 Zoomed in heatmap for the peptide at m/z 844.4 and retention time 4796 using m/z bin size of 0.1 and retention time bin of 2 seconds.

```
mz_bin_size <- 0.01
rt_bin_size <- 2

mz_range <- c(844,846)
rt_range <- c(4600,4900)

p_map_zoom_2 <- get_spectra_table(ms1_spectra,
mz_range, rt_range,
                                  mz_bin_size,
rt_bin_size) |>
    ggplot() +
    geom_raster(aes(x = rt, y = mz, fill = intensity))
```

```
  +
      scale_fill_viridis_c(limits=c(0.0, 5e6), na.value =
  "white",
                           option = "plasma") +
      coord_cartesian(ylim=c(844,846), expand = FALSE) +
      scale_x_continuous(breaks = seq(4600,4900,25)) +
      scale_y_continuous(breaks = seq(844,846,0.25)) +
      annotate("segment", x=4795.983, y=846, yend = 0,
  xend = 4795.983,
                   linewidth = 0.25, linetype="dotted",
  color="white") +
      theme_classic() +
      theme(plot.title = element_text(hjust = 0.5)) +
      theme(plot.subtitle = element_text(hjust = 0.5)) +
      ggtitle(label = "MSV000086195 Batch 1 Fraction 5",
              subtitle = "Binning: m/z=0.01 rt=2s")
```

```
  print(p_map_zoom_2)
```

To see the effects of binning, Figure 4.14 zooms in closer on the `844.4393` range. Here it becomes evident that the monoisotopic mass is shifting between bins. Notice that all four isotopes observed for this peptide have a more intense peak earlier in the chromatogram. The signal is more intense for all four isotopes at approximately `4740` seconds, however since the acquisition method specified a dynamic exclusion time of 70 seconds, it is possible that the peak was first included for MS/MS some 70 seconds earlier (maybe around 4670). This will be more evident when you look at the extracted ion chromatogram for this precursor ion.

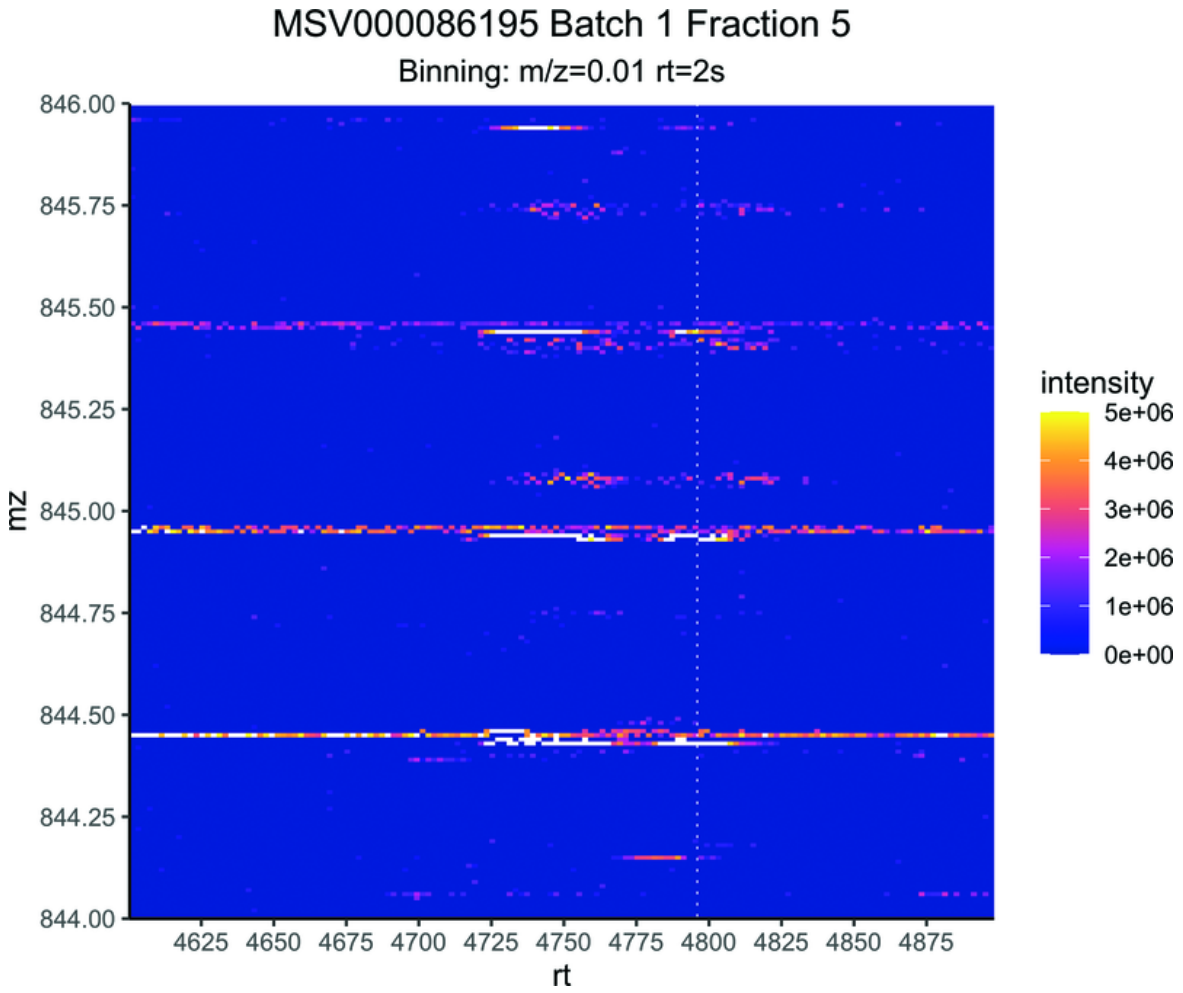


Figure 4.14 Zoomed in heatmap for the peptide at m/z 844.4 and retention time 4796 using m/z bin size of 0.01 and retention time bin of 2 seconds.

Zooming in closer will help in deciding how to create an extracted ion chromatogram from this data.

```
mz_bin_size <- 0.001
rt_bin_size <- 2

mz_range <- c(844,845)
rt_range <- c(4700,4825)

p_map_zoom_3 <- get_spectra_table(ms1_spectra,
mz_range, rt_range,
                                  mz_bin_size,
rt_bin_size) |>
    ggplot() +
        geom_raster(aes(x = rt, y = mz, fill =
intensity)) +
        scale_fill_viridis_c(limits=c(0.0, 5e6),
na.value = "white",
                             option = "plasma") +
        coord_cartesian(ylim=c(844.4,844.5), expand =
FALSE) +
        scale_y_continuous(breaks =
seq(844.4,844.5,0.01)) +
        scale_x_continuous(breaks = seq(4700,4825,25))
+
        annotate("segment", x=4795.983, y=845, yend =
0, xend = 4795.983,
                 linewidth = 0.25, linetype="dotted",
color="white") +
        theme_classic() +
        theme(plot.title = element_text(hjust = 0.5)) +
        theme(plot.subtitle = element_text(hjust =
0.5)) +
        ggtitle(label = "MSV000086195 Batch 1 Fraction
5",
                subtitle = "Binning: m/z=0.001 rt=2s")
```

```
print(p_map_zoom_3)
```

In Figure 4.15, it's clear that the exact mass value shifts slightly across the peak and that the side peak seen in Figure 4.9 is some kind of contaminate that is spread across the

entire region of interest, as opposed to a compound related to the internal control peptide, or an FTMS artifact.

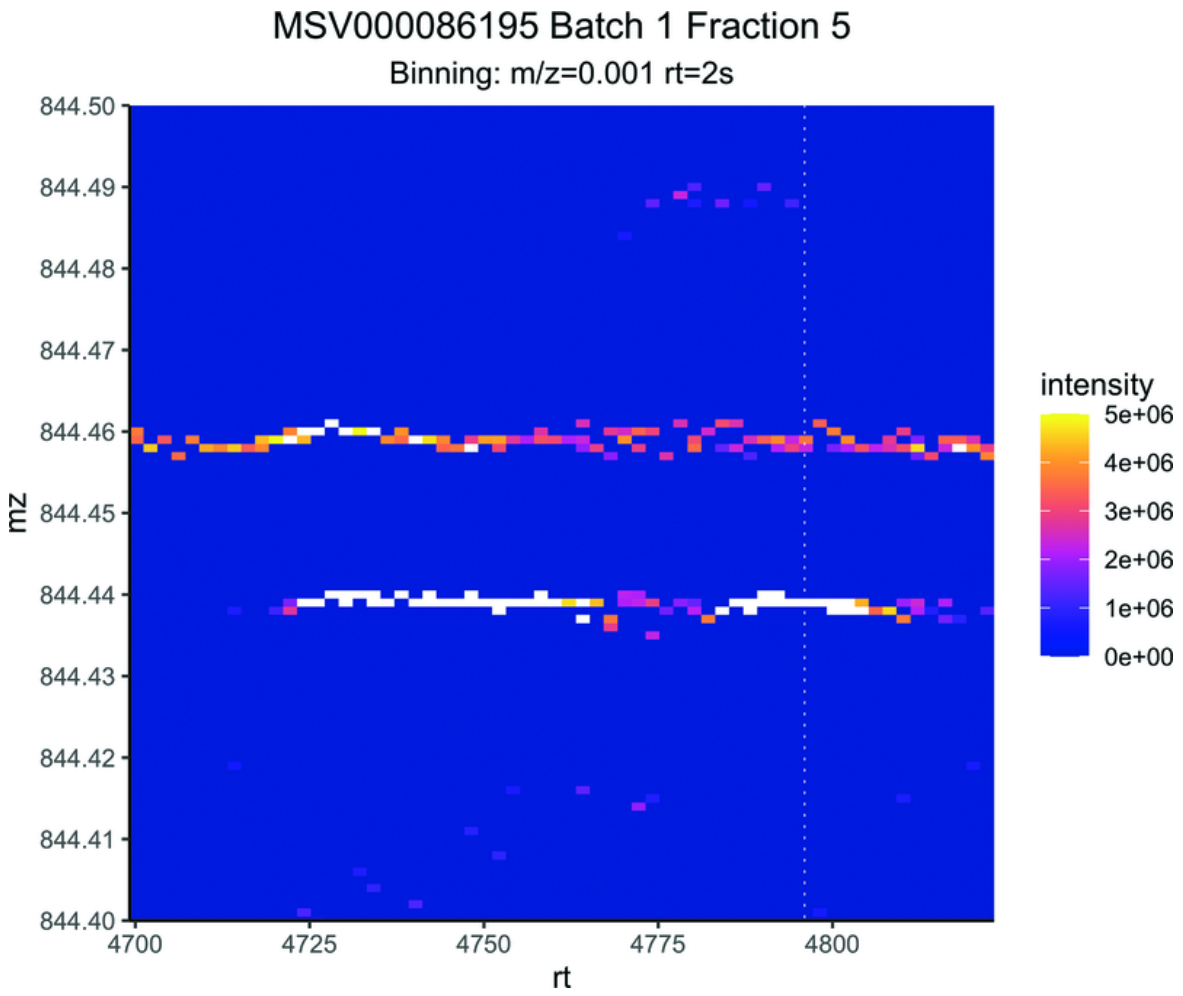


Figure 4.15 Zoomed in heatmap for the peptide at m/z 844.439 and retention time 4796 using m/z bin size of 0.001 and retention time bin of 2 seconds.

Figure 4.15 also shows that when thinking about constructing an extracted ion chromatogram, there is a mass range between 844.43 and 844.45 that is likely to include the intensity maximums of each scan for the profile spectrum shown in Figure 4.10. Despite having high mass resolution the small movements of the centroid mean that setting the window for an XIC to tight could miss the ion signal which is clearly part of the analyte. Setting the window to wide risks adding contaminating current from another compound which

is only slightly different in m/z and present throughout the retention time of the ion of interest.

This example shows the value of exploratory data analysis. Before diving into hypothesis testing, you now know the limits of what your data can deliver. In the Section 4.4, I will show you how to look at data from a chromatographic point of view, both when data is collected in full scan mode and extracted ion chromatograms can be created, and selected reaction monitoring, when the ion of interest has already been chosen and all that is available in the chromatogram.

4.4 Chromatograms and Other Chemical Separations

While mass spectrometers are selective detectors, there are many different compounds that have the exact same chemical formula and so will produce a signal at the exact same m/z value regardless of the resolution of the instrument. Often this *isobaric interference* can be dealt with using a second stage of mass spectrometry as I showed in the example of peptides which have both a precursor m/z and a post-fragmentation product m/z. However, so-called *tandem mass spectrometers* are much more complicated and usually more expensive than single-stage instruments. Prior to the commercialization of tandem mass spectrometers, isobaric interferences had to be dealt with by using a chemical separation. Early in the commercialization of mass spectrometry chemical separations were performed using **gas chromatography** (GC) which uses an oven to heat a capillary which acts as a stationary phase, and a flowing gas which acts as a mobile phase. Compounds with high volatility tend to stay in the mobile phase longer, interacting with the stationary phase less often, and so move through the capillary faster than compounds with low volatility. The difference in time spent in the mobile phase produces a

separation based on a different chemical property than molecular weight and so adds to the selectivity of the mass spectrometer. The first commercial chromatography coupled mass spectrometry instruments (GC/MS) used this approach, and are still in widespread use in many areas of chemical and trace chemical analysis.

After the commercialization of atmospheric pressure ionization (API) sources, **liquid chromatography** (LC) became the main method of chemical separation coupled with mass spectrometry. In LC, the mobile phase is a liquid mixture and the stationary phase can be constructed using a wide range of materials to take advantage of different chemical properties of target molecules to obtain a separation prior to mass spectrometry detection. Even with the additional level of selectivity added by LC, many mass spectrometry labs use a combination of LC and tandem mass spectrometry (LC-MS/MS) to achieve both a very high level of selectivity and to improve sensitivity. In this Section, I show how to visualize two types of chromatograms. When LC/MS or GC/MS is performed in such a way that the full MS spectrum is collected, then using the m/z binning idea, an *extracted ion chromatogram* (abbreviated either as EIC or XIC) can be plotted by extracting a set of m/z values to plot with intensity on the y-axis and retention time on the x-axis. In LC-MS/MS, the mass spectrometer can be configured to fix the precursor m/z and fix the product m/z so that a specific ion reaction (precursor ion to product ion) can be recorded, creating a SRM chromatogram.

First, continuing with the example from Section 4.3, I'll show how to extract the chromatogram for the ions shown in Figure 4.14.

4.4.1 Extracted Ion Chromatograms

To produce extracted ion chromatograms, I'll go back to the `MSnbase` package rather than the `Spectrum` package. The `MSnbase` package provides the `chromatogram()` function to filter the MS level and m/z range.

```
# read, filter and select the raw data to create an xic
using the maximum
# m/z intensity in the mass range as the value of the
chromatogram intensity

file_name <- file.path("large-data",
"MSV000086195","ccms_peak",

"ScltlMsclsMAvsCntr_Batch1_BRPhsFr5.mzML")

ms_level_1 <- readMSData(file_name, mode = "onDisk") |>
                filterMsLevel(msLevel = 1)

mz_xic_peptide <- ms_level_1 |>
                chromatogram(mz = c(844.43, 844.45),
aggregationFun = "max")

# the chromatogram() function returns a type that can
hold an array of
# chromatograms, but only the first one is populated

mz_range <- mz(mz_xic_peptide[1])

# tidy up the data for plotting

xic_chrom_peptide <- mz_xic_peptide[1] |>
    (function(x) {tibble(rtime(x),
replace_na(intensity(x),0))})() |>
    setNames(c('rt', 'inten'))
```

With the XIC computed, it can now be plotted:

```
p_xic <- xic_chrom_peptide |>
    ggplot() +
    geom_line(aes(x=rt, y=inten)) +
    scale_y_continuous(labels = inten_label) +
    scale_x_continuous(breaks = seq(0,14700,1000)) +
    geom_segment(aes(x=4795.983, y=max(inten),
                     yend = 0, xend = 4795.983),
                 linewidth = 0.25, linetype="longdash",
color=pal$darkorange) +

    xlab("Retention Time (s)") +
    ylab("Intensity (counts)") +
    theme_classic() +
    theme(plot.title = element_text(hjust = 0.5)) +
    theme(plot.subtitle = element_text(hjust = 0.5)) +
    ggtitle(label = "MSV000086195 Batch 1 Fraction 5",
            subtitle = paste0("MS Level 1: ",
                              sprintf("%.2f - %.2f",
mz_range[1], mz_range[2])))

print(p_xic)
```

In Figure 4.16, the dashed line shows the retention time for the spectrum shown in Figure 4.9. Obviously there are more ions that show up in this mass range at different times, especially the intense peaks around 10 000 seconds. I'd also like to see if the small adjacent ion seen in Figure 4.9 is an artifact of the peptide ion, or a chromatographically unrelated ion. Plotting a different XIC will help evaluate the overlapping m/z ion (Figure 4.16).

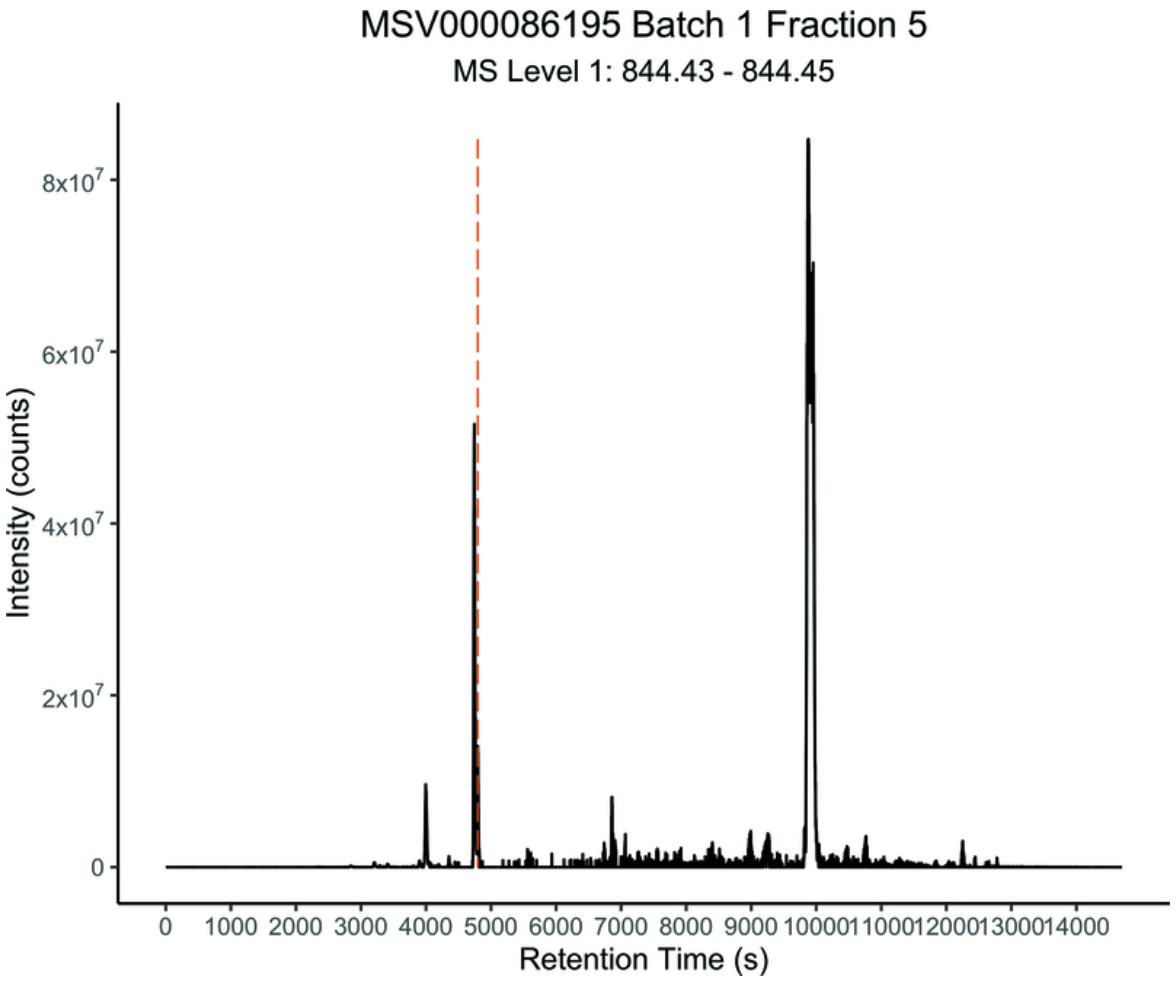


Figure 4.16 Extracted ion chromatogram for the ions between 844.43 and 844.45.

```
# read, filter and select the raw data to create an xic
using the maximum
# m/z intensity in the mass range as the value of the
chromatogram intensity

mz_xic_interference <- ms_level_1 |>
            chromatogram(mz = c(844.45, 844.46),
aggregationFun = "max")

# the chromatogram() function returns a type that can
hold an array of
# chromatograms, but only the first one is populated

mz_range <- mz(mz_xic_interference[1])
```

```
# tidy up the data for plotting

xic_chrom_interference <- mz_xic_interference[1] |>
    (function(x) {tibble(rtime(x),
replace_na(intensity(x),0))})() |>
    setNames(c('rt', 'inten'))
```

If the overlapping ion has the same chromatographic properties, it will have the same shape as the target analyte:

```
p_interference <- xic_chrom_interference |>
    ggplot() +
        geom_line(aes(x=rt, y=inten)) +
        scale_y_continuous(labels = inten_label) +
        scale_x_continuous(breaks = seq(0,14700,1000))
+
        geom_segment(aes(x=4795.983, y=max(inten),
                         yend = 0, xend = 4795.983),
                         linewidth = 0.25,
linetype="longdash",
                         color=pal$darkorange) +
        xlab("Retention Time (s)") +
        ylab("Intensity (counts)") +
        theme_classic() +
        theme(plot.title = element_text(hjust = 0.5)) +
        theme(plot.subtitle = element_text(hjust =
0.5)) +
        ggtitle(label = "MSV000086195 Batch 1 Fraction
5",
            subtitle = paste0("MS Level 1: ",
                              sprintf("%.2f - %.2f",
mz_range[1], mz_range[2])))

print(p_interference)
```

From Figure 4.17 it's clear that overlapping ion is part of an ill-defined chromatographic artifact very early in the gradient. This region is sometimes called the *crash peak* because these compounds come "crashing through the

column” without being retained. They appear as long streaks in heatmaps and can complicate the analysis of mass spectra.

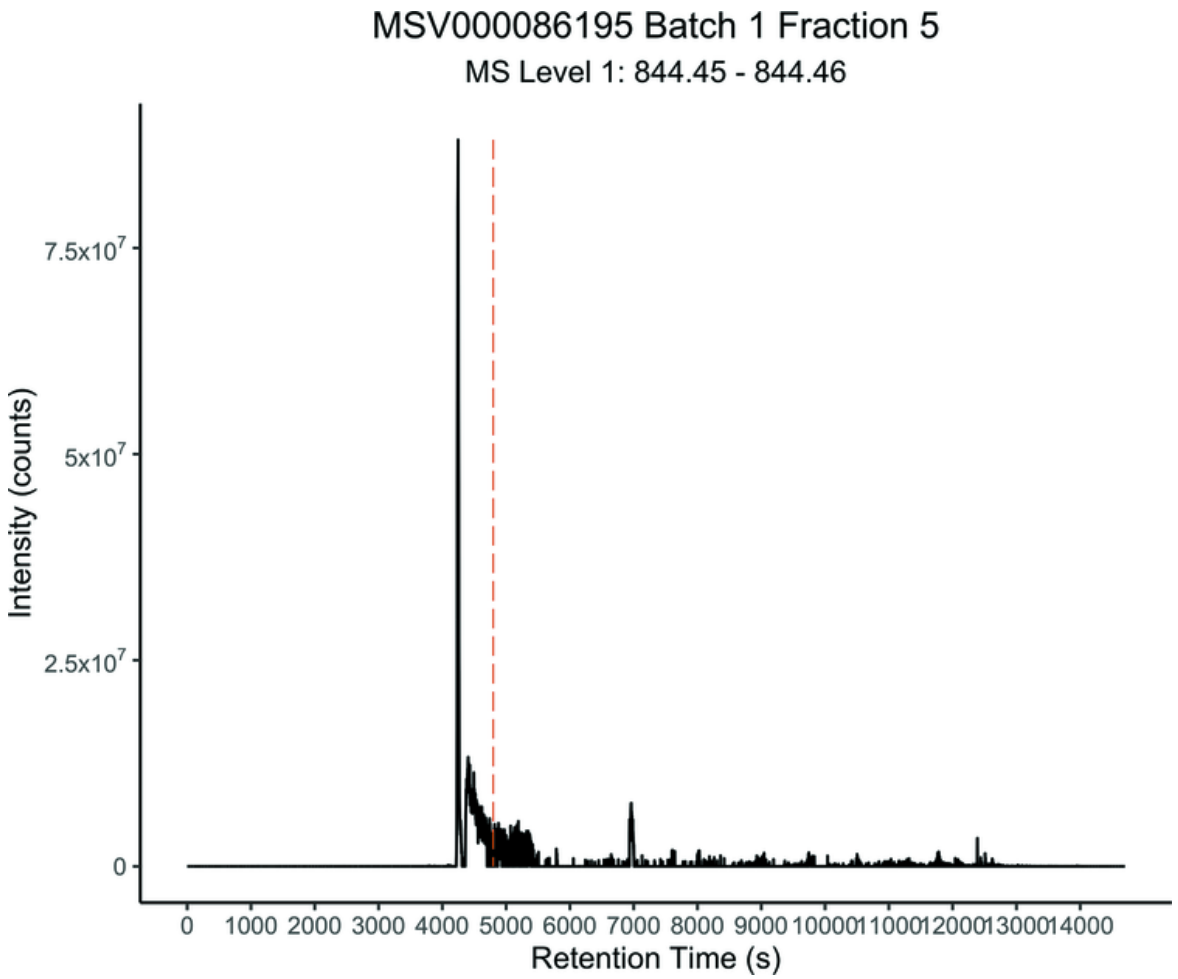


Figure 4.17 Extracted ion chromatogram for the ions between 844.43 and 844.45. The retention time window was narrowed to the region of interest.

I would like to look at this region in more detail and show both plots together. I could plot two separate graphs, or I could overlap the two line plots on a single graph, which will give a sense of how to distinguish this type of contaminate from the target peptide chromatogram.

```
p_two_xic <- ggplot() +
    coord_cartesian(xlim=c(4500,5050), ylim=c(0, 6e7))
+
    scale_y_continuous(labels = inten_label) +
    scale_x_continuous(breaks = seq(4500,5000,100)) +
    geom_segment(aes(x=4795.983,
y=max(xic_chrom_peptide$inten),
                     yend = 0, xend = 4795.983),
                     linewidth = .5,
linetype="longdash", color=pal$darkorange) +
    geom_line(data=xic_chrom_peptide,
              aes(x=rt, y=inten, color="844.43 -
844.45")) +
    geom_line(data=xic_chrom_interference,
              aes(x=rt, y=inten, color="844.45 -
844.46")) +
    scale_color_manual(name = "      m/z range",
                       values = c("844.43 - 844.45" =
pal$black,
                                  "844.45 - 844.46" =
pal$blue)) +
    xlab("Retention Time (s)") +
    ylab("Intensity (counts)") +
    ggtitle(label = "MSV000086195 Batch 1 Fraction 5")
+
    theme_classic() +
    theme(plot.title = element_text(hjust = 0.5)) +
    theme(plot.subtitle = element_text(hjust = 0.5)) +
    theme(
        legend.position.inside = c(.9, .9),
        legend.justification = c("right", "top"),
        legend.box.just = "right",
        legend.margin = margin(4, 4, 4, 4)
    )


print(p_two_xic)
```

Figure 4.18 uses several features of `ggplot2` to show what's happening with these two ions. For this plot, I used two

different data sources, one for each chromatogram, and then added the color specification to the `scale_color_manual()` layer so that it would create a legend for the plot. This plot shows several key things about the data set that I will investigate more later. First, as suspected from the heatmap, the small peak in the mass spectrum is actually an interfering ion that is bleeding across the entire retention time. Second, the selected scan to perform MS/MS (MS level 2) was at the peak maximum of the second peak in this retention time range. Finally, it may be that the peak that shows up earlier in the XIC is the same peptide, but for some reason it was not selected for MS/MS. It could also be that this ion was selected but the MS/MS spectrum obtained was not matched to the same peptide.

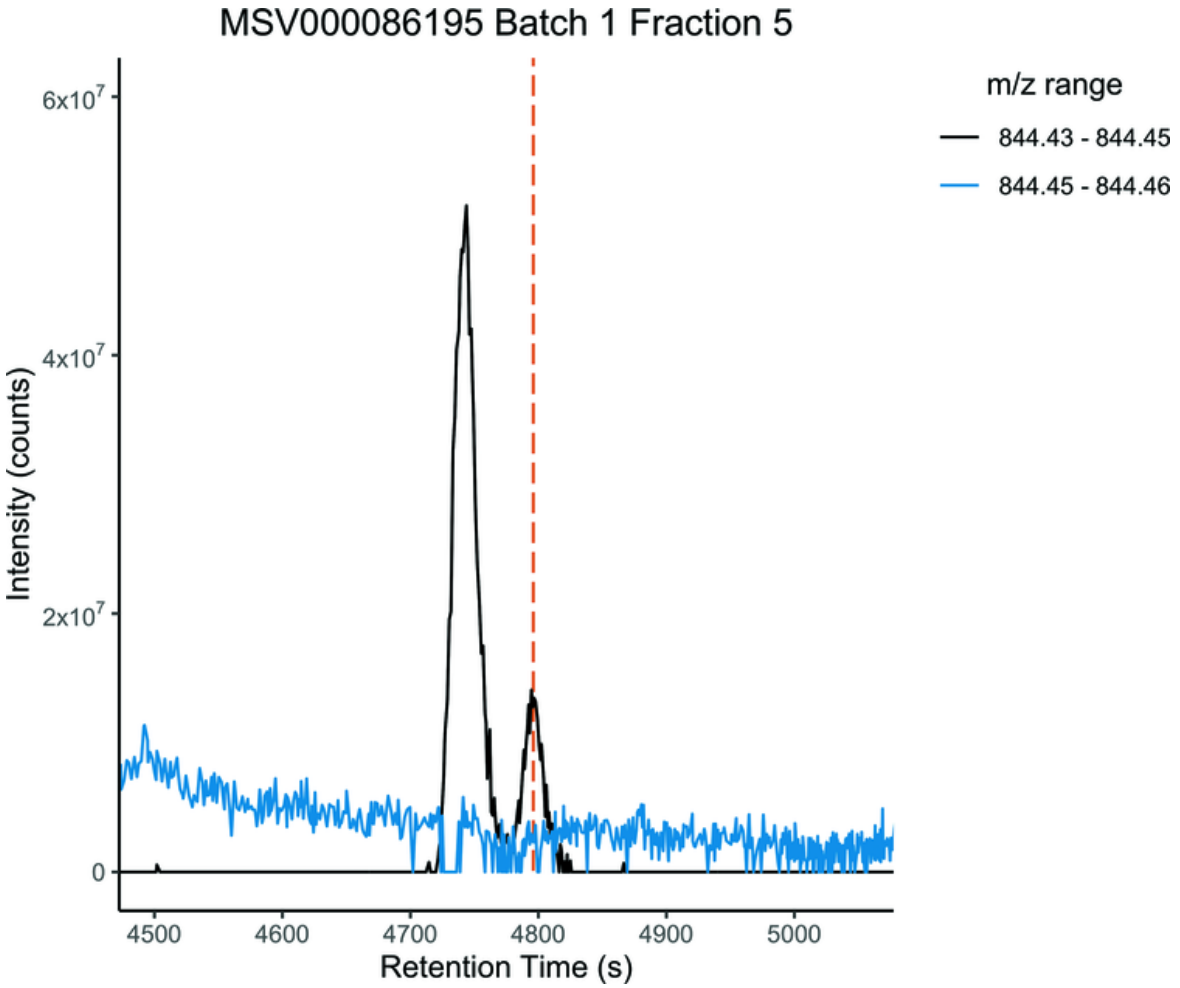


Figure 4.18 Extracted ion chromatograms overlayed and shown for the retention time of interest.

The exploratory data analysis on this data set and specific peptide mass spectra allows us to proceed with more detailed analysis related to IS. That analysis will be outlined in more detail in the next few chapters. Before moving on to data analysis of mass spectra in Chapter 5, I will show how to perform exploratory data analysis on mass spectral data that was collected explicitly as targeted chromatograms rather than extracting them from full scan data.

4.4.2 Reaction Monitoring

In Section 4.2, I explored tabular data from quantitative LC-MS/MS measurements obtained from SRM. SRMs are chromatograms in which a chromatographic signal is created by fixing the precursor ion *and* the product ion. This means there will only be an ion signal when a reaction occurs to produce a specific product ion from a specific precursor ion. Often, many more than one precursor-product pair are monitored in the same chromatographic run, and in Section 4.2.1, I showed peak areas for Codeine which was computed by monitoring the reaction of m/z 300.5 being converted into m/z 152.0. Another reaction, 300.5 to 165.1 was also monitored for codeine. I also showed two reactions for oxycodone, m/z 316.5 to 241.0, and 316.5 to m/z 212.1. In this section, I'll show how to visualize the actual chromatograms, which is extremely valuable when exploring data from these type of lower-dimension measurements.

First, I'll just take a look at what the raw files contain, and get some summary information about each run in the batch. I'll start with the first calibrator for Codeine used in the results summary example in Section 4.2.1.

```
  file_name <- "srm_001.mzML"
  srm_filename <- file.path("data", file_name)
  srm <- readSRMData(srm_filename)

  # Get a data.frame from the Feature data using the
  fData() accessor function
  id_df <- fData(srm)

  # The row number of the data frame is the index into
  the feature data (fData)
  # that allows specific scans to be extracted
  id_df$srm_index <- row(id_df)[,1]

  id_df[1,] |>
      pivot_longer(names_transform = as.character,
                   values_transform = as.character,
                          everything()) |>
      pandoc.table(split.cells=c(50,40), justify="left")
```

```
##
## ----------------------------------------------------
--------------------
## name                                    value
## ---------------------------------------- ----------------
--------------------
## chromatogramId                          SRM SIC Q1=136.1
Q3=119.1 sample=1
##                                         period=1
experiment=1 transition=10
##                                         start=0.79
end=1.59 ce=10
##                                         name=Amphetamine
1
##
## chromatogramIndex                       13
##
## polarity                                1
##
## precursorIsolationWindowTargetMZ        136.1
##
```

```
## precursorIsolationWindowLowerOffset   NA
##
## precursorIsolationWindowUpperOffset   NA
##
## precursorCollisionEnergy              10
##
## productIsolationWindowTargetMZ        119.1
##
## productIsolationWindowLowerOffset     NA
##
## productIsolationWindowUpperOffset     NA
##
## srm_index                             1
## ------------------------------------------------------------
--------------------
```

From this look at the raw data, I can see that there is information in the column `chromatogramId` that is not in the rest of the table and that should be parsed and put in its own column, and that some of it are redundant. There are also columns that have `NA` values which only clutter the table so they can be removed. I can also see that the `chromatogramIndex` is different from the row number, which is the index I have to use to access a specific SRM. Finally, I can see that the compound name is available, and by looking at other entries in the table, the naming convention for this data adds a -Dn to the name (where n is the number of deuterium atoms) to identify the IS. Based on this, I can extract the names, and identify if the compound is an IS or not. The next block of code extracts information from `chromatogramId` and cleans up the table for use in plotting the SRMs.

```r
# Get the compound names from the Id string
id_df$compound <- str_match(id_df$chromatogramId,
"name=(.+)")[,2]

# Mark the IS compounds which are named with a -Dx
where x is the number of
# 2H atoms in the isotopically labeled IS
id_df$IS <- str_match(id_df$compound, "-D\d")[,1] |>
    (function(data) {replace(data, !is.na(data),
TRUE)})() |>
    (function(data) {replace(data, is.na(data),
FALSE)})()


# Tidy up the table
id_df <- id_df |>
    dplyr::rename(all_of(c(Q1 =
"precursorIsolationWindowTargetMZ",
             Q3 = "productIsolationWindowTargetMZ")))
|>
    select(!c(precursorIsolationWindowLowerOffset,
              precursorIsolationWindowUpperOffset,
              productIsolationWindowLowerOffset,
              productIsolationWindowUpperOffset,
              chromatogramId)) |>
    relocate(c("srm_index", "compound", "IS"))

head(id_df)
```

```
##   srm_index          compound    IS chromatogramIndex
polarity    Q1
## 1         1     Amphetamine 1 FALSE                13
1 136.1
## 2         2     Amphetamine 2 FALSE                14
1 136.1
## 3         3   Amphetamine-D11  TRUE                96
1 147.2
## 4         4     Phentermine 2 FALSE               148
1 150.0
## 5         5     Phentermine 1 FALSE               147
```

```
1 150.0
## 6         6 Methamphetamine 2 FALSE                54
1 150.1
##   precursorCollisionEnergy    Q3
## 1                       10 119.1
## 2                       10  91.1
## 3                       25  98.1
## 4                       51  65.0
## 5                       27  91.0
## 6                       15 119.1
```

More can be learned from this table. The qualifier and the quantifier might not always be in an expected order. The quantifier is usually the more intense of the two product ions monitored, and I will make the assumption that when there are two compounds with a number in their name, the lower number is the quantifier, and the higher is the qualifier. This assumption needs to be checked: in the `chromatogramIdex` column, the quantifier seems to be collected first, and the qualifier collected second. It also looks like my guess at the right `str_detect()` pattern for finding the IS will work for more than -D9 compounds.

```
# Select the compound to be plotted and get the
index(s)
analyte <- "Codeine"
selected_analyte <- dplyr::filter(id_df,
str_detect(compound, analyte))

# sort out the order of the quant and qual SRM indexes
under the assumption
# that the lower chromatogramIndex is the quantifier
if(selected_analyte$chromatogramIndex[1] <
        selected_analyte$chromatogramIndex[2]) {
    quant_idx <- selected_analyte$srm_index[1]
    qual_idx <- selected_analyte$srm_index[2]
} else {
    quant_idx <- selected_analyte$srm_index[2]
    qual_idx <- selected_analyte$srm_index[1]
}

# Assume there is no matching IS for the analyte
analyte_has_IS <- FALSE
IS_idx <- NA

# If there is a matching IS, then set the flag and get
the IS index
if(length(selected_analyte$srm_index) == 3){
    if(selected_analyte$IS[3] == TRUE) {
        analyte_has_IS <- TRUE
        IS_idx <- selected_analyte$srm_index[3]
    }
}
```

Extract all of the x and y pairs for retention time and intensity for the quantifier and, if present, the matching IS using the `rtime()` and `intensity()` functions:

```
rt_quant <- rtime(srm[quant_idx])
inten_quant <- intensity(srm[quant_idx])
rt_qual <- rtime(srm[qual_idx])
inten_qual <- intensity(srm[qual_idx])

if(analyte_has_IS){
    rt_IS <- rtime(srm[IS_idx])
    inten_IS <- intensity(srm[IS_idx])
} else {
    IS_rt <- 0
    rt_IS <- 0
    inten_IS <- 0
}
```

With all of the data prepared, I can now plot any analyte with or without an exact IS as shown in Figure 4.19. This code is an example of building plots with layers that are conditional on the data being plotted. The approach is simple, assign the output of the `ggplot` layers to a variable depending on what is available to plot.

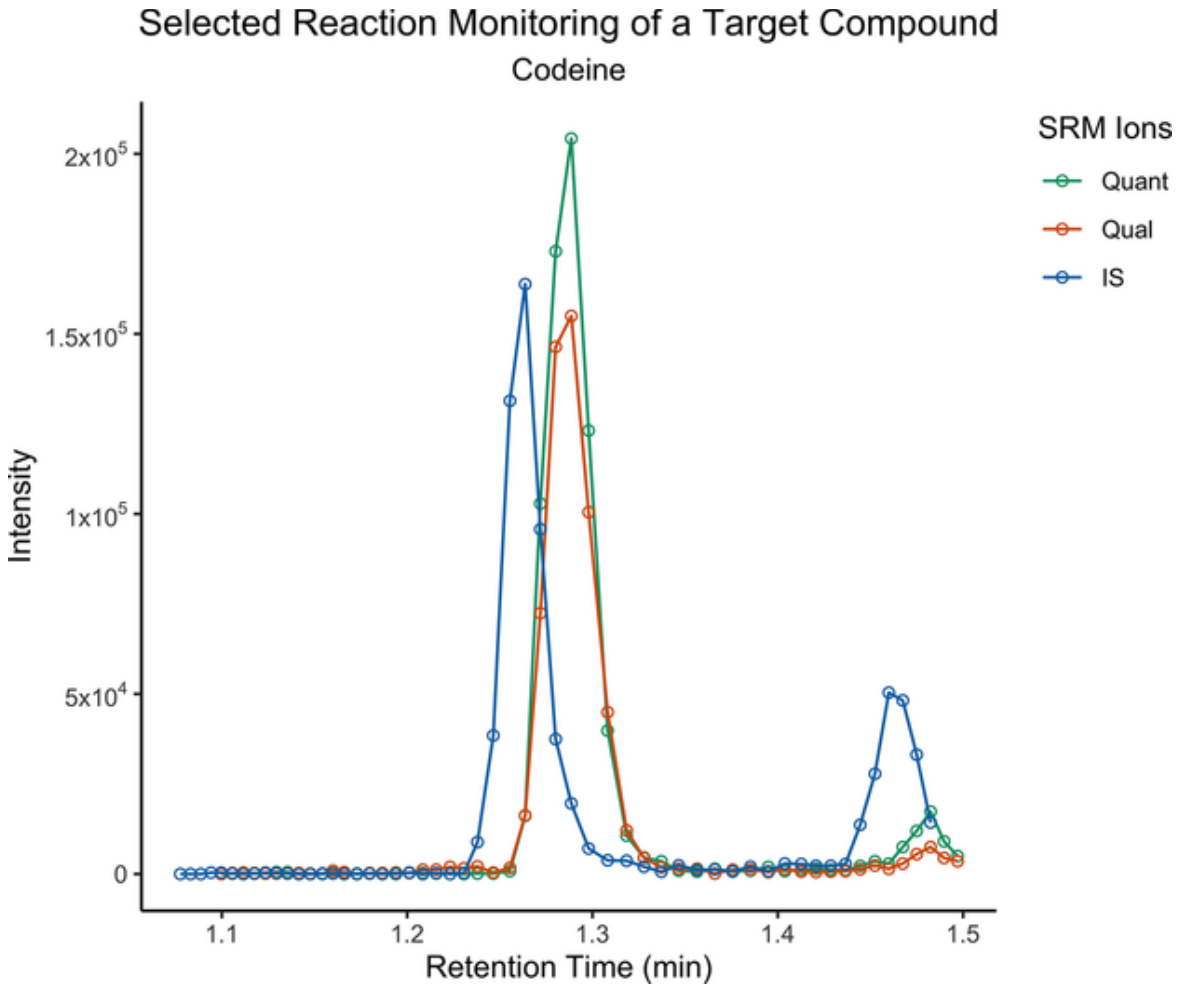


Figure 4.19 Plot of the three SRMs used for quantifying and qualifying codeine.

```
# build the basic part of the plot
p_selected_srm <- ggplot() +
    scale_y_continuous(labels = inten_label) +
    geom_line(aes(x=rt_quant, y=inten_quant,
color="Quant")) +
    geom_point(aes(x=rt_quant, y=inten_quant,
color="Quant"), shape=1 ) +
    geom_line(aes(x=rt_qual, y=inten_qual,
color="Qual")) +
    geom_point(aes(x=rt_qual, y=inten_qual,
color="Qual"), shape=1 )

# If there is an exact IS add the layers to the plot to
show it, otherwise
```

```
  # leave the IS off the plot and only plot the Quant and
  Qual
  if(analyte_has_IS){
      p_selected_srm <- p_selected_srm +
          geom_line(aes(x=rt_IS, y=inten_IS, color="IS"))
  +
          geom_point(aes(x=rt_IS, y=inten_IS,
  color="IS"), shape=1 ) +
          scale_color_manual(name='SRM Ions',
              breaks=c('Quant', 'Qual', 'IS'),
              values=c('Quant'=pal$green,
  'Qual'=pal$darkorange, 'IS'=pal$blue))
  } else {
      p_selected_srm <- p_selected_srm +
          scale_color_manual(name='SRM Ions',
              breaks=c('Quant', 'Qual'),
              values=c('Quant'=pal$green,
  'Qual'=pal$darkorange))
  }

  # Finish the plot and set the theme
  p_selected_srm <- p_selected_srm +
      xlab("Retention Time (min)") +
      ylab("Intensity") +
      ggtitle(label = "Selected Reaction Monitoring of a
  Target Compound",
          subtitle = analyte) +
      theme_classic() +
      theme(plot.title = element_text(hjust = 0.5)) +
      theme(plot.subtitle = element_text(hjust = 0.5)) +
      theme(
          legend.position.inside = c(.9, .9),
          legend.justification = c("right", "top"),
          legend.box.just = "right",
          legend.margin = margin(4, 4, 4, 4)
      )

  print(p_selected_srm)
```

Finally, the `xcms` package can be used to perform peak picking and area determination to get areas and responses as

shown in Section 4.2.1. In Chapter 6, I use this to look at important concepts like IS recovery and monitoring QC results over time. For now, I show how to pick and integrate the three codeine peaks and compute the instrument response for quantitation and the ion ratio for qualification.

```
picked_quant = findChromPeaks(srm[quant_idx],
                  param = CentWaveParam(peakwidth =
c(0.025,0.1), integrate = 2))

picked_qual = findChromPeaks(srm[qual_idx],
                  param = CentWaveParam(peakwidth =
c(0.025,0.1), integrate = 2))

picked_IS = findChromPeaks(srm[IS_idx],
                  param = CentWaveParam(peakwidth =
c(0.025,0.1), integrate = 2))


plot(picked_IS, main = "Peak Picking for Codeine
Internal Standard")
```

Figure 4.20 shows the peaks that `xcms` picked. The peak list below shows several descriptors, including retention time ranges, areas, and signal-to-noise ratio. This analysis used the `CentWave` algorithm [104] for peak picking, noise and area estimation, and baseline correction. In Chapter 6, I'll go into more detail about various algorithms for analyzing chromatographic peaks generated from mass spectrometry. For now, I will use `xcms` to explore the data without getting too detailed about the specific results it generates.

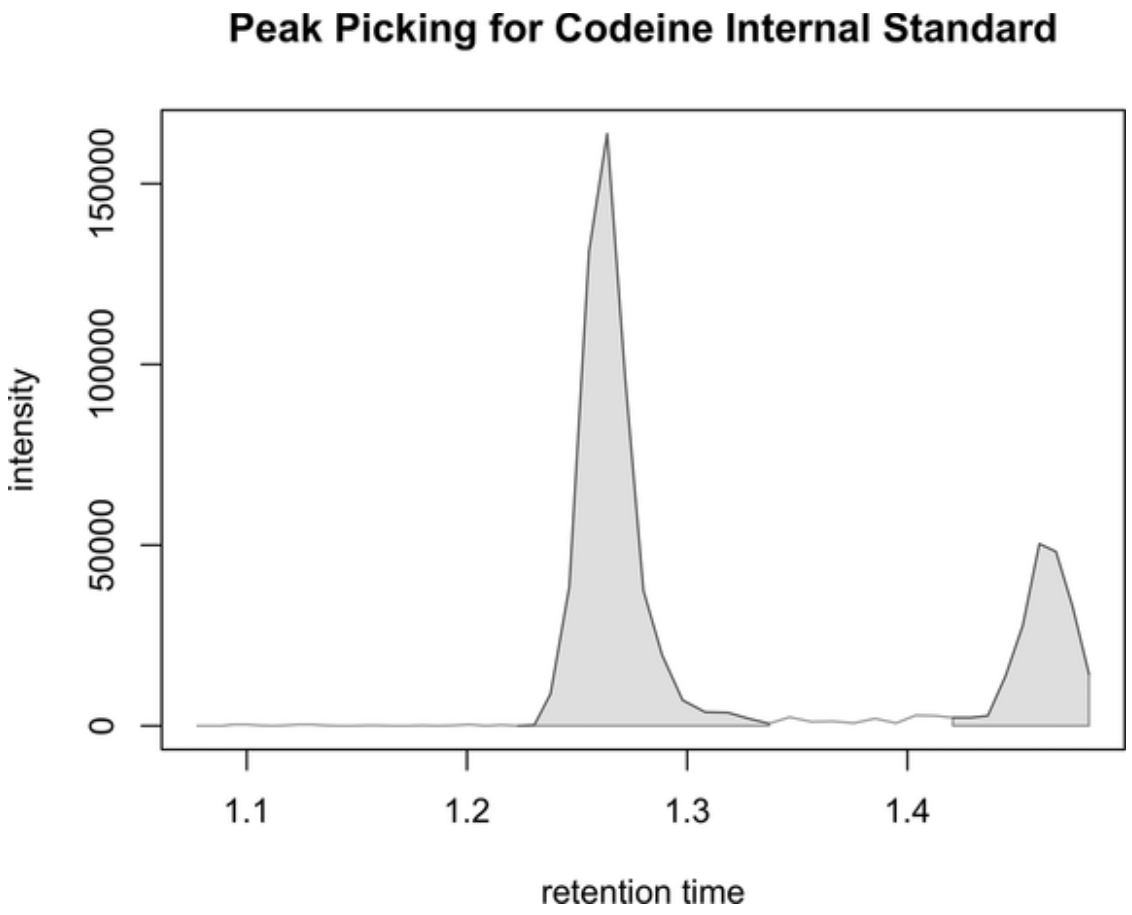


Figure 4.20 Picked IS peak using xcms CentWave algorithm.

```
picked_IS@chromPeaks
```

```
##             rt    rtmin    rtmax     into     intb
maxo  sn
## [1,] 1.263667 1.223183 1.337183 4496.756 4406.447
163819 192
## [2,] 1.459933 1.420633 1.482450 1505.468 1351.182
50364  15
```

And now, calculate the response and ion ratio from the picked peaks. The first peak listed is the analyte of interest, and the value for peak area (computed from a zero baseline) is given by the `into` variable.

```
  quant_area <-
  as.numeric(picked_quant@chromPeaks[1,"into"])
  qual_area <-
  as.numeric(picked_qual@chromPeaks[1,"into"])
  IS_area <- as.numeric(picked_IS@chromPeaks[1,"into"])

  instrument_response <- quant_area / IS_area
  ion_ratio <- quant_area / qual_area

  print(paste0("Quant Area: ", quant_area))
```

```
## [1] "Quant Area: 6187.50768854166"
```

```
  print(paste0("Qual Area : ", qual_area))
```

```
## [1] "Qual Area : 5141.51382564102"
```

```
  print(paste0("IS Area.  : ", IS_area))
```

```
## [1] "IS Area.  : 4496.75630769231"
```

```
  print(paste0("Inst Resp : ", instrument_response))
```

```
## [1] "Inst Resp : 1.37599355294328"
```

```
  print(paste0("Ion Ratio : ", ion_ratio))
```

```
## [1] "Ion Ratio : 1.20344083442589"
```

These values turn out to be reasonably close to those using vendor software:

```
  t(opioid_msdata[1,])
```

```
##             [,1]
## injection   "1"
## compound    "Codeine"
## sample_type "standard"
## quant_area  "6045.656"
## qual_area   "4994.051"
## quant_rt    "1.288583"
## qual_rt     "1.288583"
## response    "1.45256"
## ion_ratio   "1.210572"
```

This suggests that `xcms` could help perform various analyses of the system performance. In Chapter 6, I will discuss the wavelet analysis used by `xcms` in much more detail.

4.5 Summary

In this chapter, I have shown several ways to explore various kinds of data in order to understand what deeper analysis the data will support and what tools can be used. In the next two chapters, I will dive deeper into the analysis of mass spectra using the TMT example from Section 4.3 to analyze the internal controls discussed in Chapter 3. In Chapter 6, I will show how to perform more complex analysis on the quality and meaning of chromatographic data, both from extracted ion chromatograms and from SRM data.

Chapter 5
Data Analysis of Mass Spectra

5.1 Introduction

Each peak in a mass spectrum represents the mass-to-charge ratio (m/z) of an ion or a collection of ions. Characterizing individual peaks in a mass spectrum depends on the mass range under consideration. For relatively low molecular weight ions (fragments or otherwise), it is often desirable to compute monoisotopic weights and the weights and distributions of isotope peaks from the same ion structure. Other practical molecular weight calculations include calculating the weights of common adduct ions created in various ionization methods.

Related to calculating monoisotopic mass is the inverse calculation of possible chemical formulas from a given mass value. This calculation appears straightforward; however, many thousands of possible chemical formulas can match a narrow mass range. The usual method for narrowing the list of possible formulas is to place constraints on the number of rings, double bonds, and atoms.

Additionally, all ions, especially ions generated from atmospheric ionization methods, can have multiple charges (z) on a single ion, changing their position in the m/z axis. The number of charges can be considerable for relatively high molecular weight ions. This effect can be so extreme that the probability of observing the monoisotopic ion is extremely low and, thus, not observable in the spectrum. In these cases, an algorithm can characterize the distribution of charge states and estimate the molecular weight.

When a mass analyzer separates ions by m/z, the signal measured by the detector follows a characteristic shape with properties that are beneficial to know when performing further calculations. Characteristics such as resolution, center-of-mass (centroid), signal-to-noise, and, in some cases, peak shape. In this chapter, I will give examples of computing the centroid from quadrupole mass analyzer peaks and using models of time-of-flight peaks to improve molecular weight estimates.

Improvements in instrument design have made high-resolution mass spectrometry available beyond the electric/magnetic sector and FT-ICR instruments. However, analyzing very high-resolution spectra requires considering additional factors. This chapter will also provide examples and techniques for working with high-resolution data.

The chapter will conclude with an example of quantitative mass spectrometry and show how to use statistical methods on high-resolution mass spectra to determine the differences between quantities of proteins and peptides in samples. In this section, I will introduce the `infer` package from `tidymodels` [105] and the resampling approach to hypothesis testing. Statistical analysis will play an important role in the remainder of the book.

5.2 Molecular Weight Calculations

The history of mass spectrometry is tightly coupled to the discovery of atomic isotopes [106]. Since that time, one of the primary components of the analysis of mass spectra has been to understand the masses and charges of ions observed in a spectrum. One of the first tasks in analyzing the spectrum of a compound is to compute the expected mass-to-charge (m/z) of expected ions. When the molecule is relatively small, it is often the case that the monoisotopic ion is the most abundant, and the higher mass isotopes have a

lower relative abundance. At higher molecular weights, especially for intact proteins and other polymers, the monoisotopic mass is not easily detectable, and other methods are used to compute the molecular weight of the compound. Another important aspect of m/z values observed in mass spectra is the mass of the ionizing moiety. For atmospheric pressure ionization (API) methods, the molecular ion is created by the gain or loss of one or more protons. The rest mass of a proton (1.007276470 Da) [107] is important in computing the observed ions in a mass spectrum, adding or subtracting from an observed weight when working backward from a positive or negatively charged ion's m/z to a formula [108]. In addition, API can also create molecular ions through the addition of NH_4^+ and Na^+. Finally, there are many ionization methods used in mass spectrometry besides API. Some of the more common methods are electron impact ionization (EI) and chemical ionization (CI), which create molecular ions by adding or removing electrons to create odd electron molecules of either positive or negative polarity. Also, inductively coupled plasma/mass spectrometry (ICP/MS) appears in both industrial and research applications, making exact molecular weight and isotope distribution calculations very important in mass spectrometry. It has been the International Union of Pure and Applied Chemists (IUPAC) that has maintained and updated both the molecular weights of atoms and their isotopic distributions. Through a series of technical reports published over the years [109–111], IUPAC provides the definitive values for the terrestrial weights and isotope distributions for all the known atoms. I'll use both the weights and the relative abundances to compute the mass values and the expected intensities of ions observed in mass spectra.

In the next section, I will go through examples of calculating the monoisotopic mass (as shown in Figure 4.10 in Chapter

4) for various compounds. After that, I'll explain how to calculate masses for isotopes.

5.2.1 Monoisotopic Mass Calculations

Calculating monoisotopic mass values from a chemical formula is a straightforward process of multiplying the number of atoms by the molecular weight provided by IUPAC and then adding the number of ionizing adducts. The basic formula is:

$$\frac{m}{z} = \frac{[M + n \times Adduct^{\pm}]}{n} \quad (5.1)$$

where the observed m/z peak is the mass-to-charge ratio, M is the monoisotopic mass, and n is the number of adducts (protons or other ionizing moieties) divided by the number of ionizing moieties. From the IUPAC technical reports cited above, it is simple to compute the theoretical monoisotopic weight for any chemical formula. For example, the mass-tagged, deamidated control peptide APLDNDIGVSEATR in Figure 4.10 has the chemical formula $C_{68}{}^{13}C_4H_{119}N_{18}{}^{15}NO_{27}$. The calculation of the monoisotopic mass of this formula is given in Table 5.1.

TABLE 5.1

Calculation of monoisotopic mass for $C_{68}{}^{13}C_4H_{119}N_{18}{}^{15}NO_{27}$

Atom	Isotope	Mass (Da)	Count	Total
C	12	12	68	816
C	13	13.0033548380	4	52.0134193520
H	1	1.0078250319	119	119.9311787961
O	16	15.9949146223	27	431.8626948021
N	14	14.0030740074	18	252.0553321332
N	15	15.000108973	1	15.0001089730
Total				**1686.8627341**

The monoisotopic mass for this formula is `1686.8627341` when the limiting significant digits of the mass of H are taken into account. To compute the m/z value of the ion observed in Figure 4.10, I will use Eq. (5.1), where there are two protons added, leading to a charge state z of 2.

$$\begin{aligned}\frac{m}{z} &= \frac{1686.8627341 + 2 \times 1.007276470}{2} \\ &= \frac{1688.8772870}{2} \\ &= 844.4386435\end{aligned}$$

Note that the masses used come from IUPAC and that these only apply to samples of terrestrial origin. Further, they are updated occasionally, so the IUPAC reports [109–111] and any errata should be checked for updates. The primary changes made by IUPAC are not in the monoisotopic masses, which have remained quite stable, but rather in the abundance of various isotopes. As a wider range of materials are measured, more natural isotopic enrichment and

depletion are observed and reported by IUPAC. When working with isotopes in mass spectrometry the isotopic abundances should be checked against the latest IUPAC values. In the labeled peptide described in Table 5.1, there were two stable isotopes used in the TMT-10plex mass tag: ^{13}C and ^{15}N which contributed their exact isotopic mass to the monoisotopic mass since they were added to the molecule synthetically, not via natural processes. In the calculation performed, I assumed that all four of the ^{13}C atoms were 100% pure ^{13}C. However, due to manufacturing impurities, it is possible that some of the ^{13}C atoms were actually ^{12}C [112]. The result is that there could be some low abundance ions observed at a lower m/z value than the theoretical isotopic mass. While the occurrence of these impurities is generally ignored, it is worth using the exploratory techniques described in Chapter 4 to determine if additional steps should be taken to account for impurities in isotopically labeled compounds.

While synthetic isotopic abundances might be negligible in a particular mass spectrum, the naturally occurring isotopes are not, and I'll address that next.

5.2.2 Isotope Abundance Calculations

The discovery of isotopes, atoms with extra neutrons, using mass spectrometry has a long and fascinating history [113]. Using what could be called the first mass spectrometer, J.J. Thomson reported [114] that Neon was comprised of atoms with two masses: 20 and 22. Building on this foundation, his student F.W. Aston used improved designs of mass spectrometers to discover [115] the presence of isotopes for many atoms.

Since that time, better instrumentation and theories have been developed that improve our understanding of both the weights and relative abundances of atomic isotopes, turning

them into tools that are used in many ways in modern mass spectrometry [112, 113].

In the previous section, I used the current values for the isotopes ^{13}C and ^{15}N to compute the exact mass of compounds measured in mass spectra. Now, I will show how to use an R package to compute isotope distributions from arbitrary chemical formulas. A detailed review of the various isotope distribution calculation algorithms can be found in Valkenborg et al. [116].

While there are several packages for computing isotope distributions, I will focus the examples in this chapter on the CRAN package `IsoSpecR` [117, 118]. IsoSpecR is written in C++ and has both Python and R bindings. Using the C/C++ programming languages for computationally intensive algorithms is common in R but can, on occasion, lead to support issues. The Bioconductor package `Rdisop` [119–122] is also an option, but at the time of writing, this package has limited support and its maintainer suggests that it be removed from the Bioconductor repository [123], which means that bugs may or may not get any attention.

`IsoSpecR` is not part of Bioconductor, so it can be installed like any other CRAN package. In addition to the functions needed to compute isotopic distributions, it includes the isotope distribution data, which can be loaded using the `data()` function and then extended. The data are in data frames that contain the name, exact mass, and isotopic abundance for a large number of atoms.

```
library(IsoSpecR)

# load the package data
data("isotopicData")

names(isotopicData)
```

```
## [1] "enviPat"          "enviPatShort"     "IsoSpec"
"IsoSpecShort"
## [5] "IsoSpecShortZero"
```

From this list, `IsoSpec`, `IsoSpecShort`, and `IsoSpecShortZero` are data frames that contain the mass/abundance values and the names of the atoms as used by the package. The `IsoSpecShort` data frame has only those atoms typically found in peptides, and `IsoSpecShortZero` includes the zero abundance isotope $^{35}\mathrm{S}$. For calculations on most biological molecules, the `IsoSpecShort` data frame will be sufficient and faster than the full `IsoSpec` data frame.

In Section 4.2.1, codeine was described in a low-resolution method as having a precursor ion at m/z 300.5. Codeine has a chemical formula $C_{18}H_{21}NO_3$. Using the method described above, this gives an exact monoisotopic mass of `299.15214`. For the observed m/z value in the singly charged case, I have to add the proton mass to get `300.15942`, which matches the information in the GNPS [93] for codeine (Spectrum ID CCMSLIB00011429539) [124].

```r
# Use the short isotope table
isotope_table <- isotopicData$IsoSpecShort

# Specify the chemical formula
molecule <- c(C=18, H=21, N=1, O=3)

# calculate all isotopes above 1% of the total
probability
isotope_dist <- IsoSpecify(molecule=molecule,
                           showCounts = TRUE,
                           stopCondition = 0.999,
                           isotopes=isotope_table)

isotope_dist <- as.data.frame(isotope_dist)
isotope_dist$mass <- format(isotope_dist$mass,
digits=10)
isotope_dist$prob<- format(isotope_dist$prob, digits=5)
print(arrange(isotope_dist, mass))
```

```
##           mass       prob C12 C13 H1 H2 N14 N15 O16 O17
O18
## 1  299.1521435 0.81169971  18   0 21  0   1   0   3   0
0
## 2  300.1491784 0.00296700  18   0 21  0   0   1   3   0
0
## 3  300.1554984 0.15933890  17   1 21  0   1   0   3   0
0
## 4  300.1563607 0.00093003  18   0 21  0   1   0   2   1
0
## 5  300.1584203 0.00197258  18   0 20  1   1   0   3   0
0
## 6  301.1525333 0.00058243  17   1 21  0   0   1   3   0
0
## 7  301.1563885 0.00500752  18   0 21  0   1   0   2   0
1
## 8  301.1588532 0.01477048  16   2 21  0   1   0   3   0
0
## 9  302.1597434 0.00098299  17   1 21  0   1   0   2   0
1
```

```
## 10 302.1622080 0.00085911  15   3 21  0   1   0   3   0
0
```

From this output, you can tell that `IsoSpecR` computes the *isotopic fine structure*, which would require a high-resolution instrument to observe, and even the highest resolution instruments might not be able to resolve to the level computed here. The same calculation done above regarding the m/z value of the protonated ions can be performed by adding the mass of the proton to each value in the mass column. The most abundant isotopes are those with 0, 1, and 2 $^{13}\mathrm{C}$ atoms. The next most abundant isotope introduces one $^{18}\mathrm{O}$ in place of a $^{16}\mathrm{O}$ atom.

With fine structure algorithms, you can specify the probability of the occurrence of an isotopologue you want to be returned in the output. In the program above, the stopping value was set so that `0.999%` of the abundance was accounted for. That means that depending on the resolution of your instrument, it might be necessary to *combine* probabilities into a single peak when the instrument would be unable to distinguish between peaks and very close m/z values. Setting the stopping value for the default algorithm to `0.99` would return all the isotopes with a joint probability of `99%`.

The process is similar when calculating isotope distributions for more complicated molecules such as the APLDNDIGVSEATR control peptide shown in Section 5.2.1. To perform a calculation similar to the example from Table 5.1, the `IsoSpecR` package will be extended to add the label atoms from the TMT-10plex tag and the two protons that created the multiply charged ion in the raw data. This is done by extending the `IsoSpecShortZero` isotope data from the `IsoSpecR` package. This approach can be used to add any atoms or other moieties needed to calculate distributions.

```r
data("isotopicData")

# This short list includes only isotopes found in
peptides and includes
# the zero abundance isotope for Sulfur at 35 Da
isotope_table <- isotopicData$IsoSpecShortZero

# convention: Ch for 13C (C-Heavy) and Nh for 15N (N-
Heavy) Pr is a proton
labeled_atoms = data.frame(
    element = c('Ch', 'Nh', 'Pr'),
    isotope = c('Ch', 'Nh', 'Pr'),
    mass    = c(isotope_table[isotope_table$isotope ==
'C13', 'mass'],
                isotope_table[isotope_table$isotope ==
'N15', 'mass'],
                1.007276466621),
    abundance = c(1, 1, 1),
    ratioC  = c(NA, NA, NA)
)

# Add the new isotopes to the list
isotope_table <- rbind(isotope_table, labeled_atoms)

# this molecule has 4 C13 atoms, 1 N15 atom, and is in
charge state 2
# so has 2 protons (Pr)

molecule <- c(C=68, Ch=4, H=119, Pr=2, N=18, Nh=1,
O=27)
charge_state <- 2

# calculate all isotopes above 1% of the total
probability
isotope_dist <- IsoSpecify(molecule=molecule,
                           stopCondition = 0.99,
                           isotopes=isotope_table,
                           algo=0)

# tidy up the output of the calculation and correct for
charge state and
```

```r
  # generate a relative intensity of the isotope
  distribution
  isotopes <- as_tibble(isotope_dist) |>
                  dplyr::mutate(mass = mass/charge_state)
  |>
                  dplyr::mutate(mass=format(mass, digits
  = 10),
                                prob=format(prob, digits=5)) |>
                  dplyr::arrange(mass)

  print(n=length(isotopes$mass), isotopes)
```

```
## # A tibble: 21 x 2
##    mass        prob
##    <chr>       <chr>
##  1 844.4386434 0.4136340
##  2 844.9371609 0.0272152
##  3 844.9403208 0.3067461
##  4 844.9407520 0.0042654
##  5 844.9417818 0.0056962
##  6 845.4388383 0.0201825
##  7 845.4407659 0.0229660
##  8 845.4419983 0.1120670
##  9 845.4424294 0.0031632
## 10 845.4434592 0.0042242
## 11 845.9392834 0.0015111
## 12 845.9405157 0.0073735
## 13 845.9424433 0.0170313
## 14 845.9436757 0.0268877
## 15 845.9441068 0.0011556
## 16 845.9451366 0.0015433
## 17 846.4409608 0.0011206
## 18 846.4421931 0.0017691
## 19 846.4441207 0.0062223
## 20 846.4453531 0.0047650
## 21 846.9457982 0.0014929
```

```
# There are many standalone and online tools for
computing relative
# abundances of the isotopes of specific structures.

theoretical_x <- as.double(isotopes$mass)
theoretical_y <- as.double(isotopes$prob)

theoretical_y <- theoretical_y / theoretical_y[1]

# load the profile data from the profile raw data

profile_file_name <- file.path("large-data",

"ScltlMsclsMAvsCntr_Batch1_BRPhsFr5_prof.mzML")

batch1_fraction05_profile <- Spectra(profile_file_name)

# Based on the summary table, the peptide of interest
is at scan number 21515

prof_prec_scan <- batch1_fraction05_profile[21515]
prof_x <- mz(prof_prec_scan)[[1]]
prof_y <- intensity(prof_prec_scan)[[1]]

# Use the which() function to find the maximum peak
height in the desired range
# in order to scale the theoretical isotope intensities
to to experimental
# data

scaled_y <- theoretical_y *
    max(prof_y[which(prof_x>844)
[1]:which(prof_x>844.55)[1]])
```

A simple overlay plot can be used to show the measured isotopic peaks combined with their theoretical m/z and intensity values. Often in data visualization, it is useful to create a multipanel plot. The `ggpubr` package provides the `ggarrange()` function, which combines multiple plot objects generated from `ggplot()` into a single figure. When creating

multiple plots that plot in the exact same way, creating a specialized plot function is easier to read and maintain than replicating the plot code. Here I create such a plot function called `peak_plot()`, which will make sure that each panel will use the same code regardless of how many isotopes I choose to plot:

```
peak_plot <- function(theoretical_x, scaled_y, prof_x,
prof_y,
                      xrange, yrange, main_title) {
    ggplot() +
        coord_cartesian(xlim=xrange, ylim=yrange) +
        scale_y_continuous(labels = inten_label) +
        geom_segment(aes(x=theoretical_x, y=scaled_y,
                yend = 0, xend = theoretical_x),
                linewidth = 0.5, color=pal$darkorange)
+
        geom_line(aes(x=prof_x, y=prof_y)) +
        geom_point(aes(x=prof_x, y=prof_y), shape=1 ) +
        xlab("m/z") +
        ylab("Intensity") +
        theme_classic() +
        theme(plot.title = element_text(hjust = 0.5)) +
        ggtitle(label = main_title)
}
```

Now I will call the `peak_plot()` function with the various ranges to generate a `ggplot` plot object for each isotope:

```
# create plots for the first four isotopes
mono <- peak_plot(theoretical_x, scaled_y,
                  prof_x, prof_y,
                  xrange = c(844.425,844.475),
                  yrange = c(0,1.5e7),
                  main_title = "Monoisotopic (M)
844.43864")

iso_1 <- peak_plot(theoretical_x, scaled_y,
                  prof_x, prof_y,
                  xrange = c(844.925,844.975),
                  yrange = c(0,1.5e7),
                  main_title = "M+1 Isotopes
844.94032")

iso_2 <- peak_plot(theoretical_x, scaled_y,
                  prof_x, prof_y,
                  xrange = c(845.425,845.475),
                  yrange = c(0,5e6),
                  main_title = "M+2 Isotopes
845.44200")

iso_3 <- peak_plot(theoretical_x, scaled_y,
                  prof_x, prof_y,
                  xrange = c(845.925,846.25),
                  yrange = c(0,1e6),
                  main_title = "M+3 Isotopes
845.94368")
```

Using `ggarrange()` `ggplot2` plot objects can be combined into a single diagram and labeled to make Figure 5.1.

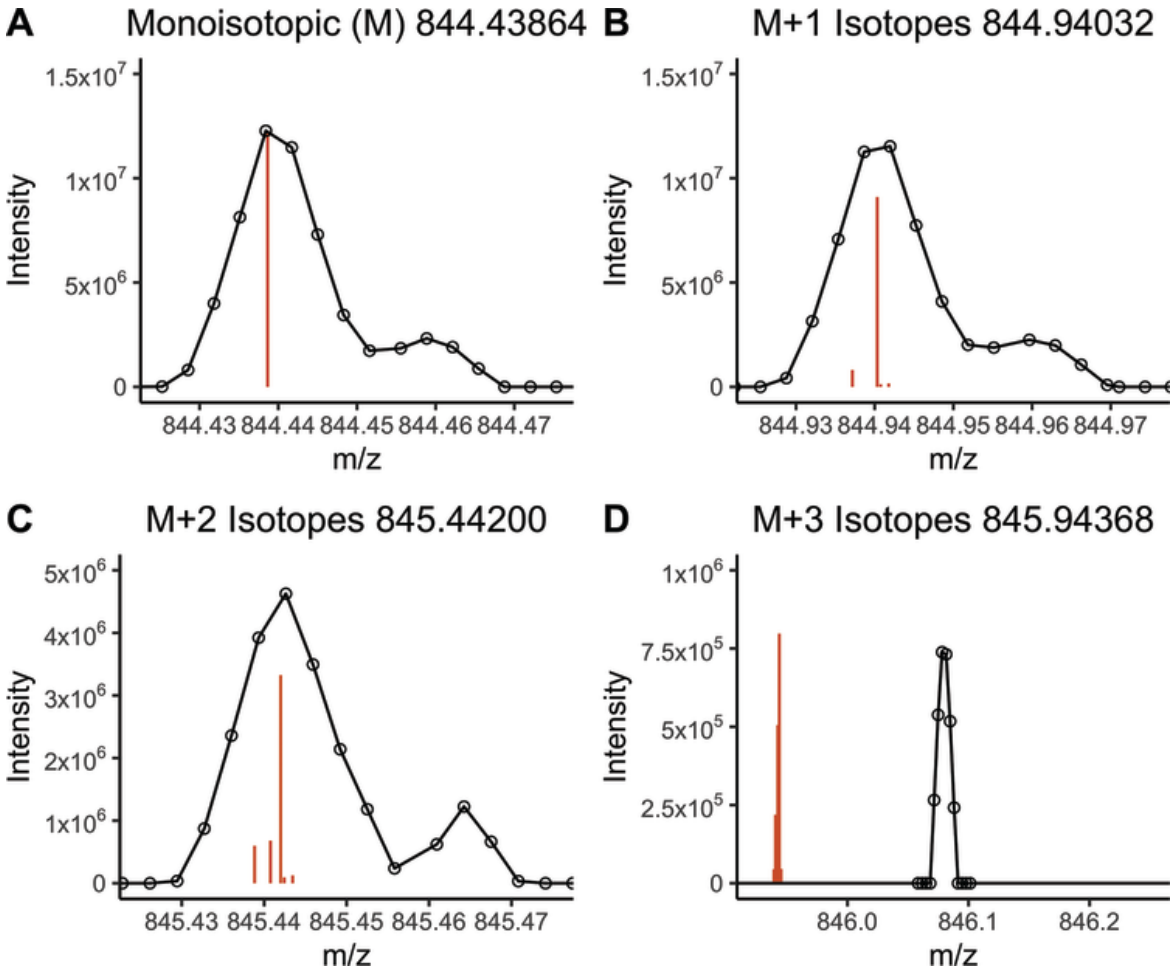


Figure 5.1 Comparison of theoretical and observed isotope m/z and intensities for MS1 spectrum of APLDNDIGVSEATR 2+ (Deaminidated N, TMT10plex).

```
library(ggpubr)

p_isotopes <- ggarrange(mono, iso_1, iso_2, iso_3, ncol
= 2, nrow = 2,
                        labels = c("A", "B", "C", "D"))

print(p_isotopes)
```

When the theoretical isotope abundances are scaled to the observed monoisotopic peak (Figures 5.1a), there are discrepancies between the observed isotope intensities and the theoretical values (Figures 5.1b,c). Further, either there

is a mass shift in the fourth isotope, or it wasn't observed at all, and the experimental peak belongs to a different compound (Figure 5.1d).

5.2.3 Adducts

So far, I've covered mostly positive ion mode API, which usually produces protonated ions. As discussed, in the positive ion mode, a molecule could have multiple protons and, therefore, multiple charges. The same is true in the negative ion mode. Deprotonation (single and multiple) produces negative ions. In addition to protons, it is also common to observe ions that are formed from other adducts. The most common of these are Sodium (Na^+), Ammonium (NH_4^+), and Potassium (K^+). In an often-cited paper by Huang et al., 30 different positive ion adducts are given along with 14 different negative ion adducts [125]. Many more adducts and adduct/neutral-loss combinations have been reported using various spectral libraries [126]. To compute the m/z value for an ion created by an adduct, the adduct mass is simply added to the monoisotopic mass of the molecule and then divided by the charge as discussed in Section 5.2.1. Table 5.2 shows some of the most common positive ion adducts with the observed change in molecular ion mass.

TABLE 5.2

Common positive ion adducts and their m/z values

Adduct	Mass added to M	Charge
H^+	1.0072765	+1
Na^+	22.989218	+1
NH_4^+	18.033823	+1
K^+	38.963158	+1

In addition to the loss of a proton $[M - H^+]^-$, another common negative ion adduct is Chlorine $[M + Cl]^-$ which increases the observed mass by `34.969402`. More negative ion adducts can also be found in Huang et al. [125].

Using the example of the codeine molecule given above, from earlier in this section, the formula $C_{18}H_{21}NO_3$ has an exact monoisotopic mass of `299.15214`. The `.mgf` file format is one of the raw spectrum file formats that can be read by the `Spectra()` function, but it requires an additional backend reader called `MsBackendMgf()`, which comes from the `MsBackendMgf` Bioconductor package.

```
library(MsBackendMgf)

mgf_file_name <-
file.path("data","CCMSLIB00011429539.mgf")

spectrum <- Spectra(mgf_file_name, source =
MsBackendMgf())

mz_values <- mz(spectrum)[[1]]
inten_values <- intensity(spectrum)[[1]]
```

Once the data is loaded, it can be plotted to show how the theory matches the observed values. The reference spectrum of codeine CCMSLIB00011429539 [124] is shown in Figure 5.2.

```
p_codeine_spectrum <-
    ggplot() +
    coord_cartesian(xlim=c(295,345), ylim=c(0,1.5e5)) +
    scale_y_continuous(labels = inten_label) +
    geom_segment(aes(x=mz_values, y=inten_values, yend
= 0, xend = mz_values),
                 linewidth = 0.5, color=pal$black) +
    annotate("text", x = 308, y = 1.3e5,
color=pal$blue,
             label = "Instrument Centroid\n300.1589") +
    xlab("m/z") +
    ylab("Intensity") +
    theme_classic() +
    theme(plot.title = element_text(hjust = 0.5)) +
    theme(plot.subtitle = element_text(hjust = 0.5)) +
    ggtitle(label = "Codeine ESI qTOF Spectrum",
            subtitle = "GNPS CCMSLIB00011429539")

print(p_codeine_spectrum)
```

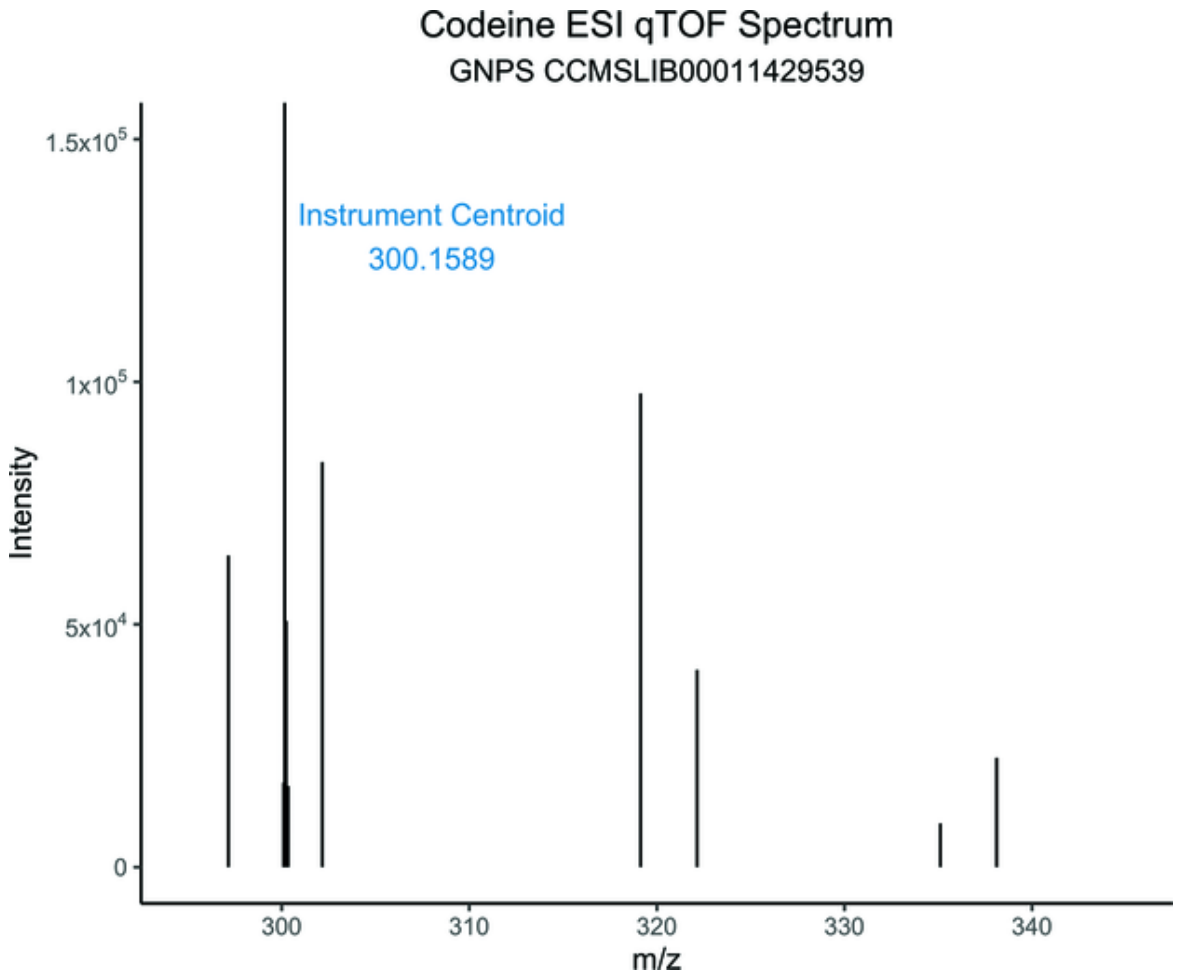


Figure 5.2 ESI spectrum of codeine from Waters LC-MS qToF.

The MGF or “Mascot generic file” is simply a text file containing mass and intensity values and a few header elements. It can be opened with any text editor, and examining the masses near the monoisotopic mass of codeine shows several of the expected adducts. The specific adducts observed depend on the molecule, the matrix from which it was introduced, and the specific ionization source and settings.

For the most common adducts, the mass errors observed are quite small and provide solid evidence of the codeine molecular weight. Not all of the ions above the monoisotopic mass of codeine seen in the spectrum match up with the

common adducts shown in Table 5.3. This is most likely because the codeine structure has a complicated fragmentation pathway, which normally starts from the loss of a water molecule, as suggested by Zhang et al. [127]. Ions gain internal energy during the ionization process and can undergo rearrangement, which, when adduct ions are added, lead to various masses higher than the monoisotopic weight but don't match up to simple ionization adduct masses.

TABLE 5.3

Calculation of adducts to codeine: $C_{18}H_{21}NO_3$

Observed m/z	Adduct	Mass error
282.148285	$M - H_2O + H^+$	0.0005750
300.158905	H^+	0.0.005150
322.140991	Na^+	0.0.009222
338.115112	K^+	0.0.001895

5.2.4 Computing Molecular Formulas from Mass

Using sufficiently high resolution and mass accuracy, it is possible to calculate potential chemical formulas from monoisotopic masses from spectra. The subject has a rich history since early commercial mass spectrometers were sector instruments with high resolution [128]. Magnetic sector instruments were adopted early for oil, coal, gas, and environmental analysis. Because of the analytes of interest to these industries, many of the chemical formula calculations were limited primarily to C, H, and O. Later, programs were expanded to include other elements [108, 129–131], including those of biological interest.

The package I will be using for these calculations is unusual in that it is not available on either CRAN or Bioconductor but is hosted on the source code repository system called GitHub. R has a package called `remotes`, which can be installed from CRAN using the `install.packages("remotes")` command. The `remotes` package includes functions to install from many repositories and source code locations. The package I will use for formula calculation is called `MassTools` [132, 133], located at `mjhelf/MassTools` on `GitHub`. To install it, use the command: `remotes::install_github('mjhelf/MassTools')`. This package uses functions from the `Rdisop` package, which is in Bioconductor, so it will also have to be installed. To see that `MassTools` is installed, just load it like any other package and check for errors.

```
  library(MassTools)
```

The specific function in MassTools that will calculate chemical formulas from molecular weights is called `calcMF()`. The help page for the function shows the many options available, but I will just show how to use it to compute possible formulas based on the theoretical and experimental monoisotopic masses of codeine.

First, the `calcMF()` function needs a set of elements to choose from when selecting candidate atoms to be included in the formula. The default atom list is from `Rdisop::initializeCHNOPS()`, which includes phosphorus and sulfur, which are needed to characterize peptides.

```
  calcMF(mz = 299.1521435, z=0, ppm=5) |>
      dplyr::select(c(mz, MF, RdisopScore, unsat, error,
  ppm))
```

```
##          mz           MF  RdisopScore unsat
error          ppm
## 1  299.1521    C18H21NO3 0.0837838732     9
```

```
0.000000127  0.000424533
## 3  299.1524   C11H29N3S3 0.0498169901    -1
0.000216621  0.724116490
## 7  299.1528 C10H27N3O3P2 0.0060411248     0
0.000622341  2.080349459
## 8  299.1515  C10H25N3O5S 0.0034178005     0
-0.000651769 -2.178720809
## 9  299.1528   C11H21N7OS 0.0038665037     5
0.000685579  2.291740223
## 12 299.1511  C12H22N5O2P 0.0001091148     5
-0.001031460 -3.447944541
```

The chemical formula for codeine is the top-scoring candidate formula based on using the theoretical monoisotopic peak and a maximum absolute error of 5 ppm. The same calculation can be done for the experimental mass shown in Table 5.3.

```
  calcMF(mz = 300.158905, z=1, ppm=5) |>
      dplyr::select(c(mz, MF, charge, RdisopScore, unsat,
  error, ppm))
```

```
##          mz           MF charge  RdisopScore unsat
error        ppm
## 2  300.1588  C10H26N3O5S      1 5.510520e-02  -0.5
-0.0001368139 -0.4558049
## 5  300.1594    C18H22NO3      1 1.200553e-02   8.5
0.0005150821  1.7160313
## 6  300.1584  C12H23N5O2P      1 8.735649e-03   4.5
-0.0005165049 -1.7207716
## 8  300.1596   C11H30N3S3      1 2.773479e-03  -1.5
0.0007315761  2.4372960
## 9  300.1579  C12H31NOPS2      1 1.820447e-04  -1.5
-0.0009854629 -3.2831373
## 11 300.1600 C10H28N3O3P2      1 6.175889e-05  -0.5
0.0011372961  3.7889800
## 13 300.1601   C11H22N7OS      1 3.002394e-05   4.5
0.0012005341  3.9996617
```

In this example, the codeine formula was not the top choice because of the inclusion of sulfur in the element list. When working with specific classes of molecules, it can be helpful to control the atom candidates, which can be done by modifying the list generated by `Rdisop::initializeCHNOPS()`. The names of the atoms in the list can be printed using the `lapply()` function and the `[[` operator.

```
  atoms <- Rdisop::initializeCHNOPS()
  unlist(lapply(atoms,'[[',1))
```

```
## [1] "C" "H" "N" "O" "P" "S"
```

To limit the formula to only C, H, N, and O, I can just delete the last two elements of the list, which removes the information for P and S:

```
  atoms <- atoms[1:4]
```

Now, I will repeat the formula calculation with the shortened atom list:

```
  calcMF(mz = 300.158905, elements=atoms, z=1, ppm=5) |>
      dplyr::select(c(mz, MF, charge, RdisopScore, unsat,
  error, ppm))
```

```
##         mz        MF charge RdisopScore unsat
error      ppm
## 2 300.1594 C18H22NO3      1   0.1095096   8.5
0.0005150821 1.716031
```

Now, there is only a single formula within the 5 ppm error window, and it exactly matches the formula for codeine.

5.3 Statistical Analysis of Spectra

In this section, I will go through a detailed example of the statistical analysis of peaks in a mass spectrum. Recall from Section 3.4.1 that the tandem mass tag TMT10plex [89] kit was used to label digested peptides from human samples spiked with a predigested protein, bacterial beta-galactase [90], as an internal control. In Section 3.6, I extracted selected information on all of the labeled peptides from the internal control, organized them into a data frame, and saved them to a file for later use.

In this experiment, there were 10 human samples combined into a single solution after labeling. Since the mass of the label added to a peptide is the same for each human sample, it means that all 10 TMT10plex fragment ions should be present for the internal controls.

To begin, I will load the data mentioned above and sort it by batch, fraction, and scan. By now, it should be clear that there are many formats to store intermediate data to be used later. My personal practice is to use CSV files, but you may choose another format, like the base R data serialization format, or another. For me, it is not usually worth the effort to create an XML format for intermediate results unless you are sharing them with collaborators and want to protect against file corruption while maintaining a clear text format. CSV files are notoriously bad for sharing since, as described earlier, loading them into a spreadsheet and editing them there can create all kinds of avoidable problems. So, with that warning firmly in mind, you'll notice that I tend to store intermediate results from potentially long-running loops or analyses in CSV files, and there are times when I will look at them in a text editor, but you have to be very careful with this practice.

```
control_id <- read_csv(file.path("data",
"all_top_psm.csv")) |>
    arrange(batch, fraction, scan)
```

A quick visualization of the TMT fragment region can be made to show variation in the intensity observed for each TMT fragment for a given peptide.

As usual, I first transfer from Bioconductor data structures to vectors or other classes that can be made tidy if needed:

```
# Batch 1 Fraction 5, APLDNDIGVSEATR 2+ is index 9 in
the sorted table
control_index = 9

raw_file_name <- file.path("large-
data","MSV000086195","ccms_peak",

paste0(control_id$base_filename[control_index],
".mzML"))

selected_control <- Spectra(raw_file_name)

cent_prec_scan <-
selected_control[control_id$scan[control_index]]
cent_x <- mz(cent_prec_scan)[[1]]
cent_y <- intensity(cent_prec_scan)[[1]]
```

Now plot the TMT fragments to see if anything surprising seems to be going on (Figure 5.3):

```r
p_tmt_frag <-
    ggplot() +
    coord_cartesian(xlim=c(126.25,131.25),
ylim=c(0,1.5e5)) +
    scale_y_continuous(labels = inten_label) +
    geom_segment(aes(x=cent_x, y=cent_y, yend = 0, xend
= cent_x),
                 linewidth = 0.5, color=pal$blue) +
    xlab("m/z") +
    ylab("Intensity") +
    theme_classic() +
    theme(plot.title = element_text(hjust = 0.5)) +
    theme(plot.subtitle = element_text(hjust = 0.5)) +
    ggtitle(label = "MSV000086195 Batch 1 Fraction 5
(Scan 21516)",
            subtitle = "MS2 APLDNDIGVSEATR 2+
(Deaminidated N, TMT10plex)")

print(p_tmt_frag)
```

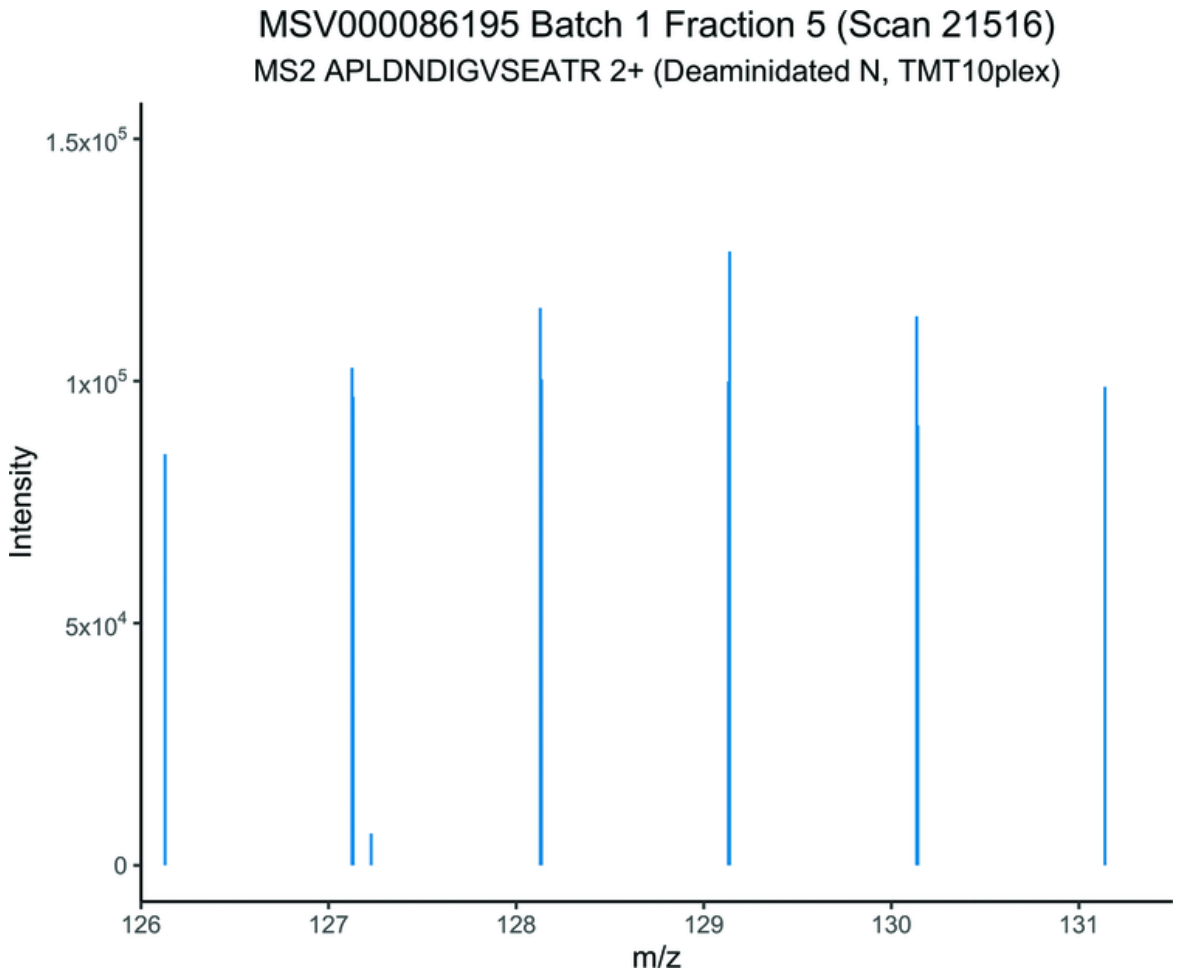


Figure 5.3 TMT 10plex fragments from MSV000086195 Batch 1, Fraction 5 (Scan 21516).

Remember that the precursor ion m/z was selected from an MS spectrum, and that ion actually represented 10 different molecules, one from each of the 10 pooled samples. Each labeled peptide has exactly the same chemical formula and, thus, the same monoisotopic mass. The idea behind TMT and other isobaric labeling methods is that when the labeled ion is fragmented, the tag fragments into different m/z values based on the location of labeled atoms in the tag. The spectrum in 5.3 is from a peptide spiked into each sample at the same concentration. So, several things could be going on in this spectrum. First, the actual amount of internal control added to each sample could be different. That would show up as different intensities of each peak. It could also be that the

labeling efficiency or the fragmentation efficiency could be different for the 10 different tags. Regardless of the cause, the variation seen in the internal control represents the variance in the experimental process, not biological variation.

The variation of all the internal control signals can be evaluated statistically using analysis of variation (ANOVA) to determine if the internal control shows that the raw intensity values of the human samples can be used to determine differences in expression or if steps need to be taken to remove the experimental variation through some normalization technique before analyzing unknown samples.

Even though this is a proteomics study, the same approach can be used for almost any other mass spectrometry measurement. When quantification, either relative or absolute, is needed, evaluating internal controls and dealing with run-to-run variations in instrument performance is an important consideration.

5.3.1 Evaluating the Internal Control Samples

To evaluate how well the experiment was controlled by analysis of the internal controls, first I need to collect the intensities of the TMT 10plex fragments. In this section, I will show how to do this for the general case, and later I will show how to use Bioconductor to get intensities for popular labeling reagents like TMT.

The `get_tmt_inten()` function below extracts the label intensity from a spectrum, where `x` is the m/z vector and `y` is the intensity vector. Depending on the resolution of the instrument and the stability of the mass calibration, I've used `min_x` and `min_y` parameters to specify the window around which to capture the maximum intensity peak.

```
# This function gets the TMT fragment intensity from
the spectrum
# Several things could occur:
# - there could be more than one fragment in the mass
range: report the max
# - there could be no fragment (length = 0): report 0
intensity

get_tmt_inten <- function(x, y, min_x, max_x) {

    tmt_inten <- y[between(x, min_x, max_x)]

    if(length(tmt_inten) > 0) {
        tmt_inten <- max(tmt_inten)
    } else {
        tmt_inten <- 0
    }
    tmt_inten
}
```

As with the accumulator used in Section 3.6, I'll create an empty tibble, defining the column names and types. When the reporter intensities are found for a particular spectrum, they will be added to this table.

```
# Create an accumulator tibble

get_empty_tmt <- function() {
    tibble(
        sample_type = character(),
        data_type = character(),
        pep_seq  = character(),
        batch    = numeric(),
        fraction = numeric(),
        scan_num = numeric(),
        tmt_126  = numeric(),
        tmt_127N = numeric(),
        tmt_127C = numeric(),
        tmt_128N = numeric(),
        tmt_128C = numeric(),
        tmt_129N = numeric(),
        tmt_129C = numeric(),
        tmt_130N = numeric(),
        tmt_130C = numeric(),
        tmt_131  = numeric()
    )
}
```

The accumulator defined above is meant for the TMT 10plex label but could be used for any type of quantification where selected m/z intensities represented quantity. It could also be adapted to handle other experimental designs. For example, if fractionation was not part of the design, there would be no need to track that information, and that field could be left out.

Next, the `get_reporters()` function collects the intensity values for the specific label reporters. Note that this function should match the structure of the accumulator tibble. It doesn't have to fill every field, but the collected intensities should match.

```
get_reporters <- function(selected, id, index,
type_name) {
    cent_prec_scan <- mzR::peaks(selected,
id$scan[index])
    cent_x <- cent_prec_scan[,1]
    cent_y <- cent_prec_scan[,2]

    tmt_inten <- tibble(
        sample_type = type_name,
        data_type = "raw",
        pep_seq  = id$seq[index],
        batch    = id$batch[index],
        fraction = id$fraction[index],
        scan_num = id$scan[index],
        tmt_126  = get_tmt_inten(cent_x, cent_y,
126.127, 126.1295),
        tmt_127N = get_tmt_inten(cent_x, cent_y,
127.124, 127.126),
        tmt_127C = get_tmt_inten(cent_x, cent_y,
127.130, 127.1325),
        tmt_128N = get_tmt_inten(cent_x, cent_y,
128.127, 128.1295),
        tmt_128C = get_tmt_inten(cent_x, cent_y,
128.133, 128.1365),
        tmt_129N = get_tmt_inten(cent_x, cent_y,
129.130, 129.1325),
        tmt_129C = get_tmt_inten(cent_x, cent_y,
129.136, 129.139),
        tmt_130N = get_tmt_inten(cent_x, cent_y,
130.133, 130.1365),
        tmt_130C = get_tmt_inten(cent_x, cent_y,
130.140, 130.1425),
        tmt_131  = get_tmt_inten(cent_x, cent_y,
131.137, 131.1395)
    )
    tmt_inten
}
```

The mass ranges passed into `get_tmt_inten()` are empirically derived. Another approach, used in a later example, focuses

on the exact m/z value of the reporter and uses a window width that represents the mass error of the instrument. Both approaches work, but it is possible for one spectrum to have a higher m/z error than expected, which would result in missing a reporter intensity.

In the following code segment, each raw file in the `control_id` table created earlier is loaded, and the reporter intensities are extracted with the `get_reporters()` function. Here, since I only need the m/z and intensity values and I'm not interacting with any of the other Bioconductor packages, I am using the low- level `openMSfile()` function from the `mzR` package. In my experience, loading and accessing data from very large XML files can create disk input/output problems for some operating systems. Using the simplest function available and promptly closing the file gives the most stability and the fastest operation of several options available.

The files from this study were deposited in the mzML format and are very large. Because of the size of the files, I will put the reading code in a separate function called `mzML_read()` and use the `retry()` function from the `retry` package in the main loop to instruct the program to attempt another read of the file if an input/output error occurs.

```
mzML_read <- function(full_path) {
    mzML_file <- mzR::openMSfile(full_path, backend =
"pwiz")
    return(mzML_file)
}
```

And now the reading function is used to loop over all the raw data files. To try and maximize the stability of reading so many large mzML files, after each pass through the loop, I close the open file, delete the variable holding the file information, and explicitly call the garbage collector `gc()` function. Deleting a variable does not free up memory until

the garbage collector is called. While this approach adds extra time to the loop, it can prevent memory problems while reading large files in a loop.

```
library(retry)

ic_tmt_inten <- get_empty_tmt()

n_iterations <- length(control_id$batch)
pb <-
knitrProgressBar::progress_estimated(n_iterations)


for(control_index in 1:length(control_id$batch)) {

    knitrProgressBar::update_progress(pb)
    raw_file_name <- file.path("large-
data","MSV000086195","ccms_peak",

paste0(control_id$base_filename[control_index],
                          ".mzML"))

    selected_control <- NULL

    # If there is an error reading the mzML file, it's
probably an I/O timing
    # problem so just try again.

    selected_control <- retry(mzML_read(raw_file_name),
                              until = ~ !is.null(.),
                              interval = 10,
                              max_tries = 3)

    tmt_inten <- get_reporters(selected_control,
                               control_id,
                               control_index,
                               "control")
    ic_tmt_inten <- bind_rows(ic_tmt_inten, tmt_inten)
    mzR::close(selected_control)
    rm(selected_control)
    gc()
}

rm(tmt_inten)
```

The design of this particular experiment makes each batch unique rather than being process or technical replicates. That means the batches contain different individual samples and cannot be evaluated statistically without knowing which reporters go with which individuals in each batch. It is true, however, that for any single batch, it is probable that there is a mix of master athlete (MA) and nonathlete (NA) samples in each batch. So, if the internal control can be shown to be consistent between samples in a single batch, then variations within that batch could be due to biological variation. This is an important step. The degree to which the internal controls are statistically consistent sets the bounds for any variation between samples that can be attributed to biological causes – rather than process or instrumental variability.

The next step in the analysis, therefore, is to get the intensities for Batch 1 and drop the batch and peptide sequence information since all the peptides I want to look at are from a single control protein.

```
ic_tmt_inten <- dplyr::filter(ic_tmt_inten, batch==1)
|>
    dplyr::select(-c(batch, pep_seq))
```

Since the internal control was added to every sample, there should be no peptides that are missing a reporter intensity. It could be that the m/z value shifted or that the fragmentation resulted in insufficient intensity to get a signal, but regardless, since I want to perform a log base two transformation, values of zero have to be eliminated.

In this code chunk, I use the `apply()` function to vectorize the operation on columns 5–14 of the table, which hold the intensity values.

```
row_sub <- apply(ic_tmt_inten[,5:14], 1, function(row)
all(row != 0 ))
ic_tmt_inten <- ic_tmt_inten[row_sub,]
```

Commonly, a log base two transformation is used when intensity values have a very wide dynamic range. Performing a log transformation is sometimes all the normalization needed depending on the stability of the instrument. It also helps when using statistical methods sensitive to big differences in the scale between measurements. The log transformation can use any base, like `10`, or the natural log (*e*); however, base `2` is commonly used in biological measurements since a power of `2` represents a doubling, or *fold change* in the intensity. Fold changes have become the most common way of describing the differential expression of biological molecules, but it is just a convention. Mathematically, any log transformation would achieve the goal of bringing all the values of a measurement into a similar numeric range. The transformation is performed on columns 5–14 because, in this table, they contain the raw intensity values for the 10 reporter ions.

```
ic_tmt_log2 <- ic_tmt_inten |>
    dplyr::mutate(across(5:14, log2))
```

The shape of the `ic_tmt_log2` table is *wide* with respect to the reporters. In other words, each reporter is a column, and the values in the column are the intensities of each observation. To perform comparisons between reporters (samples), the table has to be changed so that a new column holds the name of the reporter and another holds the intensity. This means changing the shape of the table from wide to long. The `tidyverse` package `tidyr` has many functions for changing the shape of a table. The function needed for this transformation is the `pivot_longer()` function. The inverse function, `pivot_wider()`, is used to go back to a wider table. Changing

the shape to a longer table means going from 58 rows and 14 columns to 580 rows and 6 columns. The `!c(sample_type,...)` statement tells `pivot_wider()` to keep repeating those variables. It then repeats each of the 10 reporter names into `reporter` and puts the values into a column called `intensity`, which creates the 58 × 10 length and collapses the 10 reporter columns into 2.

```
ic_tmt_table <- ic_tmt_log2 |>
            tidyr::pivot_longer(!c(sample_type,
data_type, fraction, scan_num),
                        names_to = "reporter", values_to
= "intensity") |>
            dplyr::mutate(across(1:3, as_factor)) |>
            dplyr::mutate_at('reporter', as_factor)
```

Now the data are in a format that the `geom_boxplot()` layer can create a boxplot showing the mean and quartile information for each reporter using `ggplot2`.

```
p_boxplot_tmt <- ic_tmt_table |>
    ggplot() +
    geom_boxplot(aes(reporter, intensity)) +
    xlab("Reporter") +
    ylab("log2(intensity)") +
    theme_classic() +
    theme(axis.text.x = element_text(angle = 45, hjust
= 1)) +
    theme(plot.title = element_text(hjust = 0.5, vjust
= 2)) +
    ggtitle(label = "Raw Internal Control Intensity -
Batch 1")

print(p_boxplot_tmt)
```

Figure 5.4 shows that there is variation in the medians of the reporters even after the log transformation. However, all of the means seem to be within the first and third quartiles

(25th and 75th percentiles), but this does not say that they were or were not drawn from the same population, which, in this experiment, they are supposed to. So, the next step is to perform a hypothesis test to see if the variation between the internal control samples could have occurred just by chance, especially since there are only 58 observations for each reporter.

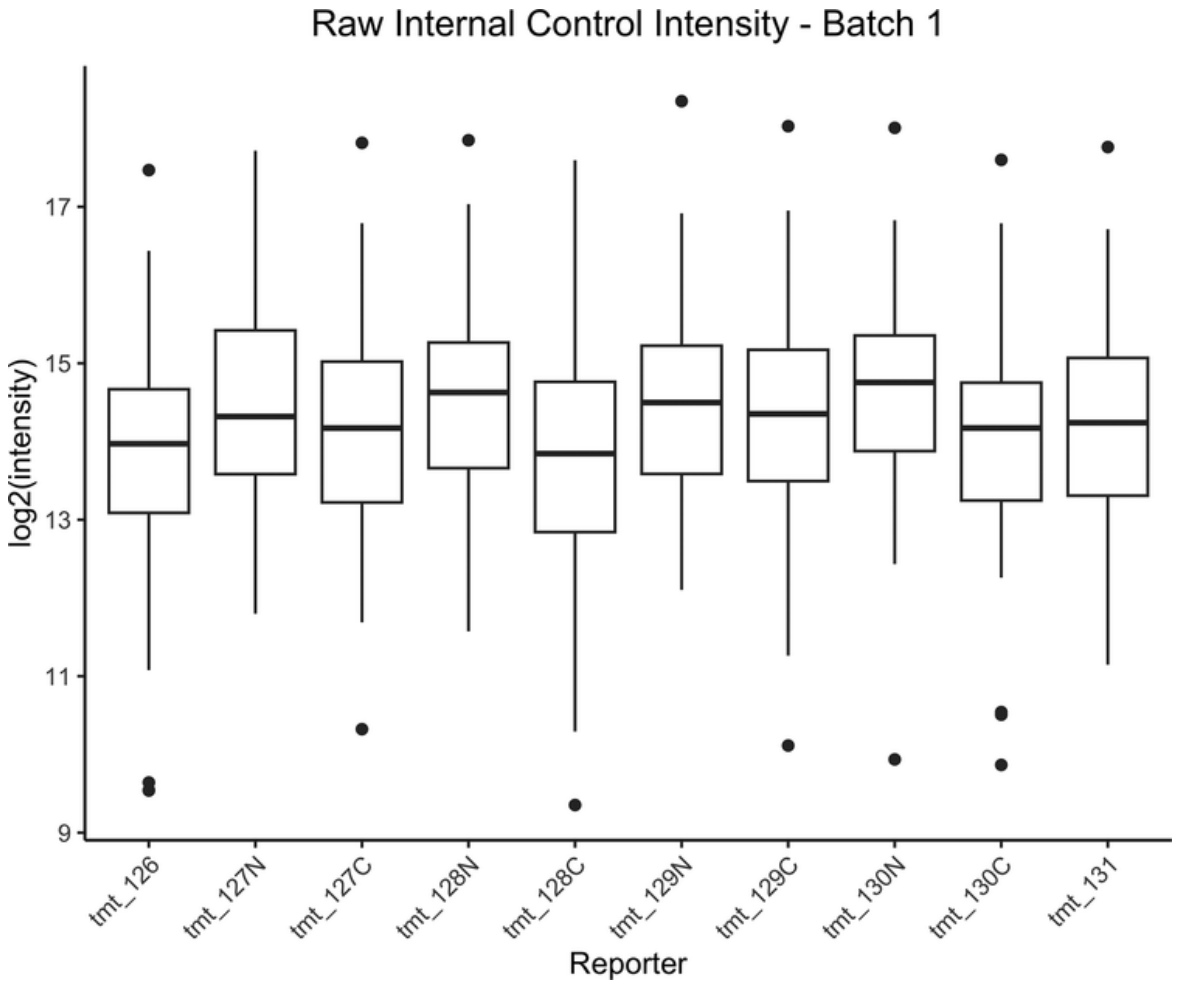


Figure 5.4 Raw intensity for all reporters of the internal control peptides for Batch 1.

There are base R functions for performing hypothesis testing, but in the following section I will use a resampling approach using the `tidymodels` [134] `infer` package to perform an ANOVA test [135], which is described in both the package

documentation and the book *Statistical Inference via Data Science: A ModernDive into R and the Tidyverse* [136].

5.3.2 Hypothesis Testing Using Resampling ANOVA

The approach to hypothesis testing that I will use is based on resampling and implemented via the `infer` package. For all hypothesis testing, regardless of the method used, the steps are the same [137, 138]. First, you pick a test statistic that is computed from the observed data, and then you compare that statistic to the range of values that could be generated by the null hypothesis. Finally, you compute the probability of obtaining the test statistic from the observed data by chance by comparing it to the null hypothesis distribution.

Hypothesis testing can be confusing when using the classical statistical approach and nomenclature. Thankfully, in the age of cheap computing, the process can be simplified. First, you have to carefully choose what test statistic is appropriate for the question you are asking. Luckily, in this example, the test statistic and null hypothesis are straightforward. I want to know if the mean values of the intensities of the 10 internal control observations differ from each other. The test statistic for this question is the F-score:

$$F = \frac{\text{Between group variability}}{\text{Within group variability}} \tag{5.2}$$

I can compute the F-score for the reporters, and then compare it to the F-score I get when I randomly assign the observed intensities to a label. Randomly assigning intensities to reporters is a way of stating the null hypothesis: there is no relationship between the reporter and the intensities observed for that reporter.

In the `infer` package, this is done by chaining a set of steps. First, specify the relationship being tested. This is passed to `specify()` using the traditional syntax for formula definition in R. Second, set the hypothesis, in this case, that the null hypothesis is that intensity is `independent` of `reporter`, and then state what test statistic to compute. In this case, the F value defined in Eq. (5.2) is the right test statistic.

```
raw_observed_f_statistic <- ic_tmt_table |>
    specify(intensity ~ reporter) |>
    hypothesize(null = "independence") |>
    calculate(stat = "F")
```

Now, I have the F-score from the observed data. Next, compute the F-score from 5000 repetitions of randomizing the intensities among the reporters.

```
raw_null_dist <- ic_tmt_table |>
    specify(intensity ~ reporter) |>
    hypothesize(null = "independence") |>
    generate(reps = 5000, type = "permute") |>
    calculate(stat = "F")
```

The `infer` package has a simple visualization that overlays the observed F-score from the distribution of possible F-scores that could be generated if the `intensity` and `reporter` were independent.

```
p_raw_null_dist <- raw_null_dist |>
    visualize() +
    shade_p_value(raw_observed_f_statistic, direction =
"greater") +
    theme_classic() +
    theme(plot.title = element_text(hjust = 0.5)) +
    xlab("F-Statistic") +
    ylab("Count") +
    theme(plot.title = element_text(hjust = 0.5)) +
    theme(plot.subtitle = element_text(hjust = 0.5)) +
    ggtitle(label = "Sampling-Based Null Distribution
(n=5000)",
            subtitle =
            paste0("Raw Internal Control Intensities: F
= ",
                   round(raw_observed_f_statistic,
digits=4))
            )

print(p_raw_null_dist)
```

The observed F-score seen in Figure 5.5 seems large compared to the distribution of F-scores created by random, but to get the probability of a score that large being observed by random, I need to calculate the p-value. The `get_p_value()` function performs this calculation and takes an argument to determine what is being calculated – in this case, the probability of getting a value "greater," which is assigned to the `direction` parameter.

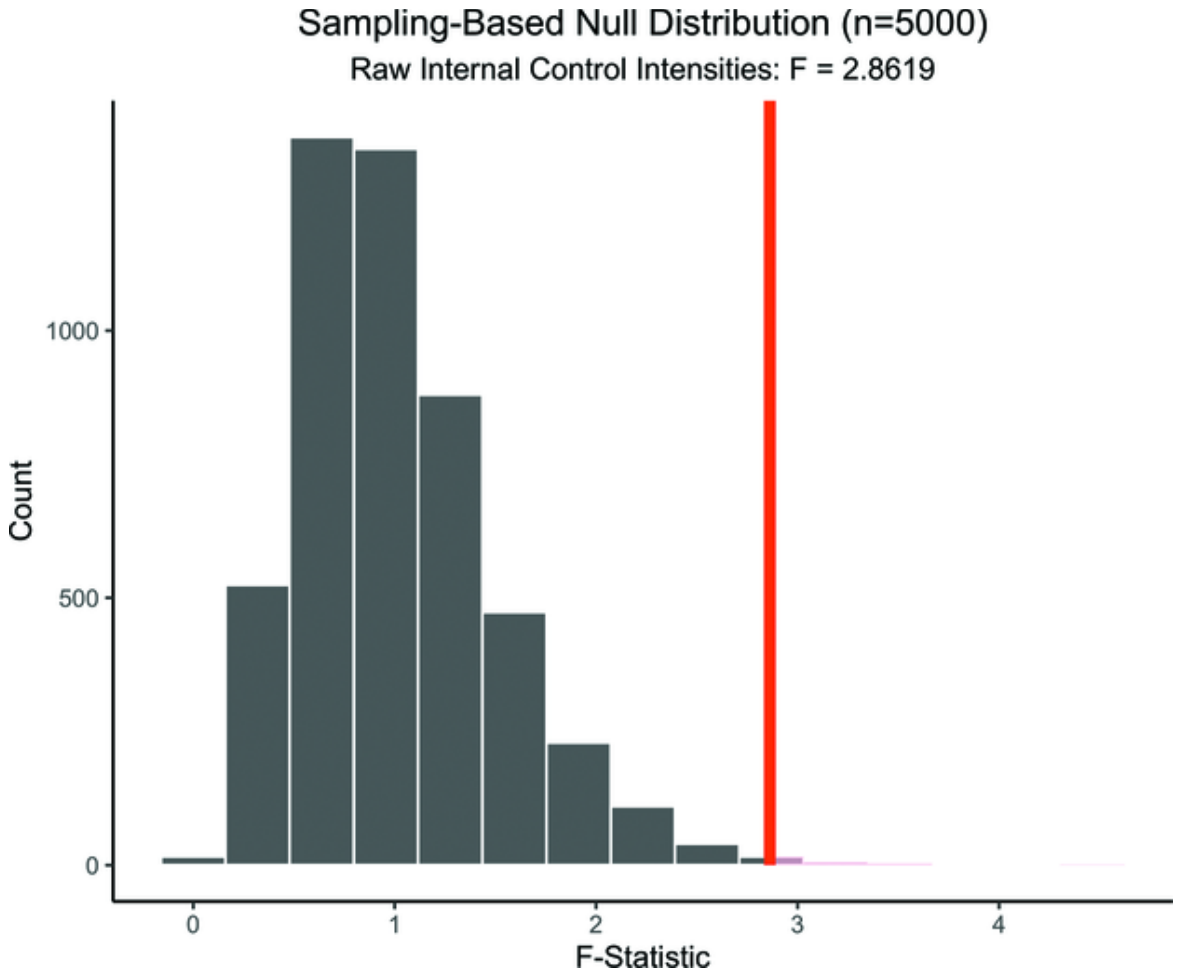


Figure 5.5 Overlay of the observed F-score with the distribution of F-scores for the null hypothesis that the intensity and reporter are independent.

```
# Tidymodels method for computing the p-value for the
observed f-statistic
raw_p_value <- raw_null_dist |>
  get_p_value(obs_stat = raw_observed_f_statistic,
direction = "greater")

raw_p_value
```

```
## # A tibble: 1 x 1
##   p_value
```

```
##     <dbl>
## 1  0.0022
```

Historically, when the probability of getting the observed F-score is below 0.05, it is taken to mean that it is very unlikely that the observed data occurred by random chance. Since the p-value for getting the F-score I saw is much less than 0.05, I can reject the null hypothesis as the explanation of the differences in the means and state that I am more than 95% confident that there is a relationship between the reporter intensity mean and the reporter. Since the reporters I am analyzing are all internal controls, there is no biological reason for any of the means to be different, which suggests that the variation is coming from the experimental process.

There are many ways to attempt to remove the nonbiological variation from experimental measurements to give a more accurate value for signals that may be different for biological reasons. The process is called *normalization* and is almost always required when using mass spectrometry for quantitative or relative quantity measurements. In the next section, I will show one way of performing intensity normalization and then use the resampling ANOVA approach to evaluate the normalized data.

5.3.3 Normalizing Intensity Data

One of the more powerful normalization methods is called *quantile normalization*, originally described by Bolstad et al. [139]. The basic idea is to adjust the intensity values of each sample to create a single averaged distribution of intensities from which each peak intensity is drawn. In this experiment, each fraction 1–33 was a separate injection and prepared separately, so differences between the distribution of intensities of each injection can reasonably be assumed to be systematic variation, not biological. This assumption seems to hold for proteomics and other measurements where the

majority of compounds do not vary in concentration between individuals. This normalization technique also requires a large number of measurements to give effective results. In Chapter 6, I will show an alternative normalization technique that uses internal controls to correct individual measurements in chromatography experiments using stable-label isotope internal standards.

In indirect normalization, the expectation is that post-normalization, the internal controls will show no significant relationship between the mean intensity and the reporter. Once experimental variation has been removed, compounds can be tested for biological variation between individuals.

In Section 5.3.1, I showed a general method for obtaining selected values for peaks of interest (reporter fragments in this case) directly from spectra without using external libraries. To perform this type of operation on an entire batch is more efficient using the Bioconductor function `quantify()` loaded by the `MSnbase` package. Conveniently, `MSnbase` also includes the required reporter information for the experiment I am examining, which will be used as an argument to `quantify()`. In this case, the `TMT10HCD` set matches what was given in the Ubaida-Mohien et al. paper. Further, the `MSnbase` package also includes the `normalise()` function from Bolstad's `preprocessCore` package [140], which includes several normalization techniques besides the quantile method. Depending on your experimental design, other methods may give better results. For this example, I'll stick to the quantile method.

First, I'll create a slightly different structure to hold the TMT information. This structure includes the precursor ion intensity, which can also be used to normalize MS level 2 data in some cases.

```
get_empty_tmt_norm <- function() {
    tibble(
        sample_type = character(),
        data_type = character(),
        fraction = numeric(),
        scan_num = numeric(),
        precursorInten = numeric(),
        tmt_126  = numeric(),
        tmt_127N = numeric(),
        tmt_127C = numeric(),
        tmt_128N = numeric(),
        tmt_128C = numeric(),
        tmt_129N = numeric(),
        tmt_129C = numeric(),
        tmt_130N = numeric(),
        tmt_130C = numeric(),
        tmt_131  = numeric()
    )
}
```

For this normalization example, I will only look at Batch 1. This is because in their manuscript, Ubaida-Mohien et al. state that there were 24 individuals tested by liquid chromatography with tandem mass spectrometry (LC-MS/MS). With three batches, there are 30 potential slots, so 24 individuals were randomly divided between the batches, and the remaining six samples were randomly chosen as technical replicates. That means that each batch contains different individuals, and the batches cannot be combined to test the significance of any particular reporter intensity except for the internal control. Even though the exact experimental design was not given in the paper or placed in the MassIVE repository, I am limiting the analysis to the first batch. It is unlikely that a single batch contains individuals of one group by random. In this study, there were two groups: MA and NA. It is also unlikely that a single batch contains all the technical replicates, although both are theoretically possible.

```
  result_files <- list.files(
      file.path("large-data",
  "MSV000086195","ccms_peak"),
      pattern = "_Batch1_")
```

To perform intensity normalization, I will use the `normalise()` function, and instead of manually finding the reporter intensities, I will use the `quantify()` method, which has a built-in list of m/z values for the TMT10plex reporters. The method used above will work for any reporter m/z values (or any specific m/z values besides labeled reporters). Be sure to check the help page for `quantify()` for a list of the mass tags supported.

Since the `quantify()` method is only looking at the m/z region of the named mass tag, I decided to shorten the spectra to only the reporter m/z range using `filterMz()`. This is not strictly necessary, but passing the smallest amount of data needed into a function is a habit I tend to follow.

The function `quantify()` returns an `MSnSet` object, which is passed to `exprs()` that extracts the intensity values. The `exprs()` function is from the `Biobase` package and is an example of how the mass spectrometry packages in Bioconductor are built on packages from other high-throughput expression packages. The `MSnSet` object directly extends the class `eSet` (expression set), which stores data in S4 slots meant to be extracted by accessor functions.

```
  getClassDef("MSnSet")
```

```
## Class "MSnSet" [package "MSnbase"]
##
## Slots:
##
## Name:      experimentData     processingData
qual
```

```
## Class:              MIAPE         MSnProcess
data.frame
##
## Name:           assayData          phenoData
featureData
## Class:          AssayData AnnotatedDataFrame
AnnotatedDataFrame
##
## Name:          annotation       protocolData
.__classVersion__
## Class:          character AnnotatedDataFrame
Versions
##
## Extends:
## Class "eSet", directly
## Class "VersionedBiobase", by class "eSet", distance 2
## Class "Versioned", by class "eSet", distance 3
```

Many different measurement types are stored in a class derived from `eSet`, and `exprs()` doesn't care which type of `eSet` you pass it. When passed the output of `quantify()`, the `exprs()` function returns a matrix with rows representing scans (*features* in the Bioconductor nomenclature) from the raw data file and columns representing the 10 reporters (called *samples* by Bioconductor). The matrix created by `exprs()` needs to be named, so I added the reporter names as syntactically correct strings for use with `tidyverse` and `tidymodels` packages. In the same loop, I also perform the step of calling `normalise()` in order to get the normalized intensity values. Here, staying in the Bioconductor domain longer than I have in previous examples helps since `quantify()`, `normalise()`, and `exprs()` were all designed to work together. Only after I have the results I want from these functions will I move back over to the `tidyverse` approach.

First, create a vector to hold the reporter names. I'll use this when creating a tidy version of the matrix generated by `exprs()`.

```
reporters <- c("tmt_126", "tmt_127N", "tmt_127C",
"tmt_128N", "tmt_128C",
                "tmt_129N","tmt_129C", "tmt_130N",
"tmt_130C", "tmt_131")
```

The following is a big chunk of code. It could be made to appear shorter by breaking some parts into separate functions, but I've kept everything in line to reduce redundant steps that take a long time when files are big. It's important to know when to break things up and when to keep them together, and it's more art than science. Generally, you should follow the DRY principle ("don't repeat yourself") [141] and minimize code duplication. However, Donald Knuth popularized another important principle in programming: "Premature optimization is the root of all evil" [142] and premature abstraction should also be avoided [143].

```r
# This could be a long-running process to read and
quantify all of the data
# from the 33 files in a single batch - so use a
progress bar
n_iterations <- length(result_files)
pb <-
knitrProgressBar::progress_estimated(n_iterations)

# Start with empty tables to accumulate results into
tmt_quant <- get_empty_tmt_norm()
tmt_norm <- get_empty_tmt_norm()

# Loop over all the raw files in batch 1
for(i in 1:length(result_files)) {

    # Get the files in fraction order (not alphabetic
sort order)
    file_name <-
result_files[grepl(paste0("Fr",i,".mzML"),result_files)
]

    mzML_file_name <- file.path("large-data",
"MSV000086195","ccms_peak",
                                file_name)

    ms_file <- readMSData(mzML_file_name, msLevel = 2,
mode = "onDisk")

    # Since the reporters are all in the 126-132 m/z
range, give the
    # quantify() function less to deal with
    ms_file <- filterMz(ms_file, c(126, 132))

    # quantify the observed TMT fragment tags and
report the peak maximum
    qty <- retry(quantify(ms_file, method="max",
TMT10),
                 when = "Can not open file",
                 interval = 10,
                 max_tries = 5
                )
```

```
    rm(ms_file)
    gc()

    # return the intensities using the `exprs()`
function and
    # fix the column names to be syntactically correct
    intensities <- exprs(qty)
    colnames(intensities) <- reporters

    # combine the columns and then drop missing values
    # turn the output into a tibble and change the
column types
    # to the appropriate types and finally convert all
intensities to
    # their log2() value
    quant <- cbind(sample_type="",
                   data_type = "raw",
                   fraction=i,

scan_num=qty@featureData@data$seqNum,

precursorInten=qty@featureData@data$precursorIntensity,
                   intensities) |>
        na.omit() |>
        as_tibble(.name_repair = "unique") |>
        dplyr::mutate(across(3:15, as.numeric)) |>
        dplyr::mutate(across(1:2, as_factor)) |>
        dplyr::mutate(across(5:15, log2))

    # normalize the quantities using the standard
quantile normalization
    qty_norm <- normalise(qty, "quantiles")

    # get the intensities and perform the same steps to
get the
    # normalized version of the quantities
    inten_norm <- exprs(qty_norm)
    colnames(inten_norm) <- reporters

    norm <- cbind(sample_type="",
                  data_type = "norm",
```

```
                    fraction=i,
                    scan_num=qty@featureData@data$seqNum,

  precursorInten=qty@featureData@data$precursorIntensity,
                    inten_norm) |>
          na.omit() |>
          as_tibble(.name_repair = "unique") |>
          dplyr::mutate(across(3:15, as.numeric)) |>
          dplyr::mutate(across(1:2, as_factor)) |>
          dplyr::mutate(across(5:15, log2))

      # Bind the just created row to the final output
      tmt_quant <- rbind(tmt_quant, quant)
      tmt_norm  <- rbind(tmt_norm, norm)

      # update the progress bar
      knitrProgressBar::update_progress(pb)
  }
```

The variables `tmt_quant` and `tmt_norm` contain the raw and normalized reporter intensities for every scan in Batch 1. I'd like to do a quick check on what the normalization process actually did. There are many ways to normalize data of the type in this example, so if quantile normalization doesn't seem to be removing the nonbiological variation from the internal controls, I can choose from the other options for the `method` parameter. The `MSnbase` documentation for the `normalise()` function gives several different options to choose from.

In the following, I will check the intensity distribution using the `stats` function `density()`, which is a kernel density estimation method. The output is then plotted in Figure 5.6.

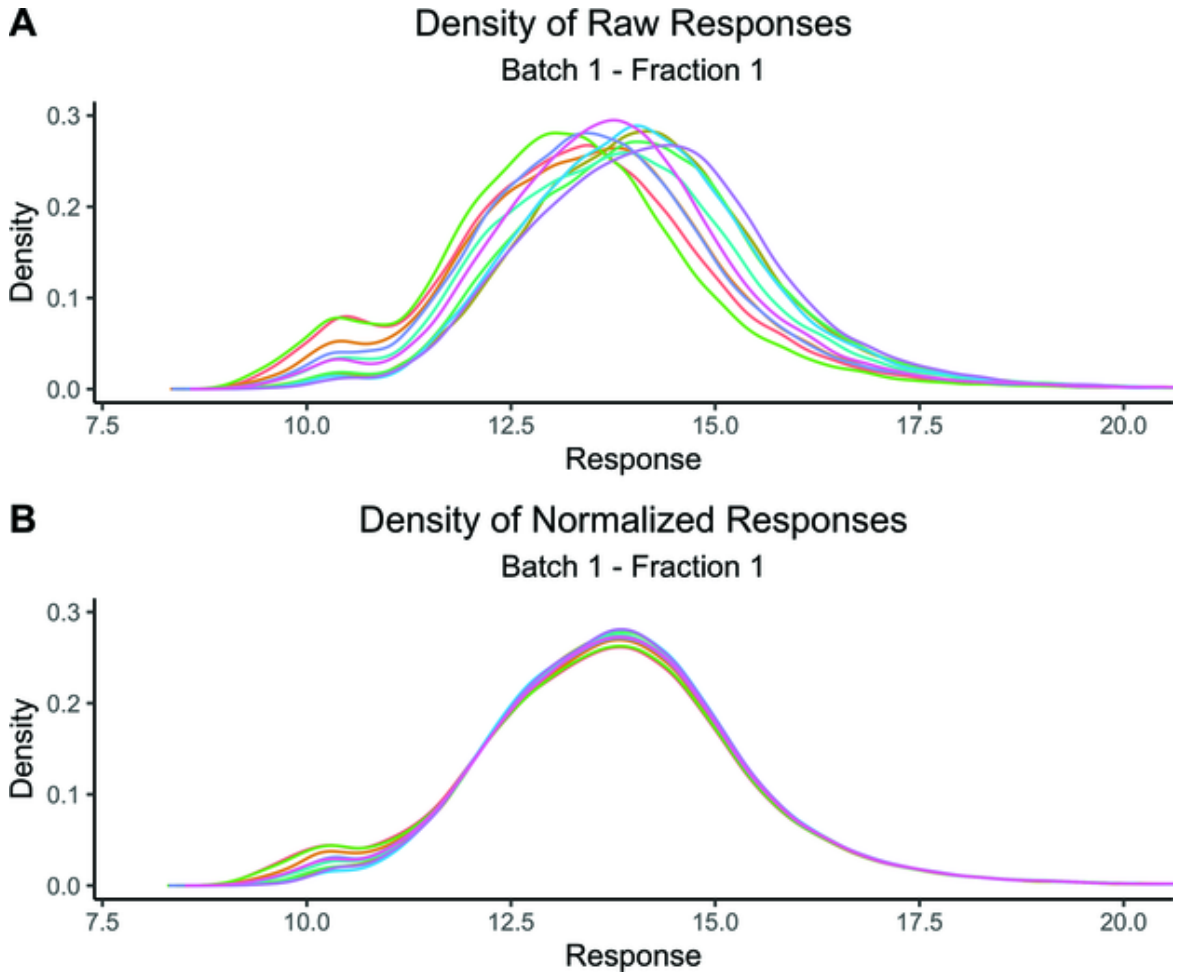


Figure 5.6 Comparison of raw and normalized responses for all the reporters in Fraction 1 in Batch 1.

First, set up the names of the reporters (samples) and create the empty tables.

```
reporters <- c("tmt_126", "tmt_127N", "tmt_127C",
"tmt_128N", "tmt_128C",
                "tmt_129N","tmt_129C", "tmt_130N",
"tmt_130C", "tmt_131")

raw_density_table <- tibble(
                    reporter = character(),
                    x = numeric(),
                    y = numeric()
)

norm_density_table <- raw_density_table
```

The `tmt_quant` table stored the 10 reporters in columns `6-15` so the density calculation is performed with a loop recording the density of each reporter. Since this is just a quick check, I only want to make an overlay plot of one of the 33 fractions. That's why I have `tmt_quant$fraction==1` in the selection of the rows from the table. There is an interfacing requirement between the inputs to the `stats` package and the output of this type of selection using `dplyr`. To pass a vector into the `density()` function, I have to use the `dplyr` function `pull()` to convert the selected column (and rows) from a `tibble` to a vector.

```
for(i in 6:15) {
    raw_density <-
density(pull(tmt_quant[tmt_quant$fraction==1,i]),
na.rm=TRUE)
    raw_density_table <- rbind(raw_density_table,

tibble(reporter=reporters[i-5],
                                    x=raw_density$x,
                                    y=raw_density$y))
}
```

The density calculation is performed in the same way for the `tmt_norm` table. Again, for this quick look, I'm only checking

fraction 1.

```
for(i in 6:15) {
    norm_density <-
density(pull(tmt_norm[tmt_norm$fraction==1,i]),
na.rm=TRUE)
    norm_density_table <- rbind(norm_density_table,

tibble(reporter=reporters[i-5],
                                        x=norm_density$x,
                                        y=norm_density$y))
}
```

Now, the two plots can be constructed and then combined into a single figure using the `ggarrange()` function that's part of the `ggpubr` package.

```
fract1_raw <-
    ggplot(raw_density_table, aes(x=x, y=y)) +
        coord_cartesian(xlim=c(8,20), ylim=c(0,0.3)) +
        geom_line(aes(color=reporter)) +
        xlab("Response") +
        ylab("Density") +
        theme_classic() +
        theme(legend.position="none") +
        theme(plot.title = element_text(hjust = 0.5)) +
        theme(plot.subtitle = element_text(hjust =
0.5)) +
        ggtitle(label = "Density of Raw Responses",
                subtitle = "Batch 1 - Fraction 1")

fract1_norm <-
    ggplot(norm_density_table, aes(x=x, y=y)) +
        coord_cartesian(xlim=c(8,20), ylim=c(0,0.3)) +
        geom_line(aes(color=reporter)) +
        xlab("Response") +
        ylab("Density") +
        theme_classic() +
        theme(legend.position="none") +
        theme(plot.title = element_text(hjust = 0.5)) +
        theme(plot.subtitle = element_text(hjust =
0.5)) +
        ggtitle(label = "Density of Normalized
Responses",
                subtitle = "Batch 1 - Fraction 1")

p_norm_density <- ggarrange(fract1_raw, fract1_norm,
          ncol = 1, nrow = 2,
          labels = c("A", "B"))

print(p_norm_density)
```

From Figure 5.6a, there do appear to be differences in the distributions of responses from the internal controls in fraction 1 when I look at all 10 reporters. Since these are internal controls, this variation cannot originate in any of the

interesting biological variations this experiment is intended to discover.

It's clear from Figure 5.6b that quantile normalization changed the intensity responses so that any individual reporter intensity looks to have come from a much more similar distribution, which is what you would expect if most nonbiological variation was removed. Since these are internal controls, there is no biological variance, so now, when the normalized values are used, I should see no significant difference in the means, as was seen in Figure 5.4 and established by the F-test and the very low p-value for the null hypothesis. Now, I will repeat the tests with the normalized intensities for the internal control.

Starting with the full set of normalized responses, first replace the individual IC TMT intensities with the normalized intensities.

```
tmt_norm <- tmt_norm |>
    dplyr::select(-c(precursorInten)) |>
    dplyr::mutate(sample_type="control")
```

Next, match up the normalized values for the control samples with the fraction and scan number, and rearrange the table so the column order is the same as the raw quantitated data.

```
ic_norm <- ic_tmt_log2 |>
    dplyr::select(-c(contains('tmt'),sample_type,
data_type)) |>
    dplyr::left_join(tmt_norm, join_by(fraction,
scan_num)) |>
    dplyr::relocate(data_type) |>
    dplyr::relocate(sample_type)
```

Finally, bind the normalized and raw tables to make a boxplot to visualize the comparison:

```
ic_levels <- rbind(ic_tmt_log2, ic_norm)
```

Like before, this code pivots the responses into a shape that allows making boxplots and running an F-test. Also, here I make sure the `reporter` column created in the pivot is treated as a factor so that categorical statistics will handle the reporter names correctly.

```
ic_levels_tbl <- ic_levels |>
            tidyr::pivot_longer(!c(sample_type,
data_type, fraction, scan_num),
                        names_to = "reporter", values_to
= "intensity") |>
            dplyr::mutate(across(1:4, as_factor)) |>
            dplyr::mutate_at('reporter', as_factor)
```

The following creates a boxplot of each reporter for all of the normalized responses for the internal control peptides next to the raw intensity boxplot.

```
p_norm_boxplot <- ic_levels_tbl |>
    ggplot() +
    geom_boxplot(aes(reporter, intensity,
color=data_type)) +
    xlab("Reporter") +
    ylab("log2(intensity)") +
    theme_classic() +
    theme(axis.text.x = element_text(angle = 45, hjust
= 1)) +
    theme(plot.title = element_text(hjust = 0.5, vjust
= 2)) +
    theme(legend.position = "top") +
    theme(legend.title = element_blank()) +
    ggtitle(label = "Raw and Normalized Internal
Control Intensities - Batch 1")

print(p_norm_boxplot)
```

It certainly appears from Figure 5.7 that quantile normalization has moved the internal control responses toward a central value. The way to see if this is true is to repeat the F-test from Figure 5.5 on the normalized responses.

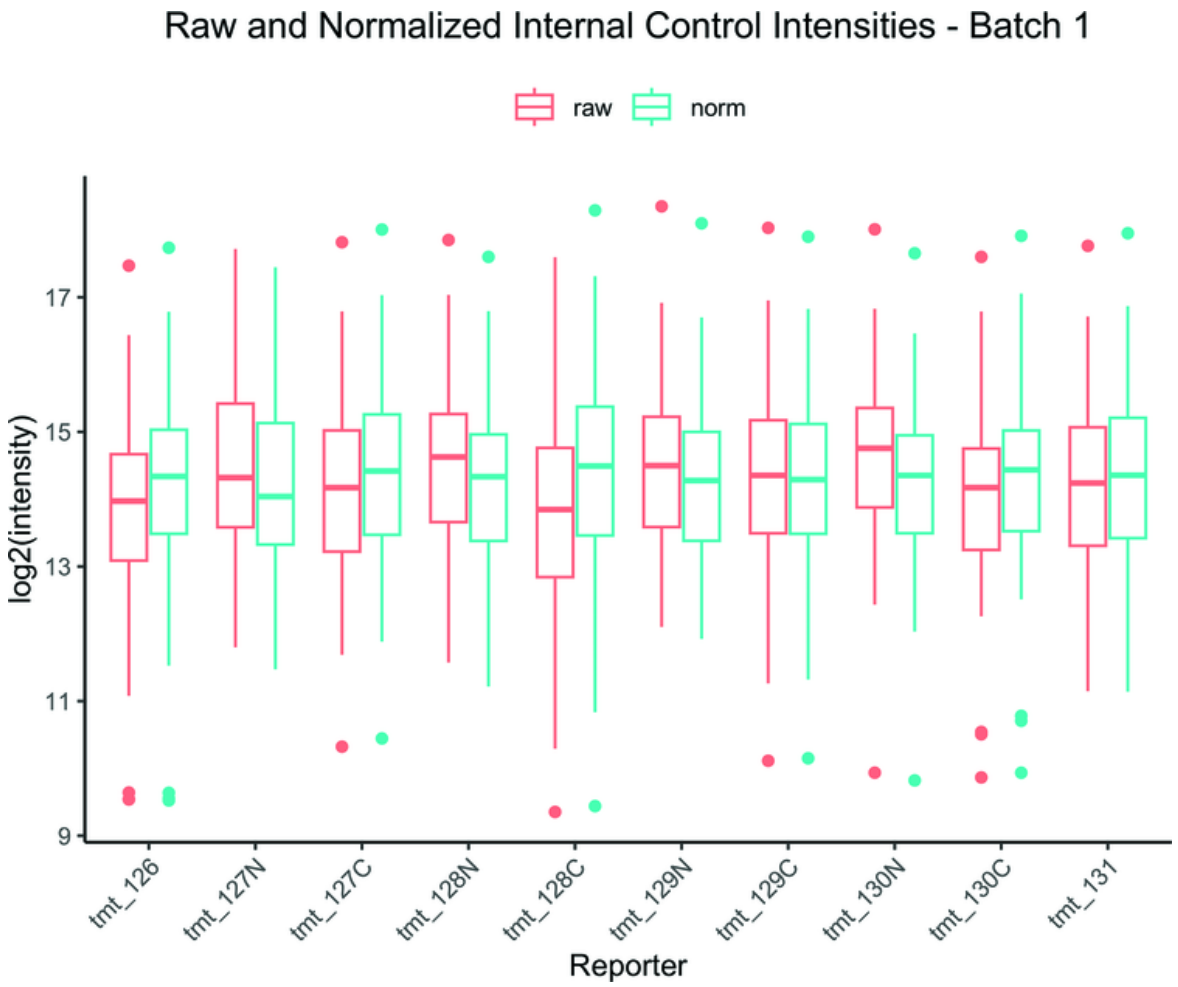


Figure 5.7 Comparison of raw and normalized responses for all the internal control reporters in Batch 1.

Compute the observed F-score from the normalized data.

```
norm_observed_f_statistic <- ic_levels_tbl |>
    dplyr::filter(data_type=="norm") |>
    specify(intensity ~ reporter) |>
    hypothesize(null = "independence") |>
    calculate(stat = "F")
```

Compute the distribution of F-scores from randomizing the normalized intensities (5000 different random assignments of intensity to the reporter).

```
norm_null_dist <- ic_levels_tbl |>
    dplyr::filter(data_type=="norm") |>
    specify(intensity ~ reporter) |>
    hypothesize(null = "independence") |>
    generate(reps = 5000, type = "permute") |>
    calculate(stat = "F")
```

Visualize the F-score for the normalized data and the distribution of F-scores for the null hypothesis (Figure 5.8).

```r
p_norm_null_dist <- norm_null_dist |>
    visualize() +
    shade_p_value(norm_observed_f_statistic, direction
= "greater") +
    theme_classic() +
    theme(plot.title = element_text(hjust = 0.5)) +
    xlab("F-Statistic") +
    ylab("Count") +
    theme(plot.title = element_text(hjust = 0.5)) +
    theme(plot.subtitle = element_text(hjust = 0.5)) +
    ggtitle(label = "Sampling-Based Null Distribution
(n=5000)",
            subtitle =
            paste0("Normalized Internal Control
Intensities: F = ",
                   round(norm_observed_f_statistic,
digits=4))
            )

print(p_norm_null_dist)
```

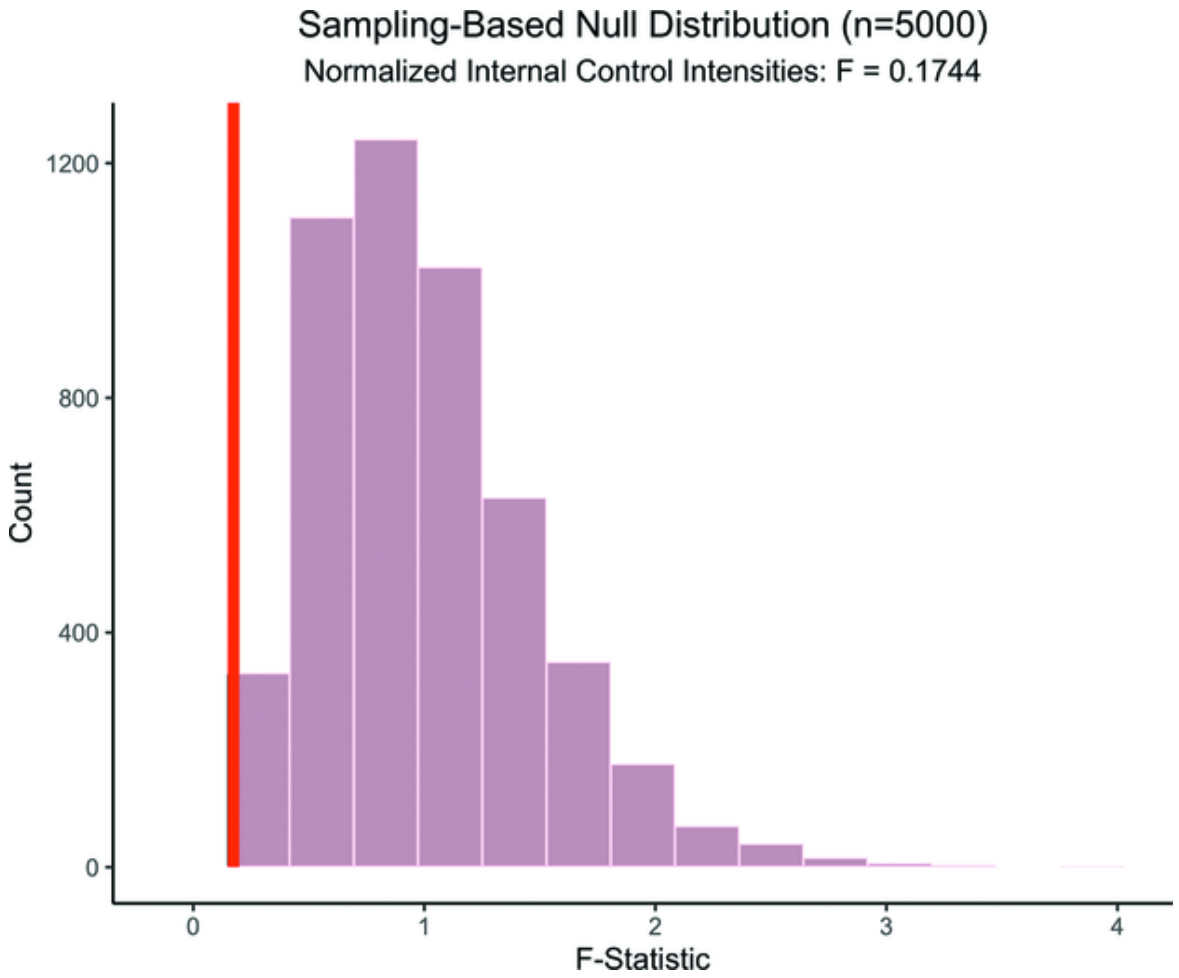


Figure 5.8 Overlay of the observed F-score with the distribution of F-scores for the null hypothesis that the intensity and reporter are independent.

And finally, compute the p-value for getting the observed F-score from the normalized data at random.

```
# calculate the p-value from the F statistic and null
distribution
p_value <- norm_null_dist |>
  get_p_value(obs_stat = norm_observed_f_statistic,
direction = "greater")

p_value
```

```
## # A tibble: 1 x 1
##   p_value
##     <dbl>
## 1   0.997
```

After normalization, the chance of getting an F-score as large as the observed value by random is over 99%, so I cannot reject the null hypothesis that the reporters and the intensities are independent. This means that the normalization process removed the variation from the internal control. However, if normalization removes both experimental *and* biological variation, it will have failed. The way to test that is to look at one of the proteins in the experiment where the authors found evidence for biological variation and see if it's still there.

5.3.4 Evaluating Peptide Reporter Intensity Variation

To conclude this chapter, I will analyze one of the proteins indicated in the Ubaida-Mohien et al. 2022 manuscript. The gene `MT-CO2` corresponds to the human protein Cytochrome C Oxidase Subunit 2 (COX2_HUMAN); its Uniprot ID is P00403 [144]. There are multiple peptides identified for this protein in all three batches of the study, which makes it a good candidate to dig into the details of how to apply the normalization methods described above to a signal expected to have biological variation. In the manuscript, the authors chose to use *median polishing*, which was described by Herbrich et al. in 2013 [145] and by Kammers et al. in 2015 [146]. My goal in this analysis has not been to reproduce the analysis of Ubaida-Mohien et al. but rather focus on the internal controls and what I could learn about how well-controlled the experiment was via an orthogonal approach. Sticking with that idea, I want to see if the normalization method I chose, quantile normalization, which removed the

experimental variation from the internal control, left any expected biological variation in proteins identified in the human samples.

The first step is to find all of the peptide spectra matches in the search performed by the authors. In this analysis, I'll focus on only one protein, but any protein could be used. In fact, an entire replicate analysis could be performed using this approach, but again, that's not my objective.

I'll want to break up the analysis much like I did in Chapter 3. Since the mzID files I am looking at now are different from the ones generated by the internal control search using X! Tandem, I'll need to validate the XML files before trying to parse them. This is one of the benefits of using XML for data like mzID files.

```
# XML Schema validation
mzID_valid <- function(mzID_filename, id_schema) {
    doc <- read_xml(mzID_filename)
    xml_validate(doc, id_schema)
}
```

Next, I define a function to produce an empty peptide spectrum match. The `.mzid` files deposited in `MassIVE` appear to have been generated by Mascot [147], a popular commercial peptide search engine. Since the scores and other data have different names in the controlled vocabulary used by mzID, I will create a Mascot-specific PSM table. You could make a generic one if that served your needs better.

```
# Create empty tibble to hold peptide-spectrum match
data
empty_psm <- function() {
    tbl_item <- tibble(
        batch = numeric(),
        fraction = numeric(),
        seq = character(),
        charge = numeric(),
        ex_mz = numeric(),
        calc_mz = numeric(),
        match_score = numeric(),
        exp_score = numeric(),
        scan = numeric(),
        rt = numeric(),
        mods = character(),
        base_filename = character()
    )
    tbl_item
}
```

Since the mzID files generated by Mascot use different controlled vocabulary terms, the variable names change from the X! Tandem example in Chapter 3.

```r
# Extract PSM data to be added to a table
extract_mascot_match <- function(psm) {
    m1 = str_match(psm$spectrum.title, "_Batch(\\d+)_")
    m2 = str_match(psm$spectrum.title,
"_BRPhsFr(\\d+)")

    tbl_item <- tibble(
        batch = as.numeric(m1[1,2]),
        fraction = as.numeric(m2[1,2]),
        seq = psm$pepseq,
        charge = psm$chargestate,
        ex_mz = psm$experimentalmasstocharge,
        calc_mz = psm$calculatedmasstocharge,
        match_score = psm$mascot.score,
        exp_score = psm$mascot.expectation.value,
        scan = psm$acquisitionnum,
        rt = as.numeric(psm$scan.start.time),
        mods = psm$modification,
        base_filename =
str_split_1(psm$spectrumFile,".mgf")[1]
    )
    tbl_item
}
```

The same holds for finding the top-scoring scans – the score is the `mascot.score` variable for these files.

```
top_mascot_scans <- function(all_id) {

    all_id <- all_id |>
        arrange(acquisitionnum)

    pep_score_best <- 0.0
    curr_scan <- all_id[1,]$acquisitionnum

    pep_tbl <- empty_psm()

    for(i in 1:length(all_id$pepseq)) {
        peptide <- all_id[i,]
        pep_scan <- peptide$acquisitionnum
        if(pep_scan == curr_scan) {
            if(peptide$mascot.score > pep_score_best) {
                pep_best <-
extract_mascot_match(peptide)
                pep_score_best <- peptide$mascot.score
            }
            next
        } else {
            pep_tbl <- bind_rows(pep_tbl, pep_best)
            pep_best <- extract_mascot_match(peptide)
            curr_scan <- peptide$acquisitionnum
            pep_score_best <- peptide$mascot.score
        }
    }
    pep_tbl
}
```

The files in the `MassIVE` repository for the `MSV000086195` study follow the `mzIdentML1.1.0.xsd` schema, so I'll validate the `.mzid` files using that schema.

```
# get the schema for the mzML files
mzID_schema <-
read_xml(file.path("schema","mzIdentML1.1.0.xsd"))
```

Now, instead of the X! Tandem results for the internal control that I generated, I will get all of the files in the repository `result` directory generated by Mascot for the study.

```
# get a list of all the mzIdentML files generated by
Mascot
result_files = list.files(
    file.path("large-data", "MSV000086195","result"),
    pattern=".mzid")
```

This chunk of code searches the `mzIdentML` files from the study for the peptides associated with the target protein mentioned above (COX2_HUMAN).

```
# create an empty tibble to accumulate desired matches
all_top_psm <- empty_psm()

n_iterations <- length(result_files)
pb <-
knitrProgressBar::progress_estimated(n_iterations)

# read and extract data from each mzIdentML file using
a loop
for(filename in result_files) {

    knitrProgressBar::update_progress(pb)

    full_path = file.path("large-data", "MSV000086195",
"result",
                          filename)

    if(!mzID_valid(full_path, mzID_schema)) {
        print(paste0(filename," file not valid
mzIdentML"))
        next
    }

    # The PSM function calls mzID with verbose=TRUE
    # to avoid excess output, call mzID directly
    all_psm <- mzID(full_path, verbose=FALSE) |>
        mzID::flatten() |>
        as("DataFrame") |>
        as("PSM")

    id <- as_tibble(all_psm)

    # If the object from mzID() is empty, skip to the
next file
    # otherwise, keep only the peptide matches that
contain at least one
    # TMT10plex modification
    if(nrow(id) < 1) {
        next
    } else {
        id <- id |>
```

```
            dplyr::filter(grepl("COX2_HUMAN",
accession, ignore.case = TRUE)) |>
            dplyr::filter(str_detect(modification,
'229.1629'))

    }

    # Any particular file may have no matches for the
protein of interest
    if(length(id$spectrumid) < 1) {
        next
    }

    # The top_scans() function returns the highest-
scoring unique MS2
    # spectra, which are added to the table
    kept_peptides <- top_mascot_scans(id)
    all_top_psm <- bind_rows(all_top_psm,
kept_peptides)
}
```

Now, sort the final table for use in finding the reporter intensities for the peptides from the selected protein.

```
# sort the table by batch, fraction, and scan number
prot_id <- arrange(all_top_psm, batch, fraction, scan)
```

Like with the internal control, use the scan numbers for the identified peptides for COX2_HUMAN to get the reporter intensities and create a table that captures the reporter intensity for each observed peptide in a column with the reporter name.

```
# Create an accumulator tibble
prot_tmt_inten <- get_empty_tmt()

# use a knitr compatible progress bar for a potentially
long-running loop
n_iterations <- length(prot_id$batch)
pb <-
knitrProgressBar::progress_estimated(n_iterations)

prev_raw_file <-""

for(prot_index in 1:length(prot_id$batch)) {

    knitrProgressBar::update_progress(pb)

    raw_file_name <- file.path("large-
data","MSV000086195","ccms_peak",

paste0(prot_id$base_filename[prot_index],
                                        ".mzML"))

    selected_prot <- mzR::openMSfile(raw_file_name,
backend = "pwiz")


    tmt_inten <- get_reporters(selected_prot, prot_id,
prot_index, "prot")
    prot_tmt_inten <- bind_rows(prot_tmt_inten,
tmt_inten)

    mzR::close(selected_prot)
    rm(tmt_inten)
    rm(selected_prot)
    gc()
}
```

Now process the results in the same way I did for the internal control. First, select Batch 1:

```
prot_tmt_inten <- dplyr::filter(prot_tmt_inten,
batch==1) |>
    dplyr::select(-c(batch, pep_seq))
```

Like before, there is no point in keeping peptides that are missing an intensity for a particular sample.

```
row_sub = apply(prot_tmt_inten[,5:14], 1, function(row)
all(row != 0 ))
prot_tmt_inten <- prot_tmt_inten[row_sub,]
```

And like before, I will perform the log base two transformation to bring all of the intensity values into the same range.

```
# transform intensity columns to log2()
prot_tmt_log2 <- prot_tmt_inten |>
    dplyr::mutate(across(5:14, log2))
```

Here, I fill in the `sample_type` to be just `prot`. I could use any string, especially if I were analyzing different proteins for differences between them.

```
tmt_norm <- tmt_norm |>
    dplyr::mutate(sample_type="prot")
```

For the internal control peptides, this step was performed in Chapter 3. The object is to join the peptide spectrum match table with the reporter intensity table by fraction and scan number (since I have already filtered the data to contain only Batch 1). The `left_join()` function keeps all of the rows of `prot_tmt_log2`. After that, I just cleaned up a little by moving the `sample_type` and `data_type` columns to be first and second in the table.

```
prot_norm <- prot_tmt_log2 |>
    dplyr::select(-c(contains('tmt'), sample_type,
data_type)) |>
    dplyr::left_join(tmt_norm, join_by(fraction,
scan_num)) |>
    dplyr::relocate(data_type) |>
    dplyr::relocate(sample_type)
```

```
prot_tmt <- prot_tmt_log2 |>
            tidyr::pivot_longer(!c(sample_type,
data_type, fraction, scan_num),
                                names_to = "reporter", values_to
= "intensity") |>
            dplyr::mutate(across(1:3, as_factor)) |>
            dplyr::mutate_at('reporter', as_factor)
```

The code below displays all the reporters for COX2_HUMAN as boxplots, one for each reporter (Figure 5.9). Here, I am using only the normalized data. The next steps are the same as before: test what visually looks like the difference between the reporters using the F-test and computing the p-value.

```
p_norm_cox2 <- prot_tmt |>
    ggplot() +
    geom_boxplot(aes(reporter, intensity)) +
    xlab("Reporter") +
    ylab("log2(intensity)") +
    theme_classic() +
    theme(axis.text.x = element_text(angle = 45, hjust
= 1)) +
    theme(plot.title = element_text(hjust = 0.5, vjust
= 2)) +
    ggtitle(label = "Normalized COX2 Reporter
Intensities - Batch 1")

print(p_norm_cox2)
```

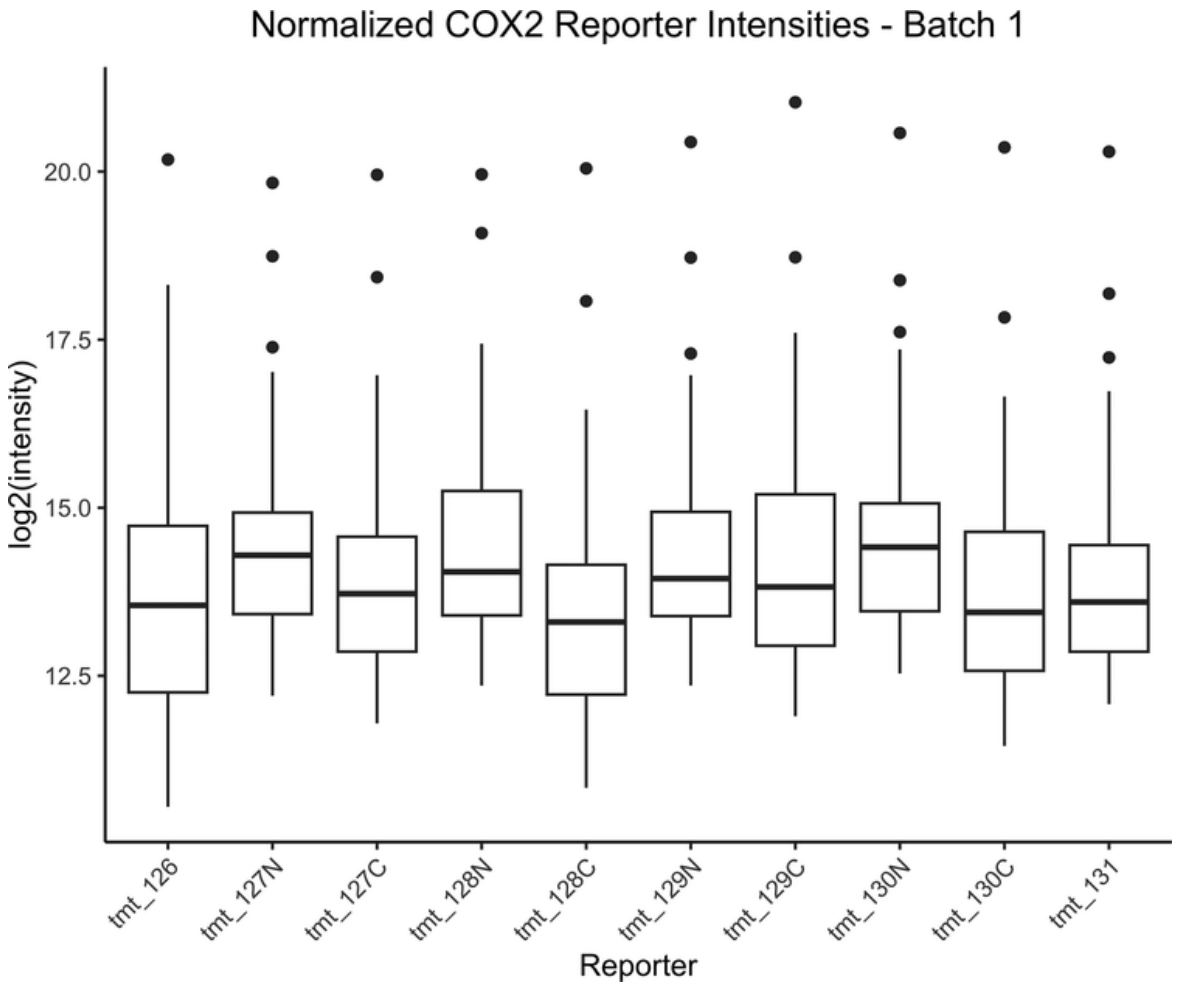


Figure 5.9 Normalized intensities for all the reporters of the human COX2 protein in Batch 1.

Computing the F-score for the normalized peptide intensities being independent of the reporter.

```
prot_observed_f_statistic <- prot_tmt |>
    specify(intensity ~ reporter) |>
    hypothesize(null = "independence") |>
    calculate(stat = "F")
```

Randomize the reporter value for each intensity and compute the F-score. By doing this 5000 times, I get a distribution that is pretty close to the theoretical F-distribution, but without assuming the F-distribution is the actual distribution that is present in the data.

```
prot_null_dist <- prot_tmt |>
    specify(intensity ~ reporter) |>
    hypothesize(null = "independence") |>
    generate(reps = 5000, type = "permute") |>
    calculate(stat = "F")
```

Visualize the observed F-score and the distribution of possible F-scores that can occur by random chance (Figure 5.10).

```
p_norm_prot_f_test <- prot_null_dist |>
    visualize() +
    shade_p_value(prot_observed_f_statistic, direction
= "greater") +
    theme_classic() +
    theme(plot.title = element_text(hjust = 0.5)) +
    xlab("F-Statistic") +
    ylab("Count") +
    theme(plot.title = element_text(hjust = 0.5)) +
    theme(plot.subtitle = element_text(hjust = 0.5)) +
    ggtitle(label = "Sampling-Based Null Distribution
(n=5000)",
            subtitle =
            paste0("Normalized COX2 Reporter
Intensities: F = ",
                   round(prot_observed_f_statistic,
digits=4))
            )

print(p_norm_prot_f_test)
```

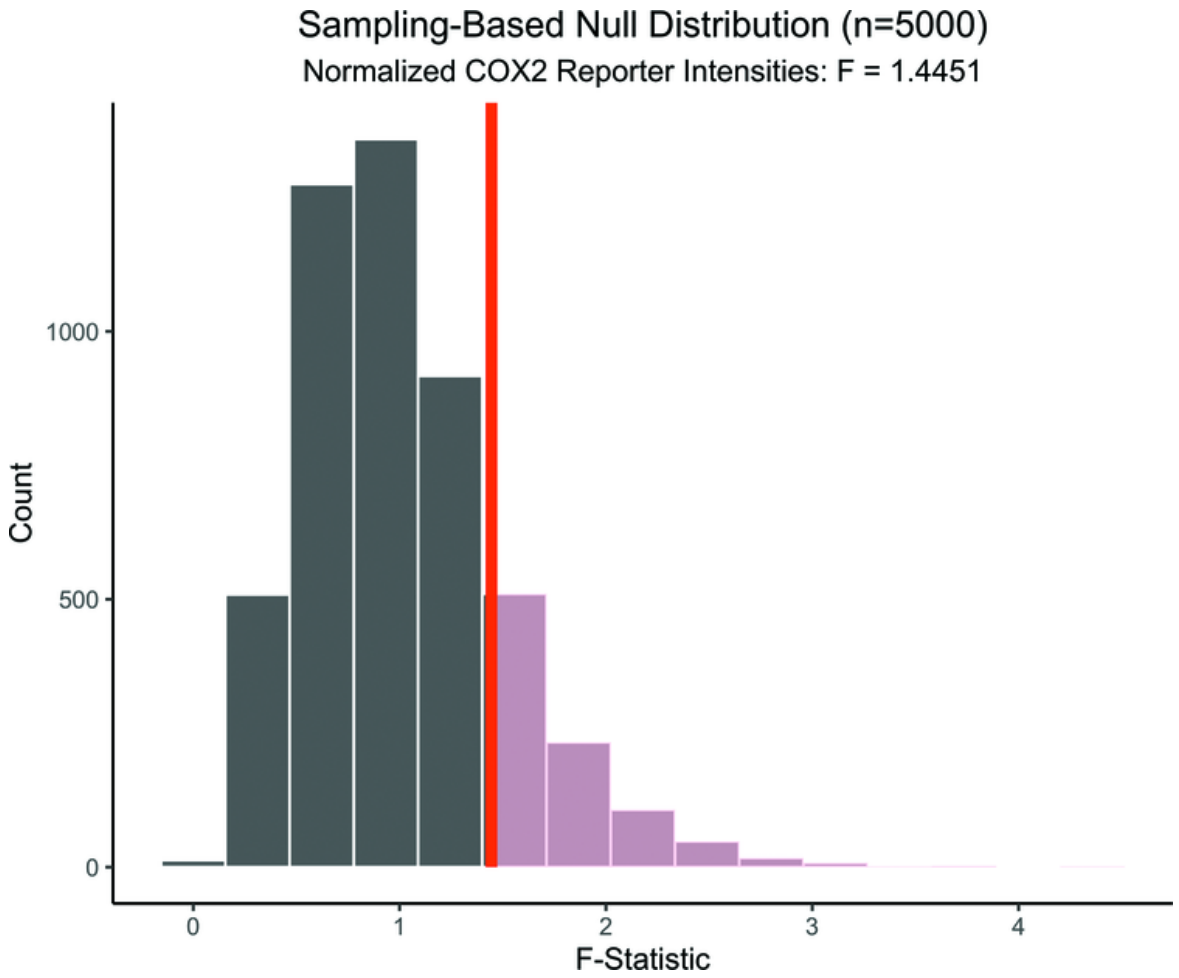


Figure 5.10 Overlay of the observed F-score with the distribution of F-scores for the Null Hypothesis that the peptide intensities and reporters are independent.

And finally, compute the p-value that the observed F-score occurred by chance.

```
p_value <- prot_null_dist |>
  get_p_value(obs_stat = prot_observed_f_statistic,
direction = "greater")

p_value
```

```
## # A tibble: 1 x 1
```

```
##   p_value
##     <dbl>
## 1   0.176
```

The post-normalization p-value for the COX2_HUMAN protein reported as differentially expressed is far below the normalized internal control, which means that the normalization process did not remove an excessive amount of biological variation. Because this hypothesis test is just to determine if *all* the samples came from the same distribution, the resulting p-value does not suggest that all of the samples are different, which is, in fact, not terribly likely based on the experimental design. So, by itself, this p-value does not establish that all of the means could not have been drawn from the same population, as there is still a ~16% chance of getting these differences by random. However, given the experimental design, I have good reason to believe that of the 10 samples represented, some were either from the same group (NA or MA) or included technical replicates and should skew the p-value a little high. A reasonable next step is to perform multiple t-tests between each pair of samples to see which ones might be different from each other. That could then be matched to the experimental design to draw conclusions about the differential expression of the COX2_HUMAN protein between the two groups of individuals.

The `rstatix` package has tidy statistical tests, including multiple t-tests, which compare each reporter against each other reporter. Since each reporter value for a given peptide can be paired (obtained from the same measurement), I'm using a *paired t-test* and dropping any pairs that fail to meet the 0.05 threshold.

```
library(rstatix)

pwc <- prot_tmt |>
    pairwise_t_test(intensity ~ reporter,
p.adjust.method = "none") |>
    dplyr::select(-c(n2,p.signif,p.adj,p.adj.signif))
|>
    dplyr::filter(p <= 0.05)
pwc
```

```
## # A tibble: 5 x 5
##   .y.       group1   group2      n1      p
##   <chr>     <chr>    <chr>    <int>  <dbl>
## 1 intensity tmt_127N tmt_128C    34 0.0281
## 2 intensity tmt_128N tmt_128C    34 0.0196
## 3 intensity tmt_128C tmt_129N    34 0.0448
## 4 intensity tmt_126  tmt_130N    34 0.0427
## 5 intensity tmt_128C tmt_130N    34 0.0112
```

The `ggpupr` package used earlier to combine multiple plots into a single plot also has a very nice way to visualize the multiple t-tests, showing the pairs of reporters whose means are potentially different after normalization.

```
p_norm_prot_t_test <- prot_tmt |>
    ggboxplot(x = "reporter", y = "intensity") +
        stat_pvalue_manual(pwc, hide.ns = TRUE, label =
"p",
                           y.position = 21, step.increase =
0.1,
                           ggtheme = theme_classic()) +
        xlab("TMT Reporter") +
        ylab("log2(Intensity)") +
        theme(axis.text.x = element_text(angle = 45,
hjust = 1, size = 10)) +
        theme(plot.title = element_text(hjust = 0.5)) +
        theme(plot.subtitle = element_text(hjust =
0.5)) +
        ggtitle(label = "Normalized COX2 Expression -
Batch 1",
            subtitle = "Significant p-values Between
Reporters"
        )

print(p_norm_prot_t_test)
```

The multiple t-tests (uncorrected for false discovery rate) show that five of the differences could be significant (Figure 5.11). It's important to note that this analysis could be improved by combining all three batches. Even though they contained different individuals, all three could be combined for normalization purposes. Further, it would increase the number of observations to something larger than n=34 which is a rather small sample size given the variance, even post-normalization. An increase in the number of observations of each peptide would improve the estimation of the means and lower the variance, thus increasing the power of the hypothesis test.

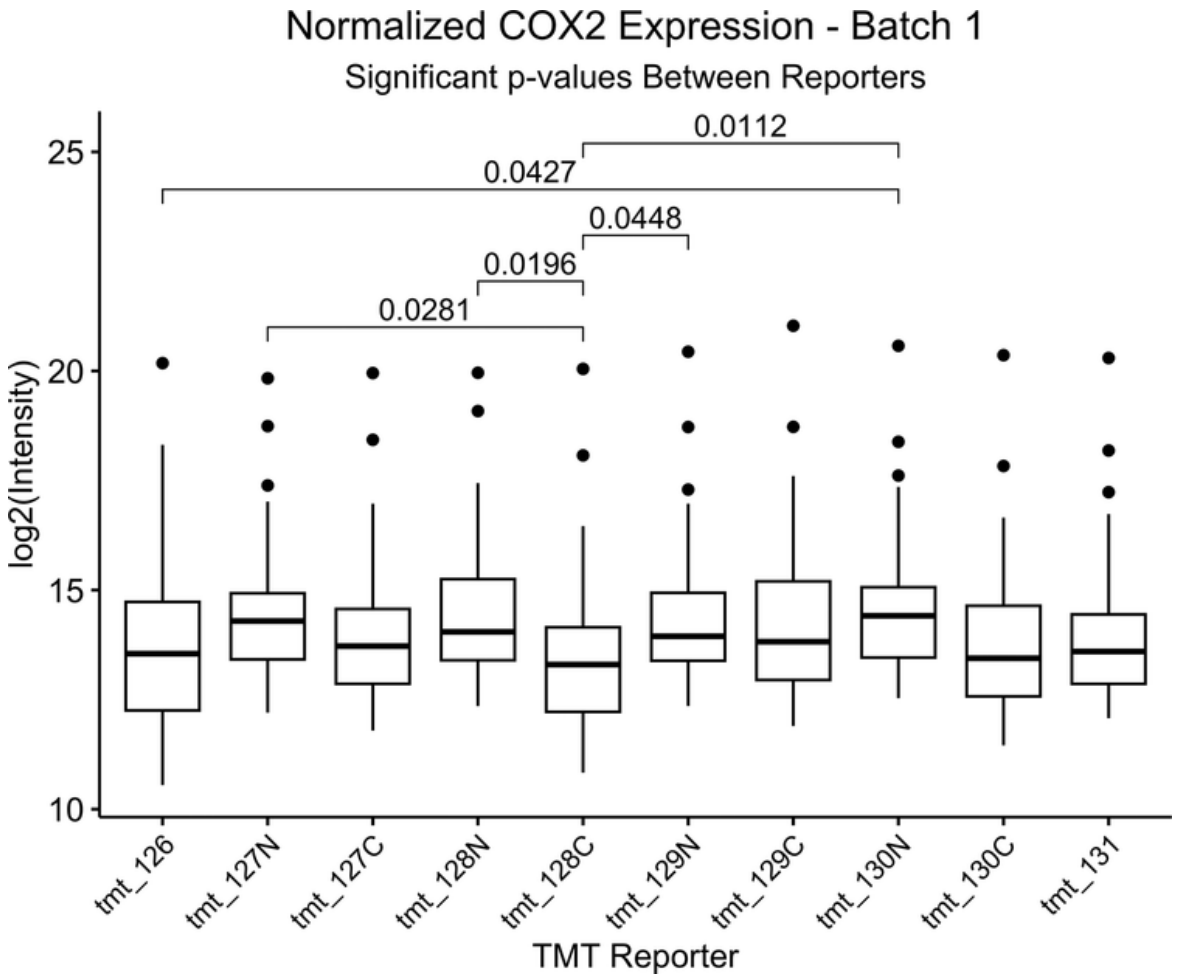


Figure 5.11 Boxplot of normalized reporter values for human COX2 reporter intensities showing the pairs of reporters with significant differences.

5.4 Summary

This chapter has covered a wide range of data analysis techniques associated with profile and centroided mass spectra, as well as calculating molecular weights from formulas and formulas from masses. I have used several different methods for reading the various kinds of files that mass spectra can be stored in. By combining packages from Bioconductor, CRAN, and GitHub, it is possible to assemble a data analysis pipeline that is custom fit for your needs. You

may find that a Bioconductor package will do everything you need, or you may decide that one does some of what you need, and you can take over the analysis using a combination of base R and the `tidyverse` and `tidymodels` packages. It's important to keep on the lookout for tools that were developed for other analytical techniques when analyzing mass spectrometry data. In this chapter, I used a normalization method originally developed for genomics studies and showed that it can be applied to mass spectrometry measurements.

In this chapter, I used stand-alone mass spectra or extracted spectra from high-performance liquid chromatography-mass spectrometry (HPLC-MS) and tandem mass spectrometry (MS/MS) measurements. In the next chapter, I will show how to analyze signals generated by mass spectrometers that are in the form of chromatographic data. Liquid chromatography-mass spectrometry (LC-MS) and LC-MS/MS are incredibly powerful techniques that are widely used for both qualitative and quantitative measurements.

Chapter 6
Analysis of Chromatographic Data from Mass Spectrometers

6.1 Introduction

This chapter will show how to analyze chromatography data generated using mass spectrometry. There are many similarities between chromatography data generated by different chemical analyzers. However, important aspects are introduced by the specificity, selectivity, and sensitivity of mass spectrometers. Chromatography systems produce data where the x-axis is time, and the characteristics of that data are described mainly in terms of chromatography *peaks*. In Chapter 4, I introduced the concept of chromatographic peaks generated from both full-scan mass spectra (MS) and selected reaction monitoring (SRM). While there are many types of chromatography, this chapter will focus mainly on liquid chromatography (LC) when paired with either single (MS) or multiple-stage mass spectrometry (MS/MS). Many approaches described in this chapter can be applied to gas chromatography (GC) and have often been applied to mass spectra traces, especially time-of-flight (TOF) m/z data, where the x-axis starts as time.

6.2 Chromatographic Peak Basics

Chromatographic peaks have characteristics determined by both the physics and chemistry of the separation process and by the mass spectrometer. Peaks are signals that represent compounds of interest when they can be detected above the various kinds of noise that are present in chromatography

and mass spectrometry data. While it is simple to state what a chromatography peak is, it is not so easy to give precise definitions of the words “detected” and “noise.” In other words [148], when does a collection of data points represent a deterministic, which carries chemical information, and when are data points random, representing imperfections in the instrumentation?

To analyze chromatography data using R, you must select precise signal and noise definitions that match what is observed in liquid chromatography-mass spectrometry (LC-MS). To start, I’ll explore some characteristics of chromatographic peaks using R, and from there, I’ll describe peak detection and various techniques for improving the amount of chemical information in chromatographic data.

6.2.1 Peak Shape Characteristics

As introduced in Chapter 4, I use exploratory data analysis techniques to describe the important concepts of chromatographic peaks. In this example, I’ll use real-world chromatograms and show how to extract the key characteristics of peaks. In this code, I’ll use the `readSRMData()` function from the `MSnbase` package and modify the data frame holding the feature data to add an index to the chromatograms.

```
file_name <- "sample_011.mzML"
srm_filename <- file.path("data", "chrom", file_name)
srm <- readSRMData(srm_filename)

# Get a data.frame from the Feature data using the
fData() accessor function
id_df <- fData(srm)
id_df$srm_index <- row(id_df)[,1]
```

```
  names(id_df)
```

```
##  [1] "chromatogramId"
"chromatogramIndex"
##  [3] "polarity"
"precursorIsolationWindowTargetMZ"
##  [5] "precursorIsolationWindowLowerOffset"
"precursorIsolationWindowUpperOffset"
##  [7] "precursorCollisionEnergy"
"productIsolationWindowTargetMZ"
##  [9] "productIsolationWindowLowerOffset"
"productIsolationWindowUpperOffset"
## [11] "srm_index"
```

These are the default names created by `readSRMData()`, and the product and precursor names are

```
  colnames(id_df)[4] <- "precursor"
  colnames(id_df)[8] <- "product"
  print(id_df[,c(1,11,4,8)])
```

```
##   chromatogramId srm_index precursor product
## 1       SRM Qual         1     468.2   396.2
## 2      SRM Quant         2     468.2   414.2
## 3   SRM Quant IS         3     472.3   400.3
## 4    SRM Qual IS         4     472.3   418.3
```

Based on the content of the feature data structure, this file contains four chromatograms, as described in Section 4.4.2. There is a quantifier trace, a qualifier trace from the quantifier precursor m/z, an isotopically labeled quantifier internal standard (IS) (+4 Da) trace, and a qualifier trace from the IS. The data from each trace is accessible by indexing the `srm` object using the row number in the table, which I've added as a column for convenience.

Now, I can extract the retention time (x-axis) and the intensity (y-axis) values using the accessor functions from

`MSnbase`. The raw time data is reported by `rtime()` in units of minutes, but it is more convenient to work in seconds, especially when working with fast acquisition rates.

```
# SRM Quant is the second element in the srm object
t_raw <- rtime(srm[2]) * 60
y_raw <- intensity(srm[2])
```

Next, I'll define the time range that represents valid acquisition. Sometimes, the very beginning of an acquisition has artifacts due to valve switching or other anomalies that should be ignored. It is also common for the mass spectrometer to enter acquisition mode before the liquid is diverted to the ion source. This results in no signal for a period, followed by an abrupt shift once ions are generated. If the liquid is diverted before the acquisition has ended, the signal will eventually drop to a lower level as the valve closes. When working with methods like these, it's usually necessary to avoid including acquisition artifacts in the analysis.

```
# define acquisition time range (s)
acquisition_start_t <- 0
acquisution_end_t <- max(t_raw)

t <- t_raw[t_raw > acquisition_start_t & t_raw <
acquisution_end_t]
y <- y_raw[t_raw > acquisition_start_t & t_raw <
acquisution_end_t]
```

Now, I want to know the sampling frequency of the data. How fast was the data acquired? This frequency will tell me the upper limit on the frequency of noise.

```
get_sampling_freq <- function(t) {
    delta_t <- array(dim=length(t)-1)

    for(i in 1:length(t)-1) {
        delta_t[i] <- t[i+1] - t[i]
    }

    median(1/delta_t)
}
```

I get the median sampling rate across the acquisition window using this function.

```
sampling_frequency <- get_sampling_freq(t)
print(sprintf("Sampling rate: %0.2f Hz",
sampling_frequency))
```

```
## [1] "Sampling rate: 16.67 Hz"
```

It is helpful to have a generic chromatogram plotting function that creates a base plot that can be modified with additional information about the peak. Here I give such a function called `chrom_plot()`. The plotting function requires a vector of x-axis data points and a vector of y-axis data points. It also requires a vector of y-axis values representing data to be drawn point-to-point. This is intended to hold either the raw or smoothed data. Data smoothing will be covered in detail in Sections 6.3.2 and 6.4. The function includes the ability to add optional titles, subtitles, and an option to plot points. This is an example of creating a `ggplot()` object and then conditionally adding new layers.

```r
chrom_plot <- function(x_vec, y_points, y_line,
                       main_title=NULL, sub_title=NULL,
                       breaks=10, points=FALSE) {

    p <- ggplot() +
        scale_x_continuous(n.breaks = breaks) +
        scale_y_continuous(labels = inten_label) +
        geom_line(aes(x=x_vec, y=y_line),
linewidth=0.75, color=pal$gray) +
        xlab("Retention Time (sec)") +
        ylab("Intensity (counts/sec)") +
        theme_classic() +
        theme(plot.title = element_text(hjust = 0.5,
size=16)) +
        theme(plot.subtitle = element_text(hjust =
0.5)) +
        theme(
            axis.text=element_text(size=11),
            axis.title=element_text(size = 14),
            legend.text = element_text(size = 11),
            legend.title = element_text(size = 14)
        )

    if(!is.null(main_title)) {
        p <- p + ggtitle(label = main_title)
    }

    if(!is.null(sub_title)) {
        p <- p + ggtitle(label= main_title, subtitle =
sub_title)
    }

    if (points==TRUE) {
        p <- p + geom_point(aes(x=x_vec, y=y_points),
shape=1 )
    }
    p
}
```

A quick plot of the quantifier trace from this sample will give some idea about the structure of the data. For now, I'll just pass the raw y-values as the filtered data to connect the data points with a line.

```
p <- chrom_plot(t, y_points = y, y_line = y,
                main_title = "Sample 11 Quantifier",
                sub_title = sprintf("Sampling rate:
%0.1f Hz",

sampling_frequency),
                points = TRUE)
print(p)
```

In Figure 6.1, there could be two peaks of interest: the large peak centered around 8 seconds and a smaller peak near 12 seconds. The spike close to the beginning of the trace is probably not part of the chemical information. For now, I'll assume that chemical information about analytes in this method starts after 2 seconds to avoid artifacts. It may be necessary to revisit this assumption later using the tools discussed in Section 6.4.

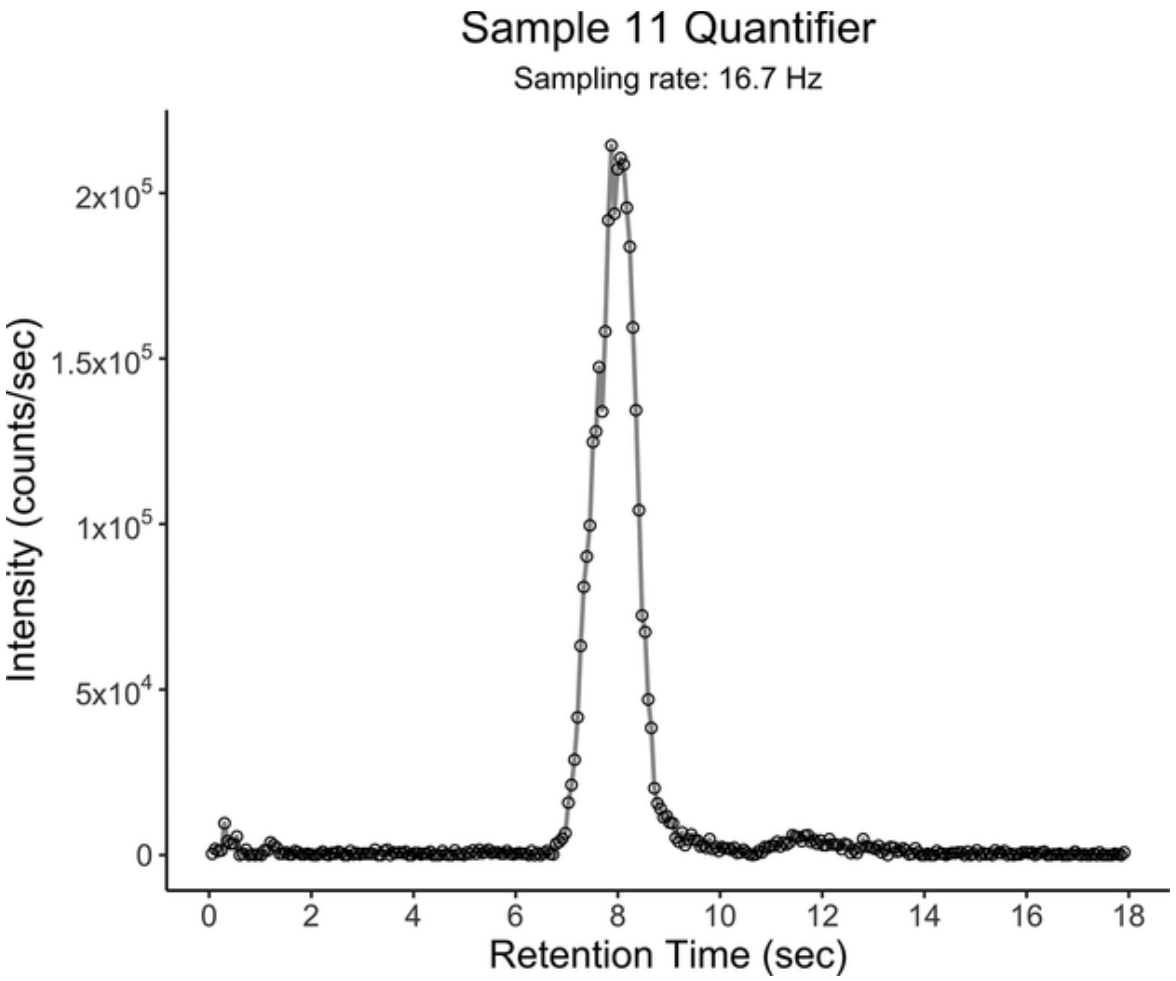


Figure 6.1 Plot of the chromatogram for quantifier ion in Sample 11.

```
acquisition_start_t <- 2

# Truncate data to include only the acquisition window
t <- t_raw[t_raw > acquisition_start_t & t_raw <
acquisution_end_t]
y <- y_raw[t_raw > acquisition_start_t & t_raw <
acquisution_end_t]
```

First, I'll focus on the large peak at 8 seconds. The important features of any peak are its apex intensity and time, the peak width at 50% of the maximum height (full width at half maximum [FWHM]), peak symmetry, and where the peak begins and ends. The peak symmetry can indicate

chromatographic performance, so it is often measured by looking at the distance to the apex on the left and right sides of the peak at both the 50% level and the 10% level. Finally, since mass spectrometers are concentration detectors, the total area of the peak is related to the concentration of the chemical being detected. Finding the area of a peak necessitates ascribing a peak start time and a peak end time, as well as any baseline offset created by the instrument.

6.2.2 Peak Location

First, I want to find the location of the peak. Visually, its apex is approximately 8 seconds. I want both the t and y-values, so I'll search for the apex in this compound's expected retention time range.

```
## expected RT range (in t units)
expected_rt <- 8
rt_tolerance <- 2
```

The `rt_tolerance` of 2 seconds is only needed in cases where the global maximum is not from the peak of interest. The `expected_rt` and `rt_tolerance` predefine a region of interest. In other peak-picking situations, the region of interest may not be known in advance. There are functions in `xcms` to locate regions of interest that I'll show later. We'll work through the example where the region of interest is known. The function `get_peak_apex()` takes the time and intensity vectors and returns a list of possible index values for local maxima in the region of interest.

```
get_peak_apex <- function(t, y, expected_rt,
rt_tolerance) {
    range_start <- expected_rt - rt_tolerance
    range_end <- expected_rt + rt_tolerance

    apex <- max(y[which(t > range_start & t <
range_end)])

    apex_index_list <- which(near(apex, y))
    apex_index_list[which.min(abs(t[apex_index_list]-
expected_rt))]
}
```

Calling `get_peak_apex()` will return the *apex index,* a single index that represents the intensity (y-axis) array index of maxima closest to the expected retention time.

```
apex_index <- get_peak_apex(t, y, expected_rt,
rt_tolerance)

print(sprintf("Index %d: RT=%.2fs Intensity=%.0f",
              apex_index, t[apex_index],
y[apex_index]))
```

```
## [1] "Index 98: RT=7.87s Intensity=214400"
```

Before moving on to the other characteristics of a peak, it is important to address the issue of characterizing the *baseline* on which the chemical data is superimposed. The apex result above is a good example of a peak parameter that could be incorrect due to an elevated baseline. The peak apex intensity might not be properly measured from zero. Some instruments generate a curved, sloping, or simply elevated baseline that will distort many critical aspects of a peak. This step in data preparation has become a field of study of its own. Using packages and directly coding solutions in R to approximate a baseline will be shown in the next section.

6.2.3 Baseline Correction

Height and area calculations are performed relative to a *baseline*, which usually represents the noise floor when no compound is present. There are many ways to estimate baselines, and depending on your data, some methods may work better than others. The R package `baseline` [149, 150] is a good starting point for testing a variety of baseline algorithms. The `baseline` package was designed to handle multiple traces in a matrix, so before it can be used, the time and intensity data have to be put into a matrix where each column is a time point in the trace. For the examples in this chapter, I'll use `baseline.als` method, which is an asymmetric least squares baseline described by Eilers [151–153]. The basic idea is to use weighted least squares to fit the data, initially assigning a weight of 1 to each data point in the trace. This ensures that the fit maximizes fidelity to the data. The algorithm then iteratively assigns a lower weight (parameter `p`) to data points above the fit and sets the weights below the fit as `1-p`. This way, the resulting fit values come closer to the least squares solution for the data below the fit. The chemically informative data will eventually (usually in 5–10 iterations) be driven to low weight and ignored by the fit. The data normally associated with the baseline will have a weight close to 1 and control the overall fit. The approach described by Eilers is a *Whittaker smoother* [154], which adds a penalty to the least squares cost function based on the value of the derivative of the data. The derivative penalty is scaled (the parameter `lambda`) to ensure that the resulting fit is as smooth as desired, given the *roughness* of the data. Different derivatives, or in discrete terms, *differences*, can be used for the roughness penalty. Eilers and Boelens suggested using the second derivative of the raw data, while others have suggested using the first derivative [155]. The higher the derivative, the larger the smoothing term. The `baseline` package uses the second derivative to constrain smoothness, so the values for `lambda`

will be different for other Whittaker smoothers that use other difference orders.

```
sp <- Matrix::t(y)
colnames(sp) <- t
new_sp <- baseline(sp, lambda = 10, p = 0.005,
method='als')
base_line <- as.vector(new_sp@baseline)
```

To see what the different baseline algorithms do, I'll plot them and zoom in on how they follow the noise and how they behave in the regions around peaks.

```
p_baseline <- p +
    ggtitle(label="Asymmetric Least Squares Baseline",
            subtitle = "lambda=10, p=0.005") +
    coord_cartesian(ylim=c(-1e3, 2.5e3)) +
    geom_vline(xintercept=t[apex_index],
col=pal$orange) +
    geom_line(aes(x=t, y=base_line), linetype="dashed",
color=pal$black)
print(p_baseline)
```

Figure 6.2 shows that the computed baseline looks reasonable for this data. The weighting penalty `p=0.005` forces the baseline to a reasonable level in the non-signal range, and the `lambda` value of `10` makes the baseline very smooth. In some situations, `als` might not give a reasonable baseline for specific acquisition methods or compounds. If different baseline characteristics are needed, the `baseline` package includes several popular approaches that can be specified with the `method=` parameter. See the `baseline` package documentation for a current list. Several variations of the asymmetric least squares approach are available from different packages, such as `airPLS` [155] and `arPLS` [156]. The *penalized least squares* method appears promising.

Implementing algorithms like `airPLS` or `arPLS` may require translating code examples from the literature to R.

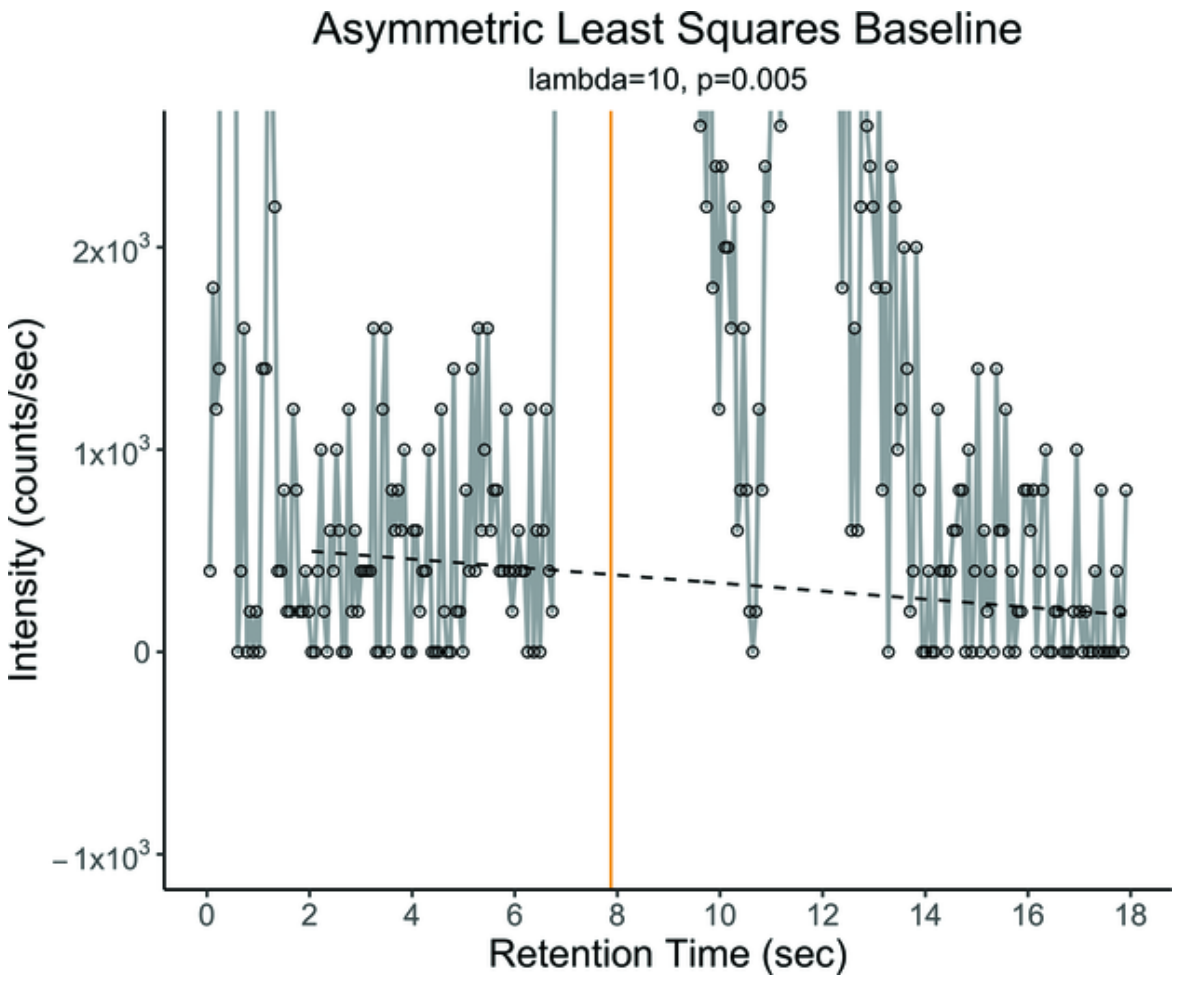


Figure 6.2 Asymmetric least squares baseline with lambda = 10 and p = 0.005.

Having settled on `als` for now, I can correct the intensity values of the trace by subtracting the `base_line` vector and recomputing the amplitude of the peak. Further processing of the trace should be done with this *baseline corrected trace* shown in Figure 6.3.

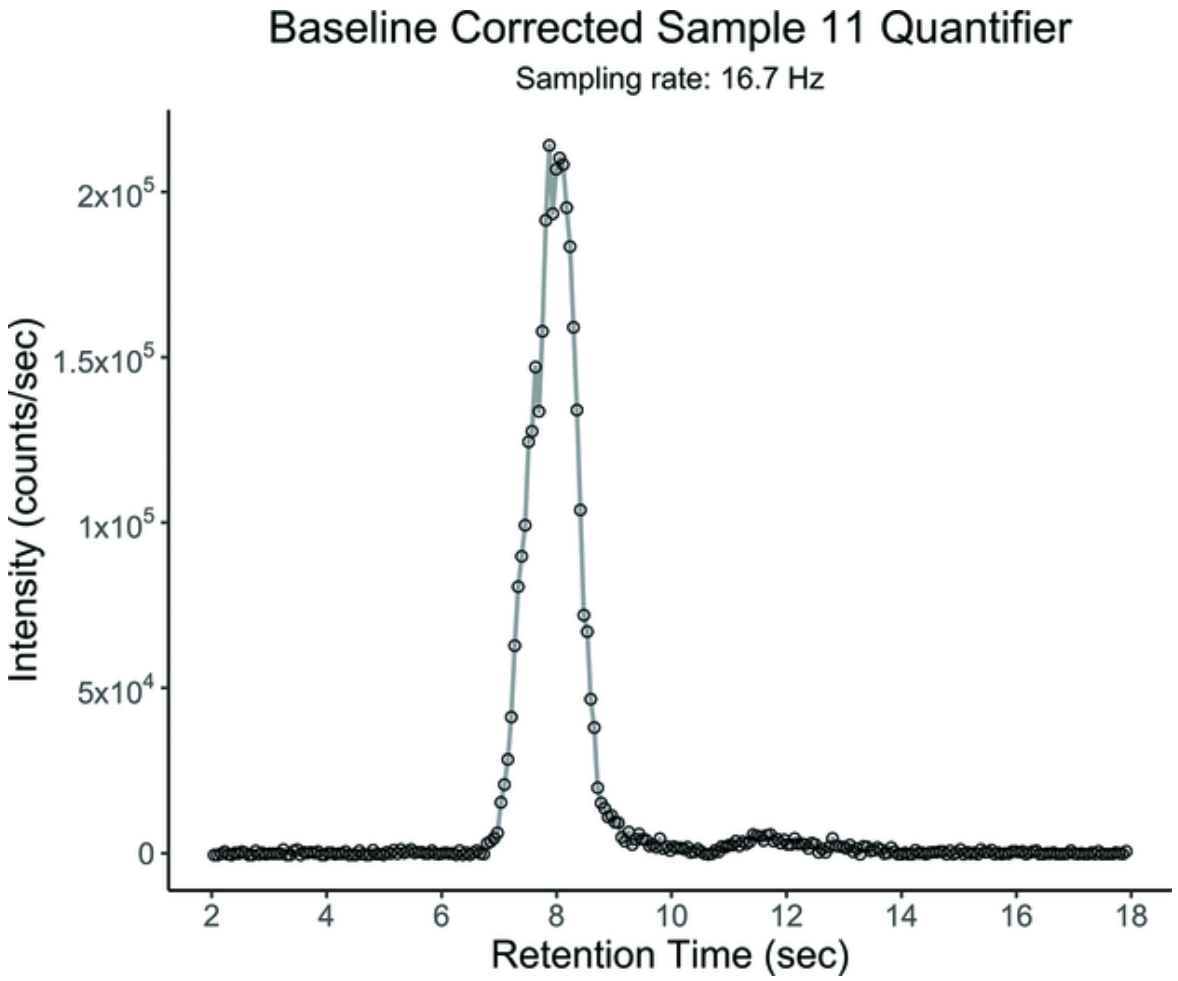


Figure 6.3 Baseline corrected signal using ALS algorithm.

```
  y_base <- y - base_line
  print(sprintf("Baseline at apex: %0.1f New maximum:
  %0.1f",
                base_line[apex_index], y[apex_index]))
```

```
## [1] "Baseline at apex: 382.5 New maximum: 214400.0"
```

```
p_base <- chrom_plot(t, y_points = y_base, y_line =
y_base,
                     main_title = "Baseline Corrected Sample
11 Quantifier",
                     sub_title = sprintf("Sampling rate:
%0.1f Hz",

sampling_frequency),
                     points = TRUE)
print(p_base)
```

6.2.4 Computing Key Peak Features

The next feature to be evaluated is the peak width and symmetry. The primary way of describing peak width is to estimate its width at half (50%) of its maximum height. This width is called FWHM or sometimes full width half height (FWHH). It is also common to compare the distance from the apex time to the front edge of the peak and the back edge of the peak at different heights. A symmetric peak would give the same value for both the front half and the back half-width, however, real chromatography peaks are rarely symmetric. Excessive asymmetry, however, can indicate problems with the chromatography process. These values are often tracked over time as part of a quality control process.

```
half_max <- max(y_base[apex_index])/2
```

To compute the width of the peak, the intercept between the peak edges and the `half_max` value needs to be found. The problem can be seen by looking closely at the intercepts between the half-maximum value and the peak.

```
p_intercept <- p_base +
    coord_cartesian(xlim=c(7.2, 8.5), ylim=c(107200-
24000, 107200+26000)) +
    geom_vline(xintercept=t[apex_index],
col=pal$orange) +
    geom_hline(yintercept=half_max, col=pal$blue) +
    ggtitle(label = "Front and Back Intercepts for
FWHM")

print(p_intercept)
```

In Figure 6.4, the half-maximum line intercepts the peak between two data points on both the front and back of the peak. Using the data point above or below the intercept would either under or overestimate the overall FWHM value and distort the estimation of the asymmetry of the peak. The retention times of the front and rear edges of the peak need to be computed to estimate the peak width. The edge times are computed from a linear interpolation between the two closest raw data points, and the intercept time is returned.

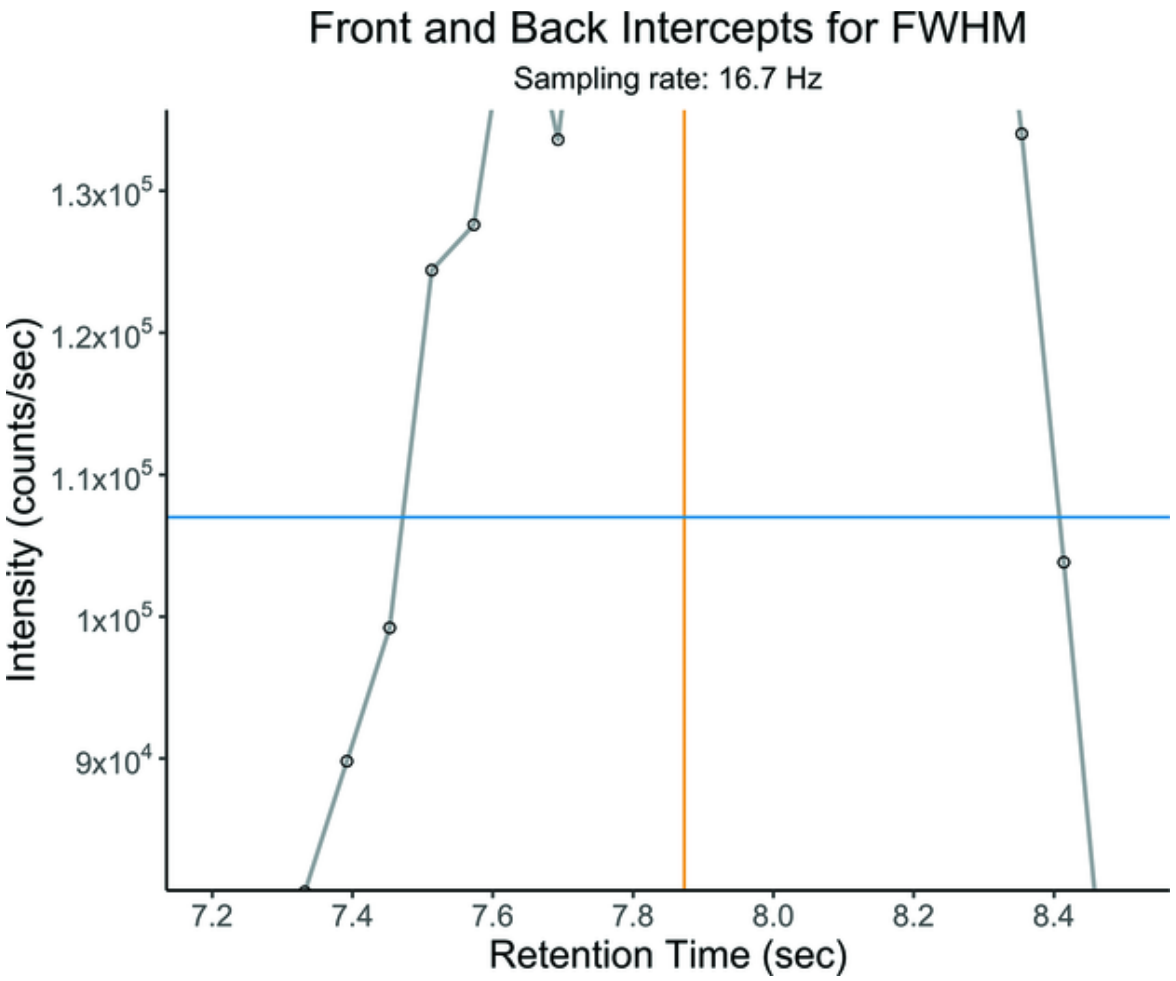


Figure 6.4 Intercepts at the front and back edges of the peak.

The interpolation between any two points and the intercept calculation can be performed using the `interp_time()` function.

```
interp_time <- function(x, y, y_in) {
    m <- (y[2] - y[1]) / (x[2] - x[1])  # slope (m) formula
    b <- y[1]-(m*x[1])                  # solve for intercept (b)

    (y_in - b) / m                      # return x: x = (y-b)/m
}
```

The front time value is computed by `get_peak_front_time()`, which starts at the apex index and works backward toward the front of the trace until the `y` value crosses the `y_intercept` value. Any value in the range of `y` values in the data can be used for `y_intercept`, which will be used to compute the width at both 50% and 10%.

```
get_peak_front_time <- function(t, y, apex_index,
y_intercept) {
    front_t_range <- NULL
    front_y_range <- NULL

    for(i in apex_index:2) {
        if(y[i] < y_intercept) {
            front_t_range <- c(t[i], t[i+1])
            front_y_range <- c(y[i], y[i+1])
            break
        }
    }
    interp_time(front_t_range, front_y_range,
y_intercept)
}
```

The back intercept is found with `get_peak_back_time()`, which starts at the apex and searches forward toward the end of the trace until it crosses the `y_intercept` and then uses the `interp_time()` function to find the time value for the intercept just like the function for the front intercept.

```
get_peak_back_time <- function(t, y, apex_index,
y_intercept) {
    back_t_range <- NULL
    back_y_range <- NULL

    for(i in apex_index:length(y)) {
        if(y[i] < y_intercept) {
            back_t_range <- c(t[i], t[i-1])
            back_y_range <- c(y[i], y[i-1])
            break
        }
    }
    interp_time(back_t_range, back_y_range,
y_intercept)
}
```

These functions can now be used to find the widths at both 50% and 10%.

```
front_50 <- get_peak_front_time(t, y_base, apex_index,
half_max)
back_50 <- get_peak_back_time(t, y_base, apex_index,
half_max)

print(sprintf("Front width: %.2fs Back width: %.2fs
(diff: %.2fs)",
              t[apex_index]-front_50, back_50-
t[apex_index],
              abs((back_50-t[apex_index])-
(t[apex_index]-front_50))))
```

```
## [1] "Front width: 0.40s Back width: 0.53s (diff:
0.13s)"
```

```
print(sprintf("FWHM: %.2fs", back_50-front_50))
```

```
## [1] "FWHM: 0.94s"
```

The same calculation can be performed at the 10% height value.

```
  width_10 <- max(y[apex_index]) * 0.1

  front_10 <- get_peak_front_time(t, y_base, apex_index,
  width_10)
  back_10 <- get_peak_back_time(t, y_base, apex_index,
  width_10)

  print(sprintf("Front width: %.2fs Back width: %.2fs
  (diff: %.2fs)",
                t[apex_index]-front_10, back_10-
  t[apex_index],
                abs((back_10-t[apex_index])-
  (t[apex_index]-front_10))))
```

```
## [1] "Front width: 0.78s Back width: 0.84s (diff:
0.06s)"
```

```
  print(sprintf("FW10: %.2fs", back_10-front_10))
```

```
## [1] "FW10: 1.61s"
```

These two calculations show that this particular peak is asymmetric, with more width at the front of the peak than at the back. This is commonly called *fronting*, whereas the opposite asymmetry is usually called *tailing*. This is interesting for this analysis because fronting is not considered common in high-performance liquid chromatography (HPLC) separations, while it is quite common in gas chromatography (GC) separations.

The peak at the 10% height from the baseline is about double the width at the 50% level, and this matches a common rule of thumb in terms of baseline peak width. Several additional characteristics of this peak that need to be computed include the location of the start and end of the peak, its area, and

how reliable the chemical information can be derived from the area calculation. The last question is often incorrectly framed in terms of the signal-to-noise ratio. As I'll show, there are several ways to think about what constitutes a signal and what constitutes noise, and it is easy to confound the probability of signal detection with point estimates of precision.

6.3 Fundamentals of Peak Detection

Ultimately, the peak can only be said to *start* when the measured intensity is statistically significantly above values obtained from the random fluctuations measured before the deterministic component of the trace begins. Further, peaks of all kinds have the characteristic width computed by the FWHM shown above. Likewise, peaks *end* when the signal falls back into the range expected to occur at random.

This is conceptually simple when dealing with a single source of deterministic signal at a time. In other words, when deterministic components of the trace (most generally molecules) do not overlap. When sources of chemical information overlap, it is still true that one deterministic process ends when it stops generating signals above random fluctuations, even if another source of chemical information is the dominant generator of the measured signal. Despite the selectivity of mass spectrometry, this kind of overlap is common. Further, even if separated at the baseline level, additional chemical signals constitute a second type of noise, which must be handled separately from the noise attributed to imperfections in the instrument. The *chemical noise* and the related problem of overlapping peaks will be addressed later in this chapter.

First, I'll show how to detect a single peak that is isolated from other peaks in a trace. There are many different ways to perform peak detection. Conceptually, the use of derivatives

is the simplest, and the approach can be extremely effective. In chromatography and mass spectrometry applications, a second, more complex method based on *continuous wavelet transforms* (CWT) is also very popular, especially within the Bioconductor community.

6.3.1 Derivative-based Peak Detection

Using derivatives to perform peak detection uses the basic relationships between the numerical first and second derivatives of a peak. The relationship between the derivatives for the peak in Figure 6.3 can be seen in Figure 6.5 using functions from the `gsignal` package.

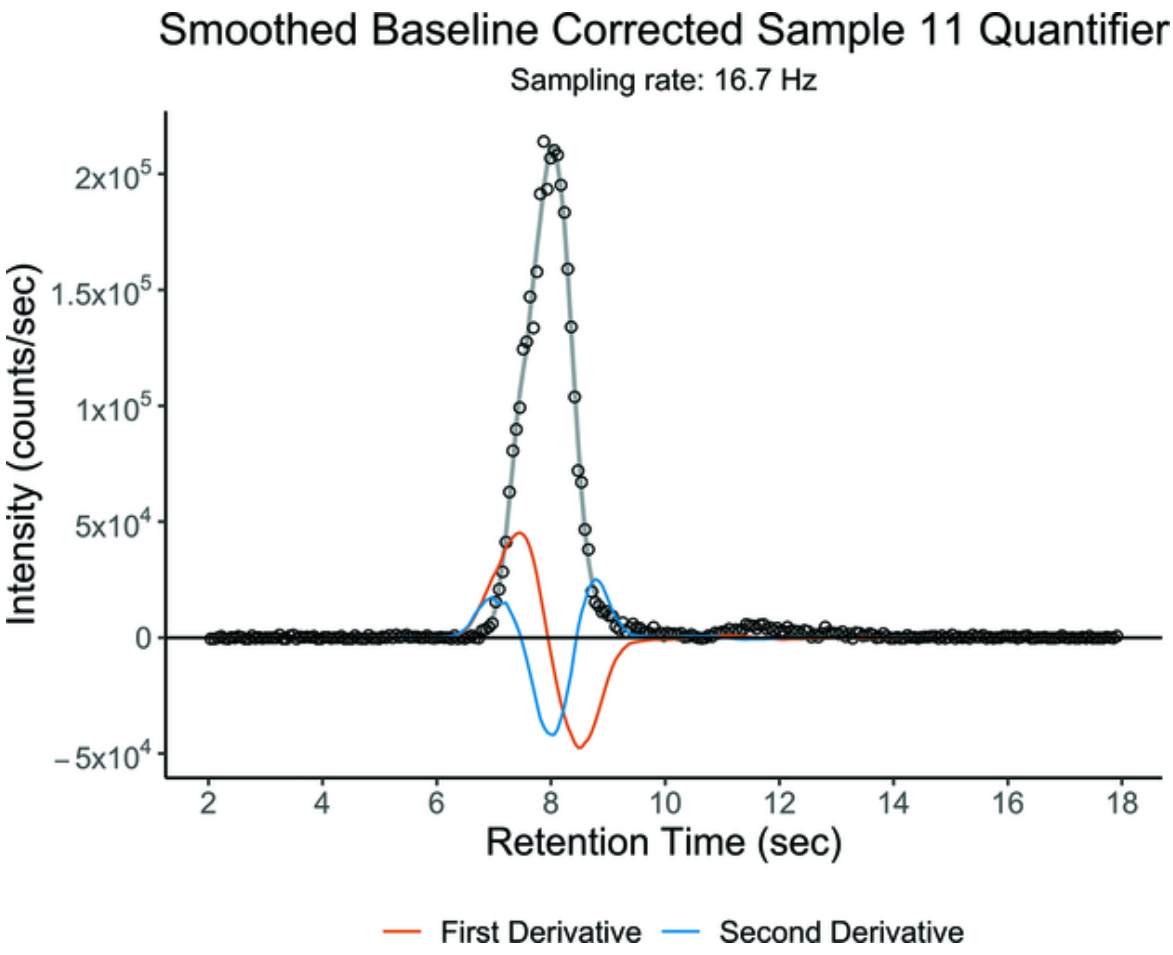


Figure 6.5 First and second derivative overlay on a real chromatographic peak.

I'll use the Savitzky–Golay (SG) smoothing and differentiating filters [157] described by Felinger [158] to get around the difficulties in numerical differentiation in the presence of noise. As Lanczos points out, taking the difference between data points from a signal containing noise greatly amplifies that noise, making the numerical differences useless [159]. In Section 6.3.2, I will discuss that the objective of smoothing is to improve the accuracy of the observed signal, which, in turn, allows useful numeric differentiation. There are two parameters that must be selected when using SG filters: the length of the filter (measured in the number of raw data points) and the order of the polynomial that will be fit across the length. In this chapter, I will follow the advice of Edwards

and Willson [160], which uses the number of data points above the FWHM to choose a filter length.

The data point count approach was later taken up by Enke [161], who showed optimum scaling factors of the data point count above the FWHM for selecting the length for SG filters. In Section 6.4, I will show that the FWHM data point count and the sampling frequency relate to the frequency domain content [162] of the deterministic component of the data. This is useful when *smoothing in the large* using Fourier analysis as described by Lanczos [163]. In Section 6.4.3, I'll show how to design optimum filters using frequency analysis.

First, I'll define a function to compute the filter length from the number of data points above the FWHM and use Willson's `0.7` factor to compute the filter width. Enke's weight is closer to `0.55`, but the number should range between `0.5` and `1.0`. You can evaluate your data and select a factor based on the expected peak shape.

```
get_sg_filter_length <- function(t, front_time,
back_time) {
    points_above_fwhm <- length(which(t > front_time &
t < back_time))
    if(points_above_fwhm < 7) {
        # The apex is a single data point: use the
smallest window possible
        filter_length <- 7

    } else {
        # smooth range 0.7 * data points above FWHM
        # Edwards, T.; Willson, P. Digital Least
Squares Smoothing of Spectra.
        # Appl Spectrosc 1974, 28 (6), 541–545.

        filter_length <- round(0.7 * points_above_fwhm)
        # if the filter length is even, then make it
one longer to make it odd
        if(filter_length %% 2 != 1) {
            filter_length <- filter_length + 1
        }
    }
    filter_length
}
```

For our example peak, the filter width is computed from the raw data using `get_sg_length()`:

```
sg_length <- get_sg_filter_length(t, front_50, back_50)
print(sg_length)
```

```
## [1] 11
```

Now, the characteristics of the peak can be recomputed from the smoothed data.

```
y_smooth <- sgolayfilt(y_base, n=sg_length, p=2)

apex_index_smooth <- get_peak_apex(t, y_smooth,
expected_rt, rt_tolerance)
half_max_smooth <- y_smooth[apex_index_smooth] / 2

front_smooth <- get_peak_front_time(t, y_smooth,
                                    apex_index_smooth,
half_max_smooth)
back_smooth <- get_peak_back_time(t, y_smooth,
                                  apex_index_smooth,
half_max_smooth)
```

The `sgolayfilt()` function can be used to compute the first and second derivatives of the raw data. This function takes the raw y-values, the filter length, and the polynomial order (in this example, I use a quadratic polynomial `p=2`) and a scaling factor (`ts`). The scaling factor is primarily used to bring the derivatives into the scale of the raw data for plotting, since the differences between raw data points can be on a very different scale than the absolute value of the data points themselves. As mentioned above, the increase in noise in the numerical derivatives of LC-MS data warrants using a longer filter to reduce the noise in the derivative trace. Empirically, I have found that doubling the length of the derivative filters improves the calculation of the numerical derivatives. You can adjust this factor based on the noise in your data, but as with all convolutional filters, ensure that the filter length is odd.

```
derivitive_length <- round(sg_length * 2)

if(derivitive_length %% 2 != 1) {
    derivitive_length <- derivitive_length + 1
}
print(derivitive_length)
```

```
## [1] 23
```

The smoothing filter length was computed to be 11, which means the derivative length was estimated as the next odd number that was double that length. The level of noise in your data will help you determine how to define the length of the derivative filters. Noise in real-world peaks is amplified when the derivative is taken, so the scale of the random noise will affect how well you can define the start and end of a peak. Choosing a longer filter length for the derivatives can compensate for noise without distorting the peak since these values are only used to determine the beginning and end of the peak. As described above, the `ts` parameter used in `sgolayfilt()` is a scaling factor. The `ts` is used in the denominator of the scaling calculation so a lower value of `ts` gives a larger value for the derivative. The value of this parameter is often either left at the default (`ts=1`) or determined empirically.

```
y_d1 <- sgolayfilt(y_base, n=derivitive_length, p=2,
m=1, ts=0.25)
y_d2 <- sgolayfilt(y_base, n=derivitive_length, p=2,
m=2, ts=0.25)
```

Plotting the derivatives over the raw data shows how the derivatives change at the start and end of the peak.

```
p_smooth <- chrom_plot(t, y_points = y_base, y_line =
y_smooth,
                main_title = "Smoothed Baseline
Corrected Sample 11 Quantifier",
                sub_title = sprintf("Sampling rate:
%0.1f Hz",

sampling_frequency),
                points = TRUE)

p_d1_d2 <- p_smooth +
    geom_line(aes(x=t, y=y_d1, color="First
Derivative")) +
    geom_line(aes(x=t, y=y_d2, color="Second
Derivative")) +
    geom_hline(yintercept=0.0, color=pal$black,
linewidth=0.5) +
    scale_color_manual(
                name='',
                breaks=c('First Derivative', 'Second
Derivative'),
                values=c(pal$darkorange, pal$blue)) +
    theme( legend.position = "bottom")

print(p_d1_d2)
```

The basic idea is that at the apex of the peak, the first derivative will be zero by definition. It is easy to see in Figure 6.5 that the zero crossing of the first derivative lines up with the peak maximum time, remembering that the absolute maximum intensity of unsmoothed data might be a noise spike. In Figure 6.5, the first derivative crosses zero at the retention time of the apex of the peak, as expected. The first derivative is positive on the front half of the peak and negative on the back half. The second derivative increases and is positive during the initial rise of the peak and then goes negative during the apex region of the peak. The second derivative crosses zero roughly at the time of the front and

back of the peak at the FWHM. After the apex region of the peak, the second derivative again turns positive.

The basic rule for detecting a real peak, rather than noise, can be created from the derivatives. The peak starts when both the first and second derivatives are positive and ends when the first derivative crosses from negative to positive when the second derivative is positive.

However, looking closely at the baseline area, it becomes clear that a simple positive/negative rule won't work in the presence of noise.

```
p_zoom <- p_d1_d2 +
    coord_cartesian(xlim=c(6, 12),
                    ylim=c(-1e3, 2.5e3)) +
    geom_vline(xintercept=front_smooth, col=pal$black,
linetype="dashed") +
    geom_vline(xintercept=back_smooth, col=pal$black,
linetype="dashed" ) +
    ggtitle(label="First and Second Derivatives")

print(p_zoom)
```

In Figure 6.6, you can see that both the first and second derivatives vary around zero until very close to the first data points, which could reasonably be called the start of the peak. The situation at the back of the peak is different. As is typical of chromatography peaks, higher-order processes become evident, and the noise is higher in the tail of the peak than in the front. The first and second derivatives bounce between positive and negative in a way similar to the front of the peak. Defining the beginning and end of a peak is significantly affected by the noise. Hard boundaries like "positive" or "negative" ignore the noise, allowing random variability to define important aspects of the peak definition.

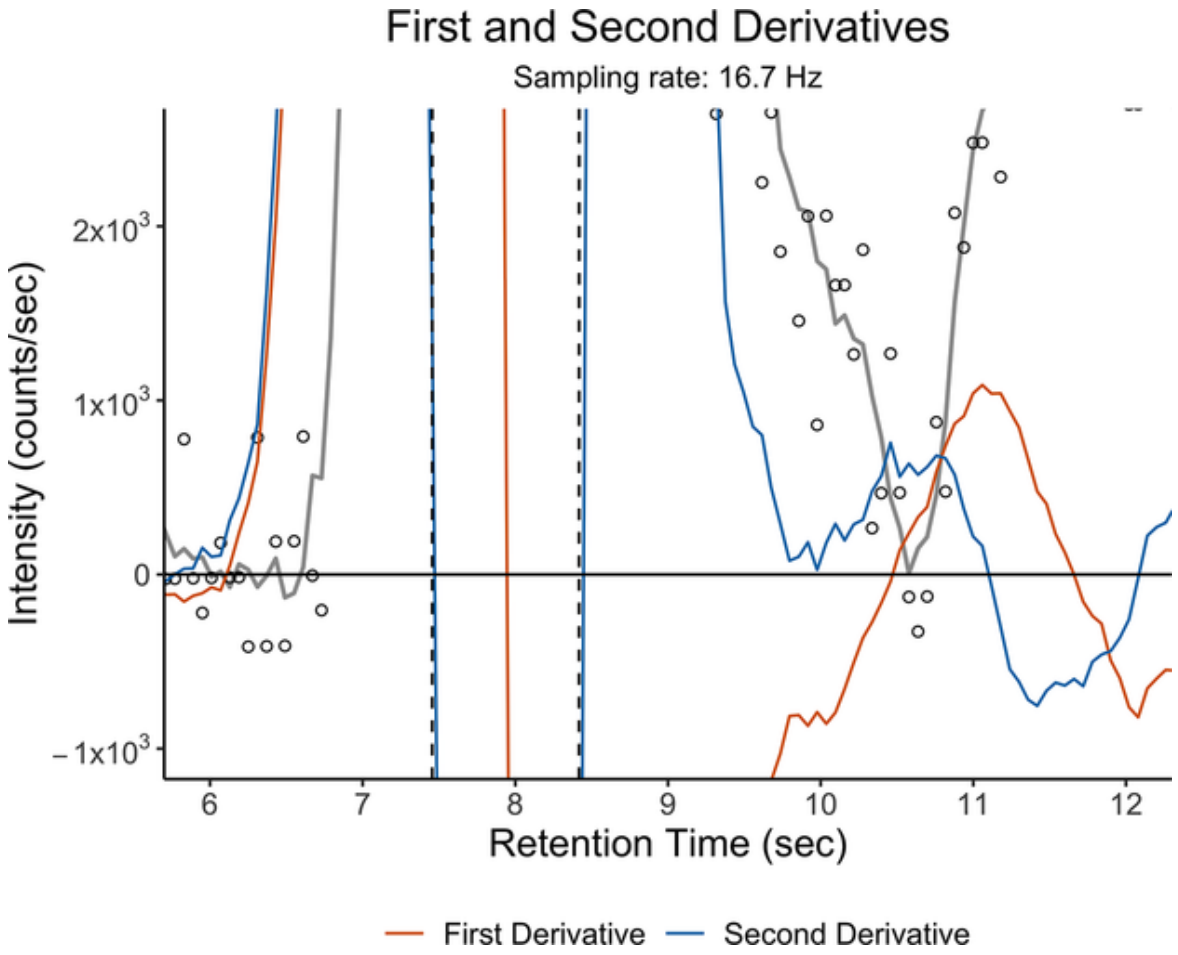


Figure 6.6 First and second derivative variations around zero. The first derivative becomes positive first, followed by the second derivative at the front of the peak. Note that the second derivative zero crossings are close to the FWHM positions (dashed lines).

It is better to define the beginning and end of a peak in terms of expected variation in the data. In this specific trace, it is relatively easy to estimate the expected variation of the random component of the data before the first peak. Still, because of chemical noise, a more robust method for estimating noise and, thus, the threshold for the derivatives is needed.

6.3.2 Data Smoothing and Noise Calculation

For many years, there has been disagreement on the objective of smoothing. Early in applying computer analysis to mass spectrometry data, smoothing has been viewed as having “cosmetic value only,” or to “make information more easily accessible to human interpretation.” [161]. This interpretation was challenged at least as early as 1886 by Sprague, addressing the subject of smoothing actuarial data [164]. Whittaker summarized Sprague’s position as: “[If you asked someone] what was the real object of graduation? [an older term for smoothing] Probably the reply would be, To get a smooth curve; but he did not think that quite correct. To his mind, the reply should be to get the most probable deaths” [154]. Likewise, in analytical chemistry, smoothing is not cosmetic. It is statistical. The goal is to replace values obtained from an imperfect instrument with values more likely to have come from the underlying chemical process, which is known to be smooth when many molecules are involved. You may not understand the underlying process completely and probably don’t know the statistical distribution of the random or deterministic processes. However, you do know that the deterministic component in chromatography involves a large number of molecules, so it must be smooth.

To perform an estimate of the dispersion of the random component of the noise, it is necessary to determine what type of random noise is generated by mass spectrometry. If the random noise comes from the same distribution with constant dispersion over the time of the measurement, it’s called *homoskedastic*. If the distribution or the dispersion of noise varies over time, then it is called *heteroskedastic*. To risk stating the obvious, it is much easier to estimate the properties of noise when the distribution and dispersion are fixed over the course of a measurement.

To evaluate the noise, I will use the smooth data to approximate the underlying chemical process and estimate the random noise in the signal. Performing this evaluation in R is straightforward. The smooth trace is computed based on the filter length established for the peak of interest as described above and then subtracted from the raw trace to obtain a *deviation* vector. If the smoothed data is treated as an approximation of the true underlying deterministic signal, then the deviation vector represents an approximation of the random noise in the data.

```
deviation <- y - y_smooth
```

Plotting this data will show if the deviation vector can be used directly for noise calculations.

```
p_deviation <- ggplot() +
        scale_y_continuous(labels = inten_label) +
        geom_point(aes(x=t, y=deviation), shape=1 ) +
        geom_vline(xintercept=2, col=pal$darkorange) +
        geom_vline(xintercept=6, col=pal$darkorange) +
        xlab("Retention Time (sec)") +
        ylab("Deviation from Smooth (counts/sec)") +
        ggtitle(label= "Deviation From Smooth",
                subtitle = "SG Quadradic, Length 9") +
        theme_classic() +
        theme(plot.title = element_text(hjust = 0.5,
size=16)) +
        theme(plot.subtitle = element_text(hjust =
0.5)) +
        theme(
            axis.text=element_text(size=11),
            axis.title=element_text(size = 14),
            legend.text = element_text(size = 11),
            legend.title = element_text(size = 14)
        )

print(p_deviation)
```

From Figure 6.7, it is clear that the noise in this data varies with the intensity of the signal. Peaks have more dispersion at the top than they do near the baseline. The plot also suggests that the signal contains only random signals from retention time 2 seconds until about 6 seconds. The simplest description for random noise is the normal distribution. Since the electronic and other types of noise involve large numbers of events, this assumption is usually good. If it holds, then much of the statistical calculations become straightforward. However, just like with the assumption of constant dispersion, the data should be tested for *normality* before using statistical functions that assume a normal distribution.

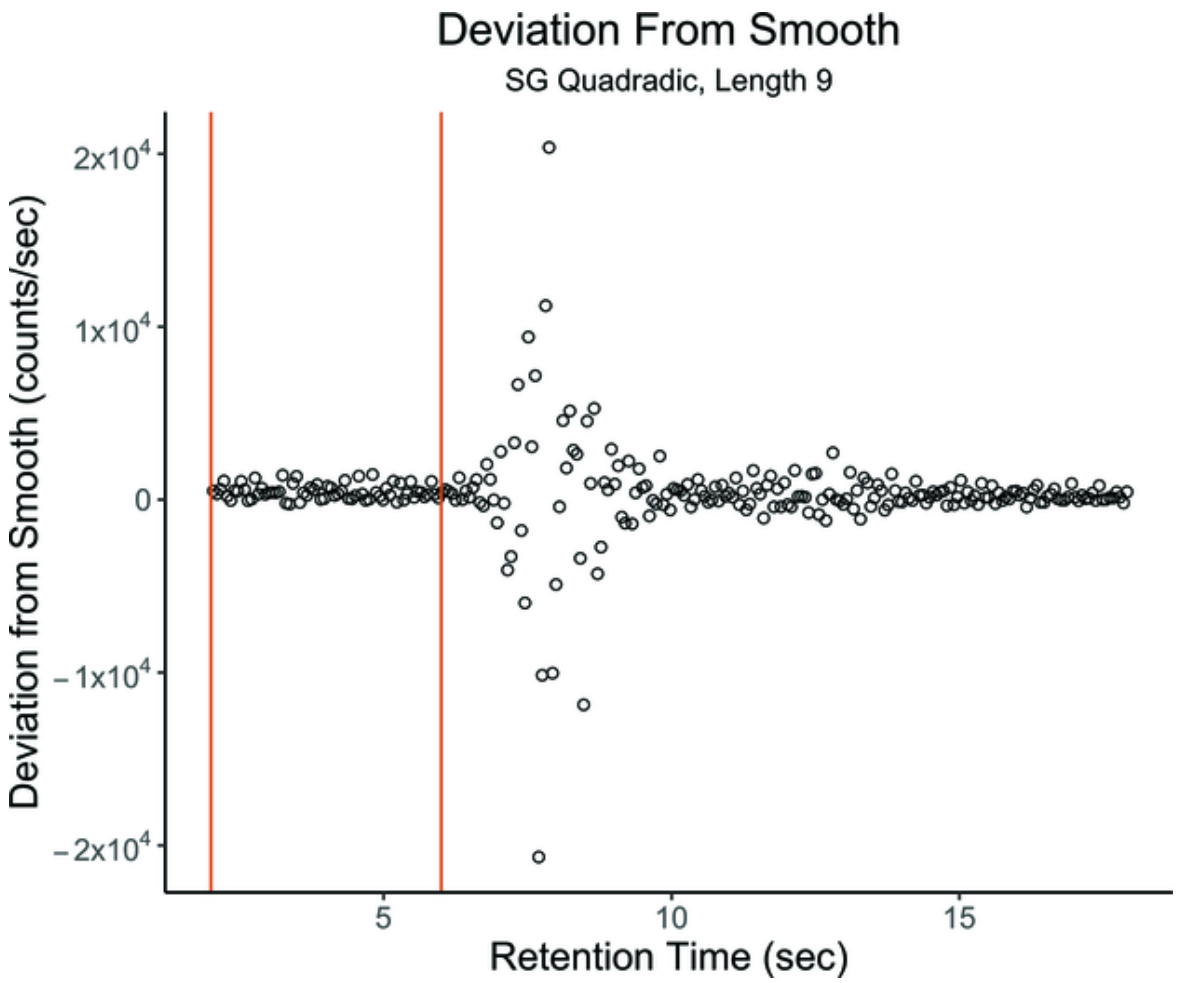


Figure 6.7 Deviations of the raw data from the smoothed trace using the SG quadratic filter with length 9. The heteroskedastic nature of the noise in mass spectrometry is clear from the larger deviations as the amplitude of the peak increases. The guidelines represent the minimum and maximum time range for the peak.

The test that the noise data is normally distributed can be done visually with the Q-Q plot and quantitatively with the Shapiro–Wilk test using the `shapiro.test()` function from the `stats` package. I'll start with the Q-Q plot, commonly used with linear regression as a diagnostic for the assumptions of the least squares method of performing a regression.

```
# Select from the start of the acquisition to the end
of the chemical-free
# portion of the signal (6 seconds in this example)
blank_region <- data.frame(t = t[(t < 6)], y = y[(t <
6)])
```

This gives a `data.frame` containing only the chemical-free region.

```
p_qq <- blank_region |>
    ggplot(aes(sample = y)) +
        stat_qq() +
        stat_qq_line() +
        xlab("Theoretical Quantiles") +
        ylab("Sample Quantiles") +
        ggtitle(label= "Normal Q-Q Plot",
                subtitle = "Raw Intensity 2-6s") +
        theme_classic() +
        theme(plot.title = element_text(hjust = 0.5,
size=16)) +
        theme(plot.subtitle = element_text(hjust =
0.5)) +
        theme(
            axis.text=element_text(size=11),
            axis.title=element_text(size = 14),
            legend.text = element_text(size = 11),
            legend.title = element_text(size = 14)
        )
print(p_qq)
```

Figure 6.8 shows some extraordinary aspects of this data. First, the data appears to be *left-censored* at zero. This means all values below some count rate are given as zero. Censored data cannot be treated in the same way as uncensored data. An observation was made, but it was just below the limit of the detector to measure. Any estimate of location (mean) or dispersion (standard deviation) will be incorrect because the data do not represent the actual

distribution but the part above the detection limit. Second, this instrument (a Sciex 5500 Triple Quadrupole Mass Spectrometer) quantized the data.

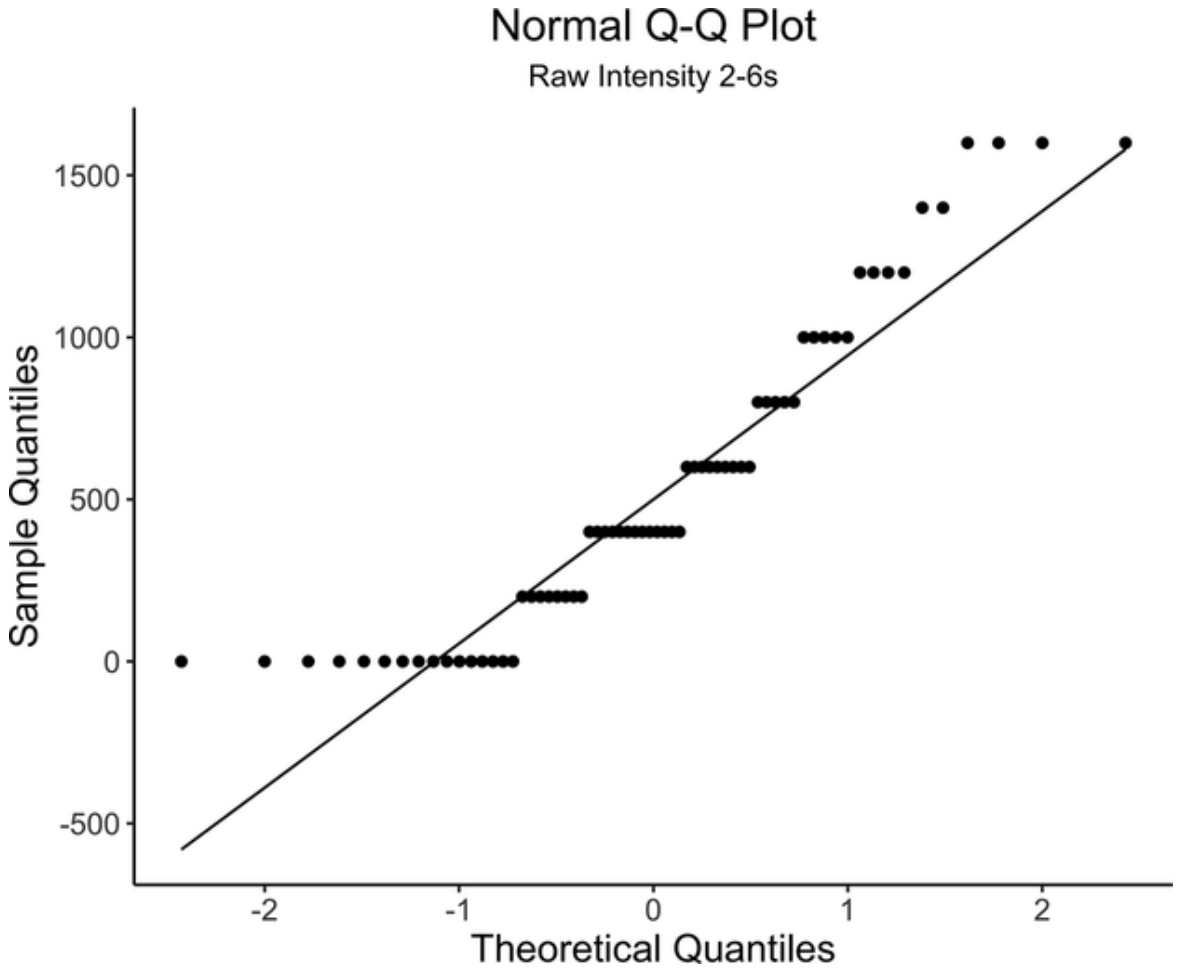


Figure 6.8 Q-Q plot of the noise component of raw intensity data. The data are not normal and show signs of censoring at zero.

All the data in the blank region reported as `0` in the Q-Q plot are likely to be *non-detects*. However, the remaining data from the upper quantiles can be used to estimate the distribution. Non-detects are a common occurrence when making measurements near a limit of detection. Certainly, the baseline noise falls into this category. Figure 6.8 visually indicates that the data are not normally distributed, so any summary statistic assuming an underlying normal

distribution will give incorrect results. The Shapiro–Wilk [165, 166] is a test of normality that can provide a quantitative measure of normality [167], which is one of the best tests, especially for relatively small datasets. The `shapiro.test()` function provided by the `stats` package implemented based on the Royston 1995 paper [168].

```
  shapiro.test(blank_region$y)
```

```
##
##  Shapiro-Wilk normality test
##
## data:  blank_region$y
## W = 0.89362, p-value = 3.602e-05
```

The Shapiro–Wilk test computes a statistic related to correlation, `W`, which is interpreted using the p-value. The null hypothesis is that the `blank_region` data comes from a normally distributed random noise process. A p-value below 0.05 means you can reject the null hypothesis and say that the data is not normally distributed. If the data were normally distributed, the familiar formulas for mean and standard deviation (which are the maximum likelihood estimates [MLE]) shown in Eqs. (6.1) and (6.2) can be found in almost any basic statistics textbook and can be used to estimate the *location* (center of the distribution) and the *dispersion* (standard deviation) of the noise.

$$mean = \frac{1}{n}\sum_{i=i}^{n} x_i \tag{6.1}$$

$$sd = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2} \tag{6.2}$$

However, these formulas implicitly assume that the mean is the highest probability expected value from the data and that the standard deviation is derived from the σ in the equation for the Normal distribution. If I use these formulas on the raw data in Figure 6.8, I will get an incorrect value for the standard deviation. I know that the data is not normally distributed, but from the plot, it appears to be *left-censored* and that no value below zero is reported. Further, the number of zero values present is too high for a normally distributed noise region. This guess might be wrong, and the noise might follow another distribution or be censored. However, if I can get an estimate of the standard deviation under the assumption that the data come from a left-censored normal distribution, I can use that to estimate noise from these measurements.

Interlude: Complete (Non-Censored) Normally Distributed Noise

Data will not always be censored. Before showing how to determine dispersion in the case of censored data, I want to show an example of *non-censored* or *complete* data. Briefly, non-censored data is shown in Figure 6.9a. The chemical-free portion is shown in Figure 6.9b, and the Q-Q plot of the blank portion of the data is shown in Figure 6.9c. It would appear from the Q-Q plot that the noise region is normally distributed, and the Shipiro–Wilk test confirms this intuition.

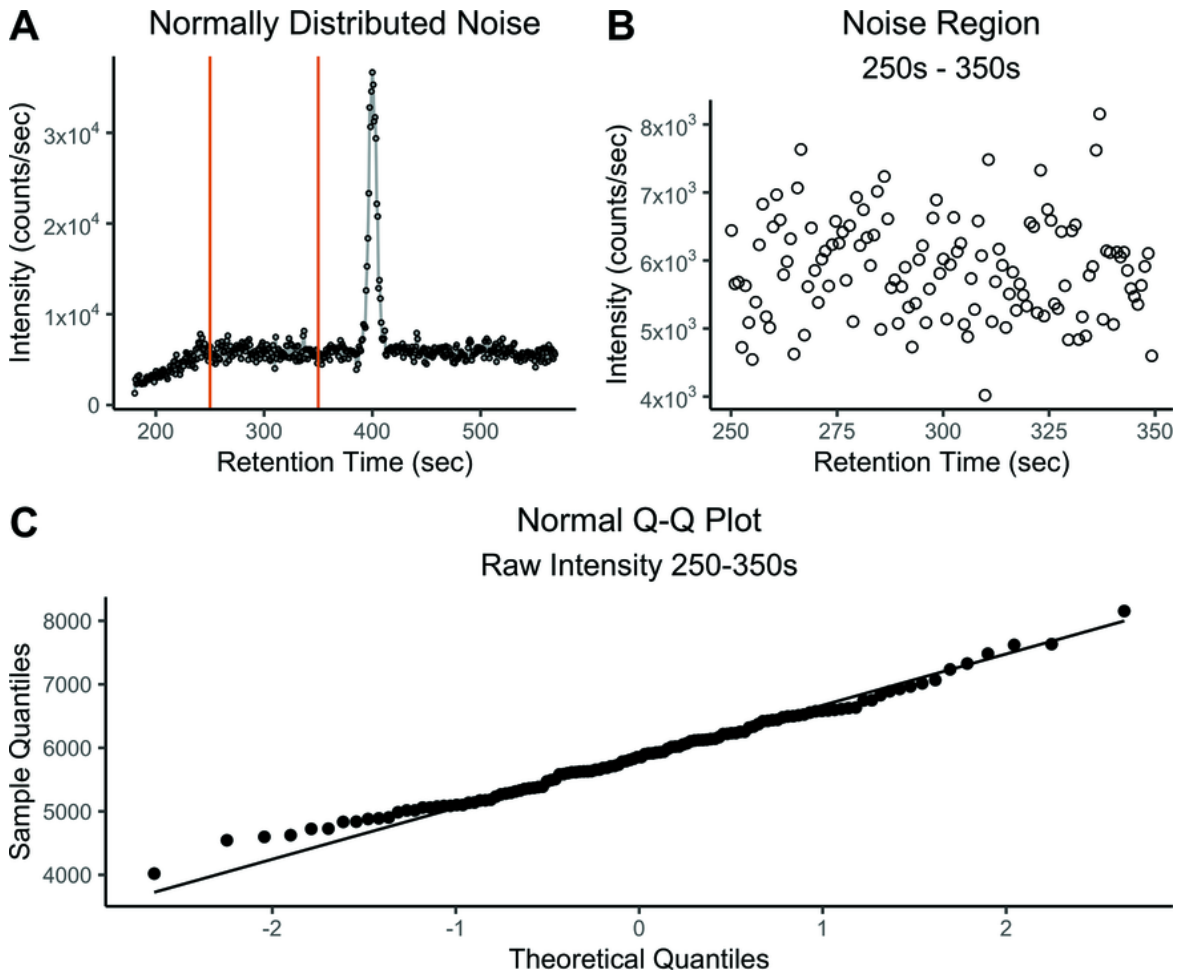


Figure 6.9 Chromatogram with normally distributed noise.

```
##
##  Shapiro-Wilk normality test
##
## data:  blank_region_complete$y
## W = 0.98955, p-value = 0.4823
```

Unlike the data in Figure 6.8, the p-value for the Shapiro-Wilk W statistic for the data in Figure 6.9 is far above 0.05, so the null hypothesis (that the data comes from a normal distribution) is not rejected, leaving the conclusion that functions like `mean()` and `sd()` will work properly on this data:

```
## Mean of complete data: 5877.9
## SD of complete data  :  737.8
```

In this example data, I can use `sd()` to determine when the raw signal exceeds a `99%` threshold roughly estimated by `2.5` standard deviations from the mean. My goal is to use the measure of dispersion to set limits based on the raw data dispersion to set thresholds for the *derivatives* shown in Figure 6.6 in order to determine when the peak begins and ends.

Back to Censored Data

For the data in Figure 6.8, I will use the `EnvStats` package [169], which includes functions to work with censored data. It's useful to know how to do this because not only can raw data be censored by the instrument, but an assay for a specific compound can also be censored by the limits of quantitation (lower and upper). First, I'll try the standard methods on the raw data from the blank region.

```
blank_y_mean <- mean(blank_region$y)
blank_y_sd   <- sd(blank_region$y)

cat(sprintf("Mean of y before peak : %6.1f",
blank_y_mean), "\n",
    sprintf("SD of y before peak   : %6.1f",
blank_y_sd), sep="")
```

```
## Mean of y before peak :  533.3
## SD of y before peak   :  480.2
```

The Q-Q plot for the censored data supports the idea that the random data come from a censored normal population. To use this and the related `EnvStats` approach, you have to create a logical vector that contains a 0 (FALSE) for each missing value.

```
censored <- blank_region$y==0
```

Now, the `qqPlotCensored()` function can be used to visualize the Q-Q plot for the censored data.

```
qqPlotCensored(blank_region$y, censored = censored,
add.line = TRUE,
               main= "Q-Q Plot for Censored Noise
Region",
               ylab= "Quantiles of Noise Region")
```

The plotting positions of data points in Figure 6.10 produced by `EnvStats` are slightly different from the `qqnorm()` function from the base R `stats` package. The default positions for `qqPlotCensored()` come from Michael and Schucany [170]. The exact details are not critical but useful when comparing results from the two packages.

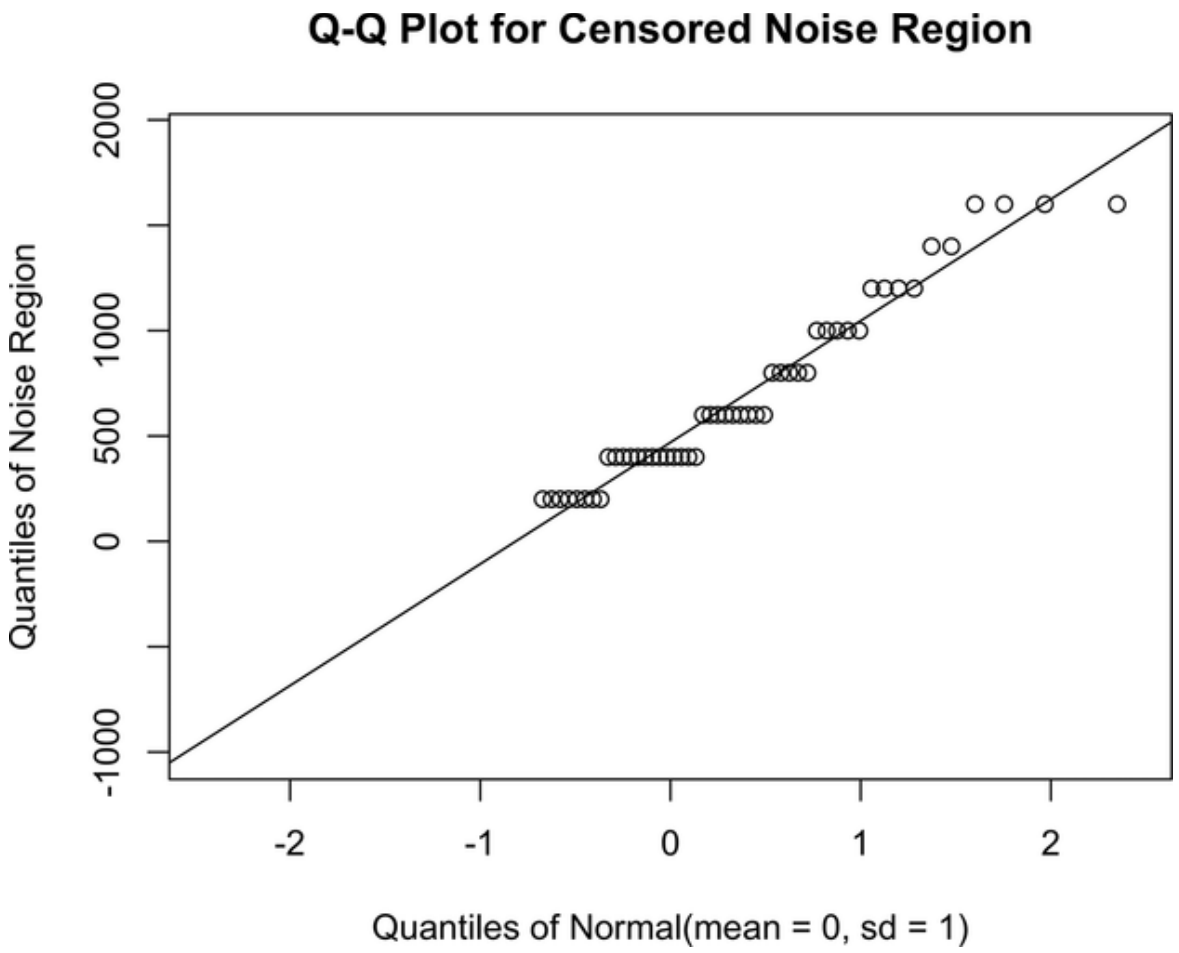


Figure 6.10 Q-Q plot for censored noise region.

The `enormCensored()` function is used to estimate the location (mean) and dispersion (sd) of censored normal data.

First, you label the censored data elements so they can be used to determine how much of the data is missing and where it is in the distribution. Then, choose a method for estimating the underlying normal distribution. The method I've chosen is called `rROS`, which stands for Robust Regression on Order Statistics [171–175]. The basic idea of `rROS` is to impute the missing values by performing a linear regression on non-censored data in the Q-Q plot. The slope of the regression line in the Q-Q plot is the standard deviation of the normal distribution, and the y-intercept is the estimated mean. Those values are then used to estimate the intensity values of the missing data. Once the missing data

has a value, then the MLE method can be applied to the complete dataset, now with imputed values for the censored observations. In the `enormCensored()` function, the parameter `method` is used to select `rROS` as the approach. The `rROS` algorithm has several advantages for mass spectrometry and chromatography data because it performs very well on small datasets. In the example above, there are only 66 data points, and they are quantized into only eight specific levels. So, with a limited number of unique values to determine slope and intercept, `rROS` is a good choice. Further, it has been recommended by the USEPA [176] for more than a decade. Robust methods that work with small sample sizes and stand the test of time are few and far between!

```
enc <- enormCensored(blank_region$y, censored =
censored, method = "rROS")
print(enc)
```

```
##
## Results of Distribution Parameter Estimation
## Based on Type I Censored Data
## ----------------------------------------------
##
## Assumed Distribution:            Normal
##
## Censoring Side:                  left
##
## Censoring Level(s):              0
##
## Estimated Parameter(s):          mean = 469.7344
##                                  sd   = 573.3882
##
## Estimation Method:               Imputation with
##                                  Q-Q Regression (rROS)
##
## Data:                            blank_region$y
##
## Censoring Variable:              censored
##
```

```
## Sample Size:                     66
##
## Percent Censored:                24.24242%
```

Looking at the `rROS` estimates for standard deviation compared to the MLE value, it's clear that treating the censored values as measured values at zero artificially lowered the estimated standard deviation and, thus, the apparent noise of the instrument. The actual noise (when it rises above the censored level) has a standard deviation ~19% larger, meaning that for a signal to be distinguished from noise, its intensity has to exceed a larger value than suggested by simply measuring the noise from the highest value to the lowest value.

Since approximately 25% of the data are missing, it is worth looking into imputing the missing values from the estimated normal distribution parameters found with `enormCensored()`. If there is too much missing data to be ignored, values can be imputed using various methods. Imputing values from either an observed or known distribution or from some other characteristic of the data can be useful. In this case, I will start by ignoring the fact that the data are censored and use the raw data as it is since it will be smoothed during the calculation of the first and second derivatives, or impute the `0` values to values drawn from the missing portion of the estimated distribution.

Next, I'll show how to impute the censored values shown in Figure 6.8 using the feature of the Q-Q plot that the slope represents the standard deviation, and the y-intercept represents the mean of the distribution.

```
cen_norm_mean <- enc$parameters["mean"]
cen_norm_sd <- enc$parameters["sd"]
```

For the censored data, I can replace the `0` (missing values) by simply using $y = sd * x + mean$ where the x is simply the

theoretical quantile (x value) seen in Figure 6.8. The x values can be computed using the `qqnorm()` function.

```
qq_blank <- qqnorm(blank_region$y)
```

Now, I'll make a new version of the `blank_region`, replacing the censored data with values selected at random from the missing theoretical quantiles.

```
# create a new y vector to hold the imputed values
imputed_y <- blank_region$y
```

The censored index values of the blank are:

```
which(censored)
```

```
##  [1]  1  2  6 11 12 22 23 26 32 33 40 41 42 45 46 50
```

Since these are ordered, the x values are also ordered from lowest to highest.

```
qq_blank$x[which(censored)]
```

```
##  [1] -2.4287371 -2.0004236 -1.7758504 -1.6161559
-1.4894700 -1.3829941
##  [7] -1.2902332 -1.2074141 -1.1321396 -1.0627865
-0.9982012 -0.9375322
## [13] -0.8801319 -0.8254945 -0.7732170 -0.7229722
```

Since censoring occurs randomly through the blank region, I will replace the imputed values randomly in the `0` positions in the data. The `sample()` function randomizes the vector of index values.

```
randomized_censored <- sample(which(censored))
```

Now, using a simple loop, I'll go through the 17 censored values and replace them with a randomly selected quantile value, which is converted into a y value using the slope and intercept of the Q-Q regression line found by the rROS method.

```
for(i in 1:length(which(censored))) {
    imputed_y[randomized_censored[i]] <-
        qq_blank$x[which(censored)[i]] * cen_norm_sd +
cen_norm_mean
}
```

To finish, I replace the raw y vector with the imputed y vector.

```
blank_region_imputed <- blank_region
blank_region_imputed$y <- imputed_y
```

```
p_imputed <- blank_region_imputed |>
    ggplot(aes(sample = y)) +
    stat_qq() +
    scale_y_continuous(labels = inten_label, n.breaks =
6) +
    geom_abline(aes(slope=cen_norm_sd, intercept =
cen_norm_mean,
                    color= "Imputed"), key_glyph =
draw_key_path) +
    geom_abline(aes(slope=sd(blank_region$y), intercept
= mean(blank_region$y),
                    color= "Raw"), key_glyph =
draw_key_path) +
        scale_color_manual(name='Q-Q Line: ',
                     breaks=c('Imputed', 'Raw'),
                     values=c(pal$darkorange,
pal$black)) +
    xlab("Theoretical Quantiles") +
    ylab("Sample Quantiles") +
    ggtitle(label = "Imputed Q-Q Plot for Censored
Normal Distribution",
                 subtitle= sprintf("Raw Intensity
%0.1fs-%0.1fs",
                     blank_region_imputed$t[1],
max(blank_region_imputed$t))) +
    theme_classic() +
    theme(plot.title = element_text(hjust = 0.5,
size=16)) +
    theme(plot.subtitle = element_text(hjust = 0.5)) +
    theme(
        axis.text=element_text(size=11),
        axis.title=element_text(size = 14),
        legend.text = element_text(size = 11),
        legend.title = element_text(size = 14)
    ) +
    theme(
        legend.position.inside = c(.1, .95),
        legend.justification = c("left", "top"),
        legend.box.just = "right",
        legend.margin = margin(6, 6, 6, 6)
```

```
    )
print(p_imputed)
```

Figure 6.11 shows the effects of imputing the missing values using the estimated mean (y-intercept) and standard deviation (slope) computed using `rROS` assuming the zero values in the data are censored data. Compare the imputed mean and standard deviation values obtained using `rROS` to the values obtained by assuming the zero values in the data are real. It's clear that treating the zero values as real lowers the estimated standard deviation, artificially lowering the estimated dispersion which artificially lowers the estimated noise content of the data.

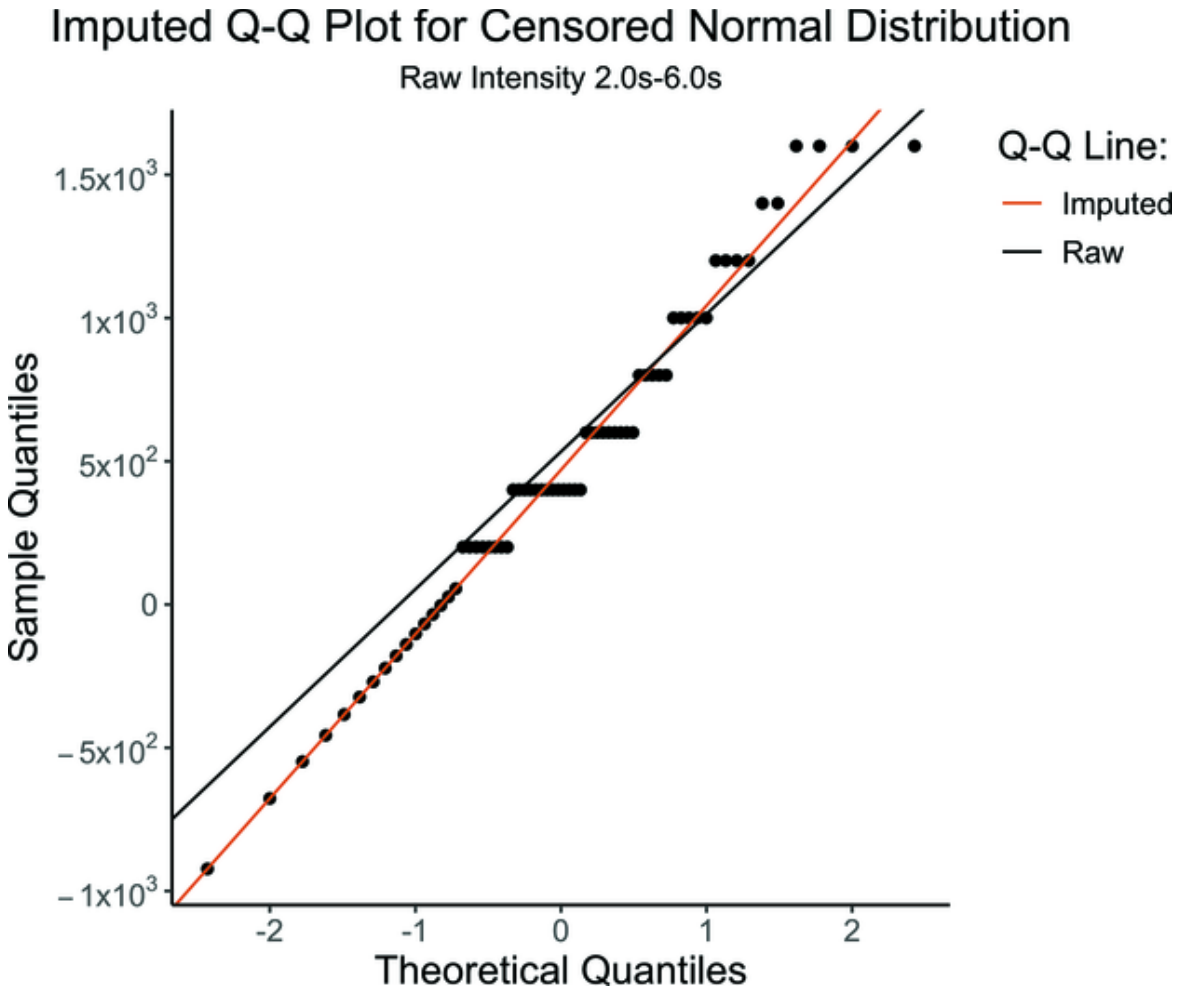


Figure 6.11 Q-Q plot showing the imputed values and the lines for the normal distributions with imputation and without (raw).

Using the derivative of the trace to detect the beginning and end of a peak relies on smoothing, so now I can check the effects of the imputation on the smoothed data to determine how the threshold value for the first and second derivatives should be calculated.

```
y_blank_raw_smooth <- sgolayfilt(blank_region$y,
n=sg_length, p=2)
y_blank_imp_smooth <-
sgolayfilt(blank_region_imputed$y, n=sg_length, p=2)
```

These raw and imputed traces are smoothed with the filter length computed from the peak in Figure 6.5 using `get_sg_filter_length()` from Section 6.3.1. Now I'll compare the smoothed raw and imputed data.

```
p_smooth_blank <- ggplot() +
    coord_cartesian(ylim=c(-1e3, 1.5e3)) +
    scale_y_continuous(labels = inten_label) +
    geom_point(aes(x=blank_region_imputed$t,
                   y=blank_region_imputed$y, color=
"Imputed")) +
    geom_point(aes(x=blank_region$t,
                   y=blank_region$y, color= "Raw")) +
    geom_line(aes(x=blank_region_imputed$t,
                  y=y_blank_imp_smooth, color=
"Imputed")) +
    geom_line(aes(x=blank_region$t,
                  y=y_blank_raw_smooth, color= "Raw"))
+
    xlab("Retention Time (sec)") +
    ylab("Intensity (counts/sec)") +
    ggtitle(label= "Smoothed Traces Prior to Peak",
            subtitle = sprintf("SG Order 2, Length %d,
RT: 2-6s", sg_length)) +
    theme_classic() +
    theme(plot.title = element_text(hjust = 0.5,
size=16)) +
    theme(plot.subtitle = element_text(hjust = 0.5)) +
    theme(
        axis.text=element_text(size=11),
        axis.title=element_text(size = 14),
        legend.text = element_text(size = 11),
        legend.title = element_text(size = 14)
    ) +
    scale_color_manual(
                name='',
                breaks=c('Raw', 'Imputed'),
                values=c(pal$blue, pal$red)) +
    theme( legend.position = "bottom")

print(p_smooth_blank)
```

Even with imputation, the two smoothed traces in Figure 6.12 are not that different. This is one of the strengths of

smoothing. I know that the imputed data are normally distributed. What about the smoothed versions of both?

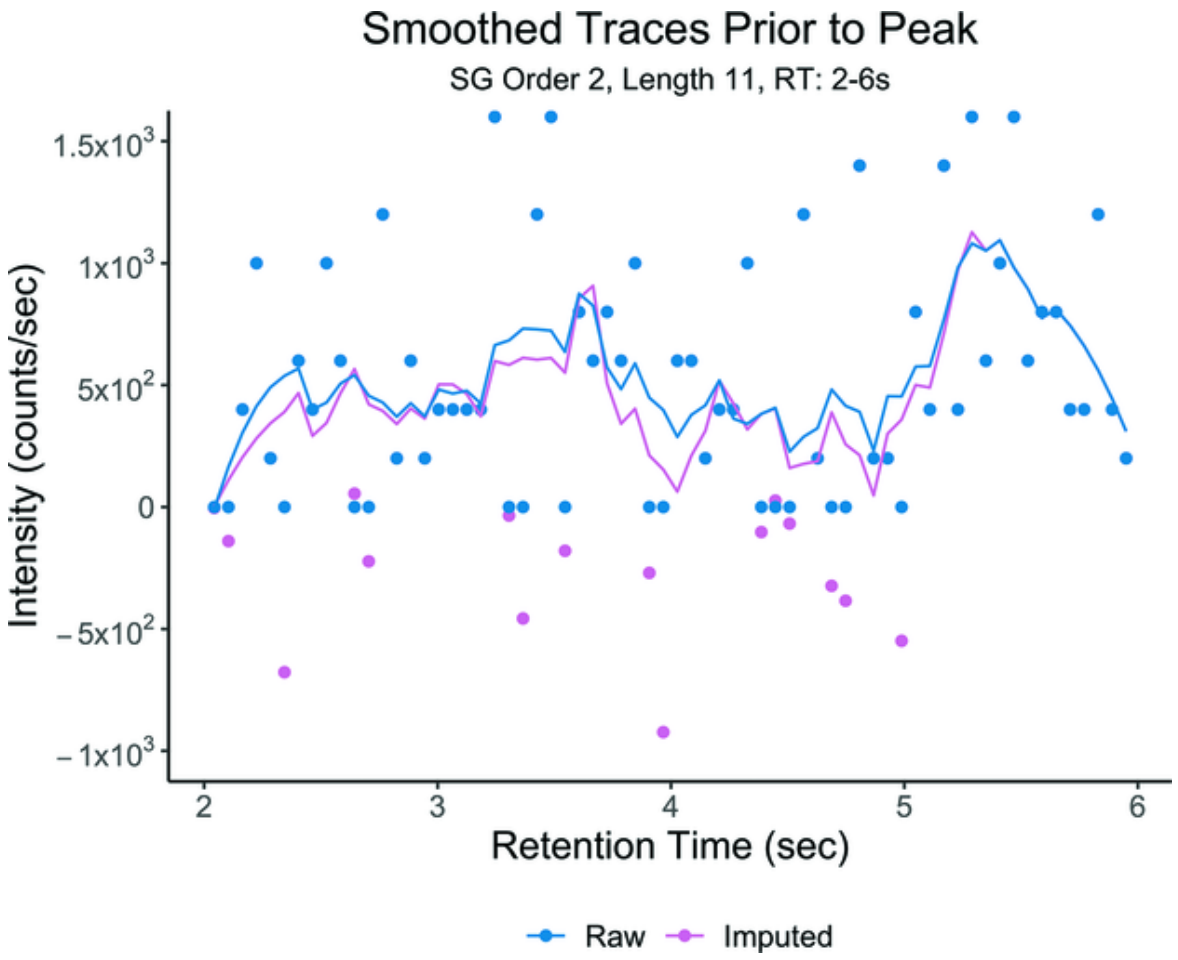


Figure 6.12 Blank region smoothed with the SG order two filters. Smoothing the imputed values is not significantly different from smoothing the raw values.

```
cat(sprintf("Mean of Raw    : %0.1f",
mean(y_blank_raw_smooth)),"\n",
    sprintf("Mean of Imputed: %0.1f",
mean(y_blank_imp_smooth)), sep="")
```

```
## Mean of Raw    : 534.7
## Mean of Imputed: 469.2
```

```
  cat(sprintf("SD of Raw    : %0.1f",
  sd(y_blank_raw_smooth)),"\n",
      sprintf("SD of Imputed: %0.1f",
  sd(y_blank_imp_smooth)), sep="")
```

```
## SD of Raw    : 225.4
## SD of Imputed: 303.8
```

```
  shapiro.test(y_blank_raw_smooth)
```

```
##
##  Shapiro-Wilk normality test
##
## data:  y_blank_raw_smooth
## W = 0.94818, p-value = 0.007958
```

```
  shapiro.test(y_blank_imp_smooth)
```

```
##
##  Shapiro-Wilk normality test
##
## data:  y_blank_imp_smooth
## W = 0.98037, p-value = 0.3797
```

While the imputed smoothed data easily exceeds the threshold for normality, smoothing the raw data also passes the minimum threshold for using MLE estimates. This means that the function `sd()` can be used on the smoothed first and second derivatives to establish a threshold.

If the smooth data were not at least minimally normal, then the data should be imputed using a procedure like what has been shown above, and then `sd()` can be used to determine when the first and second derivatives have reached a conservative value close to zero where the probability of fluctuations being random is high. This is the key to the peak

detection problem. The level where you can say a peak is present or not is the *limit of detection*. When using peak area to quantify a compound, you become interested in the detection limit for the beginning and end of a peak. The *limit of quantification* is an entirely different concept, tied to the ability to reproduce a signal height or area from a specific quantity of material responsible for the deterministic component of the trace.

For the example being used in the chapter, several simplifications were made. First, there is no chemical noise from the beginning of the acquisition time until very close to the peak of interest. Second, despite being censored, the smoothed data in this blank region is close enough to be normally distributed to use the simple `sd()` functions, which assume normally distributed noise. Third, there is no interfering peak following too close to the peak of interest, so the area can be calculated from the peak front edge to the back edge after correcting for the baseline computed in Section 6.2.3.

6.3.3 Using Derivatives to Find the Start and End of a Peak

To find the peak start and end, I will compute the dispersion of the derivatives shown in Figure 6.6. I need to know when the first and second derivatives have exceeded a value that can reliably be called noise *in each derivative trace*. Even though I've already shown that using the `sd()` function underestimates the dispersion for this data, it can still work as a threshold for the derivatives computed using the `sgolayfilt()` function. If this does not hold up for your data, you must use one of the more sophisticated dispersion estimates, like the one discussed for censored data above.

```
d1_threshold <- sd(sgolayfilt(blank_region$y,
n=derivitive_length,
                              p=2, m=1, ts=0.25))
d2_threshold <- sd(sgolayfilt(blank_region$y,
n=derivitive_length,
                              p=2, m=2, ts=0.25))
range_start <- expected_rt - rt_tolerance
range_end <- expected_rt + rt_tolerance
```

I'll create a function called `get_rising_points()` to compute a list of data points that fit the criteria of a rising edge of a peak and return them as a vector of trace index values.

```
get_rising_points <- function(y_d1, y_d2, d1_threshold,
d2_threshold) {
    which((y_d2 > d2_threshold) & (y_d1 >
d1_threshold))
}
```

The index values of the rising edge elements show a pattern for real peaks: there are consecutive rising elements in the array leading up to the FWHM level:

```
rise <- get_rising_points(y_d1, y_d2, d1_threshold,
d2_threshold)
print(rise)
```

```
##  [1]  47  48  70  71  72  73  74  75  76  77  78  79
80  81  82  83  84  85  86
## [20]  87  88  89  90  91 142 143 144 145 146 147 148
149 150 151
```

Notice that array elements `70-91` are consecutive. Consecutive rising elements can be used as a heuristic for the presence of the rising edge of a peak. One variation on this heuristic would be that a real peak has to contain at least as many rising data points as the computed filter width to be

a member of the peak with a width that produced the filter in the first place. Narrow peaks with fewer data points above FWHM will have fewer data points in the rising edge than wide peaks.

To implement the heuristic, I'll define a function called `get_peak_start()` that checks the `rise` array for consecutive indexes representing continuous segments of rising data points. The `min_segment_length` parameter is the minimum length of this segment needed for the peak to be considered present. The minimum segment length could be set to any length depending on the situation, but a reasonable minimum would be `3` since that is the minimum possible FIR filter width. The peak starts at the first element of a segment at least as long as the `minimum_segment_length`.

```
get_peak_start <- function(t, rise, range_start,
range_end,
                           apex_index,
min_segment_length=3) {

    if(length(rise) < min_segment_length) {
        return(NA)
    }

    run_length <- 1
    start_index <- 1
    range_index_list <- which(t > range_start & t <
range_end)

    for(i in 1:(length(rise)-1)) {

        if(rise[i] < range_index_list[1]) {
            start_index <- i + 1
            next
        }

        if(rise[i] >
range_index_list[length(range_index_list)]) {
            break
        }

        if(rise[i+1] == rise[i] + 1) {
            run_length <- run_length + 1
        } else {
            run_length <- 1
            start_index <- i + 1
        }
        if(run_length >= min_segment_length) {
            break
        }
    }

    peak_start_index <- rise[start_index] - 1
    if(peak_start_index < apex_index) {
        peak_start_index
    } else {
```

```
        NA
    }
}
```

Now I call `get_peak_start()` to find the index of the start of the peak (which starts at the last baseline point):

```
peak_start_index <- get_peak_start(t, rise,
range_start, range_end,
                                   apex_index_smooth,
min_segment_length = 3)

sprintf("Peak start index %d: Start time %0.2fs",
        peak_start_index, t[peak_start_index])
```

```
## [1] "Peak start index 69: Start time 6.13s"
```

To find the end of the peak, I use the same procedure as the start of the peak. I created a function that gets the array indices that represent the falling edge of a peak called `get_falling_points()`

```
get_falling_points <- function(y_d1, y_d2,
d1_threshold, d2_threshold) {
    which((y_d2 > d2_threshold) & (y_d1 < -
d1_threshold))
}
```

Get the falling edge points using the same threshold:

```
fall <- get_falling_points(y_d1, y_d2, d1_threshold,
d2_threshold)
print(fall)
```

```
##  [1] 108 109 110 111 112 113 114 115 116 117 118 119
120 121 122 123 124 125 126
## [20] 127 128 129 131 132 134 135 136 137 138 139 140
```

```
169 170 171 172 173 174 175
## [39] 176 179 180 191 199 200 201 202
```

In the function `get_peak_end()` below, instead of the first element of the contiguous segment, the end of the peak is the last element of a segment, at least as long as the `minimum_segment_length`. The peak end is one past the last falling data point to bring the end of the peak back to the baseline:

```r
get_peak_end <- function(t, fall, range_start,
range_end,
                         apex_index,
min_segment_length=3) {

    if(length(fall) < min_segment_length) {
        return(NA)
    }

    run_length <- 1
    end_index <- 1
    range_index_list <- which(t > range_start & t <
range_end)

    for(i in 1:(length(fall)-1)) {

        if(fall[i] < range_index_list[1]) {
            next
        }

        if(fall[i] >
range_index_list[length(range_index_list)]) {
            break
        }

        if(fall[i+1] == fall[i] + 1) {
            run_length <- run_length + 1
            end_index <- i + 1
        } else {
            if(run_length >= min_segment_length) {
                break
            } else {
                run_length <- 1
            }
        }
    }

    peak_end_index <- fall[end_index] + 1

    if(peak_end_index > apex_index) {
        peak_end_index
```

```
    } else {
        NA
    }
}
```

Like with the peak start index, the peak end index is found calling the function `get_peak_end()`:

```
peak_end_index <- get_peak_end(t, fall, range_start,
range_end,
                          apex_index_smooth,
min_segment_length = sg_length)

sprintf("Peak end index %d: End time %0.2fs",
        peak_end_index, t[peak_end_index])
```

```
## [1] "Peak end index 130: End time 9.80s"
```

Plotting the peak features shows how well the threshold and segment length parameters worked for this data:

```
p_picked <- chrom_plot(t, y_points = y, y_line =
y_smooth,
                main_title = "Sample 11 Quantifier",
                sub_title = sprintf("Start: %0.1fs -
End %0.1fs",

t[peak_start_index], t[peak_end_index]),
                points = TRUE)

p_picked <- p_picked +
    coord_cartesian(xlim=c(6,11), ylim=c(-2.5e2, 4e2))
+
    geom_line(aes(x=t, y=y_d1), color=pal$darkorange) +
    geom_point(aes(x=t, y=y_d1), color=pal$darkorange,
shape=16) +
    geom_line(aes(x=t, y=y_d2), color=pal$blue) +
    geom_point(aes(x=t, y=y_d2), color=pal$blue,
shape=16) +
    geom_hline(yintercept=0.0, color=pal$black) +
    geom_hline(yintercept = d2_threshold,
color=pal$blue,
               linetype= "dashed") +
    geom_hline(yintercept = -d2_threshold,
color=pal$blue,
               linetype= "dashed") +
    geom_hline(yintercept = d1_threshold,
color=pal$darkorange,
               linetype= "dashed") +
    geom_hline(yintercept = -d1_threshold,
color=pal$darkorange,
               linetype= "dashed") +
    geom_vline(xintercept=t[peak_start_index],
               linetype="dashed", color=pal$green,
linewidth=0.75) +
    geom_vline(xintercept = t[peak_end_index],
               linetype="dashed", color=pal$green,
linewidth=0.75)

print(p_picked)
```

Figure 6.13 shows the start and end time (vertical dashed lines) of the peak, as well as the estimates (positive and negative) of the noise levels from the scaled first and second derivative, smoothed, imputed non-signal region of the trace. From this type of plot, you can diagnose any issues with peak picking and the calculated start and end of a peak.

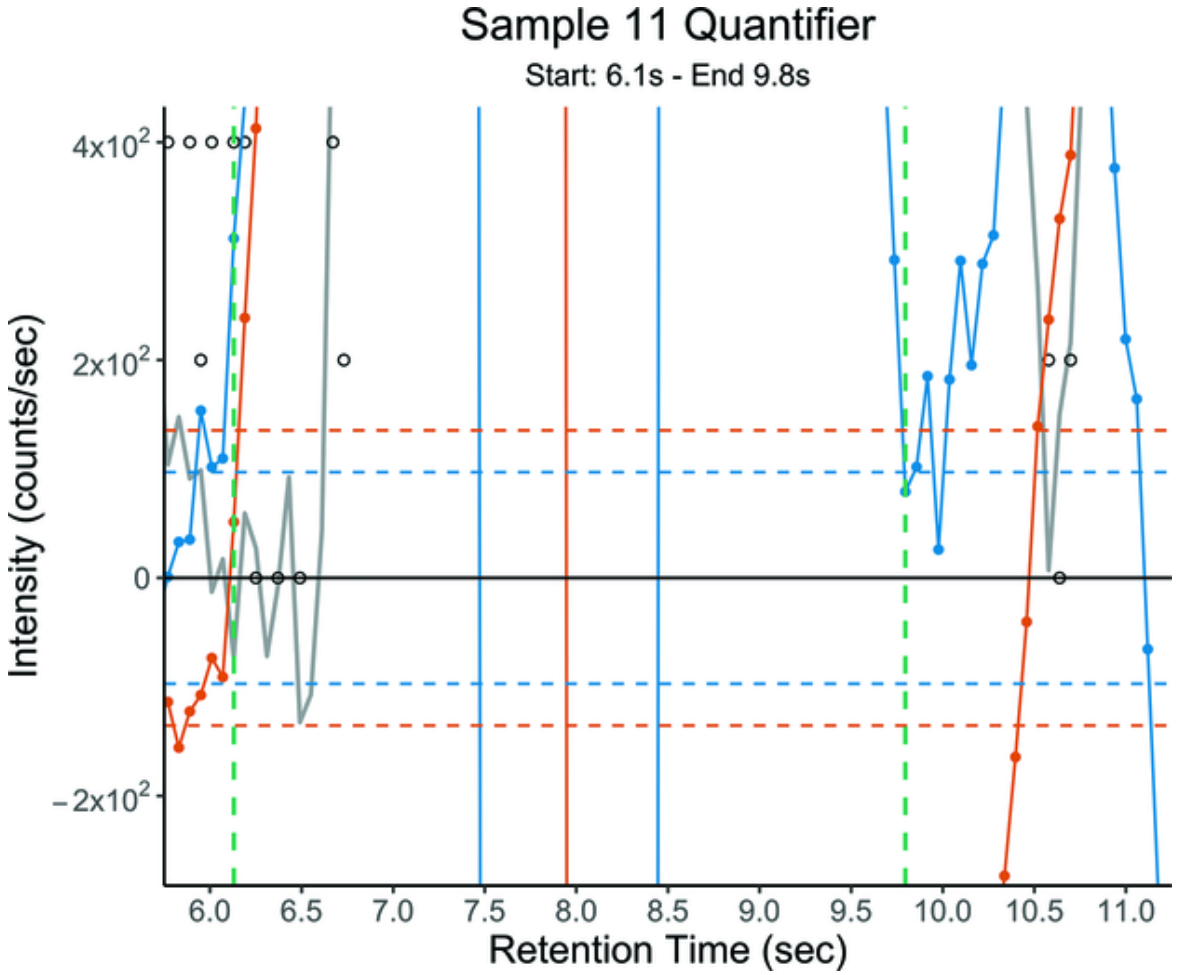


Figure 6.13 Start and end times picked using the derivative method.

6.3.4 Wavelet-based Peak Detection

The derivative method for picking peaks uses a convolutional (running average) digital filter selected to match the expected width of the peak to be characterized. The SG

filters for the first and second derivatives were used to find the start and end of the peak. The SG filters fit a polynomial locally to a set of data points, shift one data point down the array, and perform another polynomial fit. In the example above, a parabola shape was used for the fit. Another highly related method uses the *continuous wavelet transform* (CWT) to approximate the deterministic component of the trace using a different shape function.

The shape used most commonly in mass spectrometry is nicknamed the *Mexican Hat* [104, 177, 178], otherwise known as the *Maar Wavelet* [179]. The Mexican Hat gets its name from its resemblance to the cross-section of a sombrero. The equation is the negative (flipped on the x-axis) of the second derivative of a Gaussian. The function has a scale parameter that determines the width and is related to the σ of the original Gaussian. This is identical to the SG second derivative concept, in which smoothing is combined with taking the second derivative. Taking the negative of the second derivative is for convenience so that the maxima of the filter output represents the apex of a peak.

Wavelet analysis of both mass spectra and chromatograms can be performed using the Bioconductor package `MassSpecWavelet`. The `cwt()` function calculates the coefficients of the wavelet selected. The default is the `mexh()` function, but custom functions can also be used. The `mexh()` function specifies the *mother wavelet*. This basic shape will be scaled for each scale level (expected peak width) specified. The Mexican Hat used by the `MassSpecWavelet` package is defined from `-8 to 8` according to the documentation for the function. When a scale is selected, the function is sampled on intervals of `1/scale`.

```
p_mexh <- data.frame(x=seq(-8,8, by=1/64),
                     y=mexh(seq(-8,8, by=1/64))) |>
    ggplot(aes(x=x, y=y)) +
        scale_x_continuous(breaks = seq(-8,8, by=2)) +
        scale_y_continuous(limits = c(-0.5, 1.0),
                           breaks = seq(-0.75,1.25,
by=0.25)) +
        geom_line() +
        xlab("Index") +
        ylab("Normalized Amplitude") +
        ggtitle(label= "Mexican Hat (Maar) Wavelet",
            subtitle = "Scales from 1 to 64") +
        theme_classic() +
        theme(plot.title = element_text(hjust = 0.5,
size=16)) +
        theme(plot.subtitle = element_text(hjust =
0.5)) +
        theme(
            axis.text=element_text(size=11),
            axis.title=element_text(size = 14),
            legend.text = element_text(size = 11),
            legend.title = element_text(size = 14)
        )
print(p_mexh)
```

When applied to the quantitation trace from sample 11 used in the example of derivative-based peak detection, the `cwt()` function will produce a matrix of wavelet coefficients. The columns are the scale values, which, while recommended to be between `1` and `64`, can be generated at a finer resolution. I will use `0.1` increments for the scale values. The scale values are similar to the filter length selection in Section 6.3.1. Specifically, the scale value I expect to see a maximum value for is where the width of the peak (in data points above FWHM) is approximately equal to the scale. The suggested range for Gaussian peaks is a scale range from $(1, 1.9\sigma)$ [180].

The `MassSpecWavelet` package has some basic plotting functions, like most Bioconductor packages; however, I will

show how to plot the wavelet coefficients using `ggplot2`. Getting the coefficients is straightforward. You provide the `y` values of the trace and specify the scale vector. Here, I will use scales from `1` to `24` since my peak has `15` data points above the FWHM (approximately σ) and step by `0.1`. The `mexh()` function name is supplied as the wavelet.

```
scales <- seq(1, 24, 0.1)
wCoefs <- cwt(y, scales = scales, wavelet = "mexh")
```

I'll use the same approach shown in Section 4.3.3.2 to create an image from the matrix. I'll add the retention time for the x-axis and then use `pivot_longer()` so I can use `geom_raster()` to make an image.

```
w_t <- as_tibble(wCoefs) |>
    mutate(rt=t, .before=as.character(scales[1]))

w_t_long <- w_t |>
    pivot_longer(cols=!"rt", names_to="scale",
values_to = "intensity") |>
    mutate(scale=as.numeric(scale))
```

Since the shape shown in Figure 6.14 will generate large negative values unrelated to the peak location, I'll zero these values to get the data into a dynamic range that can be visualized.

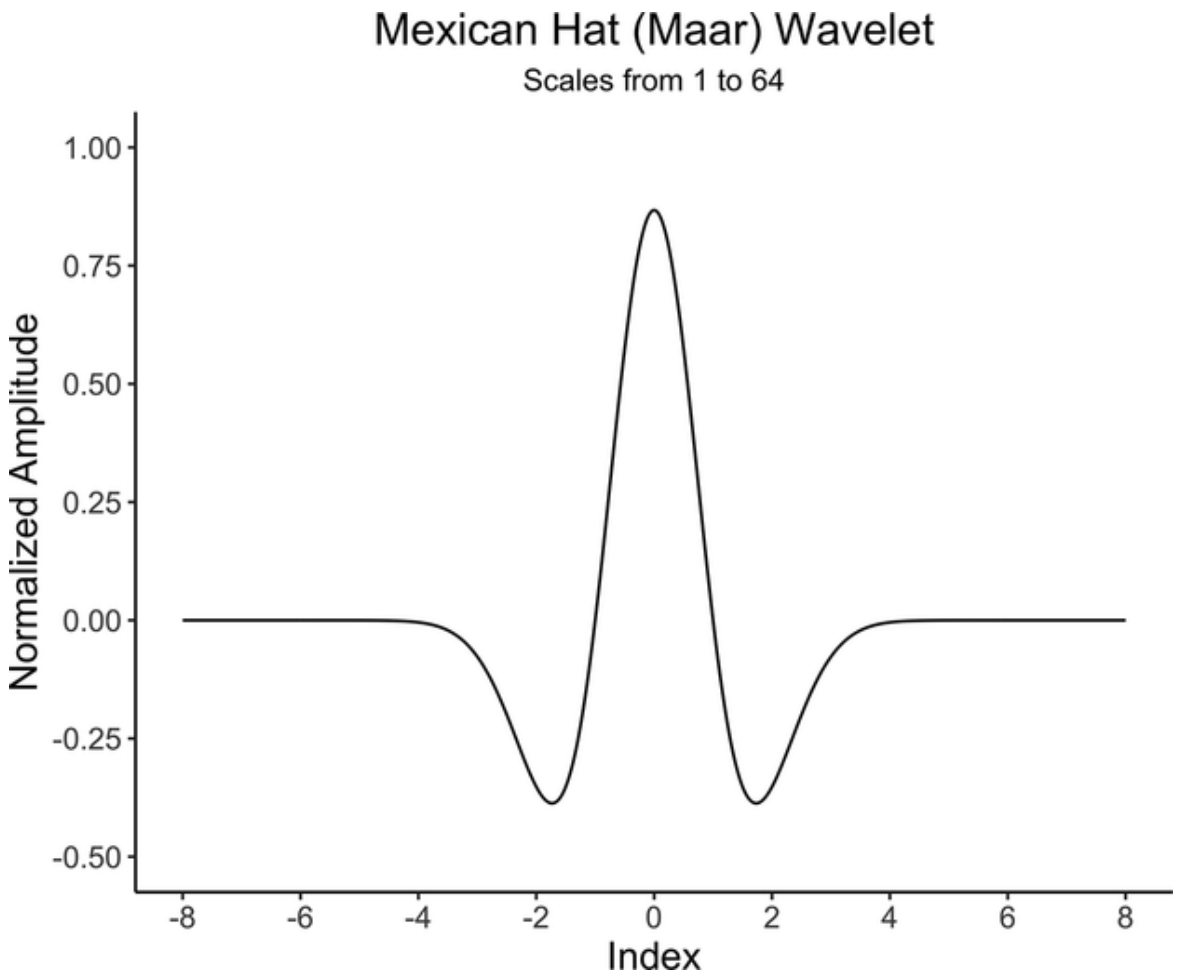


Figure 6.14 The Mexican Hat mother wavelet used in "MassSpecWavelet," which can be used with scale values from 1 to 64.

```
w_t_long$intensity[w_t_long$intensity<0] <- 0
```

I'm using the `geom_raster()` layer to plot the `intensity` as a color (`fill`) at every x and y point on the plot. If the `rt` variable is not uniformly spaced, this will cause a shift in the plot for that x value. If `geom_tile()` is used with a nonuniform axis (x or y), the spacing will be maintained, but vertical lines will run through the plot where the spacing is uneven. If this shift is a problem for your analysis, binning the axis to create uniform spacing will solve the visualization shift problem.

```
p_cwt <- w_t_long |>
    ggplot(aes(x = rt, y = scale, fill = intensity)) +
    geom_raster() +
    scale_fill_viridis_c(limits=c(0, 3e4),
                         na.value = "white", option =
"plasma") +
    coord_cartesian(xlim=c(2,18), ylim=c(1,24), expand
= FALSE) +
    scale_x_continuous(breaks = seq(2,18,1)) +
    scale_y_continuous(breaks = seq(2,24,2)) +
    xlab("Retention Time (sec)") +
    ylab("CWT Coefficient Scale") +
    theme_classic() +
    theme(plot.title = element_text(hjust = 0.5)) +
    theme(plot.subtitle = element_text(hjust = 0.5)) +
    ggtitle(label = "CWT Coefficients Sample 11 Quant
Trace",
            subtitle = "Mexican Hat Wavelet Scales 1-
24")
print(p_cwt)
```

In Figure 6.15, the main peak gives a significant intensity at almost every scale, and the smaller peak at the later retention time appears over a smaller scale range and, ultimately, a narrower width than the main peak. It's easy to see that the scale matches what was found in Section 6.3.1 for the data point count above the FWHM by zooming in on the peak apex.

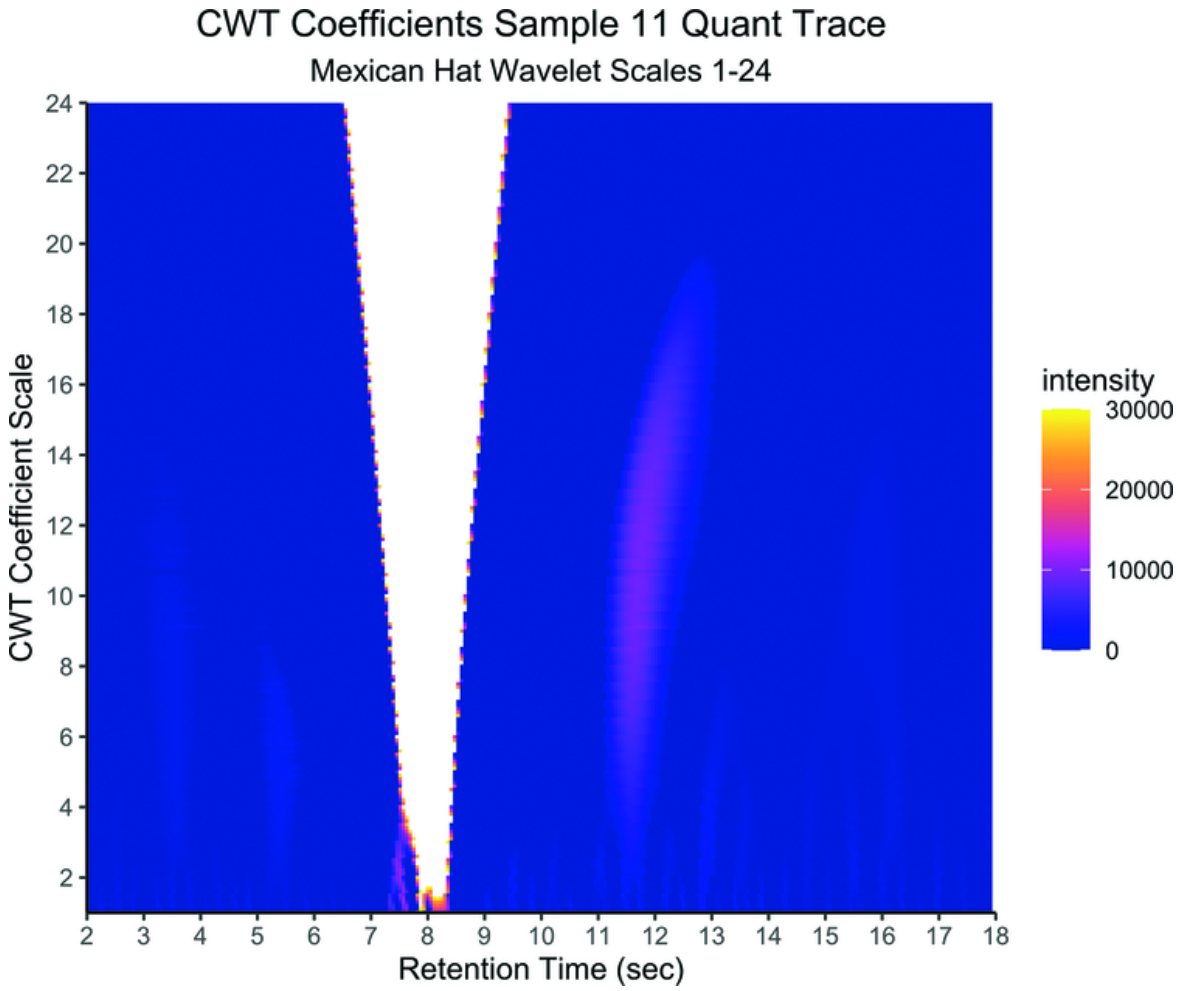


Figure 6.15 CWT coefficients for the wavelet transformation at multiple sales for the quantifier trace from sample 11.

```
points_above_fwhm <- length(which(t > front_50 & t < back_50))
sprintf("Points above FWHM: %d", points_above_fwhm)
```

```
## [1] "Points above FWHM: 15"
```

Zooming in on the retention time of the main peak, and marking the scale at which the peak was detected shows the relationship between the derivative method using FWHM and the wavelet scale method for locating the peak apex (Figure 6.16):

```
p_cwt_peak <- p_cwt +
    scale_fill_viridis_c(limits=c(0, 5.9e5),
                         na.value = "white", option =
"plasma") +
    coord_cartesian(xlim=c(6,10), ylim=c(1,24), expand
= FALSE) +
    geom_segment(aes(x=6, y=points_above_fwhm,
                     yend = points_above_fwhm, xend =
10),
                 linewidth = 0.25, linetype="dotted",
color="white") +
    scale_x_continuous(breaks = seq(6,10,1)) +
    scale_y_continuous(breaks = seq(2,24,2))

print(p_cwt_peak)
```

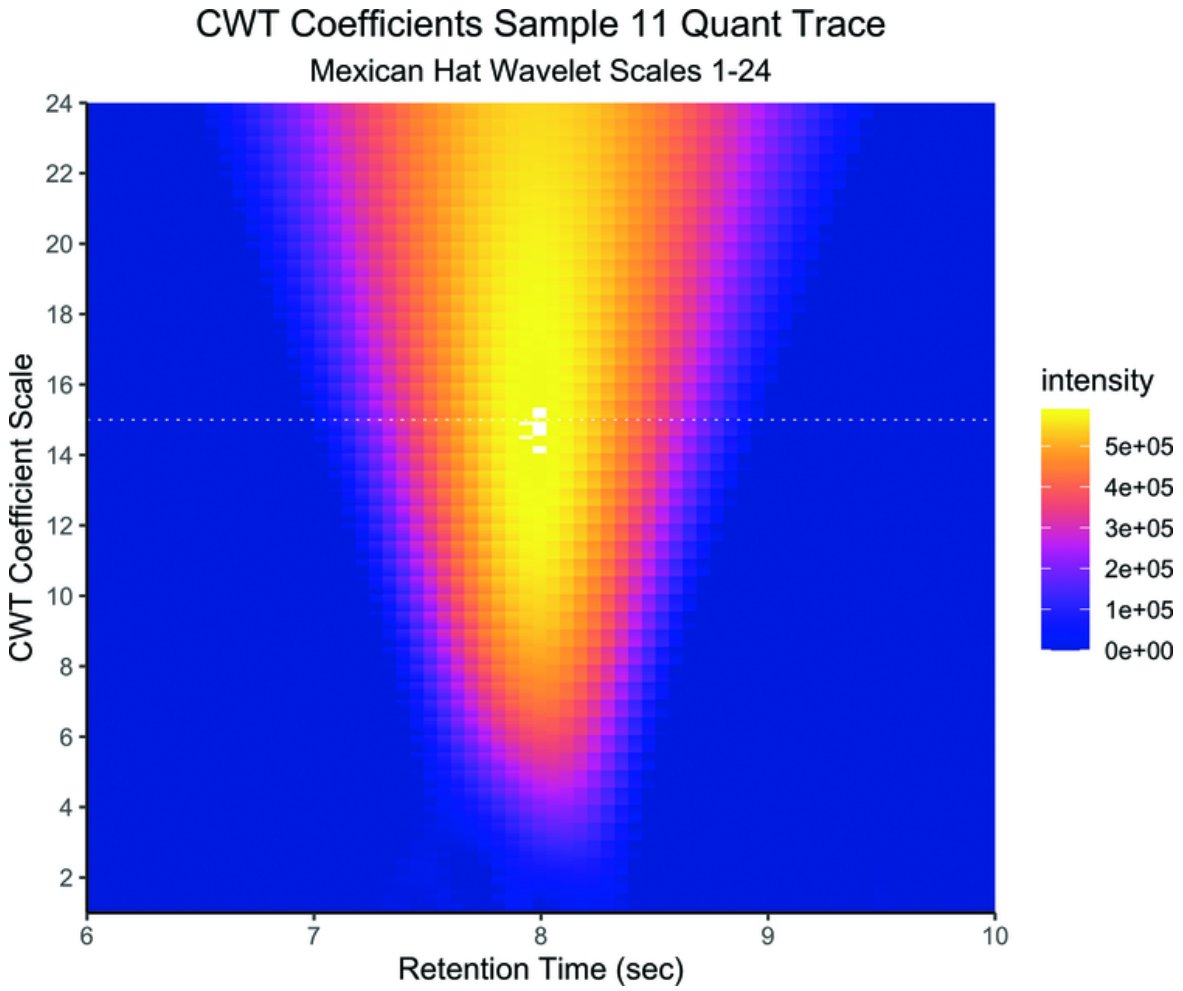


Figure 6.16 CWT coefficients main peak showing the local maximum in both intensity and scale.

Another way to analyze the wavelet coefficients is to plot the locations of the local maxima using a sliding window. This is done using the `getLocalMaximumCWT()` function.

```
local_max <- getLocalMaximumCWT(wCoefs)
```

The `local_max` matrix is similar to the `wCoef` matrix, except that locally maximum values are set to `1` and all other values are set to `0`. This allows the visualization of all peaks independent of amplitude and their scales.

```
local_max_t <- as_tibble(local_max) |>
    mutate(rt=t, .before=as.character(scales[1]))

local_max_t_long <- local_max_t |>
    pivot_longer(cols=!"rt", names_to="scale",
values_to = "local_max") |>
    mutate(scale=as.numeric(scale))
```

Now the local maxima can be plotted like the wavelet coefficients (Figure 6.17).

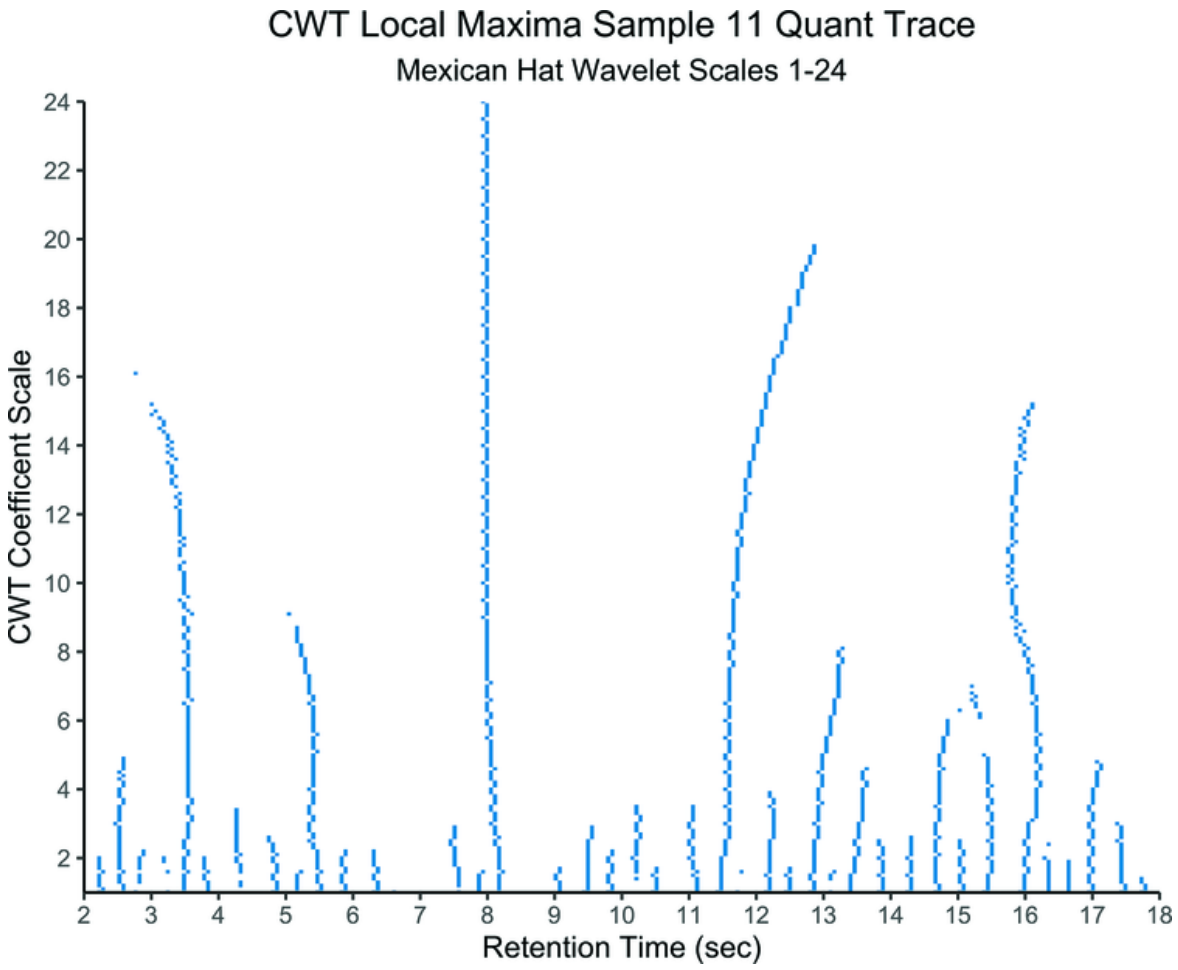


Figure 6.17 Local maxima of CWT coefficients.

```
p_local_max <- local_max_t_long |>
    ggplot(aes(x = rt, y = scale, fill = c("white",
pal$blue)[local_max + 1])) +
        geom_raster() +
        scale_fill_identity() +
        coord_cartesian(xlim=c(2,18), ylim=c(1,24),
expand = FALSE) +
        scale_x_continuous(breaks = seq(2,18,1)) +
        scale_y_continuous(breaks = seq(2,24,2)) +
        xlab("Retention Time (sec)") +
        ylab("CWT Coefficent Scale") +
        theme_classic() +
        theme(plot.title = element_text(hjust = 0.5)) +
        theme(plot.subtitle = element_text(hjust =
0.5)) +
```

```
        ggtitle(label = "CWT Local Maxima Sample 11
Quant Trace",
            subtitle = "Mexican Hat Wavelet Scales 1-
24")
print(p_local_max)
```

6.3.5 Wavelet Noise Estimation

Like the smoothed trace estimate for noise used in Section 6.3.2, the smallest scale wavelet coefficients can be used to estimate noise in regions of the trace [104, 177].

```
p_scale_1 <- ggplot() +
        scale_y_continuous(labels = inten_label) +
        geom_point(aes(x=t, y=wCoefs[,1]), shape=1 ) +
        geom_vline(xintercept=2, color=pal$darkorange)
+
        geom_vline(xintercept=6, color=pal$darkorange)
+
        xlab("Retention Time (sec)") +
        ylab("CWT Coefficient Intensity") +
        ggtitle(label= "CWT Coefficients - Sample 11",
                subtitle = "Mexican Hat - Scale 1") +
        theme_classic() +
        theme(plot.title = element_text(hjust = 0.5,
size=16)) +
        theme(plot.subtitle = element_text(hjust =
0.5)) +
        theme(
            axis.text=element_text(size=11),
            axis.title=element_text(size = 14),
            legend.text = element_text(size = 11),
            legend.title = element_text(size = 14)
        )

print(p_scale_1)
```

Notice the similarities between Figures 6.18 and 6.7. The smallest scale wavelet coefficients are from the highest frequency components in the trace, so they are similar to the deviations computed from using a smoothed trace. The CWT coefficients also show the same trace intensity dependency as the deviation vector, giving more evidence to the heteroskedastic nature of LC-MS data discussed throughout this chapter. As suggested by Du et al. [177], the standard deviation of the region without chemical noise can be used to determine the noise in the trace because the values are approximately normally distributed.

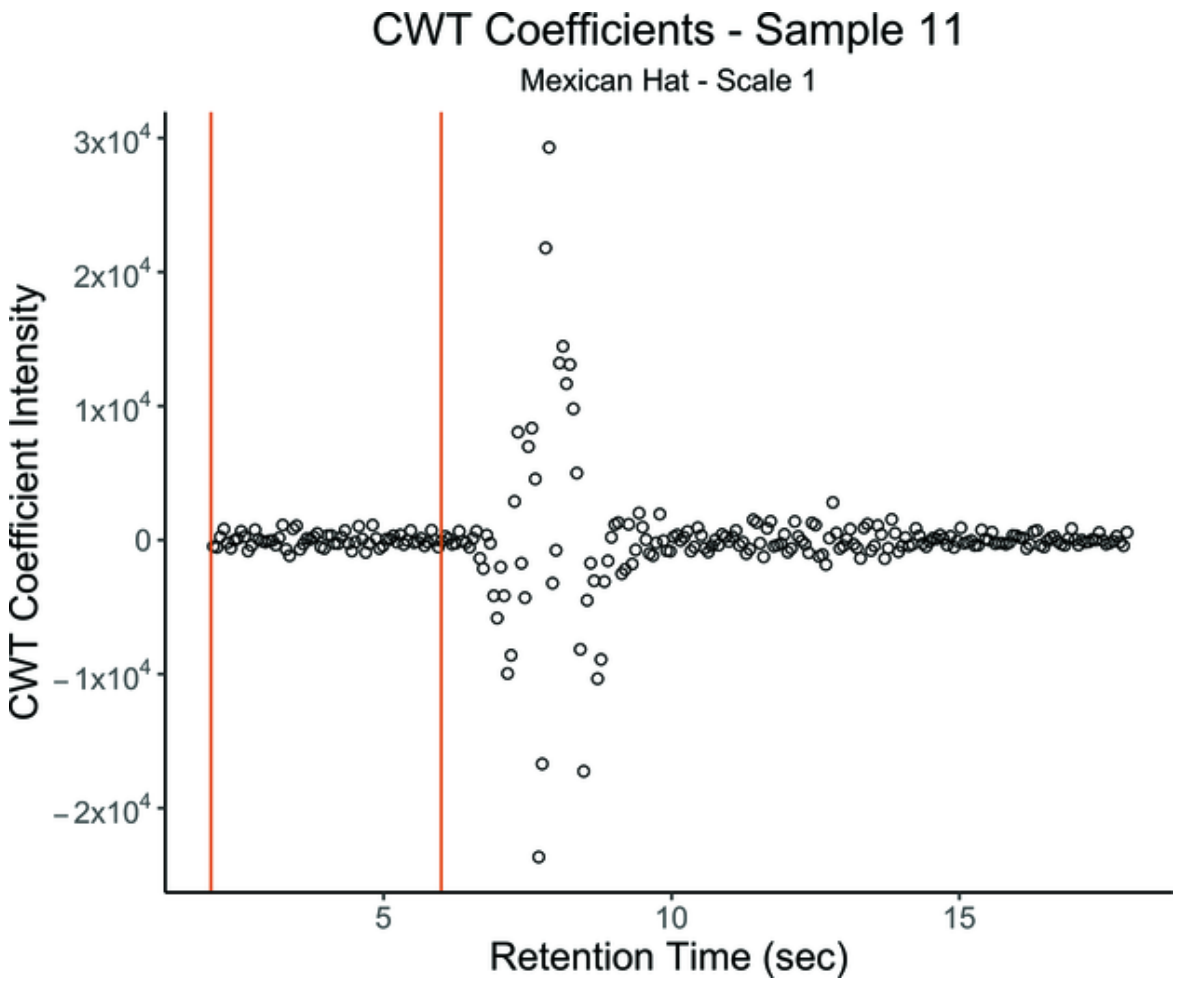


Figure 6.18 Wavelet coefficients from the scale = 1 transform of the sample 11 trace.

```
noise_cwt <- wCoefs[which(t<6),1]

shapiro.test(noise_cwt)
```

```
##
##  Shapiro-Wilk normality test
##
## data:  noise_cwt
## W = 0.97843, p-value = 0.3045
```

```
sd(noise_cwt)
```

```
## [1] 523.6226
```

The noise estimate using the lowest CWT scale is close to the MLE estimate from the raw data but lower than the estimate when the raw data's censored nature is considered. It is larger than the standard deviation computed by subtracting the raw from the SG smoothed data, which suggests that the scale 1 Mexican hat wavelet did more smoothing than the length 11 SG filter.

What the SG and Gaussian filters do to the data before the derivatives are taken can be understood by looking at the data from the perspective of *frequency content*. What I have been doing with both the SG convolution and the CWT convolution can be seen using Fourier analysis since the convolution in the time domain is identical to multiplication in the frequency domain. No matter what shape I use to perform a running average-type smoothing of the data, it will affect which frequencies present in the data are attenuated or amplified by the multiplication operation.

6.3.6 Using Wavelets to Find the Start and End of a Peak Start

The `xcms` package has functions to compute peak parameters from the CWT. The function `peaksWithCentWave()` picks *all* the peaks in a trace that fit between a minimum and maximum peak width specified in the retention time `t` units. For the example trace, the peak FWHM is approximately 0.9 seconds, and from the trace plot and the CWT plot, there is a second peak at a later time, which is smaller and narrower. In this case, I'll specify peaks between 0.5 and 1.5 seconds wide. This will be used to compute the scales for the analysis.

```
peak_width <- c(.5, 1.5)
peak_info <- as_tibble(peaksWithCentWave(y, t,
peakwidth = peak_width,
                                         verboseColumns = TRUE))
|>
    dplyr::select(-c("into", "sn", "egauss", "mu",
"sigma",
                "h", "f", "dppm", "lmin", "lmax")) |>
    dplyr::mutate(scale=as.integer(scale),
           scpos=as.integer(scpos),
           scmin=as.integer(scmin),
           scmax=as.integer(scmax))
print(peak_info[1,])
```

```
## # A tibble: 1 x 9
##      rt rtmin rtmax    intb   maxo scale scpos scmin
scmax
##   <dbl> <dbl> <dbl>   <dbl>  <dbl> <int> <int> <int>
<int>
## 1  7.99  6.97  8.83 202576. 214400     6   100    94
106
```

The output of the `xcms` peak-picking process generates basic parameters, and when `verboseColumns` is set to `TRUE`, it generates a few more. In the code above, I've kept for display the most useful for this example. The retention time (`rt`), the peak-start (`rtmin`), the peak-end (`rtmax`), the intensity of the apex measured from `0` (`maxo`), and the wavelet scale that was used for peak detection (`scale`). It also generates the index of the apex (`scpos`) as well as the index of the peak start (`scmin`) and peak end (`scmax`).

The apex index was found to be `100` using the `get_peak_apex()` function from Section 6.2.2. The retention time reported by `peaksWithCentWave()` is the maximum of the CWT coefficients, which includes the Gaussian smoothing associated with the Mexican hat at scale `6`.

```
which(near(max(wCoefs[,"6"]),wCoefs[,"6"]))
```

```
## [1] 100
```

The CWT method gives the same retention time for the peak as the SG-smoothed trace apex. However, there is a difference between the two algorithms for locating peak start and end. This will affect how much of the peak is integrated to calculate area and concentration.

```
p_compare <- p +
    ggtitle(label="Peak Start - Apex - End Times",
            subtitle = "derivative and centWave
algorithms") +
    coord_cartesian(xlim=c(6,11), ylim=c(0, 2.5e4)) +
    geom_vline(aes(xintercept = t[peak_start_index],
color='Derivative'),
               linetype= "dashed") +
    geom_vline(aes(xintercept = t[peak_end_index],
color='Derivative'),
               linetype= "dashed") +
    geom_vline(aes(xintercept = peak_info$rtmin[1],
color='centWave'),
               linetype= "dashed") +
    geom_vline(aes(xintercept = peak_info$rtmax[1],
color='centWave'),
               linetype= "dashed") +
    scale_color_manual(name='Algorithm: ',
                       breaks=c('Derivative',
'centWave'),
                       values=c(pal$green,
pal$darkorange)) +
    theme(legend.position = "bottom")


print(p_compare)
```

The approach in `xcms` is to start at the `scmin` index and then walk down the front and back of the peak until the threshold

is crossed to determine the start and end of the peak. From Figure 6.19, it would appear that `peaksWithCentWave()` produces start and end times that are more conservative than the derivative method. A significant part of the peak, which is statistically above the estimated noise, is ignored on the tail of the peak. As long as it's done consistently, I will show that this conservative approach can produce acceptable quantitative results.

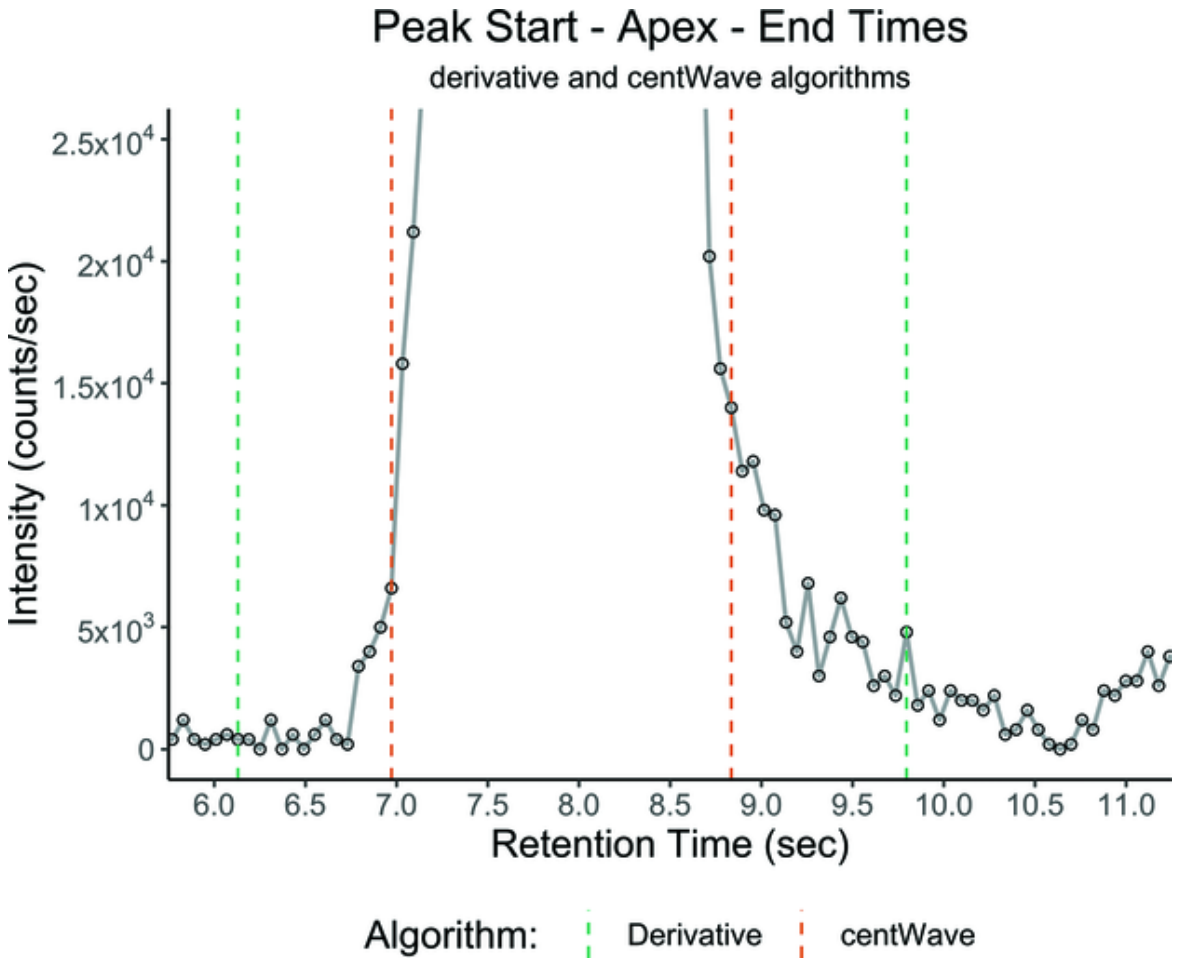


Figure 6.19 Comparing peak start and end times computed from derivatives and xcms.

6.4 Frequency Analysis

It should be clear that the main goal of smoothing and peak

detection, as described so far, is to approximate two primary functions: the function that generates a signal as a result of the presence of a chemical, i.e. the deterministic component and the function that generates a signal as a result of random processes which obscure or corrupt the deterministic signal, i.e. the random component. Function estimation has, so far, been described in both parametric and nonparametric terms. The random component has been ascribed to some function, like the Gaussian function, with parameters that need to be estimated to characterize the noise (like standard deviation). When using the SG filters, the deterministic component of the signal is approximated by local linear regressions of a selected polynomial function. There are no parameters ascribed to the specific polynomials representing the behavior of the system, but rather, the deterministic function is simply approximated by a collection of linear functions. In the wavelet analysis presented, a specific function related to an assumed underlying function for the deterministic component (again Gaussian) is used for the deterministic function approximation. In both cases, the deterministic component of the trace is approximated in order to obtain data that are less corrupted by noise (smoothed) to carry out calculations to obtain the first and second derivatives of the trace, which are exceptionally sensitive to noise.

In calculus and analysis, it is common to use an infinite series to represent a function and then truncate the series to make a problem tractable. When using a truncation approach to approximation, the rate of convergence of the series is important. A mathematical series that converges rapidly for the deterministic component of the data and slowly for the random component is a logical choice for using truncation to isolate chemical information from random noise.

In signal processing, the Fourier series [162, 181] is a natural selection because it converges very rapidly. The coefficients of the terms drop by n^{-3} for smooth functions with smooth first and second derivatives [163]. For functions

with rough first and second derivatives, the series converges much more slowly. That means that the deterministic component of chromatography can be represented by a small early subset of the series, while the random component is represented by later terms, which, when truncated, serve to reduce the contribution of random processes to the signal in a trace. By performing Fourier analysis on the trace, the frequency content of the deterministic component can be selected, and a function can be designed to eliminate corrupting noise.

Another convenient feature of the Fourier series is that it can be generated by performing the Fourier transform (FT), which converts data collected in the time domain (time is the x-axis) to the frequency domain (frequency is the x-axis). The time domain and the frequency domain are complementary, and one important relationship between the two is that convolution (moving average) in the time domain is equivalent to multiplication in the frequency domain.

Any function that lowers the high-frequency coefficients of the Fourier series of raw data will cause it to be smoother. Selecting the function and the domain in which it is applied (time domain or frequency domain) requires some thought. Ideally, a square function, which is `1` where frequencies contain chemical information and `0` everywhere else, could be multiplied by the frequency-domain version of the data and then converted back to the time domain. However, as I will show in the next section, a square in the frequency domain transforms to a time-domain function with ripples that can badly distort the data. The same is true for the simple moving average in time. The square used in the time domain contains ripples in the frequency domain that will attenuate deterministic signal frequencies and amplify random signal frequencies, again distorting the data in ways that could interfere with the analysis.

6.4.1 Fourier Analysis of Traces

Fourier analysis is one of many function approximations that use a *basis system* to approximate the data with an underlying function. A basis system is a way of combining (usually additive) functions to approximate a known or unknown function. The Fourier series uses a sum of *trigonometric functions* as the basis system and assumes that the data can be represented by a sum of sin and cos functions. There are many basis systems used in signal processing. Other common approaches include *splines* [182, 183], which are a piecewise polynomial basis system. Exponential and power series bases can be used, as well as polynomial bases. The *Taylor* and *Maclaurin* series taught in calculus are specialized methods for estimating polynomial expansions. Earlier in this chapter, I described the use of polynomials via the SG filters, as well as wavelets, of which I only described one basis system: the Maar functions. There are many other basis sets to choose from. For example, tree-based methods [184, 185] are a *step-function* basis system, which can be thought of as an order one B-spline with one interior knot [182].

There are two concepts that are critical to understanding the frequency analysis of data using the Fourier series and the FT. First, since the basis functions are periodic trigonometric functions, time-bound data like chromatography, and mass spectrometer data are treated *as if* they were periodic. A metaphor for this approach is to imagine the data printed on a sheet of paper that is rolled into a tube so that the last data point and the first data point connect. The metaphor is useful because the trigonometric functions are operations on a circle, and the Fourier series is defined on $[-\pi, \pi]$ with a period of 2π. This requires mapping the x-axis of instrument data onto a circle so finite data is processed as periodic data when using a Fourier series approximation of empirical data.

$$f(x) \approx \frac{a_0}{2} \sum_{n=1}^{\infty} (a_n \cos nx + b_n \sin nx) \tag{6.3}$$

In practice, this equation is represented as:

$$f(x) \approx \sum_{n=-\infty}^{\infty} \left(c_n e^{-inx}\right) \tag{6.4}$$

where

$$\begin{aligned} c_n &= \frac{1}{2}(a_n + ib_n), n = 1, 2, \ldots \\ c_n &= \frac{1}{2}(a_n - ib_n), n = -1, -2, \ldots \\ c_0 &= \frac{a_0}{2} \end{aligned} \tag{6.5}$$

All of the math for the FT is done by R, using the base `stats` package and the `fft()` function, which implements the *Fast Fourier Transform* (FFT) [186]. The FFT is an optimized *Discrete Fourier Transform* (DFT). Textbooks, such as Bracewell's *The Fourier Transform and Its Applications* [162], usually spend a significant amount of time teaching the continuous version of the FT, which is used to find analytic solutions to transform an equation from one domain to another. When FT is performed on discrete sampled data rather than analytical equations, the DFT approach is typically used. The DFT shown in Eqs. (6.3–6.5) is meant to point out that the DFT (and FFT) produce coefficients that are complex numbers.

The real part of the coefficient represents the amplitude of the sinusoidal function, and the imaginary part represents its phase (the angle of the sine/cosine function), which can be thought of as a time shift. This will become important when working with the output of the DFT.

Since the frequencies computed from a time domain dataset can only represent 1/2 the total number of data points and the finite dataset is converted into an infinite set by treating it as a repeating loop, another consequence of performing an FT is that the output looks like it is duplicated. This is, in fact, the case. The output runs from frequency 0 to 1/2 the sampling frequency, and then the rest of the array is filled with a mirror image of the first half.

Because the results of the DFT are complex numbers, to get a sense of how the transform works, I have to decide what I want to plot. The FT gives the amplitude of the sinusoidal function as the real part of the complex number and its phase as the imaginary part. While it is instructive to plot both the amplitude and phase, I am interested in using the DFT to separate signal from noise. My main interest is *how much power is present at each frequency component of the trace*. In signal processing, this is known as *power density* estimation, and the DFT is one of the most common ways to perform this calculation [181]. The power or energy is the squared value of the magnitude of the coefficient c_n. The magnitude, or *modulus* of an FT coefficient, can be computed either using the `abs()` function or the base package `Mod()` function. In most R programs, you will see magnitudes computed using the `abs()` function, which is equivalent since both compute $\sqrt{Re^2 + Im^2}$. Base R provides complex number functions like `Re()` and `Im()`, which access the real and imaginary components of a complex number.

```
# create a complex number in R
c <- complex(real=0.5, imaginary = 0.5)
c
```

```
## [1] 0.5+0.5i
```

```
# get the magnitude (length of the vector)
print(sprintf("Magnitude %0.5f, Absolute Value %0.5f",
Mod(c), abs(c)))
```

```
## [1] "Magnitude 0.70711, Absolute Value 0.70711"
```

The FFT of the trace in Figure 6.1 is computed using the `fft()` function from the `stats` package and then converted to a magnitude using `abs()`.

```
ft_y <- abs(fft(y))
rad_x <- seq(0, 2*pi, by=(2*pi/length(y)))[1:length(y)]
```

And then plotted (Figure 6.20) in the natural units of the circle (radians) over which sin and cos are defined:

```
p_fft <- ggplot() +
    scale_y_continuous(labels = inten_label) +
    geom_line(aes(x=rad_x, y=ft_y), linewidth=0.75,
color=pal$blue) +
    xlab(TeX(r"($\textrm{Radians\;}(0-2\pi)$)")) +
    ylab("Magnitude") +
    theme_classic() +
    theme(plot.title = element_text(hjust = 0.5,
size=16)) +
    theme(plot.subtitle = element_text(hjust = 0.5)) +
    theme(
        axis.text=element_text(size=11),
        axis.title=element_text(size = 14),
        legend.text = element_text(size = 11),
        legend.title = element_text(size = 14)
    ) +
    ggtitle(label = "Raw FFT of Sample 11 Quant
Chromatogram")
print(p_fft)
```

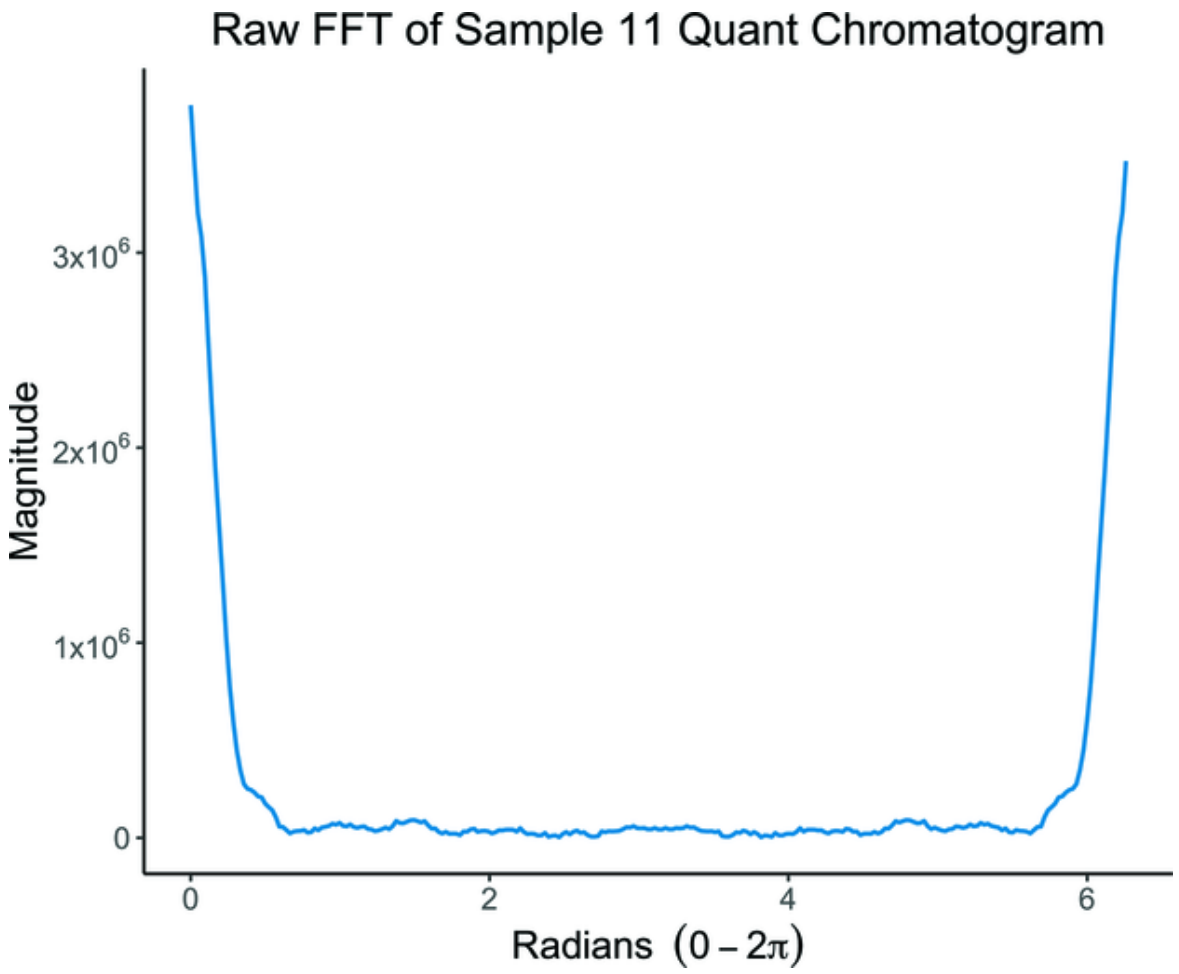


Figure 6.20 DFT of the sample 11 quant trace obtained using the FFT algorithm.

Figure 6.20 makes it easy to see how the left side wraps around to match the right side of the plot. If I change the x-axis from radians to frequency, which is a more natural way to use the data, the entire right half of the plot is removed because of the Nyquist criteria, which limits the highest frequency that can be determined to half the sampling frequency. In plots and calculations done from this point on, I will truncate the x-axis to the Nyquist limit. The data in Figure 6.20 can be plotted using frequency units and obeying the Nyquist criteria.

```
sample_length <- length(y)
nyquist_length <- as.integer(floor(sample_length / 2)
+1 )
nyquist_limit <- sampling_frequency / 2

freq_x <- seq(0, nyquist_limit,
by=nyquist_limit/nyquist_length)[1:(nyquist_length)]
freq_y <- ft_y[1:(nyquist_length)]
```

Now that the frequency spectrum component of the FFT has been selected, it can be plotted (Figure 6.21). I could use power (magnitude squared), but since the chromatogram is not an electrical transmission, converting to signal power adds little extra value. The objective is to find the frequencies that represent the smooth signal and truncate the series at the point where the frequencies represent noise.

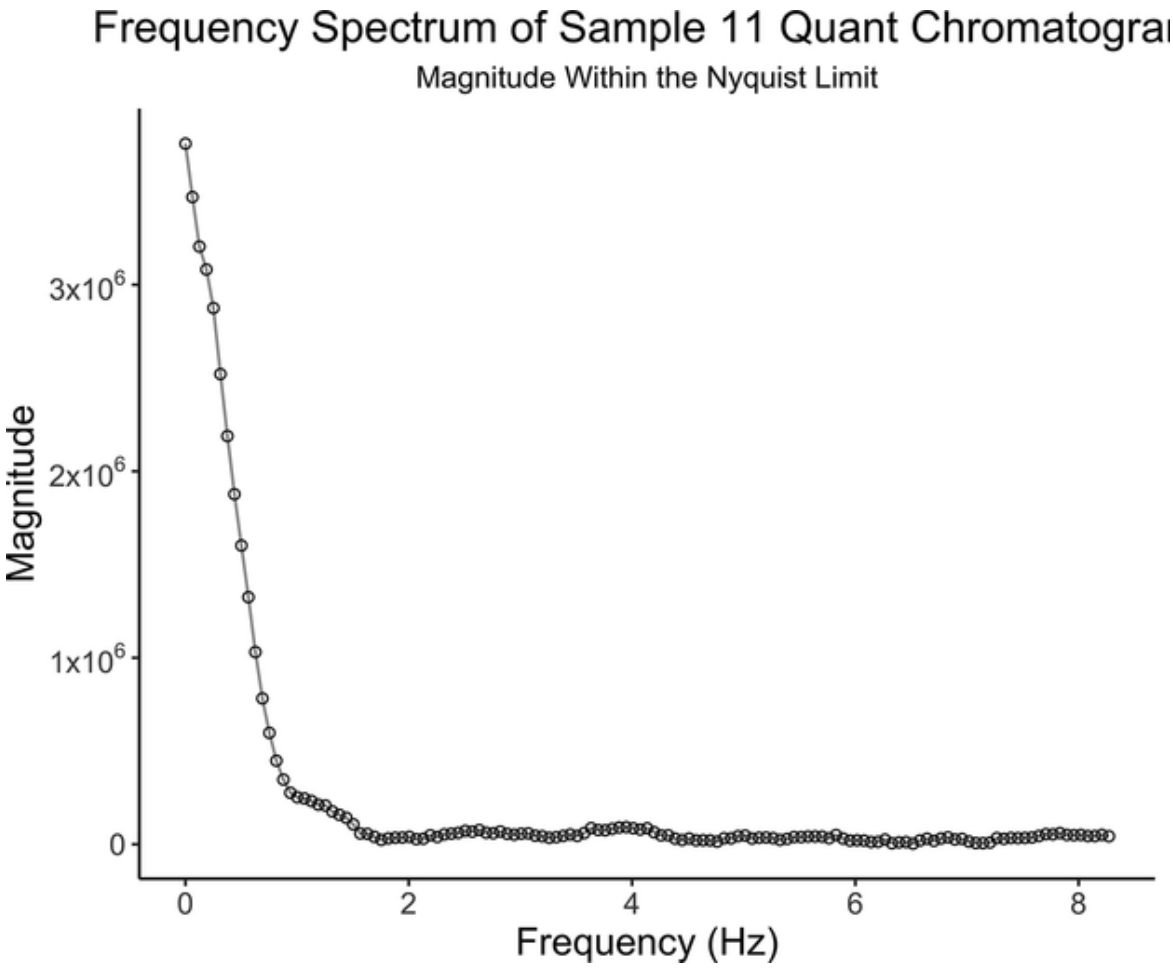


Figure 6.21 The magnitude spectrum of the sample 11 quant trace in the Nyquist limit.

```
p_freq <- data.frame(x=freq_x, y=freq_y) |>
    ggplot(aes(x=x, y=y)) +
        scale_y_continuous(labels = inten_label) +
        geom_line(linewidth=0.5, color=pal$gray) +
        geom_point(shape=1) +
        xlab("Frequency (Hz)") +
        ylab("Magnitude") +
        theme_classic() +
        theme(plot.title = element_text(hjust = 0.5,
size=16)) +
        theme(plot.subtitle = element_text(hjust =
0.5)) +
        theme(
            axis.text=element_text(size=11),
```

```
            axis.title=element_text(size = 14),
            legend.text = element_text(size = 11),
            legend.title = element_text(size = 14)
        ) +
        ggtitle(label = "Frequency Spectrum of Sample
11 Quant Chromatogram",
                subtitle = "Magnitude Within the
Nyquist Limit")
print(p_freq)
```

In both Figures 6.20 and 6.21, the rapid drop off of the coefficients is clear. In the chromatogram, most of the signal strength is below 2 Hz. This is in the range of what was calculated in Section 6.2.4. The FWHM of the main peak is 0.9 seconds, which you can imagine as roughly having the shape of a sine wave with a frequency of `1/t` or 1.1 Hz. The lower frequencies in Figure 6.21 represent the base of the peak all the way down to the roughly flat part of the chromatogram, which you can think of as having a frequency close to `0`, which is a flat line in the time domain. The higher frequencies in the signal represent the rougher parts of the chromatogram and, thus, the random noise components of the trace. Removing these coefficients (by setting them to zero) will leave only the smooth function, which carries the chemical information in the trace.

It's also apparent from the FFT plot that truncating the series too early will remove some of the deterministic parts of the data. When this occurs, it's commonly referred to as *over-smoothing*, and leads to distortion of the peak shape. In addition to not wanting to attenuate any of the deterministic components of the trace, smoothing should not accidentally amplify any of the random components. When that occurs, the smoothing process can generate *filter artifacts* in the chromatogram, which can show up as ripples before and after a real peak or as a peak in the noise region where no peak existed before applying the filter. Since both distortions are undesirable, in the next section, I will look at how to see

what a smoothing process does to data in the frequency domain and show a way to select an optimum digital filter.

6.4.2 Analyzing Digital Filters

In Section 6.3.1, I used the SG filter in order to combine smoothing with numerical differentiation. The SG filters are normally applied like *moving average* (MA) smoothers. Mathematically, the filter coefficients are *convolved* with the raw data. A key idea in Fourier analysis is the *convolution theorem* [162], which simply states that convolution in the time domain is equivalent to multiplication in the frequency domain. That means that in moving average-type filters, the frequency-domain coefficients of the filter are multiplied by the frequency-domain coefficients of the data. For smoothing to occur, multiplying the data with the filter has to result in high-frequency coefficients being reduced.

The easiest smoothing process to understand is a simple moving average, sometimes called a *box car* filter. The box car filter gets its name from its square shape in the time domain. While this is simple to understand in the time domain, the effect on real data is not so simple when you see what happens to a simple time domain square in the frequency domain. The `gsignal` package includes functions to build a wide range of digital filters, characterize them, and apply them to data.

The coefficients of a filter, which performs a moving average of a particular length, can be computed using the `boxcar()` function. The function will generate a filter where all the coefficients are `1` for whatever length you choose. In order for the filter to produce smooth data of the same amplitude as the raw data, the filter coefficients have to be *normalized* so that they sum to `1`. The normalization causes the filter to perform the averaging in the name “moving average.” Some filter design functions will produce normalized coefficients,

but it's always a good idea to ensure filter coefficients sum to 1.

```
bc <- boxcar(11)
bc <- bc / sum(bc)
print(bc)
```

```
##  [1] 0.09090909 0.09090909 0.09090909 0.09090909
0.09090909 0.09090909
##  [7] 0.09090909 0.09090909 0.09090909 0.09090909
0.09090909
```

Digital filters don't get simpler than that! A straight line that is convolved with the data. One thing to remember with time-domain convolution is that the filter values are multiplied by the raw data values starting at the beginning of the data and then shifted by one position in the raw data until the last filter coefficient is aligned with the last data point in the raw data. At each step, the sum of the multiplied values is substituted for the central value of the filter. For example, with a nine-point Boxcar filter, the center point is the fifth coefficient. The first four raw data points are consumed, but there is no value for them from this filter. In normal time-domain filtering, you always lose half the width of the filter at the beginning and end of the raw data trace. There are ways to replace the missing data so that the smoothed trace is the same length as the raw trace. The `gsignal` function `conv()` solves this problem by extending the length of the overall result vector. There are other approaches, including simply using the unsmoothed data to replace what was left out by the convolution, or you can smooth it with shorter filters, which would leave only the start and end data points unsmoothed.

The quadratic SG filter of length 11 was used in Section 6.3.1. It is worth comparing the Boxcar filter to the SG filter to see what the SG filter is doing differently. The `gsignal` package has a function `sgolay()`, which will compute the SG

filter coefficients of different polynomial orders, lengths, and different orders of differentiation. The `sgolay()` function creates a matrix of coefficients that can be used to filter the beginning, middle, and end of the raw data with shorter filters. Since I am using `conv()`, I only need the coefficients in the center of the matrix. For SG filters with length 11, this is the middle row (row 6). The filter used earlier in this chapter was a quadratic or second-order polynomial. The `sgolay()` function produces a coefficient matrix, and the default output is the smoothing filter with no differentiation.

```
sg <- sgolay(p=2, n=sg_length)[ceiling(sg_length/2),]
sg <- sg/sum(sg)
print(sg)
```

```
##  [1] -0.08391608  0.02097902  0.10256410  0.16083916
0.19580420  0.20745921
##  [7]  0.19580420  0.16083916  0.10256410  0.02097902
-0.08391608
```

To see the filter shapes, I will create a plot (Figure 6.22) of the coefficients:

```
bc_x <- seq(0,length(bc)-1)/(floor(length(bc))-1)
sg_x <- seq(0,length(sg)-1)/(floor(length(sg))-1)

p_filter_coeff <- ggplot() +
        coord_cartesian(ylim=c(-0.1, 0.3)) +
        geom_line(aes(x=bc_x, y=bc, color="Box Car"),
            linewidth=0.5) +
        geom_point(aes(x=bc_x, y=bc), shape=1) +
        geom_line(aes(x=sg_x, y=sg, color="SG
Quadratic"),
            linewidth=0.5) +
        geom_point(aes(x=sg_x, y=sg), shape=1) +
        ggtitle(label = "Filter Coefficients") +
        xlab("Filter Index (0-1)") +
        ylab("Magnitude") +
        scale_color_manual(name='Filter: ',
                           breaks=c('Box Car', 'SG
Quadratic'),
                           values=c(pal$green,
pal$darkorange)) +
        theme_classic() +
        theme(plot.title = element_text(hjust = 0.5,
size=16)) +
        theme(plot.subtitle = element_text(hjust =
0.5)) +
        theme(
            axis.text=element_text(size=11),
            axis.title=element_text(size = 14),
            legend.text = element_text(size = 11),
            legend.title = element_text(size = 14)
        ) +
        theme(
            legend.position.inside = c(.99, 1.0),
            legend.justification = c("right", "top"),
            legend.box.just = "right",
            legend.margin = margin(4, 4, 4, 4)
            )
print(p_filter_coeff)
```

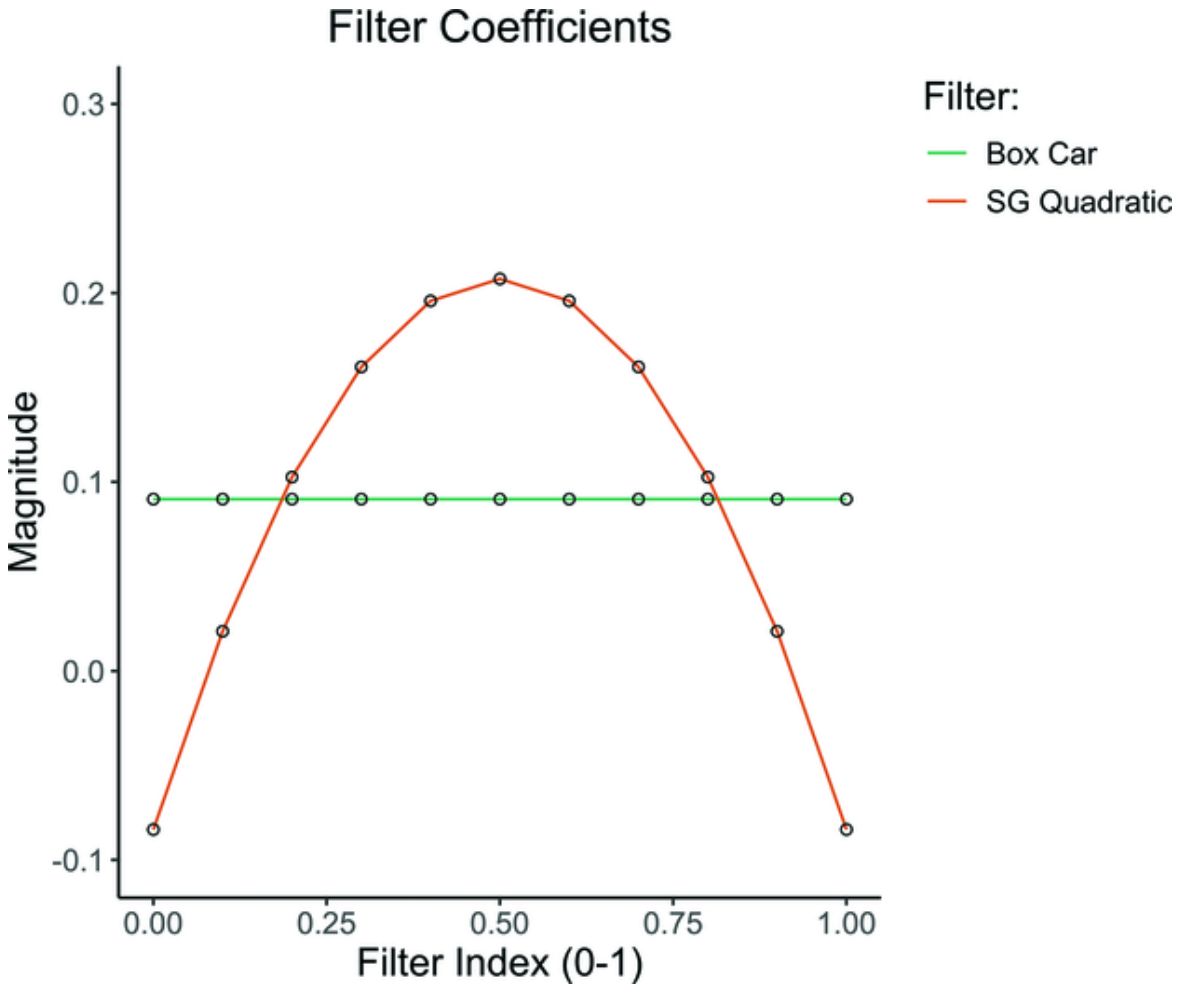


Figure 6.22 Filter coefficients for Boxcar and SG filters. Both are length 9, and the SG filter is a quadratic (order 2).

As expected, Figure 6.22 shows that the SG filter has a parabolic shape consistent with its quadratic form. Now, I can compare how these two filters perform on the data.

```
smooth_y_bc <- gsignal::conv(y, bc, shape="same")
smooth_y_sg <- gsignal::conv(y, sg, shape="same")
```

The `conv()` function returns the full convolution by default (`shape="full"`), which has a length of `length(y) + length(bc) -1`. To get a filtered vector the same length as the raw data vector, the raw vector has to be the first argument in the

function, and the shape has to be set to “same.” This will take the central part of the convolution of the same length as the original raw data.

```
p_sg_bc_smooth <- ggplot() +
        scale_y_continuous(labels = inten_label) +
        geom_line(aes(x=t, y=smooth_y_sg, color='SG
Quadratic'),
            linewidth=0.5) +
        geom_line(aes(x=t, y=smooth_y_bc,
color='Boxcar'),
            linewidth=0.5) +
        geom_point(aes(x=t, y=y), shape=1) +
        scale_color_manual(
                name='Filter: ',
                breaks=c('Boxcar', 'SG Quadratic'),
                values=c(pal$blue, pal$red)) +
        ggtitle(label = "Smoothing with Boxcar and
Savitzky-Golay",
                subtitle = sprintf("Length %d, SG
(order=2)", sg_length)) +
        xlab("Retention Time (s)") +
        ylab("Intensity (counts/s)") +
        theme_classic() +
        theme(plot.title = element_text(hjust = 0.5,
size=16)) +
        theme(plot.subtitle = element_text(hjust =
0.5)) +
        theme(
            axis.text=element_text(size=11),
            axis.title=element_text(size = 14),
            legend.text = element_text(size = 11),
            legend.title = element_text(size = 14)
        ) +
        theme(
            legend.position.inside = c(.95, .95),
            legend.justification = c("right", "top"),
            legend.box.just = "right",
            legend.margin = margin(6, 6, 6, 6)
        )

print(p_sg_bc_smooth)
```

This code generates Figure 6.23, where it can be seen that the Boxcar filter distorts the data at both the apex and slightly at the base of the peak. Based on the discussion earlier, attenuation of frequencies that are part of the signal will cause distortion. To see these effects directly, comparing the frequency response of these two filters gives a clue to what is happening, as well as how to do an even better job.

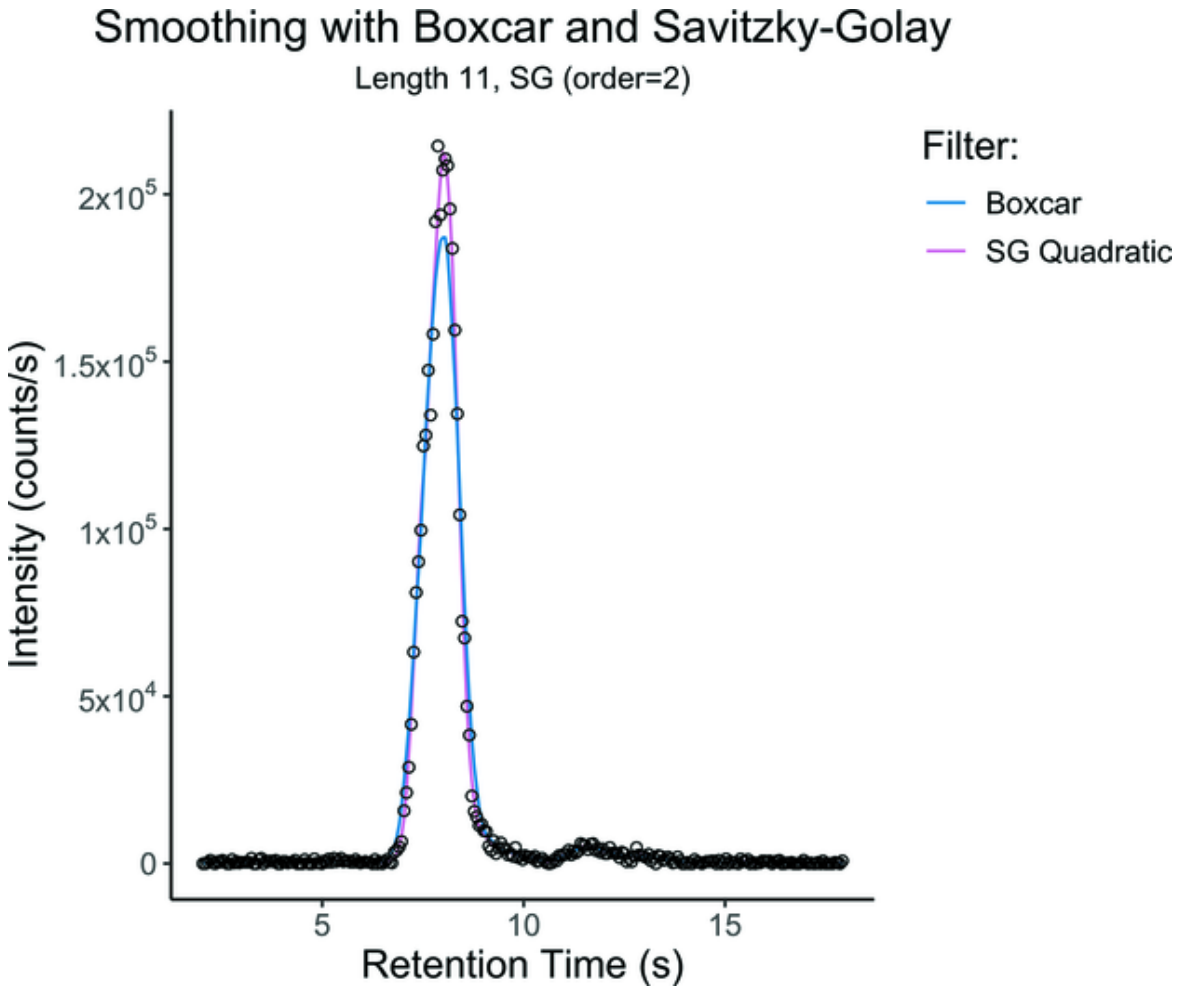


Figure 6.23 Raw data smoothed with Boxcar and Savitzky–Golay filters. Note the distortion in the peak apex for the boxcar compared to the SG smooth.

The `gsignal` package has a function called `freqz()`, which can be used to visualize the frequency content of signals and filters.

```
bc_fr <- freqz(bc, fs=sampling_frequency)
sg_fr <- freqz(sg, fs=sampling_frequency)
```

Since `freqz()` uses the DFT to compute the coefficients, they are complex, and the lowest frequency (0) is normalized to 1.0:

```
bc_fr$h[1]
```

```
## [1] 1+0i
```

And the magnitude is calculated with `abs()`:

```
abs(bc_fr$h[1])
```

```
## [1] 1
```

This is how I get the y-values for the frequency response plot.

```
p_freqz <- ggplot() +
        geom_line(aes(x=sg_fr$w, y=abs(sg_fr$h),
color='SG Quadratic'),
            linewidth=0.5) +
        geom_line(aes(x=bc_fr$w, y=abs(bc_fr$h),
color='Boxcar'),
            linewidth=0.5) +
        geom_line(aes(x=freq_x, y=freq_y/max(freq_y),
color='Raw Trace'),
            linewidth=0.5) +

        scale_color_manual(name='Signal: ',
                breaks=c('Boxcar', 'SG Quadratic',
'Raw Trace'),
                values=c(pal$blue, pal$red,
pal$black)) +
        ggtitle(label = "Frequency Response of Boxcar
and Savitzky-Golay",
            subtitle = "Overlayed on the Frequency
Content of the Raw Trace") +
        xlab("Frequency (Hz)") +
        ylab("Relative Magnitude") +
        theme_classic() +
        theme(plot.title = element_text(hjust = 0.5,
size=16)) +
        theme(plot.subtitle = element_text(hjust =
0.5)) +
        theme(
            axis.text=element_text(size=11),
            axis.title=element_text(size = 14),
            legend.text = element_text(size = 11),
            legend.title = element_text(size = 14)
        ) +
        theme(
            legend.position.inside = c(.95, .95),
            legend.justification = c("right", "top"),
            legend.box.just = "right",
            legend.margin = margin(6, 6, 6, 6)
        )
```

```
print(p_freqz)
```

This code produces Figure 6.24, which shows that the length 9 Boxcar filter reaches a relative magnitude of zero (maximum attenuation) at a much lower frequency than the SG filter. Worse, the Boxcar starts attenuating at a very low frequency, which means terms in the Fourier series needed to approximate the chromatographic peak are being lowered in a more severe way than the SG case for the same filter width. Both filters have two undesirable properties: they “roll off” in a sloping manner rather than in a sharp cut at the end of the deterministic component of the signal, and they have ripples, in which higher frequencies are not strongly attenuated which means that a new ripple will be created in the data maximized at the top of each hump in the filter frequency response.

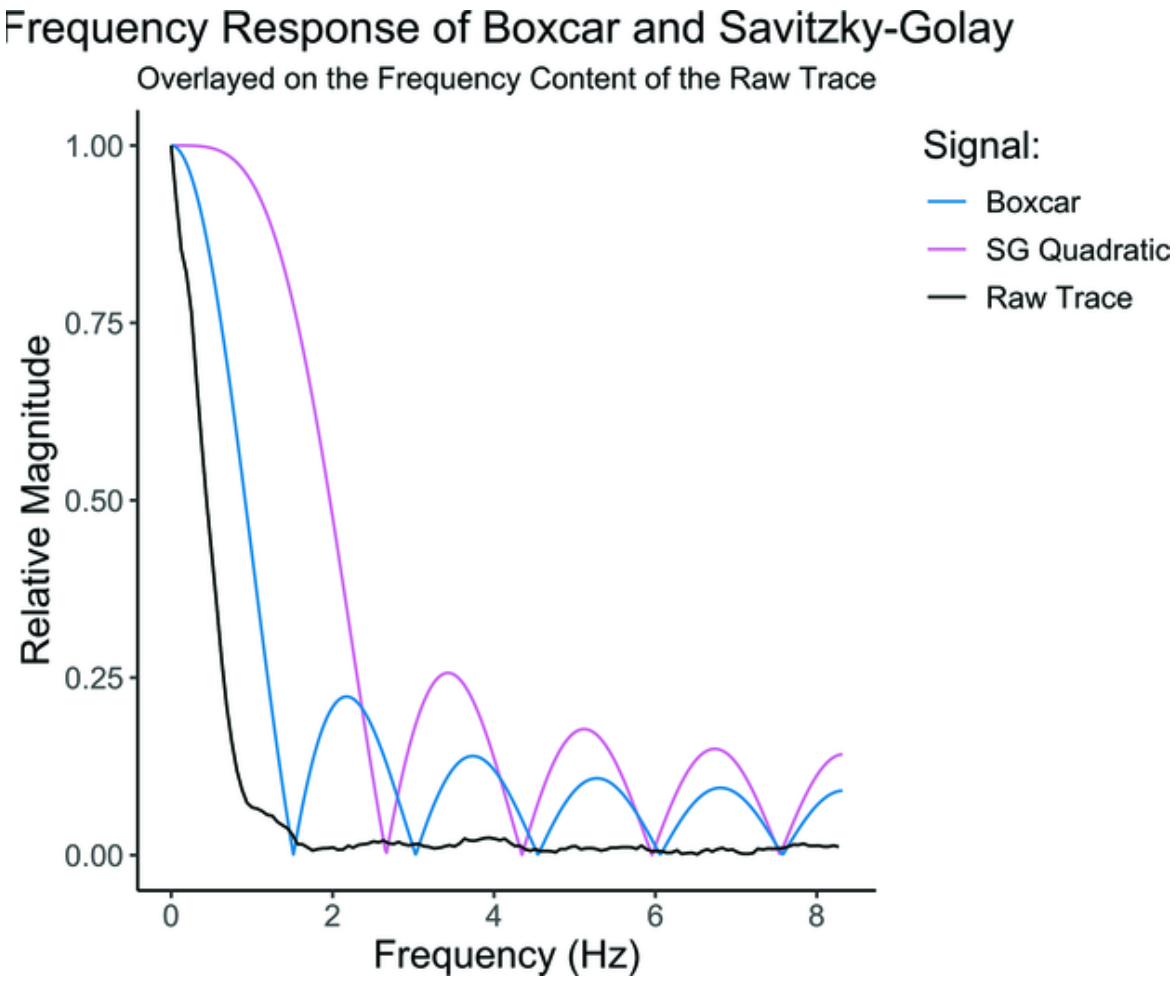


Figure 6.24 Frequency response of the Boxcar and Savitzky–Golay filters showing the difference in attenuation in the low-frequency range. Excess attenuation for the boxcar causes peak distortion compared to the SG filter.

The distortion and ripple effects of both Boxcar and SG are well-known problems, leading some to suggest that we should stop using SG filters altogether [187]. However, for a simple, short filter that can be easily used, especially in combination with calculating derivatives, the SG piecewise polynomial function approximation works surprisingly well.

6.4.3 Optimal Filters

As discussed at the start of Section 6.4, it is possible to use the rapid convergence of the Fourier series for smooth functions to design an optimal filter. In this section, I show one way to design a good moving average type filter based on the frequency characteristics of the data with a better-behaved frequency response using the functions in the `gsignal` package.

The approach I will describe has two steps. First, determine a cutoff frequency that defines a boundary between the deterministic component of the signal and the random component. Second, select a windowing function that minimizes artifacts in the time domain and has a reasonable length.

Felinger [188] suggests using an approach developed by Lanczos [189] to empirically determine the optimum cutoff frequency based on the variance of the Fourier coefficients. This is based on the convergence rate of the Fourier series for smooth functions. The high-frequency coefficients are low and effectively constant until the deterministic frequency components appear at lower frequencies. Lanczos suggested computing the variance of the coefficients starting from the end of the frequency spectrum and iteratively adding more elements to the array to produce a new variance number until reaching the start of the array.

The function that performs the variance calculation of cutoff frequency is shown below. Lanczos points out that the exact cutoff value is not critical. There are at least two reasons for this. First, the noise in the data itself could make the cutoff different by a few coefficients, and second, a windowing function will ultimately be applied to the square cutoff which will determine where the filter starts to attenuate the signal. In the function `get_cutoff()`, I use a robust threshold that computes the median σ^2 value as the expected value of the

variance and then uses the median absolute deviation (MAD) `mad()` as the upper and lower range of the variance. The cutoff frequency is the frequency value where the variance value drops below the upper MAD value of the median. Below this threshold, the DFT coefficients are considered noise, which can be attenuated without distorting the chromatographic peak shape.

The `get_ft_variance()` function calculates the backward calculated variance values of the FT coefficients using the `var()` function.

```
get_ft_variance <- function(y_ft) {
    s2_list <- NULL
    y_length <- length(y_ft)
    for(i in y_length:2) {
        s2 <- var(y_ft[y_length:(i-1)])
        s2_list <- append(s2_list, s2)
    }
    rev(s2_list)
}
```

The `get_cutoff()` function calculates the threshold variance starting at the lowest frequency and loops toward the highest frequency, stopping when the variance is lower than the upper MAD bound on the median value. It returns both the cutoff frequency and the threshold value.

```
get_cutoff <- function(freq_x, freq_y) {
    s2_list <- get_ft_variance(freq_y)

    freq_thres <- median(s2_list) + stats::mad(s2_list)

    for(i in 1:length(s2_list)) {
        if(s2_list[i] < freq_thres) {
            freq_cutoff_index <- i + 1
            break
        }
    }

    c(freq_x[freq_cutoff_index], freq_thres)
}
```

The cutoff frequency and the threshold can then be calculated for use in the filter design.

```
cutoff <- get_cutoff(freq_x, freq_y)

cutoff_freq <- cutoff[1]
threshold <- cutoff[2]
```

The method can be visualized by plotting the values of σ^2 with the threshold and the cutoff. First, get the variance list:

```
s2_list <- get_ft_variance(freq_y)
```

Then cutoff can be visualized using `ggplot()`:

```
p_ft_variance <- ggplot() +
        coord_cartesian(ylim=c(0, 7e9)) +
        scale_y_continuous(labels = inten_label) +
        geom_line(aes(x=freq_x[1:(length(freq_x)-1)],
y=s2_list,
            color='Variance'),
            linewidth=0.5, key_glyph = draw_key_path) +
        geom_hline(aes(yintercept = threshold,
color='Threshold'),
            linewidth=0.5, key_glyph = draw_key_path) +
        geom_vline(aes(xintercept=cutoff_freq,
color='Cut Off'),
            linewidth=0.5, key_glyph = draw_key_path) +

        scale_color_manual(name='Signal: ',
                  breaks=c('Variance', 'Cut Off',
'Threshold'),
                  values=c(pal$black, pal$darkorange,
pal$blue)) +
        ggtitle(label = "Backward Calculated FT
Variance",
                subtitle=sprintf("Cutoff Frequency:
%0.2f Hz", cutoff_freq)) +
        xlab("Frequency (Hz)") +
        ylab(TeX(r"($\sigma^2$)")) +
        theme_classic() +
        theme(plot.title = element_text(hjust = 0.5,
size=16)) +
        theme(plot.subtitle = element_text(hjust =
0.5)) +
        theme(
            axis.text=element_text(size=11),
            axis.title=element_text(size = 14),
            legend.text = element_text(size = 11),
            legend.title = element_text(size = 14)
        ) +
        theme(legend.position = "bottom")

print(p_ft_variance)
```

This code produces Figure 6.25 which shows that the median value for a threshold does a reasonable job of finding the level below which coefficients are noise. The cutoff frequency of 1.57 Hz also seems sensible based on the data. If I know the frequency cutoff for real signal, you could ask, “why not just cut the coefficients off there, and transform back to the time domain?” The answer is in another fundamental identity in Fourier analysis. A square in the frequency domain is the sin^2 or *sinc* function. Since I showed the Mexican hat and the other filter functions earlier, I will show the FT that is obtained from the square.

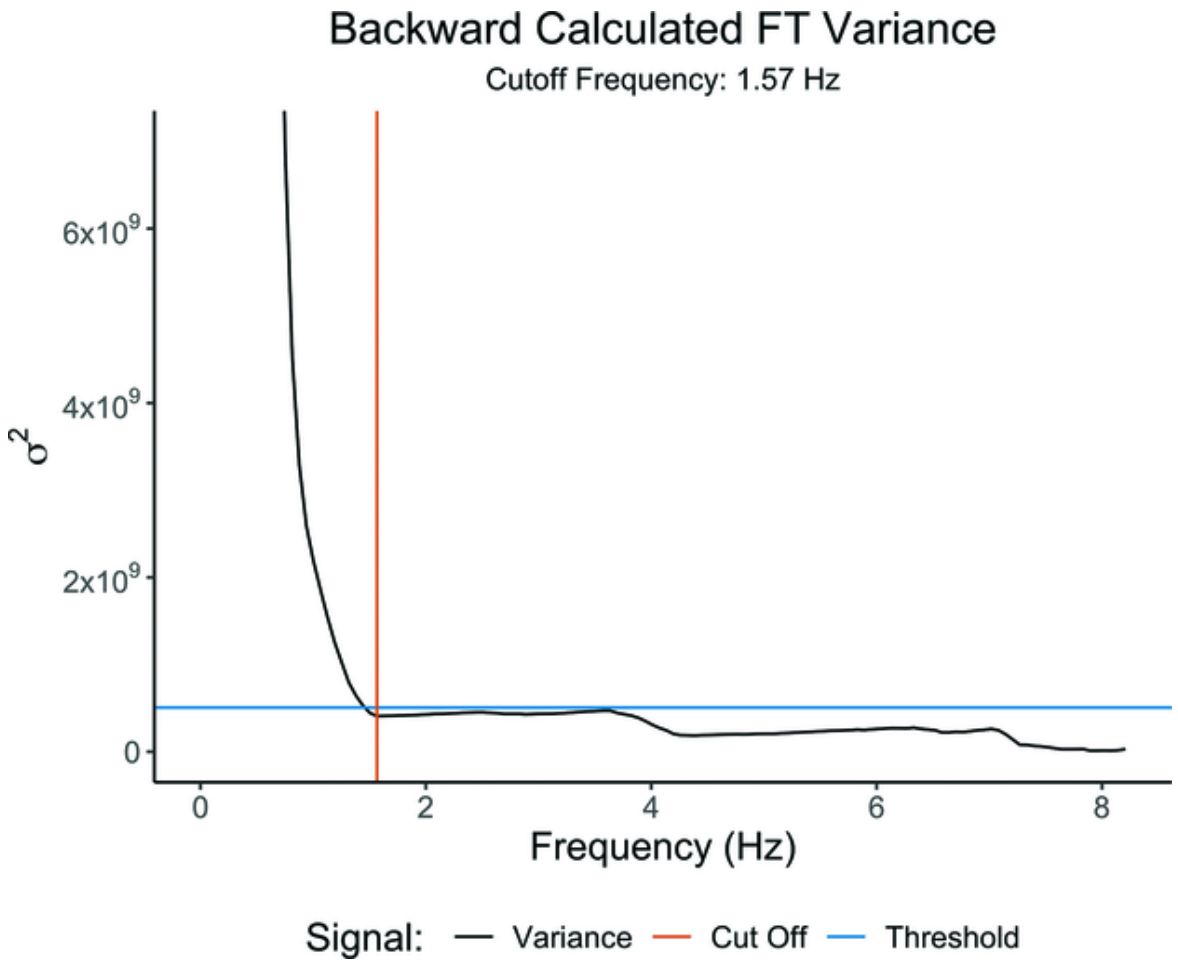


Figure 6.25 Backward calculated sum of the variance of the FT coefficients showing the median value as the threshold and the calculated cutoff frequency.

```
# calculate the index of the region of the full DFT
that will be set to zero
low_pass_begin <- which(freq_x==cutoff_freq)
low_pass_end <- length(ft_y)-low_pass_begin + 2
```

The central part of Figure 6.20 begins at `low_pass_begin` and ends at `low_pass_end`. The function that will be transformed has a value of `1`, and then a central region with a value of `0`, and ends back at `1`. This is a square function centered at zero.

```
h_freq<-rep(1,265)
h_freq[low_pass_begin:low_pass_end]<-0
cutoff_y <- fft(h_freq, inverse = TRUE)
```

```
cutoff_y <- c(cutoff_y[134:265],cutoff_y[1:133])
range_x <- seq(-0.5, 0.5, by=(1/length(cutoff_y)))
[1:length(cutoff_y)]

p_sinc <- ggplot() +
    geom_line(aes(x=range_x,
y=Re(cutoff_y)/length(cutoff_y)),
              linewidth=0.75, color=pal$lightblue) +
    xlab("Function Range") +
    ylab("Amplitude") +
    theme_classic() +
    theme(plot.title = element_text(hjust = 0.5,
size=16)) +
    theme(plot.subtitle = element_text(hjust = 0.5)) +
    theme(
        axis.text=element_text(size=11),
        axis.title=element_text(size = 14),
        legend.text = element_text(size = 11),
        legend.title = element_text(size = 14)
    ) +
    ggtitle(label = "Sinc Function Filter from Simple
Frequency Cutoff")
print(p_sinc)
```

This code produces Figure 6.26. It's not hard to imagine what will happen to the raw data if it's convolved with a filter of this shape. To see the effect, I can set the raw data frequency coefficients in the cutoff region to zero, and take the *inverse* FT.

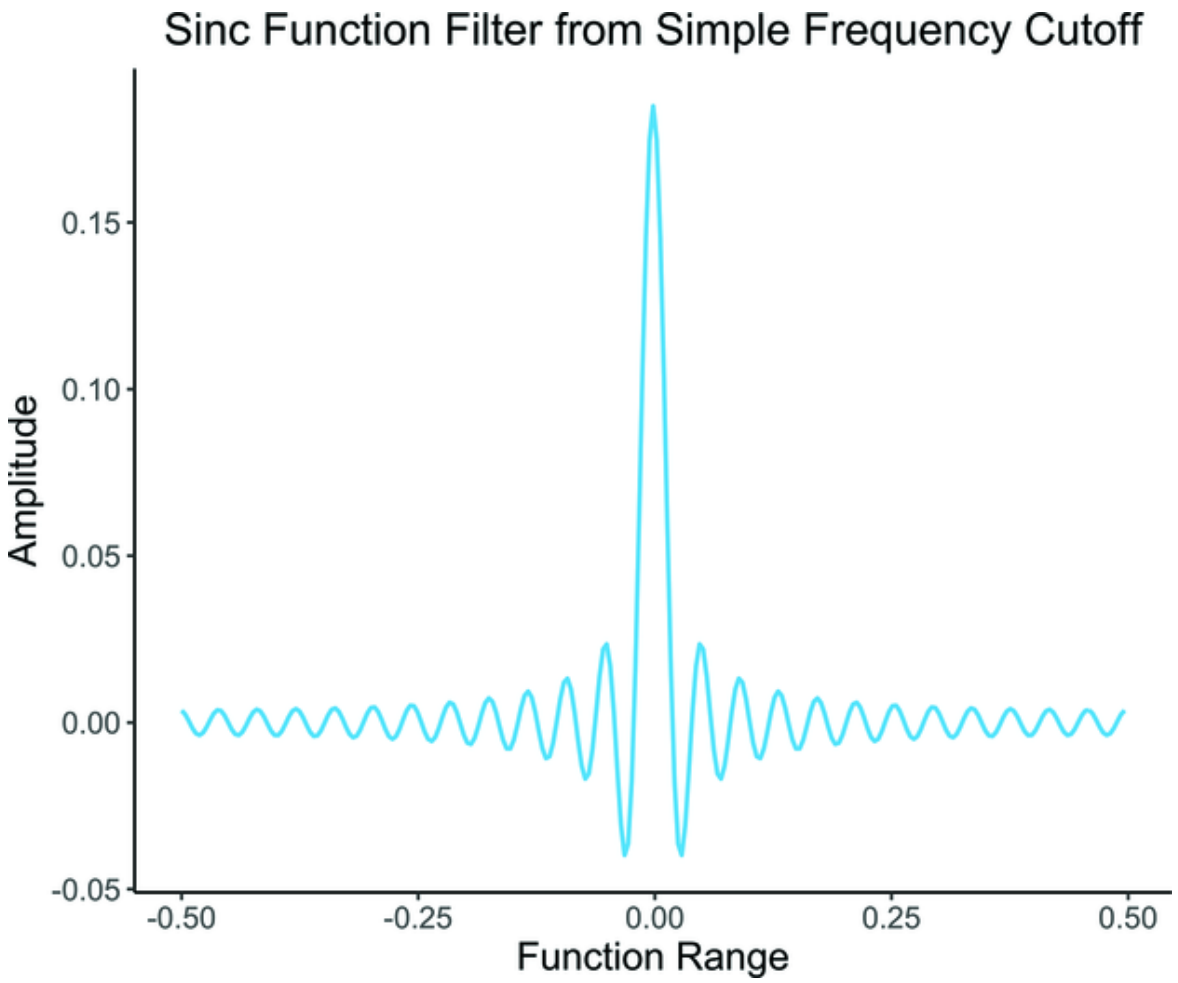


Figure 6.26 The sinc function is created in the time domain by a simple cutoff in the frequency domain.

```
trunc_ft_y<-fft(y)
trunc_ft_y[low_pass_begin:low_pass_end] <- 0
trunc_y <- fft(trunc_ft_y, inverse = TRUE)
```

And then plot (Figure 6.27) the time-domain version of the data:

```
p_sinc_smooth <- ggplot() +
        coord_cartesian(ylim=c(-10,5000)) +
        scale_y_continuous(labels = inten_label) +
        geom_line(aes(x=t,
y=abs(trunc_y)/sample_length, color='Sinc Filter'),
                  linewidth=0.75) +
        geom_point(aes(x=t, y=y, color='Raw Data'),
shape=1, size=1) +
        scale_color_manual(name='',
                breaks=c('Sinc Filter', 'Raw Data'),
                values=c(pal$lightblue, pal$black),
                guide = guide_legend(override.aes =
list(
                        linetype = c("solid",
"blank"),
                        shape = c(NA,1)
                        ))) +
        ggtitle(label="Smoothing with Frequency
Truncation",
                subtitle = "Sinc Function") +
        xlab("Retention Time (s)") +
        ylab("Intensity (counts/s)") +
        theme_classic() +
        theme(plot.title = element_text(hjust = 0.5,
size=16)) +
        theme(plot.subtitle = element_text(hjust =
0.5)) +
        theme(
            axis.text=element_text(size=11),
            axis.title=element_text(size = 14),
            legend.text = element_text(size = 11),
            legend.title = element_text(size = 14)
        ) +
        theme(
            legend.position.inside = c(.95, .95),
            legend.justification = c("right", "top"),
            legend.box.just = "right",
            legend.margin = margin(6, 6, 6, 6)
        )
```

```
print(p_sinc_smooth)
```

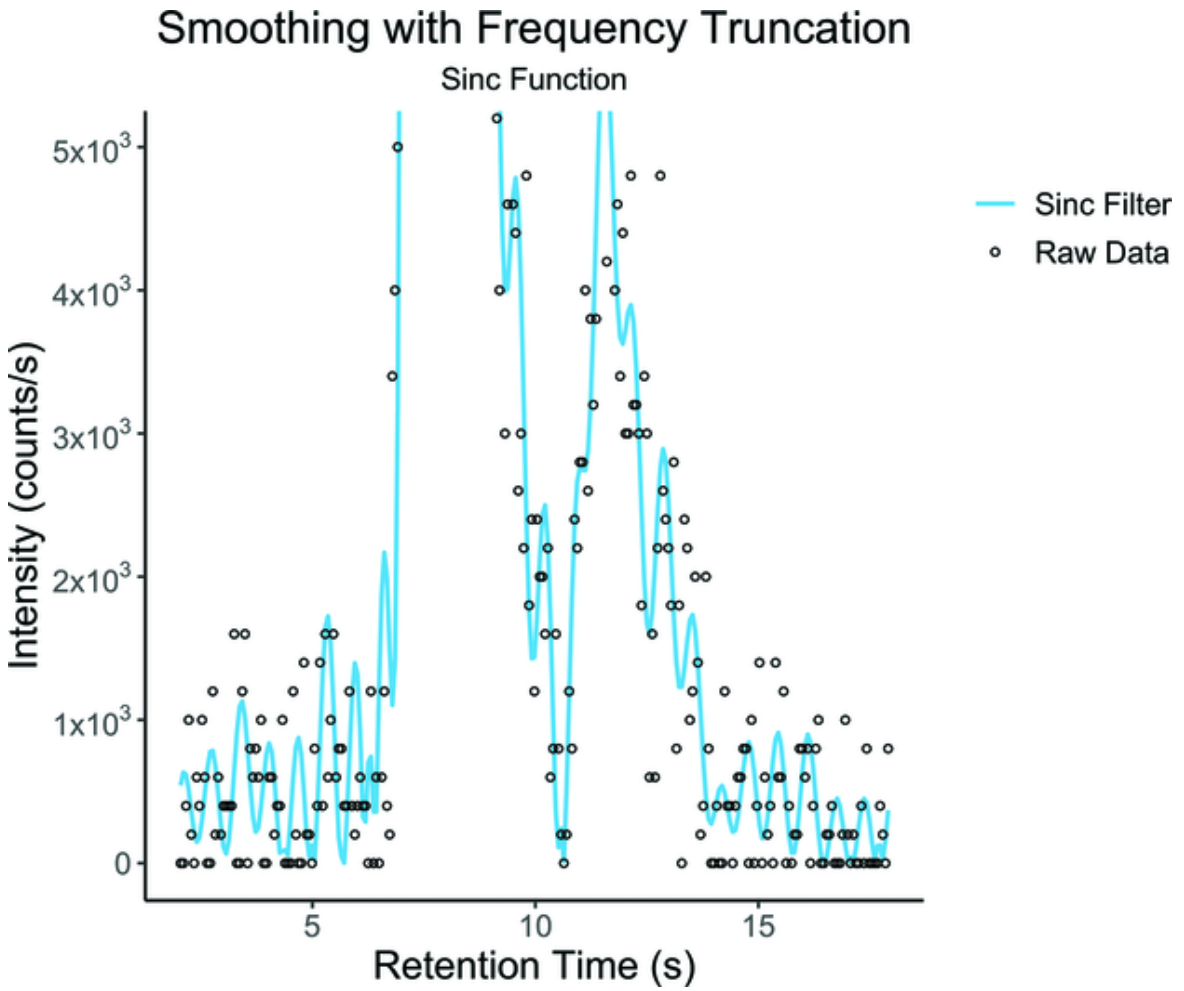


Figure 6.27 Raw data filtered by simple truncation of high-frequency coefficient results in artifacts created by the sinc function in the time domain.

As expected, there are ringing-like distortions throughout the smoothed data. In Figure 6.27, the ripples are not actually in the data but introduced by the sinc function. The solution is to *window* the sinc function to lower the intensity of the ripples. This is a very common task in signal processing, and the `gsignal` package provides a large number of window functions that can be used. Depending on your data, you may find one or more of the windowing functions work best for

your situation. In chromatography, it is desirable to have the filter concentrate most of the signal power in the main lobe where chemical information is present, rather than in the side lobes, which can be seen in Figure 6.24. The *Kaiser window* [190, 191] has exactly these properties. I will show how to design a Kaiser window using the `kaiserord()` function to produce the window parameters for use with `fir1()` function and the `kaiser()` function to design a convolution digital filter that has the desired cutoff and removes the ripple artifacts.

The first step is to determine the parameters of the Kaiser window based on the cutoff computed with `get_ft_variance()` combined with basic parameters, such as the magnitude of the upper and lower coefficients. Just like above, I'll use `1` for the upper magnitude (passband) and `0` for the lower (stopband). Another parameter is how much ripple in the frequency domain is allowed in the passband and the stopband.

```
bands <- c(cutoff_freq, cutoff_freq + 1)
magnitude <- c(1,0)
ripple_amplitude <- 0.01
```

The `bands` and `magnitude` parameters indicate that up to the cutoff frequency, I want to pass all the frequencies, and then I want to attenuate to zero the frequencies 1 Hz above the cutoff. The more narrow the band is set, the longer the filter will be. Remember that the longer the filter, the more raw is lost (or is left unsmoothed) at the beginning and end of a trace. To keep the filter length reasonable, you should experiment based on the total number of data points you have and the sampling frequency of your system. For the current example, a 1 Hz wide band gives reasonable performance. The `ripple_amplitude` parameter specifies how much ripple or ringing in the frequency domain is allowed.

Look at the R help page for `kiaserord()` for more details on the design formulas used.

```
  kord <- kaiserord(bands, magnitude, ripple_amplitude,
  fs=sampling_frequency)
  kord
```

```
## $n
## [1] 38
##
## $Wc
## [1] 0.2479699
##
## $type
## [1] "low"
##
## $beta
## [1] 3.395321
##
## attr(,"class")
## [1] "FilterSpecs"
```

The output of `kaiserord()` is the length (called the filter order) of the filter (`n=38`), the passband edge where `1` is the Nyquist frequency, and `beta` which is the shape parameter for the Kaiser window. The `fir1()` function makes a *finite impulse response* (FIR) with a single cutoff frequency, which is just another name for the moving average type filters I've been showing so far. The FIR filter design parameters are the length, the cutoff, the type, and the window function. This is where the Kaiser window, implemented in `kaiser()` function, is applied. The Kaiser window has to be one longer than the order and be given the shape parameter. I manually normalize the scale so that the entire filter sums to `1`, so the `scale` parameter for `fir1()` is set to `FALSE`. The result is normalized filter coefficients that can be used as a convolutional filter in the time domain.

```
fir <- fir1(kord$n, kord$Wc, type="low", scale=FALSE,
kaiser(kord$n+1, beta=kord$beta))
fir <- fir / sum(fir)
```

The filter coefficients generated are quite different from the Boxcar and SG filters:

```
fir_x <- seq(0,length(fir)-1)/(floor(length(fir))-1)

p_filter_compare <- p_filter_coeff +
    coord_cartesian(ylim=c(-0.1, 0.3)) +
    geom_line(aes(x=fir_x, y=fir, color="Kaiser"),
            linewidth=0.5) +
    geom_point(aes(x=fir_x, y=fir), shape=1) +
    scale_color_manual(name='Filter: ',
              breaks=c('Box Car', 'SG Quadratic',
'Kaiser'),
              values=c(pal$blue, pal$red,
pal$darkorange))

print(p_filter_compare)
```

Figure 6.28 shows that the Kaiser window smoothed out the ripple in the time domain of the sinc function shown in Figure 6.26 and will leave less distortion in the smoothed data that the other filters as a result of beginning and ending very close to zero, which is the entire goal of Kaiser windowing.

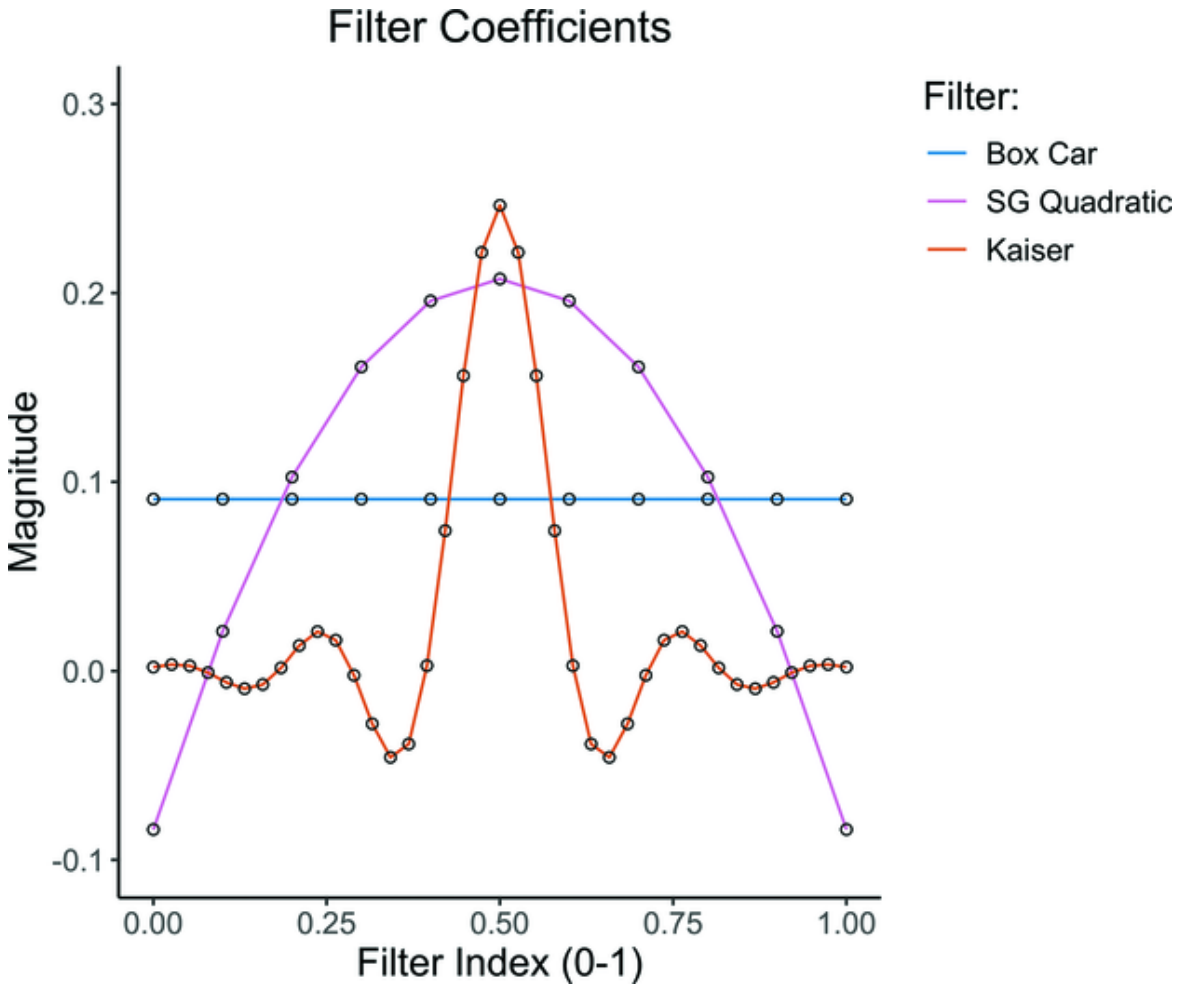


Figure 6.28 The Kaiser windowed sinc function compared to the shapes of both the Boxcar and the SG filters. Note the similarity between the windowed sinc function and the Mexican Hat used for CWT analysis.

The frequency response of the Kaiser filter is also dramatically different from the Boxcar and SG filters.

```
fir_fr <- freqz(fir, fs=sampling_frequency)

p_freqz_compare <- p_freqz +
    geom_line(aes(x=fir_fr$w, y=abs(fir_fr$h),
color='Kaiser'),
            linewidth=0.5) +

    scale_color_manual(name='Signal: ',
         breaks=c('Boxcar', 'SG Quadratic', 'Kaiser',
'Raw Trace'),
        values=c(pal$blue, pal$red, pal$darkorange,
pal$black)) +
    ggtitle(label = "Frequency Response of FIR
Filters",
        subtitle = "Overlayed on the Frequency Content
of the Raw Trace")

print(p_freqz_compare)
```

Notice that in Figure 6.29, the Kaiser filter has almost no side nodes compared to the other filters. Also, notice a slight ripple at the top and bottom of the frequency response. These will be within the `0.01` (1%) limit specified.

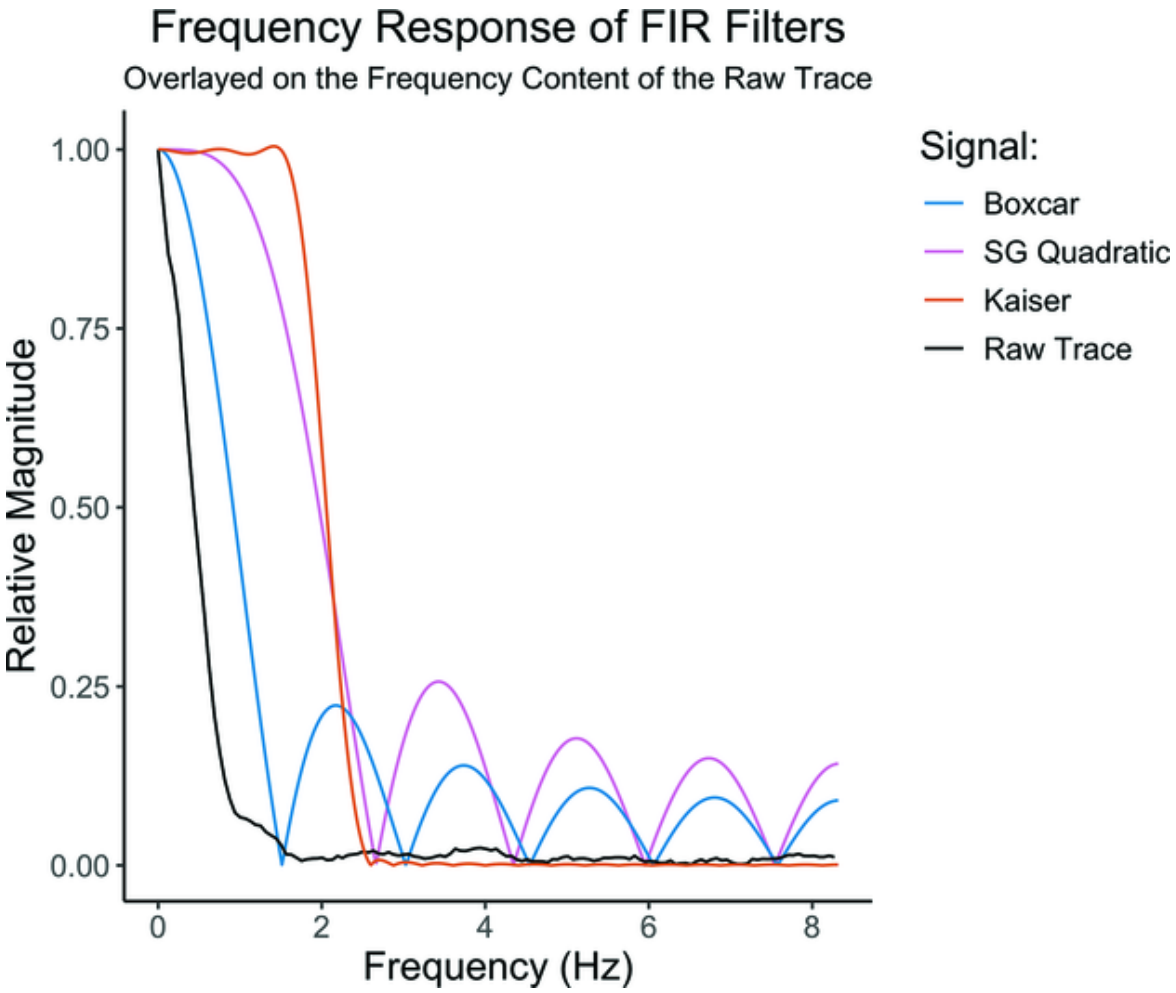


Figure 6.29 The Kaiser windowed sinc function compared to the frequency responses of the Boxcar and the SG filters. The goal of the Kaiser window is to reduce the power in the side lobes, which are dramatically lower than the other filters.

First, smooth the data using the same convolution method shown in Figure 6.23:

```
smooth_y_fir <- gsignal::conv(y, fir, shape="same")
```

And then plot the smoothed data:

```
p_fir_smooth <- ggplot() +
        scale_y_continuous(labels = inten_label) +
        geom_line(aes(x=t, y=smooth_y_fir,
color='Kaiser FIR'),
            linewidth=0.5) +
        geom_point(aes(x=t, y=y), shape=1) +
        scale_color_manual(name='Filter: ',
                 breaks=c('Kaiser FIR'),
                 values=c(pal$darkorange)) +
        ggtitle(label = "Smoothing with Optimized
Kaiser Windowed Sinc") +
        xlab("Retention Time (s)") +
        ylab("Intensity (counts/s)") +
        theme_classic() +
        theme(plot.title = element_text(hjust = 0.5,
size=16)) +
        theme(plot.subtitle = element_text(hjust =
0.5)) +
        theme(
            axis.text=element_text(size=11),
            axis.title=element_text(size = 14),
            legend.text = element_text(size = 11),
            legend.title = element_text(size = 14)
        ) +
        theme(
            legend.position.inside = c(.95, .95),
            legend.justification = c("right", "top"),
            legend.box.just = "right",
            legend.margin = margin(6, 6, 6, 6)
        )

p_sinc_windowed <- p_sinc_smooth +
    geom_line(aes(x=t, y=smooth_y_fir, color='Kaiser
FIR'),
              linewidth=0.5) +
    scale_color_manual(name='',
            breaks=c('Raw Data', 'Sinc Filter', 'Kaiser
FIR', 'Raw Data'),
            values=c(pal$black, pal$lightblue,
pal$darkorange),
            guide = guide_legend(override.aes = list(
```

```
                            linetype = c("blank", "solid",
"solid"),
                            shape = c(1, NA, NA)
                        ))) +
    ggtitle(label="Frequency Truncation vs. Windowing",
            subtitle = "Kaiser Window") +
    theme(  legend.title=element_blank(),
            legend.position.inside = c(0.95, 1.2),
            legend.justification = c("right", "top"),
            legend.box.just = "right",
            legend.margin = margin(4, 4, 4, 4)
        )

ggarrange(p_fir_smooth, p_sinc_windowed,
          ncol = 1, nrow = 2,
          labels = c("A", "B"))
```

In Figure 6.30a, the smoothed plot shows very little attenuation at the apex, and in Figure 6.30b the ripple artifacts are significantly reduced, and the remaining values appear much closer to a mean value for the noise in the raw data.

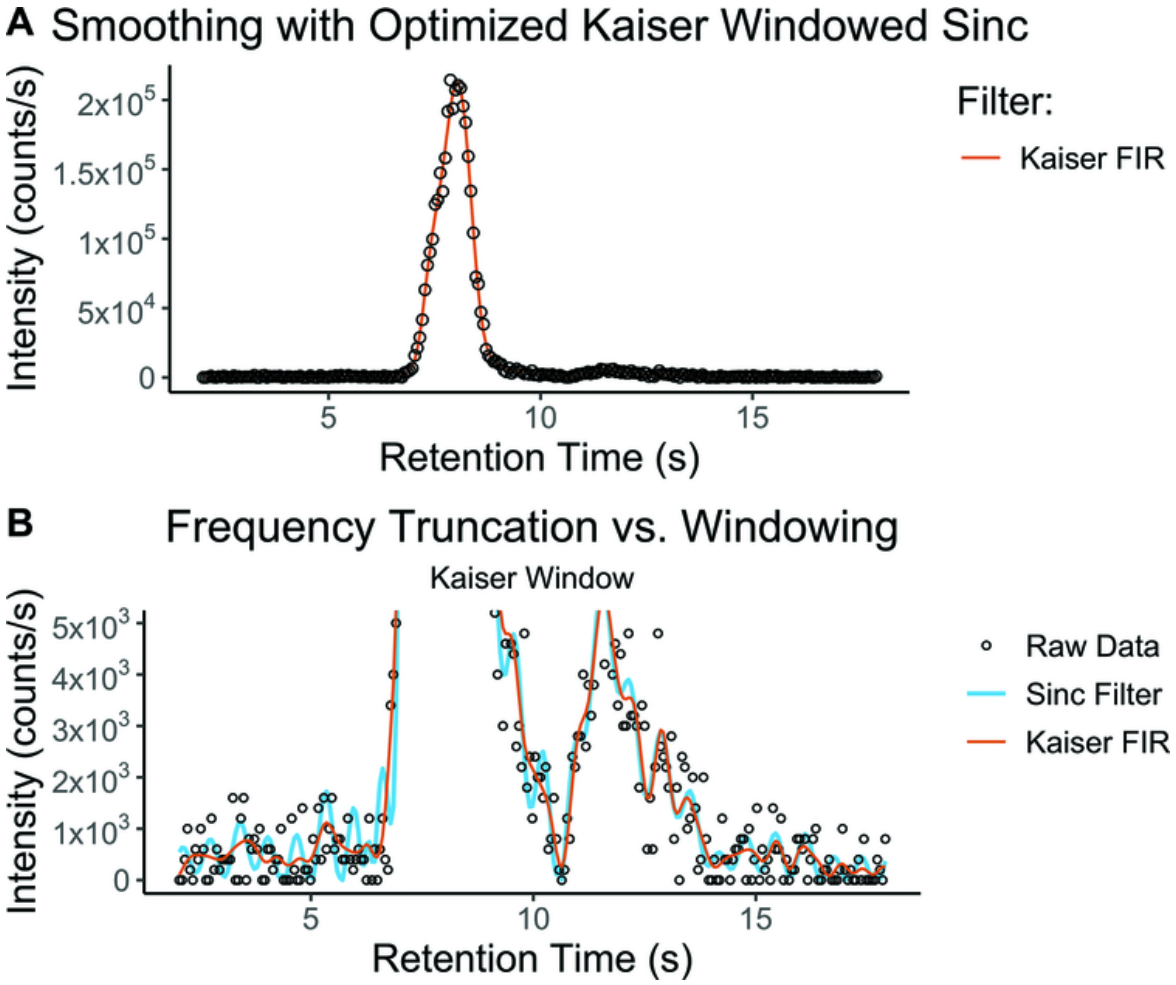


Figure 6.30 Raw data smoothed with an optimized FIR filter.

6.5 Quantification

Once a peak has been detected, its baseline corrected, and its start and end times determined, the area of the peak must be determined in order for it to be used for quantitative analysis. The peak area is almost always used for quantification instead of peak height because of asymmetries, which are apparent in Figure 6.5 and more common in LC separation than GC. The most basic area calculations are performed using numeric integration of the raw or smoothed data. To improve sample-to-sample

variation, often a *stable isotopic labeled* (SIL) version of the target compound is used as an IS. The goal is to match the chromatographic properties of the IS with the target but use differences in molecular weight and fragmentation properties to distinguish the two compounds. When an IS of any kind is used, instrument response can be computed as the ratio of the areas of the target and the IS ion peaks. Regardless of how it is computed, the instrument response can then be converted into a concentration using a calibration curve. Calibration curves are made from samples with known concentrations of the target ion. Deviations of the back-calculated standard concentrations are used to evaluate the calibration process, and the calculated concentrations of the QC samples are used to evaluate the accuracy of the method when applied to unknown samples.

6.5.1 Numeric Integration

The first step in quantification is calculating the area of the analyte peak picked from the trace. The idea behind the numeric integration of empirical data is quite basic. Each data point in an LC-MS chromatogram represents a certain amount of ion signal at a particular time. Between any two data points, a polygon can be drawn from the baseline, between the points, and back to the baseline. One way to estimate the area under a curve is to sum the areas of all the polygons that fall between the peak start and peak end. This ancient method, also called the *trapezoidal rule*, is adequate when you have very narrowly spaced data points, such as when you are integrating a mathematical formula and can choose whatever spacing you like. For real data, the use of parabolas (quadratic polynomials) gives a better approximation of the underlying smooth function representing a chromatogram. This approach is called *Simpson's Rule* and is one of the most commonly used methods for numerical integration of empirical data [192]

and can be performed using the `int.simpsons2()` function, which is available from the `fda.usc` package [193].

To apply Simpson's Rule in R, I'll select the part of the baseline corrected trace to be integrated using the peak start and end calculated in the previous section.

```
peak_t <- t[peak_start_index:peak_end_index]
peak_y <- y_smooth[peak_start_index:peak_end_index]
```

And then call `int.simpsons2()` with the parameter `equi` set to `FALSE` since the data in a raw data trace from LC-MS measurements will usually **not** have equally spaced data points.

```
peak_area <- int.simpson2(peak_t, peak_y, equi=FALSE)
print(peak_area)
```

```
## [1] 210652.3
```

The area determined by Simpson's rule is slightly different from the value calculated by `xcms`:

```
peak_info[1,]$intb
```

```
## [1] 202575.6
```

The difference can be explained by the more conservative bounds for the peak shown in Figure 6.19. However, they are less than 4% different, as mentioned before. If done consistently, the end results should be comparable.

This process can be repeated for all of the traces in the sample in order to compute area ratios for normalization (quant area/IS area) and for identification (quant area/qual area).

6.5.2 Normalized Instrument Response

A common task in quantitative analysis by mass spectrometry is controlling for run-to-run variation. One of the most common methods for normalizing a quantitative measurement is to add IS to the sample and measure a unique tandem mass spectrometry (MS/MS) transition. Typically, IS normalization methods treat the concentration of labeled ions as having no error in the quantity from pre-analytical steps, so any change in the IS peak area is systemic and also affects the analyte compound. Dividing the analyte peak area by the IS peak area yields an instrument response that removes variability from the overall measurement.

The areas of all four traces in each of the example samples can be computed using the procedure described in Sections 6.2 and 6.3.3 and used for normalization, calibration, and QC. I've saved these values in a file so they can be loaded into a data table.

```
result_table <- read_csv(file.path("data",
"results.csv"))
```

```
names(result_table)
```

```
##  [1] "sample"          "rt_Quant"        "rt_Qual"
"rt_Quant.IS"
##  [5] "rt_Qual.IS"      "area_Quant"      "area_Qual"
"area_Quant.IS"
##  [9] "area_Qual.IS"    "height_Quant"    "height_Qual"
"height_Quant.IS"
## [13] "height_Qual.IS"  "fwhm_Quant"      "fwhm_Qual"
"fwhm_Quant.IS"
## [17] "fwhm_Qual.IS"    "noise_Quant"     "noise_Qual"
"noise_Quant.IS"
## [21] "noise_Qual.IS"   "snr_Quant"       "snr_Qual"
```

```
"snr_Quant.IS"
## [25] "snr_Qual.IS"
```

The assay this example came from had the following design: two *double blank* samples (meaning they had no IS added) were run, then seven calibrators in increasing concentration, followed by a blank with IS added, and then four QC samples. This design can be added to the result table.

```
sample_type <- c(rep("blank", 2), rep("cal",7),
"blank", rep("qc",4))

known_concentration <- c(0, 0, 5, 10, 25, 100, 250,
1000, 2500, 0,
                         145.2, 129.1, 1204, 602.5)

result_table$type <- sample_type
result_table$conc <- known_concentration
```

Now, for all the samples, the area ratios can be computed:

```
result_table$instrument_response <-
    result_table$area_Quant /
result_table$area_Quant.IS
```

The area ratios will be used to fit the instrument responses to the known concentrations using an equation that can then be used to back-calculate concentrations for standard and QC and, ultimately, unknown samples.

6.5.3 Calibration Using Standards

The recommended practice for performing concentration calibration for LC-MS data is to use the lowest variance model possible, taking into account the observed heteroskedastic nature of the variance in the measurement, which increases with signal intensity. This change in variance

with intensity was shown with both smoothing-based deviation calculations and using continuous wavelet transformations. As discussed throughout this chapter, the underlying calibration function is not completely known, so it has to be estimated. However, the number of factors that go into both the instrument response and the expected concentration are large enough to expect the function to be smooth with low variance. Under perfect conditions, the instrument response can be perfectly linear, requiring only a weighted linear regression to obtain the slope and intercept of the relationship. When using least squares to solve the linear regression, the concentration is the independent variable and is assumed to have no error. The instrument response is the independent variable and is assumed to carry all the experimental error.

Because of ion source saturation, detector saturation, or other upper limits to detection, calibration curves often *roll off* at the high-concentration end of the curve. If this is observed in your experiments, the quadratic curve can be used, adding curvature to the straight line equation while keeping weighting and the closed-form solution offered by the least squares method.

With any type of model, it is important to keep an eye on the diagnostic metrics associated with the model. In the case of lines and parabolas (first- and second-order polynomials), the residuals should be examined to determine if there is a pattern present. When low variance models are working properly, the differences between the actual and predicted values (the residuals) will be randomly scattered around zero. Patterns indicate problems with either the model, the fit, or the samples used.

Calibrations in LC-MS often use a weighted linear least squares fit. To perform a weighted fit, you have to supply a vector of weight values (one for each data point in the dataset) to the `lm()` function. Many LC-MS calibrations use

$1/x$ or $1/x^2$ weighting, which can be computed from the concentration (x) and used in the fit.

```
cals <- result_table |>
    dplyr::filter(type=="cal") |>
    dplyr::select(c(conc, instrument_response))

# use 1/x^2 weighting
conc_weights <- 1/cals$conc^2

m <- lm(instrument_response ~ conc,
weights=conc_weights, data=cals)
```

The `lm()` function uses the base R *model formula* to specify a relationship between variables. In the model formula `instrument_response ~ conc`, the ~ should be read as *is described by*. The R model formula in the code above states that the independent variable `instrument_response` is described by `conc`. The default for model formulas is to include a *bias* term, which in linear equations is usually called the y-intercept. The model formula used here will be parsed and create a standard equation for a line: $y = a_1 x + a_0$ where y is `instrument_response`, a_1 (slope) is the coefficient multiplied by `conc` and a_0 is the bias/y-intercept which will also be computed by `lm()` using the matrix algebra approach to obtain the least squares solution for the coefficients (see the `lm()` help page for more detail).

The return value from `lm()` is a linear model. Information about the fit performed can be best accessed through the `summary()` function.

```
model_summary<-summary(m)
print(model_summary)
```

```
##
## Call:
```

```
## lm(formula = instrument_response ~ conc, data = cals,
weights = conc_weights)
##
## Weighted Residuals:
##          1          2          3          4          5
6          7
## -1.866e-05  4.423e-05  8.163e-06 -8.438e-05 -5.827e-05
2.680e-05  8.212e-05
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 9.403e-03  3.477e-04   27.05 1.29e-06 ***
## conc        5.796e-03  2.988e-05  193.94 6.91e-11 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.'
0.1 ' ' 1
##
## Residual standard error: 6.379e-05 on 5 degrees of
freedom
## Multiple R-squared:  0.9999, Adjusted R-squared:
0.9998
## F-statistic: 3.761e+04 on 1 and 5 DF,  p-value: 6.915e-
11
```

From the model, the fit looks to have performed very well. The standard errors on the coefficients are low, and the probability of them occurring by random (the Pr(>|t|) value) is extremely low.

The `predict()` function can be used to get the predicted instrument responses from the concentrations used in the fit.

```
cals$predicted <- as.numeric(predict(m))
```

Plotting the calibration and its residuals gives a hint about the quality of the fit:

```
p_cals <- cals |>
    ggplot() +
    geom_point(aes(x=conc, y=instrument_response),
shape=1) +
    geom_line(aes(x=conc, y=predicted)) +
    ggtitle(label = "Linear Calibration",
            subtitle=parse(text=paste0(sprintf("~ R^2:
%0.4f",

model_summary$r.squared),
                                       " - ",
                                    sprintf("Adjusted ~
R^2: %0.4f",

model_summary$adj.r.squared)))) +
        xlab("Concentration") +
        ylab("Instrument Response (area ratio)") +
        theme_classic() +
        theme(plot.title = element_text(hjust = 0.5,
size=16)) +
        theme(plot.subtitle = element_text(hjust =
0.5)) +
        theme(
            axis.text=element_text(size=11),
            axis.title=element_text(size = 14),
            legend.text = element_text(size = 11),
            legend.title = element_text(size = 14)
        )

print(p_cals)
```

In most applications, goodness of fit scores based on the *coefficient of determination* (R-squared and Adjusted R-squared) are not considered particularly useful as metrics of calibration quality. The concentration of the expected instrument response is computed by inverting the calibration equation (solving the equation for x) and then compared to the known concentration (Figure 6.31). If the expected values are within an acceptable tolerance range, then the

calibration is considered successful. It is, however, prudent to plot the differences between the expected and known instrument responses (the residuals) to ensure there is no obvious pattern. It also helps when the number of calibrators is low, and the calibration equation is more complex than a line.

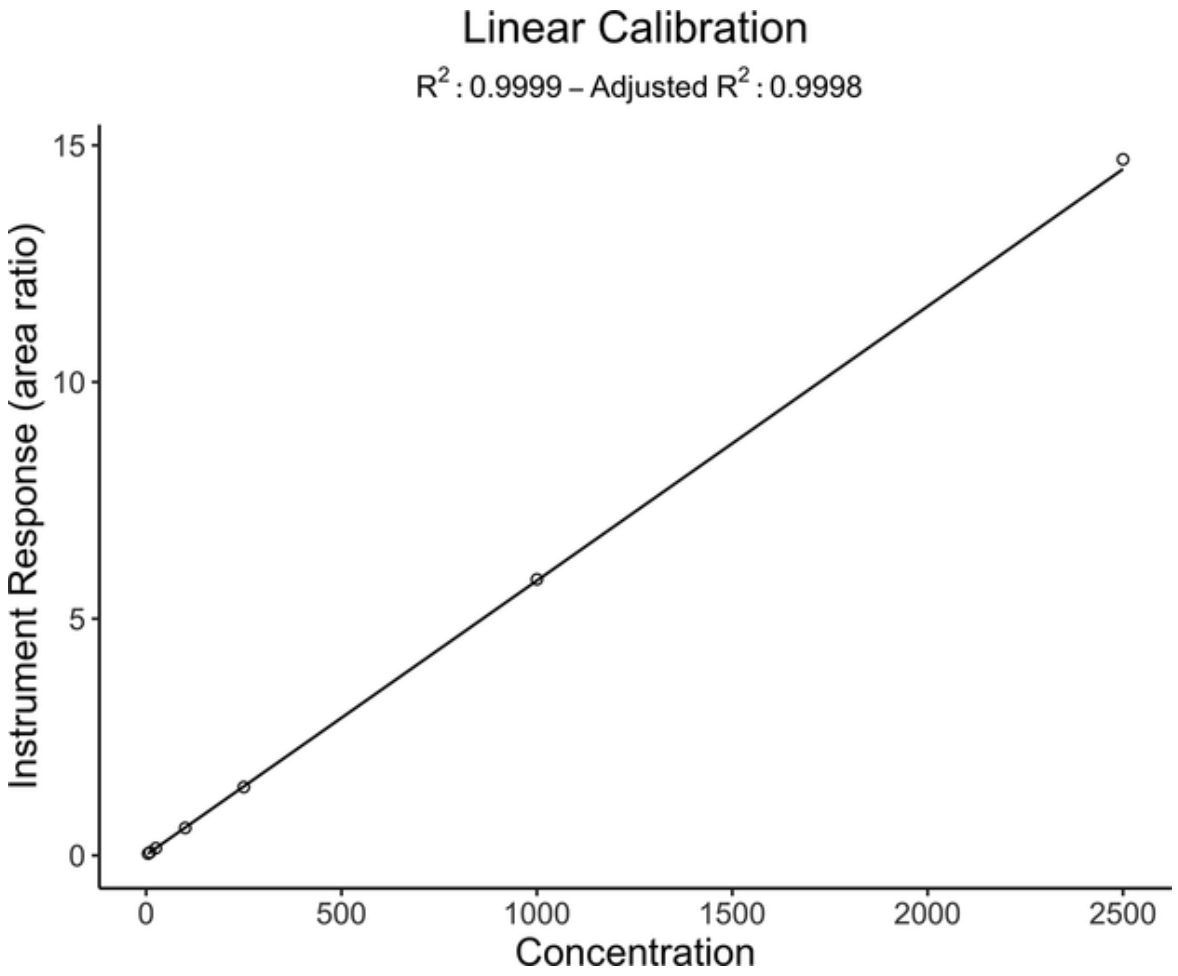


Figure 6.31 Linear calibration using $1/x^2$ weighting.

```
p_residuals <- cals |>
    ggplot() +
    geom_point(aes(x=conc, y=m$residuals), shape=1) +
    geom_hline(yintercept = 0) +
    ggtitle(label = "Residuals of Linear Calibration",
            subtitle=parse(text="~ 1/x^2 ~ Weighting"))
+
        xlab("Concentration") +
        ylab("Residual") +
        theme_classic() +
        theme(plot.title = element_text(hjust = 0.5,
size=16)) +
        theme(plot.subtitle = element_text(hjust =
0.5)) +
```

```
            theme(
                axis.text=element_text(size=11),
                axis.title=element_text(size = 14),
                legend.text = element_text(size = 11),
                legend.title = element_text(size = 14)
            )

    print(p_residuals)
```

In the residual plot shown in Figure 6.32, there is a clear pattern suggesting that the data used in the calibration has some nonlinearity. Before changing the equation or the weighting, however, I need to look at the calculated concentrations of the calibrators to see if the nonlinearity creates an error large enough to be significant for my application.

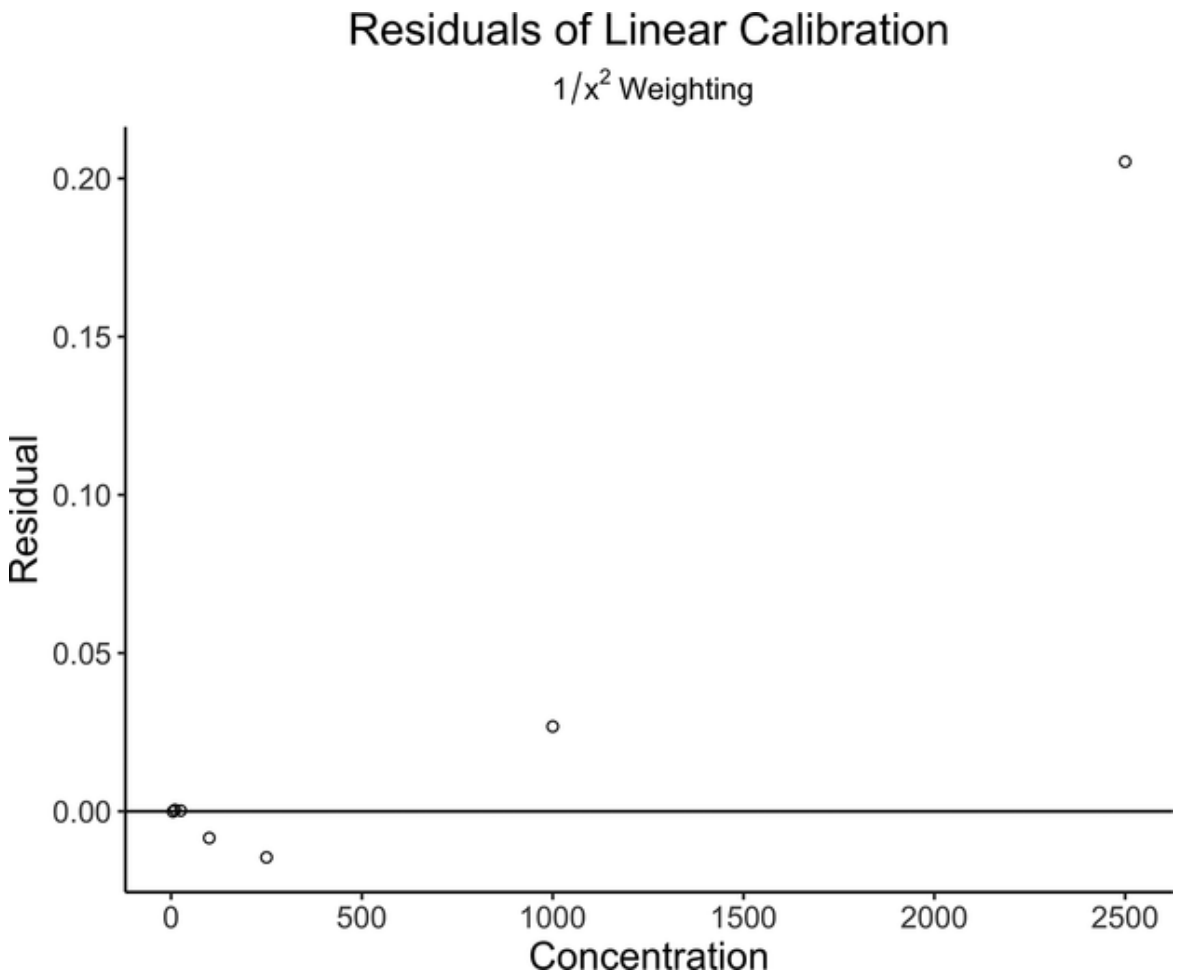


Figure 6.32 Residuals from the linear fit of the calibration data using $1/x^2$ weighting.

```
calc_conc <- (cals$instrument_response -
m$coefficients[1])/m$coefficients[2]
cals$calc_conc <- calc_conc
```

The percent difference between what the calibration will assign and the known values:

```
conc_deviation <- 100*(1-cals$conc/cals$calc_conc)
cals$conc_deviation <- conc_deviation
cals |>
    dplyr::select(c(conc,calc_conc,conc_deviation))
```

```
## # A tibble: 7 x 3
##    conc calc_conc conc_deviation
##   <dbl>     <dbl>          <dbl>
## 1     5      4.98         -0.323
## 2    10     10.1           0.757
## 3    25     25.0           0.141
## 4   100     98.5          -1.48
## 5   250    247.           -1.02
## 6  1000   1005.            0.460
## 7  2500   2535.            1.40
```

The calculated concentration error `conc_deviation` is less than `1.5%` for all concentrations, and because of the weighting, it is extremely small for all the calibrators. For a biological assay in which reproducible and accurate results are demanded, this calibration is extremely good. At the time of writing, the allowance for clinical diagnostic tests is `20%` for the lowest reported value (the lowest calibrator) and `15%` for the highest. For biomarker studies like the proteomics example used in Chapter 5, the precision of repeated studies needs to take clinical guidelines into consideration in order to ensure the measurement can be reproduced in other labs or eventually be used in clinical or pharmaceutical applications.

For other applications of LC-MS, these tolerances, along with the observation of nonlinearity in the residuals, might make this calibration unacceptable. In those cases, an investigator might view the nonlinear residuals as justifying the use of a higher order polynomial to account for the evidence of curvature. The next simplest model is a quadratic polynomial, which adds a $a_2 x^2$ term to the equation. Since the coefficients to be fit are *pre-exponential* (not in the exponent of any variable), the model can be fit using the linear least squares approach using `lm()`.

The quadratic calibration curve introduces several new issues that need to be addressed. First, the model formula

must be changed to include the squared concentration term. To fit a quadratic model to the calibration data using the R formula, the `I()` (also called `AsIs`) function prevents the formula parser from interpreting the symbols +, -, *, and ^ as formula operators so they are used as arithmetic operators. Formula operators carry completely different meanings than their arithmetic counterparts and are important for describing various interactions between factors [47, 194]. The new formula `instrument_response` *is described by* (~), the linear combination of concentration and concentration squared: `conc + I(conc^2)`.

```
q <- lm(instrument_response ~ conc + I(conc^2),
        weights=conc_weights, data=cals)
```

```
quadratic_model_summary<-summary(q)
print(quadratic_model_summary)
```

```
##
## Call:
## lm(formula = instrument_response ~ conc + I(conc^2),
data = cals,
##     weights = conc_weights)
##
## Weighted Residuals:
##          1          2          3          4          5
6          7
## -3.373e-05  5.794e-05  3.854e-05 -4.915e-05 -2.899e-05
1.845e-05 -3.064e-06
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 9.694e-03  2.999e-04  32.321 5.46e-06 ***
## conc        5.752e-03  3.070e-05 187.341 4.87e-09 ***
## I(conc^2)   5.134e-08  2.422e-08   2.119    0.101
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.'
0.1 ' ' 1
```

```
##
## Residual standard error: 4.895e-05 on 4 degrees of
freedom
## Multiple R-squared:  0.9999, Adjusted R-squared:
0.9999
## F-statistic: 3.194e+04 on 2 and 4 DF,  p-value: 3.92e-
09
```

The first thing to notice is that the p-value for the `I(conc^2)` coefficient is above the traditional 5% chance of happening by random. That doesn't mean it's not useful (it is), but that it had a 10% chance of occurring at random. The small value for the `I(conc^2)` coefficient makes sense, given how small the deviation from linearity the first calibration appeared to be as shown in Figure 6.32.

```
p_q_residuals <- cals |>
    ggplot() +
    geom_point(aes(x=conc, y=q$residuals), shape=1) +
    geom_hline(yintercept = 0) +
    ggtitle(label = "Residuals of A Quadratic
Calibration",
            subtitle=parse(text="~ 1/x^2 ~ Weighting"))
+
        xlab("Concentration") +
        ylab("Residual") +
        theme_classic() +
        theme(plot.title = element_text(hjust = 0.5,
size=16)) +
        theme(plot.subtitle = element_text(hjust =
0.5)) +
        theme(
            axis.text=element_text(size=11),
            axis.title=element_text(size = 14),
            legend.text = element_text(size = 11),
            legend.title = element_text(size = 14)
        )

print(p_q_residuals)
```

The importance of the small value for `I(conc^2)` can be seen in the residuals shown in Figure 6.33. The residuals are more randomly distributed compared to those in Figure 6.32, and the magnitudes of the residuals are quite a bit smaller than the residuals from the linear fit. This is exactly what is expected when you increase the *variance* of a model but keep the number of data points the same. Taken further, increasing the complexity of the model can lead to *overfitting* producing an equation that goes through every data point perfectly, which produces residuals of zero. Overfitting is a major concern in model fitting and will be discussed in more detail in the next chapter when applying machine learning models to mass spectrometry data.

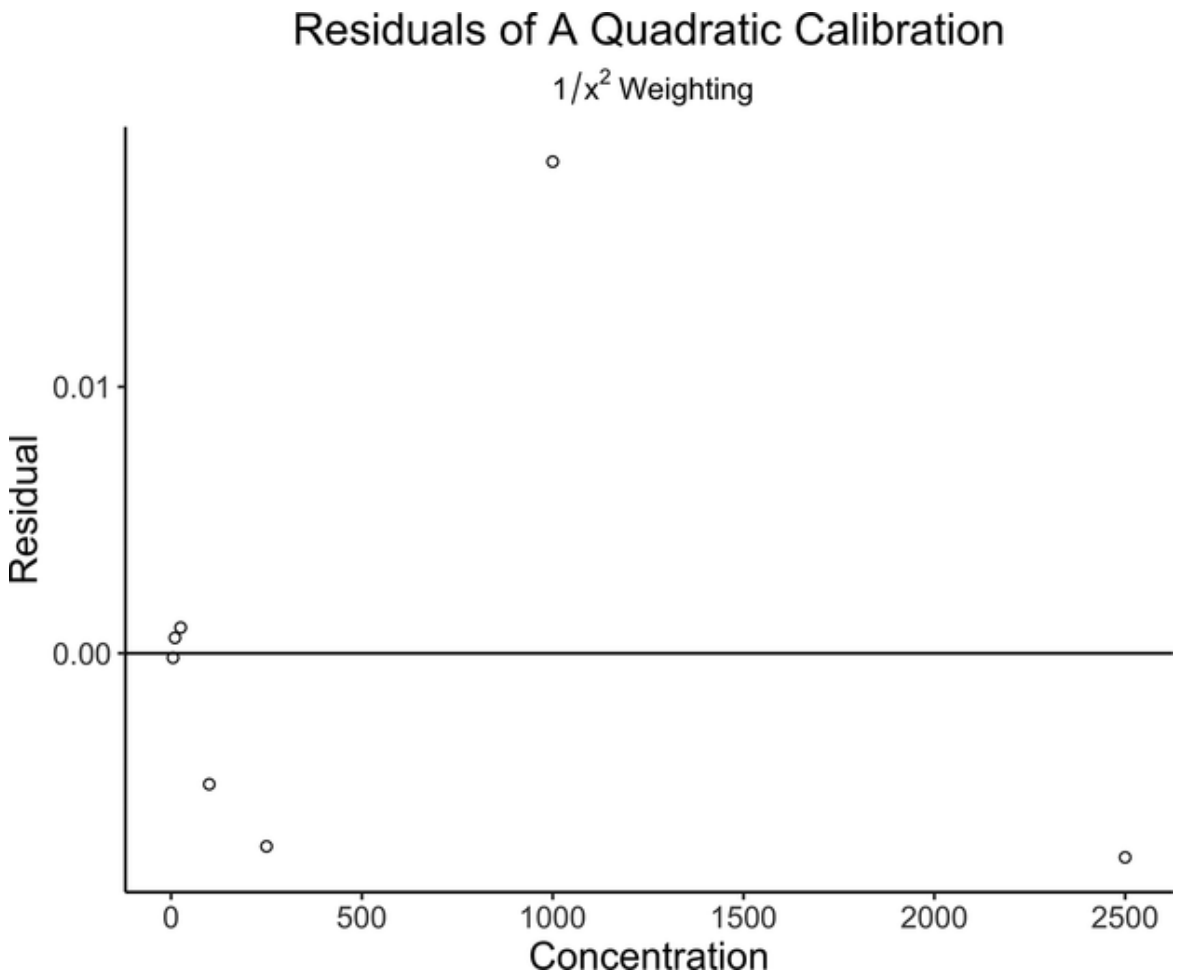


Figure 6.33 Residuals from a quadratic fit of the calibration data using $1/x^2$ weighting.

The next issue with quadratic fits is that the inversion of the equation is the well-known *quadratic equation*. This equation is the inverse of the quadratic polynomial and can be used to solve for concentration from instrument response from the polynomial coefficients.

The equation just fit is the quadratic polynomial shown in Eq. (6.6):

$$y = a_0 + a_1 x + a_2 x^2 \tag{6.6}$$

where a_0 is the intercept, a_1 is the coefficient for concentration (`conc`), and a_2 is the coefficient for the

concentration squared: `I(conc^2)`. The inverse of the fit equation is the general solution to quadratic equations from basic algebra shown in Eq. (6.7).

$$x = \frac{-a_1 \pm \sqrt{a_1^2 - 4a_0a_2 + 4a_2y}}{2a_2} \tag{6.7}$$

Three well-known problems can arise with Eq. (6.7) when applied to real physical systems. First, it is possible for this equation to generate imaginary numbers if the $4a_0a_2$ term is large. Second, the equation can produce two possible values because of the $\pm$ in the numerator. Finally, the equation can produce a negative concentration. A back calculation program has to take these nonphysical solutions into account, which means the concentration could be *undefined* for a given instrument response. It also means that the program will have to be able to choose between two real solutions to return a single concentration from an instrument response.

In chromatography calibration, only the first real positive solution is physically meaningful. This physical constraint makes it easier to write a program for the quadratic inverse. However, the program may have to return an `NA` value if the concentration is undefined (negative or complex). One common occurrence in quadratic calibration is when the quadratic equation is a parabola with an apex instrument response lower than the highest possible instrument response. In this case, any instrument response above the apex will have no real solution. These are called *no- intercept* results when they occur in practice.

First, I'll show that the quadratic equation does indeed fit the example data better than the linear equations.

```
inverse_quad <- function(y, a0, a1, a2) {
     x <- rep(NA, times=length(y))

    # bail out if there is going to be a divide by zero
error
    if(a2 != 0) {
        for(i in 1:length(x)) {
            A <- a1^2 + 4*a2*y[i]
            B <- 4*a0*a2
            # if the solution is real
            if(B <= A) {
                C <- sqrt(A-B)
                # if the solution is positive
                if(C >= -a1) {
                    x[i] <- (-a1 + C)/(2*a2)
                }
            }
        }
    }
    x
}
```

```
quad_conc <- inverse_quad(cals$instrument_response,
                          q$coefficients[1],
                          q$coefficients[2],
                          q$coefficients[3])
cals$quad_conc <- quad_conc
```

And now, I want to see the change in the deviations:

```
quad_deviation <- 100*(1-cals$conc/cals$quad_conc)
cals$quad_deviation <- quad_deviation
cals |>
    dplyr::select(c(conc, calc_conc, quad_conc,
conc_deviation, quad_deviation))
```

```
## # A tibble: 7 x 5
##    conc calc_conc quad_conc conc_deviation
```

```
quad_deviation
##   <dbl>     <dbl>     <dbl>          <dbl>
<dbl>
## 1     5      4.98      4.97         -0.323
-0.590
## 2    10     10.1      10.1           0.757
0.997
## 3    25     25.0      25.2           0.141
0.665
## 4   100     98.5      99.1          -1.48
-0.860
## 5   250    247.      249.           -1.02
-0.504
## 6  1000   1005.     1003.            0.460
0.314
## 7  2500   2535.     2499.            1.40
-0.0510
```

Notice that the quadratic calibration gives a uniformly improved deviation, being fractionally worse than the linear fit at the low concentrations but significantly better in the upper four calibrators. This small check may give you a good reason to use the quadratic equation, especially in this fit, where the nonlinear term a_2 is small and positive. A positive nonlinear term means that the curvature is positive, and there will always be a single value for every instrument response as they get larger.

However, in real experiments, there is a risk of a high-concentration calibrator saturating the detector in some way that calls for curvature in the calibration equation. As an example, I'll artificially change the highest calibrator to 10 to represent a significant degree of saturation.

```
alt_cals <- result_table |>
    dplyr::filter(type=="cal") |>
    dplyr::select(c(conc, instrument_response))

alt_cals[7,]$instrument_response <- 10.0
```

```r
alt_q <- lm(instrument_response ~ conc + I(conc^2),
        weights=conc_weights, data=alt_cals)
```

```r
alt_quadratic_model_summary<-summary(alt_q)
print(alt_quadratic_model_summary)
```

```
##
## Call:
## lm(formula = instrument_response ~ conc + I(conc^2),
data = alt_cals,
##     weights = conc_weights)
##
## Weighted Residuals:
##          1          2          3          4          5
6          7
##  3.944e-05 -1.610e-05 -1.147e-04 -1.903e-04 -6.456e-05
5.517e-04 -2.055e-04
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept)  8.183e-03  1.942e-03   4.214   0.0135 *
## conc         5.985e-03  1.988e-04  30.103 7.25e-06 ***
## I(conc^2)   -7.131e-07  1.569e-07  -4.546   0.0104 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.'
0.1 ' ' 1
##
## Residual standard error: 0.000317 on 4 degrees of
freedom
## Multiple R-squared:  0.997,  Adjusted R-squared:
0.9955
## F-statistic: 667.7 on 2 and 4 DF,  p-value: 8.918e-06
```

```
alt_quad_conc <-
inverse_quad(alt_cals$instrument_response,
                                 alt_q$coefficients[1],
                                 alt_q$coefficients[2],
                                 alt_q$coefficients[3])
alt_cals$alt_quad_conc <- alt_quad_conc
```

Calculating the predicted values is done in the same way as above. For plotting the curved calibration line, I'm using the model to predict a high-resolution version of the equation between `0` and `2500` stepping by `1`:

```
alt_cals$predicted <- as.numeric(predict(alt_q))
alt_cal_line <- as.numeric(predict(alt_q,
newdata=data.frame(conc=seq(0, 2500, 1))))
```

Now I can plot the calibrators and the curve:

```
p_alt_cals <- alt_cals |>
    ggplot() +
    geom_point(aes(x=conc, y=predicted), shape=19,
color=pal$red) +
    geom_point(aes(x=conc, y=instrument_response),
shape=1) +
    geom_line(data=data.frame(x=seq(0, 2500, 1),
y=alt_cal_line),
              aes(x=x,y=y), color=pal$red) +
    ggtitle(label = "Saturated Quadratic Calibration",
            subtitle=parse(text=paste0(
                        sprintf("~ R^2: %0.4f",

alt_quadratic_model_summary$r.squared),
                        " - ",
                        sprintf("Adjusted ~ R^2:
%0.4f",

alt_quadratic_model_summary$adj.r.squared)))) +
        xlab("Concentration") +
        ylab("Instrument Response (area ratio)") +
        theme_classic() +
        theme(plot.title = element_text(hjust = 0.5,
size=16)) +
        theme(plot.subtitle = element_text(hjust =
0.5)) +
        theme(
            axis.text=element_text(size=11),
            axis.title=element_text(size = 14),
            legend.text = element_text(size = 11),
            legend.title = element_text(size = 14)
        )

print(p_alt_cals)
```

```
alt_quad_deviation <- 100*(1-
alt_cals$conc/alt_cals$alt_quad_conc)
alt_cals$alt_quad_deviation <- alt_quad_deviation
alt_cals |>
    dplyr::select(c(conc, alt_quad_conc,
alt_quad_deviation))
```

```
## # A tibble: 7 x 3
##    conc alt_quad_conc alt_quad_deviation
##   <dbl>         <dbl>              <dbl>
## 1     5          5.03              0.655
## 2    10          9.97             -0.270
## 3    25         24.5              -1.97
## 4   100         96.7              -3.37
## 5   250        247.               -1.16
## 6  1000       1123.               11.0
## 7  2500       2300.               -8.72
```

All of the calibrators are within the expected tolerance for a clinical application and there is nothing obviously wrong with the calibration plot shown in Figure 6.34. However, the curvature (quadratic) term is slightly negative, which means it is a downward-facing parabola that has an apex and will give a *no-intercept* result for some instrument responses above the apex. Choosing an instrument response that is only slightly larger than the value from the experimental data shows the effect:

```
high_response <- 18

inverse_quad(high_response,
             alt_q$coefficients[1],
             alt_q$coefficients[2],
             alt_q$coefficients[3])
```

```
## [1] NA
```

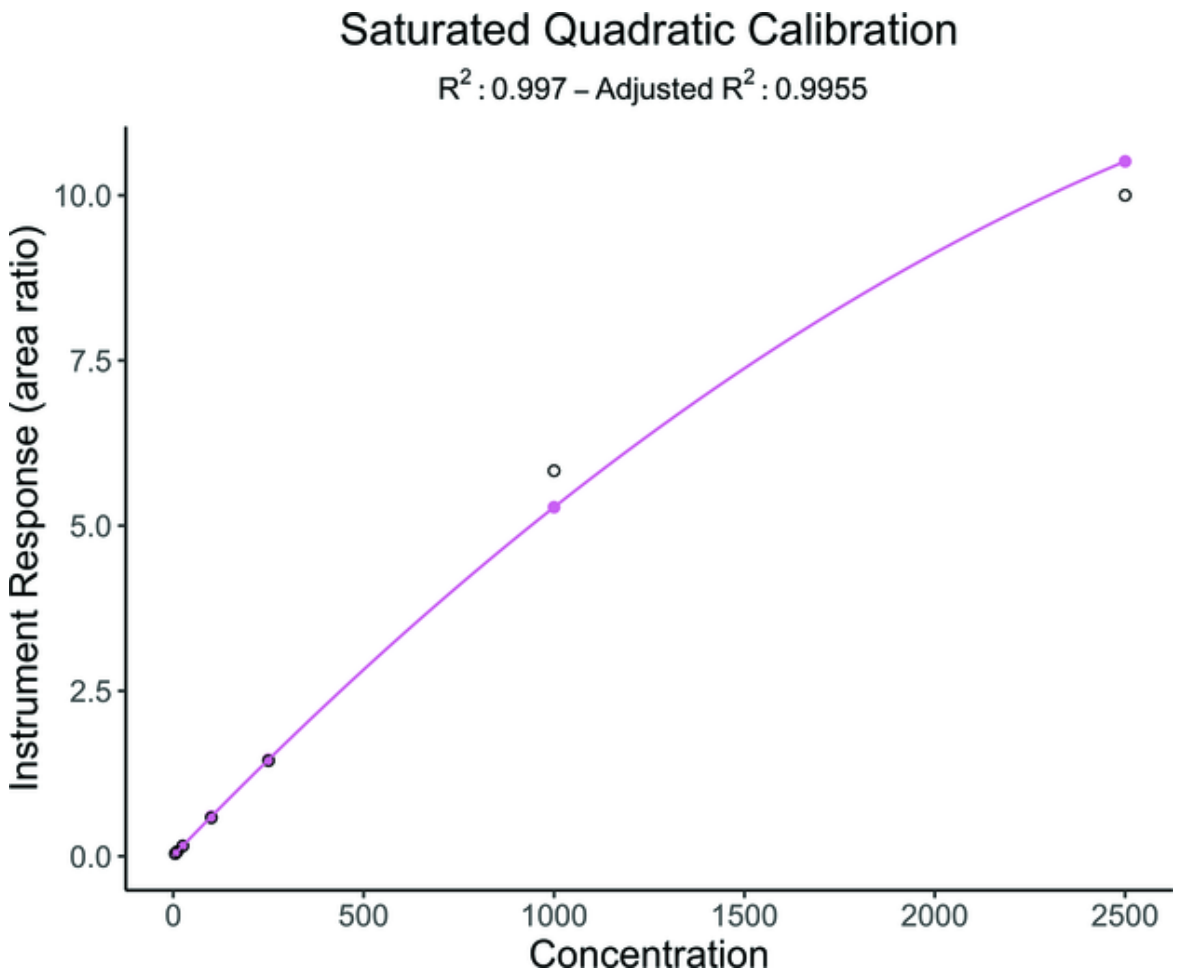


Figure 6.34 An example of a calibration fit that rolls over, which means there are instrument responses that will have no intercept and produce no concentration.

This is a classic no-intercept situation for a quadratic calibration. In this case, it was caused by a slight saturation in the high calibrator. For sample concentrations within the `5-2500` range specified by the calibrators, the calibration functions acceptably, but for a sample with a concentration higher than 2500, the equation gives no result. This is the problem with polynomial calibrations: there are some simple situations that can produce nonsensical results.

There are other function families that have a low variance that could be used for calibration when curvature is present in the data. One such family of functions is the Padé

approximates. The Padé approximates were developed in the 1890s by Henri Padé [195] to deal with certain problems with truncating Taylor series expansions. However, the Padé equation also approximates general power series [196], which includes the polynomial series and the quadratic equation specifically. The equation has been used in mass spectrometry for calibrating isotope dilution measurements [197].

The general form of the Padé approximate is a rational function of the form:

$$x = \frac{a_0 + a_1 x + a_2 x^2 + \cdots + a_m x^m}{1 + b_1 x + b_2 x^2 + \cdots + b_n x^n} \tag{6.8}$$

The specific Padé approximate is specified by two numbers, m and n, which represent the orders of the polynomials in the numerator and denominator. The approximate for the quadratic is $m = 1$ and $n = 1$ or the Padé[1,1]. Replacing the b_1 term with the a_2 term from the quadratic the Padé[1,1] takes the form:

$$x = \frac{a_0 + a_1 x}{1 + a_2 x} \tag{6.9}$$

This equation has three parameters to fit, just like the quadratic. However, it cannot be rearranged into a form that meets all of the requirements of the least squares method to determine the coefficients [197]. To compute accurate coefficients, the Padé[1,1] equation has to be fit using a nonlinear approach such as the one provided by R's `nls()` function. The default algorithm for the `nls()` function is the Gauss–Newton method [198], which minimizes the sum of squared errors iteratively, similar to Newton's Method [196] for fitting nonlinear equations. The Gauss–Newton can be thought of as an extension of Newton's method, which doesn't require the calculation of a second derivative [199].

To fit the Padé[1,1] to my example data, first, I put the equation in a function:

```
pade_1_1 <- function(x, a0, a1, a2) {
    (a0 + a1*x)/(1 + a2*x)
}
```

Then, calculate the coefficients with `nls()`. When using `nls()` or any iterative error-minimizing method, it is important to provide a good starting guess for the parameters. Looking at the form of the Padé[1,1] in Eq. (6.9), when the a_2 parameter is `0`, then the equation is the straight line shown in Figure 6.31. Since the linear fit was roughly correct, the a_0 and a_1 found from `lm()` and setting $a_2 = 0$ seems like a good place to start.

```
pade_model <- nls(instrument_response ~
pade_1_1(conc,a0,a1,a2), data=cals,

start=list(a0=model_summary$coefficients[1],

a1=model_summary$coefficients[2],
                                a2=0.0),
                  weights = conc_weights)
```

```
pade_summary <- summary(pade_model)
print(pade_summary)
```

```
##
## Formula: instrument_response ~ pade_1_1(conc, a0, a1,
a2)
##
## Parameters:
##      Estimate Std. Error t value Pr(>|t|)
## a0  9.692e-03  2.993e-04  32.386 5.42e-06 ***
## a1  5.752e-03  3.058e-05 188.121 4.79e-09 ***
## a2 -8.722e-06  4.055e-06  -2.151   0.0979 .
```

```
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.'
0.1 ' ' 1
##
## Residual standard error: 4.896e-05 on 4 degrees of
freedom
##
## Number of iterations to convergence: 2
## Achieved convergence tolerance: 3.46e-06
```

The curvature term for the Padé[1,1] (a_2) is negative as it was in the quadratic fit. However, the shape of this equation is not a parabola, so unlike the quadratic, it does not roll over but rather continues increasing with an upper bound that looks flatter and flatter as x increases. This is a more realistic image of how the response of a mass spectrometer detection system might operate when compared to a parabola, which is implausible beyond a certain saturation point.

```
cals$pade_predicted <- as.numeric(predict(pade_model))
pade_cal_line <- as.numeric(predict(pade_model,

newdata=data.frame(conc=seq(0, 2500, 1)))
                                )
```

```
p_pade_cals <- cals |>
    ggplot() +
    geom_point(aes(x=conc, y=pade_predicted), shape=19,
color=pal$red) +
    geom_point(aes(x=conc, y=instrument_response),
shape=1) +
    geom_line(data=data.frame(x=seq(0, 2500, 1),
y=pade_cal_line),
              aes(x=x,y=y), color=pal$red) +
    ggtitle(label = "Padé[1,1] Calibration",
        subtitle=
            sprintf("Residual standard error: %0.3e on
%d degrees of freedom",
                    pade_summary$sigma,
pade_summary$df[2])) +
        xlab("Concentration") +
        ylab("Instrument Response (area ratio)") +
        theme_classic() +
        theme(plot.title = element_text(hjust = 0.5,
size=16)) +
        theme(plot.subtitle = element_text(hjust =
0.5)) +
        theme(
            axis.text=element_text(size=11),
            axis.title=element_text(size = 14),
            legend.text = element_text(size = 11),
            legend.title = element_text(size = 14)
        )

print(p_pade_cals)
```

Figure 6.35 looks quite reasonable, but as always, you need to check the residuals:

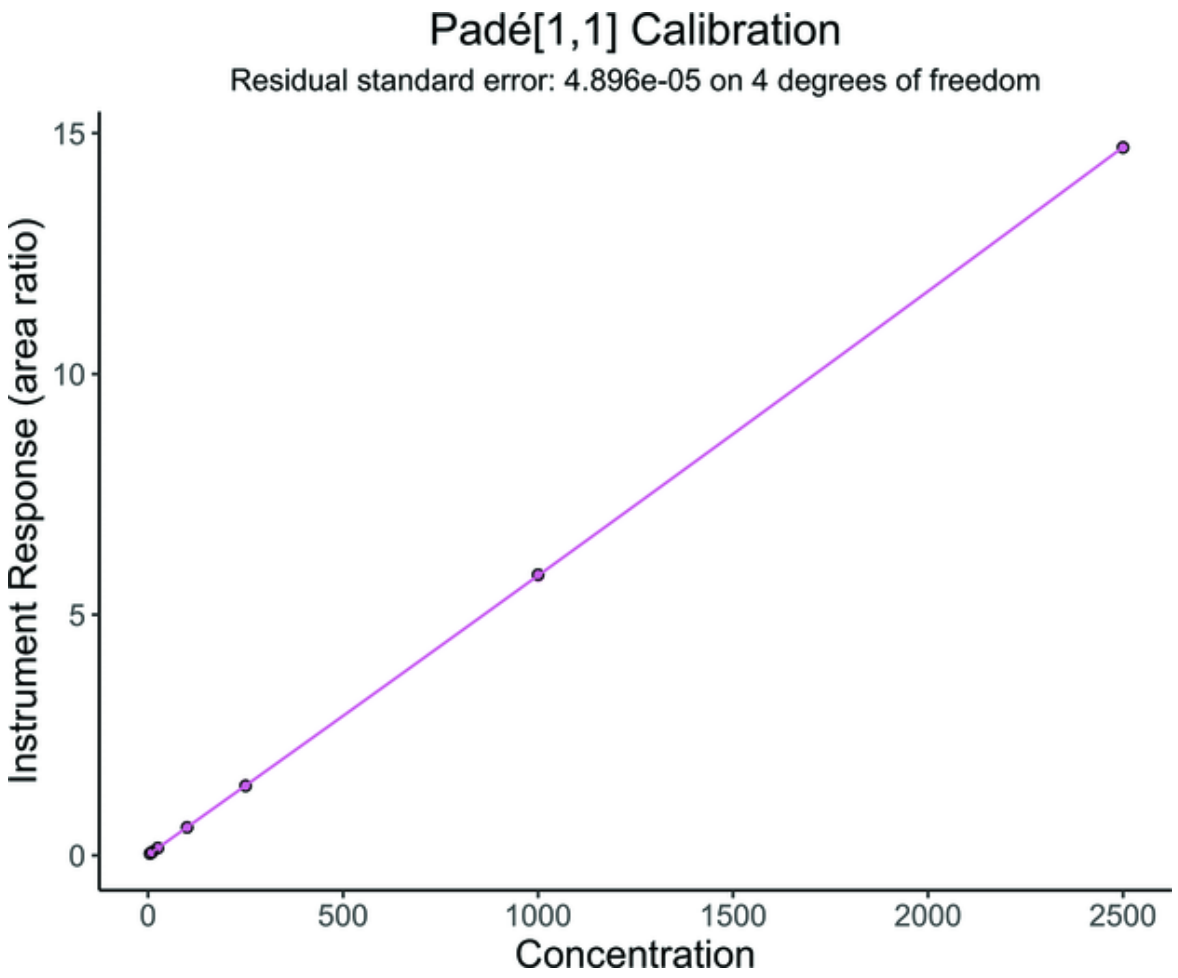


Figure 6.35 An example of a Padé[1,1] calibration fit.

```
p_pade_residuals <- cals |>
    ggplot() +
    geom_point(aes(x=conc, y=pade_summary$residuals),
shape=1) +
    geom_hline(yintercept = 0) +
    ggtitle(label = "Residuals of Padé[1,1]
Calibration",
            subtitle=parse(text="~ 1/x^2 ~ Weighting"))
+
        xlab("Concentration") +
        ylab("Residual") +
        theme_classic() +
        theme(plot.title = element_text(hjust = 0.5,
size=16)) +
        theme(plot.subtitle = element_text(hjust =
```

```
  0.5)) +
          theme(
              axis.text=element_text(size=11),
              axis.title=element_text(size = 14),
              legend.text = element_text(size = 11),
              legend.title = element_text(size = 14)
          )

  print(p_pade_residuals)
```

The residuals in Figure 6.36 look randomly distributed and exceedingly small.

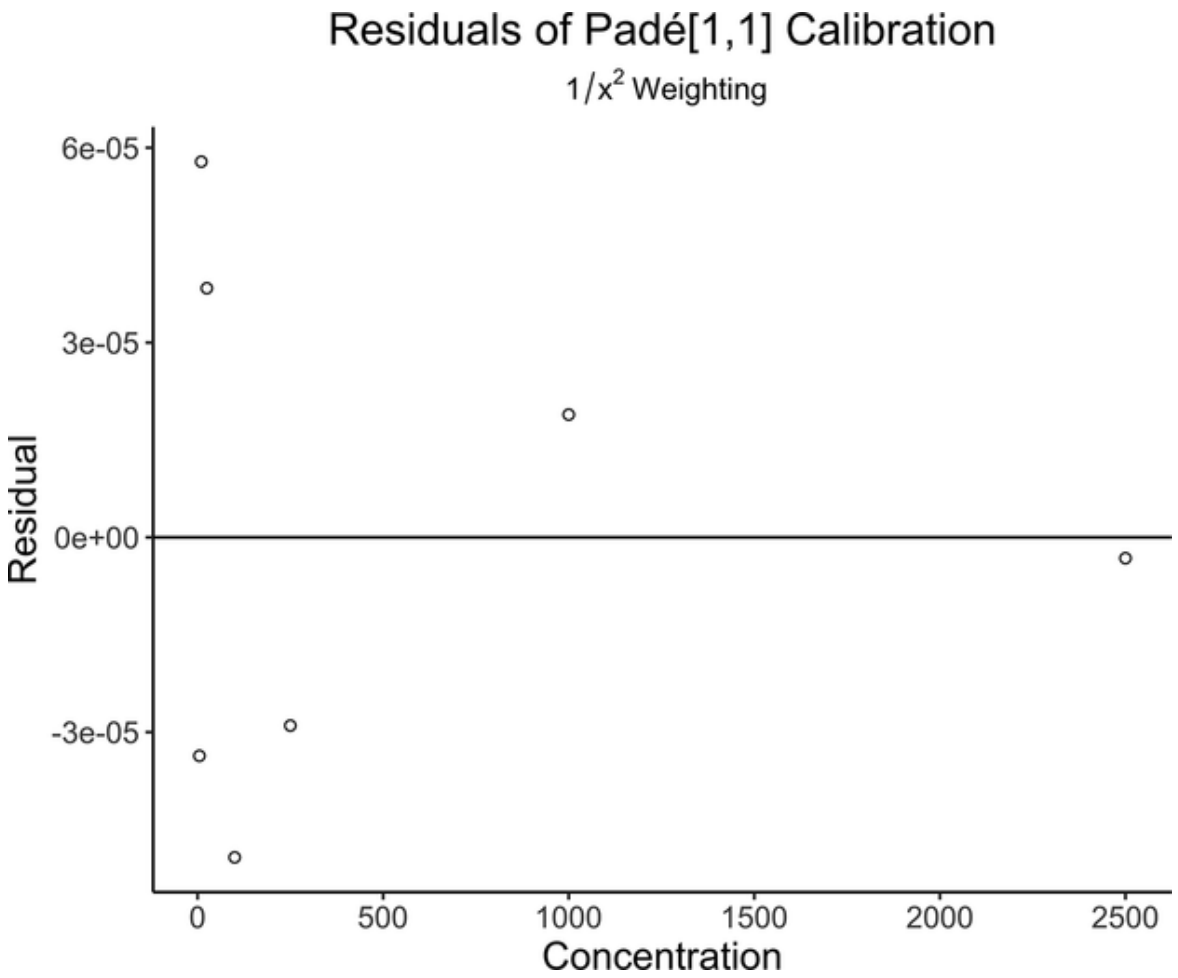


Figure 6.36 Residuals from a Padé[1,1] fit of the calibration data using $1/x^2$ weighting.

Another advantage of the Padé[1,1] equation is that its inverse has only one possible value:

$$x = \frac{a_0 - y}{a_2 y - a_1} \tag{6.10}$$

There is still the possibility of no result from the inverse. First, if $a_2 y = a_1$, then the denominator will be zero, and the result will be `Inf` in R. Also, there are a couple of ways the equation could return negative values.

And now I can put Eq. (6.10) into a function:

```
inverse_pade_1_1 <- function(y, a0, a1, a2) {
    x <- rep(NA, times=length(y))

    for(i in 1:length(x)) {
        # bail out if there will be a divide by zero
error
        if(a2*y[i] == a1) {
            next
        } else {
            r <- (a0 - y[i])/(a2*y[i]-a1)
        }
        # don't allow negative concentrations to be
returned
        if(r < 0) {
            next
        } else {
            x[i] <- r
        }
    }
    x
}
```

And use the inverse equation to obtain the back-calculated concentrations for the calibrators:

```
pade_conc <- inverse_pade_1_1(cals$instrument_response,

pade_summary$coefficients[1],

pade_summary$coefficients[2],

pade_summary$coefficients[3])
cals$pade_conc <- pade_conc
```

It’s interesting to compare the deviations between the three calibration methods:

```
pade_deviation <- 100*(1-cals$conc/cals$pade_conc)
cals$pade_deviation <- pade_deviation
cals |>
    dplyr::select(c(conc, conc_deviation,
quad_deviation, pade_deviation))
```

```
## # A tibble: 7 x 4
##    conc conc_deviation quad_deviation pade_deviation
##   <dbl>          <dbl>          <dbl>          <dbl>
## 1     5         -0.323        -0.590         -0.588
## 2    10          0.757         0.997          0.996
## 3    25          0.141         0.665          0.662
## 4   100         -1.48         -0.860         -0.863
## 5   250         -1.02         -0.504         -0.504
## 6  1000          0.460         0.314          0.322
## 7  2500          1.40         -0.0510        -0.0535
```

The Padé[1,1] and the quadratic deviations are nearly identical. However, as discussed above, the shape of the Padé[1,1] equation does not represent a parabola, which means that the equation will always return an intercept, unlike the case of the saturated calibration shown in Figure 6.34.

I'll show this using the same data I used to show the no-intercept situation for the quadratic.

```
alt_pade_model <- nls(instrument_response ~
pade_1_1(conc,a0,a1,a2),
                  data=alt_cals,

start=list(a0=model_summary$coefficients[1],

a1=model_summary$coefficients[2],
                                a2=0.0),
                  weights = conc_weights)
```

```
  alt_pade_model_summary<-summary(alt_pade_model)
  print(alt_pade_model_summary)
```

```
##
## Formula: instrument_response ~ pade_1_1(conc, a0, a1,
a2)
##
## Parameters:
##     Estimate Std. Error t value Pr(>|t|)
## a0 8.155e-03  2.367e-03   3.446   0.0261 *
## a1 5.993e-03  2.530e-04  23.690 1.88e-05 ***
## a2 1.528e-04  5.329e-05   2.867   0.0456 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.'
0.1 ' ' 1
##
## Residual standard error: 0.0003723 on 4 degrees of
freedom
##
## Number of iterations to convergence: 6
## Achieved convergence tolerance: 1.074e-06
```

```
  alt_cals$pade_predicted <-
  as.numeric(predict(alt_pade_model))
  alt_pade_line <- as.numeric(predict(alt_pade_model,

  newdata=data.frame(conc=seq(0, 2500, 1))))
```

Figure 6.37 shows how this calibration responds to the saturated calibrator:

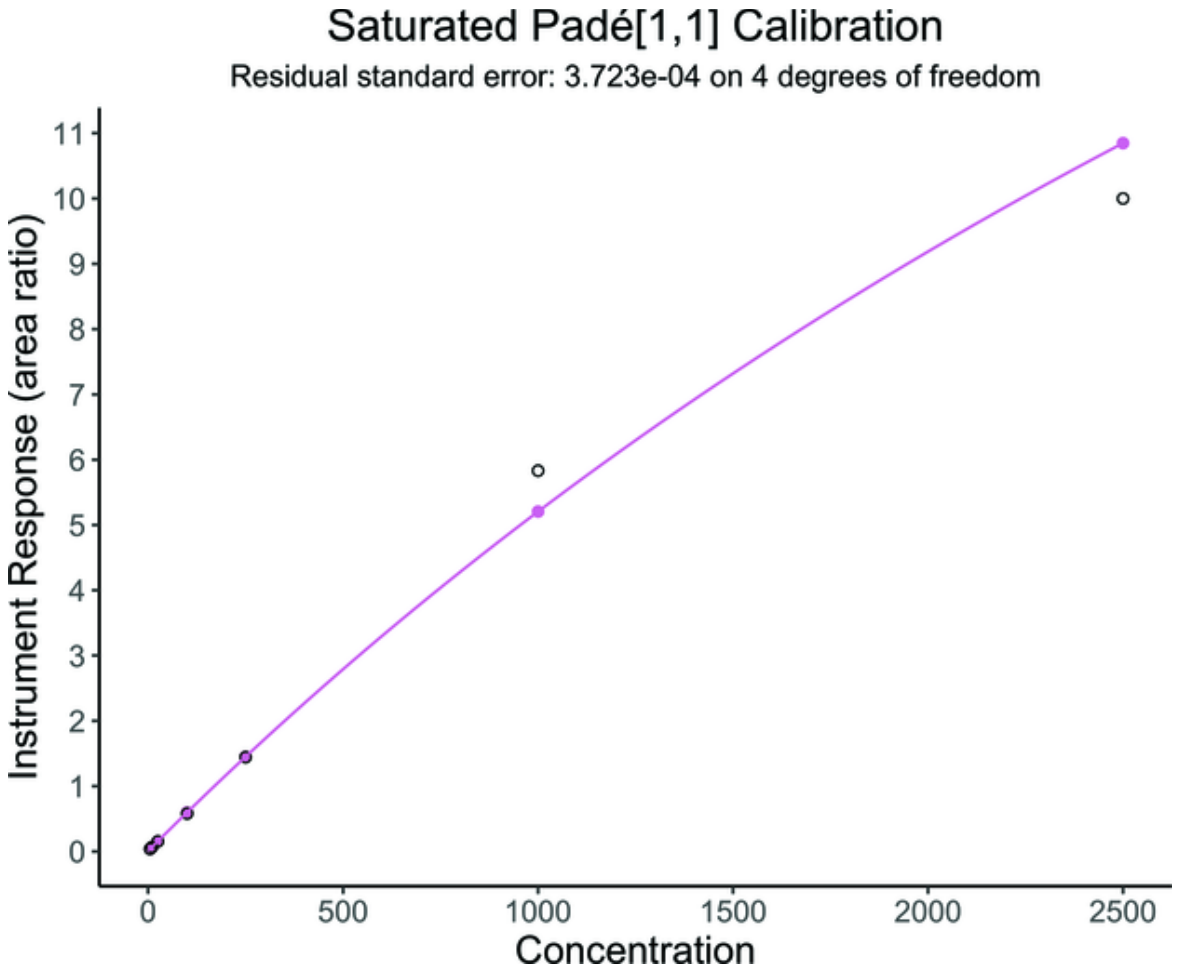


Figure 6.37 An example of a Padé[1,1] calibration fit of a saturated calibrator.

```
p_alt_pade_cals <- alt_cals |>
    ggplot() +
    scale_y_continuous(n.breaks=10) +
    geom_point(aes(x=conc, y=pade_predicted), shape=19,
color=pal$red) +
    geom_point(aes(x=conc, y=instrument_response),
shape=1) +
    geom_line(data=data.frame(x=seq(0, 2500, 1),
y=alt_pade_line),
              aes(x=x,y=y), color=pal$red) +
 ggtitle(label = "Saturated Padé[1,1] Calibration",
        subtitle=
            sprintf("Residual standard error: %0.3e on
%d degrees of freedom",
```

```
                      alt_pade_model_summary$sigma,
                      alt_pade_model_summary$df[2])) +
        xlab("Concentration") +
        ylab("Instrument Response (area ratio)") +
        theme_classic() +
        theme(plot.title = element_text(hjust = 0.5,
size=16)) +
        theme(plot.subtitle = element_text(hjust =
0.5)) +
        theme(
            axis.text=element_text(size=11),
            axis.title=element_text(size = 14),
            legend.text = element_text(size = 11),
            legend.title = element_text(size = 14)
        )

print(p_alt_pade_cals)
```

Calculate the concentrations:

```
alt_pade_conc <-
inverse_pade_1_1(alt_cals$instrument_response,

alt_pade_model_summary$coefficients[1],

alt_pade_model_summary$coefficients[2],

alt_pade_model_summary$coefficients[3])
alt_cals$alt_pade_conc <- alt_pade_conc
```

And then compute the deviations:

```
alt_pade_deviation <- 100*(1-
alt_cals$conc/alt_cals$alt_pade_conc)
alt_cals$alt_pade_deviation <- alt_pade_deviation
alt_cals |>
    dplyr::select(c(conc, alt_quad_deviation,
alt_pade_deviation))
```

```
## # A tibble: 7 x 3
##    conc alt_quad_deviation alt_pade_deviation
##   <dbl>              <dbl>              <dbl>
## 1     5              0.655              0.651
## 2    10             -0.270             -0.305
## 3    25             -1.97              -1.98
## 4   100             -3.37              -3.16
## 5   250             -1.16              -0.527
## 6  1000             11.0               12.4
## 7  2500             -8.72             -11.7
```

As with the quadratic calibration, the Padé[1,1] calibration passes inside the 15% deviation limit for the top two calibrators. You can see that the quadratic seems to have a slightly higher variance at the high end, giving slightly lower concentration deviations for the top two calibrators. However, the back-calculated concentration for even an exceptionally high instrument response will still return a real value.

```
high_response <- 20

inverse_pade_1_1(high_response,

alt_pade_model_summary$coefficients[1],

alt_pade_model_summary$coefficients[2],

alt_pade_model_summary$coefficients[3])
```

```
## [1] 6804.987
```

This result may be far outside the reporting range as defined by the lower and upper calibrators. It is, however, a useful number that can be used, for example, to simply convert to >2500 or possibly used to estimate how much a sample needs to be diluted to bring it into a reportable range. It would be questionable to use the number as a reportable concentration

for the sample unless experiments are done to show that very high concentrations don't have detector saturation issues that would have changed the curvature term in the equation (a_2). Finally, this real, positive value eliminates the need to decide what to do with a no-intercept result.

I have shown three ways to perform concentration calibration for a quantification experiment using LC-MS data: linear, quadratic, and the Padé[1,1] approximate using both the `lm()` and `nls()` functions. I have also shown how to gauge the success of the calibration process by monitoring metrics of the fit and checking for intercept problems. I also showed how to back-calculate the concentrations of the calibrators to check the accuracy of the fit.

The real test of the performance of a quantification assay is the accuracy and precision of measured QC samples that also have known concentrations. In the next section, I will show how to use the calculated concentrations of QC samples to confirm the calibration is performing correctly and the measurement is under control.

6.6 Quality Control

There are many possible points of failure in liquid chromatography with tandem mass spectrometry (LC–MS/MS) quantitative assay. The LC column is performing a chemical separation that is intended to introduce molecules with identical chemical formulas and thus *identical precursor m/z* values and identical fragment ions with *identical product m/z* values (*completely isobaric compounds*) to the mass spectrometry source at *different times*. The LC performs other tasks, too, including allowing high-concentration compounds that are very water soluble (proteins and sugars, for example) to exit a reverse-phase column long before the analyte of interest arrives at the mass spectrometer. In fact, the early part of the LC run is normally diverted to waste to

prevent these compounds from contaminating the instrument. However, there are other compounds that can elute at nearly the same time as the analyte of interest and either increase the ionization efficiency of the target analyte (ion enhancement) or decrease the efficiency (ion suppression). These events are usually normalized by the IS compound. Any enhancement or suppression of the target analyte will enhance or suppress the IS, and the area ratio will remain the same. It is a good idea to monitor the recovery of the IS across a single run to make sure that there is no trend in the area of the IS peak.

Another important consideration is the performance of the LC separation. If something were to shift in the separation timing, a completely isobaric compound, which normally has a different retention time than the target compound, can appear in the expected time of the target, giving a false measurement based on mistaken identity. A second product ion (qualifier) is normally selected to act as a check on the *identity* of the measured compound to address this possibility.

6.6.1 Compound Identification with Ion Ratios

If a compound is isobaric in both the precursor and product ions, it is very helpful to select a third product ion with a different intensity response between isobaric compounds. Since both the quantifier ion and the qualifier ion are derived from the precursor molecule, it means that they must have the same chromatographic timing. This is because the two ions were formed in the mass spectrometer *after* they exited the chromatography column. If chosen with care, the ratio of the areas of the quantifier and qualifier ions will remain constant independent of concentration. So if a molecule shifts into the target molecule time range, which is capable of generating the quantifying fragment ion, and its qualifier ion area ratio does not match what was found in validation

experiments or even in the calibrator samples, the identity of the molecule is uncertain, and its concentration cannot be reported. This is especially important in medical and forensic toxicology measurements, where the identity of the target analyte must be as certain as possible. It is common in some areas of testing for there to be two or more qualifying ions to make certain the measured analyte has the expected *identity*.

The ratio of the quantifier peak area to the qualifier peak area, often simply called the *ion ratio*, is a powerful tool for determining compound identity. For the example data used in the calibration example above, the expected ion ratio for this batch can be estimated from the calibrators. Since the relative intensities of fragment ions are related to the amount of internal energy deposited into the precursor ion leading to fragmentation, there are many instrumental factors that can affect the ion ratio, and they are normally estimated for a given contiguous run or batch, just as the calibration is normally performed for every run or batch.

```
result_table$ion_ratio <-
    result_table$area_Quant / result_table$area_Qual

cals$ion_ratio <- result_table |>
    dplyr::filter(type=="cal") |>
    dplyr::pull(ion_ratio)

cals |>
    dplyr::select(c(conc, ion_ratio))
```

```
## # A tibble: 7 x 2
##    conc ion_ratio
##   <dbl>     <dbl>
## 1     5      4.13
## 2    10      3.97
## 3    25      3.80
## 4   100      3.69
## 5   250      3.62
```

```
## 6  1000      3.67
## 7  2500      3.75
```

As expected, the area ratios of the quant and the qual are nearly constant across the measurement interval. Sometimes, there is more variability for low-concentration peaks, which could affect what tolerance is allowed for unknown samples at different concentrations.

```
  sprintf("Ion Ratio - Mean: %0.4f SD: %0.4f",
          mean(cals$ion_ratio),
          sd(cals$ion_ratio))
```

```
## [1] "Ion Ratio - Mean: 3.8040 SD: 0.1849"
```

Many laboratories use a tolerance limit on ion ratio of 20% depending on the application, and sometimes a lab will change its tolerance based on the relative areas of the quant and the qual. In this assay, the tolerance in the calibrators is less than 10% based on a 95% confidence interval (2SD).

This expected value can now be used on the QC samples to confirm identity in addition to checking the concentrations.

6.6.2 Evaluation of Quality Control Samples

In the example data, there are four QC samples. Each has an expected concentration but also has a standard deviation based on replicate testing.

```
  qc <- result_table |>
      dplyr::filter(type=="qc") |>
      dplyr::select(c(conc, instrument_response,
  ion_ratio))
```

A quick look at the ion ratios shows that the QC samples are well with expectations based on the calibrators:

```
  qc$ion_ratio_deviation <- 100 * (1 -
  mean(cals$ion_ratio)/qc$ion_ratio)


  qc |>
      dplyr::select(c(conc, ion_ratio,
  ion_ratio_deviation)) |>
      dplyr::arrange(conc)
```

```
## # A tibble: 4 x 3
##    conc ion_ratio ion_ratio_deviation
##   <dbl>     <dbl>               <dbl>
## 1  129.      3.71               -2.59
## 2  145.      3.89                2.15
## 3  602.      3.69               -2.99
## 4 1204       3.74               -1.62
```

All of the QC ion ratios are within 3% of the mean value from the calibrators. That is an excellent indication that the QC samples are the correct molecule and experienced the same internal energy deposition as the calibrators, so the estimated value for the ion ratio can be confidently applied to the unknown sample.

A Student's t-test can be used with the null hypothesis that the two means (the QCs and Calibrators) are the same:

```
  t.test(qc$ion_ratio, mu=mean(cals$ion_ratio),
  conf.level = 0.95, alternative="two.sided")
```

```
##
##  One Sample t-test
##
## data:  qc$ion_ratio
## t = -1.0315, df = 3, p-value = 0.3782
## alternative hypothesis: true mean is not equal to
3.804022
## 95 percent confidence interval:
##  3.616882 3.899546
```

```
## sample estimates:
## mean of x
##  3.758214
```

The p-value for the test is so high that the alternative hypothesis is rejected, and the ion ratios of the QCs can be assumed to be drawn from the same population as the calibrators.

Now, I can move on to checking the back-calculated values of the QC samples. Besides the mean value of the different QC sample concentrations, the standard deviations from multiple runs are given along with dilution factors. The dilution factors are needed to correct the back-calculated concentrations to their predilution values, and the SD values can be used to determine if a QC sample value is close enough to the expected value to trust any unknown value that will be reported.

```
sample_dilution <- c(2, 1, 1, 2)

conc_sd <- c(17.4, 9, 72.2, 72.3)

qc$conc_sd <- conc_sd
```

And now, calculate the concentrations for the QC sample using all three calibrations and correct for the dilution of the QC samples.

```
qc$lin_calc_conc <- ((qc$instrument_response -
m$coefficients[1]) /
                        m$coefficients[2]) *
sample_dilution

qc$quad_calc_conc <-
inverse_quad(qc$instrument_response,
                           q$coefficients[1],
                           q$coefficients[2],
                           q$coefficients[3]) *
sample_dilution

qc$pade_calc_conc <-
inverse_pade_1_1(qc$instrument_response,

pade_summary$coefficients[1],

pade_summary$coefficients[2],

pade_summary$coefficients[3]) * sample_dilution
```

Using QC concentration standard deviations, I can compute the upper and lower bound for the calculated concentration at the 2SD confidence interval:

```
qc$lower_conc <- qc$conc - 2 * qc$conc_sd
qc$upper_conc <- qc$conc + 2 * qc$conc_sd
```

Now compare the calculated concentrations with the ranges expected from the mean and standard deviation of the QC samples:

```
qc |>
    dplyr::select(c(conc, lower_conc, upper_conc,
                    lin_calc_conc, quad_calc_conc,
pade_calc_conc )) |>
    dplyr::arrange(conc)
```

```
## # A tibble: 4 x 6
##    conc lower_conc upper_conc lin_calc_conc
quad_calc_conc pade_calc_conc
##   <dbl>      <dbl>      <dbl>         <dbl>
<dbl>          <dbl>
## 1  129.       111.       147.          152.
153.           153.
## 2  145.       110.       180           180.
181.           181.
## 3  602.       458.       747.          730.
733.           733.
## 4 1204       1060.      1348.         1314.
1309.          1309.
```

The results show that using the equations that fit the calibrators best, the lowest two QCs are out of specification for this batch. Using the linear equation, three out of the four QCs pass. However, it is usually considered a problem when all of the calculated concentrations for the QC samples are above (or below) the stated concentration of the QC. This is almost always a bad sign, and even though there are ways to argue that this batch should pass, the QC samples suggest that despite an excellent calibration, the unknowns are at risk of being incorrectly calculated.

6.7 Summary

In this chapter, I have covered many important aspects of working with chromatographic data from mass spectrometers. While the examples focused on atmospheric pressure ionization from LC–MS and MS/MS instruments, the techniques can be applied to many types of mass spectrometers. The methods can be most immediately applied to gas chromatography systems where peak picking, integration, filtering, and quality control are nearly identical. The methods for computing baselines, picking peaks, and noise reduction can also be directly applied to profile-mode

full-scan mass spectra collected by any type of mass separator.

I also stressed the importance of quality control. There is a serious problem with reproducibility in the application of mass spectrometry to biological systems. This problem can only be addressed by correctly assessing the quality of the data being analyzed and by ensuring that quality controls, as shown in this chapter and Chapter 5, are used to ensure that results are not only statistically significant but have sufficient effect size to be reproduced by other laboratories.

Chapter 7
Machine Learning in Mass Spectrometry

7.1 Introduction

There are many ways to use machine learning in mass spectrometry. In this chapter, I will introduce machine learning ideas and methods that can help with data analysis beyond basic statistical inference and tests. Since machine learning is a vast field of study, I can only give a glimpse of what is possible and describe the main framework used to perform machine learning tasks in R. In this chapter, unlike earlier sections, the primary source of machine learning packages is a metapackage (like the `tidyverse`) called `tidymodels` [200] available from Comprehensive R Archive Network (CRAN).

I use the phrase *machine learning* to mean computing (learning) the parameters of a model from data. You can visually learn from data with your eyes, and as discussed in Chapter 4, this is an essential step in data analysis. You can also use statistics to learn from data. If your goal is to predict a numerical value related to some new observation, the main statistical tool is *regression*. Regression is thought of as a mainstream statistical tool but is also an essential part of machine learning. Tasks involving predicting the category or class of a new observation is called *classification*, which, while using some of the same tools as regression, is what most people associate with the phrase machine learning. Although there are many other types of machine learning (e.g. reinforcement learning, and searching for optimum

paths or plans), this chapter will demonstrate algorithms for performing regression and classification.

Another primary consideration for machine learning algorithms is whether the data includes the actual value of what is being predicted. In regression problems like calibration, standards have a known quantity, and the algorithm uses the true value of the outcome to calculate the model's parameters. Fitting parameters of a model based on knowing the correct answer is called *supervised learning*. The algorithm makes a prediction, and the prediction error can be measured and used to correct the model's behavior. In classification, mistakes can be assessed in terms of the probability of predicting the correct class of an observation, which can be used in many ways. If a threshold probability is defined for a correct class, then false positives and false negatives are a strong indicator of performance. The probabilities assigned to class outcomes can also be used in more sophisticated ways, which will be covered in Section 7.6. Classifier algorithms can always be extended to observations that come from many classes.

When the truth about an outcome (which class an observation belongs to) is unknown, you are left with the techniques of *unsupervised learning*. A measure of similarity between observations determines class membership. Two observations that are very similar in terms of their descriptors are thought of as being "close" in the parameter space. This idea can be used to create a small set of predictors from a larger set, which causes similar observations to be grouped even without knowing group membership. Another way to group observations is by using their similarity directly in one of many clustering algorithms.

In the `tidyverse` view of data organization, *observations* are stored in rows of a table, and *features* are stored as columns. In machine learning, the goal is to use past observations to make a prediction about a new observation based on what

can be learned from the features (also called variables, predictors, or dimensions) of existing observations using a model. If the model can correctly make predictions, then the way the model works can be used to understand some aspects of the populations of possible observations. In this way, machine learning and statistics are, again, highly overlapped, because any set of observations will always represent only a sample of the population. When generalizing from a sample to a population, all of the tools of statistical inference are needed to determine the reliability of the generalization.

In statistical inference, distance measurements are central to building models. Therefore, distance and the equivalent concept of similarity play a critical role in machine learning. For supervised learning, the distance between the predicted value and the truth, often called the error or loss, is used to correct model parameters. In unsupervised learning, the distance between the descriptors of an observation, which is equivalent to the similarity of two observations, is used to place observations into groups or clusters.

Some basic issues arise when you try to measure distance using features:

- Numeric features may be represented in different units.
- Features could have wildly different numerical scales, which makes any simple idea of “distance” very hard to calculate.
- Features could be categorical and have no numerical order, such as a color: “red,” “green,” or “blue.”
- Some of the observations may be missing the values for some features.

These and other aspects of features have to be evaluated and addressed, based on the goals of the analysis, what machine

learning algorithms might be used, and how distance (or error) is measured.

To use data to build models in a reliable and reproducible way, it is always helpful to first organize your data into a tidy format and understand any feature issues. Making data tidy does not mean you must follow overly strict data organization policies like keeping tables free from duplication, as one would do for a relational database. The algorithms discussed in this chapter are primarily used for tabular data, which is the most common format for data obtained from mass spectrometry. Unless your data are in rectangular tables, it will be hard to ensure that you will get sensible results from a machine learning algorithm. Further, once organized, it is easier to assess the issues of scale, data type, and completeness listed above.

Data preparation tasks can be complicated, repetitive, and tedious to debug. Therefore, in this chapter, I will show how to keep things organized using `tidyverse` packages and address feature data preparation steps needed for unsupervised and supervised learning using the `tidymodels` framework [134, 201].

7.2 Tidymodels

The `tidymodels` metapackage is a collection of R packages designed to work together to make it easier to build reproducible machine learning pipelines. The `tidymodels` packages work by composing pipelines using specifications for data preparation, modeling, and testing. The `tidymodels` pattern is to build a *recipe* for data preparation steps, a *model specification* for machine learning algorithms and hyperparameters, and then combine them into a *workflow*, which is applied to data.

Like with the `tidyverse`, the `tidymodels` framework uses a core set of packages to handle each phase of modeling [202]:

- `rsample` provides the functions for data splitting, data sampling, and resampling methods like *cross-validation*.
- `parsnip` allows machine learning models to be specified using a unified interface so that different models can be handled in a consistent fashion.
- `recipes` is an interface to preprocessing tools that effectively brings data preparation *inside* the modeling process, rather than something that is done before modeling starts.
- `workflows` organizes preprocessing, modeling, and post-processing into a single operation that can be executed reproducibly, on existing and new data sets.

Other packages like `tune`, `yardstick`, `broom`, and `dials` are automatically loaded when you load the `tidymodels` package. They each help with different modeling and model evaluation tasks. There are also other specialized packages, like the `infer` package used in Section 5.3.2 to perform hypothesis testing and the `tidyclust` package used in Section 7.4.1. Packages outside the `tidymodels` core set are not loaded automatically, so must be installed and loaded individually.

The steps needed to prepare data for a particular analysis can be organized into recipes and used whenever data of that type is encountered. Models use a uniform interface, so changing algorithms or model configurations simply requires building a new model specification and inserting it into the workflow. Programming in this way can initially feel awkward because sometimes the objects created by a step are complex, deeply nested lists. However, like using pipes (`|>`) and the functional programming approach, with a little practice, the workflow concept of `tidymodels` will make your modeling easier to write and read and easier to get right and

keep correct, even as things change with the data you are analyzing.

7.3 Feature Conditioning, Engineering, and Selection

I briefly mentioned that features often need to be prepared in some way before models can be fit and tested. Three main tasks are performed on features before modeling: feature conditioning, feature engineering, and, in some cases, feature selection. This section will be a brief introduction to a deep subject. More technical details can be found in Kuhn and Johnson's excellent text, *Feature Engineering and Selection* [203].

Feature conditioning cleans up data before it is used in a computation. Tasks include dealing with missing values, scaling data from widely different value ranges or units, centering the data so that the mean is zero, and other conditioning, such as log transformations. Feature engineering, or extractions, usually refers to creating new features from existing features provided in the data. In Chapter 4, the feature `ion_ratio` was used in Section 4.2 to analyze codeine and oxycodone data. If `ion_ratio` had not been in the data file provided, it could be computed as a feature engineering step from the quantifier and qualifier peak areas. In the `opioids_peaks.csv` file, there is a feature called `response`, computed as `quant_area/IS_area`. The IS areas are not given in the file, but using `IS_area = quant_area/response`, the IS value could be computed as a new feature and used in further analysis. In Chapter 6, many new features of a chromatography trace were computed, ranging from peak features to the frequency content of peaks and traces. Mass spectrometry experiments often involve collecting many features from relatively few observations. The number of features could grow when the measured

features are combined with any engineered features. A large number of features and a few observations create a situation where the number of features must be reduced to fit a model. Highly correlated, noisy, or low information content features can often be removed from the data set to improve model performance.

7.3.1 Conditioning Data

In this section, I will use data from an assay that quantified benzodiazepines. The data set in `benzos_small_1000.csv` is similar to the opioid data used in Chapter 4. As expected, there are clear indications that some features are on different scales.

```
benzos_msdata <- read_csv(file.path("data",
"benzos_small_1000.csv")) |>
  mutate_if(is.character, as.factor)
```

As will be discussed more in Section 7.3.2.2, after the data is read, columns containing strings are converted to *factors* using the `mutate_if(is.character, as.factor)` statement.

```
summary(benzos_msdata)
```

```
##      index           response         peakAreaQuant
peakRTQuant
##  Min.   :   1.0   Min.   :  0.00004   Min.   :     10.1
Min.   :3.176
##  1st Qu.: 250.8   1st Qu.:  0.00158   1st Qu.:     18.9
1st Qu.:3.763
##  Median : 500.5   Median :  0.00450   Median :     46.0
Median :4.071
##  Mean   : 500.5   Mean   :  1.54611   Mean   :  14723.8
Mean   :3.984
##  3rd Qu.: 750.2   3rd Qu.:  0.35718   3rd Qu.:   3526.9
3rd Qu.:4.194
```

```
##  Max.   :1000.0   Max.   :186.85807   Max.   :1763875.7
Max.   :4.278
##   peakAreaQual       peakRTQual       ionRatio
compound
##  Min.   :     10   Min.   :3.266   Min.   : 0.002014
lorazepam:412
##  1st Qu.:     83   1st Qu.:3.770   1st Qu.: 0.166986
temazepam:588
##  Median :    320   Median :4.071   Median : 0.252648
##  Mean   :  67506   Mean   :3.981   Mean   : 0.453542
##  3rd Qu.:   9536   3rd Qu.:4.185   3rd Qu.: 0.551159
##  Max.   :8113087   Max.   :4.278   Max.   :10.001550
```

Retention times (`qual_rt`, `quant_rt`) are in minutes, ranging from about `3.3` to `4.3`, and area values have units of ion counts and range from about `10` to `8e6`. Additionally, all of these values come from wildly different distributions.

```
benzos_recipe <- recipe(compound ~ ., data =
benzos_msdata) |>
    step_rm(index) |>
    step_log(all_numeric()) |>
    step_normalize(all_numeric())
```

This is a template for the `recipe()` function. The recipe is constructed from the `benzos_msdata` tibble and associates the outcome to the `compound` column and the predictors to all the other columns. The `recipe()` function only creates a *specification* for data preparation. It only uses the `data=` parameter to get the outcome and predictor names for later use. In this data, there is a column called `index`, which is simply a row number. Since the data are sorted by `compound`, this column must be removed. The `step_rm()` function removes unwanted columns from the data. Next, I perform a log transformation to get all the data into approximately the same range of values using `step_log()` and specify that this should be done to all numeric features using the selector function `all_numeric()`. Finally, the `step_normalize()` function

divides all of the values in a column by the variance of the column and subtracts the mean value. Once created, the `benzos_recipe` object can be used on any data with the same outcomes and predictors as the original data specified. At this point, no actual conditioning has occurred. It will be applied in the appropriate way later to training sets, test sets, or during cross-validation.

In supervised learning, performing simple numeric transformations like `log()` on the data in advance is not a problem. However, when it comes time to split the data into a *training set* and a *test set*, or use cross-validation, care must be taken. It is critical to perform normalization and centering transformations on each group separately to prevent information from the test data from leaking into the training data. If information about the test set or cross-validation fold is allowed to leak into the training data, the result of the test will overestimate the performance of the model and lead to overfitting and a failure to generalize. The `recipe()` function makes it easy to condition the data and keep the training and test sets and cross-validation folds isolated automatically.

The `benzos_recipe` now contains all of the data and all of the instructions on how to condition it:

```
  benzos_recipe
```

```
##
## -- Recipe----------------------------------------------
-----------------------------------
##
## -- Inputs
## Number of variables by role
## outcome:   1
## predictor: 7
##
## -- Operations
## * Variables removed: index
```

```
## * Log transformation on: all_numeric()
## * Centering and scaling for: all_numeric()
```

The `prep()` function is used to estimate all of the necessary statistics to perform all of the preprocessing steps in a recipe. To normalize, for example, the mean and standard deviation are computed and used on future datasets. When used for general preparation, `prep()` uses all the data from the recipe to perform these calculations. This allows these statistics to be used on future data sets. In a `workflow`, the calculations done by `prep()` are performed behind the scenes on the training part of the data so the steps can be applied to test data (or new data). It would be a mistake to use `prep()` on the full dataset in supervised learning before the train-test split or before cross-validation, because it would cause the information leakage mentioned above. In unsupervised learning, there is no label or test set, so `prep()` is used in combination with `bake()` using the entire dataset.

```
benzos_prep <- prep(benzos_recipe)
```

Once all of the statistics needed to create a dataset are computed, the parameters are used by the `bake()` function to create a new, preprocessed dataset. The `new_data` parameter is set to `NULL` when using a prepped recipe on the original data. To apply a recipe to new data, set the `new_data` parameter to the name of the new dataset. Like `prep()`, when a recipe is used in a `workflow`, the data creation performed by `bake()` is performed on the training data or the cross-validation training folds. The `prep()` and `bake()` functions are, however, very useful when performing unsupervised learning where there is no label or information leakage possible.

```
benzos_data <- bake(benzos_prep, new_data = NULL)
```

The output of `bake()` is a tibble that has been conditioned using `prep()` on the recipe specification:

```
  summary(benzos_data)
```

```
##     response       peakAreaQuant      peakRTQuant       
peakAreaQual
##  Min.   :-1.9904   Min.   :-1.0541   Min.   :-3.5091   
Min.   :-1.5300
##  1st Qu.:-0.8057   1st Qu.:-0.8473   1st Qu.:-0.8606   
1st Qu.:-0.8219
##  Median :-0.4733   Median :-0.5519   Median : 0.3668   
Median :-0.3602
##  Mean   : 0.0000   Mean   : 0.0000   Mean   : 0.0000   
Mean   : 0.0000
##  3rd Qu.: 0.9159   3rd Qu.: 0.8851   3rd Qu.: 0.8333   
3rd Qu.: 0.7980
##  Max.   : 2.9042   Max.   : 2.9435   Max.   : 1.1411   
Max.   : 3.0996
##    peakRTQual         ionRatio             compound  
##  Min.   :-3.0648   Min.   :-4.90612   lorazepam:412  
##  1st Qu.:-0.8210   1st Qu.:-0.46429   temazepam:588  
##  Median : 0.3802   Median :-0.04797                  
##  Mean   : 0.0000   Mean   : 0.00000                  
##  3rd Qu.: 0.8142   3rd Qu.: 0.73628                  
##  Max.   : 1.1561   Max.   : 3.65044
```

Checking the normalization shows that while the data are not necessarily expected to follow a normal distribution, the mean of each column is `0` and the standard deviation is `1`.

```
  mean(benzos_data$peakRTQuant)
```

```
## [1] -3.841372e-14
```

```

  sd(benzos_data$peakRTQuant)
```

```
## [1] 1
```

I'll cover missing value treatment in more detail later. However, the simplest way to remove `NA` values is to add `step_naomit()` to the recipe.

7.3.2 Feature Engineering

Most of the chapters in this book deal with computing features of high performance liquid chromatography–mass spectrometry (HPLC-MS) and multiple-stage mass spectrometry (MS/MS) data based on physical and chemical characteristics. All of those characteristics can become features for machine learning and fall into the class of feature engineering. All predictors shown in the `benzo_msdata` data were computed from a raw chromatographic trace as discussed in Chapter 6. Almost all the feature engineering in mass spectrometry happens before the data are assembled for a machine learning task. There are a few steps outside the realm of conditioning that are commonly used beyond the extraction and creation of features from raw data that are discussed throughout this book.

7.3.2.1 Missing Value Imputation

As mentioned above, many machine learning algorithms cannot handle missing values, and many times, simply removing observations with a missing predictor value might remove too much data. The alternative is to replace the missing value with an imputed value. Data imputation was done using statistical methods for the data values below the limit of quantitation in Chapter 6 in Section 6.3.2. In that example, missing values were imputed from the estimated distribution of the variable. Imputation can also replace missing values by finding similar observations based on non-

missing features and then typically taking the average of those values to replace the missing value. Imputation using related observations is a form of unsupervised learning called K-nearest neighbors (KNN). To demonstrate the imputation, I will randomly remove 10% of the `peakAreaQual` predictor in the `benzo_msdata` data and use `step_impute_knn()` to replace it and evaluate how similar the replacements are.

```
# Since sample() uses a random number make sure to
always generate the
# same sequence of random numbers for this example

set.seed(42)

missing_peaks <- sample(1:1000, size=100)

missing_qual_peaks <-
benzos_msdata$peakAreaQual[missing_peaks]

benzos_msdata_na <- benzos_msdata
benzos_msdata_na$peakAreaQual[missing_peaks] <- NA

vis_miss(benzos_msdata_na)
```

The values for the first `10` peaks selected to be treated as missing (Figure 7.1):

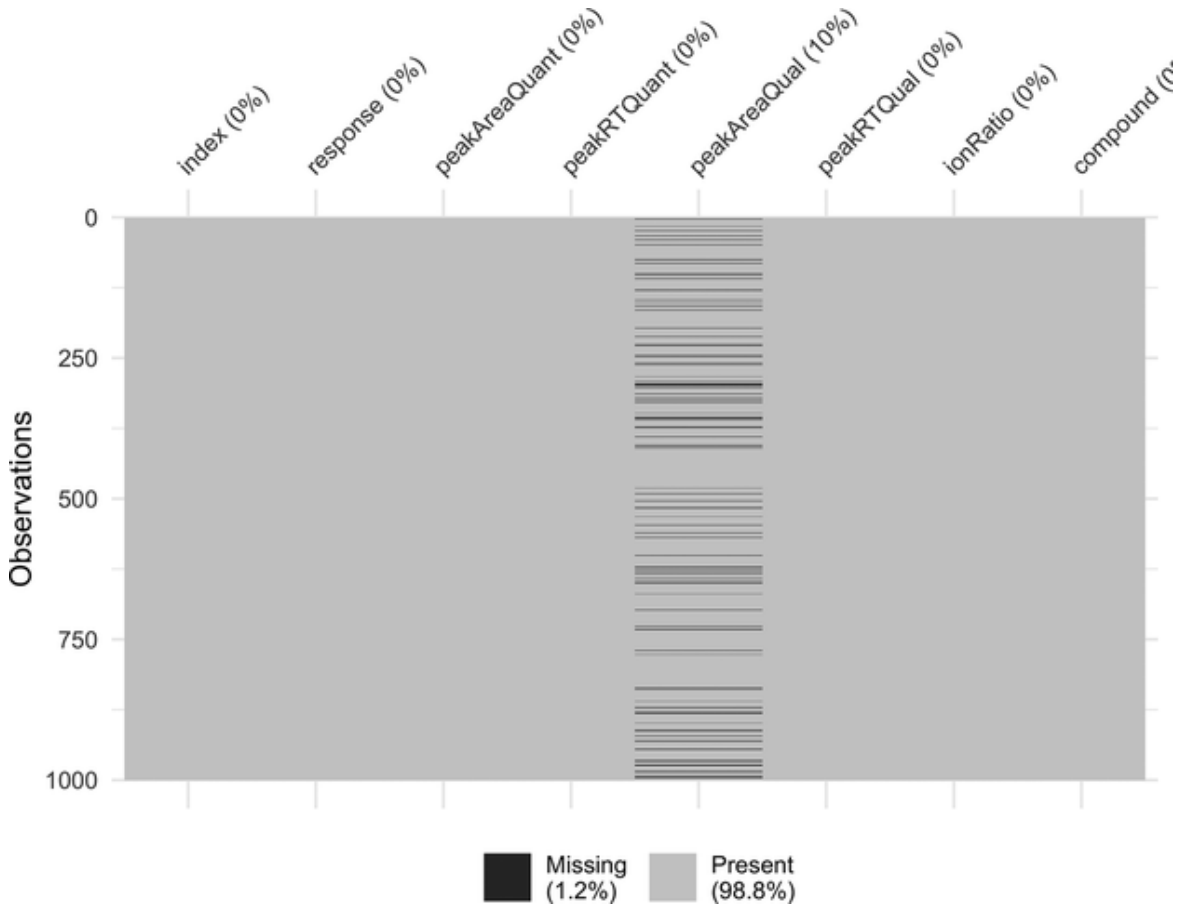


Figure 7.1 Visualization of missing values in a data set.

```
benzos_msdata$peakAreaQual[missing_peaks[1:10]]
```

```
##  [1]  28245.20249    244.57632    100.27751
23.40354    519.59353
##  [6]     15.94680     75.31189 347807.91822
110.18554     93.22363
```

Now, use imputation to replace them, prep the recipe, and then create a new tibble called `benzo_impute` from the data given in the `recipe()` function by setting `new_data=NULL` in the ‘bake() function:

```
benzos_impute <- recipe(compound ~ ., data =
benzos_msdata_na) |>
    step_impute_knn(peakAreaQual) |>
    prep() |>
    bake(new_data=NULL)
```

Now compare the imputed values to the original values:

```
benzos_impute$peakAreaQual[missing_peaks[1:10]]
```

```
##  [1] 34150.89530   764.59945    55.28039  4457.83311
2013.99192  2199.29993
##  [7]  4814.76347 92500.02977  1271.97416   631.84359
```

The actual values imputed for the first 10 missing values are not particularly close to the original values but compare the mean and standard deviation of the original data to the imputed data:

```
mean(benzos_msdata$peakAreaQual)
```

```
## [1] 67506.03
```

```
sd(benzos_msdata$peakAreaQual)
```

```
## [1] 427303.5
```

The mean and standard deviation of the imputed feature:

```
mean(benzos_impute$peakAreaQual)
```

```
## [1] 66760.1
```

```
sd(benzos_impute$peakAreaQual)
```

```
## [1] 426245.8
```

With missing values, the `mean()` and `sd()` functions won't work, which is true for many calculations performed in machine learning. Distance or error measurements often simply have no way of handling a value that is missing. The mean and standard deviation after KNN imputation are close to the original value for that feature. This saved 10% of the data from being thrown away. As mentioned before, the `recipe` package has other imputation functions, so if KNN doesn't make sense for your data, you can try others and see how they work. Remember to use imputation in a recipe so that the parameters needed for imputation are computed on the training set and then applied to the test set and new data to prevent information leakage.

The `tidymodels` package `recipe` includes many other imputation methods, and you should determine which approach best fits your application.

7.3.2.2 Encoding Categorical Variables

Another important feature engineering step is dealing with categorical variables used as predictors. One approach is to convert categorical features to factors. When predictors are converted into factors, they are given a numeric value, which is often all that is needed for simple machine learning algorithms.

```
  set.seed(2112)

  compound_class <- c(rep("opioid",5), rep("benzo",5),
  rep("SSRI",5), rep("NSAID",5))
  compound_present <- sample(rep(0:1,100), 20)

  drug_list <- tibble(present=compound_present,
  class=compound_class) |>
       mutate_if(is.character, as.factor)

  as.numeric(drug_list$class[drug_list$class=="opioid"])
```

```
## [1] 3 3 3 3 3
```

```
  as.numeric(drug_list$class[drug_list$class=="benzo"])
```

```
## [1] 1 1 1 1 1
```

Using the `mutate_if()` function with the `is.character` selector and `as.factor()` function, converts all character type columns to factors.

The compound_class “opioid” was assigned a numeric value of 3, while the compound_class “benzo” was assigned a value of 1. In many algorithms, this will be interpreted as “opioid” having a larger value in calculations than “benzo” when there is no order to the feature `compound_class`.

Using the function `step_string2factor()` in a recipe is **not recommended** because of how levels are assigned when the recipe is applied to training and test data sets and cross-validation folds. The full set of possible levels needs to be known by the recipe so they can be assigned correct test data. According to the documentation for the `recipes` package, except for special cases, factors should be assigned prior to using the data with any `tidymodels`.

The implied ordering of a numeric factor value and its scale compared to other variables can create problems for some machine learning algorithms. A common approach to preventing these problems is to create *dummy variables* using a technique called *one-hot encoding*. In a recipe, one-hot encoding is performed using the `step_dummy()` function.

```
drug_list_dummy <- recipe(present ~ ., data =
drug_list) |>
    step_dummy(all_nominal_predictors(), one_hot =
TRUE) |>
    prep() |>
    bake(new_data=NULL)

drug_list_dummy
```

```
## # A tibble: 20 x 5
##    present class_benzo class_NSAID class_opioid
class_SSRI
##      <int>       <dbl>       <dbl>        <dbl>
<dbl>
##  1       1           0           0            1
0
##  2       1           0           0            1
0
##  3       1           0           0            1
0
##  4       0           0           0            1
0
##  5       0           0           0            1
0
##  6       1           1           0            0
0
##  7       0           1           0            0
0
##  8       0           1           0            0
0
##  9       0           1           0            0
0
## 10       1           1           0            0
```

```
0
## 11       0            0            0            0
1
## 12       1            0            0            0
1
## 13       0            0            0            0
1
## 14       1            0            0            0
1
## 15       1            0            0            0
1
## 16       0            0            1            0
0
## 17       1            0            1            0
0
## 18       1            0            1            0
0
## 19       1            0            1            0
0
## 20       1            0            1            0
0
```

When using one-hot encoding, each categorical value in the `class` column is converted into a column of its own. When the value of `class` is “opioid,” a value of `1` is put in the column, and all the other dummy variables are set to `0`. No implied order exists to the `class` values when one-hot encoding is used.

Some care should be taken when using `step_dummy()`. The default parameter for `one_hot` is `FALSE`, which will compute four dummy variables rather than five. Dummy encoding by leaving one categorical value out is the default because some algorithms (including least squares regression) have numerical problems in the underlying matrix algebra when all the dummy columns add up `1`. Leaving out a categorical variable doesn’t lose any information because if all the dummy variables are `0`, it indicates that the observation contains the left-out categorical value [134].

7.3.2.3 Principal Component Analysis

Principal component analysis (PCA) generates new features by using a linear combination of existing features to transform the data to a new coordinate system. The transformation is performed so that the new dimensions (principal components) capture the most variation in the data. PCA is a general feature engineering process that is used to reduce the number of dimensions in a dataset independent of the domain-specific feature engineering discussed in earlier chapters.

PCA is described in many places, but the book, *An Introduction to Statistical Learning: With Applications in R* [204] gives one of the clearest explanations available with examples given in R. The essential idea is that the *first principal component* is calculated by multiplying each feature value by a coefficient and then adding them together. The coefficients are calculated so that the linear combination has the largest variance. A second component can be calculated with the highest variance of all linear combinations *uncorrelated* with the first component. These create orthogonal dimensions, which allow any pair of principal components to be plotted as a 2D plot. The values of the coefficients are called *loadings* and can be used to show how much variance is captured by each component. In R, the function `prcomp()` can be used to perform PCA. To avoid information leakage, discussed throughout this chapter, the `step_pca()` function can be used in a recipe so that the method of combining features is established on the training data and applied to the test data and new data.

To show how to perform basic PCA, I will use data from the 2023 paper by Yang et al. *L-leucine increases the sensitivity of drug-resistant Salmonella to sarafloxacin by stimulating central carbon metabolism and increasing intracellular reactive oxygen species level* [205]. The study performed positive and negative ion liquid chromatography with tandem

mass spectrometry (LC-MS/MS) on samples of *Salmonella typhimurium*. A sarafloxacin-resistant strain (SAR-R) and a nonresistant strain (SAR-S) were analyzed in replicate (six biological replicates from each strain). The results were deposited in the Metabolights data repository as `MTBLS7711` for the positive ion LC-MS/MS analysis and `MTBLS7713` for the negative ion analysis.

This study used an AB SCIEX TripleTOF 6600, a high-resolution MS/MS system that uses a time-of-flight mass analyzer as the second mass analyzer. In the Metabolights repository, the raw data are available as `mzML` files. The study is described using the Investigation-study-assay (ISA) format described in Chapter 3, which includes the metabolite identification information (peak areas and chemical identification) using the metabolite assignment file (MAF) format mentioned in Section 3.4.

```
# read the positive ion metabolite profile ISA file and
clean it up

sal_pos <- read_tsv(file.path("data","MTBLS7711",
        "m_MTBLS7711_LC-
MS_positive_hilic_metabolite_profiling_v2_maf.tsv"))
```

Like in previous chapters, the first step is to organize the data according to Tidy Data principles. For the MAF format, this requires a little `pivot` gymnastics. Since the columns contain the peak areas of metabolites, they have chemical names that are mostly not syntactic. However, all this table requires is that the column names are unique. For most machine learning applications, the observations need the same label, so after getting the table correctly formatted, the observation names are changed to simply reflect group membership. In this example, there are three groups: `QC`, `SAR-R`, and `SAR-S`.

```
sal_pos_dt <- pivot_longer(sal_pos, cols =
ends_with("POS")) |>
              select(metabolite_identification, name,
value) |>
              pivot_wider(names_from =
metabolite_identification,
                          values_from = value,
names_vary = "slowest",
                          names_repair = "unique")

sal_pos_dt$name <- c("QC","QC","QC","QC",
                     "SAR-R","SAR-R","SAR-R","SAR-
R","SAR-R","SAR-R",
                     "SAR-S","SAR-S","SAR-S","SAR-
S","SAR-S","SAR-S")
```

Principal component analysis benefits from variable normalization. So, I will create a `recipe()` to normalize the data for use in PCA:

```
sal_pos_rec <- recipe(name ~ ., data = sal_pos_dt) |>
    step_normalize(all_numeric())
```

Then use `prep()` and `bake()` to get the normalized data table:

```
sal_pos_data <- sal_pos_rec |>
    prep() |>
    bake(new_data = NULL)
```

To perform PCA on the normalized data, I drop the label `name` and call `prcomp()` from the `stats` base package. Since the data is already centered, this process can be skipped (`center=FALSE`).

```
pca_sal <- sal_pos_data |>
    select(-name) |>
    prcomp(center=FALSE)
```

Throughout this book, I have used `ggplot()` directly to customize plots for mass spectrometry. The `autoplot()` function gives sensible output for most machine learning plots. Since the plot produced by `autoplot()` is a `ggplot()` object, you can customize it by adding additional layers (Figure 7.2).

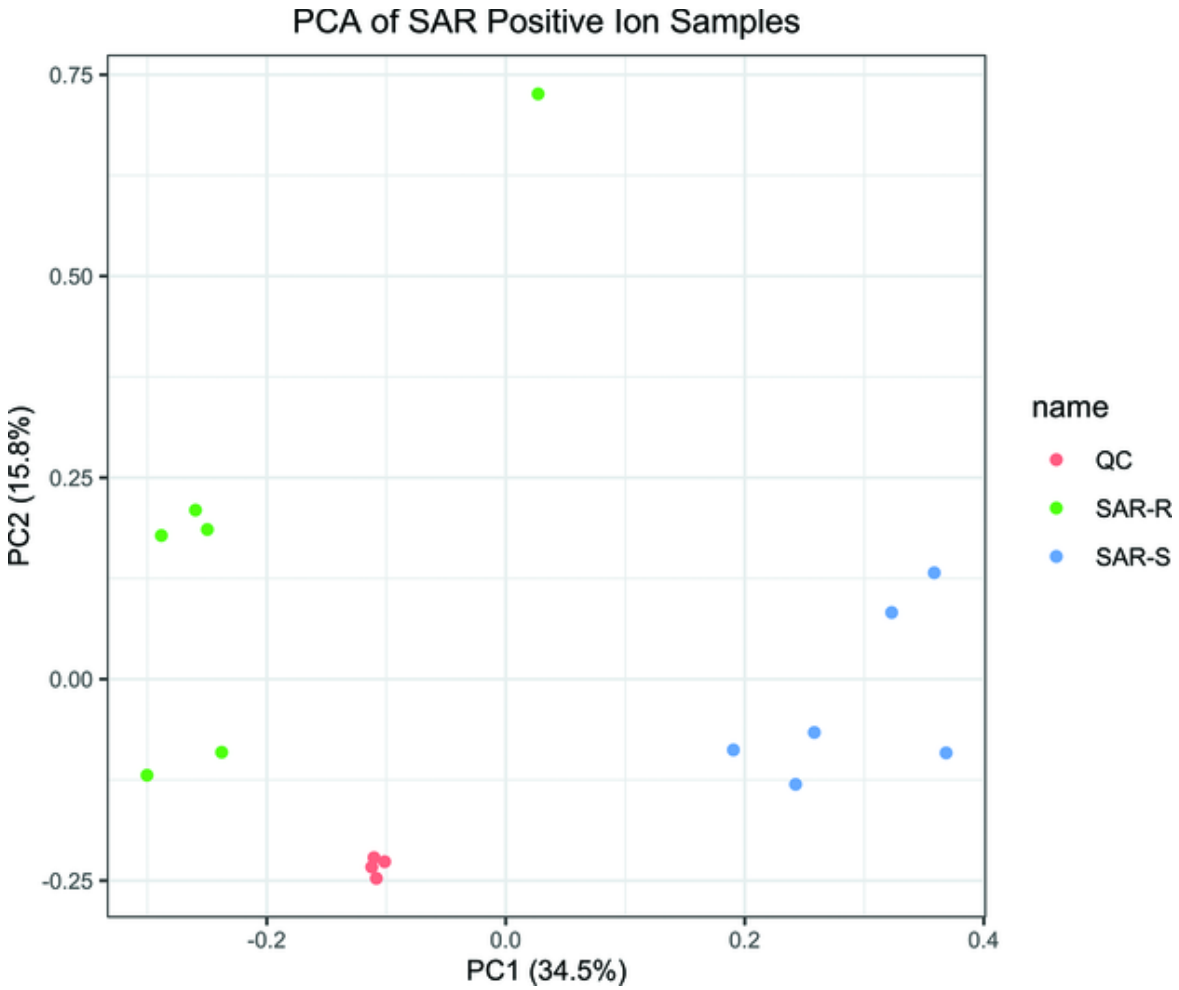


Figure 7.2 First and second principal components for the SAR positive ion samples.

```
pca_plot <- autoplot(pca_sal, data=sal_pos_data,
color="name") +
    theme_bw() +
    theme(plot.title = element_text(hjust = 0.5,
size=12)) +
    theme(plot.subtitle = element_text(hjust = 0.5)) +
    theme(
        axis.text=element_text(size=8),
        axis.title=element_text(size = 10)
    ) +
    ggtitle(label= "PCA of SAR Positive Ion Samples")

print(pca_plot)
```

The `fviz_eig()` function from the `factoextra` package can be used to see the variance explained by each principal component. The percentage of the explained variance attributed to each principle component is also called *loading*. The name *scree plot* is used because loading percentages tend to start high and then level off quickly, so the plots look like the loose rock debris (scree) that accumulates at the bottom of a mountain (Figure 7.3).

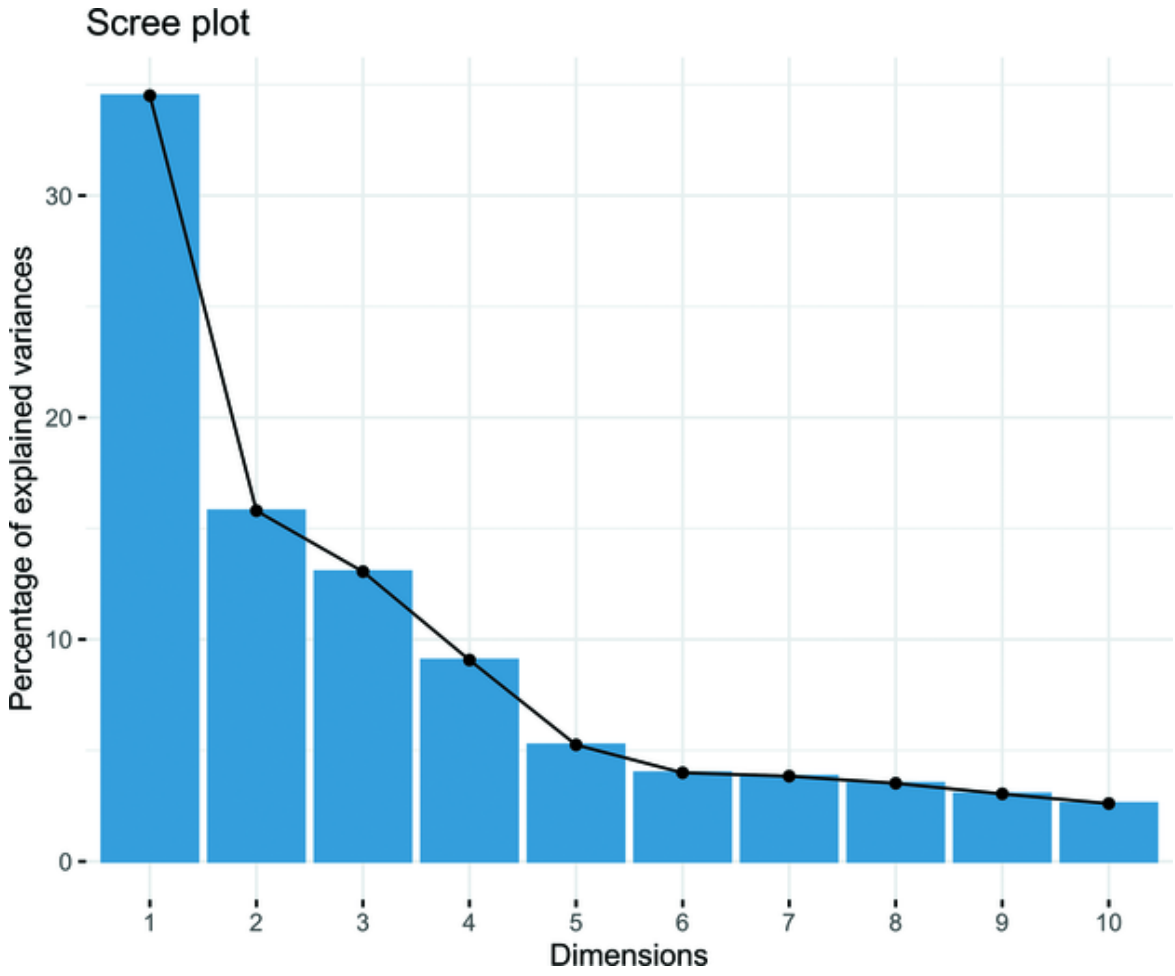


Figure 7.3 Plot of the top 10 principal components and their percent of variance explained.

```
fviz_eig(pca_sal)
```

7.3.3 Feature Selection

There are many reasons to use a subset of the features available rather than all of them when working with models. One reason that occurs often in mass spectrometry data is that there are too few observations and too many features describing them. In this case, any statistical model relating the outcome to the features runs into the common linear algebra problem of having too many variables and insufficient equations, which requires *dimensionality reduction*. Some other reasons for performing feature selection are that some features are highly correlated with each other and that some features have low information content. In many cases, building models using a subset of features can help produce better models.

Regardless of the feature selection method used, it is very important to understand that feature selection is part of modeling, not part of data preparation. Improper feature selection can lead to *label leakage*, which will be discussed more in Section 7.6. Briefly, using a recipe to perform feature selection reduces the risk of label leakage by performing the feature selection only on the training set or cross-validation training fold. Splitting data into training and testing sets will be covered in more detail in Section 7.6.2.

Feature selection can be performed in either an unsupervised or supervised manner. Examples of unsupervised feature selection are removing features with missing values, removing low-variance features, or selecting performing PCA and keeping only the components above some threshold of variance representation. Supervised feature selection uses the relationship with the label to identify the most important features. As mentioned, this must be done with care, and the best way is to use the `tidymodels` recipe approach to make sure that the features are selected from the training set and then applied to the test set and new data.

To perform supervised feature selection using the `recipes()` function, a specialized package has to be installed from GitHub as was done in Section 5.2.4. The `colino` package [206] is the latest version of `recipeseclectors` that performs feature selection based on a ranking of a metric. To install `colino`, use the command `remotes::install_github('stevenpawley/colino')`.

Going back to the benzodiazepines data, I will build a recipe to perform both supervised feature selection. The `step_select_linear()` function is told to use the magnitude of the linear relationship between the outcome (label) and all the predictors (using the `all_predictors()` selector function) and select the top four:

```
  benzos_select_recipe <- recipe(compound ~ ., data =
  benzos_msdata) |>
      step_rm(index) |>
      step_log(all_numeric()) |>
      step_normalize(all_numeric()) |>
      step_select_linear(all_predictors(),
  outcome="compound", top_p=4)

  benzos_select_recipe
```

```
##
## -- Recipe ----------------------------------------------
-----------------------------------
##
## -- Inputs
## Number of variables by role
## outcome:   1
## predictor: 7
##
## -- Operations
## * Variables removed: index
## * Log transformation on: all_numeric()
## * Centering and scaling for: all_numeric()
## Variable importance feature selection
```

The recipe output shows that it will be operating on seven predictors and one outcome:

```
  benzos_select <- benzos_select_recipe |>
      prep() |>
      bake(new_data=NULL)

  summary(benzos_select)
```

```
##  peakAreaQuant      peakRTQuant       peakAreaQual
ionRatio
##  Min.   :-1.0541   Min.   :-3.5091   Min.   :-1.5300
Min.   :-4.90612
##  1st Qu.:-0.8473   1st Qu.:-0.8606   1st Qu.:-0.8219
1st Qu.:-0.46429
##  Median :-0.5519   Median : 0.3668   Median :-0.3602
Median :-0.04797
##  Mean   : 0.0000   Mean   : 0.0000   Mean   : 0.0000
Mean   : 0.00000
##  3rd Qu.: 0.8851   3rd Qu.: 0.8333   3rd Qu.: 0.7980
3rd Qu.: 0.73628
##  Max.   : 2.9435   Max.   : 1.1411   Max.   : 3.0996
Max.   : 3.65044
##       compound
##  lorazepam:412
##  temazepam:588
##
##
##
##
```

Applying the recipe to all of the data shows of the original 6 columns, 4 were selected according to the `top_p=4` parameter to the `step_select_linear()` function.

When used correctly, both unsupervised and supervised feature selection can produce models with very good performance by reducing the number of model parameters and lowering the variance of the model. The importance of low-variance models will be discussed more in Section 7.6.1.

7.4 Unsupervised Learning

Unsupervised learning is often used in mass spectrometry to aid with exploratory data analysis (see Chapter 4). It is also the only option available when working with data that have no label or class. These algorithms use measures of distance to group-related observations. This can be done to show how data clusters, either in general or hierarchically. Clustering methods often depend on the use of PCA discussed above as the features to be grouped. The concept of related observations can also be used without dimensionality reduction for spectral comparison. For example, spectral library searching can be performed by ordering library spectra based on their similarity to a spectrum from an unknown compound.

7.4.1 Clustering

Clustering is the process of grouping observations based on how similar they are. Most clustering algorithms use Euclidian distance to determine how similar two observations are. Often, this distance is calculated in a reduced-dimensional space, like the principal components computed from an observation. The algorithm is simple to understand: each observation is placed in one of a specified number of clusters, where the *within-cluster variation* is minimized. The variation minimization step is framed as an optimization task, and many measures of variation and optimization methods are available. K-means clustering is a classic method [207] available in R as `kmeans()` from the `stats` package. The default algorithm for the optimization step is the Hartigan–Wong algorithm [208]. All K-means algorithms start by randomly assigning an observation to a cluster and then iterating until the conditions are met for convergence. Because of randomization, multiple runs on the same data can give different results. To get the same results from

multiple runs, you can use the `set.seed()` function to ensure the random numbers generated are the same for each run. Processing the data with different random seeds and combining the results can be helpful. The `kmeans()` function has a parameter `nstart`, which, if used, controls how many random sets are chosen and combined.

The `tidyclust` package also includes a `k_means()` function for use in `parsnip` models. The `k_means()` function lets you use `set_engine` in the `parsnip` model specification. The default package is `stats`, which results in the model using the `kmeans()` function. See the `tidyclust` package documentation for more details [209]. I will use the `kmeans()` function on the data from the *Salmonella typhimurium* study [205] used in Section 7.3.2.3:

```
set.seed(421)

kmeans_sal <- sal_pos_data |>
    select(-name) |>
    kmeans(algorithm = "Hartigan-Wong", centers=3)
```

Given that there were three groups (SAR-R, SAR-S, and QC), I chose three `centers` or centroids to see how well the K-means solution matched the intuition from Figure 7.2.

```
km_plot <-autoplot(kmeans_sal,
                   data=sal_pos_data,
                   frame=TRUE, frame.type='norm') +
        theme_bw() +
        theme(plot.title = element_text(hjust = 0.5,
size=12)) +
        theme(plot.subtitle = element_text(hjust =
0.5)) +
        theme(
            axis.text=element_text(size=8),
            axis.title=element_text(size = 10)
        ) +
        ggtitle(label= "K-Means Clustering SAR Positive
Ion Samples")
print(km_plot)
```

The centers shown do a good job of finding reasonable clusters (Figure 7.4). A new observation from one of the three groups is expected to fall in one of the regions defined by the 95% confidence interval of the two-dimensional normal distributions determined by each group's variance in PC1 and PC2. The `autoplot()` function calls `ggbiplot()` function automatically to generate the cluster plot object. The `ggbiplot()` function defaults to a 95% confidence interval, which can be changed by setting the `frame.level` argument to the desired level.

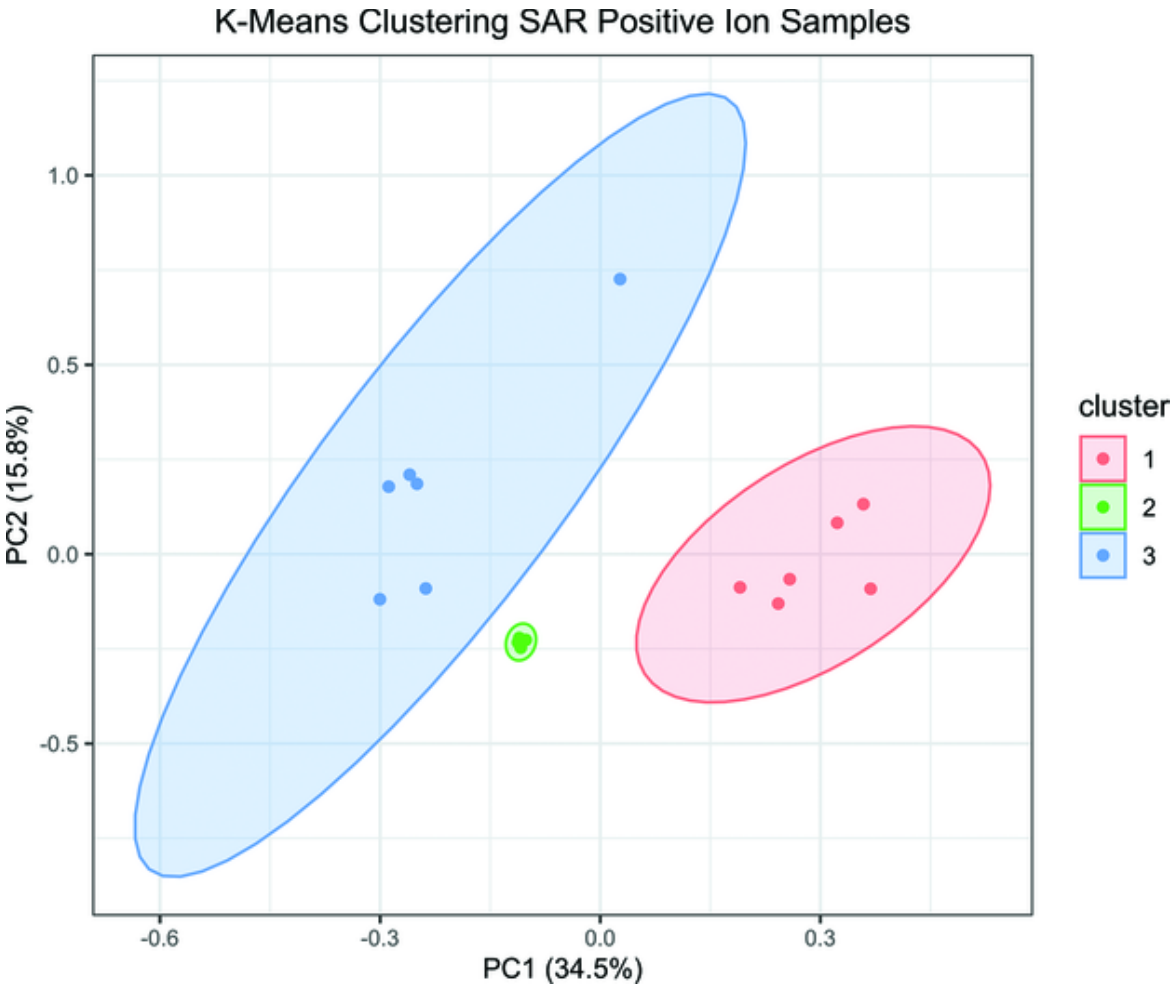


Figure 7.4 K-means clustering of the positive ion samples for the SAR-R, SAR-S, and QC samples.

7.4.2 Hierarchical Clustering

One disadvantage of the K-means clustering approach is that it requires the specification of the number of clusters, which is often unknown. Hierarchical clustering (HC) does not require telling the algorithm the number of clusters in advance. In mass spectrometry analysis, HC often appears as a combination of one or two tree-based representations of the cluster assignments, or *dendrograms*, combined with a *heatmap* representing the data associated with the observations.

Using the data from MTBLS7711 (positive ions) and MTBLS7713 (negative ions), I will show how to perform an HC analysis and avoid a common error. First, I will combine the positive and negative ion profiling data into a single data structure.

Here, I load the negative ion data and condition it the same way I did the positive ion data earlier in Section 7.3.2.3.

```
sal_neg <- read_tsv(file.path("data","MTBLS7713",
            "m_MTBLS7713_LC-
MS_negative_hilic_metabolite_profiling_v2_maf.tsv"))

sal_neg_dt <- pivot_longer(sal_neg, cols =
ends_with("NEG")) |>
    select(metabolite_identification, name, value) |>
    pivot_wider(names_from = metabolite_identification,
                values_from = value, names_vary =
"slowest",
                names_repair = "unique")

sal_neg_dt$name <- c("QC","QC","QC","QC",
                     "SAR-R","SAR-R","SAR-R","SAR-
R","SAR-R","SAR-R",
                     "SAR-S","SAR-S","SAR-S","SAR-
S","SAR-S","SAR-S")
```

The instrument responses for positive and negative ions can be combined into a single data structure, as the original publication did. However, the publication does not suggest that the two groups (SAR-R and SAR-S) can be separated using either the positive or negative ionizing compounds alone.

```
# for HC combine both positive ion and negative ion
features
# drop the extra name column and fix the remaining name
column

sal_dt <- dplyr::bind_cols(sal_pos_dt, sal_neg_dt) |>
    select(-name...264) |>
    rename(name=name...1)
```

Since HC depends on a distance calculation, normalizing the responses is critical.

```
# Normalize the entire data set by column (feature)
# These will become rows in the HC matrix

sal_dt_rec <- recipe(name ~., data = sal_dt) |>
    step_normalize(all_numeric())
```

The `recipe()` function creates a data preparation *specification*, which can then be applied to any data using the `prep()` and `bake()` functions:

```
sal_dt_norm <- sal_dt_rec |>
    prep() |>
    bake(new_data = NULL)
```

The table `sal_dt_norm` now contains rows representing samples and columns holding normalized compound intensities. Normally, HC plots are shown with the sample groups as columns matched to a dendrogram. In almost all cases where HC is used for unsupervised learning, there are many more features than observations. So plots tend to be shown as rectangles in “portrait orientation.” The table can be filtered to remove samples that are not part of the experimental design, and the label (`name`) can be removed and transposed as a matrix to orient the data to a portrait format. After transposition with `t()`, the new column names

can be added back and the row names can be removed to replace them with matrix row numbers.

```
# Drop the QC samples and the name column then make the
matrix and
# transpose to orient the clustering in the prefered
direction

sal_matrix <- sal_dt_norm |>
    filter(name!="QC") |>
    select(-name) |>
    as.matrix() |>
    t()

# set the column names to the sample names

colnames(sal_matrix) <-
    c("SAR-R-1","SAR-R-2","SAR-R-3","SAR-R-4","SAR-R-
5","SAR-R-6",
      "SAR-S-1","SAR-S-2","SAR-S-3","SAR-S-4","SAR-S-
5","SAR-S-6")
```

The combined dendrogram and the heatmap typically shown for hierarchical clustering can be produced using the `heatmaply()` function from the `heatmaply` package. The `heatmaply()` function is designed to generate hyper-text markup language (HTML) output that contains interactive graphic elements using the `plotly` package. This is very useful if your plot is part of an HTML report. A static version of an HC plot can be created by modifying the output of the `heatmaply()` function. Because the heatmap needed for the SAR data includes so many rows, there is no easy way to include the compound names associated with the rows. By generating a `ggplot` object with `heatmaply()` using the `plot_method = "ggplot"` parameter, the y-axis ticks and text can be removed with the `element_blank()` function:

```
hc_plot <- heatmaply(percentize(sal_matrix),
                     scale = "row",
                     show_dendrogram=c(FALSE,TRUE),
                     return_ppxpy = TRUE, plot_method =
"ggplot")

hc_plot[[1]]$theme$axis.ticks.y <- element_blank()
hc_plot[[1]]$theme$axis.text.y <- element_blank()
```

Then, the `arrange_plots()` function from the `heatmaply` package can be used to place the components of the HC plot together in one image that properly combines the dendrogram, heatmap, and legend:

```
heatmaply:::arrange_plots(hc_plot, widths = NULL,
heights = NULL,
                          row_dend_left = FALSE,
                          hide_colorbar = FALSE)
```

In Figure 7.5, you can see that the two strains of *Salmonella typhimurium* can be separated using all the features. However, there is nothing particularly compelling about the heatmap. Figure 7.5 differs significantly from the published figure. One reason for the difference could be that the features most correlated with the label were preselected before clustering. As I've mentioned repeatedly, preselecting features based on correlation with the outcome is a common and dangerous form of label leakage. When there are many features, some will be correlated with the label by chance. If these are selected before performing hierarchical clustering, it will appear as if some compounds strongly explain why the samples clustered together. When correlated features are preselected, the HC plot will look compelling but could represent only spurious correlations. In this example, HC does split the samples into two groups; the features measured separate the two groups; however, the explanatory power of the heatmap is weak. While I have continuously

described how label leakage is damaging to building models, this has mostly been described in the context of supervised learning, where the label is used to cheat the testing step to overstate the quality of a model when used on test data. If feature leakage had been allowed, Figure 7.5 would have shown a more drastic difference between the two classes, but it would have been an example of label leakage damaging an unsupervised learning method.

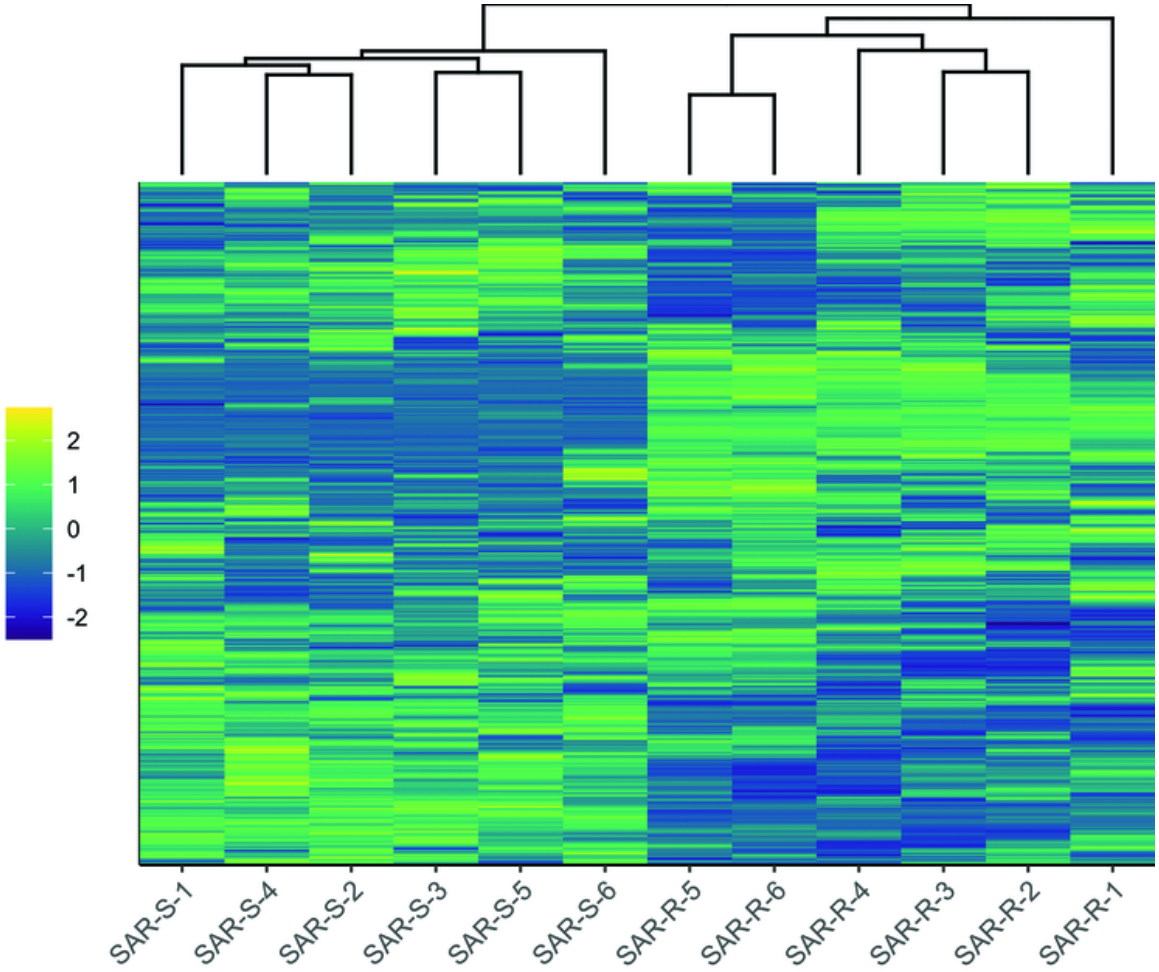


Figure 7.5 Hierarchical clustering of positive and negative features for SAR-R and SAR-S strains.

7.5 Using Unsupervised Methods with Mass Spectra

The use of unsupervised learning with spectra has a long history in the form of spectral matching and library search. Performing spectral library searches using a similarity measure has been a central element of structure identification using electron impact (EI) ionization mass spectra and matching MS/MS spectra to the theoretical spectra of peptides, as I showed in Chapter 3. In this section, I will describe various ways to estimate the similarity of two spectra in the context of library searching.

7.5.1 Measures of Spectral Similarity

Using algorithms to compute the similarity between two spectra has been used since the early days of the computerization of mass spectrometry [210, 211]. The most common approach to calculating spectral similarity is to treat the mass spectrum as a vector and apply one of the many methods for determining the distance between two vectors and applying the basic relationship between distance and similarity [212]:

$$similarity = \frac{1}{1 + distance} \tag{7.1}$$

One of the earliest and most used spectral similarity measures is the *cosine similarity* [213]. The idea is to treat a mass spectrum as a vector. If two mass spectra are binned into equal-length vectors, with the intensity values at each binned m/z value (element of the vector), then any two vectors, regardless of the number of dimensions (vector elements), will lie in a plane, and there is an angle defined between them. While there is chemical information in the relative intensities of the m/z values, it is usually recommended that the intensities be scaled, with the most logical scaling resulting in the vector having a length of 1 so that it represents a vector that points to a location on a unit

hypersphere. Other scaling methods have also been used [213], some combining m/z values to increase or decrease the importance of different parts of the mass spectrum.

With an angle θ computed between the two vectors using the dot product, the $cos\,\theta$ is a convenient similarity measure that goes smoothly between 1 (identical vectors) and 0 (orthogonal vectors). The $cos\,\theta$ measure is also called the *contrast angle*, used in many domains ranging from voice recognition to satellite and image analysis [214].

In Figure 7.6, you can see that the angle θ relates to the distance between the endpoints of the two vectors derived from a mass spectrum. The relationship can also be shown mathematically [215]:

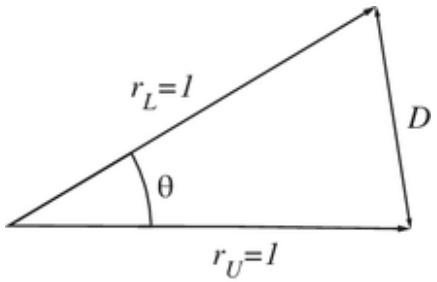


Figure 7.6 The angle θ between the normalized vector from an unknown mass spectrum (r_U), and the normalized vector from a library spectrum (r_L), is perfectly correlated with the Euclidian distance D between the endpoint locations on a hypersphere.

$$cos\,\theta = 1 - \frac{D^2}{2} \qquad (7.2)$$

Equation 7.2 means that when unit scaling is used to normalize a vector derived from a mass spectrum, the order of similarities when comparing a target spectrum to a library of known spectra will be the same with both the contrast

angle and Euclidian distance, and that the results are equivalent to using a k-nearest neighbors clustering method.

I will use the `philentropy` package [216], which implements many distance and similarity measures. I will use the low-level functions from `philentropy`, which are nominally intended to be used via a wrapper function called `distance()`. The only consequence of this choice is that you have to pay attention to what the low-level functions return since both distance and similarity measures are implemented. Of course, similarity and distance can be interconverted using Eq. (7.1).

To demonstrate how spectral comparison can be performed in R using distance and similarity functions, I will use a library from the MassBank of North America (MoNA) [217, 218]. The MoNA repository contains many different libraries that are available in multiple formats. For this example, I downloaded the LC-MS/MS Positive Mode library, which contained 99 260 LC-MS/MS spectra at the time of writing. I downloaded the library in the NIST MSP file format, a flat text format that holds spectra and chemical structure information. The file can be read using the `Spectra` package with the addition of the `MsBackendMsp` package [16] to parse the file. The manual for the NIST MS Library search program [219] contains details on the MSP file format.

To compute spectral similarities using `philentropy`, I will read the MoNA positive ion library file and tidy the data up. As mentioned by Rainer et al. in their paper [16], the MSP file format has several variations depending on who generated it, so moving from the Bioconductor data structures to tidyverse structures is, again, the first step.

```
mona_positive <- Spectra(
    file.path("large-data","mona","MoNA-export-LC-MS-
MS_Positive_Mode.msp"),
    source=MsBackendMsp())
```

Since I'm interested in small molecules and want manageable vectors, I will remove library entries for compounds with a molecular weight (`MW`) greater than 500 Da using the base R selection syntax for the S4 object returned by `Spectra()`. Also, since not all library entries are equally complete, I'll remove any entries that don't have a precursor m/z value (`PrecursorMZ`) in the entry.

```
mona_positive_clean <- mona_positive[
    as.numeric(mona_positive@backend@spectraData$MW)
<=500 &

!is.na(mona_positive@backend@spectraData$PrecursorMZ)]
```

Now, I can create uniform length vectors from the m/z and intensity values using the `bin()` function in the `Spectra` package. Since all the intensity values will share the same mass axis after binning, only one copy of the m/z values needs to be saved, so `lib_mz` is assigned to the first element in the list returned by `mz()`.

```
bin_width = 1

#keep zero value m/z elements
lib_mz <- mz(Spectra::bin(mona_positive_clean,
                          binSize=bin_width,
                          zero.rm=FALSE))[[1]]

#keep zero value intensity elements
lib_inten <-
intensity(Spectra::bin(mona_positive_clean,
                                    binSize=bin_width,
                                    zero.rm=FALSE))
```

It's worth checking the vector to ensure the binning worked as expected. The default way of handling multiple peaks in the `bin_width` is to sum them. However, there are situations where summing is not the appropriate way of combining

intensities. The default behavior can be changed by providing a function using the `FUN=` parameter. For example, to select only the largest peak in the bin, set the parameter to `FUN=max`.

```
  head(lib_mz, 20)
```

```
##  [1]  2.5  3.5  4.5  5.5  6.5  7.5  8.5  9.5 10.5 11.5
12.5 13.5 14.5 15.5 16.5
## [16] 17.5 18.5 19.5 20.5 21.5
```

The bins are set at 0.5 m/z intervals, so the first bin includes all the measured signals between 1.5 m/z and 2.5 m/z, with a center at 2.0 m/z. The low value is a consequence of the acquisition settings across all the spectra entries. The m/z values below `25` are unlikely to be structurally diagnostic or even physically real. While, mathematically, there is nothing wrong with leaving these entries as part of the vector, since they are most likely not real, they should be dealt with when creating the final library data structure.

```
  max(lib_mz)
```

```
## [1] 2994.5
```

The upper m/z value also seems to be an artifact of the acquisition process. Since I will limit the molecular weight to `500`, I will truncate the final vector for each spectrum. While the binning worked as expected, the final vectors must be cleaned to represent real spectra.

Next, I'll get the compound names from the `Spectra` object and prepare to assemble a search result score for the entire library against a selected compound.

```
lib_names <- spectraData(mona_positive_clean,"Name") |>
                as_tibble() |>
                pull()
```

Since I want to perform a vectorized similarity calculation on all the vectors in the library, a matrix will allow me to use the `apply()` function, which should speed up the calculations for even a relatively complex similarity measure to the whole library.

```
lib_mat <- matrix(unlist(lib_inten),
ncol=length(lib_inten[[1]]), byrow=TRUE)
```

The library is now stored as a matrix, with the columns representing m/z, the rows representing compounds, and the value in each vector element representing the sum of all ion intensities within the 1 Da range.

```
lib_inten_mat <- lib_mat[,25:499]
lib_mz_mat <- lib_mz[25:499]
```

The library has been truncated to use vectors that are 475 elements long, begin at m/z `26`, and end at `500`. The vector length 475 and the start and end m/z values are somewhat arbitrary based on my intention to search for small molecules. The same procedure will work for larger molecules and different m/z ranges, so change this part of the preprocessing to meet your analysis needs.

Now, I'll choose a target compound. I'd like a common metabolite with multiple related substances for this example. Tryptophan is a very common metabolite and has many related substances. Multiple laboratories have likely collected spectra for tryptophan and L-tryptophan, which should make a good target compound:

```
length(which(grepl("^tryptophan$|^l-tryptophan$",
             lib_names, ignore.case=TRUE)))
```

```
## [1] 85
```

As expected, there are many entries for compounds with these two names. I'll pick the first one on the list as the target spectra:

```
target_index <- which(grepl("^tryptophan$|^l-
tryptophan$",
                            lib_names,
ignore.case=TRUE))[1]
print(target_index)
```

```
## [1] 240
```

As shown in Figure 7.6, the similarity measures benefit from being normalized to unit length. I'll create a simple function `unit_norm()` to perform unit normalization. If you want to try using one of the more complex normalizations in the literature, they can also be expressed in a similar function.

```
unit_norm <- function(intensity_vector) {
    intensity_vector/sqrt(sum(intensity_vector^2))
}
```

Now the target vector can be normalized so that its Euclidian length is 1:

```
target <- unit_norm(lib_inten_mat[target_index,])

sqrt(sum(target^2))
```

```
## [1] 1
```

Plotting the binning spectrum indicates how much fragmentation was generated for this molecule.

```
p_target_spectrum <- ggplot() +
    coord_cartesian(xlim=c(25,500)) +
    geom_segment(aes(x=lib_mz_mat, y=target,
                     yend = 0, xend = lib_mz_mat),
                 linewidth = 0.5, color=pal$blue) +
    xlab("m/z") +
    ylab("Intensity") +
    theme_classic() +
    theme(plot.title = element_text(hjust = 0.5)) +
    theme(plot.subtitle = element_text(hjust = 0.5)) +
    ggtitle(label =

paste0(mona_positive_clean[target_index]@backend@spectr
aData$Name,
                            " - Index ", target_index),
            subtitle = "Unit Normalized")

print(p_target_spectrum)
```

Using the `apply()` function row-wise (`MARGIN=1`), I can use any distance and similarity functions from the `philentropy` package to calculate the scores for the entries in the library for the target spectrum (Figure 7.7).

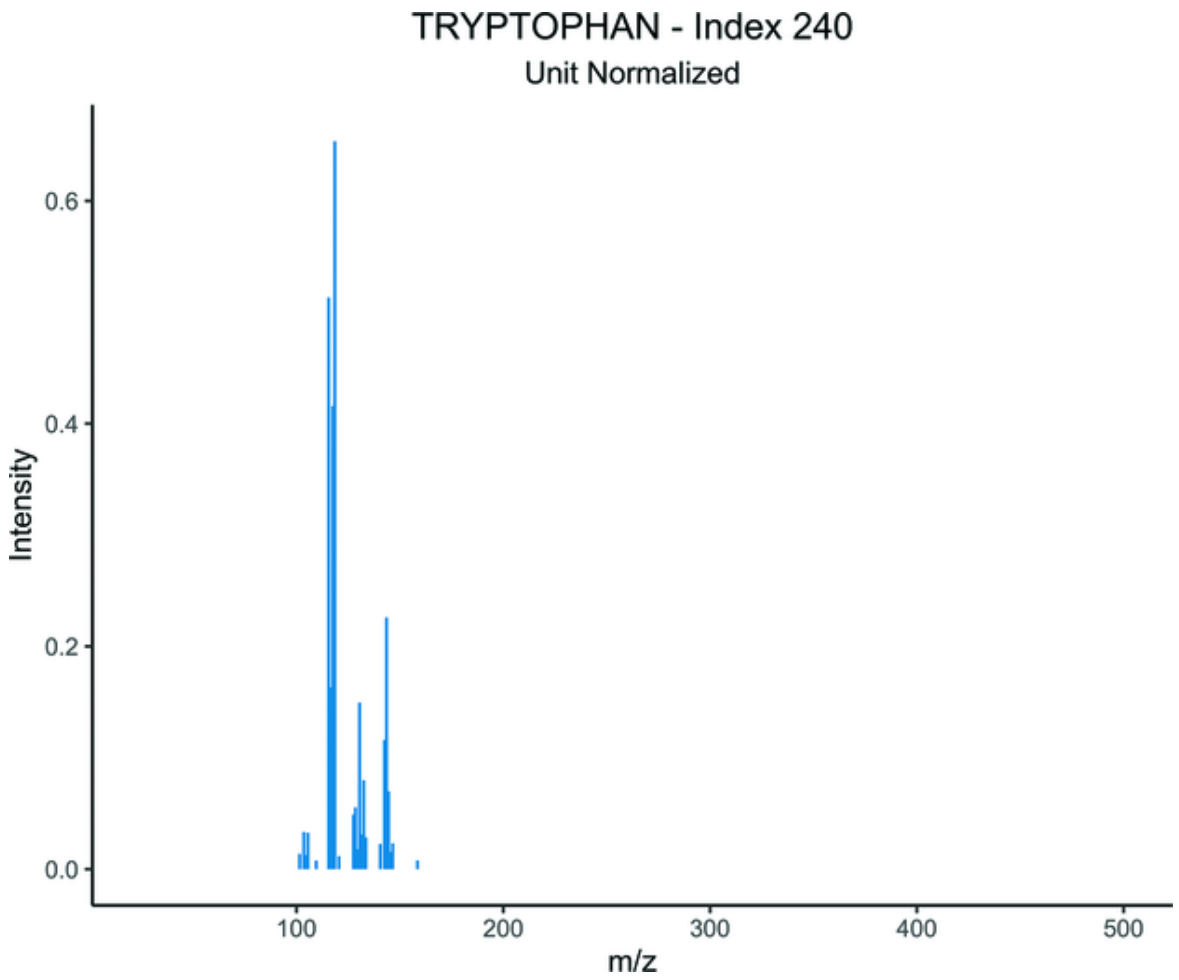


Figure 7.7 Binned m/z vector for tryptophan from MoNA library.

```
scores <- apply(lib_inten_mat, MARGIN=1,
                function(x)
cosine_dist(unit_norm(x),target, testNA = FALSE))
```

All the cosine scores have been calculated and represent similarity (`1` is identical, `0` means nothing in common). The name `cosine_dist()` in the `philentropy` package is somewhat confusing. This low-level function actually returns the cosine similarity. The `philentropy` package documentation states that its functions either return similarity or distance. In this case, the return value and the name do not seem to match.

The search score results from highest to lowest can be seen with a little tidying up:

```
sim.df <- tibble(index=1:length(scores), score=scores)

search <- sim.df[order(sim.df$score, decreasing=TRUE),]

search$compound <- lib_names[pull(search[,"index"])]
search$order <- 1:length(scores)
search <- dplyr::relocate(search, order)
```

You can print the `search` table, but if you want a nicer table, you can also use the `flextable` package to make Table 7.1, which has many options for output to HTML, PDF, and various Microsoft Office formats.

TABLE 7.1

Cosine similarity search results

order	index	score	compound
1	240	1.0000000	TRYPTOPHAN
2	43781	1.0000000	TRYPTOPHAN
3	72907	0.9727516	Abrine
4	72817	0.9711557	L-Tryptophan
5	30436	0.9111296	Tryptophan
6	41907	0.8892105	L-Tryptophan
7	37673	0.8886585	TRYPTOPHAN
8	52533	0.8819955	Tryptophan
9	39440	0.8773309	TRYPTOPHAN
10	52534	0.8573843	Tryptophan
11	73006	0.8549529	Fusaric acid
12	48337	0.8491632	Trp
13	39300	0.8409194	N-ACETYLTRYPTOPHAN
14	38552	0.8307040	N-ACETYLTRYPTOPHAN
15	40244	0.8292659	N-Acetyl-tryptophan; LC-tDDA; CE40

```
  set_flextable_defaults(fonts_ignore = TRUE)
  flextable(search[1:15,]) |>
      colformat_int(big.mark = "") |>
      autofit()
```

These results are quite revealing. First, there is a duplicate library entry at index 43781. The only way that another entry besides the target can get a perfect score is for every element in the vector to be identical.

The submitter of the library spectrum for MoNA is in the `Comments` variable and can be extracted with a regular expression using the `str_match_all()` function from the tidyverse.

```
comments <-
mona_positive_clean@backend@spectraData$Comments[240]
str_match_all(comments, "submitter=submitter = (\\w+
\\w+)")[[1]][,2]
```

```
## [1] "Arpana Vaniya"
```

```
comments <-
mona_positive_clean@backend@spectraData$Comments[43781]
str_match_all(comments, "submitter=submitter = (\\w+
\\w+)")[[1]][,2]
```

```
## [1] "Arpana Vaniya" "Xing Wang"
```

The target and the duplicate were both originally submitted by the same author. The duplicate included both the original submitter and the duplicate submitter. This type of duplication can happen in public repositories through no fault of the duplicate submitter. However, it shows that there can be issues using machine learning algorithms on these repositories when the algorithm is sensitive to duplicates.

Another important thing to notice in this search list is that the cosine score has no semantic meaning. Scores very close to `1` mean that spectra are very similar, and those close to `0` are very different, but the values in the midrange convey no indication of partial similarity. A score of `0.5` has very little meaning based on how cosine similarity is defined.

Based on work done by Stein and Scott [213], cosine similarity was thought to produce a different search result

than Euclidian distance. As Alfassi [215] has since pointed out, Figure 7.6 shows that when the vectors are normalized correctly, the angle θ and the distance D must be correlated and thus must produce identical ordering of search results. The equivalence of the search result order can be demonstrated by performing the comparison using the `euclidian()` function.

```
euclidian_distance <- apply(lib_inten_mat, MARGIN=1,
                      function(x)
euclidean(unit_norm(x),target, testNA = FALSE))
```

Since `euclidian()` returns a distance, I'll convert it to a similarity using Eq. (7.1):

```
# convert distance into similarity
scores <- 1/(1 + euclidian_distance)
```

Tidy up the data for display (shown in Table 7.2):

TABLE 7.2

Euclidian distance search results

order	index	score	compound
1	240	1.0000000	TRYPTOPHAN
2	43781	1.0000000	TRYPTOPHAN
3	72907	0.8107372	Abrine
4	72817	0.8063318	L-Tryptophan
5	30436	0.7034362	Tryptophan
6	41907	0.6799382	L-Tryptophan
7	37673	0.6793973	TRYPTOPHAN
8	52533	0.6730347	Tryptophan
9	39440	0.6687549	TRYPTOPHAN
10	52534	0.6518605	Tryptophan
11	73006	0.6499398	Fusaric acid
12	48337	0.6454744	Trp
13	39300	0.6393626	N-ACETYLTRYPTOPHAN
14	38552	0.6321566	N-ACETYLTRYPTOPHAN
15	40244	0.6311725	N-Acetyl-tryptophan; LC-tDDA; CE40

```
sim.df <- tibble(index=1:length(scores), score=scores)

search <- sim.df[order(sim.df$score, decreasing=TRUE),]

search$compound <- lib_names[pull(search[,"index"])]
search$order <- 1:length(scores)
search <- dplyr::relocate(search, order)
```

While the similarity scores computed from D differ from the values of $cos\theta$, the order of the index values of library

matches in Table 7.2 is the same as in Table 7.1. This confirms that Euclidian distance and cosine similarity produce the same search results as suspected based on Figure 7.6, and shown mathematically by Alfassi [215].

Since Euclidian distance and cosine similarity produce the same search results, having an alternative method could be valuable when trying to identify candidate structures for a mass spectrum from an unknown compound. A library search uses spectral similarity scores to find chemical structures that might have generated the spectrum, thus a natural place to look for a more chemically meaningful comparison. One approach is to treat the mass spectrum as a *molecular fingerprint* and use molecular fingerprint similarity measures to compare spectra.

A molecular fingerprint is an abstraction of structural features usually represented as a binary vector of various lengths in which a structural feature is either present or absent. Because of their importance in drug screening, there are a huge number of ways to compute molecular fingerprints [220]. The molecular fingerprint features have values of either `1` or `0`, and the fingerprint is handled as a binary vector. While the most common measure of distance between two vectors is Euclidian distance [220], for molecular fingerprints the industry standard is the *Tanimoto coefficient* [212, 221, 222]. When searching spectral libraries or comparing an unknown spectrum to a theoretical spectrum, the mass spectrum is used as a type of molecular fingerprint. A vector representing a mass spectrum has continuous feature values (peak intensities), however, rather than binary values. There is, however, a continuous version of the Tanimoto coefficient, which, like the binary version, takes common and unique peaks into more consideration than the Euclidian distance. It’s worth testing the Tanimoto coefficient for library search to see if the similarity measure gives more chemically meaningful scores based on the intuitive logic of giving extra weight to the number of peaks

in common when considering the similarity of two mass spectra.

The details of computing the continuous Tanimoto coefficient and related measures are outside the scope of this book, but more details can be found in Drost [216] and implemented in the `philentropy` R package.

As with other functions in the `philentropy` package, the Tanimoto coefficient is computed in much the same way as the cosine and Euclidian measures:

```
tanimoto_distance <- apply(lib_inten_mat, MARGIN=1,
                           function(x)
tanimoto(unit_norm(x),target, testNA = FALSE))
scores <- 1/(1 + tanimoto_distance)

sim.df <- tibble(index=1:length(scores), score=scores)

search <- sim.df[order(sim.df$score, decreasing=TRUE),]

search$compound <- lib_names[pull(search[,"index"])]
search$order <- 1:length(scores)
search <- dplyr::relocate(search, order)
```

Notice that in Table 7.3, the top four spectra are the same as found by the other measures (Tables 7.1 and 7.2). However, the list order varies after that. Also, the scores drop faster than the cosine distance measure. Also, the score's meaning can be understood in terms of the number of peaks shared between the two spectra combined with features of the space-like distance measure. Tanimoto similarity is more physically intuitive than the other measures and incorporates many of the heuristic ideas proposed for spectral similarity over the years [211, 213, 214, 223–225].

TABLE 7.3

Tanimoto coefficient search results

order	index	score	compound
1	240	1.0000000	TRYPTOPHAN
2	43781	1.0000000	TRYPTOPHAN
3	72907	0.7991675	Abrine
4	72817	0.7923571	L-Tryptophan
5	37673	0.7242599	TRYPTOPHAN
6	39440	0.7147021	TRYPTOPHAN
7	41907	0.7055148	L-Tryptophan
8	52533	0.7048442	Tryptophan
9	30436	0.6998091	Tryptophan
10	52534	0.6765164	Tryptophan
11	30435	0.6755546	Tryptophan
12	42113	0.6713750	TRYPTOPHAN
13	48336	0.6653411	Trp
14	39300	0.6627067	N-ACETYLTRYPTOPHAN
15	38529	0.6617035	TRYPTOPHAN ETHYL ESTER

Unsupervised learning has serious limitations because of the basic lack of knowledge of ground truth regarding the observation. When you don’t know what class observations belong to or the true value of an outcome of a predicted numeric value, then the unsupervised learning methods are the method of last resort. When you don’t know what sample belongs to what group, just knowing that two samples are somehow similar can lead to chemical and biological insights. It is, however, not a great use of an unsupervised learning method to withhold the known class and use your eye to confirm that groups you already knew existed ended up

similar in some way. For that, it is much better to use *supervised learning*, which uses the known class to test the algorithm's prediction so that you can use the errors made to estimate the quality of future predictions.

7.6 Supervised Learning

Supervised learning is learning from observations where the outcome or grouping is known. It is learning from history by fitting a model to data from the past to use it to predict the future. The term *supervised* simply means that for historical data, you know the outcome, and you use statistical methods to minimize the error your model makes when using the inputs to calculate the output compared to the *known outcome*. The known outcome is called the *label* or *target*, and the inputs to the model are the features described in Section 7.3.3.

In all supervised learning algorithms, a calculation is performed using the prediction of the model and the known outcome. The calculation uses a *cost function*, sometimes called the *loss function* or the *objective function*. Put simply, the difference between the model prediction and the known outcome is used to adjust parameters in the model. So, the *learning* in machine learning is identical to *fitting* in statistics. It sounds more impressive when you say the model *learns* or that it is *trained* to produce the correct response. The term *learning* is a bit of anthropomorphizing that is mostly harmless. However, this jargon can make it easy to forget that problems with learning models can usually be framed in statistical terms of fitting a model to data.

7.6.1 Overfitting and the Bias-variance Tradeoff

Another example of anthropomorphizing in machine learning is the saying that there is a risk that the model will *memorize* the data, meaning that training error will go to zero but the performance on the test set and subsequent data sets may get worse. In very familiar statistical terms, this is *overfitting*. An overfit model will fail to work properly when presented with new data. As described above, the loss function defines how well a model works. Often the loss function is based on the *mean squared error* (MSE) or the average of the square of the differences between what the model predicts and the actual outcome value. It is easy to imagine that a sufficiently complex model can be fit such that it goes through every point in a data set. The MSE would be zero, and the *bias* would also be zero. If such a model uses a high degree of curvature to achieve zero bias, then the computed *variance* of the values predicted by the model will be large. Since the data used to fit the model is only a sampling of the total population of possible observations, the real goal of a model would be to minimize the global MSE, not just the local MSE of the sample. In other words, the goal is to minimize the expected value of the MSE when tested on arbitrarily many points *outside of the training set*, based only on the training set. The expected value for the MSE for new data can be decomposed using Eq. (7.3) [226]:

$$Total\ Error = Irreducible\ Error + Bias^2 + Variance \quad (7.3)$$

The *Irreducible Error* term is caused by noise in the data and cannot be eliminated no matter how good the model is. The bias term is how much the estimated value differs from the true value, and the variance term is how much the model is the square of the deviation of the predictions of the model around the true value.

Equation 7.3 means that while some error will always be present because of noise, the amount of expected *total error* from a model is a balance between the contribution from bias and the contribution from variance. Figure 7.8 shows the *bias-variance tradeoff* in terms of total error and model complexity.

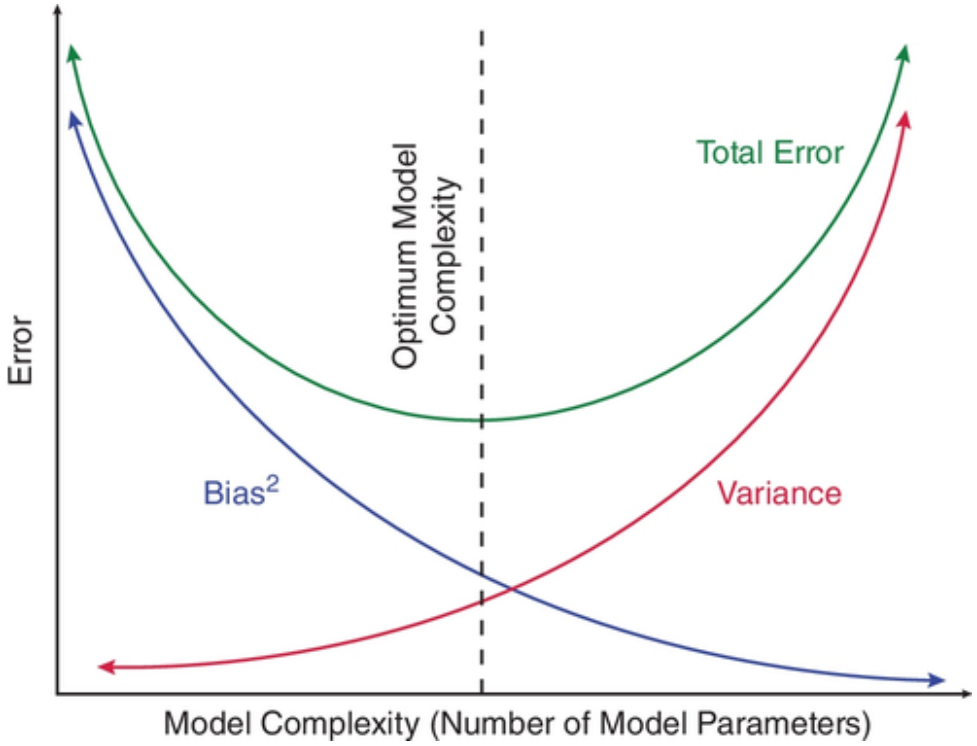


Figure 7.8 The bias-variance trade-off is shown in terms of model complexity. The total error is based on the mean square error decomposition, and model complexity is expressed in terms of the number of model parameters.

In this chapter, I will use two powerful methods for using the available data to estimate the performance of a model on a population: splitting the data into training and test sets and the method of cross-validation.

7.6.2 Splitting Training and Testing Data

One of the central techniques used in machine learning to find the optimum model complexity and avoid overfitting is to hold out a subset of the data available and use it to check how well the model will perform on data it was not trained on. When the data is split to create a hold-out set, the procedure is called a train-test split. Simply, the approach is to train the model on the training split of the data and minimize the errors made. If the model is overfit, the training error will go to zero. So, the objective is to choose a model using the training error to pick parameters that minimize the test data error. It is remarkably easy to cheat at this task and fool yourself into believing that your model is working better than it is. The most common and difficult error to eradicate is letting information from the test set leak in various ways into the training process. That leads to a central rule in machine learning:

> **The Prime Directive:** Whenever you perform an operation involving a test set (or any kind of hold-out data), ensure it is isolated from the training set. If any information from the test set *leaks* into the model or the training data, your model will be compromised, sometimes beyond repair.

Using the benzodiazepines data and the data preparation recipe built in Section 7.3.1, I will show how to build several models using the tidymodels framework. This time, before the recipe is specified, I will split the data into two parts: the *training set* and the *test set*. There is no perfect split ratio, although there are rules of thumb to follow. The training data set is usually between 70 and 80% of the data. Later, I will show how to use *cross-validation* to estimate model performance using only the training set. Another good practice is to ensure the training and testing sets have

roughly the same proportions of the outcome variable, which is called *stratification*.

In the benzodiazepine data, observations represent characteristics of chromatographic peaks found for two compounds: lorazepam and temazepam. The goal of this example is to use the chromatographic features to predict which compound is present. Predicting the class of an observation is a typical classification problem in machine learning. Classification is not restricted to two groups; the methods shown in this chapter will generalize to any number of groups. It's easy to imagine wanting to classify an observation related to benzodiazepines into all of the compounds in that class that are measured by an assay. Visualizing binary classification results is easy, and its prevalence in real-world problems makes it more than a toy example.

First, I'll specify how to split the data, using 75% for training and 25% ensuring that the ratio of lorazepam to temazepam observations is the same in both sets using `initial_split()`:

```
set.seed(2112)

benzo_data_split <- initial_split(benzos_msdata, prop =
0.75, strata=compound)

benzo_data_split
```

```
## <Training/Testing/Total>
## <750/250/1000>
```

The `initial_split()` function uses sampling to place each observation in the training or test set. The `training()` and `testing()` functions extract the training and test sets from the original data using split specifications.

```
benzo_training_data <- training(benzo_data_split)
benzo_testing_data  <- testing(benzo_data_split)
```

The preprocessing recipe is specified based on the training data:

```
benzo_classification_recipe <- recipe(compound ~ .,
data = benzo_training_data) |>
    step_rm(index) |>
    step_log(all_numeric()) |>
    step_normalize(all_numeric())
```

It is important to remove the `index` variable since the data are sorted by compound, which puts all the lorazepam observations within one range of index values and the temazepam values in a different range. The perfect correlation of `index` values with the compound class is an example of label leakage described above.

In the next few sections, I will show how to use some well-known machine learning algorithms to point out some key ideas for their application to mass spectrometry data. You can consult the many references for the details of the models I show here and learn about many others I won't cover. The excellent text *An Introduction to Statistical Learning* (ISL) is a great place to start [204]. It can be followed by the legendary book, *The Elements of Statistical Learning* (ESL) [184]. For an even deeper dive, Kevin Murphy's *Probabilistic Machine Learning* books [227, 228] provide a comprehensive coverage of the most recent advances in machine learning.

7.6.3 Logistic Regression

The `tidymodels` package `parsnip` provides functions for model specification for a wide range of model types and helper functions to work with models and modeling results. I will

start with one of the least complicated and most used machine learning algorithms for classification: *logistic regression*. The word regression in the name means that the algorithm uses a *maximum likelihood* method to fit parameters to the model. Unlike the linear regression used in Section 6.5.3, where the model predicts the outcome value based on fitting a linear equation, in classification, the prediction is the *probability* of an observation belonging to a class. Logistic regression fits a logistic function to qualitative data by encoding the class as a numeric value and performing optimization to estimate the parameters that best separate the classes. The value of the logistic function ranges from `0` to `1`, representing the probability of an observation belonging to a particular class. Logistic regression models are linear models, so to work well, the classification boundary needs to be described by a linear equation.

```
logistic_model <- logistic_reg() |>
    set_engine("glm") |>
    set_mode("classification")
```

Next, I'll build a `workflow` object that combines the recipe with the model:

```
benzo_logreg_workflow <- workflow() |>
    add_model(logistic_model) |>
    add_recipe(benzo_classification_recipe)

benzo_logreg_workflow
```

```
## == Workflow
==========================================================
============
## Preprocessor: Recipe
## Model: logistic_reg()
##
## -- Preprocessor ----------------------------------------
```

```
-------------------------------------
## 3 Recipe Steps
##
## * step_rm()
## * step_log()
## * step_normalize()
##
## -- Model ----------------------------------------------
-------------------------------------
## Logistic Regression Model Specification
(classification)
##
## Computational engine: glm
```

In binary classifications, the first level of an outcome is treated as the “positive” case, and the second is used as the “negative” case:

```
  levels(benzos_msdata$compound)
```

```
## [1] "lorazepam" "temazepam"
```

These factors are coded automatically by R into numeric values for use in the regression:

```
  as.numeric(unique(benzos_msdata$compound))
```

```
## [1] 1 2
```

The two factors are coded as numbers `1` and `2`. For logistic regression, these codings are fine. To perform the classification, a probability for every outcome is computed. The predicted class is the outcome that has the highest probability. In a two-class dataset situation, a probability above `0.5` means that the observation is predicted to be from the first class in the list, which is called positive. The terms positive and negative in binary classification come from the historical use of these tools to distinguish between a

treatment having an effect (positive) and no effect (negative). If the two classes are compound names, then a true positive is the correct call of the first class, and a true negative is the correct call of the second. In this classification, lorazepam is the first level and is treated as the positive case. In a binary classification, the positive or negative class value is arbitrary. The default is to treat the first categorical level as positive and the second as negative. There are no mathematical limits to the number of classes that can be predicted: there is always a boundary between a specific class of observations and all other classes.

Now, the workflow can be used to train the specified model using the `parsnip` `fit()` function:

```
benzo_logreg_fit <- benzo_logreg_workflow |>
    fit(data = benzo_training_data)
```

To see the fit, you can extract it from the workflow using the `parsnip` helper function `extract_fit_parsnip()`:

```
benzo_logreg_fit |>
    extract_fit_parsnip() |>
    tidy()
```

```
## # A tibble: 7 x 5
##   term          estimate std.error statistic
p.value
##   <chr>            <dbl>     <dbl>     <dbl>
<dbl>
## 1 (Intercept)    4.46e-1   8.92e-2   4.99
0.000000591
## 2 response      -6.12e-1   3.48e-1  -1.76    0.0788
## 3 peakAreaQuant  3.61e+8   9.45e+8   0.382   0.702
## 4 peakRTQuant    9.49e-1   7.63e-1   1.24    0.214
## 5 peakAreaQual  -3.52e+8   9.21e+8  -0.382   0.702
## 6 peakRTQual    -2.93e-3   7.66e-1  -0.00382 0.997
## 7 ionRatio      -1.21e+8   3.15e+8  -0.382   0.702
```

Since logistic regression divides the classes along linear boundaries, the model's parameters can be evaluated using normal statistical measures, such as the standardized parameter estimate and the p-value for the test statistic. In logistic regression, the z-score is used as the test statistic instead of the t-score, which is used for linear regression. The z-score is computed from the parameter estimate and the standard error:

$$z = \frac{estimate}{standard\ error} \tag{7.4}$$

One of the reasons that logistic regression is still used and still important is that the z-score can be used to determine *variable importance* in the model. This makes it easy to determine how a model makes predictions and why a particular class was predicted for a specific observation. While highly correlated features can cause problems for this simple approach to explaining a model [229], it is a very desirable quality. The ability to observe all the model parameters directly and to measure their importance makes logistic regression a *self-explanatory model*.

A key idea in the `tidymodels` approach to machine learning is to standardize how models are used so that you can use a common approach to all models. This standardization allows *variable importance*, also called *feature importance*, to be computed for any model. The `vip()` function package works with a `workflow` object to show which variables contribute the most to a classification outcome. In Figure 7.9 obtained from the `vip()` function matches the z-score for the variables in the model shown above. As I will show for the next two machine learning algorithms, variable importance is not so easily estimated for other model types. The lack of a simple model explanation will require additional tools to explain how a model works in general and why a specific outcome was obtained for a particular observation. The `vip` package can

perform variable importance calculations for a wide range of models using a variety of approaches [230].

```
vip(benzo_logreg_fit)
```

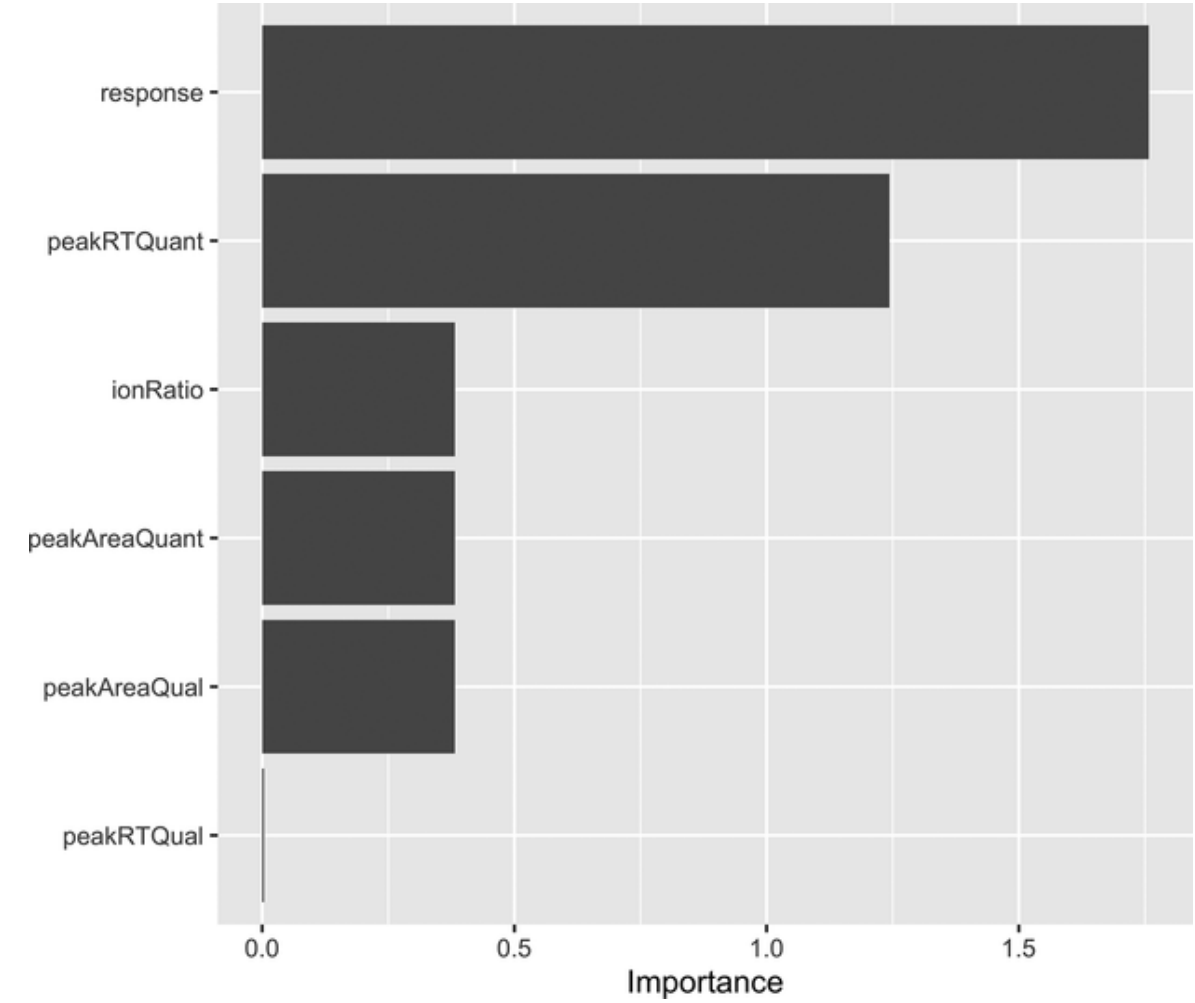


Figure 7.9 A visual representation of the importance attributed to variables in the logistic regression model fit to the benzodiazepine data.

The model has been fit and can now be applied to the testing_data (Figure 7.9). The `predict()` function is used with the `type= "prob"` parameter to compute the probabilities for each observation being in a class.

```
benzo_logreg_test_pred <- benzo_logreg_fit |>
  predict(new_data=benzo_testing_data, type = "prob")
|>
  bind_cols(select(benzo_testing_data, compound))
```

To see what the predictions looks like, you can take a random sample of the predictions using `sample_n()`:

```
set.seed(42)
sample_n(benzo_logreg_test_pred, size=6)
```

```
## # A tibble: 6 x 3
##   .pred_lorazepam .pred_temazepam compound
##             <dbl>           <dbl> <fct>
## 1           0.172           0.828 lorazepam
## 2           0.106           0.894 temazepam
## 3           0.564           0.436 lorazepam
## 4           0.206           0.794 temazepam
## 5           0.274           0.726 lorazepam
## 6           0.211           0.789 temazepam
```

In the first observation, the probability of the outcome matching the compound lorazepam (the positive case) is below the threshold of `0.5`, which means it would be considered a false negative. The classifier predicted the observation would be the compound temazepam (the negative case). The classifications are correct in the rest of the sampled observations and fall into the true positive category.

> **Important definitions:** *Sensitivity*, *Recall*, *Specificity*, and *Precision* have special meanings in the context of machine learning. The definitions I will be using are given below:

$$Sensitivity = recall = \frac{True\ Positives}{True\ Positives + False\ Negatives} \tag{7.5}$$

$$Specificity = \frac{True\ Negatives}{True\ Negatives + False\ Positives} \tag{7.6}$$

$$Precision = positive\ predicitive\ power = \frac{True\ Positives}{True\ Positives + False\ Positives} \tag{7.7}$$

One of the most common visual tools for evaluating a binary classifier is to plot the *Receiver Operating Characteristic* (ROC) curve. It is a plot of the sensitivity (also called the true positive rate) against one-sensitivity (the false positive rate) at every threshold value (`0-1`). In this type of curve, a 45-degree line represents the performance of a classifier that randomly selects the class for observation. A well-performing classifier will produce an ROC curve with a low false positive rate as the true positive rate increases to 100%. Remember that positive and negative are arbitrarily assigned to compound names in this classifier, so the ROC curve is plotted using the positive case `.pred_lorazepam`. Using the predictions made on the test data, the ROC curve is computed with the `roc_curve()` function. Figure 7.10 shows the ROC curve for the logistic regression classifier.

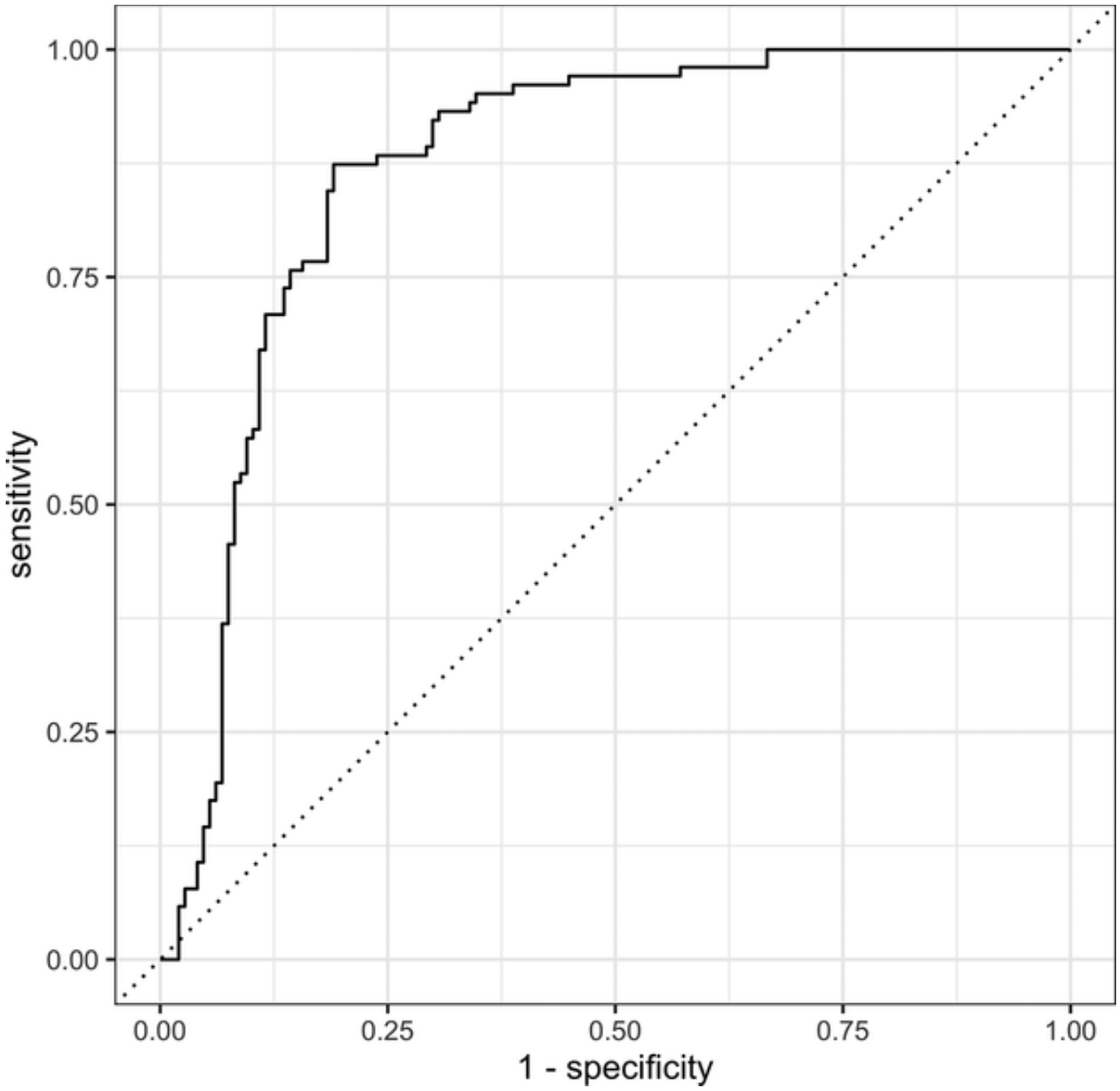


Figure 7.10 ROC curve for logistic regression classification of lorazepam (positive) and temazepam (negative). The dashed line represents random (no skill) classification.

```
benzo_logreg_test_pred |>
  roc_curve(compound, .pred_lorazepam) |>
  autoplot()
```

The area under the ROC curve (ROC AUC) shown in Figure 7.10 is computed by the `roc_auc()` function and shows how the model performed on the test data:

```
  benzo_logreg_test_pred |>
      roc_auc(compound, .pred_lorazepam)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc binary         0.871
```

The confusion matrix indicates performance, especially regarding false positive and false negative calls. You need the predicted class to plot the confusion matrix rather than the probability of class membership. A class is assigned when it has the highest probability based on the fit. In the case of a binary classifier, the threshold for being in the positive class is `0.5`.

```
  benzo_logreg_test_pred_class <- benzo_logreg_fit |>
    predict(new_data=benzo_testing_data, type = "class")
  |>
    bind_cols(select(benzo_testing_data, compound))
```

Sampling this prediction shows the predicted class compared to the actual class.

```
  set.seed(112)
  sample_n(benzo_logreg_test_pred_class, size=6)
```

```
## # A tibble: 6 x 2
##   .pred_class compound
##   <fct>       <fct>
## 1 temazepam   temazepam
## 2 temazepam   lorazepam
## 3 temazepam   temazepam
## 4 temazepam   lorazepam
## 5 temazepam   temazepam
## 6 lorazepam   temazepam
```

The confusion matrix for the classifier is shown in Figure 7.11

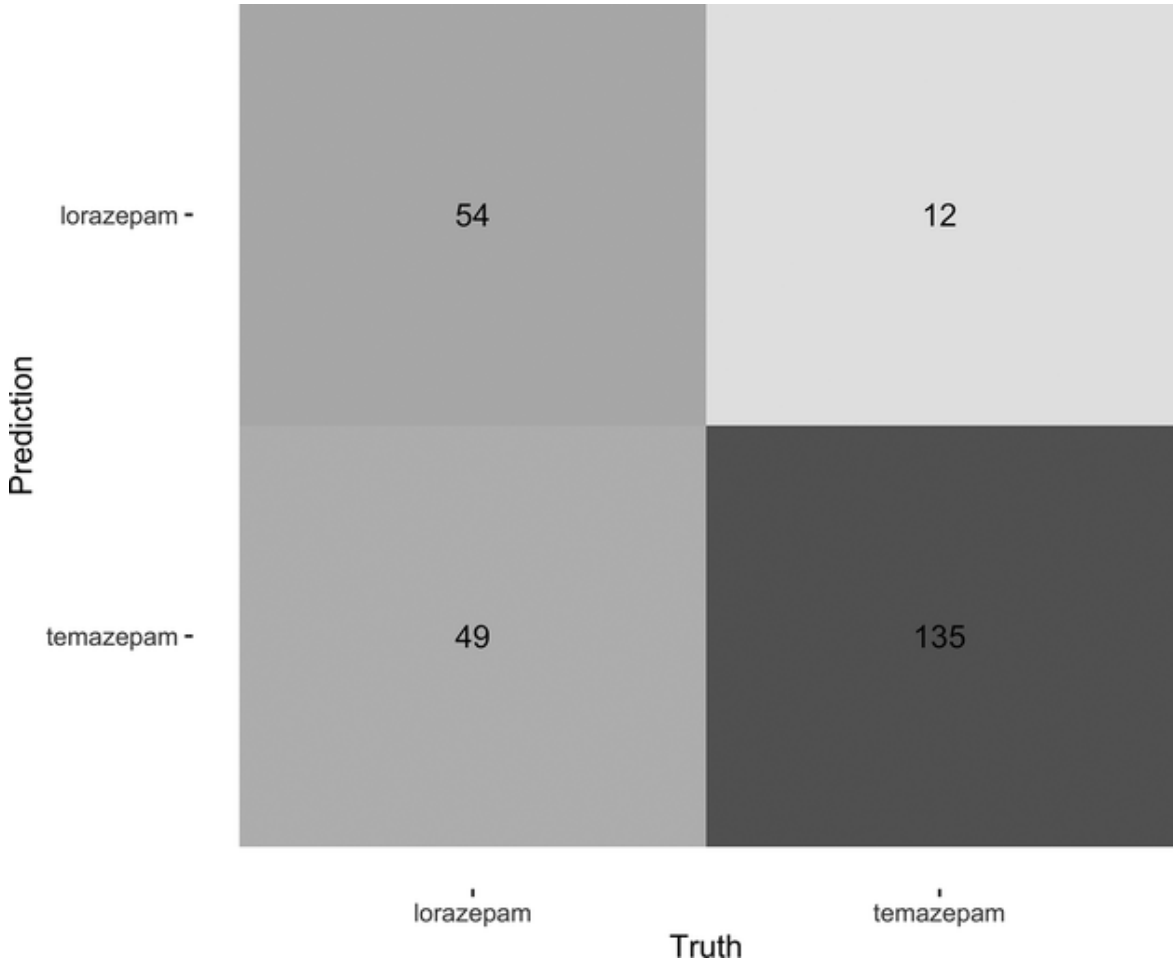


Figure 7.11 The confusion matrix for the logistic regression classifier showing the number of true and false positive calls and the number of true and false negative calls.

```
benzo_logreg_test_pred_class |>
  conf_mat(compound, .pred_class) |>
  autoplot(type="heatmap")
```

There seems to be a problem with the classification of lorazepam. The classifier does only slightly better than random. The class-specific problems are not revealed in the

ROC's shape or AUC. Another informative plot is the *precision-recall* (PR) curve.

Like in the ROC curve, a no-skill classifier would assign classes randomly, which for the PR curve is simply the ratio of positive observations in the dataset:

```
confusion_matrix <-
tidy(conf_mat(benzo_logreg_test_pred_class,
                                    compound,
.pred_class))
no_skill <- sum(confusion_matrix$value[1:2]) /
sum(confusion_matrix$value)

no_skill
```

```
## [1] 0.412
```

The precision-recall curve for the logistic regression classifier is shown in Figure 7.12

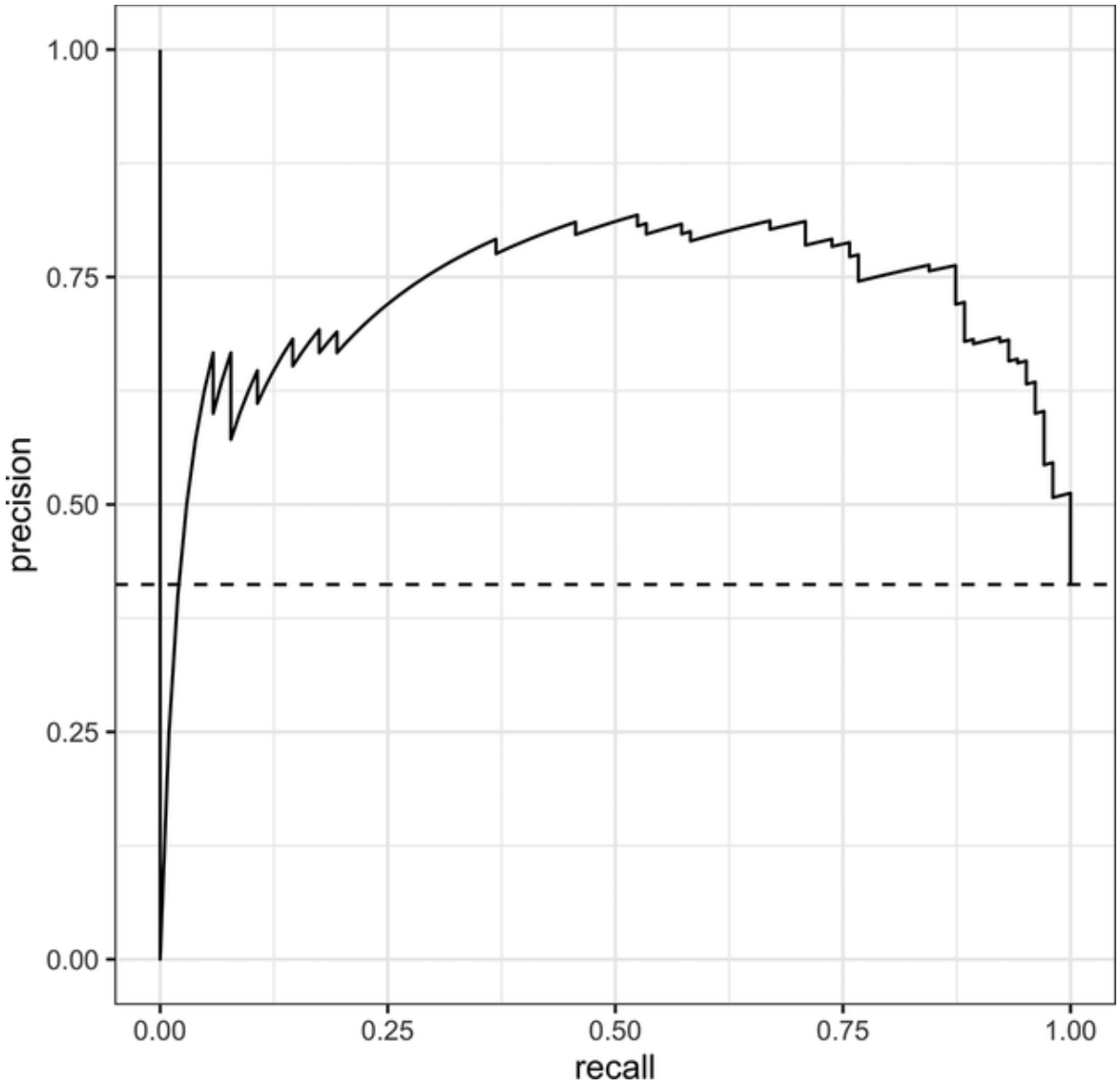


Figure 7.12 Precision-recall curve for the classification of the positive case (compound = lorazepam). The dashed line represents random (no skill) classification.

```
benzo_logreg_test_pred |>
  pr_curve(truth=compound, .pred_lorazepam) |>
  autoplot() +
    geom_hline(yintercept=no_skill, linetype="dashed")
```

```
  benzo_logreg_test_pred |>
      pr_auc(compound, .pred_lorazepam)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 pr_auc  binary         0.722
```

While the total area under the PR curve seems reasonable, Figure 7.12 shows that for lorazepam, the classifier is only moderately better than random. The shape of the PR curve (curving from low to high and back to low) also suggests a poorly performing classifier.

These results are unsurprising, considering that logistic regression can only make linear separations. However, it is always worth starting a simple, base model, like logistic regression, before moving to more complex models. A nonlinear separation might perform better, but if it's not significantly better than logistic regression, it might not be worth the extra complexity.

7.6.4 Support Vector Machines

The *support vector machine* (SVM) is the first nonlinear classification algorithm we'll consider. Instead of using a maximum likelihood estimator to compute model parameters, *support vector classifiers* find a linear boundary that has the largest margin of separation between classes, allowing for some misclassifications. They belong to a family of algorithms called *large margin classifiers*. Support vector machines are support vector classifiers combined with a nonlinear kernel to allow nonlinear decision boundaries to be computed. The mathematical details are quite technical, but the idea is to use a nonlinear function (called a *kernel*) to generate new nonlinear features from the existing ones in hopes of finding

a linear boundary in a higher dimensional space. It seems counterintuitive to add extra dimensions to improve a classifier when the problem is usually that a dataset has too many features, but the process works amazingly well, and SVMs remain one of the most powerful classification algorithms available.

The support vector classifier uses a loss function different from the logistic regression, which includes a parameter not computed from the data. Parameters that must be selected before training on data are called *hyperparameters*. The new parameter is called *cost* in the most basic support vector classifier. It is the penalty added to the loss function for predicting that an observation is within or on the wrong side of the margin. Originally, the cost parameter was considered unimportant, usually set to some small value like 1 and ignored. Later, it was discovered that the value of the cost parameter was very important to the success of support vector classifiers [204]. In `parsnip`, when a model has an important hyperparameter, these can be tuned using various optimization methods.

One of the most common nonlinear SVMs uses a *radial basis function* (RBF) as the kernel to transform the input features into new, nonlinear features. Therefore, the RBF kernel adds a second hyperparameter to the model, representing the *shape* of the kernel. A model's hyperparameters must be selected before the training of the actual model parameters begins. This selection is normally performed by searching for values of each hyperparameter using only the training data.

Cross-validation is the most common method of testing the performance of a classifier using only the training data. Cross-validation is performed by dividing the training data into several *folds* and treating one of the folds as a hold-out test set. The classifier's performance is measured on one fold after being trained on the combination of the remaining folds, which are used as a training set. The number of folds

can be selected based on the size of the training set, and the general process is referred to as *k-fold* cross-*validation*. Often, the number of folds chosen is `5` or `10`, but there is no optimum number. The individual performance measures for each of the hold-out folds are averaged together to estimate the performance of the classifier using a specific set of hyperparameter values. This process is repeated for multiple values of each hyperparameter.

As has been stressed throughout this chapter, cross-validation will overestimate the performance of a classifier if feature selection is done before the folds are created. The correct way to perform cross-validation is to first create the folds and then perform conditioning and feature selection on the combined training folds. The parameters for conditioning and selection are then applied to the hold-out fold. The performance of the hold-out group can then be measured on the features selected for that fold. When this is done, any randomly correlated features will be averaged across all folds, reducing the risk of selecting features correlated by chance. If features are selected before the folds are created, the cross-validation process can dramatically overstate the performance of the classifier for a given set of hyperparameters, resulting in poor model tuning, which may then lead to a model that fits the training data poorly, and ultimately performs poorly on the test set. A bigger mistake would be to select features before splitting the data into testing and training sets. Then, after cross-validation and model selection, the model will seem to perform very well on the test set, too, but can fail to generalize when new data is introduced. For more details on cross-validation and its pitfalls, see ISL and ESL [184, 204].

To improve on the logistic regression classifier for the benzodiazepine data, I'll show how to use an SVM with a radial basis function (RBF) kernel and how to tune the two hyperparameters: *cost* and RBF kernel *shape*. The SVM is specified using the `parsnip` model `svm_rbf()`. To optimize the

model, the hyperparameters are indicated using the `tune()` function. The cost hyperparameter is called `cost` and the shape hyperparameter for the RBF version of an SVM is called `rbf_sigma`. To use SVM RBF implemented in the `kernlab` package, the `set_engine()` function is given the name of the specific SVM package to use.

```
benzo_svm_model <- svm_rbf(cost=tune(), rbf_sigma =
tune()) |>
  set_mode("classification") |>
  set_engine("kernlab")
```

Next, I'll create a regular grid of values for the two hyperparameters to be tuned:

```
svm_grid <- grid_regular(cost(),
                         rbf_sigma(),
                         levels = 5)
```

To perform tuning, you must estimate how well the classifier works for a particular hyperparameter setting using only the training data. To do this, I will use k-fold cross-validation for each grid point. For this example, I am using a 10-fold cross-validation because I have so many observations to work with that 1/10 being treated as a test is still a significant number. Like in the initial split, I want the folds to contain roughly the same ratio of positive and negative observations, so the folds are stratified by compound, just like the train-test split.

```
set.seed(42)

benzo_folds <- vfold_cv(benzo_training_data, v=10,
strata=compound)
```

```
  benzo_svm_workflow <- workflow() |>
      add_model(benzo_svm_model) |>
      add_recipe(benzo_classification_recipe)

  benzo_svm_workflow
```

```
## == Workflow
==========================================================
======================
## Preprocessor: Recipe
## Model: svm_rbf()
##
## -- Preprocessor ----------------------------------------
----------------------------------
## 3 Recipe Steps
##
## * step_rm()
## * step_log()
## * step_normalize()
##
## -- Model -----------------------------------------------
----------------------------------
## Radial Basis Function Support Vector Machine Model
Specification (classification)
##
## Main Arguments:
##   cost = tune()
##   rbf_sigma = tune()
##
## Computational engine: kernlab
```

Using the `metric_set()` function, you can specify the set of metrics you want to collect during the hyperparameter tuning process. In this case, I want to look at both accuracy and the area under the ROC curve.

```
  benzo_svm_metrics <- metric_set(accuracy, roc_auc)
```

Now the `workflow` can be created that uses `tune_grid()` to tune the `cost` and `rbf_sigma` hyperparameters using the folds specified by `vfold_cv()`, and the grid specified by `grid_regular()` collecting the metrics in the metric set:

```
benzo_svm_tune <- benzo_svm_workflow |>
    tune_grid(resamples = benzo_folds,
              grid = svm_grid,
              metrics = benzo_svm_metrics)
```

The result is a set of performance metrics for various levels of the two hyperparameters, which can be plotted (Figure 7.13):

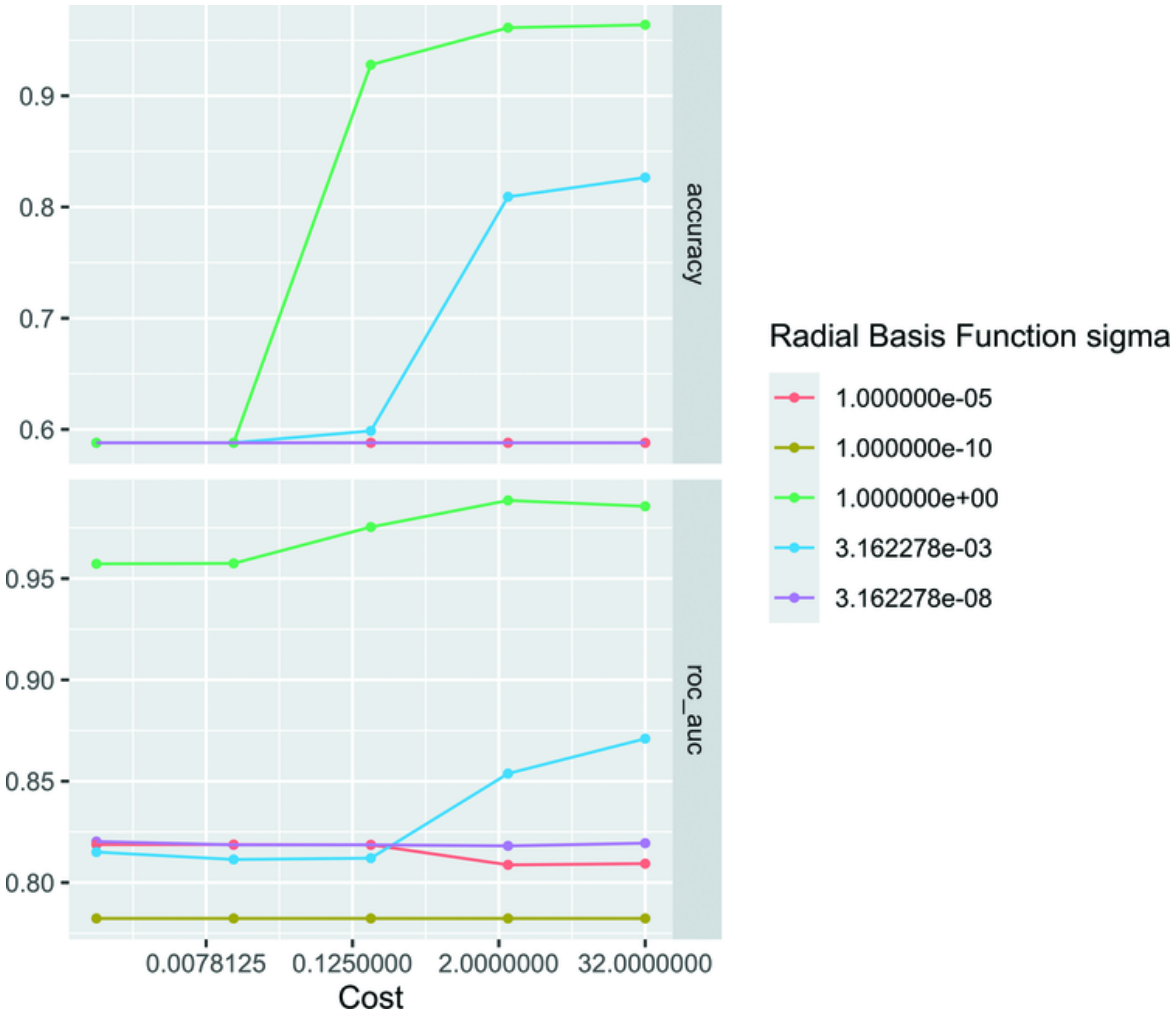


Figure 7.13 Measurement of SVM performance on the benzodiazepine data at different settings of the cost and sigma hyperparameters. Both accuracy and area under the ROC curve were computed using cross-validation on the training data set to evaluate performance.

```
autoplot(benzo_svm_tune)
```

Figure 7.13 shows that the SVM model using the radial basis function kernel is very sensitive to the `cost` and `rbf_sigma` values. A model with a low `cost` parameter will give poor accuracy, and a model with a shape parameter above or below 1 performs worse when measured by ROC area. It is

tempting to say that since the data were normalized and centered, resulting in each column having a standard deviation of `1` and a mean of `0`, the most effective kernel would also have a sigma of `1`, but it is not that simple. Unlike logistic regression models, support vector machines are an example of a *black box model*. Because of the nonlinearity introduced by adding many additional dimensions to the observation using the kernel, it is not immediately obvious why the model makes the classifications it does. You need tools to act as *explainers* to get variable importance from black box models. In Section 7.7, I will go into more detail about explainers and what can be learned from them.

Now that the cross-validation tuning has been performed, I can use `select_best()` to pick the best hyperparameter values according to a selected metric. Both accuracy and ROC AUC were measured during tuning. Accuracy is defined in Eq. (7.8).

$$Accuracy = \frac{True\ Positives + True\ Negatives}{True\ Positives + True\ Negatives + False\ Positives + False\ Negatives} \tag{7.8}$$

Because accuracy treats the positive and negative cases equally, it can be misleading for imbalanced populations (many more of one class of observations than another). You will see both measured, but typically, the classifier with the highest ROC AUC is considered one of the best ways to choose hyperparameters and evaluate models in general. From all the hyperparameters tested, the highest performing model is selected with the `select_best()` function.

```
benzo_svm_best <- select_best(benzo_svm_tune, metric =
"roc_auc")

benzo_svm_best
```

```
## # A tibble: 1 x 3
##    cost rbf_sigma .config
##   <dbl>     <dbl> <chr>
## 1  2.38         1 Preprocessor1_Model24
```

Once the best hyperparameters for the model have been selected, the workflow is updated with those values using the `finalize_workflow()` function. Finalizing the workflow does not fit the workflow to any data set but creates the specification, which includes both the sample preparation and the model with all its hyperparameters fixed.

```
benzo_svm_final_wf <-
finalize_workflow(benzo_svm_workflow, benzo_svm_best)
```

The finalized workflow can then be fit to the cross-validation folds and tested on the hold-out folds. The performance of the final classifier can be estimated by averaging the performance across all the hold-out folds.

```
benzo_svm_cv_fit <-
  benzo_svm_final_wf |>
  fit_resamples(resamples = benzo_folds,
                metrics = benzo_svm_metrics,
                control=control_grid(save_pred=TRUE))

collect_metrics(benzo_svm_cv_fit)
```

```
## # A tibble: 2 x 6
##   .metric  .estimator  mean     n std_err .config
##   <chr>    <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy binary     0.961    10 0.00953
Preprocessor1_Model1
## 2 roc_auc  binary     0.989    10 0.00559
Preprocessor1_Model1
```

This estimate of performance is based on the cross-validation performed by `fit_resamples()` using training data. It is not the final trained model. If these metrics show that the model can perform well in cross-validation, you can proceed to the final fitting step. If cross-validation shows that your model is not performing very well, additional tuning or choosing a different model might be in order. It is extremely unlikely that a model that performs poorly in cross-validation will perform well on new data it has never seen before.

In this case, the cross-validation performance is promising.

```
benzo_svm_final_fit <- benzo_svm_final_wf |>
    last_fit(split=benzo_data_split)
```

The `last_fit()` function performs the final fit on the training data and tests the fit model on the test split to evaluate the model's performance.

```
benzo_svm_pred <- benzo_svm_final_fit |>
  collect_predictions()
```

The `collect_predictions()` function extracts the test data predictions for model evaluation. These predictions can be used to show the confusion matrix:

```
benzo_svm_pred |>
    conf_mat(compound, .pred_class)
```

```
##            Truth
## Prediction  lorazepam temazepam
##   lorazepam        98         4
##   temazepam         5       143
```

The ROC and Precision-Recall curves are shown in Figures 7.14 and 7.15.

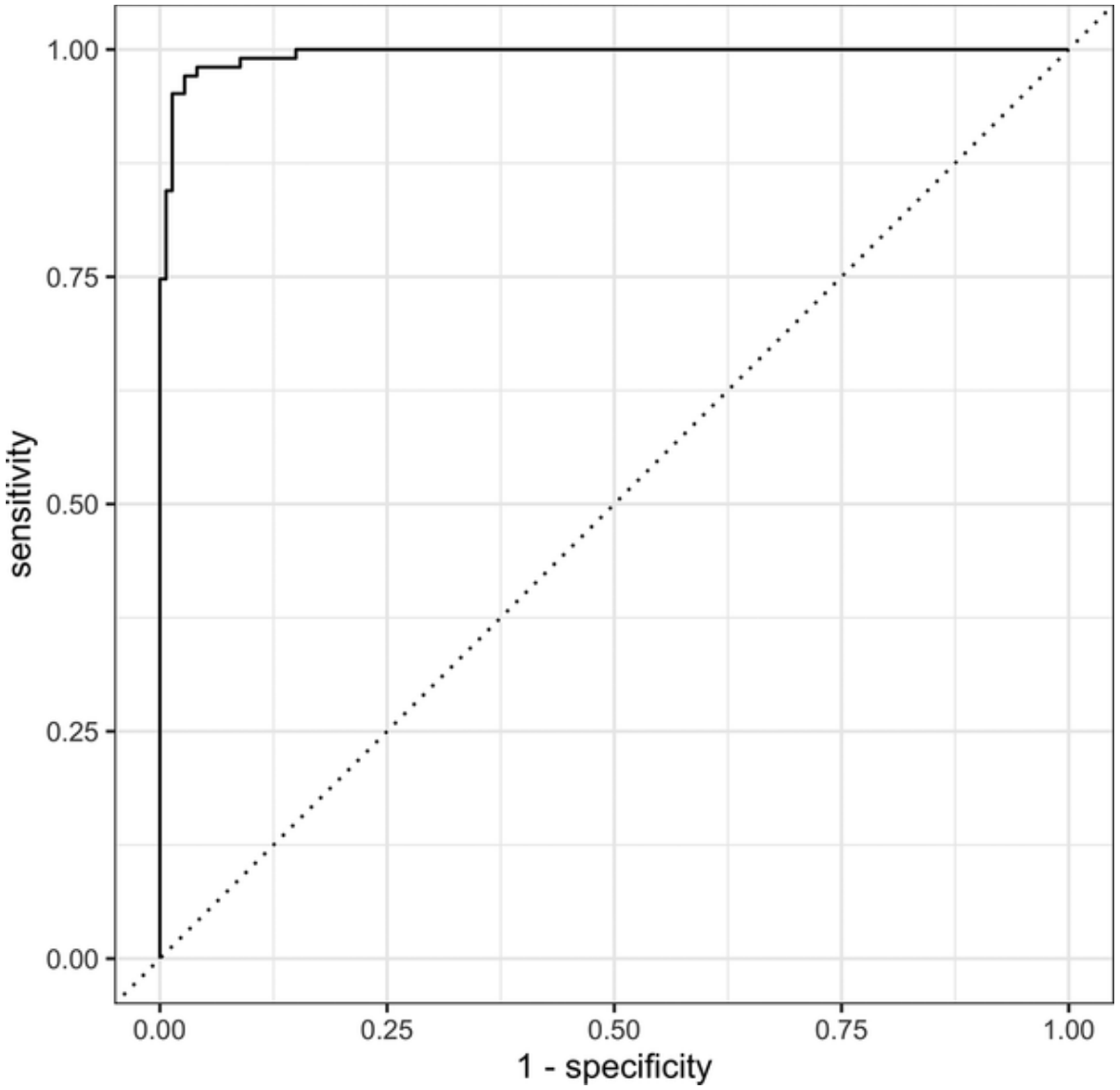


Figure 7.14 The ROC curve for the tuned support vector machine classifier using a radial basis function. This plot represents the performance on only the test data and suggests that the model will generalize well to new observations.

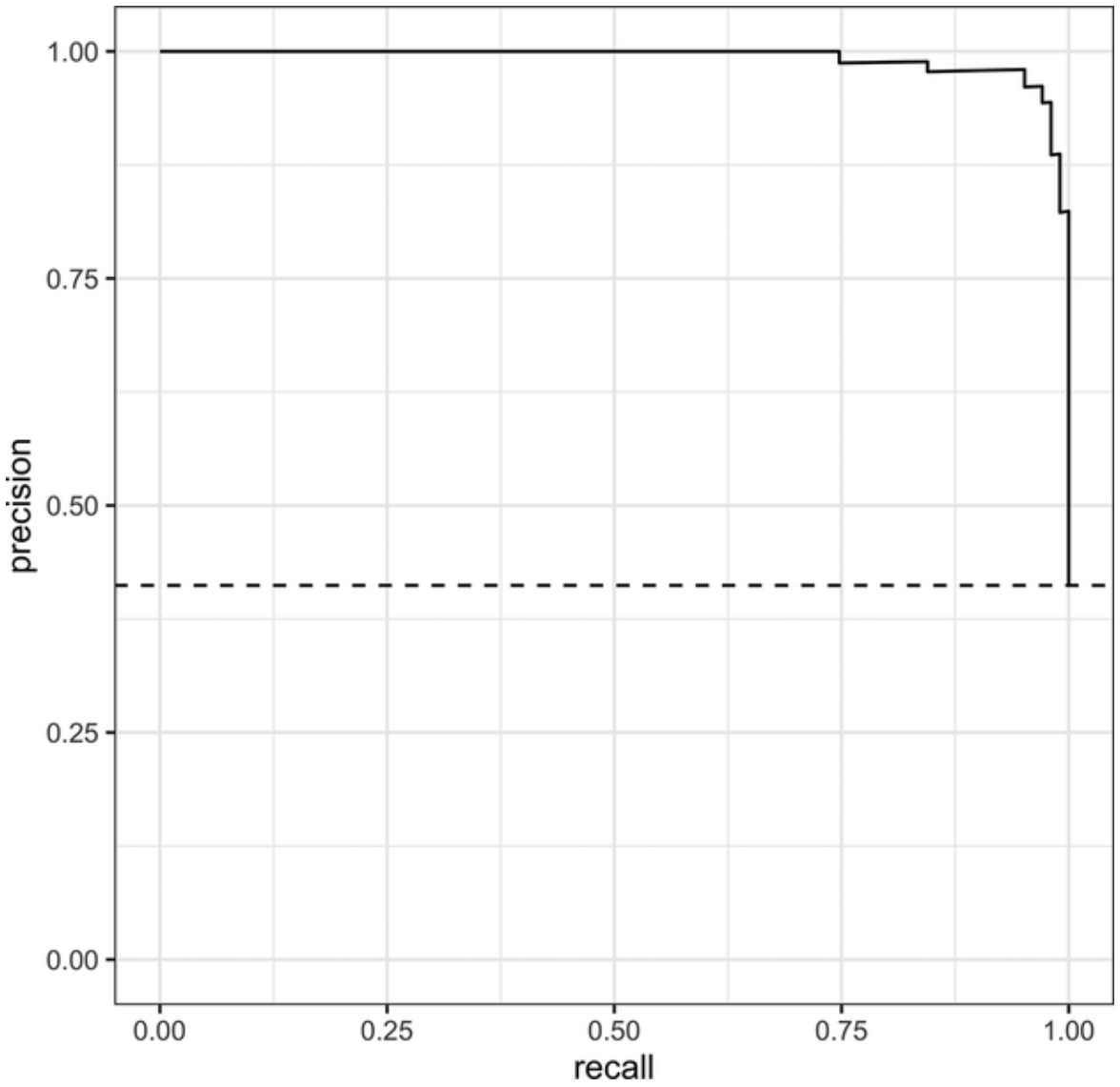


Figure 7.15 Precision-recall curve for the SVM classification of the positive case (compound = lorazepam). The dashed line represents random (no skill) classification.

```
benzo_svm_pred |>
roc_curve(compound, .pred_lorazepam) |>
  autoplot()
```

The ROC curve for the SVM is significantly improved over the logistic regression ROC curve shown in Figure 7.10. The precision-recall curve also reflects significant improvement:

```
benzo_svm_pred |>
  pr_curve(truth=compound, .pred_lorazepam) |>
  autoplot() +
    geom_hline(yintercept=no_skill, linetype="dashed")
```

The PR curve (Figure 7.15) shows the shape expected for a high-performance classifier. The positive predictive value (precision) remains nearly perfect at all thresholds until the recall or true positive rate reaches 100%. At this point, the classifier performs the same as randomly assigning outcomes. In general, the more a classifier avoids acting like it has no skill, the better.

What changed between the logistic regression and the SVM was the introduction of curved boundaries between classes. Curved boundaries can also change the importance placed on the variables in the model. To get variable importance from the SVM model, I will use the `kernelshap()` function from the `kernelshap` package [231], which implements the *Kernel SHAP* (SHapley Additive exPlanations). algorithm described by Lundburg and Lee [232] and extended by Covert and Lee [233].

Kernel SHAP is a complex algorithm, but at a high level, the idea is to approximate Shapley scores for all the features in a dataset. Shapley's idea was to measure the contribution of individual players to an outcome in a multiplayer game using game theory [234]. In machine learning, the variables of an observation are treated like players in a cooperative game, and their contribution to the outcome, according to Lundberg and Lee, is estimated by estimating the effect of removing a variable from the model. The values must be approximated because computing all the Shapley values requires the model to be run with every possible combination of features. Computing the contribution of every combination of features is impractical except for a very small number of features. For more insight into SHAP and other ML-explainer methods, see

Molnar's *Interpretable Machine Learning: A Guide for Making Black Box Models Explainable* [235].

Kernel SHAP is the only SHAP-based explainer that will work with any type of model. Kernel SHAP's method of approximating all the combinations of features needed to compute Shapley values is still computationally expensive compared to a model-specific explainer if one is available for the base learner you are using. In Section 7.7, I'll show how to efficiently collect SHAP values from tree-based models using the `treeshap` package [236].

Using `kernelshap()` requires the training data, a background data set to estimate the mean of each column, and a `predict()` function for the specific model type used.

First, I use the classification recipe to create a version of the entire training set and then remove the label `compound` for use by `kernelshap()`. The Kernel SHAP algorithm needs the column means, which can obviously be computed from the entire data set. For large datasets (more than 100 rows), a subsample is recommended to be used and provided to the algorithm using the `bg_X` parameter. Since this data set was normalized, I already know the mean of every column is zero. However, this may not be the case for every dataset you want to use, so I'm including the creation of the background dataset to complete the example.

```
benzo_classification_training <-
prep(benzo_classification_recipe) |>
    bake(new_data=NULL) |>
    select(-compound)

bg_X <- benzo_classification_training[
    sample(nrow(benzo_classification_training), 100), ]
```

Next, I extract the final model that was fit to the training data using the `extract_fit_parsnip()` function:

```
benzo_svm_fit <- benzo_svm_final_fit |>
    extract_fit_parsnip()
```

Finally, I will provide a custom `pred_fun` function that returns the class probabilities for the positive case (the first column) from SVM models built with `kernlab` [237].

```
predict_function <- function (m, X) {
    predict(m, X, type="prob")[,1]
}
```

Now, the variable importance can be computed. Here, `kernelshap()` is called, and its output is prepared for visualization using the `shapviz()` function from the `shapviz` package [238], which provides a range of functions like `sv_importantance()`, which creates the variable importance bar plot.

```
shap <- kernelshap(benzo_svm_fit,
                   X=benzo_classification_training,
                   bg_X = bg_X,
                   pred_fun = predict_function,
                   verbose = FALSE) |>
    shapviz()

sv_importance(shap, kind = "bar", fill=pal$blue)
```

The feature importance plot shown in Figure 7.16 hints at why the SVM model performs so much better than the logistic regression model. The `response` feature, which relates to the absolute quantity of the compound in the sample, went from the most important feature in Figure 7.9 to the least important feature for the SVM. Based on the SVM feature importance plot, the retention times of the quantifier and the qualifier, along with the ratio of the quantifier and qualifier areas (`ionratio`), are important in assigning the correct

compound name to an observation. This order of importance matches our intuition of how compounds should be identified and is also suggested in guidelines for evaluating data from LC-MS/MS measurements [239]. Beyond variable importance estimation, SHAP can also explain why a specific observation produced a specific output. In Section 7.7, I show how SHAP can be used to show both global and local explanations that can help understand how a complex model produces the predicted results.

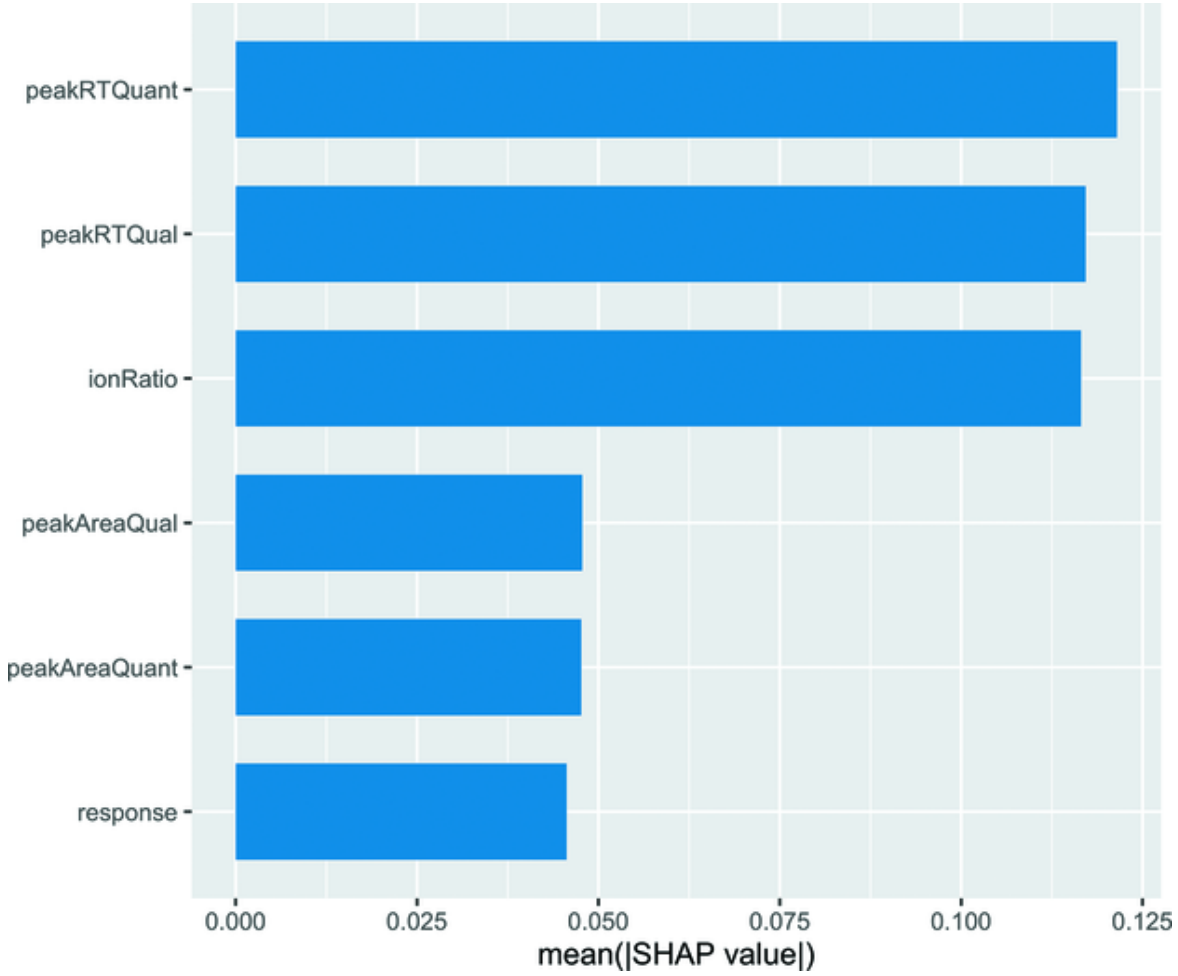


Figure 7.16 Feature importance for the SVM model fit to the benzodiazepine data using the Kernel SHAP algorithm.

7.6.5 Boosted Trees

Both logistic regression and support vector machines perform calculations on the numerical values of the predictors. As you saw in the examples, preparing the data with normalization and scaling was important before modeling. The family of regression and classification algorithms based on *decision trees* operate very differently. As usual, the details can be found in ISL [204] and ESL [184], but the essential idea is to perform splits of the data on a single predictor at a time, resulting in two groups that can then be split on the same or other features. The split point is chosen based on a split rule, usually intended to create the highest class purity in the two split groups (called nodes). Traditional decision trees continue to split nodes until the final nodes (called leaves) contain observations of only one class. Exhaustive splitting can lead to very deep trees and usually results in overfitting.

There are many ways to fight against overfitting when using tree methods. One of the most successful approaches is to create an ensemble of trees. When the tree results are averaged together, the technique is called *bagging*. The most successful of the bagged tree models is called a *random forest* (RF) [184, 204], in which many low-bias, high-variance tree models are built and their outputs are averaged together. Since averaging reduces variance and has little impact on the bias, the result can be a high-performing low-bias, low-variance model. Another ensemble approach, called *boosting* builds models sequentially. The residuals (mistakes) of the first model are used to fit a second model, and so on until the minimum total error is achieved. When the slope of the residuals is used to estimate a gradient toward an error minimum, the method is called *gradient boosting* [240]. Gradient boosting addresses the bias-variance trade-off from the opposite direction of RF. With boosted trees, a weak learner with high bias and low variance is built first, and the bias is reduced by adding additional complexity to the model in the form of additional weak learners. As discussed above,

the goal is to strike a balance between bias and variance to minimize total error in the population. Boosted trees effectively strike this balance while also providing other benefits, and so have become one of the most popular machine learning algorithms for tabular data available [241].

As the base learner for ensemble methods, tree models have many desirable attributes. Trees can handle features that are a mixture of numerical and categorical variables without encoding. Trees can handle missing values without dropping data or imputation. Trees are robust to outliers in the inputs and don't require scaling or normalization. They are also computationally scalable and can handle correlated and zero information predictors.

In the next example, I will show how to use one of the most powerful implementations of boosted trees called *XGBoost* [242], using the `tidymodels` framework. XGBoost computes the first derivative of the loss function to create a gradient, so it is part of the *Gradient Boosted Machine* family of algorithms. Gradient boosting is a first-order method to minimize error. XGBoost goes a step further, using Newton's method for minimizing error, which is a second-order method [243]. XGBoost has gained widespread adoption since its success in the Higgs Boson Machine Learning Challenge [244] and since its introduction, has become a powerful boosting library available for use in many languages.

To show how to use XGBoost, I'll look at a challenging classification problem from a large multiomics study. Yazd et al. recently published a study combining small molecule metabolites with lipids to classify meningioma biopsy samples [245]. One of the goals was to use machine learning to discover specific biomarkers related to the stage of the disease. The paper, titled *Metabolomic and Lipidomic Characterization of Meningioma Grades Using LCHRMS and Machine Learning*, measured the chromatographic peak areas of over 16 000 detected features in 85 biopsy samples

from patients with either Class I or Class II/III grade meningiomas. Multiple machine-learning algorithms were tested to find the best ML algorithm for this type of data. In the comparison, an SVM appeared to perform the best, but gradient-boosted trees were not tested. Given the number of features, it is very probable that many are correlated or uninformative.

The very large number of features also means that feature selection could result in selecting features correlated with the disease class by chance, making establishing the importance of specific molecules as biomarkers difficult. The authors took great care to ensure that all the data was deposited Metabolights repository as `MTBLS4938`. Applying an algorithm like XGBoost to this data could add extra insight to the conclusions of this analysis.

Because the data were going to be used to test models like support vector machines and logistic regression, the authors performed data conditioning steps needed for algorithms that perform calculations on the data:

> "...we applied a k-Nearest-Neighbors imputation method to find the k nearest samples and imputed the missing elements (Python's scikit-learn package). Then, we normalized the data set to sum in order to correct the instrumental and technical variation, followed by transforming the data (log transformation) and autoscaling each data set (to allow a more direct comparison between features of wildly varying intensities) (Figure 6A). Next, the normalized abundances of all of the metabolomic and lipidomic features (in both polarities) were merged into one feature table in Python." [245]

This statement appears to violate the prime directive of machine learning: they normalized and imputed all of the data before performing any splitting used to test classifier performance. Based on the quoted description of the data preparation, the normalization and imputation steps resulted

in *information leakage*, which interfered with any future performance testing. It means that in this study, there was no data available for testing that could represent the larger population of meningioma patients. Any information leakage between the training set and the testing set can lead to overfitting and overestimating how well a model will work on new data.

The impact of creating a connection between the test set and the training set via normalization has been studied empirically by Brouke and Abdullah [246]. In their analysis of information leakage from normalization, they found that information leakage always results in some overestimation of performance compared to models that had no information leakage. The types of data sets used in the leakage analysis study contained many more observations than features. For example, the KDDCUP99 data set contains more than 4.8 million observations and 41 features [247]. The impact of information leakage across millions of observations may be lower than leakage across only 85 observations.

While there was information leakage in the Meningioma study, its impact was far less significant than the apparent *label leakage* that resulted from the feature selection process:

> "Next, the normalized abundances of all of the metabolomic and lipidomic features (in both polarities) were merged into one feature table in Python. Then, we used the scikit-learn ExtraTreesClassifier feature selection algorithm to sort all of the detected features based on their ability to discriminate patients with grade II/III meningiomas from grade I." [245]

By preselecting the top 30 features using a supervised classifier on all of the data, the label information of the entire data set was leaked *prior* to any cross-validation or train-test split. The data from these preselected features were subjected to cross-validation to estimate the performance of

different classifiers. The preselection of features from the entire data set means that any train-test split or cross-validation will overestimate the performance of any classifier used. When the label information is used on the entire data set for feature selection, the selected features will be highly correlated with the label. Since there are approximately 15 000 features that were used in this study, there is a very high probability that at least 30 could be correlated with the label purely by chance. The subsequent use of cross-validation or a train-test split is useless.

Despite the classifier performance degradation expected from information leakage that occurred during normalization and imputation, I would still like to know how well a classifier could perform if I removed the label leakage. Further, I would like to show how to use XGBoost on this data.

The conditioned data is contained in a file called `metabolites_all.csv` and is available on the Metabolights repository with the raw data. It can be loaded and the label converted into a factor:

```
metabolites <- read_csv(file.path("large-
data","metabolites_all.csv")) |>
    mutate(Class=as.factor(Class))
```

Following the basic ML pattern, I'll first create the train/test split specification using 75% of the data for training and 25% for testing and stratifying on class.

```
set.seed(2112)

metabolite_data_split <- initial_split(metabolites,
prop = 0.75, strata=Class)

metabolite_data_split
```

```
## <Training/Testing/Total>
```

<63/22/85>

Two data sets are created using the `tidymodels` split specification:

```
metabolite_training_data <-
training(metabolite_data_split)
metabolite_testing_data  <-
testing(metabolite_data_split)
```

Since the data were already conditioned, the recipe simply associates the outcome with all the predictors and specifies the training data:

```
metabolites_xgb_recipe <- recipe(Class ~ .,
data=metabolite_training_data)
```

The XGBoost model available via `parsnip` gives access to a large number of hyperparameters that can be tuned to help minimize overfitting.

Before performing the fit on the training set and evaluating the test set, I will tune the hyperparameters using a two-step approach described by Kuhn and Silge [134]. The two-step process tunes all of the hyperparameters at once, allowing for possible interactions, but does so in a reasonable (but not necessarily small) amount of time.

```
metabolites_xgb_model <- boost_tree() |>
    set_engine("xgboost") |>
    set_mode("classification") |>
    set_args(trees = tune(),
             mtry = tune(),
             tree_depth = tune(),
             min_n = tune(),
             sample_size = tune(),
             loss_reduction = tune(),
             learn_rate = tune()
)
```

The model and the recipe are combined into a workflow, as with the models used earlier:

```
metabolites_xgb_workflow <- workflow() |>
  add_model(metabolites_xgb_model)  |>
  add_recipe(metabolites_xgb_recipe)
```

The hyperparameter tuning folds are constructed and stratified based on the outcome variable: the `Class` column.

```
set.seed(1234)
metabolite_data_folds <- metabolite_training_data |>
    vfold_cv(strata = Class, v=5)
```

A sparse grid based on a Latin hypercube sampling [248] is constructed using the `grid_latin_ hypercube()` function to find the starting values for a Bayesian optimization algorithm to perform the final tune. This two-step approach to hyperparameter tuning will work for any machine learning algorithm with multiple hyperparameters that need to be tuned. I will use a grid size of `30` to give the `tune_bayes()` step more samples to work with than when tuning only one or two hyperparameters.

```
set.seed(6116)
xgb_grid <- grid_latin_hypercube(
                trees(),
                tree_depth(),
                min_n(),
                loss_reduction(),
                sample_size = sample_prop(),
                finalize(mtry(),
metabolite_training_data),
                learn_rate(),
                size = 30
)
```

I'm going to select the best model on the ROC AUC, so for this step in the process, the only metric I need is `roc_auc`:

```
metabolite_xgb_metrics <- metric_set(roc_auc)
```

Now, find the initial values of all the hyperparameters by averaging performance over the five folds. If the hardware supports parallel processing, the `control_grid()` parameter `parallel_over` will put each fold to be averaged on a separate thread or worker and then combine them into an average.

```
set.seed(73)
metabolite_xgb_tune <- metabolites_xgb_workflow |>
    tune_grid(resamples = metabolite_data_folds,
              grid = xgb_grid,
              metrics = metabolite_xgb_metrics,
              control =
control_grid(parallel_over="resamples",
                                        save_pred = TRUE))
```

Since the `mtry` hyperparameter is the number or proportion of predictors that will be sampled at each split, the value will not be automatically extracted from the workflow object using the `extract_parameter_ set_dials()` function. Unlike all

the other hyperparameters, `mtry` has to be finalized on the training set for the next step in the tuning process:

```
bayes_param <- metabolites_xgb_workflow |>
  extract_parameter_set_dials() |>
  update(mtry = finalize(mtry(),
metabolite_training_data))
```

The final tuning step uses the `tune_bayes()` function to iteratively search the parameter space for a local optimum of the metrics. The algorithm is set to try up to `20` iterations, but it will stop if additional iterations stop improving the metric. Despite setting a random number seed, the Bayesian optimization process is not deterministic and can produce different results each time it is run.

```
set.seed(73)
xgb_tune_bayes <- metabolites_xgb_workflow |>
  tune_bayes(
    iter = 20,
    resamples = metabolite_data_folds,
    param_info = bayes_param,
    metrics = metabolite_xgb_metrics,
    initial = metabolite_xgb_tune,
    control = control_bayes(parallel_over="resamples",
                            save_pred = TRUE)
  )
```

To see how the results of the Bayesian hyperparameter tuning process, the `xgb_tune_bayes` object can be plotted:

```
autoplot(xgb_tune_bayes)
```

Figure 7.17 shows that the tuning process found parameters that can give good performance on the training data and that tuning is important since some hyperparameter values result in a classifier that is no better than flipping a coin (`0.5`

represents no skill). Like with the SVM tuning, I will select the final hyperparameter set from the best model found by `tune_bayes()` based on ROC AUC.

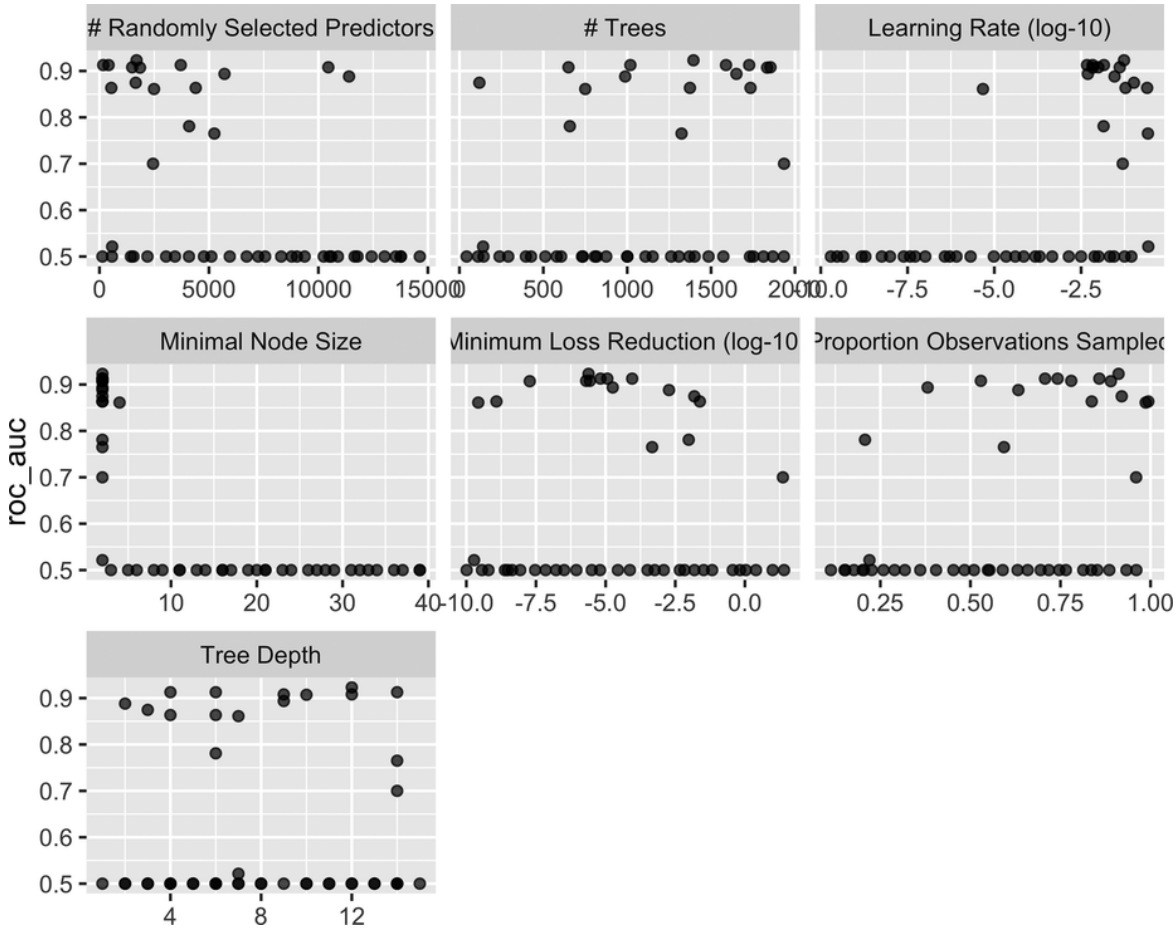


Figure 7.17 Performance measures for hyperparameter values found using a Bayesian search using the results of a Latin square search as starting values.

```
metabolite_xgb_best <- select_best(xgb_tune_bayes,
metric = "roc_auc")

metabolite_xgb_best
```

```
## # A tibble: 1 x 8
##    mtry trees min_n tree_depth learn_rate
loss_reduction sample_size .config
```

```
##   <int> <int> <int>      <int>      <dbl>
<dbl>       <dbl> <chr>
## 1  1693  1395     2         12     0.0544
0.00000244       0.911 Iter7
```

This model will use 1395 trees that are 12 nodes deep. The final nodes have to have at least two observations on each side of the last split, and the splits will be based on the best separation that can be achieved from a feature selected from 1693 features selected at random. The model will use as many as $2.85\,696 \times 10^6$ splits learned from the data.

Using cross-validation on the training set will overestimate the classifier's performance compared to the training set due to the reuse of data between the cross-validation folds. However, it is a useful step because if the classifier performs poorly in cross-validation, it is definitely a bad model.

```
metabolites_xgb_workflow_final <-
metabolites_xgb_workflow |>
    finalize_workflow(metabolite_xgb_best)
```

Fit the training set folds with the best model finalized in the final workflow:

```
set.seed(1661)
metabolites_cv_xgb_fit <-
metabolites_xgb_workflow_final |>
  fit_resamples(metabolite_data_folds,
                metrics=metric_set(accuracy,roc_auc),
                control=control_grid(save_pred=TRUE))

collect_metrics(metabolites_cv_xgb_fit)
```

```
## # A tibble: 2 x 6
##   .metric  .estimator  mean     n std_err .config
##   <chr>    <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy binary     0.838     5  0.0518
Preprocessor1_Model1
```

```
## 2 roc_auc  binary     0.904     5  0.0410
Preprocessor1_Model1
```

Using a five-fold cross-validation on the training set using the final values of all the hyperparameters shows high performance in terms of ROC AUC. Again, this level of performance is not expected to hold up for the test data or other new data, but it means that the tuned model is good enough to take to the testing step:

```
metabolites_cv_xgb_pred <- metabolites_cv_xgb_fit |>
  collect_predictions()
```

```
metabolites_cv_xgb_roc <- metabolites_cv_xgb_pred |>
  roc_curve(`Class`, .pred_Group1)

autoplot(metabolites_cv_xgb_roc)
```

The ROC curve in Figure 7.18 suggests that even in cross-validation, the model doesn't show as much overfitting as expected from the average ROC AUC. Now, the model's performance on the test data can begin. I have to go back to collecting both accuracy and roc_auc for the predictions to include the predicted class. The `roc_auc` metric only collects the class probabilities, not the class prediction, which is needed for the confusion matrix.

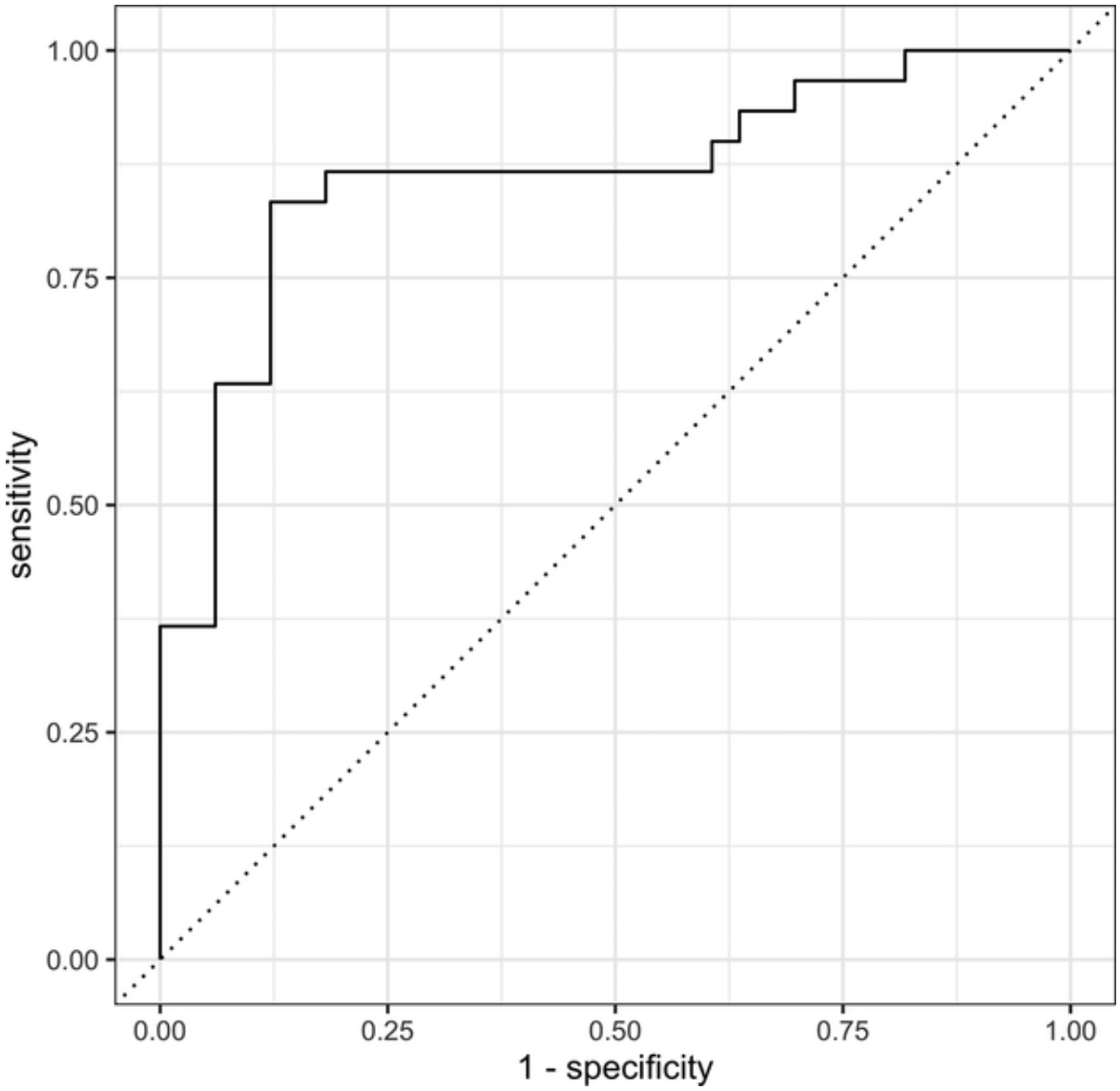


Figure 7.18 The ROC curve computed from the cross-validation results for the best model selected after hyperparameter tuning. This estimate of performance is based only on the training data, and new data might perform worse, but this is the best that tuning can achieve when cross-validation is employed.

```
metabolites_xgb_final_fit <-
metabolites_xgb_workflow_final |>
    last_fit(metabolite_data_split, metrics =
metric_set(accuracy,roc_auc))
```

The `last_fit()` function fits the model to the full training set and then applies the model to make predictions on the test set.

```
  collect_metrics(metabolites_xgb_final_fit)
```

```
## # A tibble: 2 x 4
##   .metric  .estimator .estimate .config
##   <chr>    <chr>          <dbl> <chr>
## 1 accuracy binary         0.682 Preprocessor1_Model1
## 2 roc_auc  binary         0.783 Preprocessor1_Model1
```

As expected, the performance on the test set is lower than on the training set. The confusion matrix from the test set shows a problematic situation:

```
  metabolites_xgb_conf_mat <- metabolites_xgb_final_fit
  |>
      collect_predictions() |>
      conf_mat(Class, .pred_class)

  metabolites_xgb_conf_mat
```

```
##           Truth
## Prediction Group1 Group2
##     Group1      9      6
##     Group2      1      6
```

The confusion matrix shows that the classifier does a good job of predicting Class I samples, while Class II/III samples have a 50% chance of being called either Class I or Class II/III.

```
metabolites_xgb_test_roc <- metabolites_xgb_final_fit
|>
  collect_predictions() |>
  roc_curve(Class, .pred_Group1)

autoplot(metabolites_xgb_test_roc)
```

The ROC curve in Figure 7.19 has a reasonable shape for a test set of only 22 samples, but, as the confusion matrix shows, the classifier is only modestly better than random, and that is only due to its performance on Class I samples.

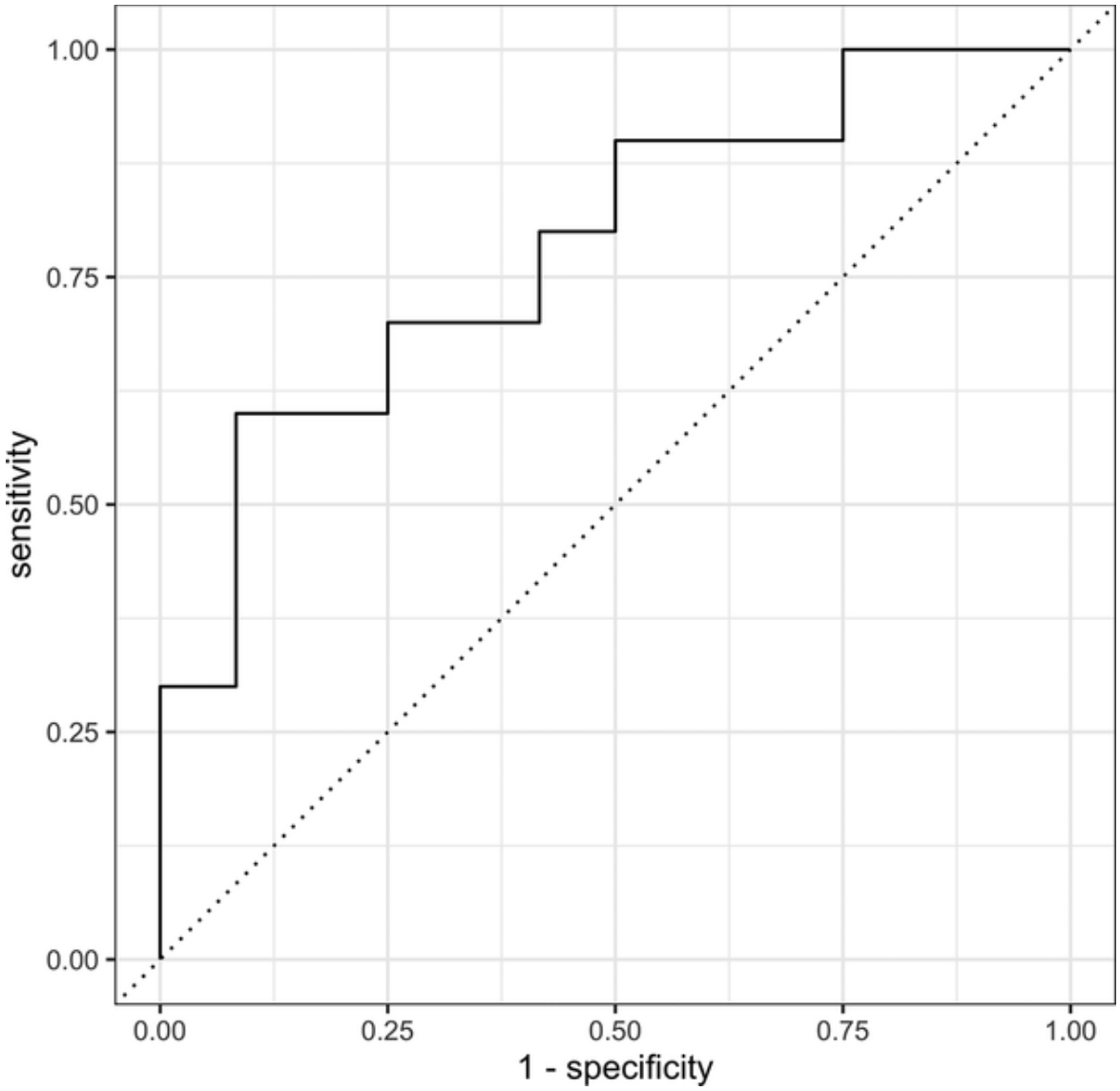


Figure 7.19 The ROC curve computed from the test data using the best model selected from cross-validation-based hyperparameter tuning.

```
metabolites_xgb_test_pr <- metabolites_xgb_final_fit |>
  collect_predictions() |>
  pr_curve(Class, .pred_Group1)

no_skill_xgb <-
sum(tidy(metabolites_xgb_conf_mat)$value[1:2]) /
    sum(tidy(metabolites_xgb_conf_mat)$value)

autoplot(metabolites_xgb_test_pr) +
```

```
    geom_hline(yintercept=no_skill_xgb,
linetype="dashed")
```

The precision-recall curve in Figure 7.20 has some of the same characteristics of the logistic regression PR curve shown in Figure 7.12. Like the logistic regression classifier, the shape of the PR curve suggests that this classifier has problems and is not likely to perform well on new data, especially from Class II/III patients.

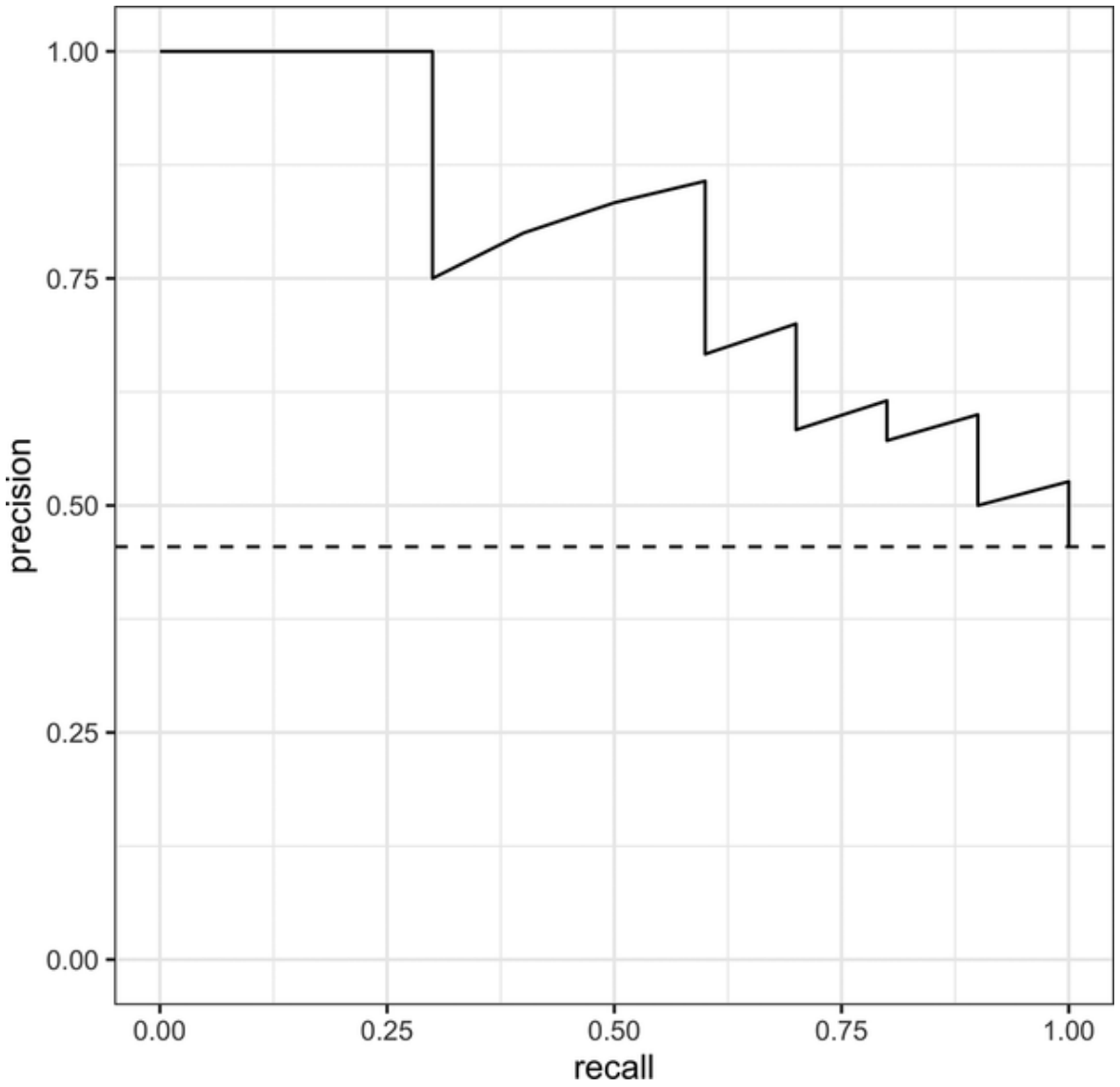


Figure 7.20 Precision-recall curve for the selected model applied to the test data.

It appears from a straight reading of the methods section of this study that the authors violated two key principles of machine learning. The result was a classifier that appeared to perform well. However, the performance appears overstated, and the classification component of the study is unlikely to be reproducible because of information and label leakage.

While 85 patients might seem like a large number of biopsy samples, it is tiny compared to the number of features that

were measured. There are almost 200 times as many features as observations, which, on the face of it, seems like an impossible classification task. I've already shown that variable selection would be of no help since there are possibly thousands of features randomly correlated with the outcome variable. However, tree-based models like gradient boosting and random forest models perform variable selection automatically, and because of this aspect of tree-based learners, the problem became tractable.

The design of `tidymodels` uses deeply nested objects. To get the model from the `last_fit()` object, the `workflow` object has to be extracted using the `extract_workflow()` function, and then the model has to be extracted from the workflow using the `extract_fit_parsnip()` function:

```
metabolites_xgb_final_model <-
metabolites_xgb_final_fit |>
  extract_workflow() |>
  extract_fit_parsnip()
```

The `metabolites_xgb_final_model` object is an XGBoost model of the class `_xgb.Booster`, which can be passed to the `vip()` function to extract the feature importance values directly from the model fit to the training data:

```
xgb_var <- vip(metabolites_xgb_final_model,
num_features = 30)
print(xgb_var)
```

A quick look at the native XGBoost variable importance measure in Figure 7.21, based on the use of variables in splits, shows that a few compounds appear significant, and the model seems to justify further follow-up on several of these as potential biomarkers.

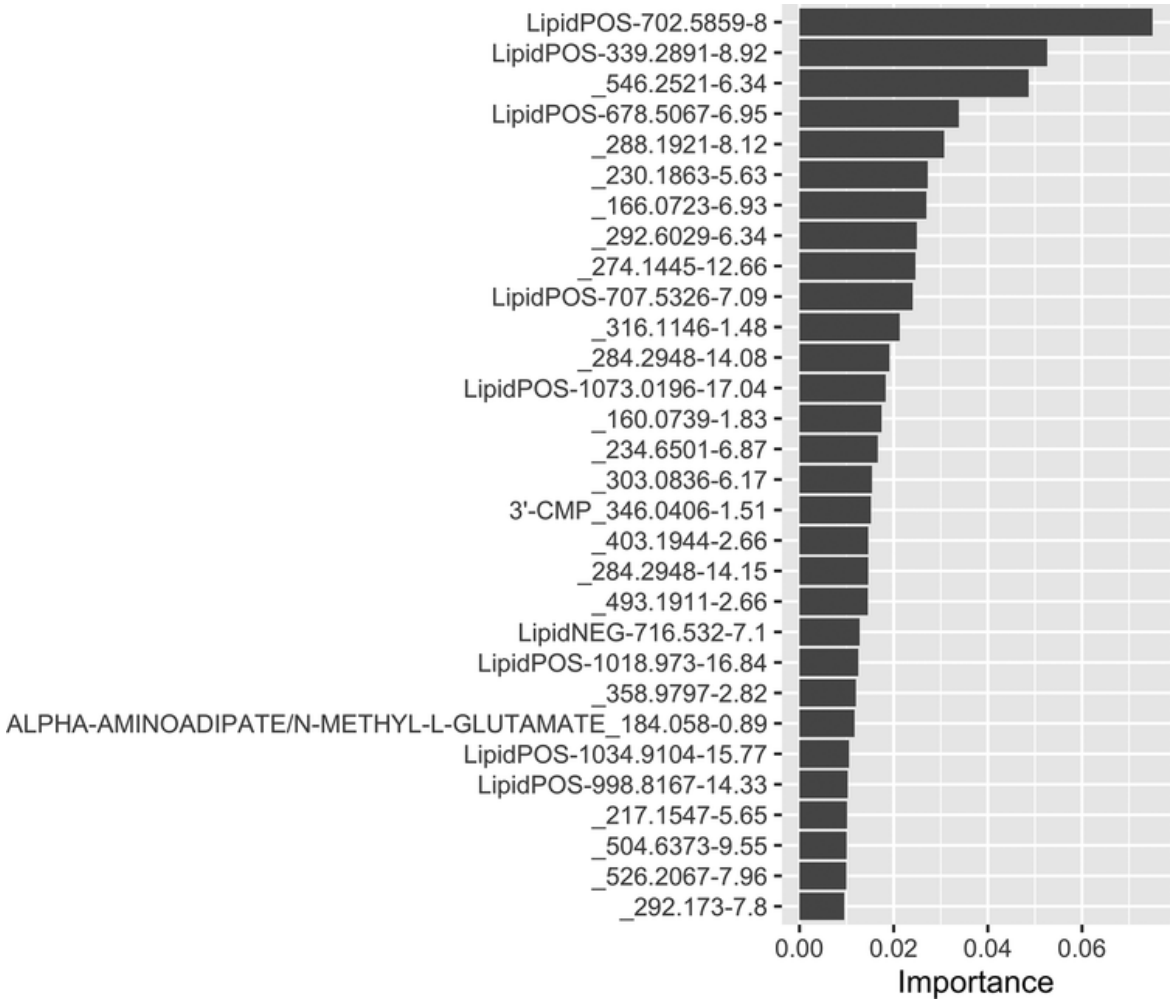


Figure 7.21 Feature importance for the top 30 metabolites using the native XGBoost importance measure.

The compound described as having the highest variable importance (an unidentified lipid) by Yazd et al. [245] made it into the top 30 compounds ranked by variable importance by XGBoost shown in Figure 7.21. Several other compounds are also on both lists. However, the most significant compound found by XGBoost was not even evaluated by Yazd et al. since it was eliminated during the initial feature selection step. It is entirely possible that the overlap between the two top 30 lists is an artifact of the initial normalization and imputation information leakage. It's also possible, although remote, that the initial classification used for

feature selection, a variation on the RF algorithm, managed to avoid selecting features correlated by chance. However, without a test set or cross-validation of the initial classification, it is impossible to tell if the features used for the classification part of the study were spurious or not. While I can speculate about the conclusions of the machine learning process, the approach used in the paper does not represent one that is likely to generate reproducible results.

7.6.6 Other Machine Learning Algorithms

One of the key ideas behind `tidymodels` is that it harmonizes the interface to a large number of data preparation methods and a large number of models. As I'm writing, there are more than 150 different machine learning models available in the `parsnip` package, which can be used as I've described in this chapter [249]. For example, Yazd et al. [245] tested multiple algorithms, including random forests, decision trees, k-nearest neighbors, Naïve Bayes (GaussianNB), logistic regression, and support vector machines, all of which are supported by `parsnip`.

Another important idea is that models of many types can be combined. In RFs, deep trees that tend to overfit are built from random subsets of the data. In the RF, all the learners are of the same type, and it's an ensemble of *homogeneous models*. When different algorithms are used to generate diversity and then combined, it's an ensemble of *heterogeneous models*.

It's worth mentioning the models that are *not* in `parsnip` at the time of writing. While neural network models are available in `parsnip`, they are limited to three-layer networks, or *multilayer perceptrons*. These network architectures allow you to select the number of hidden nodes in the middle layer but do not support multiple hidden layers. So-called *deep learning* networks; therefore, require a completely different

framework, such as the `keras`, `torch`, or `tensorflow` all available from CRAN. Deep learning algorithms have been especially effective for image analysis tasks. I did not include an imaging mass spectrometry example in this chapter. However, deep learning algorithms could be applied to the multidimensional data collected from mass spectra, collected from spatial coordinates rather than the time coordinates found in chromatography.

The `parsnip` package also does not support *reinforcement learning*, in which examples are given to an agent trained based on maximizing a reward function. Reinforcement learning is a powerful tool and is available in R using the `ReinforcementLearning` package, which is also available from CRAN. Analyzing mass spectrometry data using deep learning and other advanced algorithms requires more theoretical background than I can provide here, but can be found in Murphy's *Probabilistic Machine Learning* books [227, 228] and many others.

7.7 Explaining Machine Learning Models

As machine learning models become more complex, it becomes less apparent how the model works on a global basis and, often, more importantly, why a specific observation generated a predicted outcome. In this section, I will discuss how to interpret machine learning models both at the global level of the whole dataset and at the local level of a single observation.

7.7.1 Global Variable Importance

For each of the algorithms discussed so far, I have shown variable importance in terms of the average of the impact of

a variable across all observations. This type of measurement is known as *global explanation*. Global explanations attempt to answer the question: *how does the trained model make predictions?* [235]. In logistic regression, no other explanation is needed beyond the model's parameters since they represent the slopes of lines in a linear model. However, in the SVM example, since a kernel function was applied as part of the model, the model coefficients only tell how important a feature is to the overall model since a hidden nonlinear kernel function is applied to inputs before prediction. In the case of the trees, I didn't even bother to plot the trees themselves because the best model used more than 1000 three-level trees. Since each tree has seven split points, the model has over 7000 parameters that were fit from the data. In that case, looking at the parameters provides no insight into how the model works. However, in the case of tree-based models, you can look at all the nodes, measure which features were used for the splits and how effective they were in performing correct assignments, and use those to estimate the global variable importance for each input parameter. In the case of the metabolite experiment, only the highest impact variables are useful for a global explanation.

7.7.2 Explaining Machine Learning Outcomes for Individual Examples

The global explanation is often not enough, especially when working with people, in this case, cancer patients. The global explanation does not tell a specific patient *why* they were predicted to have Class I rather than Class II/III meningioma. A more powerful *local explainer* is needed for that explanation. I introduced the concept of SHAP at the end of Section 7.6.4. Now, I will show how SHAP can be used to explain specific observations.

7.7.2.1 Local SHAP Explanations

First, I will look at how the model used the features to explain specific instances of the testing data. I'll pick examples from the test data representing each confusion matrix quadrant. Test data elements 1 and 11 are correctly called examples from Group1 and Group2. Test data element 13 is a Group1 observation that was miscalled Group2, and element 3 is a Group2 observation miscalled Group1. By looking at the variable importance for the correct calls, you can see if there are specific compounds that drive specific calls. Since the global variable importance is an average over all instances, it can mask the importance of variables in specific instances.

```
metabolites_xgb_final_fit |>
    collect_predictions() |>
    dplyr::slice(c(1,11,13,3)) |>
    dplyr::select(-id,-.config)
```

```
## # A tibble: 4 x 5
##   .pred_class .pred_Group1 .pred_Group2  .row Class
##   <fct>              <dbl>        <dbl> <int> <fct>
## 1 Group1            0.826         0.174     4 Group1
## 2 Group2            0.0828        0.917    44 Group2
## 3 Group1            0.793         0.207    56 Group2
## 4 Group2            0.264         0.736    13 Group1
```

To use `treeshap()`, the test data must be prepared as a matrix, and the class label `Class` must be removed.

```
metabolite_explain_data <- metabolite_testing_data |>
    select(-Class) |>
    as.matrix()
```

The `treeshap` package uses what the authors call a unified model, which can be used for any tree-based algorithm. The tree model `metabolites_xgb_final_model` The unified model is

then used to compute Shapley values with `treeshap()`, which can then be prepared for visualization with `shapviz()`.

```
unified <- unify(metabolites_xgb_final_model$fit,
metabolite_explain_data)
xgb_shap_values <- treeshap(unified,
metabolite_explain_data, verbose=FALSE)
xgb_shap <- shapviz(xgb_shap_values, X =
metabolite_testing_data)
```

```
p_force_sample_1  <- sv_force(xgb_shap, row_id=1,
bar_label_size=2.5) +
    ggtitle(label="Correct Prediction Class I")

p_force_sample_11 <- sv_force(xgb_shap, row_id=11,
bar_label_size=2.5) +
    ggtitle(label="Correct Prediction Class II/III")

p_correct <- ggarrange(p_force_sample_1,
p_force_sample_11,
          ncol = 1, nrow = 2,
          labels = c("A", "B"))

print(p_correct)
```

Figure 7.22 shows two correct calls from Class I and II/III. One observation is that in both cases, some small amount of importance comes from all the features in the model. These small values combine to be the most significant force in calling the observation a member of one or the other class. Since all the features were normalized and centered, the expected value $E[f(x)]$ is zero, which is another way of saying that the column means are all zero. The output of the SHAP algorithm is shown as $f(x)$, and if it is positive, the observation is called Class I. If it is negative, the observation is called Class II/III.

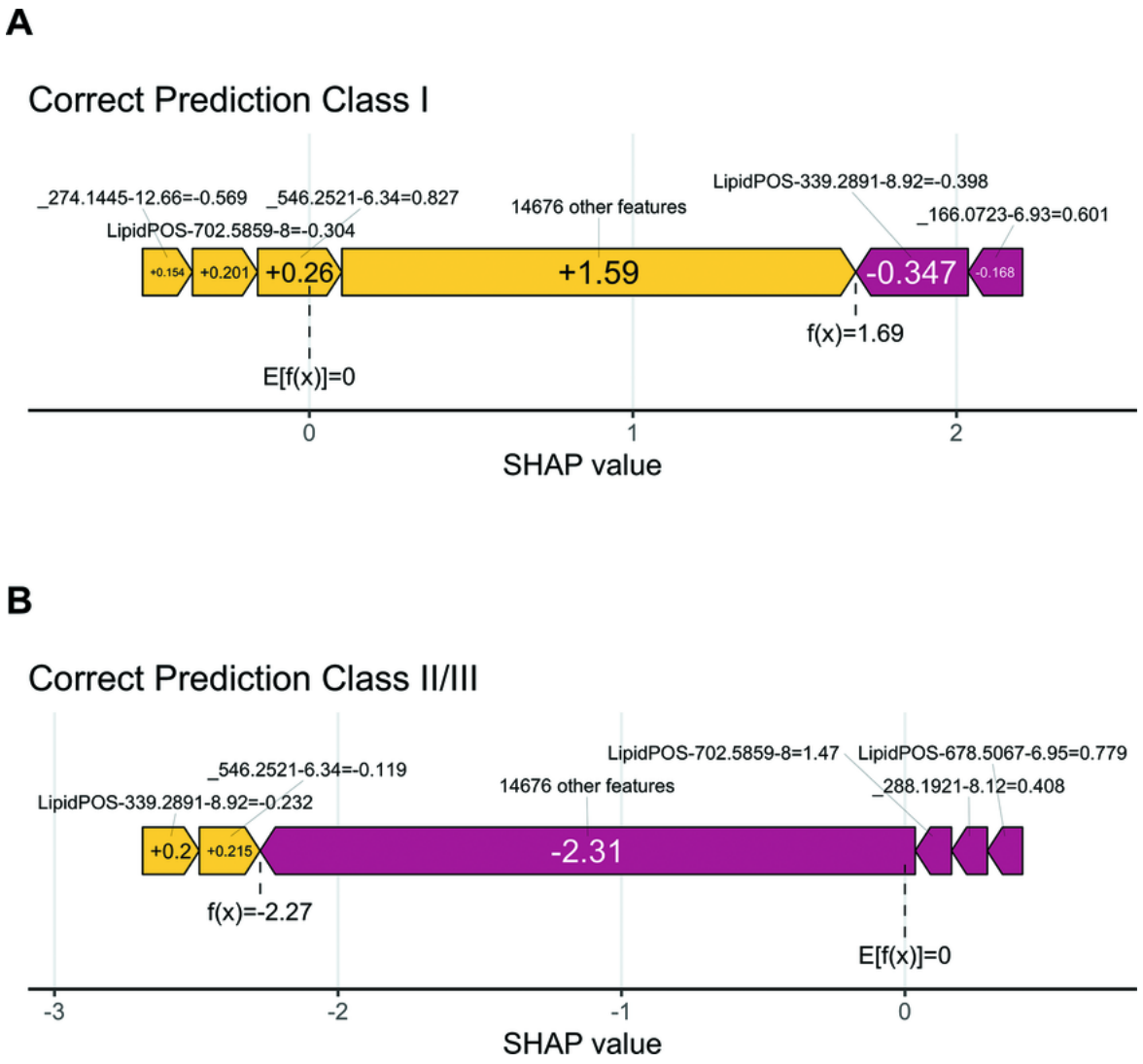


Figure 7.22 Force plots using SHAP values that show which features drove the prediction toward the correct class for (a) Class I and (b) Class II/III.

The force plot in Figure 7.22 shows that several compounds have levels (normalized peak areas) that are higher in one class than another and contribute to the observations' classification. The positive and negative forces (around a mean of zero) suggest following up on any compound that rises above the level where it would be grouped with the rest of the features.

```
p_force_sample_3  <- sv_force(xgb_shap, row_id=3,
bar_label_size=2.5) +
    ggtitle(label="Class I Predicted as Class II/III")
p_force_sample_13 <- sv_force(xgb_shap, row_id=13,
bar_label_size=2.5)+
    ggtitle(label="Class II/III Predicted as Class I",
)

p_incorrect <- ggarrange(p_force_sample_3,
p_force_sample_13,
          ncol = 1, nrow = 2,
          labels = c("A", "B"))

print(p_incorrect)
```

An sample that was actually in Class I but predicted to be Class II/III (Figure 7.23a) shows that the levels of several compounds were so extreme that even without the contribution of the minor features, the observation would have been called Class II/III. In debugging a model of this type, plot (a) suggests looking at any call where the compound that forced the call in the wrong direction is one of the ones shown in the plot. The same thing is true, for another sample (Figure 7.23b). Even without the low-importance features, the responses from three compounds would have pushed this observation into the Class I category.

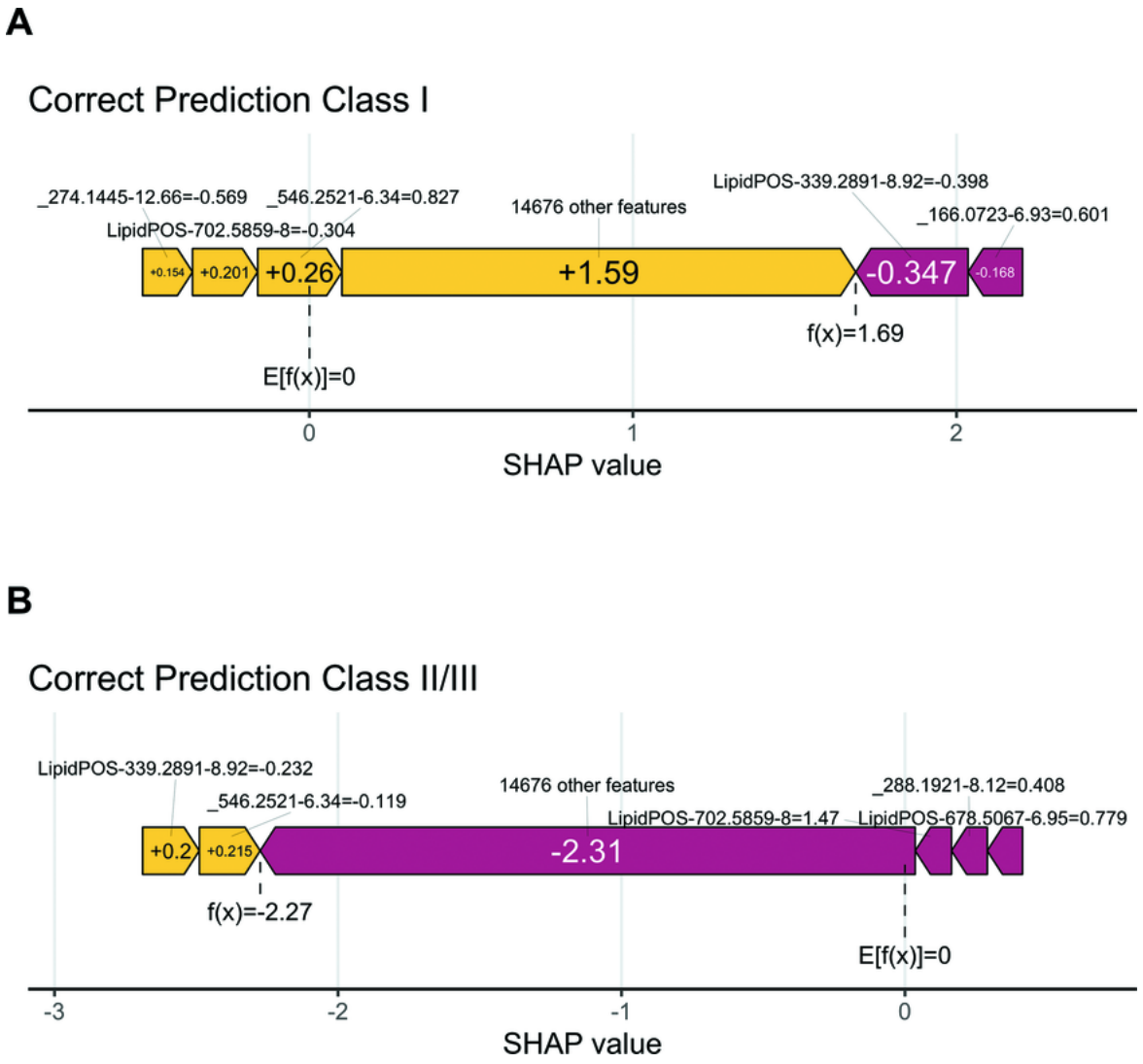


Figure 7.23 Force plots using SHAP values that show which features drove the prediction toward the incorrect class for (a) Class I predicted to be Class II/II and (b) Class II/III predicted to be Class I.

It is encouraging, but not at all inevitable, that the highest average variable significance compounds show up in the force plots as primary drivers. With the identity of the specific patient for each test sample, it should now be possible to look for the spectra in the correctly and incorrectly classified cases and see what signals generated the values used in the model. The additional guidance of the interpretation of the model for specific observations means

that a more thorough understanding of the individual observation and compounds can be developed by targeting the compound evaluation to those that drive specific calls in specific samples.

7.8 Summary

In this chapter, I have given a few examples of unsupervised and supervised machine learning applications to mass spectrometry. I have focused on practical approaches with wide application. At the core of all, statistical learning is the concept of distance. It shows up over and over again. When grouping observations without knowledge of their true grouping, the distance between observations is the only available approach. There is a strong connection between various distance measures and the dot product of the feature vectors of the observation being compared. The dot product is also at the core of the support vector machine algorithm, which connects it with regression, and ultimately unsupervised methods.

The choice of how to measure the difference between predicted values from a model and the true values is subjective, and ultimately distinguishes all the different machine learning algorithms from one another. There is no single distance measure (or error measure) that applies to all data sets. So, while some algorithms are structured to handle tabular data better than, for example, images, there is no one best machine learning algorithm. In the absence of assumptions about the data or the classification task, there is no one best machine learning algorithm, which is called the *No Free Lunch Theorem* (NFLT) [227, 250]. The NFLT is one of the motivations behind the `tidymodels` package: to allow different types of models to be tested within a consistent framework that is compatible with the `tidyverse` principles of

clean, consistent, and reproducible data representation, analysis, and code.

Finally, an appropriate word of warning and encouragement from the great George Box: “All models are wrong, but some models are useful” [251].

References

1 B. E. Winger, R. K. Julian, R. G. Cooks, et al., Surface reactions and surface-induced dissociation of polyatomic ions at self-assembled organic monolayer surfaces, *Journal of the American Chemical Society*, vol. 113, no. 23, pp. 8967–8969, 1991.

2 R. K. Julian, H. P. Reiser, and R. Graham Cooks, Large scale simulation of mass spectra recorded with a quadrupole ion trap mass spectrometer, *International Journal of Mass Spectrometry and Ion Processes*, vol. 123, no. 2, pp. 85–96, 1993.

3 R Core Team, R: A Language and Environment for Statistical Computing, R Foundation for Statistical Computing, Vienna, Austria, manual, 2022. [Online]. Available: https://www.R-project.org/.

4 RStudio IDE. [Online]. Available: https://www.posit.co/ (Accessed 2024-07-10).

5 W. Huber, V. J. Carey, R. Gentleman, et al., Orchestrating high-throughput genomic analysis with Bioconductor, *Nature Methods*, vol. 12, no. 2, pp. 115–121, 2015.

6 M. Morgan and M. Ramos, *BiocManager: Access the Bioconductor Project Package Repository*, 2024, R package version 1.30.23. [Online]. Available: https://bioconductor.github.io/BiocManager/.

7 M. Morgan, BiocManager: Access the Bioconductor project package repository, manual, 2023. [Online]. Available: https://CRAN.R-project.org/package=BiocManager.

8 S. Neumann, L. Gatto, and Q. Kou, *mzR: parser for netCDF, mzXML and mzML and mzIdentML files (mass spectrometry data)*, 2024, R package version 2.38.0. [Online]. Available: https://bioconductor.org/packages/mzR.

9 P. G. A. Pedrioli, J. K. Eng, R. Hubley, et al., A common open representation of mass spectrometry data and its application to proteomics research, *Nature Biotechnology*, vol. 22, no. 11, pp. 1459–1466, 2004.

10 A. Keller, J. Eng, N. Zhang, et al., A uniform proteomics MS/MS analysis platform utilizing open XML file formats, *Molecular Systems Biology*, vol.1, no. 1, pp. 1–8, 2005.

11 D. Kessner, M. Chambers, R. Burke, et al., ProteoWizard: open source software for rapid proteomics tools development, *Bioinformatics*, vol. 24, no. 21, pp. 2534–2536, 2008.

12 L. Martens, M. Chambers, M. Sturm, et al., mzML – A community standard for mass spectrometry data, *Molecular & Cellular Proteomics*, vol. 10, no. 1, R110.000133, 2010.

13 M. C. Chambers, B. Maclean, R. Burke, et al., A cross-platform toolkit for mass spectrometry and proteomics, *Nature Biotechnology*, vol. 30, no. 10, pp. 918–920, 2012.

14 L. Gatto, S. Gibb, and J. Rainer, MSnbase, efficient and elegant R-based processing and visualization of raw mass spectrometry data, *Journal of Proteome Research*, vol. 20, no. 1, pp. 1063–1069, 2021.

15 L. Gatto and K. S. Lilley, MSnbase-an R/Bioconductor package for isobaric tagged mass spectrometry data visualization, processing and quantitation, *Bioinformatics*, vol. 28, no. 2, pp. 288–289, 2012.

16 J. Rainer, A. Vicini, L. Salzer, et al., A modular and expandable ecosystem for metabolomics data annotation in R, *Metabolites*, vol. 12, no. 2, p. 17, 2022.

17 L. Gatto, J. Rainer, and S. Gibb, *MsBackendMgf: Mass Spectrometry Data Backend for Mascot Generic Format (mgf) Files*, 2024, R package version 1.12.0. [Online]. Available: https://bioconductor.org/packages/MsBackendMgf.

18 N. Steffen and J. Rainer, *MsBackendMsp: Mass Spectrometry Data Backend for NIST msp Files*, 2024, R package version 1.8.0. [Online]. Available: https://bioconductor.org/packages/MsBackendMsp.

19 R. Ihaka and R. Gentleman, R: A language for data analysis and graphics, *Journal of Computational and Graphical Statistics*, vol. 5, no. 3, pp. 299–314, 1996.

20 R. A. Becker, J. M. Chambers, and A. R. Wilks, *The New S-language: A Programming Environment for Data Analysis and Graphics*, ser. The Wadsworth & Brooks/Cole statistics/probability series. Pacific Grove, CA: Wadsworth & Brooks/Cole, 1988.

21 G. Sussman and G. Steele, *SCHEME: An Interpreter for Extended Lambda Calculus*, MIT Artificial Intelligence Laboratory, Tech. Rep. AIM-349, 1975. [Online]. Available: https://dspace.mit.edu/handle/1721.1/5794 (Accessed 2022-05-18).

22 D. E. Knuth, *Literate Programming*, ser. CSLI lecture notes. Stanford, CA: Center for the Study of Language and Information, 1992, no. 27.

23 K. Hornik and R. C. Team, *Frequently Asked Questions on R. The Comprehensive R Archive Network*. 5.1: Which add-on packages exist for R? [Online]. Available: https://cran.r-project.org/doc/FAQ/R-FAQ.html#R-Add_002dOn-Packages (Accessed 2022-05-17).

24 D. S. Wishart, D. Tzur, C. Knox, et al., HMDB: The human metabolome database, *Nucleic Acids Research*, vol. 35, no. Database issue, pp. D521–D526, 2007.

25 D. S. Wishart, C. Knox, A. C. Guo, et al., HMDB: A knowledgebase for the human metabolome, *Nucleic Acids Research*, vol. 37, no. Database issue, pp. D603–D610, 2009.

26 D. S. Wishart, T. Jewison, A. C. Guo, et al., HMDB 3.0 – The human metabolome database in 2013, *Nucleic Acids Research*, vol. 41, no. Database issue, pp. D801–D807, 2013.

27 D. S. Wishart, Y. D. Feunang, A. Marcu, et al., HMDB 4.0: The human metabolome database for 2018, *Nucleic Acids Research*, vol. 46, no. D1, pp. D608–D617, 2018.

28 D. S. Wishart, A. Guo, E. Oler, et al., HMDB 5.0: The human metabolome database for 2022, *Nucleic Acids Research*, vol. 50, no. D1, pp. D622–D631, 2022.

29 H. Wickham, *Advanced R*, 2e. Boca Raton: CRC Press/Taylor and Francis Group, 2019.

30 H. Wickham, M. Çetinkaya Rundel, and G. Grolemund, *R for Data Science: Import, Tidy, Transform, Visualize, and*

Model Data, 2e. Sebastopol, CA: O'Reilly Media, Inc., 2023.

31 R for Data Science. [Online]. Available: https://r4ds.had.co.nz/ (Accessed 2022-06-21).

32 Color Universal Design (CUD)/Colorblind Barrier Free. [Online]. Available: https://jfly.uni-koeln.de/color/ (Accessed 2024-02-25).

33 L. Wilkinson and G. Wills, *The Grammar of Graphics*, 2e, ser. Statistics and computing. New York: Springer, 2005.

34 W. Hadley, *ggplot2*. New York, NY: Springer Science + Business Media, LLC, 2016.

35 W. Chang, *R Graphics Cookbook: Practical Recipes for Visualizing Data*, 2e. Beijing, Boston: O'Reilly, 2018.

36 K. Healy, *Data Visualization: A Practical Introduction*. Princeton, NJ: Princeton University Press, 2018.

37 R. C. Gentleman, V. J. Carey, D. M. Bates, et al., Bioconductor: open software development for computational biology and bioinformatics, *Genome Biology*, vol. 5, no. 10, p. R80, 2004.

38 J. M. Chambers, *Software for Data Analysis: Programming with R*, ser. Statistics and Computing. New York, NY: Springer, 2010.

39 Bioconductor – BiocViews. [Online]. Available: https://www.bioconductor.org/packages/release/BiocViews.html (Accessed 2022-07-08).

40 C. Gandrud, *Reproducible Research with R and RStudio*, 3e, ser. The R series. Boca Raton, FL: CRC Press, 2020.

41 What is AnVIL? [Online]. Available: https://anvilproject.org/overview (Accessed 2022-07-12).

42 Y. Xie, *Dynamic Documents with R and Knitr*, 2e. Boca Raton: CRC Press/Taylor & Francis, 2015.

43 J. Gruber, Markdown, 2004. [Online]. Available: https://daringfireball.net/projects/markdown/ (Accessed 2022-07-13).

44 Welcome to MassIVE. [Online]. Available: https://massive.ucsd.edu (Accessed 2021-10-31).

45 D. R. M. Smith, A. R. Uria, E. J. N. Helfrich, et al., An unusual Flavin-dependent halogenase from the metagenome of the marine sponge *Theonella swinhoei* WA, *ACS Chemical Biology*, vol. 12, no. 5, pp. 1281–1287, 2017.

46 Relational database, 2024, page Version ID: 1217554613. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Relational_database&oldid=1217554613 (Accessed 2024-07-02).

47 E. Jones, S. Harden, and M. J. Crawley, *The R Book*, 3e. Hoboken, NJ: Wiley, 2023.

48 H. Xu and M. A. Freitas, A dynamic noise level algorithm for spectral screening of peptide MS/MS spectra, *BMC Bioinformatics*, vol. 11, no. 1, p. 436, 2010.

49 Pandoc. [Online]. Available: https://pandoc.org/ (Accessed 2024-06-18).

50 Unidata | NetCDF. [Online]. Available: https://www.unidata.ucar.edu/software/netcdf/ (Accessed 2022-08-30).

51 The HDF Group – ensuring long-term access and usability of HDF data and supporting users of HDF technologies. [Online]. Available: https://www.hdfgroup.org/ (Accessed 2022-08-30).

52 openspecs office [MS-XLS]: Excel Binary File Format (.xls) Structure. [Online]. Available: https://docs.microsoft.com/en-us/openspecs/office_file_formats/ms-xls/cd03cb5f-ca02-4934-a391-bb674cb8aa06 (Accessed 2022-08-25).

53 openspecs office, [MS-XLSX]: Excel (.xlsx) Extensions to the Office Open XML Spreadsheet ML File Format. [Online]. Available: https://docs.microsoft.com/en-us/openspecs/office_standards/ms-xlsx/2c5dee00-eff2-4b22-92b6-0738acd4475e (Accessed 2022-08-25).

54 E. W. Deutsch, File formats commonly used in mass spectrometry proteomics, *Molecular & Cellular Proteomics*, vol. 11, no. 12, pp. 1612–1621, 2012.

55 C. F. Taylor, N. W. Paton, K. S. Lilley, et al., The minimum information about a proteomics experiment (MIAPE), *Nature Biotechnology*, vol. 25, no. 8, pp. 887–893, 2007.

56 P.-A. Binz, R. Barkovich, R. C. Beavis, et al., Guidelines for reporting the use of mass spectrometry informatics in proteomics, *Nature Biotechnology*, vol. 26, no. 8, p. 862, 2008.

57 A. R. Jones, K. Carroll, D. Knight, et al., Guidelines for reporting the use of column chromatography in proteomics, *Nature Biotechnology*, vol. 28, no. 7, p. 654, 2010.

58 P. J. Domann, S. Akashi, C. Barbas, et al., Guidelines for reporting the use of capillary electrophoresis in

proteomics, *Nature Biotechnology*, vol. 28, no. 7, pp. 654–655, 2010.

59 F. Gibson, L. Anderson, G. Babnigg, et al., Guidelines for reporting the use of gel electrophoresis in proteomics, *Nature Biotechnology*, vol. 26, no. 8, pp. 863–864, 2008.

60 S. Martínez-Bartolomé, E. W. Deutsch, P.-A. Binz, et al., Guidelines for reporting quantitative mass spectrometry based experiments in proteomics, *Journal of Proteomics*, vol. 95, pp. 84–88, 2013.

61 L. Martens, M. Chambers, M. Sturm, et al., mzML – A community standard for mass spectrometry data, *Molecular & Cellular Proteomics*, vol. 10, no. 1, p. R110.000133, 2011.

62 F. Gibson, C. Hoogland, S. Martinez-Bartolomé, et al., The gel electrophoresis markup language (GelML) from the proteomics standards initiative, *PROTEOMICS*, vol. 10, no. 17, pp. 3073–3081, 2010, _eprint:https://onlinelibrary.wiley.com/doi/pdf/10.1002/pmic.201000120.

63 ISA commons | Welcome. [Online]. Available: https://www.isacommons.org/ (Accessed 2022-09-09).

64 S.-A. Sansone, P. Rocca-Serra, D. Field, et al., Toward interoperable bioscience data, *Nature Genetics*, vol. 44, no. 2, pp. 121–126, 2012.

65 ISA Model and Serialization Specifications – ISA Model and Serialization Specifications 1.0 documentation. [Online]. Available: https://isa-specs.readthedocs.io/en/latest/ (Accessed 2022-10-03).

66 P. Rocca-Serra, M. Brandizi, E. Maguire, et al., ISA software suite: supporting standards-compliant experimental annotation and enabling curation at the community level, *Bioinformatics*, vol. 26, no. 18, pp. 2354–2356, 2010.

67 S.-A. Sansone, P. Rocca-Serra, A. Gonzalez-Beltran, et al., Isa Model And Serialization Specifications 1.0, 2016, publisher: Zenodo. [Online]. Available: https://zenodo.org/record/163640 (Accessed 2022-10-03).

68 F. Team, FAIRsharing record for: Investigation Study Assay JSON, 2018. [Online]. Available: https://fairsharing.org/FAIRsharing.yhLgTV (Accessed 2022-10-03).

69 A. González-Beltrán, S. Neumann, E. Maguire, et al., The Risa R/Bioconductor package: integrative data analysis from experimental metadata and back again, *BMC Bioinformatics*, vol. 15 Suppl 1, p. S11, 2014.

70 D. Johnson, D. Batista, K. Cochrane, et al., ISA API: An open platform for interoperable life science experimental metadata, *GigaScience*, vol. 10, no. 9, p. giab060, 2021.

71 N. S. Kale, K. Haug, P. Conesa, et al., MetaboLights: An open-access database repository for metabolomics data, *Current Protocols in Bioinformatics*, vol. 53, pp. 14.13.1–14.13.18, 2016.

72 J. Guo and T. Huan, Comparison of full-scan, data-dependent, and data-independent acquisition modes in liquid chromatography-mass spectrometry-based untargeted metabolomics, *Analytical Chemistry*, vol. 92, no. 12, pp. 8072–8080, 2020.

73 A. Gonzalez-Beltran, A. Kauffmann, S. Neumann, et al., Risa: Converting experimental metadata from ISA-tab into Bioconductor data structures, 2023, R package version 1.44.0. [Online]. Available: https://bioconductor.org/packages/Risa.

74 A. González-Beltrán, S. Neumann, E. Maguire, et al., The Risa R/Bioconductor package: integrative data analysis from experimental metadata and back again, *BMC Bioinformatics*, vol. 15, p. (Suppl 1):S11, 2014. [Online]. Available: http://www.biomedcentral.com/1471-2105/15/S1/S11

75 R. Jackson, N. Matentzoglu, J. A. Overton, et al., OBO Foundry in 2021: operationalizing open data principles to evaluate ontologies, *Database*, vol. 2021, p. baab069, 2021.

76 M. J. Dürst and M. Suignard, Internationalized Resource Identifiers (IRIs), Internet Engineering Task Force, Request for Comments RFC 3987, 2005, num Pages: 46. [Online]. Available: https://datatracker.ietf.org/doc/rfc3987 (Accessed 2022-12-06).

77 V. Sharma, J. Eckels, B. Schilling, et al., Panorama public: a public repository for quantitative data sets processed in Skyline*, *Molecular & Cellular Proteomics*, vol. 17, no. 6, pp. 1239–1244, 2018.

78 Panorama Dashboard: /Panorama Public/2018/gRED – Automated QC of targeted MS data. [Online]. Available: https://panoramaweb.org/targetedmsqc.url (Accessed 2022-10-04).

79 S. Toghi Eshghi, P. Auger, and W. R. Mathews, Quality assessment and interference detection in targeted mass

spectrometry data using machine learning, *Clinical Proteomics*, vol. 15, no. 1, p. 33, 2018.

80 D. Temple Lang, *XML*: Tools for parsing and generating XML within R and S-Plus, 2024, R package version 3.99-0.17. [Online]. Available: https://www.omegahat.net/RSXML/.

81 H. Wickham, J. Hester, and J. Ooms, xml2: Parse XML, 2023, R package version 1.3.6. [Online]. Available: https://xml2.r-lib.org/.

82 ProteoWizard/pwiz: The ProteoWizard Library is a set of software libraries and tools for rapid development of mass spectrometry and proteomic data analysis software. [Online]. Available: https://github.com/ProteoWizard/pwiz (Accessed 2022-10-13).

83 pwiz/pwiz_tools/Skyline/TestUtil/Schemas at master · ProteoWizard/pwiz. [Online]. Available: https://github.com/ProteoWizard/pwiz/tree/master/pwiz_tools/Skyline/TestUtil/Schemas (Accessed 2022-10-13).

84 xpath cover page – W3C. [Online]. Available: https://www.w3.org/TR/xpath/ (Accessed 2022-10-19).

85 Data files. [Online]. Available: https://www.ddbj.nig.ac.jp/metabobank/datafile-e.html (Accessed 2024-03-29).

86 MetaboBank. [Online]. Available: https://mb2.ddbj.nig.ac.jp/ (Accessed 2024-03-29).

87 T. Ara, Y. Kodama, T. Tokimatsu, et al., DDBJ update in 2023: the MetaboBank for metabolomics data and associated metadata, *Nucleic Acids Research*, vol. 52, no. D1, pp. D67–D71, 2024.

88 C. Ubaida-Mohien, S. Spendiff, A. Lyashkov, et al., Unbiased proteomics, histochemistry, and mitochondrial DNA copy number reveal better mitochondrial health in muscle of high-functioning octogenarians, *eLife*, vol. 11, p. e74335, 2022.

89 TMT10plex™ Isobaric Label Reagents and Kits. [Online]. Available: https://www.thermofisher.com/order/catalog/product/90110 (Accessed 2023-01-26).

90 Beta-Galactosidase Digest. [Online]. Available: https://us-store.sciex.com/USD/-/Beta-Galactosidase-Digest-zid4465938 (Accessed 2023-01-25).

91 C. Ubaida-Mohien, M. Gonzalez-Freire, A. Lyashkov, et al., Physical activity associated proteomics of skeletal muscle: being physically active in daily life may protect skeletal muscle from aging, *Frontiers in Physiology*, vol. 10, p. 312, 2019.

92 cRAP protein sequences. [Online]. Available: https://www.thegpm.org/crap/ (Accessed 2023-01-25).

93 M. Wang, J. J. Carver, V. V. Phelan, et al., Sharing and community curation of mass spectrometry data with global natural products social molecular networking, *Nature Biotechnology*, vol. 34, no. 8, pp. 828–837, 2016.

94 Mass Spectrometry File Conversion – GNPS Documentation. [Online]. Available: https://ccms-ucsd.github.io/GNPSDocumentation/fileconversion/#conversion-with-msconvert (Accessed 2023-02-05).

95 ProteoWizard/pwiz, Jan. 2023. [Online]. Available: https://github.com/ProteoWizard/pwiz (Accessed 2023-02-05).

96 Join the GNPS community – GNPS Documentation. [Online]. Available: https://ccms-ucsd.github.io/GNPSDocumentation/gnps_community/ (Accessed 2023-02-05).

97 R. S. McDonald and P. A. Wilks, JCAMP-DX: A Standard form for exchange of infrared spectra in computer readable form, *Applied Spectroscopy*, vol. 42, no. 1, pp. 151–162, 1988.

98 P. Lampen, H. Hillig, A. N. Davies, et al., JCAMP-DX for mass spectrometry, *Applied Spectroscopy*, vol. 48, no. 12, pp. 1545–1552, 1994.

99 Mascot database search | Data file format for mass spectrometry peak lists. [Online]. Available: http://www.matrixscience.com/help/data_file_help.html (Accessed 2023-02-05).

100 J. E. F. Friedl, *Mastering Regular Expressions*, 3e. Sebastapol, CA: O'Reilly, 2006.

101 E. Gamma, R. Helm, R. Johnson, et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional. Part of the Addison-Wesley Professional Computing Series, 1994.

102 MSnbase: MS data processing, visualisation and quantification. [Online]. Available: https://www.bioconductor.org/packages/release/bioc/vignettes/MSnbase/inst/doc/v01-MSnbase-demo.html (Accessed 2023-05-31).

103 D. Sarkar, *Lattice: Multivariate Data Visualization with R*, ser. Use R!. New York: Springer, 2008.

104 R. Tautenhahn, C. Böttcher, and S. Neumann, Highly sensitive feature detection for high resolution LC/MS, *BMC Bioinformatics*, vol. 9, p. 504, 2008.

105 M. Kuhn and H. Wickham, tidymodels: Easily install and load the tidymodels packages, 2024, R package version 1.2.0. [Online]. Available: https://tidymodels.tidymodels.org.

106 M. A. Grayson, (ed.), *Measuring Mass: From Positive Rays to Proteins*. Philadelphia: Chemical Heritage Press, 2002.

107 PubChem, Proton. [Online]. Available: https://pubchem.ncbi.nlm.nih.gov/compound/5460653 (Accessed 2023-07-06).

108 T. Kind and O. Fiehn, Seven golden rules for heuristic filtering of molecular formulas obtained by accurate mass spectrometry, *BMC Bioinformatics*, vol. 8, p. 105, 2007.

109 J. R. D. Laeter, J. K. Böhlke, P. D. Bièvre, et al., Atomic weights of the elements. Review 2000 (IUPAC Technical Report), *Pure and Applied Chemistry*, vol. 75, no. 6, pp. 683–800, 2003.

110 J. R. D. Laeter, J. K. Böhlke, P. D. Bièvre, et al., Errata: Atomic weights of the elements: Review 2000 (IUPAC Technical Report), *Pure and Applied Chemistry*, vol. 81, no. 8, pp. 1535–1536, 2009.

111 T. Prohaska, J. Irrgeher, J. Benefield, et al., Standard atomic weights of the elements 2021 (IUPAC Technical Report), *Pure and Applied Chemistry*, vol. 94, no. 5, pp. 573–600, 2022.

112 A. L. Rockwood and M. Palmblad, Isotopic distributions, In: *Mass Spectrometry Data Analysis in Proteomics*, (R. Matthiesen, ed.). New York, NY: Springer New York, vol. 2051, pp. 79–114, 2020.

113 H. Budzikiewicz and R. D. Grigsby, Mass spectrometry and isotopes: A century of research and discussion, *Mass Spectrometry Reviews*, vol. 25, no. 1, pp. 146–157, 2006.

114 J. J. Thomson, Bakerian Lecture: Rays of positive electricity, *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, vol. 89, no. 607, pp. 1–20, 1913.

115 F. Aston, CXIX. The mass-spectra of chemical elements. – Part VI. Accelerated anode rays continued, *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 49, no. 294, pp. 1191–1201, 1925.

116 D. Valkenborg, I. Mertens, F. Lemière, et al., The isotopic distribution conundrum, *Mass Spectrometry Reviews*, vol. 31, no. 1, pp. 96–109, 2012.

117 M. K. Łącki, M. Startek, D. Valkenborg, et al., IsoSpec: Hyperfast fine structure calculator, *Analytical Chemistry*, vol. 89, no. 6, pp. 3272–3277, 2017.

118 M. K. Łącki, D. Valkenborg, and M. P. Startek, IsoSpec2: Ultrafast fine structure calculator, *Analytical Chemistry*, vol. 92, no. 14, pp. 9472–9475, 2020.

119 S. Böcker and Z. Lipták, A fast and simple algorithm for the money changing problem, *Algorithmica*, vol. 48, no. 4, pp. 413–432, 2007.

120 S. Böcker, M. Letzel, Z. Lipták, et al., Decomposing metabolomic isotope patterns, In: *Proc. of Workshop on Algorithms in Bioinformatics (WABI 2006*), ser. Lect. Notes Comput. Sci., vol. 4175. Berlin: Springer, pp. 12–23, 2006.

121 S. Böcker, M. C. Letzel, Z. Lipták, et al., SIRIUS: Decomposing isotope patterns for metabolite identification†, *Bioinformatics*, vol. 25, no. 2, pp. 218–224, 2009.

122 S. Böcker, Z. Lipták, M. Martin, et al., Decomp – from interpreting mass spectrometry peaks to solving the money changing problem, *Bioinformatics*, vol. 24, no. 4, pp. 591–593, 2008.

123 Deprecate Rdisop in BioC and archive this repo. [Online]. Available: https://github.com/sneumann/Rdisop/issues/23 (Accessed 2023-07-17).

124 UCSD/CCMS – Spectrum Library. [Online]. Available: https://gnps.ucsd.edu/ProteoSAFe/gnpslibraryspectrum.jsp?SpectrumID=CCMSLIB00011429539#%7B%7D (Accessed 2023-07-17).

125 N. Huang, M. M. Siegel, G. H. Kruppa, et al., Automation of a Fourier transform ion cyclotron resonance mass spectrometer for acquisition, analysis, and e-mailing of high-resolution exact-mass electrospray ionization mass spectral data, *Journal of the American Society for Mass Spectrometry*, vol. 10, no. 11, pp. 1166–1173, 1999.

126 M. R. Blumer, C. H. Chang, E. Brayfindley, et al., Mass spectrometry adduct calculator, *Journal of chemical*

information and modeling, vol. 61, no. 12, pp. 5721–5725, 2021.

127 Z. Zhang, B. Yan, K. Liu, et al., Fragmentation pathways of heroin-related alkaloids revealed by ion trap and quadrupole time-of-flight tandem mass spectrometry, *Rapid Communications in Mass Spectrometry*, vol. 22, no. 18, pp. 2851–2862, 2008.

128 P. G. Hatcher, K. J. Dria, S. Kim, et al., Modern analytical studies of humic substances, *Soil Science*, vol. 166, no. 11, pp. 770–794, 2001.

129 N. W. Green and E. M. Perdue, Fast graphically inspired algorithm for assignment of molecular formulae in ultrahigh resolution mass spectrometry, *Analytical Chemistry*, vol. 87, no. 10, pp. 5086–5094, 2015.

130 S. K. Schum, L. E. Brown, and L. R. Mazzoleni, MFAssignR: Molecular formula assignment software for ultrahigh resolution mass spectrometry analysis of environmental complex mixtures, *Environmental Research*, vol. 191, p. 110114, 2020.

131 E. M. Perdue and N. W. Green, Isobaric molecular formulae of C, H, and O: A view from the negative quadrants of van Krevelen space, *Analytical Chemistry*, vol. 87, no. 10, pp. 5079–5085, 2015.

132 M. J. Helf, B. W. Fox, A. B. Artyukhin, et al., Comparative metabolomics with Metaboseek reveals functions of a conserved fat metabolism pathway in *C. elegans*, *Nature Communications*, vol. 13, no. 1, p. 782, 2022.

133 M. J. Helf, mjhelf/MassTools: MassTools Version 0.2.12, 2021. [Online]. Available:

https://zenodo.org/record/5725620 (Accessed 2023-08-08).

134 M. Kuhn and J. Silge, *Tidy Modeling with R: A Framework for Modeling in the Tidyverse*, 1e, Beijing, Boston, Farnham, Sebastopol, Tokyo: O'Reilly, 2022.

135 S. Couch, A. Bray, C. Ismay, et al., infer: An R package for tidyverse-friendly statistical inference, *Journal of Open Source Software*, vol. 6, no. 65, p. 3661, 2021.

136 C. Ismay and A. Y.-S. Kim, *Statistical Inference via Data Science: A ModernDive into R and the Tidyverse*, ser. Chapman & Hall/CRC the R Series. Boca Raton: CRC Press/Taylor & Francis Group, 2020.

137 A. Downey, Probably overthinking it: There is only one test! 2011. [Online]. Available: http://allendowney.blogspot.com/2011/05/there-is-only-one-test.html (Accessed 2023-10-05).

138 A. Downey, Probably overthinking it: There is still only one test. [Online]. Available: http://allendowney.blogspot.com/2016/06/there-is-still-only-one-test.html (Accessed 2023-10-05).

139 B. Bolstad, R. Irizarry, M. Åstrand, et al., A comparison of normalization methods for high density oligonucleotide array data based on variance and bias, *Bioinformatics*, vol. 19, no. 2, pp. 185–193, 2003.

140 B. Bolstad, preprocessCore: A collection of pre-processing functions, manual, 2023. [Online]. Available: https://bioconductor.org/packages/preprocessCore.

141 D. Thomas and A. Hunt, *The Pragmatic Programmer: Your Journey to Mastery*, 2e, 20th anniversary edition. Boston: Addison-Wesley, 2020.

142 D. E. Knuth, Structured programming with go to statements, *ACM Computing Surveys*, vol. 6, no. 4, pp. 261–301, 1974.

143 The Wrong Abstraction. [Online]. Available: https://sandimetz.com/blog/2016/1/20/the-wrong-abstraction (Accessed 2024-07-05).

144 MT-CO2 – Cytochrome c oxidase subunit 2 – Homo sapiens (Human)|UniProtKB|UniProt. [Online]. Available: https://www.uniprot.org/uniprotkb/P00403/entry (Accessed 2023-10-06).

145 S. M. Herbrich, R. N. Cole, K. P. West, et al., Statistical inference from multiple iTRAQ experiments without using common reference standards, *Journal of Proteome Research*, vol. 12, no. 2, pp. 594–604, 2013.

146 K. Kammers, R. N. Cole, C. Tiengwe, et al., Detecting significant changes in protein abundance, *EuPA Open Proteomics*, vol. 7, pp. 11–19, 2015.

147 D. N. Perkins, D. J. C. Pappin, D. M. Creasy, et al., Probability-based protein identification by searching sequence databases using mass spectrometry data, *ELECTROPHORESIS*, vol. 20, no. 18, pp. 3551–3567, 1999.

148 Chapter 40: Signal processing, In: *Handbook of Chemometrics and Qualimetrics: Part B*, ser. Data Handling in Science and Technology (eds., B. Vandeginste, D. Massart, L. Buydens, S. de Jong, P. Lewi, and J. Smeyers-Verbeke) Amsterdam: Elsevier, vol. 20, pp. 507–574, 1998.

149 K. H. Liland and B.-H. Mevik, baseline: Baseline Correction of Spectra, 2023, R package version 1.3–5.

[Online]. Available: https://github.com/khliland/baseline/.

150 K. H. Liland, T. Almøy, and B.-H. Mevik, Optimal choice of baseline correction for multivariate calibration of spectra, *Applied Spectroscopy*, vol. 64, no. 9, pp. 1007–1016, 2010.

151 P. H. C. Eilers, A perfect smoother, *Analytical Chemistry*, vol. 75, no. 14, pp. 3631–3636, 2003.

152 P. H. C. Eilers, Parametric time warping, *Analytical Chemistry*, vol. 76, no. 2, pp. 404–411, 2004.

153 P. H. C. Eilers and H. F. Boelens, Baseline correction with asymmetric least squares smoothing, 2005.

154 E. Whittaker and G. Robinson, A method of graduation based on probability. In: *The Calculus of Observations: An Introduction to Numerical Analysis*. New York: Dover, pp. 303–306, 1967.

155 Z.-M. Zhang, S. Chen, and Y.-Z. Liang, Baseline correction using adaptive iteratively reweighted penalized least squares, *The Analyst*, vol. 135, no. 5, pp. 1138–1146, 2010.

156 S.-J. Baek, A. Park, Y.-J. Ahn, et al., Baseline correction using asymmetrically reweighted penalized least squares smoothing, *Analyst*, vol. 140, no. 1, pp. 250–257, 2014.

157 A. Savitzky and M. J. E. Golay, Smoothing and differentiation of data by simplified least squares procedures. *Analytical Chemistry*, vol. 36, no. 8, pp. 1627–1639, 1964.

158 A. Felinger, Peak detection by derivatives. In: *Data Analysis and Signal Processing in Chromatography*, ser.

Data handling in science and technology. Amsterdam [Netherlands]; New York: Elsevier, no. v. 21, pp. 185–189, 1998.

159 C. Lanczos, Chapter 5: Data analysis. Section 5: The difficulties of a difference table. In: *Applied Analysis*. New York: Dover Publications, pp. 313–315, 1988.

160 T. H. Edwards and P. D. Willson, Digital least squares smoothing of spectra, *Applied Spectroscopy*, vol. 28, no. 6, pp. 541–545, 1974.

161 C. G. Enke and T. A. Nieman, Signal-to-noise ratio enhancement by least-squares polynomial smoothing, *Analytical Chemistry*, vol. 48, no. 8, pp. 705A–712A, 1976.

162 R. N. Bracewell, *The Fourier Transform and Its Applications*, 2e, ser. McGraw-Hill series in electrical engineering. New York: McGraw-Hill, 1986.

163 C. Lanczos, Chapter 5: Data analysis. Section 11: Smoothing in the large by Fouier analysis. In: *Applied Analysis*. New York: Dover Publications, pp. 331–336, 1988.

164 T. B. Sprague, The graphic method of adjusting mortality tables. – A description of its objects, and its advantages as compared with other methods, and an application of it to obtain a Graduated Mortality Table from Mr. A. J. Finlaison’s Observations on the Mortality of the Female Government Annuitants, 4 years and upwards after purchase, *Journal of the Institute of Actuaries*, vol. 26, no. 2, pp. 77–120, 1886.

165 S. S. Shapiro and M. B. Wilk, An analysis of variance test for normality (complete samples), *Biometrika*, vol.

52, no. 3/4, p. 591, 1965.

166 J. P. Royston, “An extension of Shapiro and Wilk’s W test for normality to large samples, *Applied Statistics*, vol. 31, no. 2, p. 115, 1982.

167 K. M. Ramachandran and C. P. Tsokos, Categorical data analysis and goodness-of-fit tests and applications, In: *Mathematical Statistics with Applications in R*. Elsevier, pp. 461–490, 2021.

168 P. Royston, Remark AS R94: A remark on algorithm AS 181: The W-test for normality, *Applied Statistics*, vol. 44, no. 4, p. 547, 1995.

169 S. P. Millard, *EnvStats: An R Package for Environmental Statistics*, 2e. New York, NY: Springer, 2013.

170 J. R. Michael and W. R. Schucany, Analysis of data from censored samples, In: *Goodness-of-fit Techniques*, ser. Statistics, textbooks and monographs (eds., R. B. D’Agostino and M. A. Stephens). New York: M. Dekker, no. vol. 68, pp. 461–496, 1986.

171 L. Hashimoto and R. Trussell, Evaluating water quality data near the detection limit, Las Vegas, 1983.

172 R. J. Gilliom and D. R. Helsel, Estimation of distributional parameters for censored trace level water quality data: 1. Estimation techniques, *Water Resources Research*, vol. 22, no. 2, pp. 135–146, 1986.

173 D. R. Helsel and R. J. Gilliom, Estimation of distributional parameters for censored trace level water quality data: 2. Verification and applications, *Water Resources Research*, vol. 22, no. 2, pp. 147–155, 1986.

174 A. H. El-Shaarawi, Inferences about the mean from censored water quality data, *Water Resources Research*, vol. 25, no. 4, pp. 685–690, 1989.

175 D. R. Helsel and D. R. Helsel, *Statistics for Censored Environmental Data Using Minitab and R*, 2e, ser. Wiley series in statistics in practice. Hoboken, NJ: Wiley, 2012.

176 USEPA, Statistical analysis of groundwater monitoring data at RDRA facilities: unified guidance, Office of Resource Conservation and Recovery, Program Implementation and Information Division, US Environmental Protection Agency, Washington D.C., Tech. Rep. EPA-530-R-09-007, 2009.

177 P. Du, W. A. Kibbe, and S. M. Lin, Improved peak detection in mass spectrum by incorporating continuous wavelet transform-based pattern matching, *Bioinformatics*, vol. 22, no. 17, pp. 2059–2065, 2006.

178 W. Zhang, J. Zhou, M. Yang, et al., Efficient mass spectrometry peak detection by combining resolution enhancement and image segmentation, *Analytical Letters*, vol. 57, no. 8, pp. 1227–1240, 2023.

179 C. Bai, S. Xu, J. Tang, et al., A 'shape-orientated' algorithm employing an adapted Marr wavelet and shape matching index improves the performance of continuous wavelet transform for chromatographic peak detection and quantification, *Journal of Chromatography A*, vol. 1673, p. 463086, 2022.

180 J. M. Gregoire, D. Dale, and R. B. Van Dover, A wavelet transform algorithm for peak detection and application to powder x-ray diffraction data, *Review of Scientific Instruments*, vol. 82, no. 1, p. 015105, 2011.

181 S. V. Vaseghi, *Advanced Digital Signal Processing and Noise Reduction*, 4e. Chichester, UK: Wiley, 2008.

182 J. O. Ramsay and B. W. Silverman, *Functional Data Analysis*, 2e, ser. Springer series in statistics. New York: Springer, 2005.

183 R. L. Eubank, *Nonparametric Regression and Spline Smoothing*, 2e, ser. Statistics, textbooks and monographs. New York: Marcel Dekker, no. v. 157, 1999.

184 T. Hastie, R. Tibshirani, and J. H. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2e, ser. Springer series in statistics. New York, NY: Springer, 2009.

185 J. N. Morgan and J. A. Sonquist, Problems in the analysis of survey data, and a proposal, *Journal of the American Statistical Association*, vol. 58, no. 302, pp. 415–434, 1963.

186 J. W. Cooley and J. W. Tukey, An algorithm for the machine calculation of complex Fourier series, *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965.

187 M. Schmid, D. Rath, and U. Diebold, Why and how savitzky–golay filters should be replaced, *ACS Measurement Science Au*, vol. 2, no. 2, pp. 185–196, 2022.

188 A. Felinger, *Data Analysis and Signal Processing in Chromatography*, ser. Data handling in science and technology. Amsterdam [Netherlands]; New York: Elsevier, no. v. 21, 1998.

189 C. Lanczos, *Applied Analysis*. New York: Dover Publications, 1988.

190 F. Harris, On the use of windows for harmonic analysis with the discrete Fourier transform, *Proceedings of the IEEE*, vol. 66, no. 1, pp. 51–83, 1978.

191 J. Kaiser and R. Schafer, On the use of the I0-sinh window for spectrum analysis, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 28, no. 1, pp. 105–107, 1980.

192 C. Lanczos, “Chapter 4: Quadrature Methods.” In: *Applied Analysis*. New York: Dover Publications, pp. 379–437, 1988.

193 M. Febrero-Bande and M. O. D. L. Fuente, Statistical computing in functional data analysis: The R Package fda.usc, *Journal of Statistical Software*, vol. 51, pp. 1–28, 2012.

194 W. N. Venables and B. D. Ripley, *Modern Applied Statistics with S*, 4e, ser. Statistics and computing. New York: Springer, 2002.

195 Padé approximant, 2024, page Version ID: 1205633365. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Pad%C3%A9_approximant&oldid=1205633365 (Accessed 2024-02-19).

196 W. H. Press (ed.), *Numerical Recipes: The Art of Scientific Computing*, 3e. Cambridge, UK; New York: Cambridge University Press, 2007.

197 E. Pagliano, Z. Mester, and J. Meija, Calibration graphs in isotope dilution mass spectrometry, *Analytica Chimica Acta*, vol. 896, pp. 63–67, 2015.

198 J. Nocedal and S. J. Wright, Algorithms for non-linear least-squares problems, In: *Numerical Optimization*, 2e,

ser. Springer series in operations research. New York: Springer, pp. 254–265, 2006. oCLC: ocm68629100.

199 R. C. Mittelhammer, G. G. Judge, and D. J. Miller, *Econometric Foundations*, 1e. Cambridge: Cambridge University Press, 2000.

200 tidymodels. [Online]. Available: https://www.tidymodels.org/ (Accessed 2024-07-15).

201 M. Kuhn and J. Silge, Tidy modeling with R, 2023, version 1.0.0. [Online]. Available: https://www.tmwr.org/ (Accessed 2024-03-08).

202 Tidymodels packages – tidymodels. [Online]. Available: https://www.tidymodels.org/packages/ (Accessed 2024-07-15).

203 M. Kuhn and K. Johnson, *Feature Engineering and Selection: A Practical Approach for Predictive Models*. Boca Raton, FL: CRC Press, 2020.

204 G. James, D. Witten, T. Hastie, et al., *An Introduction to Statistical Learning: With Applications in R*, ser. Springer texts in statistics. New York: Springer, no. 103, 2013.

205 H. Yang, Y. Zhou, Q. Luo, et al., L-leucine increases the sensitivity of drug-resistant Salmonella to sarafloxacin by stimulating central carbon metabolism and increasing intracellular reactive oxygen species level, *Frontiers in Microbiology*, vol. 14, 2023.

206 S. Pawley, M. Kuhn, and R. Jacques-Hamilton, colino: Recipes steps for supervised filter-based feature selection, manual, 2024. [Online]. Available: https://stevenpawley.github.io/colino (Accessed 2024-03-11).

207 J. MacQueen, Some methods for classification and analysis of multivariate observations, In: *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*. University of California Press, vol. 5.1, pp. 281–298, 1967.

208 J. A. Hartigan and M. A. Wong, Algorithm AS 136: A K-means clustering algorithm, *Applied Statistics*, vol. 28, no. 1, p. 100, 1979.

209 E. Hvitfeldt and K. Bodwin, tidyclust: A common API to clustering, manual, 2024. [Online]. Available: https://CRAN.R-project.org/package=tidyclust (Accessed 2024-05-20).

210 L. R. Crawford and J. D. Morrison, Computer methods in analytical mass spectrometry. Identification of an unknown compound in a catalog, *Analytical Chemistry*, vol. 40, no. 10, pp. 1464–1469, 1968.

211 F. W. McLafferty, Performance prediction and evaluation of systems for computer identification of spectra, *Analytical Chemistry*, vol. 49, no. 9, pp. 1441–1443, 1977.

212 D. Bajusz, A. Rácz, and K. Héberger, Why is Tanimoto index an appropriate choice for fingerprint-based similarity calculations? *Journal of Cheminformatics*, vol. 7, no. 1, p. 20, 2015.

213 S. E. Stein and D. R. Scott, Optimization and testing of mass spectral library search algorithms for compound identification, *Journal of the American Society for Mass Spectrometry*, vol. 5, no. 9, pp. 859–866, 1994.

214 K. X. Wan, I. Vidavsky, and M. L. Gross, Comparing similar spectra: From similarity index to spectral

contrast angle, *Journal of the American Society for Mass Spectrometry*, vol. 13, no. 1, pp. 85–88, 2002.

215 Z. B. Alfassi, On the normalization of a mass spectrum for comparison of two spectra, *Journal of the American Society for Mass Spectrometry*, vol. 15, no. 3, pp. 385–387, 2004.

216 H.-G. Drost, Philentropy: information theory and distance quantification with R, *Journal of Open Source Software*, vol. 3, no. 26, p. 765, 2018.

217 H. Horai, M. Arita, S. Kanaya, Y, et al., MassBank: a public repository for sharing mass spectral data for life sciences, *Journal of Mass Spectrometry*, vol. 45, no. 7, pp. 703–714, 2010.

218 MassBank of North America. [Online]. Available: https://mona.fiehnlab.ucdavis.edu/ (Accessed 2024-05-02).

219 chemdata:nistlibs. [Online]. Available: https://chemdata.nist.gov/dokuwiki/doku.php?id=chemdata:nistlibs (Accessed 2024-05-09).

220 A. Cereto-Massagué, M. J. Ojeda, C. Valls, et al., Molecular fingerprint similarity search in virtual screening. *Methods*, vol 71, pp. 58–63, 2015.

221 J. D. Holliday, N. Salim, M. Whittle, et al., Analysis and display of the size dependence of chemical similarity coefficients, *Journal of Chemical Information and Computer Sciences*, vol. 43, no. 3, pp. 819–828, 2003.

222 A. Bender, J. L. Jenkins, J. Scheiber, et al., How similar are similarity searching methods? A principal component analysis of molecular descriptor space, *Journal of*

Chemical Information and Modeling, vol. 49, no. 1, pp. 108–119, 2009.

223 G. T. Rasmussen and T. L. Isenhour, The evaluation of mass spectral search algorithms, *Journal of Chemical Information and Computer Sciences*, vol. 19, no. 3, pp. 179–186, 1979.

224 F. W. McLafferty, M.-Y. Zhang, D. B. Stauffer, et al., Comparison of algorithms and databases for matching unknown mass spectra, *Journal of the American Society for Mass Spectrometry*, vol. 9, no. 1, pp. 92–95, 1998.

225 Z. B. Alfassi, On the comparison of different tests for identification of a compound from its mass spectrum, *Journal of the American Society for Mass Spectrometry*, vol. 14, no. 3, pp. 262–264, 2003.

226 T. Hastie, R. Tibshirani, and J. H. Friedman, Chapter 7: Model assesment and selection. Section 7.3: The bias-variance decomposition. In: *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2e, ser. Springer series in statistics. New York, NY: Springer, pp. 233–228, 2009

227 K. P. Murphy, *Probabilistic Machine Learning: An Introduction*, ser. Adaptive computation and machine learning. Cambridge, Massachusetts, London, England: The MIT Press, 2022.

228 K. P. Murphy, *Probabilistic Machine Learning: Advanced Topics*, ser. Adaptive computation and machine learning. Cambridge, Massachusetts, London, England: The MIT Press, 2023.

229 M. H. Kutner, C. Nachtsheim, and J. Neter, *Applied Linear Regression Models*, 4e, Boston; New York:

McGraw-Hill/Irwin, 2004.

230 B. Greenwell, M. and B. Boehmke, C., Variable Importance Plots – An Introduction to the vip Package, *The R Journal*, vol. 12, no. 1, p. 343, 2020.

231 M. Mayer and D. Watson, kernelshap: Kernel SHAP, manual, 2024. [Online]. Available: https://CRAN.R-project.org/package=kernelshap.

232 S. M. Lundberg and S.-I. Lee, A unified approach to interpreting model predictions, In: *Advances in neural information processing systems 30: 31st Annual Conference on Neural Information Processing Systems (NIPS 2017):* Long Beach, California, USA, *4-9 December 2017* (eds., U. V. Luxburg, I. Guyon, S. Bengio, H. Wallach, R. Fergus, S. V. N. Vishwanathan, R. Garnett, and Neural Information Processing Systems Foundation), Red Hook, NY: Curran Associates, Inc., pp. 4765–4774, 2018.

233 I. Covert and S.-I. Lee, Improving KernelSHAP: Practical shapley value estimation using linear regression, In: *Proceedings of the 24th International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of machine learning research, (eds., A. Banerjee and K. Fukumizu), vol. 130. PMLR, pp. 3457–3465, 2021.

234 L. S. Shapley, *A Value for N-Person Games*. RAND Corporation, 1952.

235 C. Molnar, *Interpretable Machine Learning: A Guide for Making Black Box Models Explainable*. Victoria, BC: Leanpub, 2020.

236 K. Komisarczyk, P. Kozminski, S. Maksymiuk, et al., treeshap: Compute SHAP values for your tree-based models using the 'TreeSHAP' algorithm, manual, 2024. [Online]. Available: https://CRAN.R-project.org/package=treeshap.

237 A. Karatzoglou, A. Smola, K. Hornik, et al., kernlab - An S4 package for Kernel methods in R, *Journal of Statistical Software*, vol. 11, pp. 1–20, 2004.

238 M. Mayer, shapviz: SHAP visualizations, manual, 2024. [Online]. Available: https://CRAN.R-project.org/package=shapviz.

239 Clinical and Laboratory Standards Institute (CLSI). Liquid Chromatography-Mass Spectrometry Methods. 2nd CLSI guideline C62. Clinical and Laboratory Standards Institute; 2022.

240 J. H. Friedman, Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, vol. 29, no. 5, 2001.

241 P. Li, Robust logitboost and adaptive base class (ABC) logitboost, In: *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, ser. UAI'10. Arlington, Virginia, USA: AUAI Press, pp. 302–311, 2010.

242 T. Chen and C. Guestrin, XGBoost: A scalable tree boosting system, In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 785–794, 2016, arXiv: 1603.02754. [Online]. Available: http://arxiv.org/abs/1603.02754 (Accessed 2021-01-21).

243 G. Kunapuli, *Ensemble Methods for Machine Learning*. Shelter Island, NY: Manning, 2023.

244 C. Adam-Bourdarios, G. Cowan, C. Germain-Renaud, et al., The higgs machine learning challenge, *Journal of Physics: Conference Series*, vol. 664, no. 7, p. 072015, 2015.

245 H. Safari Yazd, S. F. Bazargani, G. Fitzpatrick, et al., Metabolomic and lipidomic characterization of meningioma grades using LC–HRMS and machine learning, *Journal of the American Society for Mass Spectrometry*, vol. 34, no. 10, pp. 2187–2198, 2023.

246 M. A. Bouke and A. Abdullah, An empirical study of pattern leakage impact during data preprocessing on machine learning-based intrusion detection models reliability, *Expert Systems with Applications*, vol. 230, p. 120715, 2023.

247 R. C. Staudemeyer and C. W. Omlin, Extracting salient features for network intrusion detection using machine learning methods, *South African Computer Journal*, vol. 52, 2014.

248 M. D. McKay, R. J. Beckman, and W. J. Conover, Comparison of three methods for selecting values of input variables in the analysis of output from a computer code, *Technometrics*, vol. 21, no. 2, pp. 239–245, 1979.

249 tidymodels – Search parsnip models. [Online]. Available: https://www.tidymodels.org/find/parsnip/#models (Accessed 2024-06-01).

250 R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern Classification*, 2e. New York: Wiley, 2001.

251 G. E. P. Box and N. R. Draper, *Empirical Model-Building and Response Surfaces*, ser. Wiley series in probability and mathematical statistics. Applied probability and statistics. New York: Wiley, p. 424, 1987.

Index

Note: Page numbers followed by *f* refer to figures and *t* refers to tables.

a

b

C

d

e

f

g

h

i

j

k

l

m

n

O

p

q

r

S

t

u

v

W

X

y

z

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley's ebook EULA.