



# Ensemble Methods for Machine Learning

Gautam Kunapuli

MEAP

 MANNING



**MEAP Edition**  
**Manning Early Access Program**  
**Ensemble Methods for Machine Learning**  
**Version 6**

Copyright 2022 Manning Publications

For more information on this and other Manning titles go to  
[manning.com](https://manning.com)

# welcome

---

Thank you for purchasing the MEAP for *Ensemble Methods for Machine Learning*.

Modern machine learning has become synonymous with Deep Learning. However, Deep Learning is often too big a hammer for many applications, requires large data sets and computational resources that are out of reach for most of us: students, engineers, data scientists and analysts and casual enthusiasts.

Ensemble methods are another powerful way to build effective and robust models for real-world applications in many areas including finance, medicine, recommendation systems, cybersecurity and many more. This book is intended to be a tutorial on the practical aspects of implementing and training deployable ensemble models.

This book is intended for a broad audience: anyone creating ML algorithms - data scientists who are interested in using these algorithms for building models; engineers who are involved in building applications and architectures; or students, Kagglers, casual enthusiasts who want to learn more about this fascinating and exciting area of machine learning.

On this journey, we'll adopt an immersive approach to ensemble methods aimed at fostering intuition and demystifying the technical and algorithmic details. You will learn how to (1) implement a basic version from scratch to gain an under-the-hood understanding, and (2) use sophisticated, off-the-shelf implementations (such as `scikit-learn`) to ultimately get the best out of your models. Every chapter also comes with its own case study: a practical demonstration of how to use different ensemble methods on real-world tasks.

It is impossible to provide a detailed introduction to the diverse area of machine learning in one book. Instead, this book assumes basic knowledge of machine learning and that you have used or played around with at least one fundamental ML technique such as decision trees.

A basic working knowledge of Python is also assumed. Examples, visualizations and chapter case studies all use Python and Jupyter notebooks. Knowledge of other commonly used Python packages such as Numpy (for mathematical computations), Pandas (for data manipulation) and matplotlib (for visualization) is useful, but not necessary. In fact, you can learn how to use these packages through the examples and case studies.

Finally, this book is dedicated to you, and your feedback will be invaluable in improving it. Please post any questions, comments, corrections and suggestions in the [liveBook's Discussion Forum](#) for this book.

—Gautam Kunapuli

# *brief contents*

---

## **PART 1: THE BASICS OF ENSEMBLES**

*1 Ensemble Learning: Hype or Hallelujah?*

## **PART 2: ESSENTIAL ENSEMBLE METHODS**

*2 Homogeneous Parallel Ensembles: Bagging and Random Forests*

*3 Heterogeneous Parallel Ensembles: Combining Strong Learners*

*4 Sequential Ensembles: Boosting*

*5 Sequential Ensembles: Gradient Boosting*

*6 Sequential Ensembles: Newton Boosting*

## **PART 3: ENSEMBLES IN THE WILD: ADAPTING ENSEMBLE METHODS TO YOUR DATA**

*7 Learning with Continuous and Count Labels*

*8 Learning with Categorical Features*

*9 Explaining Your Ensembles*

*10 Further Reading*



## 1

# *Ensemble Learning: Hype or Hallelujah?*

## **This chapter covers**

- **Defining and framing the ensemble learning problem**
- **Motivating the need for ensembles in different applications**
- **Understanding how ensembles handle fit vs. complexity**
- **Implementing our first ensemble with ensemble diversity and model aggregation**

In October 2006, Netflix announced a \$1 million prize for the team that was able to improve movie recommendations over their own proprietary recommendation system, *Cinematch*, by 10%. The Netflix Grand Prize was one of the first ever open data science competitions and attracted tens of thousands of teams.

The training set consisted of 100 million ratings that 480 thousand users had given to 17 thousand movies. Within three weeks, 40 teams had already beaten *Cinematch*'s results. By September 2007, over 40 thousand teams had entered the contest and a team from AT&T Labs took the 2007 Progress Prize by improving upon *Cinematch* by 8.42%.

As the competition progressed and the 10% mark remained elusive, a curious phenomenon emerged among the competitors. Teams began to collaborate and share knowledge about effective feature engineering, algorithms and techniques. Inevitably, they began combining their models, blending individual approaches into powerful and sophisticated ensembles of many models. These ensembles combined the best of various diverse models and features and proved to be far more effective than any individual model.

In June 2009, nearly two years after the contest began, BellKor's Pragmatic Chaos, a merger of three different teams edged out another merged team, The Ensemble (which was a merger

of over 30 teams!), to improve upon the baseline by 10% and take the \$1 million prize. Just edged out is a bit of an understatement as BellKor's Pragmatic Chaos managed to submit their final models barely 20 minutes before The Ensemble got their models in<sup>1</sup>. In the end, both teams achieved a final performance improvement of 10.06%.

While the Netflix competition captured the imagination of data scientists, machine learners and casual data science enthusiasts worldwide, its lasting legacy has been to establish Ensemble Methods as a powerful way to build practical and robust models for large-scale, real-world applications. Among the individual algorithms used are several that have become staples of collaborative filtering and recommendation systems today: k-nearest neighbors, matrix factorization and restricted Boltzmann machines. However, Andreas Töscher and Michael Jahrer of BigChaos, co-winners of the Netflix Prize, summed up<sup>2</sup> their keys to success:

“During the nearly 3 years of the Netflix competition, there were two main factors which improved the overall accuracy: the quality of the individual algorithms and the ensemble idea.  
 ...the ensemble idea was part of the competition from the beginning and evolved over time. In the beginning, we used different models with different parametrization and a linear blending.  
 ...[Eventually] the linear blend was replaced by a nonlinear one...”

In the years since, the use of ensemble methods has exploded, and they have emerged as a state-of-the-art technology for machine learning.

The next two sections provide a gentle introduction to what ensemble methods are, why they work and where they are applied. Then, we will look at a subtle but important challenge prevalent in all machine-learning algorithms: the fit vs. complexity tradeoff.

Finally, we jump into training our very first ensemble method and see in a hands-on manner how ensemble methods overcome this fit vs. complexity tradeoff and improve overall performance. Along the way, we will familiarize ourselves with several key terms that form the lexicon of ensemble methods and will be used throughout the book.

## 1.1 Ensemble Methods: The Wisdom of the Crowds

What exactly is an ensemble method? Let's get an intuitive notion of what they are and how they work by considering the allegorical case of Dr. Randy Forrest. We can then go on to frame the ensemble learning problem.

Dr. Randy Forrest is a famed and successful diagnostician, much like his idol Dr. Gregory House of TV fame. His success, however, is due not only to his exceeding politeness (unlike his cynical and curmudgeonly idol), but also his rather unusual approach to diagnosis.

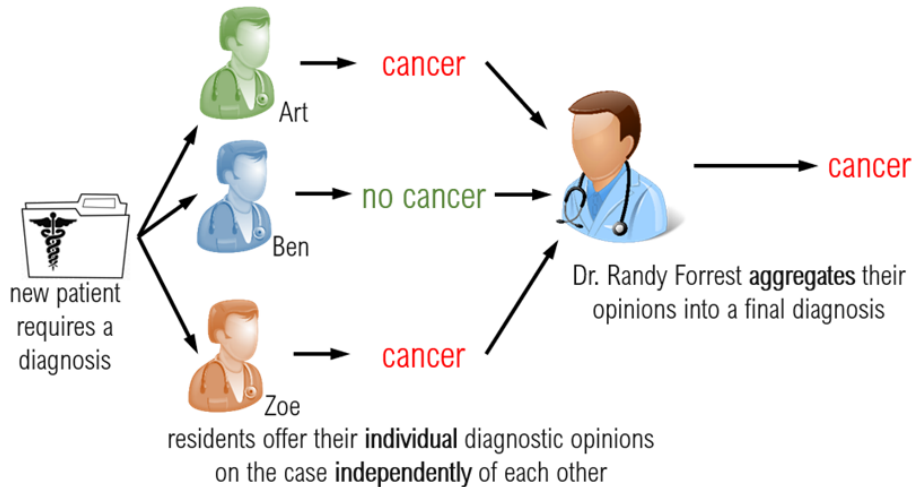
You see, Dr. Forrest works at a teaching hospital and commands the respect of a large number of doctors-in-training. Dr. Forrest has taken care to assemble a team with a *diversity*

<sup>1</sup> <https://www.netflixprize.com/leaderboard.html>

<sup>2</sup> *The BigChaos Solution to the Netflix Grand Prize*, Andreas Töscher, Michael Jahrer and Robert M. Bell.

of skills. His residents excel at different specializations: one is good at cardiology (heart), another at pulmonology (lungs), and yet another at neurology (nervous system) and so on. All in all, a rather diversely skillful bunch, each with their own strengths.

Every time Dr. Randy Forrest gets a new case he solicits the opinions of all his residents and collects possible diagnoses from all of them. He then democratically decides the final diagnosis as the *most common one* from among all those proposed.



**Figure 1.1** The diagnostic procedure followed by Dr. Randy Forrest every time he gets a new case is to get opinions from his residents. His residents offer their diagnoses: either that the patient has cancer or has no cancer. Dr. Forrest then selects the majority answer as the final diagnosis put forth by his team.

Dr. Forrest embodies a diagnostic *ensemble*: he aggregates his residents' diagnoses into a single diagnosis representative of the collective wisdom of his team. As it turns out, Dr. Forrest is right more often than any individual resident is.

Why? Because he knows that his residents are pretty smart, and a large number of pretty smart residents are *all* unlikely to make the same mistake. Here, Dr. Forrest relies on the power of *model aggregating* or *model averaging*: he knows that the average answer is most likely going to be a good one.

Still, how does Dr. Forrest know that *all* his residents are not wrong? He can't know that for sure, of course. However, he has guarded against this undesirable outcome all the same. Remember that his residents all have diverse specializations.

Because of their diverse backgrounds, training, specialization and skills, it is possible, but highly unlikely that all his residents are wrong. Here, Dr. Forrest relies on the power of *ensemble diversity*, or the diversity of the individual components of his ensemble.

Dr. Randy Forrest, of course, is an ensemble method, and his residents (who are in training) are the machine-learning algorithms that make up the ensemble. The secrets to his success, and indeed the success of ensemble methods as well, are:

- ensemble diversity, so that he has a variety of opinions to choose from, and
- model aggregation, so that he can combine them into a single final opinion.

Any collection of machine-learning algorithms can be used to build an ensemble: literally, a group of machine learners. But why do they work? James Surowiecki, in *The Wisdom of Crowds*, describes human ensembles or wise crowds thus:

“If you ask a large enough group of diverse and independent people to make a prediction or estimate a probability, the average of those answers will cancel out errors in individual estimation. Each person's guess, you might say, has two components: information and errors. Subtract the errors, and you're left with the information.”

This is also precisely the intuition behind ensembles of learners: it is possible to build a wise machine-learning ensemble by aggregating individual learners.

An **ensemble method** is a machine-learning algorithm that aims to **improve predictive performance on a task by aggregating the predictions of multiple estimators or models**. In this manner, an ensemble method learns a **meta-estimator**.

The key to success with ensemble methods is *ensemble diversity*. Informally, ensemble diversity refers to the fact that individual ensemble components, or machine-learning models, are different from each other.

Training such ensembles of diverse individual models is a key challenge in ensemble learning, and different approaches achieve this in different ways.

## 1.2 Why You Should Care About Ensemble Learning

What can you do with ensemble methods? Are they really just hype or are they hallelujah? As we see in this section, they can be used to train and deploy predictive models to build robust and effective models for many different applications.

One palpable success of ensemble methods is their domination of data science competitions (alongside deep learning), where they have been generally successful on different types of machine-learning tasks and application areas.

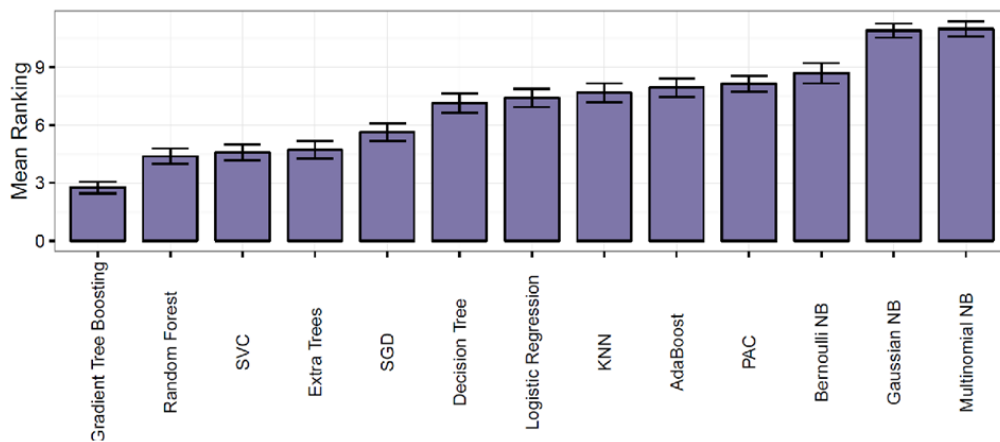
Anthony Goldbloom, CEO of Kaggle.com, revealed in 2015 that the three most successful algorithms for structured problems were XGBoost, Random Forest and Gradient Boosting, all ensemble methods. Indeed, the most popular way to tackle data science competitions these days is to combine *feature engineering* with *ensemble methods*. Structured data is generally

highly organized in tables, relational databases and other formats most of us are familiar with, and the type of data that ensemble methods have proven to be very successful on.

Unstructured data, in contrast, does not always have table structure. Images, Audio, video, waveform and text data are typically unstructured, which *deep learning approaches* -- including automated feature generation -- have demonstrated with great success. While we focus on structured data for most of this book, ensemble methods can be combined with deep learning for unstructured problems as well.

Beyond competitions, ensemble methods drive data science in several areas including financial and business analytics, medicine and healthcare, cybersecurity, education, manufacturing, recommendation systems, entertainment and many more.

In 2018, Olson et al<sup>3</sup> conducted a comprehensive analysis of 13 popular machine-learning algorithms and their variants. They ranked each algorithm's performance on 165 benchmark data sets (Figure 1.2). Their goal was to emulate the standard machine-learning pipeline to provide advice on *how to select a machine-learning algorithm*.



**Figure 1.2** Which machine learning algorithm should I use for my data set? The mean ranking of the performance of several different machine-learning algorithms on 165 different data sets is shown here. Figure reproduced from Olson et al (2018). SVC = support vector classification, SGD = stochastic gradient descent, KNN = k-nearest neighbor, PAC = passive-aggressive classifier, NB = naïve Bayes classifier.

On average, ensemble methods (1: Gradient Tree Boosting, 2: Random Forest, 4: Extra Trees) outperformed individual classifiers and classical ensemble approaches (9: AdaBoost).

<sup>3</sup> *Data-driven advice for applying machine learning to bioinformatics problems*, Randal S. Olson, William La Cava, Zairah Mustahsan, Akshay Varik, and Jason H. Moore, Pacific Symposium on Machine Learning (2018).

These results demonstrate exactly why ensemble methods (specifically, tree-based ensembles) are considered state-of-the-art.

If your goal is to develop state-of-the-art analytics from your data, or to eke out better performance and improve models you already have, this book is for you. If your goal is to start competing more effectively in data science competitions, for fame and fortune, or to just improve your data science skills, this book is also for you. If you're excited about adding powerful ensemble methods to your machine-learning arsenal, this book is definitely for you.

To drive home this point, we will build our first ensemble method: *a simple model combination ensemble*. Before we do, let's dive into the tradeoff between fit and complexity that most machine-learning methods have to grapple with, as it will help us understand why ensemble methods are so effective.

### 1.3 Fit vs. Complexity in Individual Models

In this section, we look at two popular machine-learning methods: decision trees and support vector machines. As we do so, we'll try to understand how their fitting and predictive behavior changes as they learn increasingly complex models. This section also serves as a refresher of the training and evaluation practices we usually follow during modeling.

Machine learning tasks are typically:

- *supervised learning tasks*, with a data set of *labeled examples*, where data has been annotated. For example, in cancer diagnosis, each example will be an individual patient, with label/annotation "has cancer" or "does not have cancer". Labels can be 0–1 (binary classification), categorical (multiclass classification) or continuous (regression).
- *unsupervised learning tasks*, with a data set of *unlabeled examples*, where the data lacks annotations. This includes tasks such as grouping examples together by some notion of "similarity" (clustering) or identifying anomalous data that does not fit the expected pattern (anomaly detection).

Let's say that we're looking at the Boston Housing data set, which describes the median value of owner-occupied homes in 506 U.S. census tracts in the Boston area. The machine-learning task is to learn a *regression* model to predict the median home value in a census tract using different variables. The Boston Housing data set is available from `scikit-learn`:

```
from sklearn.datasets import load_boston
from sklearn.preprocessing import StandardScaler
X, y = load_boston(return_X_y=True)

X = StandardScaler().fit_transform(X)
y = StandardScaler().fit_transform(y.reshape(-1, 1))
```

A data set is generally represented as a table, where each row is a data point or an *example*. Each example is characterized by features (also known as independent variables, or *attributes*) and a label (also known as a dependent variable, annotation or *response*).

In the Boston Housing data set, there are 13 attributes associated with each training example (that is, census tract), including crime rate (CRIM), nitric oxides concentration (NOX) and property-tax rate (TAX). The raw features (before standard scaling) are shown below.

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	price
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2

Figure 1.3 Median home price prediction is a regression task, where we want to learn a model to predict the label (median value of a home in a census tract, shaded column) given the attributes, or features of the census tract. Each row of this table is a training example, characterized by 13 features and a label (price).

### 1.3.1 Regression with Decision Trees

One of the most popular machine-learning models is the decision tree<sup>4</sup>, which can be used for classification as well as regression tasks. A decision tree is made of up of decision nodes and leaf nodes, and each decision node tests the current example for a specific condition (for example, is  $age > 50$ ?), and funnels it to the right path or the left path based on the answer (see Figure 1.4, right)

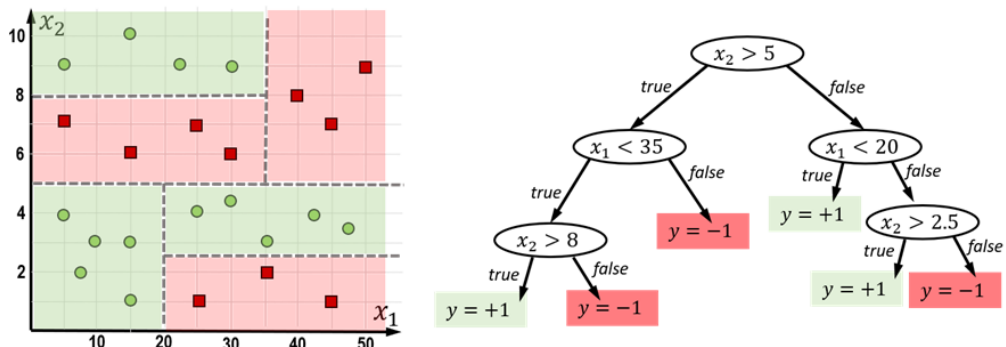


Figure 1.4 Decision trees partition the feature space into axis-parallel rectangles. When used for classification, the tree checks for conditions on the features in the decision nodes, funneling the example to the left or right after each test. Ultimately, the example filters down to a leaf node, which will give its classification label. The partition of the feature space according to this decision tree is shown on the left.

<sup>4</sup> For more details about learning with decision trees, see Chapters 3 (classification) and 9 (regression) of *Machine Learning in Action* by Peter Harrington (Manning, 2012).

Once an example reaches a leaf node, the prediction corresponding to the leaf node is returned. For classification tasks, the leaf value is a class label, while for regression tasks, the leaf returns a regression value.

A decision tree of depth 1 is called a *decision stump* and is the simplest possible tree. A decision stump contains a single decision node and two leaf nodes. A *shallow decision tree* (say, depth 2 or 3) will have a small number of decision nodes and leaf nodes and is a simple model. Consequently, it is only able to represent simple functions.

On the other hand, a deeper decision tree will have many more decision nodes and leaf nodes and is a more complex model. A deeper decision tree, thus, will be able to represent richer and more complex functions.

### ***Fit vs. Complexity in Decision Trees***

We'll randomly split the data set into a training set (with 67% of the data) and a test set (with 33%) of the data. We choose these split fractions in order to illustrate the effects of the complexity vs. fit more clearly.

**TIP** During modeling, we often have to split the data into a training and a test set. How big should these sets be? If the fraction of the data that makes up the training set is too small, the model will not have enough data. If the fraction of the data that makes up the test set is too small, there will be higher variation in our generalization estimates of how well the model performs on future data. A good rule of thumb (known as the Pareto principle) is to start with an 80%-20% train-test split.

For different depths  $d = 1$  to 10, we learn a tree on the training set and evaluate it on the test set. When we look at the training errors and the test errors for different depths, we can identify the depth of the "best tree". We characterize "best" in terms of an *evaluation metric*. For regression problems there are several evaluation metrics: mean square error, mean absolute deviation, coefficient of determination, etc.

We will use coefficient of determination, also known as the  $R^2$  score, which measures the proportion of the variance in the labels ( $y$ ) that is predictable from the features ( $x$ ).

### **Coefficient of Determination**

The coefficient of determination ( $R^2$ ) is a measure of regression performance.  $R^2$  is the proportion of variance in the true labels that is predictable from the features.

$R^2$  depends on two quantities: (1) the total variance in the true labels, or *total sum of squares* (TSS), and (2) the mean squared error, or the *residual sum of squares* (RSS) between the true and predicted labels. We have  $R^2 = 1 - \text{RSS} / \text{TSS}$ . A perfect model will have zero prediction error, or  $\text{RSS} = 0$  and its corresponding  $R^2 = 1$ . Really good models have  $R^2$  values close to 1. A really bad model will have high prediction error and high RSS. This means that for really bad models, we can have negative  $R^2$ .



One last thing to note is that we are splitting the data into a training set and test set *randomly*, which means that it is possible that we get very lucky or very unlucky in our split. To avoid the influence of randomness, we repeat our experiment 5 times and average the results across the runs. The pseudo-code for our experiment is shown below:

```

for run = 1:5,
  (Xtrn, ytrn), (Xtst, ytst) = split data (X), labels (y) into training and test subsets randomly
  for d = 1:10,
    tree[d] = train a decision tree of depth d on the training subset (Xtrn, ytrn)
    train_scores[run, d] = compute R2 score of tree[d] on the training set (Xtrn, ytrn)
    test_scores[run, d] = compute R2 score of tree[d] on the training set (Xtst, ytst)
  mean_train_score = average train_scores across runs
  mean_test_score = average test_scores across runs

```

The following code snippet does precisely this, and then plots the training and test scores. Rather than explicitly implement the pseudocode above, the listing below uses the `scikit-learn` function `sklearn.model_selection.ShuffleSplit` to automatically split the data into 20 different training and test subsets, and `sklearn.model_selection.validation_curve` to determine  $R^2$  scores for varying decision tree depths.

```

from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import validation_curve

subsets = ShuffleSplit(n_splits=5, test_size=0.33, random_state=23) #A

model = DecisionTreeRegressor()
trn_scores, tst_scores = validation_curve(model, X, y, \ #B
    param_name='max_depth', param_range=range(1, 11), \
    cv=subsets, scoring='r2')
mean_train_score = np.mean(trn_scores, axis=1)
mean_test_score = np.mean(tst_scores, axis=1)

#A set up 5 different random splits of the data into train and test sets
#B for each split, train decision trees of depths from 1 to 10 and then evaluate on the test set

```

Remember that our ultimate goal is to build a machine-learning model that *generalizes* well, that is, a model that performs well on *future, unseen data*. Our first instinct then, will be to train a model that achieves the smallest training error. Such models will typically be quite complex in order to fit as many training examples as possible.

After all, a complex model will likely fit our training data well and have a small training error. Presumably, a model that achieves the smallest training error should also generalize well in the future and predict unseen examples equally well.

Now, let's look at the results to see if this is the case. Remember that an  $R^2$  score close to 1 indicates a very good regression model, scores further away from 1 indicate worse models.

Deeper decision trees are more complex and have greater representational power. It is unsurprising, then, to see that deeper trees fit the training data better. This is clear from Figure 1.5: as tree depth (model complexity) increases, the training score approaches  $R^2=1$ . Thus, more complex models achieve better fits on the training data.

What is surprising, however, is that the *test score does not keep decreasing with complexity*. In fact, beyond `max_depth=4`, test scores remain fairly consistent. What this suggests is that a tree of depth 8 might *fit the training data better* than a tree of depth 4, but *both trees will perform roughly identically when they try to generalize and predict on new data!*



Figure 1.5 Comparing decision trees of different depths on the Boston Housing regression data set using  $R^2$  as the evaluation metric. Higher  $R^2$  scores mean that the model achieves lower error. An  $R^2$  score close to 1 means that the model achieves nearly zero error.

As decision trees become deeper, they become more complex and achieve lower training errors. However, their ability to generalize to future data (estimated by test scores) does not keep decreasing. This is a rather counter-intuitive result: *the model with the best fit on the training set is not necessarily the best model for predictions when deployed in the real world.*

It is tempting to argue that we got unlucky when we partitioned the training and test set randomly. However, we ran our experiment with 5 different random partitions and averaged the results to avoid precisely this. Let us repeat this experiment with another approach: support vector regression<sup>5</sup>.

<sup>5</sup> For more details on Support Vector Machines for classification, see Chapter 6 of *Machine Learning in Action* by Peter Harrington (Manning, 2012). For SVMs for regression, see *A Tutorial on Support Vector Regression* by Alex J. Smola and Bernhard Scholköpf, as well as the documentation pages of `sklearn.SVM.SVR()`.

### 1.3.2 Regression with support vector machines

Like decision trees, support vector machines (SVMs) are a great off-the-shelf baseline modeling approach, and most packages come with a robust implementation of SVMs.

You may have used SVMs for classification, where it is possible to learn nonlinear models of considerable complexity using kernels such as the RBF (or Gaussian) and polynomial kernels. SVMs have also been adapted for regression, and as in the classification case, they try to find a model that trades-off between regularization and fit during training. Explicitly, SVM training tries to find a model to minimize

$$\underbrace{\text{regularization}}_{\text{measures model flatness}} + C \cdot \underbrace{\text{loss.}}_{\text{measures model fit}}$$

The regularization term measures the flatness of the model: the more it is minimized, the more linear and less complex the learned model is.

The loss term measures the fit to the training data through a *loss function* (typically, mean squared error): the more it is minimized, the better the fit to the training data. The *regularization parameter*  $C$  trades-off between these two competing objectives:

- a small value of  $C$  means the model will focus more on regularization and simplicity and less on training error, which causes the model to have higher training error and *underfit*;
- a large value of  $C$  means the model will focus more on training error, learn more complex models, which causes the model to have lower training errors and possibly *overfit*.

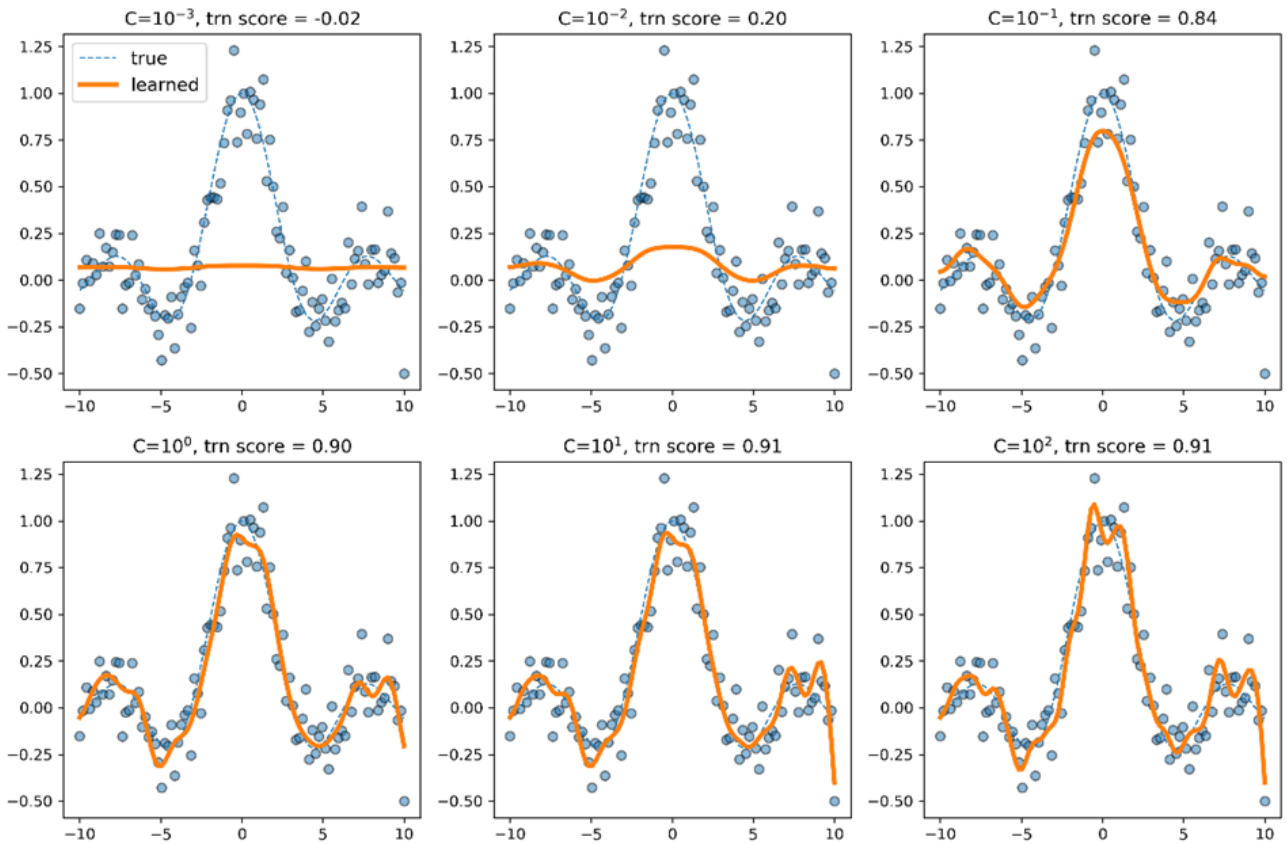


Figure 1.6 Support vector machine with a polynomial kernel of degree 3. Small values of  $C$  result in more linear (flatter) model, while large values of  $C$ , but more nonlinear and curvy models.

We can see the effect of increasing the value of  $C$  on the learned models in Figure 1.6. In particular, we can visualize the tradeoff between fit and complexity.

**CAUTION** SVMs identify “support vectors”, a smaller working set of training examples that the model depends on. Counting the number of support vectors is not an effective way to measure of model complexity as small values of  $C$  restrict the model more, forcing it to use more support vectors in the final model.

### *Fit vs. Complexity in Support Vector Machines*

Much like `max_depth` in `DecisionTreeRegressor()`, the parameter `C` in support vector regression, `SVR()` can be tuned to obtain models with different behaviors. Again, we are faced with the same question: which is the best model? To answer this, we can repeat the same experiment as with Decision Trees:

```

from sklearn.svm import SVR

model = SVR(degree=3)
trn_scores, tst_scores = validation_curve(model, X, y.ravel(),
    param_name='C',
    param_range=np.logspace(-2,4,7),
    cv=subsets, scoring='r2')
mean_train_score = np.mean(trn_scores, axis=1)
mean_test_score = np.mean(tst_scores, axis=1)

```

In this code snippet, we train an SVM with a degree 3 polynomial kernel. We try seven values of C:  $10^{-3}$ ,  $10^{-2}$ ,  $10^{-1}$ , 1, 10,  $10^2$  and  $10^3$  and visualize the train and test scores as before.

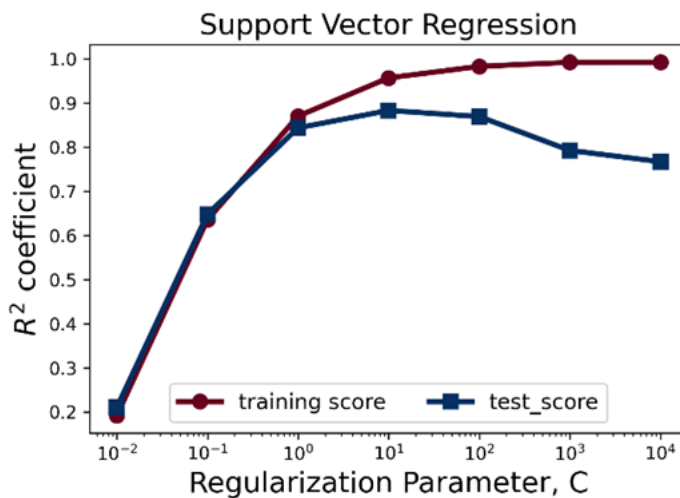


Figure 1.7 Comparing SVM regressors of different complexities on the Boston Housing data set using R<sup>2</sup> as the evaluation metric. As with decision trees, highly complex models (corresponding to higher C values) achieve fantastic fit on the training data, but don't actually generalize well.

Again, rather counter-intuitively, the model with the best fit on the training set is not necessarily the best model for predictions when deployed in the real world. Every machine-learning algorithm, in fact, exhibits this behavior:

- overly simple models tend to not fit the training data properly, and tend to generalize poorly on future data; a model that is performing poorly on training and test data is *underfitting*;
- overly complex models can achieve very low training errors but tend to generalize poorly on future data too; a model that is performing very well on training data, but poorly on test data is *overfitting*;
- the best models trade-off between complexity and fit, sacrificing a little bit of each during training so that they can *generalize most effectively when deployed*.

As we will see in the next section, ensemble methods are an effective way of tackling the issue of fit vs. complexity.

### THE BIAS-VARIANCE TRADEOFF

What we have informally seen above as the fit vs. complexity tradeoff is more formally known as the bias-variance tradeoff.

The bias (error) of a model is the error arising from the impact of modeling assumptions (such as a preference for simpler models). The variance of a model is the error arising from sensitivity to small variations in the data set.

Highly complex models (low bias) will overfit the data and be more sensitive to noise (high variance), while simpler models (high bias) will underfit the data and be less sensitive to noise (low variance). This trade-off is inherent in every machine-learning algorithm. Ensemble methods seek to overcome this issue by combining several low-bias models to reduce their variance or combining several low-variance models to reduce their bias.

## 1.4 Our First Ensemble

In this section, we see that we can overcome the fit vs. complexity issues of individual models by training an ensemble, our first. Recall from the allegorical Dr. Forrest that an effective ensemble performs *model aggregation* on a set of diverse of component models. Here:

1. We train a set of diverse *base estimators* (also known as *base learners*) using diverse base learning algorithms on the same data set. That is, we count on the significant variations in how each learning algorithm to produce a diverse set of base estimators.
2. For a regression problem (such as the Boston Housing data), the predictions of individual base estimators are continuous. We can aggregate the results into one final ensemble prediction by *simple averaging* of the individual predictions.

We use the following regression algorithms to produce base estimators from our data set: kernel ridge regression, support vector regression, decision tree regression, k-nearest neighbor regression, Gaussian processes and multi-layer perceptrons (neural networks).

Once we have the trained models, we use each one to make individual predictions and then aggregate the individual predictions into a final prediction. This is shown in the figure below.

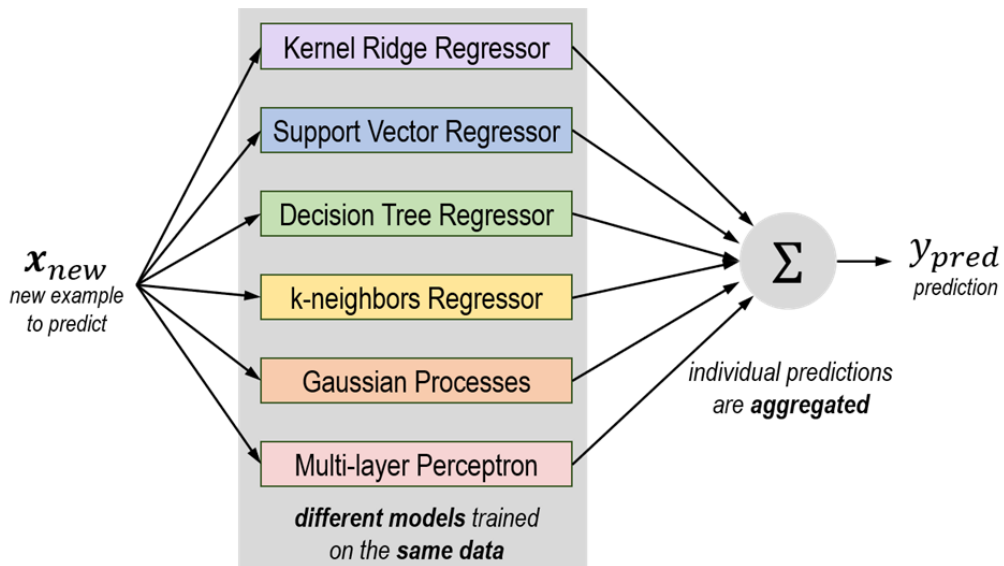


Figure 1.8 Predictions using model averaging, illustrated.

First, we split the overall data set into a training set (67%) and a test set (33%).

```
Xtrn, Xtst, ytrn, ytst = train_test_split(X, y, test_size=0.33)
```

The code for training individual base estimators is shown below.

#### Listing 1.1 Training diverse base estimators

```
from sklearn.kernel_ridge import KernelRidge
from sklearn.svm import SVR
from sklearn.tree import DecisionTreeRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.neural_network import MLPRegressor

estimators = {'krr': KernelRidge(kernel='rbf', gamma=0.1), #A
             'svr': SVR(gamma=0.1),
             'dtr': DecisionTreeRegressor(max_depth=8),
             'knn': KNeighborsRegressor(n_neighbors=3),
             'gpr': GaussianProcessRegressor(alpha=1e-1),
             'mlp': MLPRegressor(alpha=25, max_iter=1000)}

for name, estimator in estimators.items():
    estimator = estimator.fit(Xtrn, ytrn) #B

#A initialize hyperparameters of each individual base estimator
#B train the individual base estimators
```

We have now trained six diverse base estimators using six different base learning algorithms. Given new data, we can aggregate the individual predictions into a final prediction

### Listing 1.2 Aggregating base estimator predictions

```
n_estimators, n_samples = len(estimators), Xtst.shape[0]
y_individual = np.zeros((n_samples, n_estimators))
for i, (model, estimator) in enumerate(estimators.items()): #A
    y_individual[:, i] = estimator.predict(Xtst) #B
y_final = np.mean(y_individual, axis=1) #C

#A initialize individual predictions
#B individual predictions using the base estimators
#C aggregate (average) individual predictions
```

One way to understand the benefits of ensembling is if we look at all possible combinations of models for predictions. That is, we look at the performance of one model at a time, then all possible ensembles of two models (there are 15 such combinations), then all possible ensembles of three models (there are 20 such combinations) and so on. For ensemble size 1 to 6, we plot the test set performances of all these ensemble combinations.

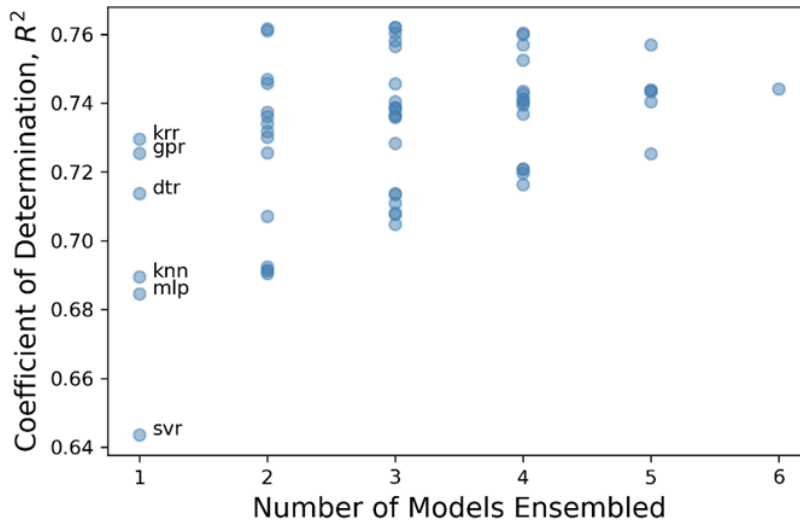


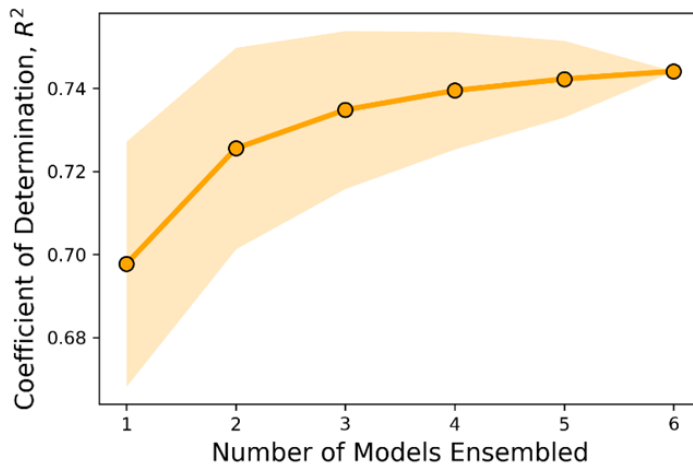
Figure 1.9 Prediction performance vs. ensemble size.

As we aggregate more and more models, we see that the ensembles generalize increasingly better. The most striking result of our experiment, though, is that the performance of the ensemble of all 6 estimators is often better than the performances of each individual estimator.



Finally, what of fit vs. complexity? It is difficult to characterize the complexity of the ensemble, as different types of estimators in our ensemble have different complexities. However, we can characterize the *variance of the ensemble*.

Recall that variance of an estimator reflects its sensitivity to the data. A high variance estimator is highly sensitive and less robust, often because it is overfitting. In Figure 1.10, we show the variance of the ensembles from Figure 1.9, which is the width of the band.



**Figure 1.10** The mean performance of the ensemble combinations increases, showing that bigger ensembles perform better. The standard deviation (square root of the variance) of performance of the ensemble combinations decreases, showing that the overall variance decreases!

As ensemble size increases, the variance of the ensemble decreases! This is a consequence of model aggregation or averaging. We know that averaging “smooths out the rough edges”. In the case of our ensemble, averaging individual predictions smooths out mistakes made by individual base estimators, replacing them instead with the wisdom of the ensemble: from many, one. The overall ensemble is more robust to mistakes, and unsurprisingly, generalizes better than any single base estimator.

Each component estimator in the ensemble is an individual, like one of Dr. Forrest’s residents. It makes predictions based on its own experiences (introduced during learning). At prediction time, when we have six individuals, we will have six predictions, or six opinions. For “easy examples”, the individuals will mostly agree. For “difficult examples”, the individuals will differ amongst each other, *but on average, are more likely to be closer to the correct answer*<sup>6</sup>.

<sup>6</sup> There are cases when this breaks down. In the UK version of *Who Wants To Be A Millionaire?* a contestant successfully made it as far as £125,000 (or about \$160,000), when he was asked which novel begins with the words: ‘3 May. Bistritz. Left Munich at 8:35PM.’ After using the 50/50 lifeline, he was left with only two choices: *Tinker Tailor Soldier Spy* and *Dracula*. Knowing he could lose £93,000 if he got it wrong, he asked the studio audience. 81% of

In this simple scenario, we trained six “diverse” models by using six different learning algorithms. Ensemble diversity is critical to the success of the ensemble as it ensures that the individual estimators are different from each other and don’t all make the same mistakes.

As we will see in this book, different ensemble methods take different approaches to train diverse ensembles. This is a key challenge for ensemble methods have to somehow ensure diversity from a single data set.

## 1.5 Summary

This chapter introduced ensemble methods and motivations for using them.

- Ensemble learning aims to improve predictive performance by training multiple models to train a meta-estimator. The component models of an ensemble are called base estimators or base learners.
- Ensemble methods leverage the power of “the wisdom of crowds”, which relies on the principle that the collective opinion of a group is more effective than any single individual in the group.
- Ensemble methods are widely used in several application areas including financial and business analytics, medicine and healthcare, cybersecurity, education, manufacturing, recommendation systems, entertainment and many more.
- Most machine-learning algorithms contend with a fit vs. complexity (also called bias-variance) tradeoff, which affects their ability to generalize well to future data. Ensemble methods use multiple models to overcome this tradeoff.
- An effective ensemble requires two key ingredients: (1) ensemble diversity and (2) model aggregation for the final predictions.

In the next few chapters, we explore several popular ensemble methods.

the audience voted for *Tinker Tailor Soldier Spy*. The audience was overwhelmingly confident... and unfortunately for the contestant, overwhelmingly wrong. As we will see in the book, we look to avoid this situation by making certain assumptions about the “audience”, in our case, the base estimators.

# 2

## *Homogeneous Parallel Ensembles: Bagging and Random Forests*

### **This chapter covers**

- Training homogeneous parallel ensembles
- Implementing and understanding how Bagging works
- Implementing and understanding how Random Forest works
- Training variants with pasting, random subspaces, random patches and ExtraTrees
- Using bagging and random forests in practice

In Chapter 1, we introduced ensemble learning and created our first rudimentary ensemble. To recap, an ensemble method relies on the notion of “wisdom of the crowd”: the *combined* answer of many *diverse* models is often better than any one individual answer.

We begin our journey into ensemble learning methods in earnest with parallel ensemble methods. We begin with this type of ensemble methods because, conceptually, parallel ensemble methods are easy to understand and implement.

Parallel ensemble methods, as the name suggests, train each component base estimator independently of the others, which means that they can be trained in parallel. As we will see, parallel ensemble methods can be further distinguished as homogeneous and heterogeneous parallel ensembles depending on the kind of learning algorithms they use.

In this chapter, we will learn about homogeneous parallel ensembles, whose component models are all trained using the same machine-learning algorithm. This is in contrast to

heterogeneous parallel ensembles (covered in the next chapter), whose component models are trained using different machine-learning algorithms.

The class of homogeneous parallel ensemble methods includes two popular machine-learning methods, one or both of which you might have come across and even used before: *bagging* and *random forest*.

Recall that the two key components of an ensemble method are: ensemble diversity and model aggregation. Since homogeneous ensemble methods use the same learning algorithm on the same data set, how can they generate a set of diverse base estimators? They do this through *random sampling* of either the training examples (as bagging does) or features (as some variants of bagging do) or both (as random forest does).

Some of the algorithms introduced in this chapter, such as random forest are widely used in medical and bioinformatics applications. In fact, random forest is still a strong *off-the-shelf baseline algorithm* to try on a new data set, owing to its efficiency (it can be parallelized or distributed easily over multiple processors).

We will begin with the most basic parallel homogeneous ensemble: bagging. Once we understand how bagging achieves ensemble diversity through sampling, we look at the most powerful variant of bagging: random forest.

We will also learn about other variants of bagging (pasting, random subspaces, random patches) and random forests (ExtraTrees). These variants are often effective for big data or in applications with high-dimensional data.

## 2.1 Parallel Ensembles

First, we concretely define the notion of a parallel ensemble. This will help us put the algorithms in this chapter and the next into a single context, so that we can easily see the similarities between them, and their differences.

Recall Dr. Randy Forrest, our ensemble diagnostician from Chapter 1. Every time Dr. Forrest gets a new case he solicits the opinions of all his residents. He then decides the final diagnosis from among those proposed by his residents (Figure 2.1, top). Dr. Forrest's diagnostic technique is successful because of two reasons.

1. He has assembled a *diverse* set of residents, with different medical specializations, which means they *think* differently about each case. This works out well for Dr. Forrest as it puts several different perspectives on the table for him to consider.
2. He *aggregates* the *independent* opinions of his residents into one final diagnosis. Here, he is democratic and selects the majority opinion. However, he can also aggregate his residents' opinions in other ways. For instance, he can weight the opinions of his more experienced residents higher. This reflects that he trusts some residents more than others, based on factors such as experience or skill, that mean they are right more often than other residents on the team.

Dr. Forrest and his residents are a *parallel ensemble* (Figure 2.1. bottom). Each resident in our example above is a component *base estimator* (or base learner) that we have to train. Base

estimators can be trained using different base algorithms (leading to heterogeneous ensembles) or the same base algorithm (leading to homogeneous ensembles).

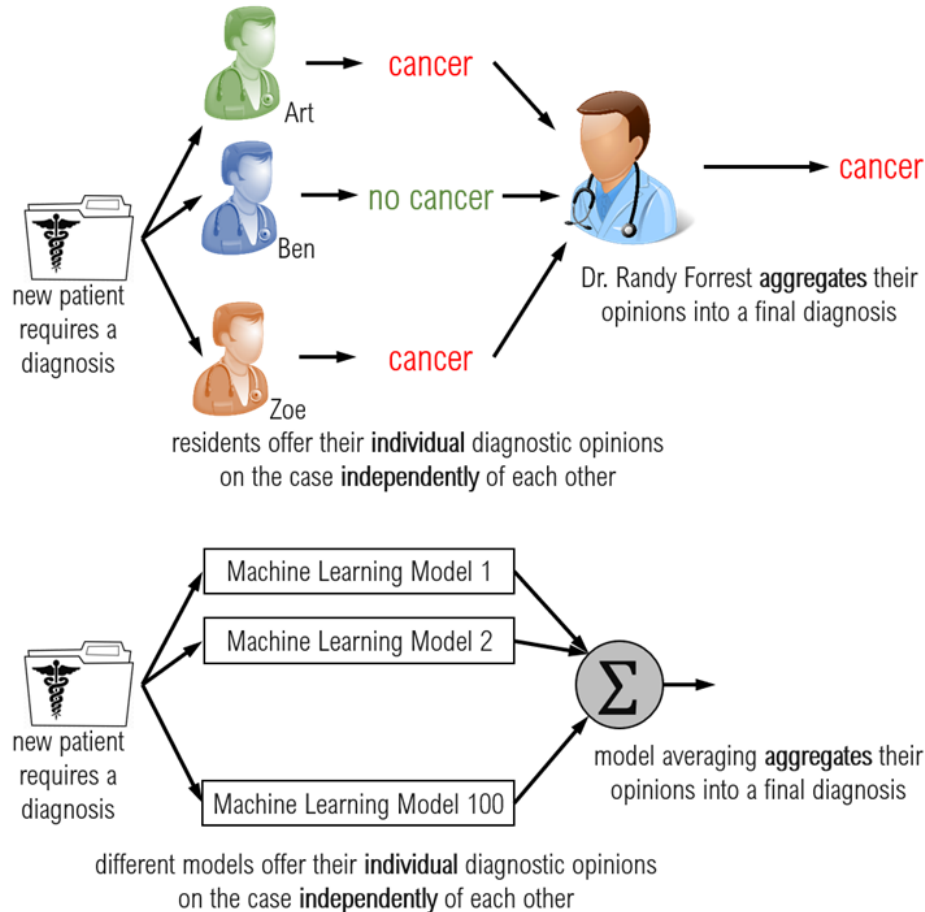


Figure 2.1 Dr. Randy Forrest's diagnostic process is an analogy of a parallel ensemble method.

In this chapter, we explore two popular parallel, homogeneous ensemble methods: bagging and random forests, and their variants. If we want to put together an effective ensemble similar to Dr. Forrest's we have to address two problems.

1. How do we create a set of base estimators with "diverse opinions" from a single data set? That is, how can we ensure *ensemble diversity* during training?
2. How can we aggregate decisions, or predictions, of each individual base estimator into a final prediction? That is, how can we perform *model aggregation* during prediction?

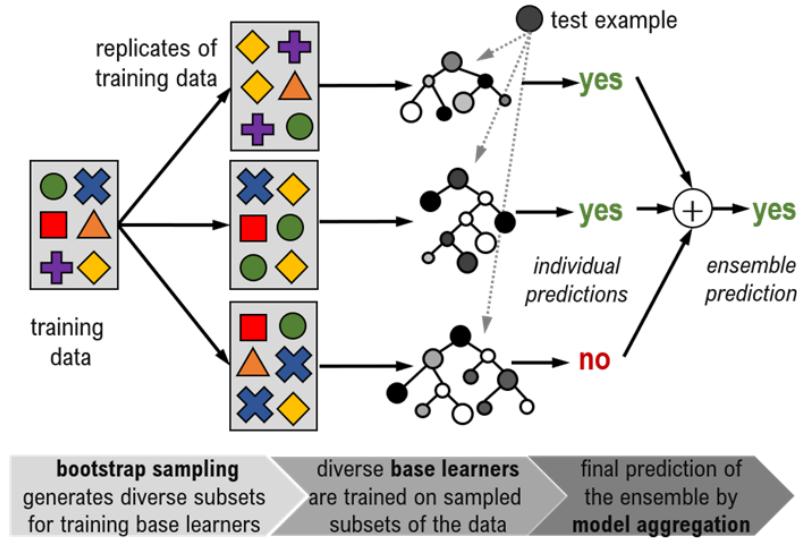
We will see exactly how to do both in the next section.

## 2.2 Bagging: Bootstrap Aggregating

Bagging, short for *bootstrap aggregating*, was introduced by Leo Breiman in 1996. The name refers to how bagging achieves ensemble diversity (through bootstrap sampling) and performs ensemble prediction (through model aggregating).

Bagging is the most basic homogeneous parallel ensemble method we can construct. Understanding bagging will be helpful in understanding the other ensemble methods in this chapter. These methods further enhance the basic bagging approach in different ways: to either improve ensemble diversity or overall computational efficiency.

Bagging uses the same base machine-learning algorithm to train base estimators. So how can we get multiple base estimators from a single data set and a single learning algorithm, let alone diversity? By training base estimators on *replicates of the data set*.



**Figure 2.2 Bagging, illustrated.** Bagging uses bootstrap sampling to generate similar but not exactly identical subsets (observe the replicates above) from a single data set. Models are trained on each of these subsets resulting in similar but not exactly identical base estimators. During prediction, the individual base estimator predictions are aggregated into a final ensemble prediction. Also observe that training examples may repeat in the replicated subsets; this is a consequence of bootstrap sampling.

Bagging consists of two steps as illustrated in Figure 2.3:

1. during training, bootstrap sampling, or *sampling with replacement* is used to generate replicates of the training data set that are different from each other but drawn from the original data set; this ensures that base learners trained on each of the replicates are also different from each other;

- during prediction, *model aggregation* is used to combine the predictions of the individual base learners into one ensemble prediction. For classification tasks, the final ensemble prediction is determined by *majority voting*, for regression tasks, by *averaging*.

### 2.2.1 Intuition: Resampling and Model Aggregation

The key challenge for ensemble diversity is that we need to create (and use) different base estimators using the same learning algorithm and the same data set. We'll now see how to (1) generate "replicates" of the data set, which in turn, can be used to train base estimators, and (2) combine predictions of base estimators.

#### *Bootstrap Sampling: Sampling with Replacement*

We'll use random sampling to easily generate *subsets of smaller size* from the original data set. In order to generate *replicates of the data set of the same size*, we will need to perform *sampling with replacement*, otherwise known as bootstrap sampling.

When sampling with replacement, some objects that were already sampled have a chance to be sampled a second time (or even a third, or fourth, and so on) because they were replaced. In fact, some objects may be sampled many times, while some objects may *never be sampled*.

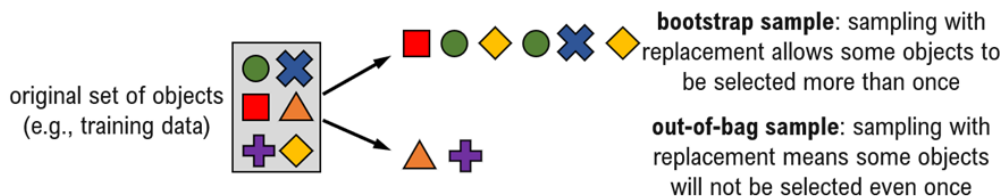


Figure 2.3 Bootstrap sampling illustrated on a data set of 6 examples. By sampling with replacement, we can get a bootstrap sample of size 6, but containing only 4 unique objects, but with repeats. Performing bootstrap sampling several times produces several replicates of the original data set all of them with repeats.

Thus, bootstrap sampling naturally partitions a data set into two sets: a bootstrap sample (with training examples that were sampled at least once) and an *out-of-bag (oob) sample* (with training examples that were never sampled even once).

We can use each bootstrap sample for training a different base estimator. Since different bootstrap samples will contain different examples repeating different number of times, each base estimator will turn out to be somewhat different from the others.

#### *The Out-of-Bag Sample*

Should we just throw away the out-of-bag sample? That's rather wasteful. Observe though, if we train a base estimator on the bootstrap sample, the oob sample is *held out*. Sound familiar?

The *oob* sample is effectively a held-out set and can be used to evaluate the ensemble without the need for a separate validation set or even a cross-validation procedure. This is great, because it allows us to utilize data more efficiently during training. The error estimate computed using out-of-bag instances is called the *out-of-bag error* or the *oob score*.

It is very easy to generate bootstrap samples with replacement using `numpy`. Suppose we have a data set with 50 training examples (say, patient records with unique ids from 0 to 49). We can generate a bootstrap sample, also of size 50 (same size as the original data set) for training (`replace=True` to sample with replacement).

```
>>> import numpy as np
>>> bag = np.random.choice(range(0, 50), size=50, replace=True)
>>> np.sort(bag)
array([ 1,  3,  4,  6,  7,  8,  9, 11, 12, 12, 14, 14, 15, 15, 21, 21, 21,
        24, 24, 25, 25, 26, 26, 29, 29, 31, 32, 32, 33, 33, 34, 34, 35, 35,
        37, 37, 39, 39, 40, 43, 43, 44, 46, 46, 48, 48, 48, 49, 49, 49])
```

Can you spot the repeats in this bootstrap sample? This bootstrap sample now serves as one replicate of the original data set and can be used for training. The corresponding oob sample is all the examples *not* in the bootstrap sample.

```
>>> oob = np.setdiff1d(range(0, 50), bag)
>>> oob
array([ 0,  2,  5, 10, 13, 16, 17, 18, 19, 20, 22, 23, 27, 28, 30, 36, 38,
        41, 42, 45, 47])
```

It is easy to verify that there is no overlap between the bootstrap subset and the oob subset. This means that the oob sample can be used as a “test set”.

To summarize: after one round of bootstrap sampling, we get one bootstrap sample (for training a base estimator) and a corresponding oob sample (to evaluate that base estimator).

When we repeat this step many times, we will have trained several base estimators and will also have estimated their individual generalization performances through individual oob errors. The averaged oob error is a good estimate of the performance of the overall ensemble.

### 0.632 Bootstrap

When sampling with replacement, the bootstrap sample will contain roughly 63.2% of the data set, while the oob sample will contain the other 36.8% of the data set.

We can show this by computing the probabilities of a data point being sampled. If our data set has  $n$  training examples, the probability of picking one particular data point  $x$  in the bootstrap sample is  $1/n$ . The probability of not picking  $x$  in the bootstrap sample (that is, picking  $x$  in the oob sample) is  $1-(1/n)$ .

For  $n$  data points, the overall probability of being selected in the oob sample is  $(1 - (1/n))^n \approx e^{-1} = 0.368$  (for sufficiently large  $n$ ).

Thus, each oob sample will contain (approximately) 36.8% of the training examples, and the corresponding bootstrap sample will contain (approximately) the remaining 63.2% of the instances.



## Model Aggregation

Bootstrap sampling generates diverse replicates of the data set, which allows us to train diverse models independently of each other. Once trained, we can use this ensemble for prediction. The key is to *combine* their (sometimes differing) opinions into a single final answer.

We've seen two examples of model aggregation: *majority voting* and *model averaging*. For classification tasks, majority voting is used to aggregate predictions of individual base learners. The majority vote is also known as the *statistical mode*. The mode is simply the most frequently occurring element and is a statistic similar to the mean or the median.

We can think of model aggregation as averaging: it smooths out imperfections among the chorus and produces a single answer reflective of the majority. If we have a set of robust base estimators, model aggregation will smooth out mistakes made by individual estimators.

Ensemble methods use a variety of aggregation techniques depending on the task including majority vote, mean, weighted mean, combination functions and even another machine-learning model! In this chapter, we will stick to majority voting as our aggregator. We will explore some other aggregation techniques for classification in Chapter 3.

### 2.2.2 Implementing Bagging

We can implement our own version of bagging easily. This illustrates the simplicity of bagging and provides a general template for how other ensemble methods in this chapter work. Each base estimator in our bagging ensemble is trained *independently* using the following steps:

1. Generate a bootstrap sample from the original data set
2. Fit a base estimator to the bootstrap sample

Independently here means that the training stage of each individual base estimator takes place without consideration of what is going on with the other base estimators.

We use decision trees as base estimators; the maximum depth can be set using the parameter `max_depth`. We will need two other parameters: `n_estimators`, the ensemble size and `max_samples`, the size of the bootstrap subset, that is, the number of training examples to sample (with replacement) per estimator.

Our naïve implementation trains each base decision tree sequentially. If it takes 10 seconds to train a single decision tree, and we are training an ensemble of 100 trees, it will take our implementation  $10 \text{ sec.} \times 100 = 1000$  seconds of total training time.

#### Listing 2.1 Bagging with Decision Trees: Training

```
from sklearn.tree import DecisionTreeClassifier

def bagging_fit(X, y, n_estimators, max_depth=5, max_samples=200):
    n_examples = len(y)
    estimators = [DecisionTreeClassifier(max_depth=max_depth)
                  for _ in range(n_estimators)] #A
```

```

for tree in estimators:
    bag = np.random.choice(n_examples, max_samples, replace=True) #B
    tree.fit(X[bag, :], y[bag]) #C

return estimators

```

**#A** Create a list of untrained base estimators  
**#B** Generate a bootstrap sample  
**#C** Fit a tree to the bootstrap sample

This function will return a list of `DecisionTreeClassifier` objects. We can use this ensemble for prediction, which is implemented in the listing below.

### Listing 2.2 Bagging with Decision Trees: Prediction

```

from scipy.stats import mode

def bagging_predict(X, estimators):
    all_predictions = np.array([tree.predict(X) for tree in estimators]) #A
    ypred, _ = mode(all_predictions, axis=0) #B
    return np.squeeze(ypred)

```

**#A** Predict each test example using each estimator in the ensemble  
**#B** Make final predictions by majority voting

We can test our implementation on 2d data and visualize the results. Our bagging ensemble has 500 decision trees, each of depth 12 and trained on bootstrap samples of size 200.

```

>>> from sklearn.datasets import make_moons
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.metrics import accuracy_score

>>> X, y = make_moons(n_samples=300, noise=.25, random_state=0) #A
>>> Xtrn, Xtst, ytrn, ytst = train_test_split(X, y, test_size=0.33)
>>> bag_ens = bagging_fit(Xtrn, ytrn, n_estimators=500,
...                       max_depth=12, max_samples=200)
>>> ypred = bagging_predict(Xtst, bag_ens) #B

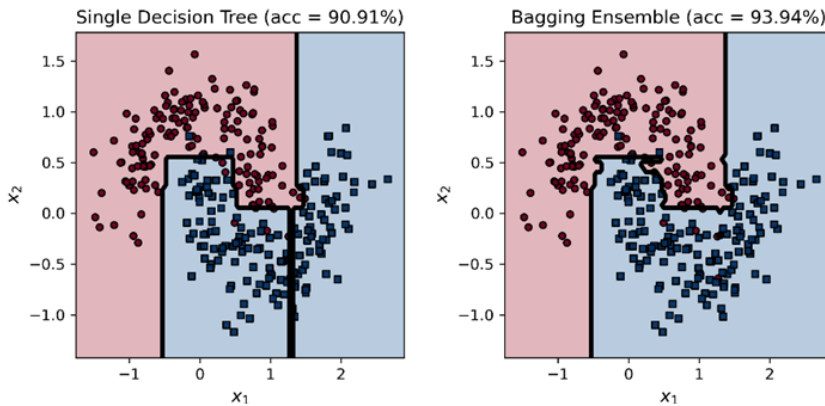
>>> accuracy_score(ytst, ypred)
0.9191919191919192

```

**#A** Create a 2d data set  
**#B** Make final predictions by majority voting

The size of the original data set is 300, though we train base estimators on smaller subsets of 200. Such subsampling can be useful for large data sets, where it might not be a good idea to train 500 decision trees, each using the full data set.

We can now see what a bagged ensemble looks like, compared to a single tree.



**Figure 2.4** A single decision tree (left) overfits the training set and can be sensitive to outliers (observe the thin spiky parts of the decision boundary). A bagging ensemble (right) smooths out the overfitting effects and misclassifications of several such base estimators and often returns a robust answer.

Bagging can learn fairly complex and nonlinear decision boundaries. Even if individual decision trees (and generally, base estimators) are sensitive to outliers, the ensemble of base learners will smooth out individual variations and will be more robust.

This smoothing behavior of bagging is due to model aggregation. When we have many highly nonlinear classifiers, each trained on a slightly different replicate of the training data, each may overfit, but they don't all overfit the same way.

Thus, when we aggregate their predictions, it smooths out the errors and the ensemble performance is improved! Much like an orchestra, the final result is a smooth symphony that can easily overcome the mistakes of any individual musician in it.

### 2.2.3 Bagging with scikit-learn

Now that we are armed with under-the-hood understanding of bagging works, we look at how to use `scikit-learn`'s `BaggingClassifier` package. `scikit-learn`'s implementation provides additional functionality including support for parallelization, ability to use other base learning algorithms beyond decision trees and most importantly, out-of-bag evaluation.

#### Listing 2.3 Bagging with `scikit-learn`

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier

base_estimator = DecisionTreeClassifier(max_depth=10) #A
bag_ens = BaggingClassifier(base_estimator=base_estimator, n_estimators=500,
                           max_samples=100, oob_score=True) #B

bag_ens.fit(Xtrn, ytrn)
ypred = bag_ens.predict(Xtst)
```

#A Set the base learning algorithm along with hyper-parameters

**#B Use out-of-bag sample to estimate generalization error**

`BaggingClassifier` supports out-of-bag evaluation and will return the oob accuracy if we set `oob_score=True`. Recall that for each bootstrap sample, we also have a corresponding out-of-bag sample that contains all the data points that were not selected during sampling.

Thus, each oob sample is a surrogate for “future data” as it is not used to train the corresponding base estimator. After training, we can query the learned model to obtain the *out-of-bag score or oob score*:

```
>>> bag_ens.oob_score_
0.9658792650918635
```

The oob score is an estimate of the bagging ensemble’s predictive (generalization) performance, here 96.6%. In addition to the oob samples, we have ourselves also held out a test set. We compute another estimate of this model’s generalization on the test set:

```
>>> accuracy_score(ytst, ypred)
0.9521276595744681
```

The test accuracy is 95.2%, which is pretty close to the oob score. We used decision trees of maximum depth 12 as base estimators. Deeper decision trees are more complex, which allows them to fit (and even overfit) the training data.

**TIP** Bagging is most effective with complex and nonlinear classifiers that tend to overfit the data. Such complex, overfitting models are *unstable*, that is highly sensitive to small variations in the training data.

To see why, consider that individual decision trees in a bagged ensemble have roughly the same complexity. However, due to bootstrap sampling, they have been trained on different replicates of the data set, and overfit differently. Put another way, they all overfit by roughly the same amount, but in different places.

Bagging works best with such models because its model aggregation smooths out the overfitting mistakes made by the individual unstable base estimators, leading to a more robust and stable ensemble.

We can visualize the smoothing behavior of `BaggingClassifier` by comparing its decision boundary to its component base `DecisionTreeClassifiers`.

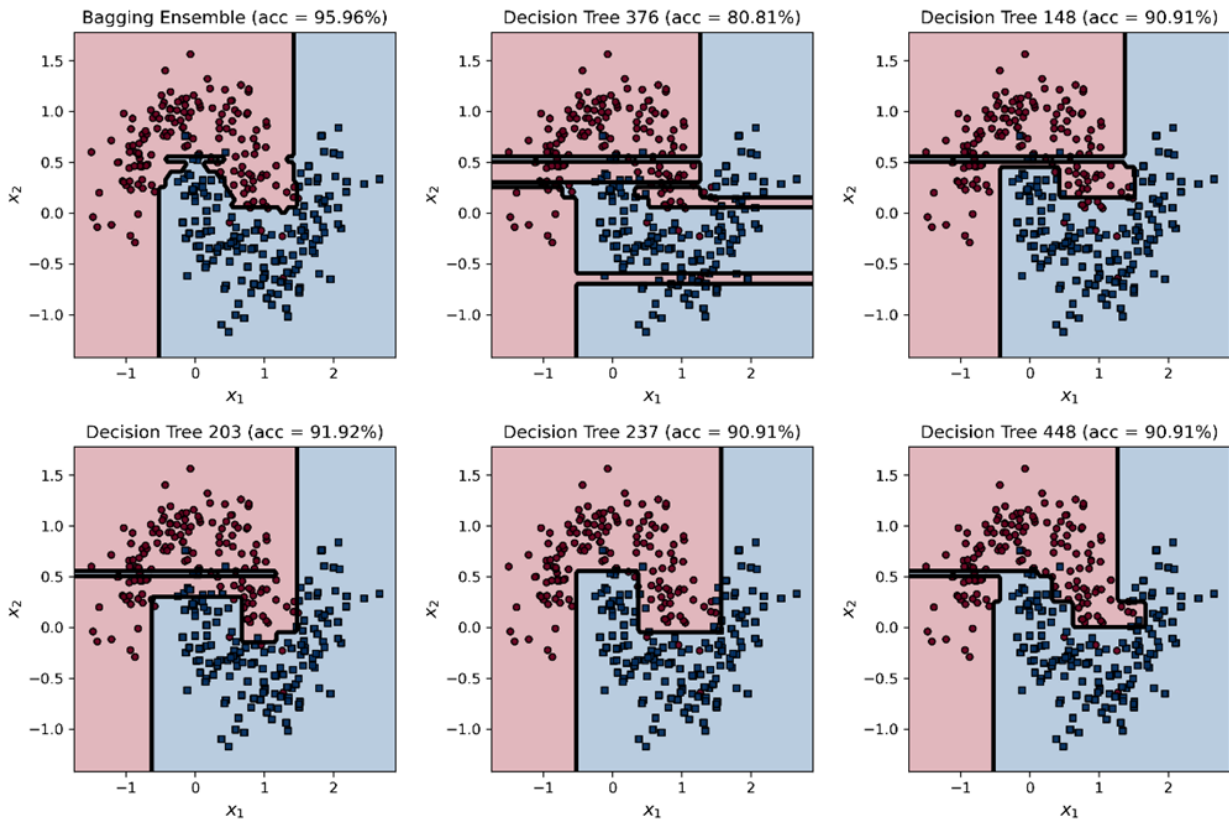


Figure 2.5 Bootstrap sampling leads to different base estimators overfitting differently, while model aggregation averages out individual mistakes and produces smoother decision boundaries.

## 2.2.4 Faster Training with Parallelization

Bagging is a parallel ensemble algorithm as it trains each base learner independently of other base learners. This means that training bagging ensembles can be parallelized if you have access to computing resources such as multiple cores or clusters.

`BaggingClassifier` supports the speed up of both training and prediction through the `n_jobs` parameter. By default, this parameter is set to 1 and bagging will run sequentially. Alternately, you can specify the number of concurrent processes `BaggingClassifier` should use with by setting `n_jobs`:

```
bag_ens = BaggingClassifier(
    base_estimator=DecisionTreeClassifier(), n_estimators=100,
    max_samples=100, oob_score=True,
    n_jobs=-1) #A
```

#A If set to `-1`, `BaggingClassifier` uses all CPUs

The figure below compares the training efficiency of sequential (with `n_jobs=1`) with parallelized bagging (`n_jobs=-1`) on a machine with 6 cores. This shows that bagging can be effectively parallelized and training times significantly improved.

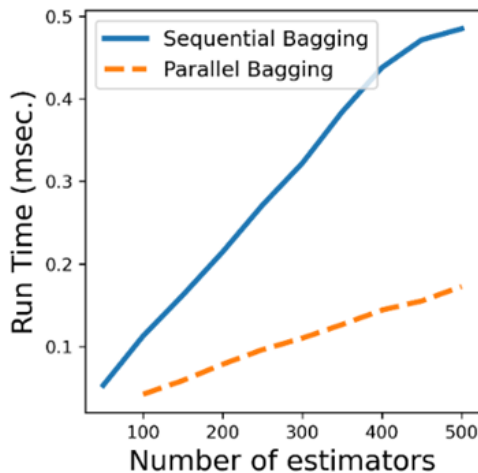


Figure 2.6 Bagging can be parallelized to increase training efficiency.

## 2.3 Random Forests

We have seen how bagging uses random sampling with replacement, or bootstrap sampling, for ensemble diversity. Now, we look at random forests, a special extension of bagging that introduces additional randomization to further promote ensemble diversity.

Until the emergence of gradient boosting (Chaps. 5 and 6), random forests were state-of-the-art and were widely utilized. They are still a popular go-to method for many applications, especially bioinformatics. Random forests can be an excellent off-the-shelf baseline for your data, as they are computationally efficient to train. They can also rank data features by importance, which makes them particularly suited for high-dimensional data analysis.

### 2.3.1 Randomized Decision Trees

“Random forest” specifically refers to an ensemble of *randomized decision trees* constructed using bagging. Random forests perform bootstrap sampling to generate a training subset (exactly like bagging), and then use randomized decision trees as base estimators.

Randomized decision trees are trained using a modified decision-tree learning algorithm, which introduces randomness when growing our trees. This additional source of randomness increases ensemble diversity and generally leads to better predictive performance.

The key difference between a standard decision tree and a randomized decision tree is in how a decision node is constructed. In standard decision tree construction, *all available features* are evaluated exhaustively to identify the best feature to split on. Since decision-tree learning is a greedy algorithm, it will choose the highest scoring features to split on.

When bagging, this exhaustive enumeration (combined with greedy learning) means that it is often possible the same small number of dominant features are repeatedly used in different trees. This makes the ensemble less diverse.

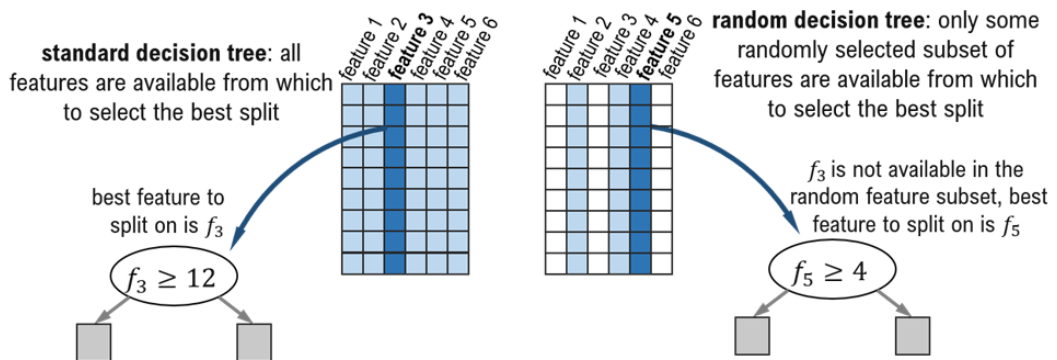
To overcome this limitation of standard decision tree learning, random forests introduce randomness in tree learning. Specifically, instead of considering all the features to identify the best split, a *random subset of features* is evaluated to identify the best feature to split on.

Thus, random forests use a modified tree learning algorithm, which first randomly samples features before creating a decision node. The resulting tree is a *randomized decision tree*.

### **Example: Randomization in Tree Learning**

Consider tree learning on a data set with six features (here,  $\{f_1, f_2, f_3, f_4, f_5, f_6\}$ ). In standard tree learning, all six features are evaluated, and the best splitting feature is identified (say,  $f_3$ ).

In randomized decision tree learning, we first identify a random subset of features (say,  $\{f_2, f_3, f_4, f_5\}$ ) and then choose the best from among them (which is, say,  $f_5$ ). Thus, randomization has inherently forced the tree learning to identify a different feature.



**Figure 2.7** Random forests use a modified tree learning algorithm, where a random feature subset is first chosen before the best splitting criterion for each decision node is identified.

Ultimately, this randomization occurs every time a decision node is constructed. Thus, even if we used the same data set, we will obtain a different randomized tree each time we train.

When randomized tree learning (with random sampling of features) is combined with bootstrap sampling (with random sampling of training examples), we obtain an ensemble of randomized decision trees, known as a random decision forest or simply random forest.

The random forest ensemble will be more diverse than bagging, which only performs bootstrap sampling. Next, we will see how to use random forests in practice.

### 2.3.2 Random Forests with scikit-learn

`scikit-learn` provides an efficient implementation of random forests that also supports out-of-bag estimation and parallelization. Since random forests are specialized to use decision trees as base learners, `RandomForestClassifier` also takes `DecisionTreeClassifier` parameters such as `max_leaf_nodes` and `max_depth` to control tree complexity. The listing below demonstrates how to call `RandomForestClassifier`.

#### Listing 2.4 Random Forests with `scikit-learn`

```
from sklearn.ensemble import RandomForestClassifier

rf_ens = RandomForestClassifier(n_estimators=500, max_depth=10, #A
                              oob_score=True, n_jobs=-1) #B #C
rf_ens.fit(Xtrn, ytrn)
ypred = rf_ens.predict(Xtst)
```

#A Control complexity of base decision trees  
 #B Parallelize, if possible  
 #C Use out-of-bag sample to estimate generalization error

Figure 2.8 illustrates a random forest classifier, along with several component base estimators.

### 2.3.3 Feature Importances

One benefit of using random forests is that they also provide a natural mechanism for scoring features based on their importance. This means that we can rank features to identify the most important ones and drop low impact features, thus performing *feature selection*!

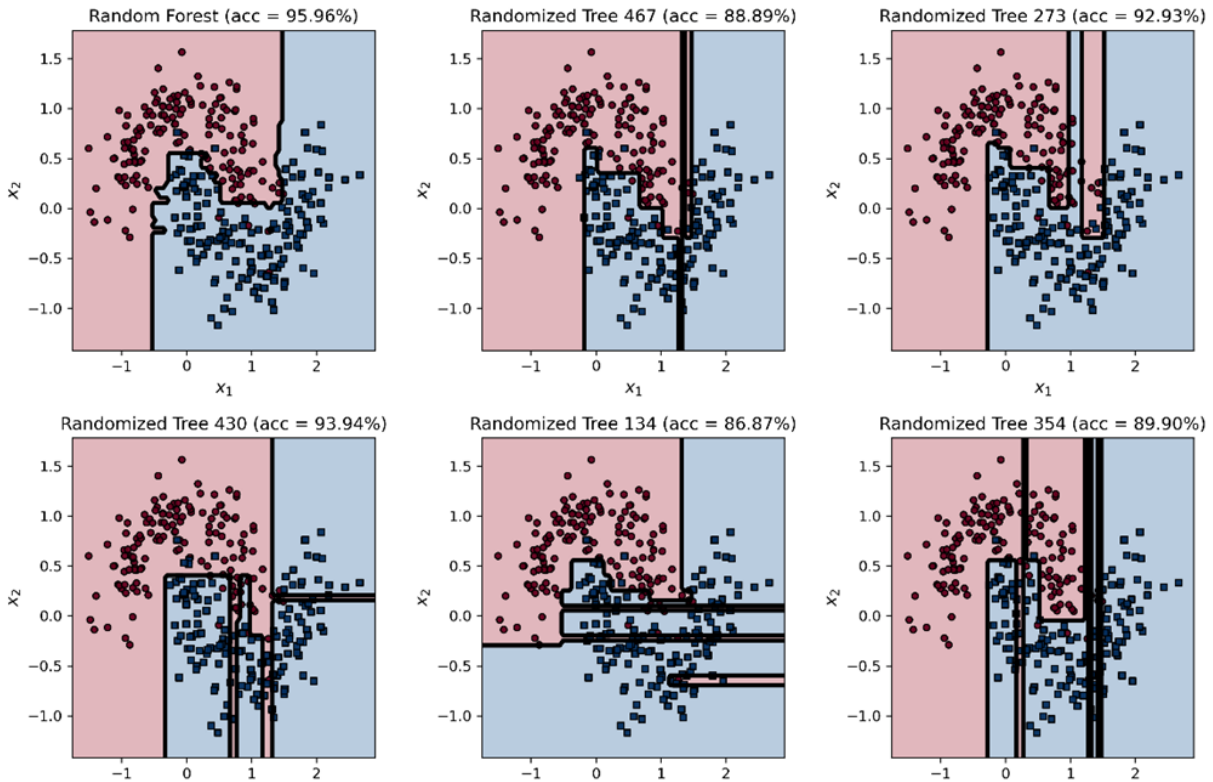
#### Feature Selection

Feature selection, also known as variable subset selection, is the procedure of identifying the most impactful or relevant data features/attributes. Feature selection is an important step of the modeling process, especially for high-dimensional data.

Dropping the least relevant features often improves generalization performance and minimizes overfitting. It also often improves computational efficiency of training. These issues are consequences of the *curse of dimensionality*, where the large number of features inhibits the model's ability to generalize effectively.

See *The Art of Feature Engineering: Essentials for Machine Learning* by Pablo Duboue to learn more about feature selection and engineering.





**Figure 2.8** Random forest (top left) compared to individual base learners (randomized decision trees). Much like bagging, the random forest ensemble also produces a smooth and stable decision boundary. Also observe the effect of randomization on the individual trees, which are far spikier than regular decision trees.

We can obtain the feature importances for the simple two-dimensional data set after fitting a random forest ensemble with the query, `rf_ens.feature_importances_`:

```
>>> for i, score in enumerate(rf_ens.feature_importances_):
    print('Feature x{0}: {1:6.5f}'.format(i, score))
Feature x0: 0.47017
Feature x1: 0.52983
```

The feature scores for the simple two-dimensional data set suggest that both features are roughly equally important. In the case study towards the end of the chapter, we will compute and visualize the feature importances for a data set from a real task: breast cancer diagnosis.

Note that feature importances sum to one and are effectively *feature weights*. Less important features have lower weights and can often be dropped without significantly affecting the overall quality of the final model, while improving training and prediction times.

**CAUTION** If two features are strongly correlated or dependent, then intuitively, we know that it is sufficient to use either one of them in the model. However, *the order in which the features are used can affect feature importance*.

For instance, when classifying abalone (sea snails), the features `size` and `weight` are highly correlated (unsurprising, since bigger snails will be heavier). This means that including them in a decision tree will add roughly the same amount of information, and cause the overall impurity to decrease by roughly the same amount. Thus, we expect that their mean impurity decrease scores will be the same.

However, say we select `weight` first as a splitting variable. Adding this feature to the tree removes information contained in both `size` and `weight` features. This means that the feature importance of `size` is reduced because any impurity that `size` could decrease was already previously decreased by adding `weight`. This results in imbalanced feature importances. Random feature selection mitigates this problem a little, but not consistently.

In general, you must proceed with caution when *interpreting feature importances* in the presence of feature correlations, so that you don't miss the whole story in the data.

## 2.4 More Homogeneous Parallel Ensembles

We have seen two important parallel homogeneous ensemble methods: bagging and random forest. We now explore a few variants that were developed for large data sets (for example, recommendation systems) or high-dimensional data (for example, image or text databases).

These include bagging variants such as pasting, random subspaces and random patches and an extreme random forest variant called ExtraTrees. All these methods introduce randomization in different ways in order to ensure ensemble diversity.

### 2.4.1 Pasting

Bagging uses bootstrap sampling, or sampling with replacement. If, instead, we sample subsets for training *without replacement*, we have a variant of bagging known as *pasting*.

Pasting was designed for very large data sets, where sampling with replacement is not necessary. Instead, since training full models on data sets of such scale is difficult, pasting aims to take small pieces of the data by sampling *without replacement*.

Pasting exploits the fact that, with a very large data set, sampling without replacement can inherently generate ensemble diversity. Pasting also ensures that each training subsample is a small piece of the overall data set and can be used to train a base learner efficiently.

Model aggregation is still used to make a final ensemble prediction. However, since each base learner is trained on small pieces of the large data set, we can view model aggregation as *pasting the predictions* of the base learners together for a final prediction.

`BaggingClassifier` can easily be extended to perform pasting by setting `bootstrap=False` and making it subsample small subsets for training by setting `max_samples` to a small fraction, say `max_samples=0.05`.

## 2.4.2 Random Subspaces and Random Patches

It is possible to make the base learners even more diverse *by randomly sampling the features as well*. Instead of sampling training examples, if we generate subsets by sampling features (with or without replacement), we obtain a variant of bagging called *Random Subspaces*.

`BaggingClassifier` supports bootstrap sampling of features through two parameters: `bootstrap_features` (default: `False`) and `max_features` (default: `1.0`, or all the features), which are analogous to the parameters `bootstrap` (default: `False`) and `max_samples` for sampling training examples.

```
bag_ens = BaggingClassifier(
    base_estimator=SVC(), n_estimators=100,
    max_samples=1.0, bootstrap=False, #A
    max_features=0.5, bootstrap_features=True) #B
```

#A Use all the training samples  
#B Bootstrap sample 50% of features

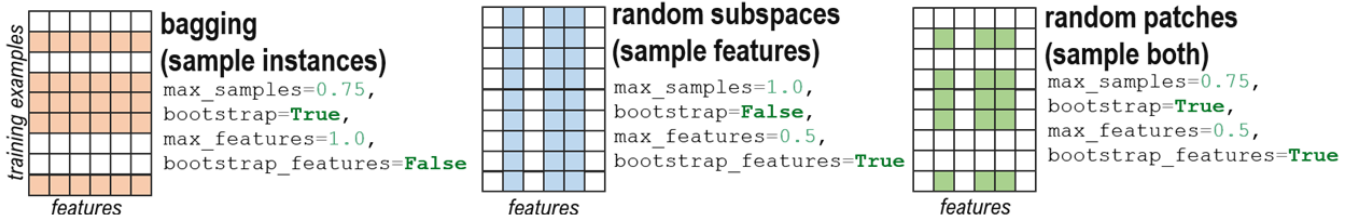


Figure 2.9 Bagging compared to random subspaces and random patches.

If we randomly sample *both training examples and features* (with or without replacement), we obtain a variant of bagging called *Random Patches*.

```
bag_ens = BaggingClassifier(
    base_estimator=SVC(), n_estimators=100,
    max_samples=0.75, bootstrap=True, #A
    max_features=0.5, bootstrap_features=True) #B
```

#A Bootstrap sample 75% of examples  
#B Bootstrap sample 50% of features

Note that in the examples above, the base estimator is the support vector classifier, `sklearn.svm.SVC`. In general, random subspaces and random patches can be applied to any base learner to improve estimator diversity. As with bagging, in practice, increasing diversity tends to increase bias slightly.

**TIP** In practice, these variants of bagging can be especially effective for *big data*. For example, since random subspaces and random patches sample features, they can be used to train base estimators more efficiently for data with lots of features, such as image data.

Alternately, since pasting samples without replacement, it can be used to train base estimators more efficiently when you have a big data set with a lot of training instances.

The key difference between random forests and bagging variants such as random subspaces and random patches is where the feature sampling occurs. Random forests exclusively use randomized decision trees as base estimators. Specifically, they perform feature sampling *inside the tree learning algorithm* each time they grow the tree with a decision node.

Random subspaces and random patches, on the other hand, are not restricted to tree learning and can use any learning algorithm as a base estimator. They randomly sample features once outside before calling the base learning algorithm for each base estimator.

### 2.4.3 ExtraTrees

Extremely randomized trees take the idea of randomized decision trees to the extreme by selecting not just the splitting variable from a random subset of features (see Figure 2.9) but also the splitting threshold!

Say we have selected the feature  $f_s$  to split on. In typical decision tree learning, we next search for the best threshold ( $\tau$ ) to split on. This leads to a decision node of the form  $f_s \leq \tau$ . Instead, when training extremely randomized trees, the threshold  $\tau$  is also selected randomly.

This extreme randomization is so effective, in fact, that we can construct an ensemble of extremely randomized trees directly from the original data set *without bootstrap sampling!* This means that we can construct an ExtraTrees ensemble very efficiently.

**TIP** In practice, ExtraTrees ensembles are well suited for high-dimensional data sets with a large number of continuous features.

`scikit-learn` provides an `ExtraTreesClassifier` that supports out-of-bag estimation and parallelization, much like `BaggingClassifier` and `RandomForestClassifier`. Note that ExtraTrees typically *do not perform bootstrap sampling* (`bootstrap=False`, by default), as we are able to achieve base estimator diversity through extreme randomization.

**CAUTION** `scikit-learn` provides two very similarly named classes: `sklearn.tree.ExtraTreeClassifier` and `sklearn.ensemble.ExtraTreesClassifier`. `sklearn.tree.ExtraTreeClassifier` provides an implementation of extremely randomized trees, and should only be used with ensemble methods. `sklearn.ensemble.ExtraTreesClassifier` provides the implementation of the ExtraTrees ensemble method discussed above.

## 2.5 Case Study: Breast Cancer Diagnosis

Our first case study explores a medical decision-making task: *breast cancer diagnosis*. We will see how to use `scikit-learn`'s homogeneous parallel ensemble modules in practice. Specifically, we will train and evaluate the performance of three homogeneous parallel algorithms, each characterized by increasing randomness: bagging with decision trees, random forests and ExtraTrees.

Doctors make many decisions regarding patient care everyday: tasks such as diagnosis (what disease does the patient have?), prognosis (how will their disease progress?), treatment planning (how should the disease be treated?), to name a few. They make these decisions based on a patient's health records, medical history, family history, test results and so on.

The specific data set we will use is the Wisconsin Breast Cancer (WDBC) data set, a common benchmark data set in machine learning. Since 1993, the WDBC data has been used to benchmark the performance of dozens of machine-learning algorithms.

The machine-learning task is to train a classification model that can diagnose patients with breast cancer. By modern standards and in the era of big data, this is a small data set. It is, however, perfectly suited to show the ensemble methods we've seen so far in action.

### 2.5.1 Loading and pre-processing

The WDBC data set was originally created by applying feature extraction techniques on patient biopsy medical images. More concretely, for each patient, the data describe the size and texture of the cell nuclei of cells extracted during biopsy.

WDBC is available in `scikit-learn` and can be loaded as shown below.

```
>>> from sklearn.datasets import load_breast_cancer
>>> dataset = load_breast_cancer()
```

	index	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	diagnosis
0	372	21.37	15.10	141.30	1386.0	0.10010	0.15150	0.19320	0
1	8	13.00	21.82	87.50	519.8	0.12730	0.19320	0.18590	0
2	228	12.62	23.97	81.35	496.4	0.07903	0.07529	0.05438	1
3	369	22.01	21.90	147.20	1482.0	0.10630	0.19540	0.24480	0
4	212	28.11	18.47	188.50	2499.0	0.11420	0.15160	0.32010	0

**Figure 2.10** The WDBC data set consists of 569 training examples, each described by 30 features. A few of the 30 features for a small subset of patients, along with each patient's *diagnosis* (training label) are shown above. `diagnosis=1` indicates malignant and `diagnosis=0` indicates benign.

It is always good practice to standardize the training data, especially since the scales of the features are vastly different. *Standardization* is a pre-processing step that rescales features to have a mean of 0 and standard deviation of 1.

Data is pre-processed this way so that all the features are scaled similarly before learning, while preserving information in each column. This ensures that one feature does not numerically dominate another simply because of their different scales.

```
>>> from sklearn.preprocessing import StandardScaler
>>> X, y = dataset['data'], dataset['target']
>>> X = StandardScaler().fit_transform(X)
```

## 2.5.2 Bagging, Random Forests and ExtraTrees

Once we have pre-processed our data set, we will train and evaluate bagging with decision trees, random forests and ExtraTrees in order to answer the following questions:

1. How does the ensemble performance change with ensemble size? That is, what happens when our ensembles get bigger and bigger?
2. How does the ensemble performance change with base learner complexity? That is, what happens when our individual base estimators become more and more complex. In this case study, since all three ensemble methods considered use decision trees as base estimators, one “measure” of complexity is tree depth, with deeper trees being more complex.

### *Ensemble size vs. ensemble performance*

First, we look at how training and testing performance change with ensemble size. That is, we compare the behavior of the three algorithms as the parameter `n_estimators` increases.

As always, we follow good machine learning practices and split the data set into a training set and a hold-out test set randomly. Our goal will be to learn a diagnostic model on the training set and evaluate how well that diagnostic model does using the test set.

Recall that since the test set is held-out during training, the test error is generally a useful estimate of how well we will do on future data, that is, generalize. However, since we don't want our learning and evaluation to be at the mercy of randomness, we will repeat this experiment 20 times and average the results.

#### **Listing 2.5 Training and test errors with increasing ensemble size**

```
max_leaf_nodes = 8 #A
n_runs = 20
n_estimator_range = range(2, 20, 1)

bag_trn_error = np.zeros((n_runs, len(n_estimator_range))) #B
rf_trn_error = np.zeros((n_runs, len(n_estimator_range)))
xt_trn_error = np.zeros((n_runs, len(n_estimator_range)))

bag_tst_error = np.zeros((n_runs, len(n_estimator_range)))
rf_tst_error = np.zeros((n_runs, len(n_estimator_range)))
xt_tst_error = np.zeros((n_runs, len(n_estimator_range)))

for run in range(0, n_runs):
    X_trn, X_tst, y_trn, y_tst = train_test_split(X, y, test_size=0.25) #C
```

```

for j, n_estimators in enumerate(n_estimator_range):

    tree = DecisionTreeClassifier(max_leaf_nodes=max_leaf_nodes) #D
    bag = BaggingClassifier(base_estimator=tree,
                           n_estimators=n_estimators,
                           max_samples=0.5, n_jobs=-1)

    bag.fit(X_trn, y_trn)
    bag_trn_error[run, j] = 1 - accuracy_score(y_trn, bag.predict(X_trn))
    bag_tst_error[run, j] = 1 - accuracy_score(y_tst, bag.predict(X_tst))

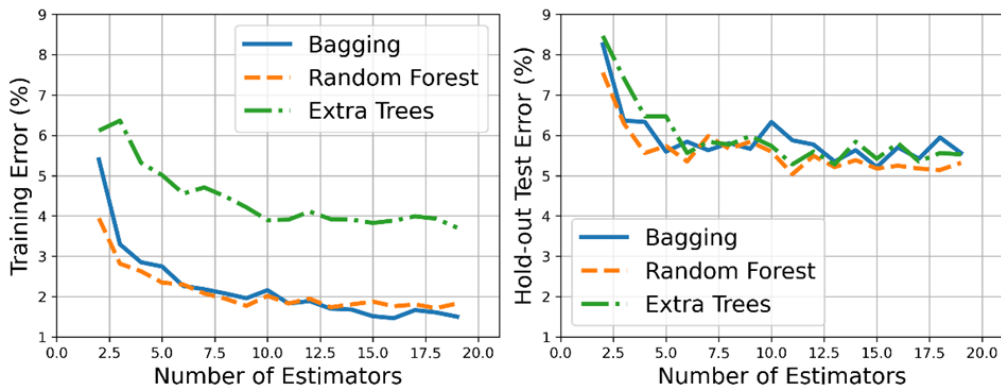
    rf = RandomForestClassifier(max_leaf_nodes=max_leaf_nodes, #E
                               n_estimators=n_estimators, n_jobs=-1)

    rf.fit(X_trn, y_trn)
    rf_trn_error[run, j] = 1 - accuracy_score(y_trn, rf.predict(X_trn))
    rf_tst_error[run, j] = 1 - accuracy_score(y_tst, rf.predict(X_tst))

    xt = ExtraTreesClassifier(max_leaf_nodes=max_leaf_nodes, #F
                              bootstrap=True, n_estimators=n_estimators,
                              n_jobs=-1)
    xt.fit(X_trn, y_trn)
    xt_trn_error[run, j] = 1 - accuracy_score(y_trn, xt.predict(X_trn))
    xt_tst_error[run, j] = 1 - accuracy_score(y_tst, xt.predict(X_tst))

```

#A Every base decision tree in every ensemble will have at most 8 nodes  
 #B Initialize arrays to store training and test errors  
 #C Perform 20 runs, each with a different split of train/test data  
 #D Train and evaluate bagging for this  
 #E Train and evaluate random forests for  
 #F Train and evaluate ExtraTrees for this



**Figure 2.11** Training and test performance of bagging, random forest and ExtraTrees as ensemble size increases. Bagging used decision trees as the base estimator, random forest used randomized decision trees and ExtraTrees used extremely randomized trees.

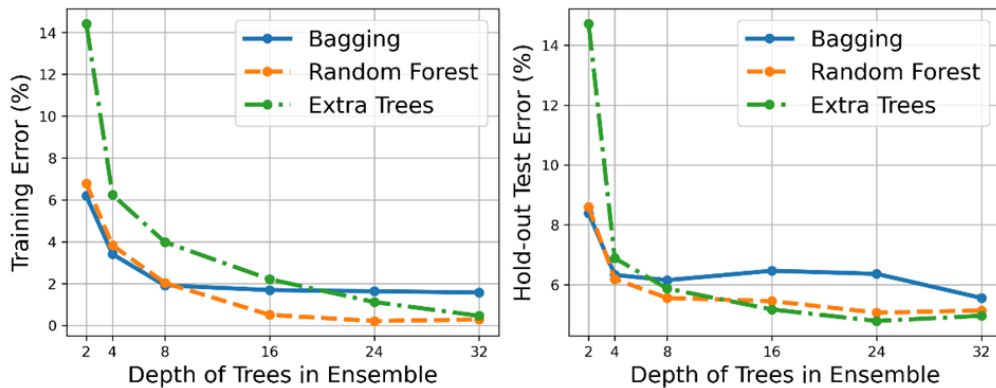
We can now visualize the averaged training and test errors on the WDBC data set.

As expected, the training error for all the approaches decreases steadily as the number of estimators increases. The test error also decreases with ensemble size and then stabilizes. As the test error is an estimate of the generalization error, our experiment confirms our intuition about the performance of these ensemble methods in practice.

Finally, all three approaches greatly outperform single decision trees (where the plot begins). This shows that, in practice, even if single decision trees are unstable, ensemble of decision trees are robust and can generalize well.

### ***Base learner complexity vs. ensemble performance***

Next, we compare the behavior of the three algorithms as the complexity of the base learners increases. There are several ways to control the complexity of the base decision trees: maximum depth, maximum number of leaf nodes, impurity criteria, etc. Here, we compare the performance of the three ensemble methods as with complexity as determined by `max_leaf_nodes`. This comparison can be performed in a manner similar to the previous one.



**Figure 2.12** Training and test performance of bagging, random forest and ExtraTrees as base learner complexity increases. Bagging used decision trees as the base estimator, random forest used randomized decision trees and ExtraTrees used extremely randomized trees.

Recall that highly complex trees are inherently unstable and sensitive to small perturbations in the data. This means that, in general, if we increase the complexity of the base learners, we will need a lot more of them to successfully reduce the variance of the ensemble overall. Here, however, we have fixed `n_estimators=10`.

One key consideration in determining the depth of the base decision trees is computational efficiency. Learning deeper and deeper trees will take more and more time, but will not produce the significant improvement in predictive performance. For instance, base decision trees of depths 24 and 32 perform roughly similarly.



### 2.5.3 Feature importances with Random Forests

Finally, we see how we can use feature importances to identify the most predictive features for breast cancer diagnosis using the random forest ensemble. Such analysis adds *interpretability to the model* and can be very helpful in communicating and explaining such models to domain experts such as doctors.

#### *Feature Importances from Label Correlations*

First, let's peek into the data set to see if we can discover some interesting relationships among the features and the diagnosis. This type of analysis is typical when we get a new data set, as we try to learn more about it. Here, our analysis will try to identify which features are most correlated with each other and with the `diagnosis` (label), so that we can check if random forests can do something similar.

In the code below, we use the `pandas` and `seaborn` packages to visualize feature and label correlations.

#### **Listing 2.6 Visualizing correlations between features and labels**

```
import pandas as pd
import seaborn as sea
import matplotlib.pyplot as plt

df = pd.DataFrame(data=dataset['data'], columns=dataset['feature_names']) #A
df['diagnosis'] = dataset['target']

fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(12, 8))
cor = np.abs(df.corr())
sea.heatmap(cor, annot=False, cbar=False, cmap=plt.cm.Red, ax=ax[0])

f = ['mean radius', 'mean perimeter', 'mean area', \ #B
     'worst radius', 'worst perimeter', 'worst area', \
     'radius error', 'perimeter error', 'area error', 'diagnosis']
cor_zoom = np.abs(df[f].corr())
sea.heatmap(cor_zoom, annot=True, cbar=False, cmap=plt.cm.Red, ax=ax[1])
fig.tight_layout()
```

**#A** Convert the data into a pandas dataframe

**#B** Compute and plot the correlation between the features and label (diagnosis)

The output of this listing is shown in Figure 2.13. There are several features that are highly correlated with each other. For example, mean radius, mean perimeter and mean area. There are several features that are also highly correlated with the label, that is, the diagnosis as benign or malignant. Let's identify the 10 features most correlated with the diagnosis label:

```
>>> label_corr = cor.iloc[:, -1]
>>> label_corr.sort_values(ascending=False)[1:11]

worst concave points    0.793566
worst perimeter        0.782914
mean concave points    0.776614
```

```
worst radius      0.776454
mean perimeter    0.742636
worst area        0.733825
mean radius       0.730029
mean area         0.708984
mean concavity    0.696360
worst concavity   0.659610
```

Thus, our correlation analysis is telling us that the 10 features above are the most highly correlated with the diagnosis. That is to say, these features are likely most helpful in breast cancer diagnosis.

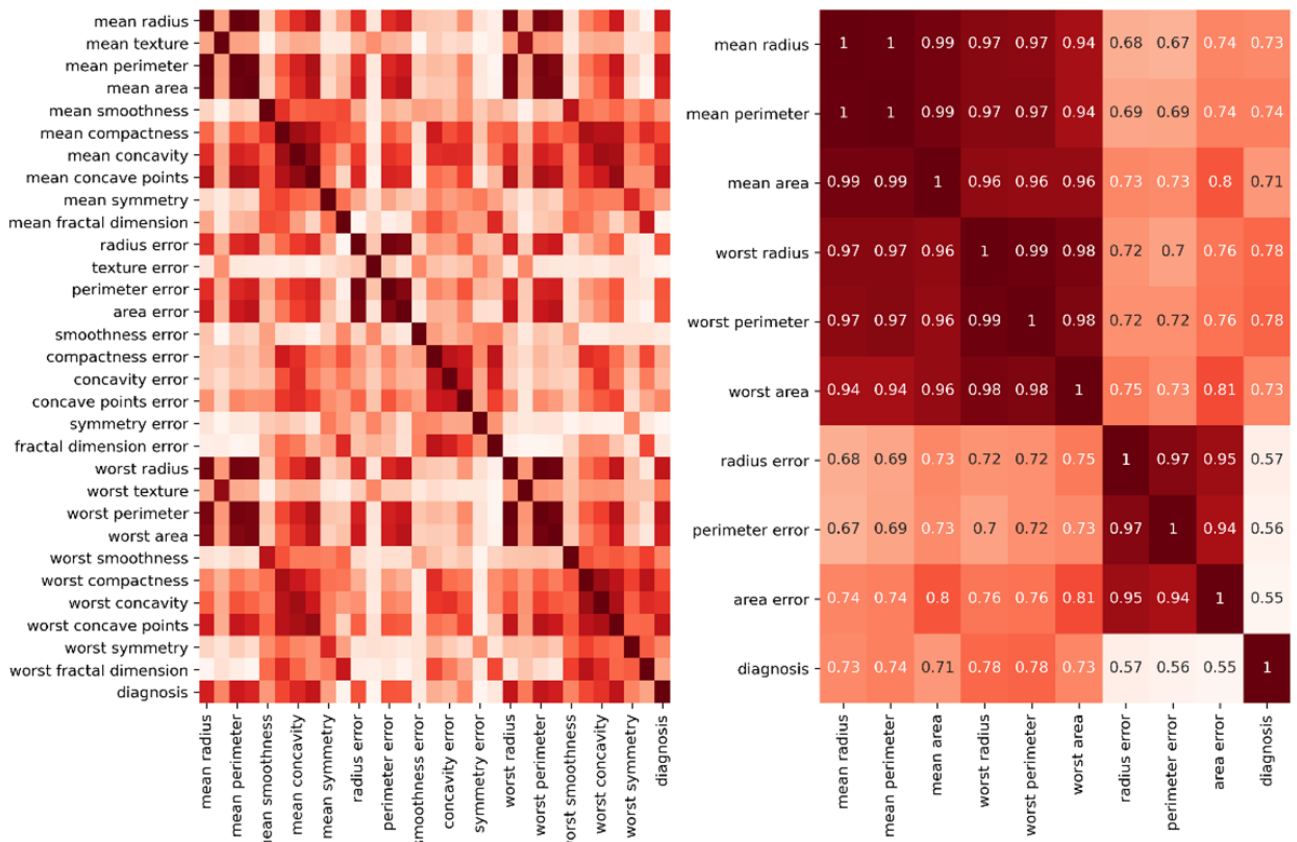


Figure 2.13 (Absolute) feature correlations between all 30 features and diagnosis (left) and a smaller subset of features and diagnosis (right).

### Feature Importances using Random Forests

Random forests can also provide feature importances. The listing below illustrates this.

**Listing 2.7 Feature importances in the WDBC data set using random forests**

```
X_trn, X_tst, y_trn, y_tst = train_test_split(X, y, test_size=0.15)
n_features = X_trn.shape[1]

rf = RandomForestClassifier(max_leaf_nodes=24, n_estimators=50, n_jobs=-1) #A
rf.fit(X_trn, y_trn)
err = 1 - accuracy_score(y_tst, rf.predict(X_tst))
print('Prediction Error = {0:4.2f}%'.format(err*100))

importance_threshold = 0.02 #B
for i, (feature, importance) in enumerate(zip(dataset['feature_names'],
                                           rf.feature_importances_)):

    if importance > importance_threshold:
        print('[{0}] {1} (score={2:4.3f})'.format(i, feature, importance)) #C
```

**#A** Train a random forest ensemble

**#B** Set an importance threshold, all the features above the threshold are important

**#C** Print the “important” features, i.e., those that are above the importance threshold

The listing above depends on an `importance_threshold`, which is set to 0.02 here. Typically, such a threshold is set by inspection such that we get a target feature set, or using a separate validation set to identify such that overall performance does not degrade.

For the WDBC data set, the random forest identifies the following features as being important. Observe that there is a considerable overlap between important features identified by correlation analysis and random forests, though their relative rankings are different.

```
[0] mean radius (score=0.077)
[1] mean texture (score=0.020)
[2] mean perimeter (score=0.045)
[3] mean area (score=0.041)
[5] mean compactness (score=0.026)
[6] mean concavity (score=0.036)
[7] mean concave points (score=0.120)
[13] area error (score=0.020)
[20] worst radius (score=0.193)
[22] worst perimeter (score=0.138)
[23] worst area (score=0.054)
[26] worst concavity (score=0.027)
[27] worst concave points (score=0.096)
```

Finally, we can plot the feature importances as identified by the random forest ensemble.

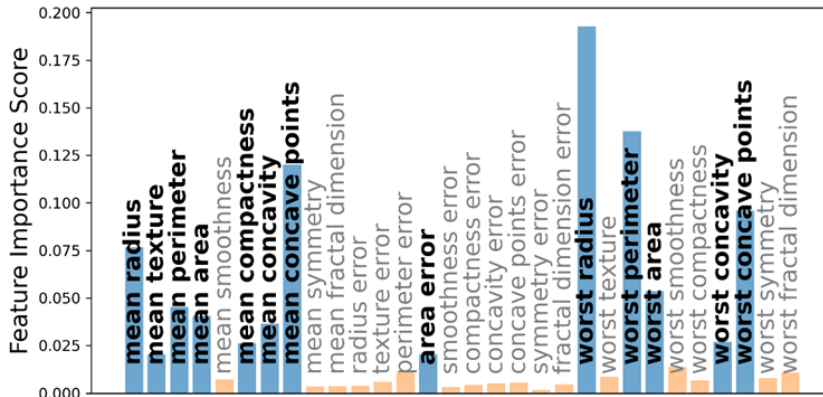


Figure 2.14 The random forest ensemble can score features by their importance. This allows us to perform feature selection, by only using features with the highest scores.

**CAUTION** Note that feature importances will often change between runs owing to randomization during tree construction. Note also that if two features are highly correlated, random forests will often distribute the feature importance between both of them, leading to their overall weights appearing smaller than they actually are.

## 2.6 Summary

In this chapter, we explored several parallel, homogeneous ensemble methods. Parallel ensemble methods train their base estimators *independently* of each other. Homogeneous ensemble methods use the *same base learning algorithm to train* their base estimators.

1. Parallel homogeneous ensembles promote ensemble diversity through *randomization*: random sampling of training examples, of features, or even introducing randomization in the base learning algorithm.
2. *Bagging* is a simple ensemble method that relies on (1) *bootstrap sampling* (or sampling with replacement) to generate diverse replicates of the data set and training diverse models, and (2) model aggregation to produce an ensemble prediction from a set of individual base learner predictions.
3. Bagging and its variants work best with any unstable estimators (unpruned decision trees, SVMs with nonlinear RBF kernels, deep neural networks, etc.), which are models of higher complexity and/or nonlinearity.
4. *Random forests* are a variant of bagging, specifically designed to use randomized decision trees as base learners. Increasing randomness increases ensemble diversity considerably, allowing the ensemble to decrease the variance, though at the cost of a slight increase in bias.
5. Pasting, a variant of bagging, samples training examples without replacement and can be effective on data sets with a very large number of training examples.

6. Other variants of bagging such as random subspaces (sampling features) and random patches (sampling both features and training examples) can be effective on data sets with high dimensionality.
7. Random forests provide feature importances to rank the most important features from a predictive standpoint.

In the next chapter, we will continue with parallel ensemble methods, and move on to another type: parallel heterogeneous ensembles.

# 3

## *Heterogeneous Parallel Ensembles: Combining Strong Learners*

### **This chapter covers**

- Combining base learning models by performance-based weighting
- Combining base learning models with meta-learning: stacking
- Avoiding overfitting by ensembling with cross validation
- A large-scale, real-world text-mining case study with heterogeneous ensembles

In the previous chapter, we introduced two parallel ensemble methods: bagging and random forest. These methods (and their variants) train *homogeneous ensembles*, where every base estimator is trained using the same base learning algorithm. For example, in bagging classification, all the base estimators are decision tree classifiers. In this chapter, we continue exploring parallel ensemble methods, this time focusing on *heterogeneous ensembles*.

Heterogeneous ensemble methods use different base learning algorithms to directly ensure ensemble diversity. For example, a heterogeneous ensemble can consist of three base estimators: a decision tree, a support vector machine (SVM) and an artificial neural network. These base estimators are still trained independently of each other.

The earliest heterogeneous ensemble methods such as stacking were developed as far back as 1992. However, these methods really came to the fore during the Netflix Prize competition in the mid-2000s. The top 3 teams, including the one that eventually won the \$1 million prize, were ensemble teams, and their solutions were a complex blend of hundreds of different base

models. This success was a striking and very public demonstration of the effectiveness of many of the methods we will be discussing in this chapter.

Inspired by this success, stacking and blending have become widely popular. With sufficient base estimator diversity, these algorithms can often boost performance on your data set and are a powerful ensembling tool in any data analyst's arsenal.

Another reason for their popularity is that they can easily combine existing models, which allows us to use previously trained models as base estimators. For example, say you and a friend were working independently on a data set for a Kaggle competition. You trained an SVM, while your friend trained a logistic regression model. While your individual models are doing ok, you both figure that you may do better if you put your heads (and models) together. You build a heterogeneous ensemble with these existing models without having to train them all over again. All you need to figure out would be a way to combine your two models.

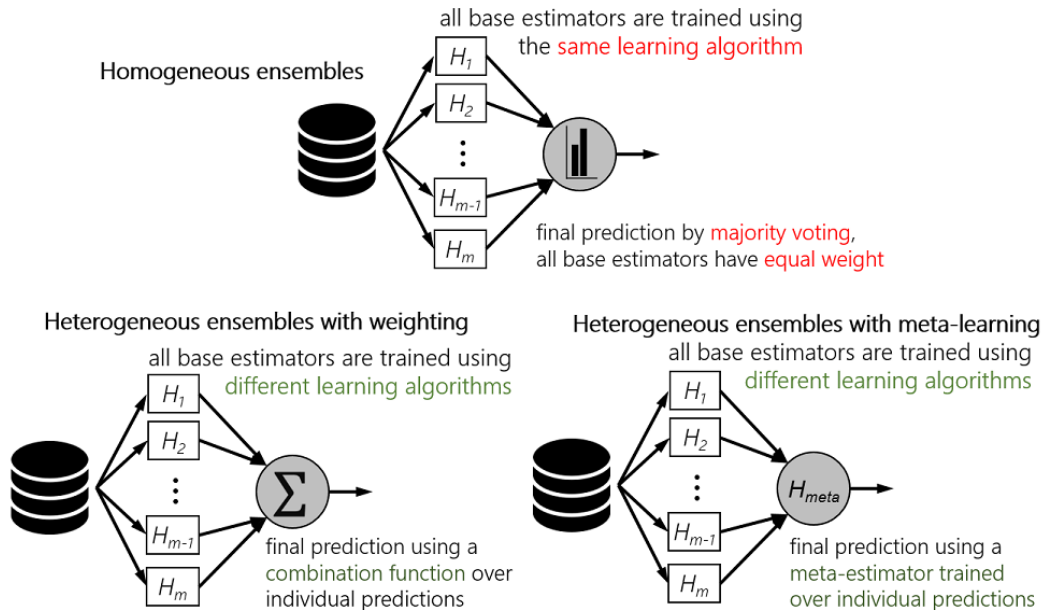


Figure 3.1 Homogeneous ensembles (Chapter 2), such as bagging and random forests, use the same learning algorithm to train base estimators, and achieve ensemble diversity through random sampling. Heterogeneous ensembles (this chapter) use different learning algorithms to achieve ensemble diversity.

Heterogeneous ensembles come in two flavors, depending on how they combine individual base estimator predictions into a final prediction (see Figure 3.1):

1. *weighting methods*, which assign individual base estimator predictions a weight that corresponds to its strength; better base estimators are assigned higher weights and

influence the overall final prediction more; the predictions of individual base estimators are fed into a pre-determined combination function, which makes the final predictions.

2. *meta-learning methods*, which use a learning algorithm to combine the predictions of base estimators; the predictions of individual base estimators are treated as meta-data and passed to a second-level meta-learner, which is trained to make final predictions.

We begin by introducing weighting methods, which combine classifiers by weighting the contribution of each one based on how effective it is.

### 3.1 Base estimators for heterogeneous ensembles

In this section, we will set up a learning framework for fitting heterogeneous base estimators and getting predictions from them. This is the first step in building heterogeneous ensembles for any application and corresponds to training the individual base estimators  $H_1, H_2, \dots, H_m$  in Figure 3.1 (bottom).

We will train our base estimators using a simple two-dimensional data set so we can explicitly visualize the decision boundaries and behavior of each base estimator as well as the diversity the estimators. Once trained, we can construct a heterogeneous ensemble using a weighting method (Section 3.2) or a meta-learning method (Section 3.3).

```
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
X, y = make_moons(600, noise=0.25, random_state=13) #A
X, Xval, y, yval = train_test_split(X, y, test_size=0.25) #B
Xtrn, Xtst, ytrn, ytst = train_test_split(X, y, test_size=0.25)
```

**#A** Set aside 25% of the data for validation

**#B** Set aside a further 25% of data for hold out-testing



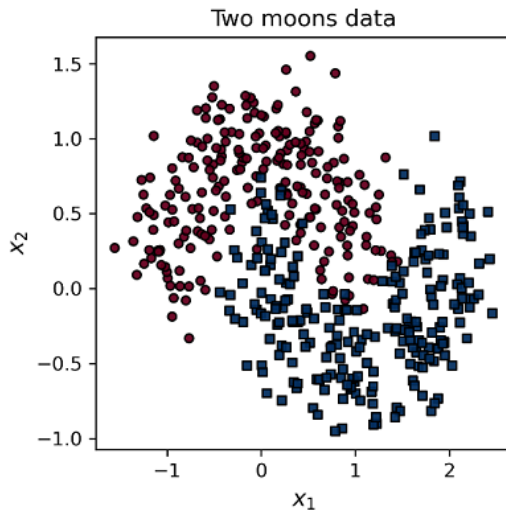


Figure 3.2 Synthetic data set with two classes: 300 examples each in Class 0 (•) and Class 1 (•).

### 3.1.1 Fitting base estimators

Our first task is to train the individual base estimators. Unlike homogeneous ensembles, we can use any number of different learning algorithms and parameter settings to train base estimators. The key is to ensure that we choose learning algorithms that are different enough to produce a diverse collection of estimators.

The more diverse our set of base estimators, the better the resulting ensemble will be. For this scenario, we use six popular machine-learning algorithms, all of which are available in `scikit-learn`: `DecisionTreeClassifier`, `SVC`, `GaussianProcessClassifier`, `KNeighborsClassifier`, `RandomForestClassifier` and `GaussianNB`.

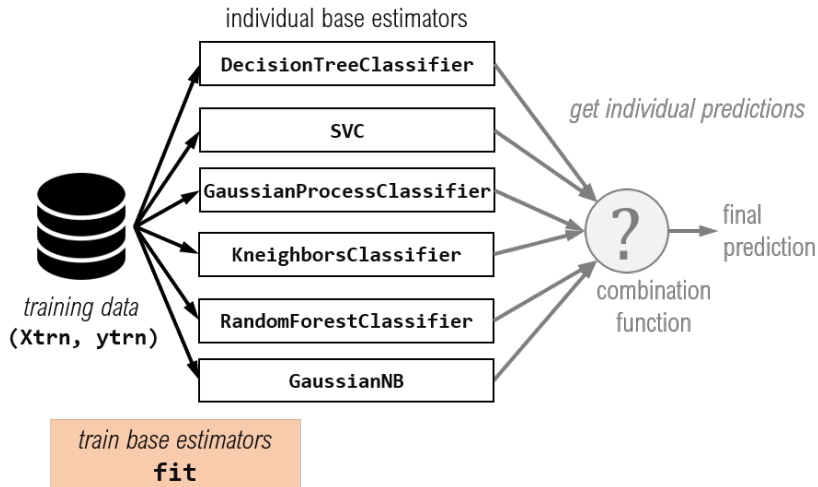


Figure 3.3 Fitting six base estimators using `scikit-learn`.

The listing below initializes six base estimators (described in Figure 3.3 above) and trains them. Note the individual parameter settings used to initialize each base estimator (for example, `max_depth=5` for `DecisionTreeClassifier` or `n_neighbors=3` for `KNeighborsClassifier`). In practice, these parameters have to be chosen carefully. For this simple data set, we can guess or just use the default parameter recommendations.

### Listing 3.1. Fitting different base estimators

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF
from sklearn.ensemble import RandomForestClassifier
from sklearn.naive_bayes import GaussianNB

estimators = [('dt', DecisionTreeClassifier(max_depth=5)), #A
              ('svm', SVC(gamma=1.0, C=1.0, probability=True)),
              ('gp', GaussianProcessClassifier(RBF(1.0))),
              ('3nn', KNeighborsClassifier(n_neighbors=3)),
              ('rf', RandomForestClassifier(max_depth=3,
                                          n_estimators=25)),
              ('gnb', GaussianNB())]

def fit(estimators, X, y):
    for model, estimator in estimators:
        estimator.fit(X, y) #B
    return estimators
```

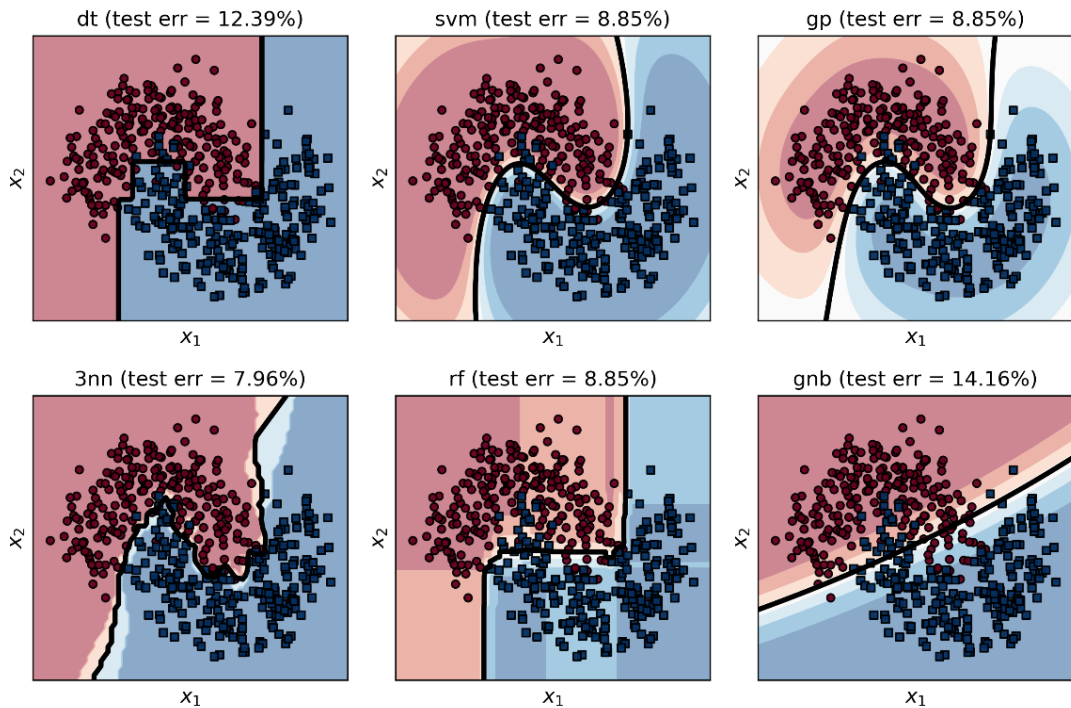
#A Initialize several base learning algorithms

#B Fit base estimators on the training data using these different learning algorithms

We train our base estimators on the training data

```
>>> estimators = fit(estimators, Xtrn, ytrn)
```

Once trained, we can also visualize how each base estimator behaves on our data set. It appears we were able to produce some pretty decently diverse base estimators.



**Figure 3.4** Base estimators in our heterogeneous ensemble. Each base estimator was trained using a different learning algorithm, which generally leads to a diverse ensemble. *Note that due to randomized data generation, the estimators you get, and their performances may be different.*

Aside from ensemble diversity, one other aspect that is immediately apparent from the visualization of individual base estimators is that they all don't perform equally well on a held-out test set. In Figure 3.4, 3-nearest neighbor (3nn) has the best test set performance, while Gaussian naïve Bayes (gnb) has the worst.

For instance, `DecisionTreeClassifier` (dt) produces classifiers that partition the feature space into decision regions using axis-parallel boundaries (because each decision node in the tree splits on a single variable). Alternately, the support vector machine (svm) classifier `svc`

uses an RBF kernel, which leads to smoother decision boundaries. Thus, while both learning algorithms can learn nonlinear classifiers, they are nonlinear in different ways.

### Kernel Methods

Support vector machines are an example of a *kernel method*, a type of machine-learning algorithm that can use kernel functions. A kernel function can efficiently measure the similarity between two data points implicitly in a high-dimensional space without explicitly transforming the data into that space. A linear estimator can be turned into a nonlinear estimator by replacing inner product computations with a kernel function. Commonly used kernels include the polynomial kernel and the Gaussian (also known as the radial basis function or RBF) kernel. See Chapter 12 of *The Elements of Statistical Learning: Data Mining, Inference, and Prediction (2<sup>nd</sup> ed.)* by Hastie, Tibshirani and Friedman for details.

### 3.1.2 Individual predictions of base estimators

Given test data to predict ( $x_{\text{tst}}$ ), we can get the predictions of each test example using each base estimator. In our scenario, given that we have six base estimators, each test example will have six predictions, one corresponding to each base estimator.

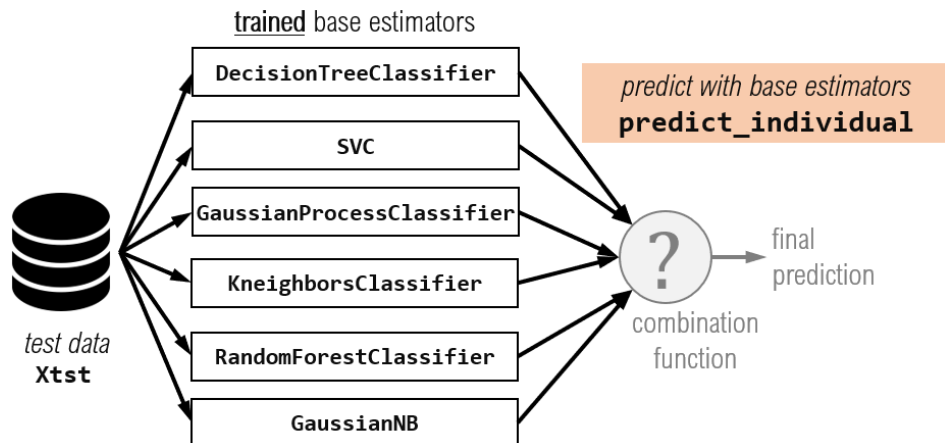


Figure 3.5 Individual predictions of a test set with the six trained six base estimators in `scikit-learn`.

Our task now is to collect the predictions of each test example by each trained base estimator into an array. In the listing below, the variable `y` is the structure that holds the predictions and is of size `n_samples * n_estimators`. That is, the entry `y[15, 1]` will be the prediction of the 2<sup>nd</sup> classifier (`svc`) on the 16<sup>th</sup> test example (indices in Python begin from 0).

#### Listing 3.2. Individual predictions of base estimators

```
import numpy as np

def predict_individual(X, estimators, proba=False):    #A
```

```

n_estimators = len(estimators)
n_samples = X.shape[0]

y = np.zeros((n_samples, n_estimators))
for i, (model, estimator) in enumerate(estimators):
    if proba:
        y[:, i] = estimator.predict_proba(X)[:, 1] #B
    else:
        y[:, i] = estimator.predict(X) #C
return y

```

#A this flag allows us to predict the labels directly or the probability over the labels  
 #B if true, predict probability of Class 1 (returns a number between 0 and 1)  
 #C otherwise, directly predict if Class 1 (returns 0 or 1)

Observe that our function `predict_individual` has a flag `proba`. When we set `proba=False`, `predict_individual` returns the predicted labels according to each estimator; the predicted labels take the values  $y_{pred} = 0$  or  $y_{pred} = 1$ , and tell us that the estimator has predicted that example belongs to Class 0 or Class 1 respectively.

When we set `proba=True`, however, each estimator will return the class prediction probabilities instead via each base estimator's `predict_proba()` function:

```
y[:, i] = estimator.predict_proba(X)[:, 1]
```

### Classification Probability

Most classifiers in `scikit-learn` can return the probability of a label rather than the label directly. Some of them, such as `SVC`, should be explicitly told to do so (notice that we set `probability=True` when initializing `SVC`), while others are natural probabilistic classifiers and can represent and reason over class probabilities. These probabilities represent each base estimator's *confidence in its prediction*.

We can use this function to predict the test examples:

```

>>> y_individual = predict_individual(Xtst, estimators, proba=False)
>>> y_individual
array([[0., 0., 0., 0., 0., 0.],
       [1., 1., 1., 1., 1., 1.],
       ...
       [1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1.]])

```

Each row contains six predictions, each one corresponding to the prediction of each base estimator. We sanity check our predictions: `Xtst` has 113 test examples, and `y_individual` has six predictions for each of them, which gives us a  $113 \times 6$  array of predictions.

```

>>> Xtst.shape
(113, 2)
>>> y_individual.shape
(113, 6)

```

When `proba=True`, `predict_individual` returns the probability that an example belongs to Class 1,  $P(y_{pred} = 1)$ . For two-class (binary) classification problems such as this one, the probability that the example belongs to Class 0 is simply  $1 - P(y_{pred} = 1)$ , as the example can only belong to one or the other and probabilities over all possibilities sum to 1.

```
>>> y_individual = predict_individual(Xtst, estimators, proba=True)
>>> np.set_printoptions(precision=2)
>>> y_individual
array([[0.00e+00, 5.77e-02, 1.69e-01, 3.33e-01, 1.26e-01, 1.66e-02],
       [1.00e+00, 9.81e-01, 9.43e-01, 1.00e+00, 9.87e-01, 9.97e-01],
       ...,
       [0.00e+00, 1.21e-02, 7.30e-02, 0.00e+00, 1.28e-02, 2.66e-03],
       [9.51e-01, 8.63e-01, 7.38e-01, 1.00e+00, 8.67e-01, 9.46e-01]])
```

In the first row of the output above, the third entry is 0.169. This indicates that our third base estimator, the `GaussianProcessClassifier`, is 16.9% confident that the first test example belongs to Class 1. Conversely, the `GaussianProcessClassifier` is 83.1% confident that the first test example belongs to Class 0.

Such prediction probabilities are often called *soft predictions*. Soft predictions can be converted to hard (0–1) predictions by simply picking the class label with the highest probability; in this example, according to the `GaussianProcessClassifier`, the hard prediction would be  $y=0$ , since  $P(y=0) > P(y=1)$ .

For the purpose of building a heterogeneous ensemble, we can either use the predictions directly or their probabilities. Using the latter typically produces a smoother output.

**CAUTION** The prediction function above is specifically written for two-class, that is, binary classification problems. It can be extended to multi-class problems, if care is taken to store the prediction probabilities for each class. That is, for multi-class problems, you will need to store the individual prediction probabilities in an array of size `n_samples * n_estimators * n_classes`.

We have now set up the basic infrastructure we need to create a heterogeneous ensemble. We have trained six classifiers, and we have a function that gives us their individual predictions on a new example. The last and, of course, the most important step is how we can combine these individual predictions. There are two ways of doing this: by weighting or by meta-learning.

## 3.2 Combining predictions by weighting

What do weighting methods aim to do? Let us return to the performance of the 3-nearest neighbor classifier (3nn) and the Gaussian naïve Bayes classifier (gnb) on our simple 2d data set (see Figure 3.6). Imagine we were trying to build a very simple heterogeneous classifier using these two as base estimators.

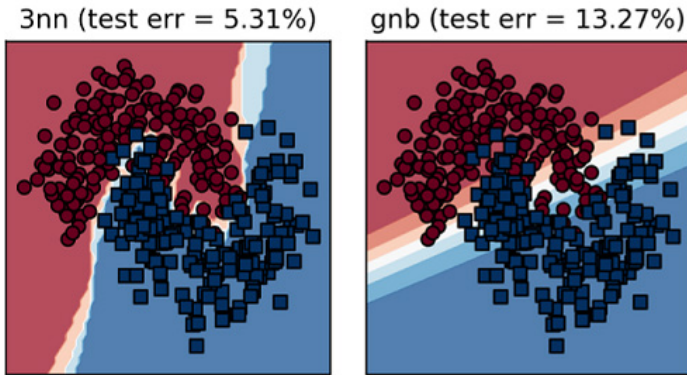


Figure 3.6 Two base estimators can have very different behaviors on the same data set. A weighting strategy should reflect their performance by weighting better performing classifiers higher.

Let's say we compare the behavior of these two classifiers using test error as our evaluation metric. The test error can be evaluated using the examples in  $X_{tst}$ , which was held-out during training; this gives us a good estimate of how the models will behave on future, unseen data.

3nn has a test error rate of 5.31%, while gnb has a test error rate of 13.27%. Intuitively, we would trust the 3nn classifier more *on this data set* than the gnb classifier. However, this does not mean that gnb is useless and should be discarded. For many examples, it can reinforce the decision made by 3nn. What we don't want it to do is contradict 3nn when it is not confident of its predictions.

This notion of base estimator confidence can be captured by assigning them weights. When we are looking to assign weights to base classifiers, we should do so in a manner consistent with this intuition, such that the final prediction is influenced more by the stronger classifiers and less by the weaker classifiers.

Say we are given a new data point  $x$ , and the individual predictions are  $y_{3nn}$  and  $y_{gnb}$ . A simple way to combine them would be to weight them based on their performance. The test set accuracy of 3nn is  $\alpha_{3nn} = 1 - 0.0531 = 0.9469$ , and the test accuracy of gnb is  $\alpha_{gnb} = 1 - 0.1327 = 0.8673$ . The final prediction can be computed as:

$$y = \underbrace{\frac{\alpha_{3nn}}{\alpha_{3nn} + \alpha_{gnb}}}_{w_{3nn}} \cdot y_{3nn} + \underbrace{\frac{\alpha_{gnb}}{\alpha_{3nn} + \alpha_{gnb}}}_{w_{gnb}} \cdot y_{gnb}$$

The estimator weights  $w_{3nn}$  and  $w_{gnb}$  are proportional to their respective accuracies and the higher accuracy classifier will have the higher weight. In this example, we have  $w_{3nn} = 0.522$  and  $w_{gnb} = 0.478$ . We have combined the two base estimators using a simple linear combination function (technically, a convex combination, since all the weights are positive and sum to one).

Let's continue with the task of classifying our 2d two-moons data set and explore various weighting and combination strategies. This will typically consist of two steps (see Figure 3.7):

1. assign weights ( $w_{clf}$ ) to each classifier ( $clf$ ) in some way, reflecting its importance;
2. combine the weighted predictions ( $w_{clf} \cdot y_{clf}$ ) using a combination function  $h_c$ .

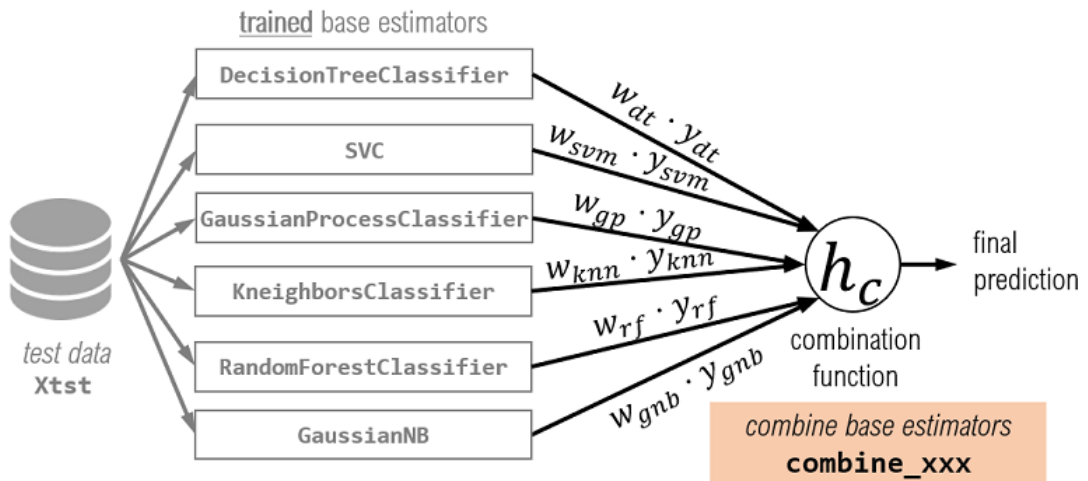


Figure 3.7 Each base classifier is assigned an importance weight that reflects how much its opinion contributes to the final decision. Weighted decisions of each base classifier are combined using a combination function.

We now look at several such strategies that generalize this intuition above for both hard and soft predictions.

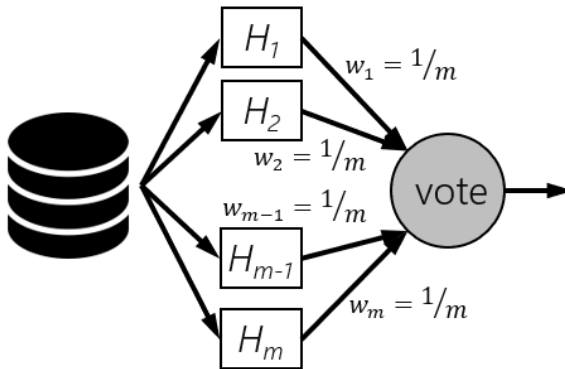
### 3.2.1 Majority Vote

We are already familiar with one type of weighted combination from the previous chapter: the majority vote. We briefly revisit it here to show that it is but one of many combination schemes and to put it into the general framework of combination methods.

Majority voting can be viewed as a weighted combination scheme: each base estimator is assigned an equal weight: that is, if we have  $m$  base estimators, each base estimator has a weight  $w_{clf} = 1/m$ . The (weighted) predictions of the individual base estimators are combined using the majority vote.

Like bagging, this strategy can be extended to heterogeneous ensembles as well. In the general combination scheme presented in Figure 3.7, to implement this weighting strategy, we set  $w_{clf} = 1/m$  and  $h_c = \text{majority vote}$ , which is the statistical mode.





**Figure 3.8** Combining by majority voting. Bagging can be viewed as a simple weighting method applied to a homogeneous ensemble. All classifiers have equal weights and the combination function  $h_c$  is the majority vote. We can adopt the majority voting strategy for heterogeneous ensembles as well.

The listing below combines the individual predictions `y_individual` from a heterogeneous set of base estimators using majority voting. Note that since the weights of the base estimators are all equal, we do not explicitly compute them.

### Listing 3.3. Combine predictions using majority vote

```
from scipy.stats import mode

def combine_using_majority_vote(X, estimators):
    y_individual = predict_individual(X, estimators, proba=False)
    y_final = mode(y_individual, axis=1)
    return y_final[0].reshape(-1, ) #A
```

#A reshape the vectors to ensure it returns one prediction per example

We can use this function to make predictions on the test data set, `Xtst`, using our previously trained base estimators:

```
>>> from sklearn.metrics import accuracy_score
>>> ypred = combine_using_majority_vote(Xtst, estimators)
>>> tst_err = 1 - accuracy_score(ytst, ypred)
>>> tst_err
0.07964601769911506
```

This weighting strategy produces a heterogeneous ensemble with test error 7.96%.

## 3.2.2 Accuracy weighting

Recall our motivating example at the start of this section, where we were trying to build a very simple heterogeneous classifier using 3-nearest neighbor (3nn) and Gaussian naïve Bayes (gnb) as base estimators. In that example, our intuitive ensembling strategy was to weight

each estimator by its performance, specifically, the accuracy score. That was a very simple example of accuracy weighting.

Here, we generalize this procedure to more than two estimators as in Figure 3.8. In order to get *unbiased performance estimates* for the base classifiers, we will use a *validation set*.

### ***Why Do We Need A Validation Set?***

When we generated our data set, we partitioned it into a *training set*, a *validation set* and a *hold-out test set*. The three subsets are mutually exclusive, that is, they don't have any overlapping examples. So, which of these three should we use to obtain unbiased estimates of the performance of each individual base classifier?

It is always good machine-learning practice to *not reuse the training set* for performance estimates. Why? Since we've already seen this data, the performance estimate will be biased.

This is like seeing a previously-seen homework problem on your final exam. It doesn't really tell the professor that you're performing well because you've learned the concept. It just tells the professor that you are good at that specific problem.

Using training data to estimate performance doesn't tell us if a classifier can generalize well; it just tells us how well it does on examples it's already seen. To get an effective and unbiased estimate, we will need to evaluate performance on data the model has never seen before.

We can get unbiased estimates using either the validation set or the hold-out test set. However, the test set will often be used to evaluate the *final model performance*, that is, the performance of *the overall ensemble*.

Here, we are interested in estimating the performance of *each base classifier*. It is for this reason that we use the validation set: we use it to obtain unbiased estimates of each base classifier's performance: accuracy.

### ***Accuracy Weights Using a Validation Set***

Once we have trained each base classifier (clf), we evaluate its performance on a validation set. Let  $\alpha_t$  be the validation accuracy of the  $t$ -th classifier,  $H_t$ . The weight of each base classifier is then computed as

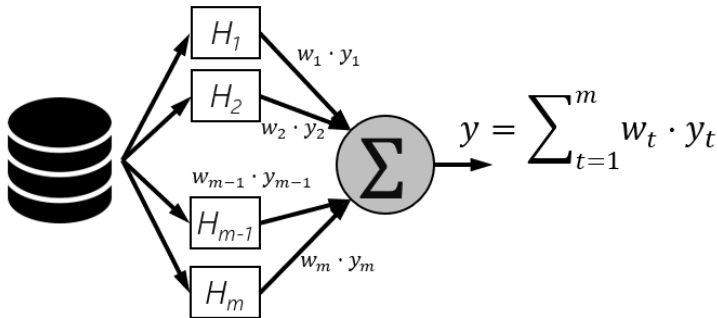
$$w_t = \frac{\alpha_t}{\sum_{t=1}^m \alpha_t}$$

The denominator is a normalization term: the sum of all the individual validation accuracies. This computation ensures that a classifier's weight is proportional to its accuracy and all the weights sum to 1.

Given a new example to predict  $x$ , we can get the predictions of the individual classifiers,  $y_t$  (using `predict_individual`). Now, the final prediction can be computed as weighted sum of the individual predictions

$$y_{final} = w_1 \cdot y_1 + w_2 \cdot y_2 + \dots + w_m \cdot y_m = \sum_{t=1}^m w_t \cdot y_t$$

This procedure is illustrated in the figure below.



**Figure 3.9** Combining by performance weighting. Each classifier is assigned a weight proportional to its accuracy. The final prediction is computed as a weighted combination of the individual predictions.

The listing below implements combination by accuracy weighting. It should be noted that, while the individual classifier predictions will have values 0 or 1, the overall final prediction will be a real number between 0 and 1, as the weights are fractions. This fractional prediction can be converted to a 0–1 final prediction easily by thresholding the weighted predictions on 0.5.

For example, a combined prediction of  $y_{final}=0.75$  will be converted to  $y_{final}=1$  (because  $0.75 >$  the 0.5 threshold), while a combined prediction of  $y_{final}=0.33$  will be converted to  $y_{final}=0$  (because  $0.33 <$  the 0.5 threshold). Ties, while extremely rare, can be broken arbitrarily.

#### Listing 3.4. Combine using accuracy weighting

```
def combine_using_accuracy_weighting(X, estimators, Xval, yval): #A
    n_estimators = len(estimators)
    yval_individual = predict_individual(Xval, estimators, proba=False) #B

    wts = [accuracy_score(yval, yval_individual[:, i])
            for i in range(n_estimators)] #C
    wts /= np.sum(wts) #D

    ypred_individual = predict_individual(X, estimators, proba=False)
    y_final = np.dot(ypred_individual, wts) #E

    return np.round(y_final) #F
```

#A pass the validation set

#B get individual predictions on the validation set

```
#C set the weight for each baseclassifier as its accuracy score
#D normalize the weights
#E compute the weighted combination of individual labels efficiently
#F convert the combined prediction into a 0–1 label by rounding
```

We can use this function to make predictions on the test data set, `Xtst`, using our previously trained `base_estimators`:

```
>>> ypred = combine_using_accuracy_weighting(Xtst, estimators, Xval, yval)
>>> tst_err = 1 - accuracy_score(ytst, ypred)
>>> tst_err
0.026548672566371723
```

This weighting strategy produces a heterogeneous ensemble with test error 2.65%.

### 3.2.3 Entropy weighting

The entropy weighting approach is another performance-based weighting approach, except that it uses entropy as the evaluation metric to judge the value of each base estimator. Entropy is a measure of *uncertainty* or impurity in a set; a more disorderly set will have higher entropy.

#### Entropy

Entropy, or information entropy to be precise, was originally devised by Claude Shannon to quantify the “amount of information” conveyed by a variable. This is determined by two factors: (1) the number of distinct values the variable can take, and (2) the uncertainty associated with each value.

Consider that three patients: Ana, Bob and Cam are in the doctor’s office awaiting the doctor’s diagnosis of a disease. Ana is told with 90% confidence she is healthy (10% chance she is sick). Bob is told with 95% confidence that he is ill (that is, 5% chance he is healthy). Cam is told that his test results are inconclusive, that is, 50%-50%.

Ana has received good news and there is little uncertainty in her diagnosis. Even though Bob has received bad news, there is little uncertainty in his diagnosis as well. Cam’s situation has the highest uncertainty: he has received neither good nor bad news, and is in for more tests.

Entropy quantifies this notion of uncertainty across various outcomes. Entropy-based measures are commonly used during decision-tree learning to greedily identify the best variables to split on, and as loss functions in deep neural networks.

Instead of using accuracy to weight classifiers, we can use entropy. However, since lower entropies are desirable, we need to ensure that base classifier weights are *inversely proportional* to their corresponding entropies.

#### Computing entropy over predictions

Let’s say that we have a test set of 10 examples and a base estimator returned a vector of predicted labels: `[1, 1, 1, 0, 0, 1, 1, 1, 0, 0]`. This set has 6 predictions of  $y = 1$  and 4 predictions of  $y = 0$ . These label counts can be equivalently expressed as label probabilities: the probability of predicting  $y = 1$  is  $P(y = 1) = 6/10 = 0.6$  and the probability of predicting  $y = 0$

is  $P(y=0)=4/10 = 0.4$ . With these label probabilities, we can compute the entropy over this set of predictions as

$$E = -P(y=0) \log_2 P(y=0) - P(y=1) \log_2 P(y=1)$$

In this simple case, we will have  $E = -0.4 \cdot \log_2 0.4 - 0.6 \cdot \log_2 0.6 = 0.971$ .

Alternately, consider that a second base estimator returned a vector of predicted labels:  $[1, 1, 1, 1, 0, 1, 1, 1, 1, 1]$ . This set has 9 predictions of  $y=1$  and 1 prediction of  $y=0$ . The label probabilities in this case are  $P(y=1) = 9/10=0.9$  and  $P(y=0) = 1/10 = 0.1$ . The entropy in this case will be  $E = -0.1 \cdot \log_2 0.1 - 0.9 \cdot \log_2 0.9 = 0.469$ . This set of predictions has a lower entropy because it is purer (mostly all predictions are  $y=1$ ). Another way of viewing this is to say that the second classifier is less uncertain about its predictions.

The listing below can be used to compute the entropy of a set of discrete values.

### Listing 3.5. Computing entropy

```
def entropy(y):
    _, counts = np.unique(y, return_counts=True) #B
    p = np.array(counts.astype('float') / len(y)) #C
    ent = -p.T @ np.log2(p) #A

    return ent
```

```
#A compute entropy as a dot product
#B compute label counts
#C convert counts to probabilities
```

### Entropy Weighting with a Validation Set

Let  $E_t$  be the validation entropy of the  $t$ -th classifier,  $H_t$ . The weight of each base classifier is

$$w_t = \frac{1/E_t}{\sum_{t=1}^m (1/E_t)}$$

here are two key differences between entropy weighting and accuracy weighting:

1. the accuracy of a base classifier is computed using both the true labels  $y_{\text{true}}$  and the predicted labels  $y_{\text{pred}}$ . In this manner, the accuracy metric measures how well a classifier performs. A classifier with high accuracy is better.
2. the entropy of a base classifier is computed using only the predicted labels  $y_{\text{pred}}$ , and the entropy metric measures how uncertain a classifier is about its predictions. A classifier with low entropy (uncertainty) is better. Thus, individual base classifier weights are inversely proportional to their corresponding entropies.

As with accuracy weighting, the final predictions need to be thresholded at 0.5. The following listing implements combining with entropy weighting.

**Listing 3.6. Combine using entropy weighting**

```
def combine_using_entropy_weighting(X, estimators, Xval): #A
    n_estimators = len(estimators)
    yval_individual = predict_individual(Xval, estimators, proba=False) #B

    wts = [1/entropy(yval_individual[:, i]) #C

    for i in range(n_estimators)]
        wts /= np.sum(wts) #D

    ypred_individual = predict_individual(X, estimators, proba=False)
    y_final = np.dot(ypred_individual, wts) #E

    return np.round(y_final) #F#A pass only the validation examples
#B get individual predictions on the validation set
#C set the weight for each base classifier as its inverse entropy
#D normalize the weights
#E compute the weighted combination of individual labels efficiently
#F convert the combined prediction into a 0–1 label by rounding
```

We can use this function to make predictions on the test data set, `Xtst`, using our previously trained base `estimators`:

```
>>> ypred = combine_using_entropy_weighting(Xtst, estimators, Xval)
>>> tst_err = 1 - accuracy_score(ytst, ypred)
>>> tst_err
0.03539823008849563
```

This weighting strategy produces a heterogeneous ensemble with test error 3.54%.

### 3.2.4 Dempster-Shafer Combination

The methods we’ve seen so far combine predictions of individual base estimators directly. (notice that we’ve set the flag `proba=False` when calling `predict_individual`). When we set `proba=True` in `predict_individual`, each classifier returns its individual estimate of the probability of belonging to Class 1. That is, when `proba=True`, instead of returning `ypred = 0` or `ypred = 1`, each estimator will return  $P(ypred = 1)$ .

This probability reflects a classifier’s belief in what the prediction should be and offers a more nuanced view of the predictions. While the methods described above can also work with probabilities, the Dempster-Shafer method is another way to fuse these base estimator beliefs into an overall final belief, or prediction probability.

#### *Dempster-Shafer Theory For Label Fusion*

Dempster-Shafer Theory is a generalization of probability theory that supports reasoning under uncertainty and with incomplete knowledge. While the foundations of DST are beyond the scope of this book, the theory itself provides a way to fuse beliefs and evidence from multiple sources into one single belief.

DST uses a number between 0 and 1 to indicate belief in a proposition, such as “the test example  $x$  belongs to Class 1”. This number is known as a *basic probability assignment* (BPA) and expresses the certainty that the text example  $x$  belongs to Class 1. BPA values closer to 1 characterize decisions made with more certainty. The BPA allows us to translate an estimator’s confidence to a belief over the true label.

Let’s say a 3-nearest neighbor classifier (3nn) is used to classify a test example  $x$ , and it returns  $P(y_{pred} = 1 | 3nn) = 0.75$ . Now, Gaussian naïve Bayes (gnb) is also used to classify the same test example and returns  $P(y_{pred} = 1 | gnb) = 0.6$ . According to DST, we can compute the basic probability assignment for the proposition “test example  $x$  belongs to Class 1 according to both 3nn and gnb”. We do this by fusing their individual prediction probabilities:

$$\begin{aligned} BPA(y_{pred} = 1 | 3nn, gnb) &= 1 - (1 - P(y_{pred} = 1 | 3nn)) \cdot (1 - P(y_{pred} = 1 | gnb)) \\ &= 1 - (1 - 0.75) \cdot (1 - 0.6) = 0.9. \end{aligned}$$

We can also compute the basic probability assignment for the proposition “test example  $x$  belongs to Class 0 according to both 3nn and gnb”:

$$\begin{aligned} BPA(y_{pred} = 0 | 3nn, gnb) &= 1 - (1 - P(y_{pred} = 0 | 3nn)) \cdot (1 - P(y_{pred} = 0 | gnb)) \\ &= 1 - (1 - 0.25) \cdot (1 - 0.4) = 0.55. \end{aligned}$$

Based on these scores, we are more certain that the test example  $x$  belongs to Class 1. The BPAs can be thought of as certainty scores, with which we can compute our final belief of belonging to Class 0 or Class 1.

The unnormalized belief that “test example  $x$  belongs to Class 1” is computed as

$$\begin{aligned} Bel(y_{pred} = 1) &= \frac{BPA(y_{pred} = 1)}{1 - BPA(y_{pred} = 1)} = \frac{0.9}{0.1} = 9, \\ Bel(y_{pred} = 0) &= \frac{BPA(y_{pred} = 0)}{1 - BPA(y_{pred} = 0)} = \frac{0.55}{0.45} = 1.22 \end{aligned}$$

These unnormalized beliefs can be normalized using the normalization factor  $Z = Bel(y_{pred} = 1) + Bel(y_{pred} = 0) + 1$ , to give us  $Bel(y_{pred} = 1) = 0.8$  and  $Bel(y_{pred} = 0) = 0.11$ . We can use these beliefs to get the final prediction: the class with the highest belief; for this test example, the Dempster-Shafer method produces a final prediction of  $y_{pred} = 1$ .

### ***Combining using Dempster-Shafer***

The listing below implements this approach.

#### **Listing 3.7. Combining using Dempster-Shafer**

```
def combine_using_Dempster_Schafer(X, estimators):
    p_individual = predict_individual(X, estimators, proba=True) #A
    bpa0 = 1.0 - np.prod(p_individual, axis=1)
    bpa1 = 1 - np.prod(1 - p_individual, axis=1)
```

```

belief = np.vstack([bpa0 / (1 - bpa0), bpa1 / (1 - bpa1)]).T #B
y_final = np.argmax(belief, axis=1) #C
return y_final

```

```

#A get individual predictions on the validation set
#B Stack the beliefs for Class 0 and Class 1 side-by-side for every test example
#C select the final label as the class with the highest belief

```

We can use this function to make predictions on the test data set, `Xtst`, using our previously trained base estimators:

```

>>> ypred = combine_using_Dempster_Schafer(Xtst, estimators)
>>> tst_err = 1 - accuracy_score(ytst, ypred)
>>> tst_err
0.07079646017699115

```

We have seen four methods of combining predictions into one final prediction. Two use the predictions directly, while two use prediction probabilities. We can visualize the decision boundaries produced by these weighting methods.

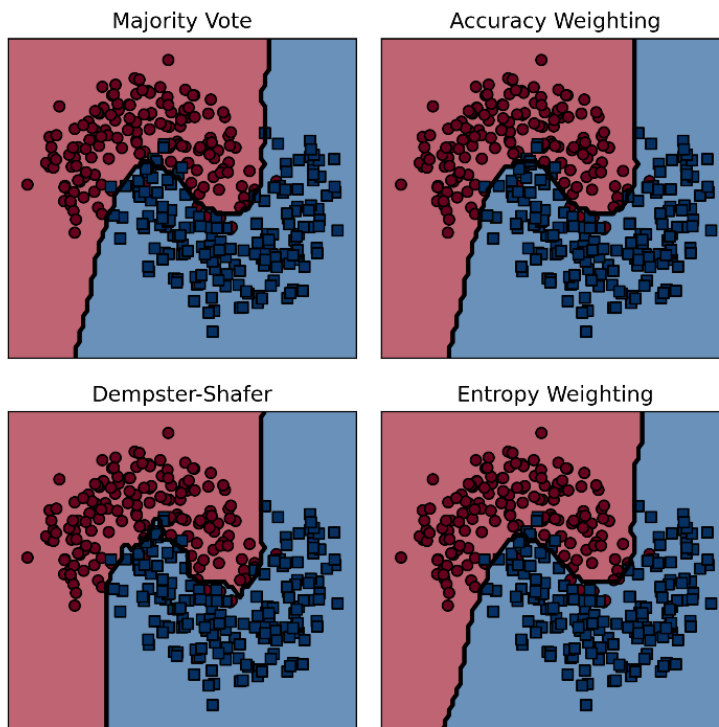


Figure 3.10 Decision boundaries of different weighting methods.



### 3.3 Combining predictions by meta-learning

In the previous section, we saw one approach to constructing heterogeneous ensembles of classifiers: weighting. We weighted each classifier by its performance and used a *pre-determined combination function* to combine predictions of each classifier. In doing so, we had to carefully design the combination function to reflect our performance priorities.

Now, we will look at another approach to constructing heterogeneous ensembles: meta-learning. Instead of carefully designing a combination function to combine predictions, we will *learn a combination function* over individual predictions. That is, the predictions of the base estimators are given as inputs to a second-level learning algorithm. Thus, rather than designing one ourselves, we will learn a second-level *meta-classification function*.

Meta-learning methods have been widely and successfully applied to a variety of tasks in chemometrics analysis, recommendation systems, text classification and spam filtering. For recommendation systems, meta-learning methods, stacking and blending, were brought to prominence after they were used by several top teams during the Netflix prize competition.

#### 3.3.1 Stacking

Stacking is the most common meta-learning method and gets its name because it stacks a second classifier on top of its base estimators. The general stacking procedure has two steps:

1. level 1: fit base estimators on the training data; this step is the same as before and aims to create a diverse, heterogeneous set of base classifiers.
2. level 2: construct a new data set from the output of the base classifiers, which become *meta-features*; meta-features can either be the predictions or the probability of predictions.

Let us return to our simple example, where we construct a simple heterogeneous ensemble from a 3-nearest neighbor classifier (3nn) and a Gaussian naïve Bayes classifier (gnb) on our 2d synthetic data set. After training the classifiers (3nn and gnb), we create new features, called *meta-features* from classifications of these two classifiers.

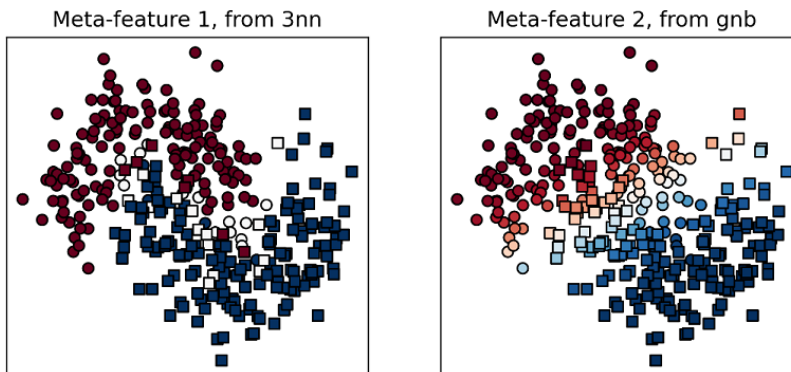


Figure 3.11 The probability of prediction of each training example according to 3nn and gnb are meta-features.

The darker a data point is, the more confident its prediction. Each training example now has two meta-features, one each from 3nn and gnb.

Since we have two base classifiers, we can use each one to generate one meta-feature in our meta-example. In this example, we use the prediction probabilities of 3nn and gnb as meta-features. Thus, for each training example, say  $x_i$ , we obtain two meta features:  $y^{i_{3nn}}$  and  $y^{i_{gnb}}$ , the prediction probabilities of  $x_i$  according to 3nn and gnb respectively.

These meta-features become meta-data for a second-level classifier. Contrast this stacking approach to combination by weighting. For both approaches, we use obtain individual predictions using the function `predict_individual`. For combination by weighting, we use these predictions directly in some *pre-determined combination function*. In stacking, we use these predictions as a new training set *to learn a combination function*.

Stacking can use any number of level 1 base estimators. Our goal, as always, will be to ensure that there is sufficient diversity among these base estimators. The figure below shows the stacking schematic for the six popular algorithms we have used previously to explore combining by weighting: `DecisionTreeClassifier`, `SVC`, `GaussianProcessClassifier`, `KNeighborsClassifier`, `RandomForestClassifier` and `GaussianNB`.

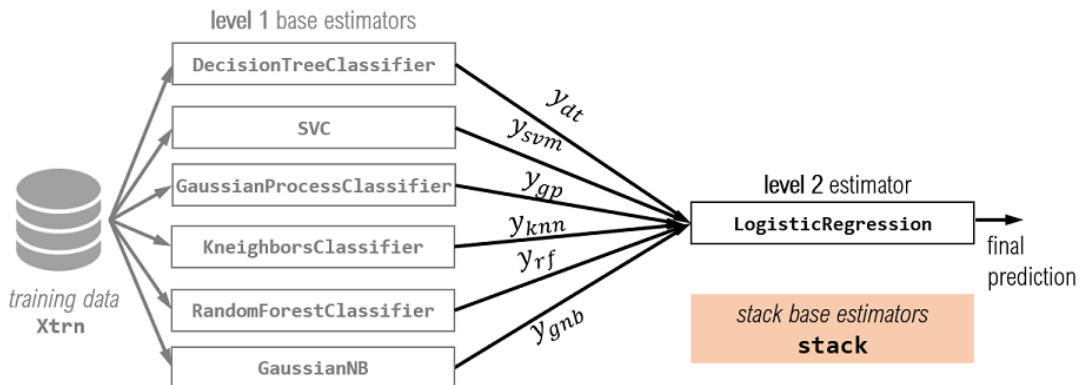


Figure 3.12 Stacking with six level 1 base estimators produces a meta-dataset of six meta-features that can be used to train a level 2 meta-classifier (here, logistic regression).

The level 2 estimator here can be trained using any base learning algorithm. Historically, linear models such as linear regression and logistic regression have been used. An ensembling method that use such linear models in the second level is called *linear stacking*. Linear stacking is generally popular because it is fast: learning linear models is generally computationally efficient, even for large data sets. Often, linear stacking can also be an effective exploratory step in analyzing your data set.

However, stacking can also employ powerful nonlinear classifiers in its second level, including SVMs and artificial neural networks. This allows the ensemble to combine meta-features in complex ways, though at the expense of interpretability inherent in linear models.

**NOTE** As of this writing, `scikit-learn` (v 0.22) contains unstable versions of `StackingClassifier` and `StackingRegressor`, and it is currently unknown when these will be available for general use. In the following subsections, we implement our own stacking algorithms. However, if you want to play around with the beta versions of these ensemble methods, you will need to install the development version of `scikit-learn`.

Let's revisit the task of classifying our 2d two-moons data set. We will implement a linear stacking procedure, which consists of the following steps: (1) train individual base estimators (level 1), (2a) construct meta-features and (2b) train a linear regression model (level 2).

We have already developed most of the framework we need to quickly implement linear stacking. We can train individual base estimators using `fit` (Listing 3.1) and obtain meta-features from `predict_individual` (Listing 3.2). The listing below uses these functions to fit a stacking model with any level 2 estimator. Since the level 2 estimator uses generated features or meta-features, it is also called a *meta-estimator*.

### Listing 3.8: Stacking with a second estimator

```
def fit_stacking(level1_estimators, level2_estimator,
                X, y, use_probabilities=False):

    fit(level1_estimators, X, y)#A

    X_meta = predict_individual(X, estimators=level1_estimators,
                              proba=use_probabilities) #B

    level2_estimator.fit(X_meta, y) #C

    final_model = {'level-1': level1_estimators,
                  'level-2': level2_estimator, #D
                  'use-proba': use_probabilities}

    return final_model
```

#A train level 1 base estimators

#B get meta features as individual predictions or prediction probabilities (proba=True/False)

#C train level 2 meta-estimator

#D save the level 1 estimators and level 2 estimator in a dictionary

This function can learn using either the predictions directly as the meta-data (`use_proba=False`) or using the prediction probabilities as the meta-data (`use_proba=True`).

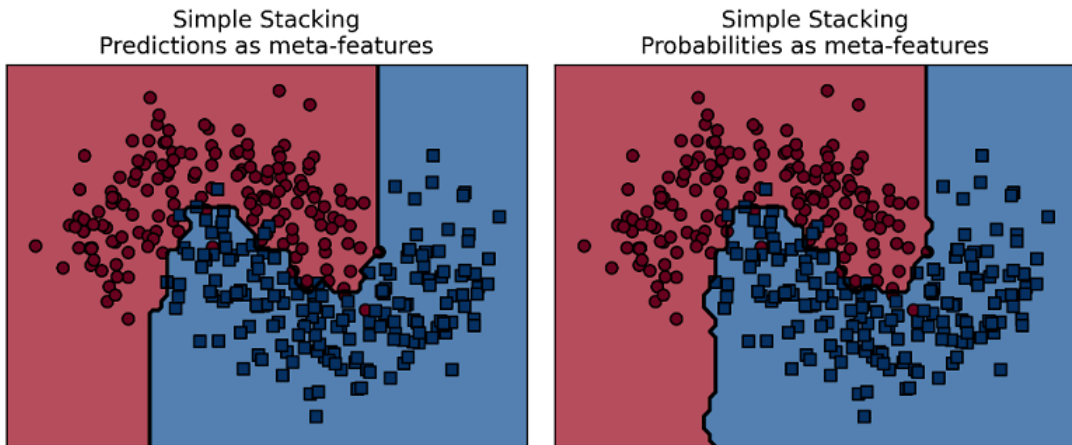


Figure 3.13 Final models produced by stacking with logistic regression using either predictions (left) or prediction probabilities (right) as meta-features.

The level 2 estimator here can be any classification model. Logistic regression is a common choice, which leads the ensemble to stack level 1 predictions using a linear model.

It is also possible to use a nonlinear model as a level 2 estimator. In general, any learning algorithm can be used to train a `level2_estimator` over the meta-features. A learning algorithm such as an SVM with RBF kernels or an artificial neural network can learn powerful nonlinear models at the second level and potentially improve performance even more.

Prediction proceeds in two steps: (1) for each test example, get the meta-features using the trained level 1 estimators and create a corresponding test meta-example, and (2) for each meta-example, get the final prediction using the level 2 estimator.

#### Listing 3.9: Making predictions with a stacked model

```
def predict_stacking(X, stacked_model):
    level1_estimators = stacked_model['level-1'] #A
    use_probabilities = stacked_model['use-proba']

    X_meta = predict_individual(X, estimators=level1_estimators,
                               proba=use_probabilities) #B

    level2_estimator = stacked_model['level-2']
    y = level2_estimator.predict(X_meta) #C

    return y
```

#A get level 1 base estimators

#B get meta-features using the level 1 base estimators

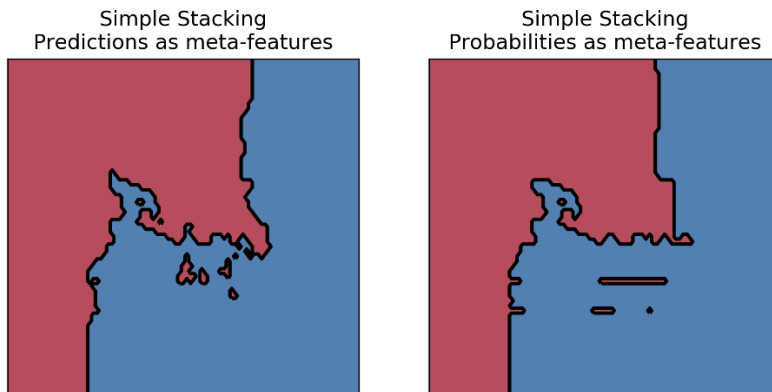
#C get level 2 estimator and use it to make the final predictions on the meta-features

In the example below, we use the same six base `estimators` from the previous section in level 1 and logistic regression as the level 2 meta-estimator:

```
>> from sklearn.linear_model import LogisticRegression
>> meta_estimator = LogisticRegression(C=1.0, solver='lbfgs')
>>> stacking_model = fit_stacking(estimators, meta_estimator,
...                               Xtrn, ytrn, use_proba=True)
>>> ypred = predict_stacking(Xtst, stacking_model)
>>> tst_err = 1 - accuracy_score(ytst, ypred)
>>> tst_err
0.056548672566371723
```

In the snippet above, we used the prediction probabilities as meta-features. This linear stacking model obtains a test error of 5.65%.

This simple stacking procedure is often effective. However, it does suffer from one significant drawback: *overfitting*, especially in the presence of noisy data. The effects of overfitting can be observed in the figure below. In the case of stacking, the overfitting is because we used the same data set to train all the base estimators.



**Figure 3.14** Stacking can overfit the data. There is clear evidence of overfitting here: the decision boundaries are highly jagged and have small islands, where the classifiers have attempted to fit individual, noisy examples.

To guard against overfitting, we can incorporate *k-fold cross validation* such that each base estimator is not trained on the exact same data set. You may have previously encountered and used cross validation for parameter selection and model evaluation.

Here, we use cross validation to partition the data set into subsets so that different base estimators are trained on different subsets. This often leads to more diversity and robustness, while decreasing the chances of overfitting.

### 3.3.2 Stacking with cross validation

Cross validation is a model validation and evaluation procedure that is commonly employed to simulate out-of-sample testing, tune model hyper-parameters and test the effectiveness of machine-learning models.

The prefix “k-fold” is used to describe the number of subsets we will be partitioning our data set into. For example, in 5-fold cross validation, data is (often randomly) partitioned into 5 non-overlapping subsets. This gives rise to 5 folds, or combinations of these subsets for training and validation, shown below.

More concretely, in 5-fold CV, let’s say the data set  $D$  is partitioned into 5 subsets:  $D_1, D_2, D_3, D_4$  and  $D_5$ . These subsets are disjoint, that is, any example in the data set appears in only one of the subsets. The third fold will comprise of the training set  $trn_3 = \{D_1, D_2, D_4, D_5\}$  (all subsets except  $D_3$ ) and the validation set  $val_3 = \{D_3\}$  (only  $D_3$ ). This fold allows us to train and validate one model. Overall, 5-fold CV will allow us to train and validate five models.

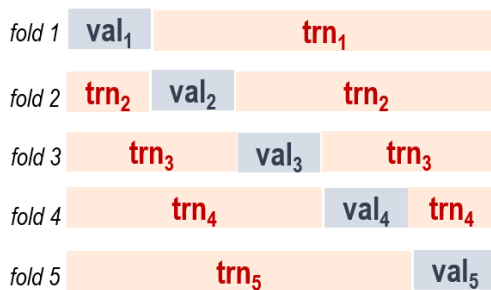
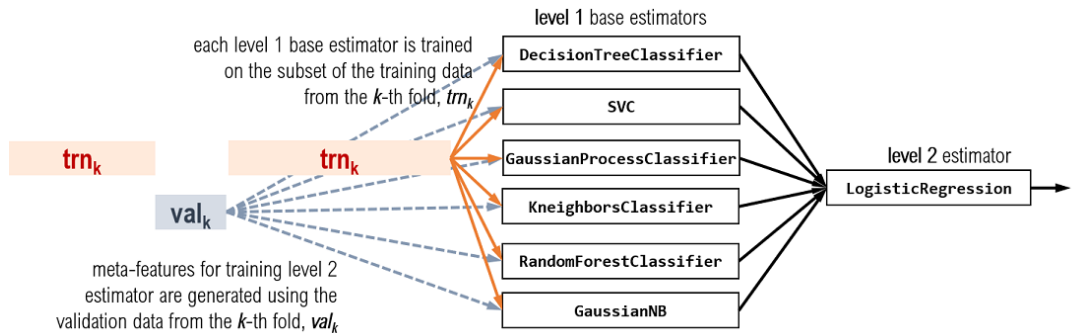


Figure 3.15 k-fold cross validation (here,  $k=5$ ) generates  $k$  different splits of the data set into a training set and a validation set. This simulates out-of-sample validation during training.

In our case, we will use the cross-validation procedure slightly differently, in order to ensure robustness of our level 2 estimator. Instead of using the validation sets  $val_k$  for evaluation, we will use them for generating meta-feature for the level 2 estimator. The precise steps for combining stacking with cross validation are as follows:

1. Randomly split the data into  $k$  equal-sized subsets;
2. Train  $k$  models for each base estimator using the training data from the corresponding  $k$ -th fold,  $trn_k$ ;
3. Generate  $k$  sets of meta-examples from each trained base estimator using the validation data from the corresponding  $k$ -th fold,  $val_k$ ;
4. Retrain each level 1 base estimator on the full data set.

The first 3 steps of this procedure are illustrated below.



**Figure 3.16 Stacking with k-fold cross validation.**  $k$  versions of each level-1 base estimator are trained using the training sets within each fold and  $k$  subsets of meta-examples are generated from the validations sets in each fold for the level-2 estimator.

A key part of stacking with cross validation is to split the data set into training and validation sets for each fold. `scikit-learn` contains many utilities to perform precisely this, and the one we will use is called `model_selection.StratifiedKFold`. The `StratifiedKFold` class is a variation of `model_selection.KFold` class that returns *stratified folds*. This means that the folds preserve the class distributions in the data set when generating folds.

For example, if the ratio of positive examples to negative examples in our data set is 2:1, `StratifiedKFold` will ensure that this ratio is preserved in the folds as well. Finally, it should be noted that rather than creating multiple copies of the data set for each fold (which is very wasteful in terms of storage), `StratifiedKFold` actually returns indices of the data points in the training and validation subsets of each fold.

The listing below demonstrates how to perform stacking with cross-validation.

#### Listing 3.10: Stacking with cross validation

```
from sklearn.model_selection import StratifiedKFold

def fit_stacking_with_CV(level1_estimators, level2_estimator,
                        X, y, n_folds=5, use_probabilities=False):
    n_samples = X.shape[0]
    n_estimators = len(level1_estimators)
    X_meta = np.zeros((n_samples, n_estimators)) #A

    splitter = StratifiedKFold(n_splits=n_folds, shuffle=True)
    for trn, val in splitter.split(X, y): #B

        level1_estimators = fit(level1_estimators, X[trn, :], y[trn])
        X_meta[val, :] = predict_individual(X[val, :],
                                         estimators=level1_estimators,
                                         proba=use_probabilities)

    level2_estimator.fit(X_meta, y) #C
```

```

level1_estimators = fit(level1_estimators, X, y)

final_model = {'level-1': level1_estimators, #D
              'level-2': level2_estimator,
              'use-proba': use_probabilities}

return final_model

```

We can use this function to train a stacking model with cross validation,

```

>>> stacking_model = fit_stacking_with_cv(estimators, meta_estimator,
...                                     Xtrn, ytrn, n_folds=5,
...                                     use_probabilities=True)
>>> ypred = predict_stacking(Xtst, stacking_model)
>>> tst_err = 1 - accuracy_score(ytst, ypred)
>>> tst_err = 1 - accuracy_score(ytst, ypred)
>>> tst_err
0.04424778761061943

```

```

#A initialize meta-data matrix
#B train level 1 estimators and then meta features for the level 2 estimators as individual predictions
#C train level 2 meta-estimator
#D save the level 1 estimators and level 2 estimator in a dictionary

```

With cross validation, stacking obtains a test error of 4.42%. As before, we can visualize our stacked model. We see that the decision boundary is smoother, less jagged and less prone to overfitting overall.

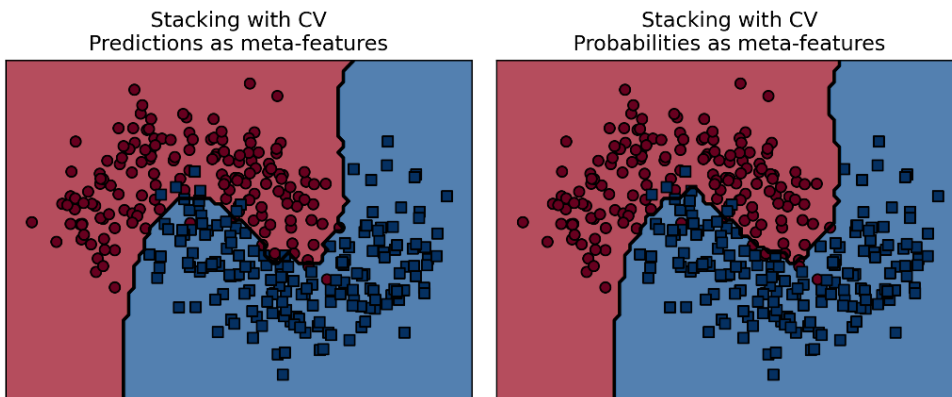


Figure 3.17 Stacking with CV is more robust to overfitting.

**TIP** In our running example scenario, we have 6 base estimators; if we choose to perform stacking with 5-fold CV, we will have to train  $6 \times 5 = 30$  models totally. Each base estimator is trained on  $(k-1)/k$  fraction of the data set. For smaller data sets, the corresponding increase in training time is modest, and is often well worth the cost. For larger data sets, this training time can be significant.



If a full cross-validation-based stacking model is too prohibitively expensive to train, then it is usually sufficient to *hold-out a single validation set*, rather than several cross-validation subsets. This procedure is known as *blending*.

We can now see meta-learning in action on a large-scale, real-world classification task with our next case study: sentiment analysis.

### 3.4 Case Study: Sentiment Analysis

Sentiment analysis is a natural language processing (NLP) task widely used to identify and analyze opinion in text. In its simplest form, it is mainly concerned with identifying the *affect* or the *polarity* of opinion as positive, neutral or negative. Such “voice of the customer” analytics are a key part of brand monitoring, customer service and market research.

This case study explores a *supervised sentiment analysis task for movie reviews*. The data set we will use is the *Large Movie Review Dataset*<sup>1</sup>, which was originally collected and curated from IMDB.com for NLP research by a group at Stanford University<sup>2</sup>. It is a large, *publicly available* data set that has become a text mining/machine learning benchmark over the last few years and has also featured in several Kaggle competitions<sup>3</sup>.

The data set contains 50,000 movie reviews split into training (25k) and test (25k) sets. Each review is also associated with a numerical rating from 1–10. This data set, however, only considers strongly opinionated labels, that is, reviews that are strongly positive about a movie (7–10) or strongly negative about a movie (1-4). These labels are condensed into binary sentiment polarity labels: strongly positive sentiment (Class 1) and strongly negative sentiment (Class 0). Here’s an example of a positive review (label = 1) from the data set:

What a delightful movie. The characters were not only lively but alive, mirroring real every day life and strife within a family. Each character brought a unique personality to the story that the audience could easily associate with someone they know within their own family or circle of close friends.

And an example of a negative review (label = 0):

This is the worst sequel on the face of the world of movies. Once again it doesn't make since. The killer still kills for fun. But this time he is killing people that are making a movie about what happened in the first movie. Which means that it is the stupidest movie ever. Don't watch this. If you value the one precious hour during this movie then don't watch it.

<sup>1</sup> <https://ai.stanford.edu/~amaas/data/sentiment/>

<sup>2</sup> Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. (2011). *Learning Word Vectors for Sentiment Analysis*. ACL 2011.

<sup>3</sup> <https://www.kaggle.com/c/word2vec-nlp-tutorial>

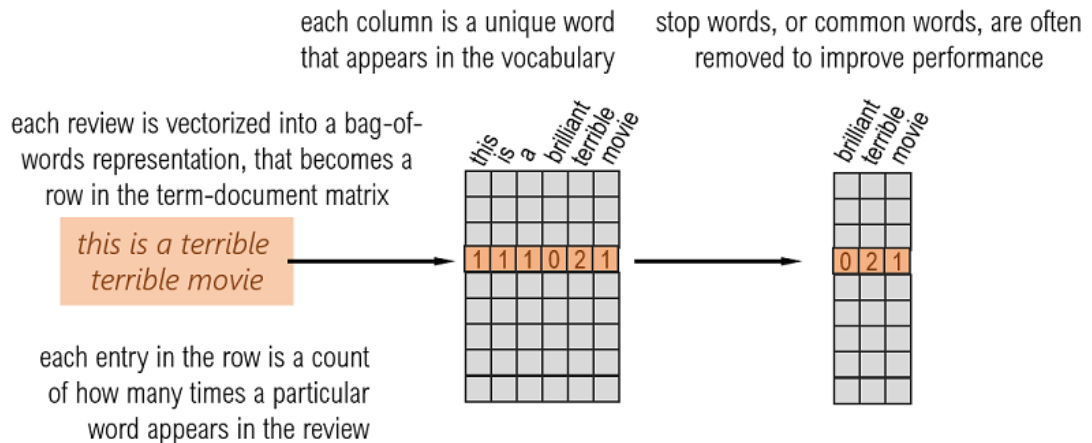
Note the misspelling of “sense” as “since” above. Real-world text data can be highly noisy due to such spelling, grammatical and linguistic idiosyncrasies, which makes these problems very challenging for machine learning. To begin, download and unzip this data set.

### 3.4.1 Pre-processing

The data set is pre-processed to bring each review from an unstructured, free-text form to a structured, vector representation. Put another way, pre-processing aims to bring this corpus (collection) of text files into a *term-document matrix* representation.

This typically involves steps such as removing special symbols, tokenization (chopping it up into tokens, typically individual words), lemmatization (recognizing different usages of the same word, e.g., organize, organizes, organizing), and count-vectorization (counting the words that appear in each document). The last step produces a *bag-of-words* (BoW) representation of the corpus. In our case, each row (example) of the data set will be a review, and each column (feature) will be a word.

The example below illustrates this representation when the sentence “this is a terrible terrible movie” is converted to a bag-of-words representation with the vocabulary consisting of the words {this, is, a, brilliant, terrible, movie}.



**Figure 3.18** Text is converted to a term-document matrix, where each row is an example (corresponding to a single review) and each column is a feature (corresponding to a word in the review). The entries are word counts, making each example a *count vector*. Stop word removal improves representation and often also improves performance.

Since the word “brilliant” does not occur in the review, its count is 0, while most of the other entries are 1 corresponding to the fact that they appear once in the review. This reviewer apparently thought the movie was doubly terrible, this is captured in our count features as the entry for the feature “terrible” is 2.

Fortunately, this data set has already been pre-processed by count vectorization. These pre-processed term-document count features, our data set, can be found in `/train/labeledBow.feats` and `/test/labeledBow.feats`. Both the train and test sets are of size  $25,000 \times 89,527$ . There are, thus, about 90k features, each feature being a word, meaning that the entire set of reviews used about 90k unique words. We pre-process the data further with two additional steps.

### ***Stop-word removal***

This step aims to remove common words such as “the”, “is”, “a”, “an”. Traditionally, stop word removal can reduce the dimensionality of the data, (to make processing faster), and can improve classification performance. This is because words like “the” are often not really informative for information retrieval and text-mining tasks.

**WARNING** Care should be taken with certain stop words such as “not”, as this common word significantly affects the underlying semantics and sentiment. For example, if we don’t account for negation and apply stop-word removal on the sentence “not a good movie”, we get “good movie”, which completely changes the sentiment.

Here, we do not selectively account for such stop words, and rely on the strength of other expressive words such as “awful”, “brilliant” and “mediocre” to capture sentiment. However, performance on your own data set can be improved by careful feature engineering based on an understanding the vocabulary as well as how pruning (or maybe even augmenting) it will affect your task.

The Natural Language ToolKit (`nltk`) is a powerful Python package that provides many tools for NLP. In the code listing below, we use `nltk`’s standard stop word removal tool. The entire vocabulary for the IMDB data set is available in the file `imdb.vocab`, sorted by their frequency, from most common to least common.

We can directly apply stop word removal on this set of features to identify which words we will keep. In addition, we only keep the 5000 most common words in order for our running time to be more manageable.

#### **Listing 3.11. Drop stop words from the vocabulary**

```
import nltk
import numpy as np

def prune_vocabulary(data_path, max_features=5000):
    with open('{}/imdb.vocab'.format(data_path), 'r', encoding='utf8') \
        as vocab_file:
        vocabulary = vocab_file.read().splitlines() #A

    nltk.download('stopwords')

    stopwords = set(nltk.corpus.stopwords.words("english")) #B
```

```

to_keep = [True if word not in stopwords #C
           else False for word in vocabulary]
feature_ind = np.where(to_keep)[0]

return feature_ind[:max_features] #D

```

```

#A load the vocabulary file
#B convert the list of stop words to a set for faster processing
#C remove stop words from the vocabulary
#D keep the top 5000 words

```

### ***Tf-Idf Transformation***

Our second pre-processing step converts the count features to *tf-idf features*. These features represent the *term frequency-inverse document frequency*, a statistic that weights each feature in a document (in our case, a single review) relative to how often it appears in that document as well as how often it appears in the entire corpus (in our case, all the reviews).

Intuitively, tf-idf weights words by how often they appear in a document, but also adjusts for how often they appear overall. This is to ensure that some words are generally used more often than others. We can use `scikit-learn`'s pre-processing toolbox to convert our count features to tf-idf features using the `TfidfTransformer`.

#### **Listing 3.12. Extract tf-idf features and save the data set**

```

import h5py
from sklearn.datasets import load_svmlight_files
from scipy.sparse import csr_matrix as sp
from sklearn.feature_extraction.text import TfidfTransformer

def preprocess_and_save(data_path, feature_ind):
    data_files = ['{0}/{1}/labeledBow.fea'.format(data_path, data_set)
                  for data_set in ['train', 'test']] #A
    [Xtrn, ytrn, Xtst, ytst] = load_svmlight_files(data_files)
    n_features = len(feature_ind)

    ytrn[ytrn <= 5], ytst[ytst <= 5] = 0, 0 #B
    ytrn[ytrn > 5], ytst[ytst > 5] = 1, 1

    tfidf = TfidfTransformer()
    Xtrn = tfidf.fit_transform(Xtrn[:, feature_ind]) #C
    Xtst = tfidf.transform(Xtst[:, feature_ind])

    with h5py.File('{0}/imdb-{1}k.h5'.format(data_path,
                                           round(n_features/1000)), 'w') as db:
        db.create_dataset('Xtrn',
                          data=sp.todense(Xtrn), compression='gzip')
        db.create_dataset('ytrn', data=ytrn, compression='gzip')
        db.create_dataset('Xtst',
                          data=sp.todense(Xtst), compression='gzip')
        db.create_dataset('ytst', data=ytst, compression='gzip') #D

```

```

#A load train and test data
#B convert sentiments to binary labels
#C convert count features to tf-idf features

```

**#D** save the pre-processed data sets in the HDF5 binary data format

The code listing above creates and saves training and test sets, each of which is of size 25,000 reviews  $\times$  5,000 tf-idf features.

### 3.4.2 Dimensionality Reduction

We continue to process the data with dimensionality reduction, which aims to represent the data more compactly. The main purpose of applying dimensionality reduction is to avoid the “curse of dimensionality”, where algorithm performance deteriorates as the dimensionality of the data increases.

We adopt the popular dimensionality reduction approach of *principal components analysis* (PCA), which aims to compress and embed the data into a lower-dimensional feature space in a manner that preserves as much of the variance as possible. This ensures that we are able to extract a lower-dimensional representation without too much loss of information.

This data set contains thousands of examples as well as features, which means that applying PCA on the entire data set will likely be highly computationally intensive and very slow. To avoid loading the entire data set into memory and to process the data more efficiently, we perform Incremental PCA instead.

Incremental PCA breaks the data set down into chunks, which can be easily loaded into memory. It should be noted that, while this chunking reduces the number of samples (rows) loaded into memory substantially, for each row it still loads all the features (columns).

`scikit-learn` provides the class `sklearn.decomposition.IncrementalPCA`, which is far more memory efficient. The listing below performs PCA to reduce the dimension of the data to 500 dimensions.

#### Listing 3.13. Perform dimensionality reduction using Incremental PCA

```
from sklearn.decomposition import IncrementalPCA

def transform_sentiment_data(data_path, n_features=5000, n_components=500):
    db = h5py.File('{0}/imdb-{1}k.h5'.format( #A
        data_path, round(n_features/1000)), 'r')

    pca = IncrementalPCA(n_components=n_components)
    chunk_size = 1000
    n_samples = db['Xtrn'].shape[0] #B
    for i in range(0, n_samples // chunk_size):
        pca.partial_fit(db['Xtrn'][i*chunk_size:(i+1) * chunk_size])

    Xtrn = pca.transform(db['Xtrn']) #C
    Xtst = pca.transform(db['Xtst'])

    with h5py.File('{0}/imdb-{1}k-pca{2}.h5'.format(data_path,
        round(n_features/1000), n_components), 'w') as db2:
        db2.create_dataset('Xtrn', data=Xtrn, compression='gzip')
        db2.create_dataset('ytrn', data=db['ytrn'], compression='gzip')
        db2.create_dataset('Xtst', data=Xtst, compression='gzip')
```

```
db2.create_dataset('ytst', data=db['ytst'], compression='gzip') #D
```

```
#A load pre-processed train and test data
#B apply Incremental PCA to the data in manageable chunks
#C reduce the dimension of both the train and
#D save the pre-processed data sets in the HDF5 binary data format
```

Note that `IncrementalPCA` is fit using *only the training set*. Recall that the test data must *always* be held-out and should only be used to provide an accurate estimate of how our pipeline will generalize to future, unseen data.

This means that we cannot use the test data during any part of pre-processing or training and can only use it for evaluation.

### 3.4.3 Stacking classifiers

Our goal now is to train a heterogeneous ensemble with meta-learning. Specifically, we will use ensemble several base estimators by *blending* them. Recall that blending is a variant of stacking, where, instead of using cross validation, we use a *single validation set*.

First, we load the data using the function below:

```
def load_sentiment_data(data_path, n_features=5000, n_components=1000):
    with h5py.File('{}/imdb-{}k-pca{}.h5'.format(data_path,
        round(n_features/1000), n_components), 'r') as db:
        Xtrn = np.array(db.get('Xtrn'))
        ytrn = np.array(db.get('ytrn'))
        Xtst = np.array(db.get('Xtst'))
        ytst = np.array(db.get('ytst'))

    return Xtrn, ytrn, Xtst, ytst
```

Next, we use five base estimators: `RandomForestClassifier` with 100 randomized decision trees, `ExtraTreesClassifier` with 100 extremely randomized trees, `Logistic Regression`, `Bernoulli naïve Bayes` (`BernoulliNB`) and a linear SVM trained with stochastic gradient descent (`SGDClassifier`).

```
from sklearn.ensemble import RandomForestClassifier, ExtraTreesClassifier
from sklearn.linear_model import LogisticRegression, SGDClassifier
from sklearn.naive_bayes import BernoulliNB

estimators = [('rf', RandomForestClassifier(n_estimators=100, n_jobs=-1)),
              ('xt', ExtraTreesClassifier(n_estimators=100, n_jobs=-1)),
              ('lr', LogisticRegression(C=0.01, solver='lbfgs')),
              ('bnb', BernoulliNB()),
              ('svm', SGDClassifier(loss='hinge', penalty='l2', alpha=0.01,
                                   n_jobs=-1, max_iter=10, tol=None))]
```

The Bernoulli naïve Bayes classifier learns linear models but is especially effective for count-based data arising from text-mining tasks such as ours. Logistic regression and SVM with `SGDClassifier` both learn linear models. Random forests and extra trees are two homogeneous ensembles which produce highly nonlinear classifiers using decision trees as

base estimators. This is a diverse set of base estimators, containing a good mix of linear and nonlinear classifiers.

To blend these base-estimators into a heterogeneous ensemble with meta-learning, we use the following procedure:

1. Split the training data into a training set  $(X_{trn}, y_{trn})$  with 80% of the data and a validation set  $(X_{val}, y_{val})$ , with the remaining 20% of the data
2. Train each of the level-1 estimators on the training set,  $(X_{trn}, y_{trn})$
3. Generate meta-features  $X_{meta}$  with the trained estimators using  $X_{val}$ ;
4. Augment the validation data with the meta-features:  $[X_{val}, X_{meta}]$ ; this augmented validation set will have 500 original features + 5 meta-features
5. Train the level-2 estimator with the augmented validation set  $([X_{val}, X_{meta}], y_{val})$

The key to our combining by meta-learning procedure is meta-feature augmentation: we augment the validation set with the meta-features produced by the base estimators.

This leaves one final decision: the choice of the level-2 estimator. Previously, we used simple linear classifiers. For this classification task, we utilize a *neural network*.

### Neural networks and deep learning

Neural networks are one of the oldest machine-learning algorithms. There has been a significant resurgence of interest in neural networks, especially deep neural networks, owing to their widespread success in many applications.

For a quick refresher on Neural Networks and Deep Learning, see Chapter 2 of *Probabilistic Deep Learning with Python, Keras and TensorFlow Probability* by Oliver Dürr, Beate Sick and Elvis Murina (Manning Early Access Program, 2019).

We will use a shallow neural network as our level-2 estimator. This will produce in a highly nonlinear meta-estimator that can combine the predictions of the level-1 classifiers.

```
from sklearn.neural_network import MLPClassifier
meta_estimator = MLPClassifier(hidden_layer_sizes=(128, 64, 32),
                               alpha=0.001)
```

The listing below implements our strategy:

#### Listing 3.14. Blending models with a validation set

```
from sklearn.model_selection import train_test_split

def blend_models(level1_estimators, level2_estimator,
                 X, y, use_probabilities=False):
    Xtrn, Xval, ytrn, yval = train_test_split(X, y, test_size=0.2) #A

    n_estimators = len(level1_estimators)
    n_samples = len(yval)
    Xmeta = np.zeros((n_samples, n_estimators))
    for i, (model, estimator) in enumerate(level1_estimators): #B
        estimator.fit(Xtrn, ytrn)
```

```

    Xmeta[:, i] = estimator.predict(Xval)

Xmeta = np.hstack([Xval, Xmeta]) #C

level2_estimator.fit(Xmeta, yval) #D

final_model = {'level-1': level1_estimators,
               'level-2': level2_estimator,
               'use-proba': use_probabilities}

return final_model

```

**#A** split into training and validation sets  
**#B** initialize and fit the base estimators on the training data  
**#C** augment the validation set with the newly generated meta-features  
**#D** fit the level-2 meta-estimator

We can now fit a heterogeneous ensemble on the training data and then evaluate it on both the training and test data to compute the training and test error:

```

>>> stacked_model = blend_models(estimators, meta_estimator, Xtrn, ytrn)

>>> ypred = predict_stacking(Xtrn, stacked_model)
>>> trn_err = (1 - accuracy_score(ytrn, ypred)) * 100
7.8359999999999985

>>> ypred = predict_stacking(Xtst, stacked_model)
>>> tst_err = (1 - accuracy_score(ytst, ypred)) * 100
17.196

```

Our model achieves a training error of 8.47% and a test error of 16.91%. So how well did we actually do? Did our ensembling procedure help at all? To answer these questions, we compare the performance of the ensemble to the performance of each base estimator in the ensemble.

The figure below shows the training and test errors of the individual base estimators as well as the stacking/blending ensemble. Some individual classifiers achieve a training error of 0%, which means they are likely overfitting the training data. This affects their performance as evidenced by the test error.

Overall, stacking/blending these heterogeneous models produces a test error of 17.2%, which is better than all the other models. In particular, let's compare this result to logistic regression, which achieves a test error of 18%. Recall that the test set contains 25,000 examples, which means that our stacked model classifies (approximately) another 200 examples correctly!

On the whole, the performance of the heterogeneous ensemble is better than a lot of the base estimators that contribute to it. This is an example of how heterogeneous ensembles can improve upon the overall performance of the underlying individual base estimators.

**TIP** Remember that any linear or nonlinear classifier can be used as a meta-estimator. Common choices include decision trees, kernel support vector machines, and even other ensembles!



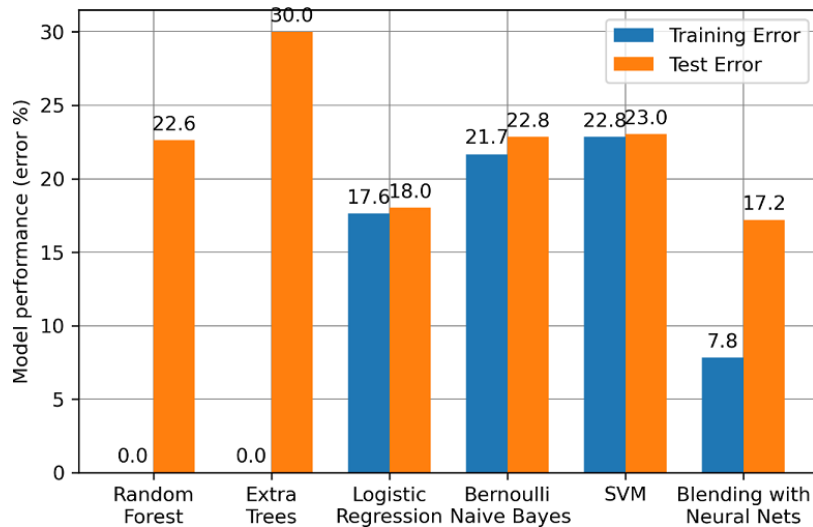


Figure 3.19 Comparing the performance of each individual base classifier with the meta-classifier ensemble. Stacking/blending improves classification performance by ensembling diverse base classifiers.

### 3.5 Summary

In this chapter, we explored the learning of parallel, heterogeneous ensembles. These ensemble methods promote ensemble diversity through *heterogeneity*, that is, they use different base learning algorithms to train the base estimators. Since different algorithms view the training data differently, and attempt to induce models in substantially different ways, the resulting models will often be naturally diverse.

1. *Weighting methods* assign individual base estimator predictions a weight that corresponds to their performance; better base estimators are assigned higher weights and influence the overall final prediction more.
2. Weighting methods use a predefined combination function to combine the weighted predictions of the individual base estimators. Linear combination functions (such as weighted sum) are often effective and easy to interpret. Nonlinear combination functions can be also be used, though the added complexity may lead to overfitting.
3. *Meta-learning methods* learn a combination function from the data, in contrast to weighting methods, where we have to make one up ourselves.
4. Meta-learning methods create multiple layers of estimators. The most common meta-learning method is *stacking*, so called because it literally stacks learning algorithms in a pyramid-like learning scheme.
5. Simple stacking creates two levels of estimators. The base estimators are trained in the first level, and their outputs are used to train a second-level estimator called a *meta-estimator*. More complex stacking models with many more levels of estimators are possible.

6. Stacking can often overfit, especially in the presence of noisy data. To avoid overfitting, stacking is combined with *cross validation* to ensure that different base estimators see different subsets of data set for increased ensemble diversity.
7. Stacking with cross validation, though it reduces overfitting, can also be computationally intensive, leading to long training times. To speed up training, while guarding against overfitting, a single validation set can be used. This procedure is known as *blending*.
8. Any machine-learning algorithm can be used as a meta-estimator in stacking. Logistic regression is most common and leads to linear models. Nonlinear models, obviously, have greater representative power, but also are at greater risk for overfitting.
9. Both weighting and meta-learning approaches can use either the base estimator predictions directly, or the prediction probabilities. The latter typically leads to a smoother, more nuanced model.

Beyond classification tasks, it is possible to construct heterogeneous ensembles for regression and unsupervised learning tasks such as clustering.

# 4

## *Sequential Ensembles: Boosting*

### **This chapter covers**

- Training sequential ensembles of weak learners
- Implementing and understanding how AdaBoost works
- Using AdaBoost in practice
- Implementing and understanding how LogitBoost works

The ensembling strategies we have seen thus far have been parallel ensembles. These include homogeneous ensembles such as bagging and random forests (where the same base learning algorithm is used to train base estimators) and heterogeneous ensemble methods such as stacking (where different base learning algorithms are used to train base estimators).

Now, we will explore a new family of ensemble methods: sequential ensembles. Unlike parallel ensembles, which exploit the *independence* of each base estimator, sequential ensembles exploit the dependence of base estimators.

More specifically, during learning, sequential ensembles train a new base estimator in such a manner that it minimizes mistakes made by the base estimator trained in the previous step.

The first sequential ensemble method we will investigate is *boosting*. Boosting aims to combine *weak learners*, or “simple” base estimators. Put another way, boosting literally aims to boost the performance of a collection of weak learners.

This is in contrast to algorithms like bagging, which combine “complex” base estimators, also known as *strong learners*. Boosting commonly refers to AdaBoost, or *adaptive boosting*. This

approach was introduced by Freund and Schapire in 1995<sup>1</sup>, for which they eventually won the prestigious Gödel Prize for outstanding papers in theoretical computer science.

Since 1995, boosting has emerged as a core machine-learning method. Boosting is surprisingly simple to implement, computationally efficient and can be used with a wide variety of base-learning algorithms. Prior to the re-emergence of deep learning in the mid-2010s, boosting was widely applied to computer vision tasks such as object classification and natural language processing tasks such as text filtering.

For most of this chapter, we will focus on AdaBoost, a popular boosting algorithm that is also quite illustrative of the general framework of sequential ensemble methods. Other boosting algorithms can be derived by changing aspects of this framework, such as the loss function. Such variants are usually not available in packages and must be implemented. We also implement one such variant: LogitBoost.

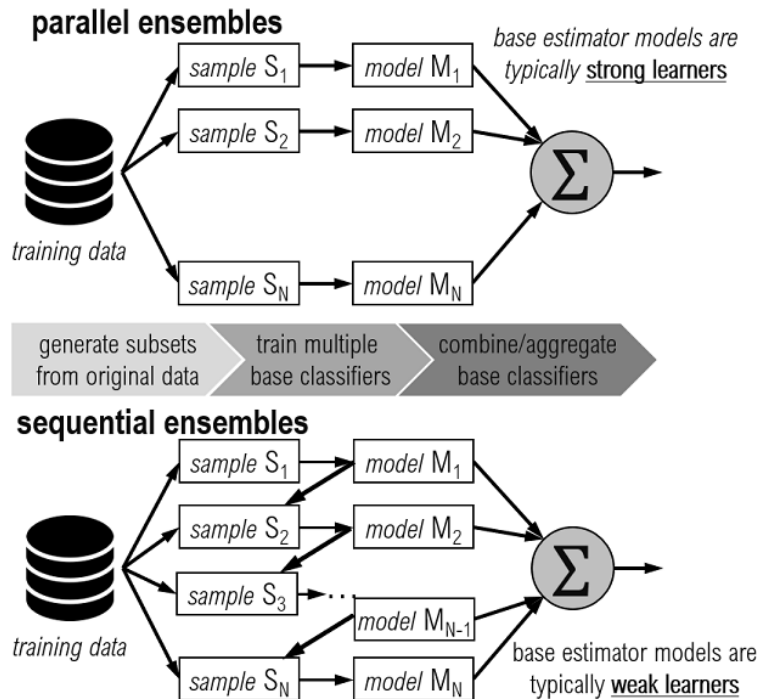


Figure 4.1 Differences between parallel and sequential ensembles: (1) base estimators in parallel ensembles are trained independently of each other, while in sequential ensembles they are trained to improve upon the

<sup>1</sup> Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119-139, 1997.

predictions of the previous base estimator; (2) sequential ensembles typically use weak learners as base estimators.

## 4.1 Sequential Ensembles of Weak Learners

There are two key differences between parallel and sequential ensembles:

- Base estimators in parallel ensembles can usually be trained independently, while in sequential ensembles, the base estimator in the current iteration depends on the base estimator in the previous iteration. This is shown in Figure 4.1, where (in iteration  $t$ ) the behavior of base estimator  $M_{t-1}$  influences the sample  $S_t$  and the next model  $M_t$ .
- Base estimators in parallel ensembles are typically strong learners, while in sequential ensembles they are typically weak learners. Sequential ensembles aim to combine several weak learners into one strong learner.

Intuitively, we can think of strong learners as professionals: highly confident people who are independent and sure about their answers. Weak learners, on the other hand, are like amateurs: not so confident and unsure about their answers. How can we get a bunch of not-so-confident “amateurs” to come together? By “boosting” them, of course. Before we see how exactly, let’s characterize what weak learners are.

### *Weak Learners*

While the precise definition of the strength of learners is rooted in machine-learning theory, for our purposes, a strong learner is a “good” model (or estimator).

In contrast, a weak learner is a very simple model that doesn’t perform that well. The only requirement of a weak learner (for binary classification) is that it perform better than random guessing. Or put another way, its accuracy should be slightly better than 50%. Decision trees are often used as base estimators for sequential ensembles. Boosting algorithms typically use decision stumps, or decision trees of depth 1 (see below).

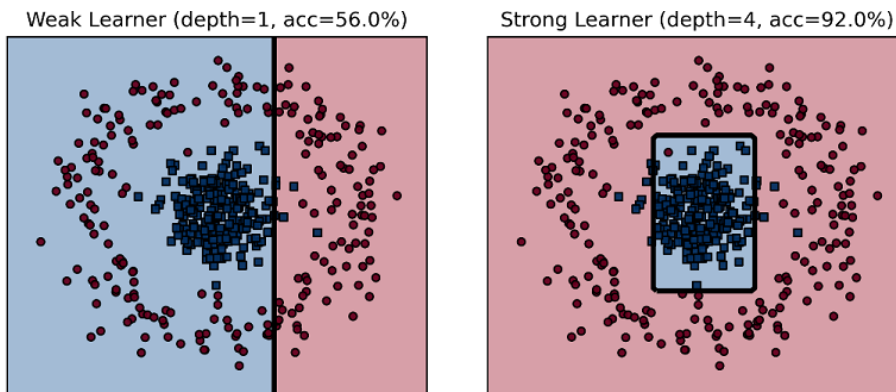


Figure 4.2 Decision stumps (trees of depth 1, left) are commonly used as weak learners in sequential ensemble

methods such as boosting. As tree depth increases, it becomes a stronger classifier, and its performance improves. However, it is not possible to arbitrarily increase the strength of classifiers as they will begin to overfit during training, which decreases their prediction performance when deployed.

Sequential ensemble methods such as boosting aim to combine several weak learners into a single strong learner. These methods literally “boost” weak learners into a strong learner.

**TIP** A weak learner is a simple classifier that is easy and efficient to train. Sequential ensemble methods are generally agnostic to the underlying base learning algorithms, meaning that you can use any classification algorithm as a weak learner. In practice, weak learners such as shallow decision trees and shallow neural networks are common.

Recall Dr. Randy Forrest’s ensemble of interns from Chapter 2. In a parallel ensemble of knowledgeable medical personnel, each intern can be considered a strong learner. To understand how different the philosophy of sequential ensembles is, we turn to Freund and Schapire, who describe boosting as a “a committee of blockheads that can somehow arrive at highly reasoned decisions”.

This would be akin to the situation where Dr. Randy Forrest sent away his interns and decided to crowd-source medical diagnoses instead. While this is certainly a far-fetched (and unreliable) strategy for diagnosing a patient, it turns out that “garnering wisdom from a council of fools” works surprisingly well in machine learning. This is the underlying motivation for sequential ensembles of weak learners.

## 4.2 AdaBoost: ADaptive BOOSTing

In this section, we begin with an important sequential ensemble: AdaBoost. AdaBoost is simple to implement code and computationally efficient to use. As long as the performance of each weak learner in AdaBoost is slightly better than random guessing, the final model converges to a strong learner.

However, beyond applications, understanding how AdaBoost works is also key to understanding two state-of-the-art sequential ensemble methods we will look at next: gradient boosting and Newton boosting.

### A BRIEF HISTORY OF BOOSTING

The origins of boosting lie in computational learning theory, when learning theorists Leslie Valiant and Michael Kearns posed the following question in 1988: can one can “boost” a weak learner to a strong learner? This question was answered affirmatively two years later by Rob Schapire, in his now landmark paper called “The Strength of Weak Learnability”.

The earliest boosting algorithms were limited because weak learners did not adapt to fix the mistakes made by weak learners trained in previous iterations. Freund and Schapire’s AdaBoost, or *adaptive boosting* algorithm, proposed in 1994, ultimately addressed these limitations. Their original algorithm endures to this day and has been widely applied in several application domains including text mining, computer vision and medical informatics.

### 4.2.1 Intuition: Learning with Weighted Examples

AdaBoost is an adaptive algorithm: at every iteration, it trains a new base estimator that *fixes the mistakes made by the previous base estimator*. Thus, it needs some way to ensure that the base learning algorithm prioritizes misclassified training examples. AdaBoost does this by maintaining *weights over individual training examples*.

Intuitively, weights reflect the relative importance of training examples. Misclassified examples have higher weights, while correctly classified examples have lower weights.

When we train the next base estimator sequentially, the weights will allow the learning algorithm to prioritize (and hopefully fix) mistakes from the previous iteration. This is the “adaptive” component of AdaBoost, which ultimately leads to a powerful ensemble.

**CAUTION** When implementing ensemble methods that use weights on training examples, care must be taken to ensure that the base learning algorithm can actually utilize these weights. Most weighted classification algorithms utilize modified loss functions in order to prioritize correct classification of examples with higher weights.

Let’s visualize the first few iterations of boosting. Each iteration performs the same steps:

1. train a weak learner (here, a decision stump) that learns a model to ensure training examples with higher weights are prioritized;
2. update the weights of the training examples such that misclassified examples are assigned higher weights; the worse the error, the higher the weight.

Initially (iteration  $t - 1$ ), all examples are initialized with *equal weights*. The decision stump trained in iteration 1 is a simple, axis-parallel classifier with an error rate of 15%.

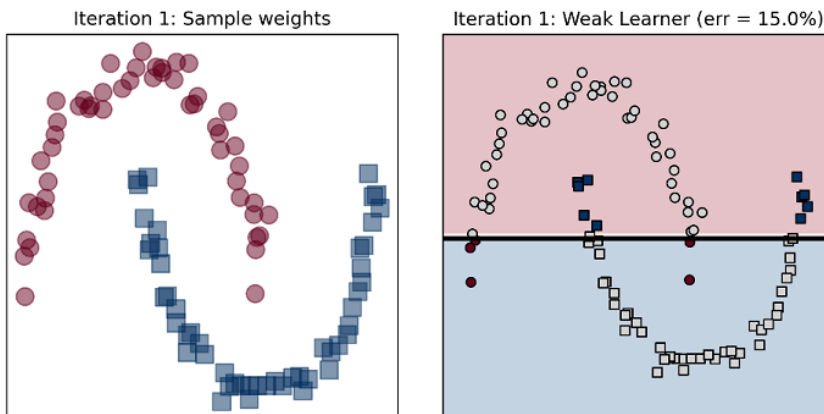
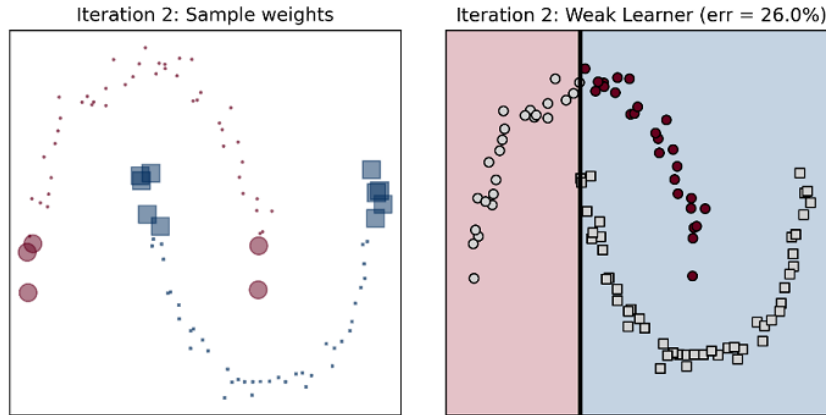


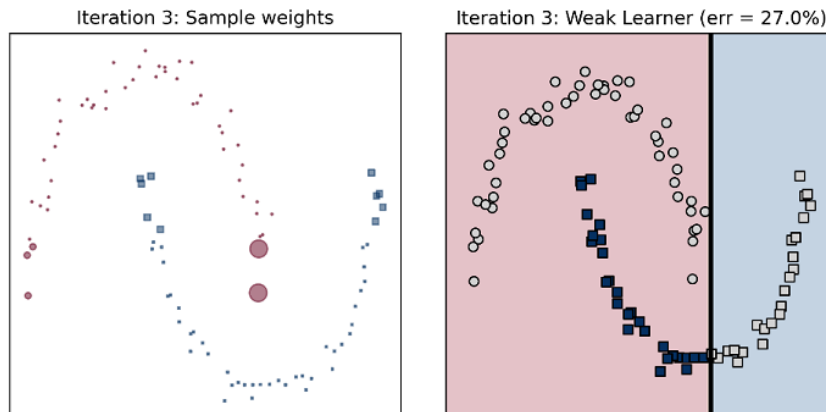
Figure 4.3 Initially (iteration 1), all the training examples are weighted equally (and hence plotted with the same size on the left). The decision stump learned on this data set is shown on the right. The correctly classified examples are colored gray, while the misclassified examples are darker.

The next decision stump (in iteration 2) to be trained must correctly classify the examples misclassified by the previous decision stump (in iteration 1). Thus, mistakes are weighted higher, which enables the decision tree algorithm to prioritize them during learning.



**Figure 4.4** At the start of iteration 2, training examples misclassified in iteration 1 (darker points in Figure 4.3, right) are assigned higher weights. This is visualized on the left, where each example's size is proportional to its weight. Since weighted examples have higher priority, the new decision stump in the sequence (right) ensures that these are now correctly classified.

The decision stump trained in the second iteration does indeed correctly classify the training examples with higher weights, though it makes mistakes of its own. In the third iteration, a third decision stump can be trained that aims to rectify these mistakes.



**Figure 4.5** At the start of iteration 3, training examples misclassified in iteration 2 (darker points in Figure 4.4, right) are assigned higher weights. Note that misclassified points also have different weights. The new decision stump in the sequence trained in this iteration (right) ensures that these are now correctly classified.



After three iterations, we can combine the three individual weak learners into a strong learner, shown below. Some useful points to note:

- Observe the weak estimators learned in the three iterations. They are all different from each other and classify the problem in diversely different ways. Recall that at each iteration, base estimators are trained on the same training set *but with different weights*. Reweighting allows AdaBoost to train a different base estimator at each iteration, one that is often different from an estimator trained at the previous iterations.

Thus, *adaptive reweighting*, or updating adaptively, promotes ensemble diversity.

- The resulting ensemble of weak (and linear) decision stumps is stronger (and nonlinear). More precisely, each base estimator had training error rates of 15%, 26% and 27%, while their ensemble has an error rate of 7%.

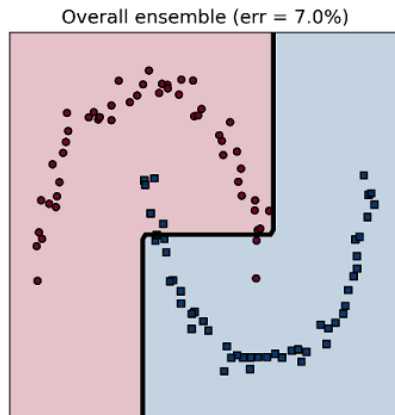


Figure 4.6 The three weak decision stumps from above can be boosted into a stronger ensemble.

The algorithm is called “boosting” as it boosts the performance of weak learners into a more powerful and complex ensemble, a strong learner.

## 4.2.2 Implementing AdaBoost

First, we will implement our own version of AdaBoost. As we do so, we’ll keep the following key properties of AdaBoost in mind:

1. AdaBoost uses decision stumps as base estimators, which can be trained extremely quickly, even with a large number of features. Decision stumps are *weak learners*. Contrast this to bagging, which uses deeper decision trees, which are strong learners.
2. AdaBoost keeps track of *weights on individual training examples*. This allows AdaBoost to ensure ensemble diversity by *reweighting* training examples. We saw how reweighting helped AdaBoost learn different base estimators in the example above. Contrast this to bagging and random forests, which use resampling.

3. AdaBoost keeps track of *weights on individual base estimators*. This is similar to combination methods, which weight each classifier differently

AdaBoost is fairly straightforward to implement. The basic algorithmic outline at the  $t$ -th iteration can be described by the following steps:

1. Train a weak learner  $h_t(x)$  using the weighted training examples,  $(x_i, y_i, D_i)$
2. Compute the training error  $\epsilon_t$  of the weak learner  $h_t(x)$
3. Compute the weight of the weak learner  $\alpha_t$  that depends on  $\epsilon_t$
4. Update the weights of the training examples
5. Increase the weight of misclassified examples to  $D_i e^{\alpha_t}$
6. Decrease the weight of correctly classified examples to  $D_i / e^{\alpha_t}$

At the end of  $T$  iterations, we have weak learners  $h_t$  along with the corresponding weak learner weight  $\alpha_t$ . The overall classifier after  $t$  iterations is just a weighted ensemble:

$$H(x) = \sum_{t=1}^T \alpha_t h_t(x)$$

This form is a *weighted linear combination of base estimators*, similar to the linear combinations used by homogeneous ensembles we've seen previously, such as combination methods or stacking. The main difference from those methods is that the base estimators used by AdaBoost are weak learners.

Two key questions we now need to answer are: (1) how do we update the weights on the training examples,  $D_i$ ; and (2) how do we compute the weight of each base estimator,  $\alpha_t$ ?

AdaBoost uses the same intuition as the combination methods we have seen previously in Chapter 3, with combination methods. Recall that weights are computed to reflect base estimator *performance*: base estimators with better performance (say, accuracy) should have higher weights than those with worse performance.

### ***Weak Learner Weights ( $\alpha_t$ )***

At each iteration  $t$ , we obtain a base estimator  $h_t(x)$ . The training error  $\epsilon_t$  of  $h_t(x)$  is a simple and immediate measure of its performance. AdaBoost computes its weight as

$$\alpha_t = \frac{1}{2} \log \left( \frac{1 - \epsilon_t}{\epsilon_t} \right)$$

Why this particular formulation? Let's look at the relationship between  $\alpha_t$  and the error  $\epsilon_t$ , by visualizing how  $\alpha_t$  changes with increasing error  $\epsilon_t$  (Figure 4.7). Recall our intuition: better performing base estimators (those with lower errors) must be weighted higher so that their contribution to the ensemble prediction is higher.

Conversely, the weakest learners are the ones which perform the worst. Sometimes, they are barely better than random guessing. Put another way, in a binary classification problem, they would only be slightly better than flipping a coin to decide the answer.

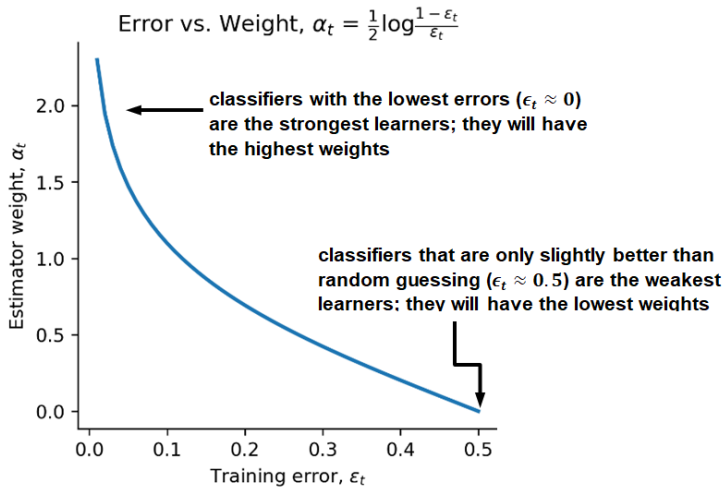


Figure 4.7 AdaBoost assigns stronger learners (which have lower training errors) higher weights, and weaker learners (which have higher training errors) lower weights.

Concretely, the weakest learners have error rates only slightly better than 0.5 (or 50%). These weakest learners have the lowest weights,  $\alpha_t \approx 0$ . The strongest learners achieve a training errors close to 0.0 (or 0%). These learners have the highest weights.

### Training Example Weights ( $D_t$ )

The base estimator weight ( $\alpha_t$ ) can also be used to update the weights of each training example. AdaBoost updates example weights as

$$D_i^{t+1} = D_i^t \cdot \begin{cases} e^{\alpha_t}, & \text{if misclassified,} \\ e^{-\alpha_t}, & \text{if correctly classified} \end{cases}$$

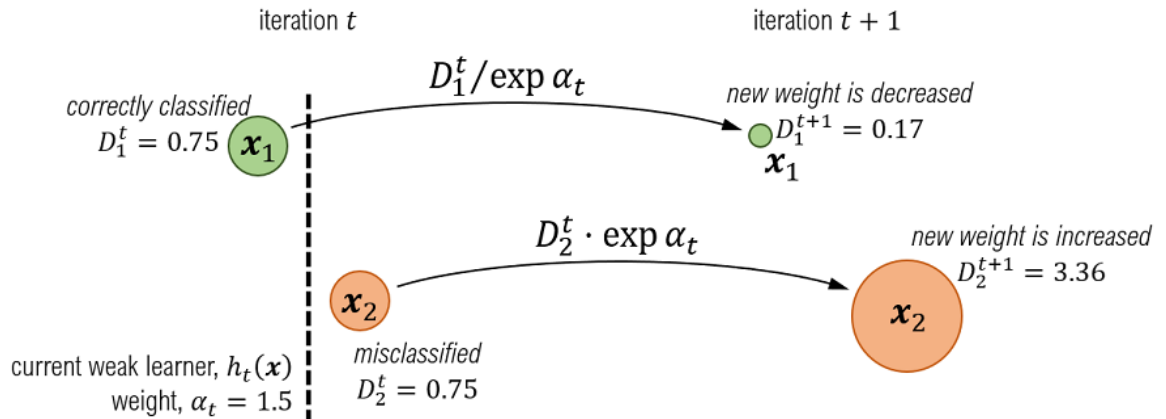
When examples are correctly classified, the new weight is decreased by  $e^{\alpha_t}$ :  $D_i^{t+1} = D_i^t / e^{\alpha_t} = D_i^t e^{-\alpha_t}$ . Stronger base estimators will decrease the weight more because they are more confident in their correct classification.

Similarly, when examples are misclassified, the new weight is increased by  $e^{\alpha_t}$ :  $D_i^{t+1} = D_i^t \cdot e^{\alpha_t}$ . In this manner, AdaBoost ensures that misclassified training examples receive higher weights, which then be fixed in the next iteration,  $t+1$ .

For example, let's say we have two training examples  $x_1$  and  $x_2$ , both with weights  $D_1=D_2=0.75$ . The current weak learner  $h_t$  has weight  $\alpha_t = 1.5$ . Let's say  $x_1$  is correctly classified by  $h_t$ ;

hence its weight should decrease by a factor of  $e^{\alpha_t}$ . The new weight will be  $D_1^{new} = D_1/e^{\alpha_t} = 0.75/e^{1.5} = 0.1673$ .

Conversely,  $x_1$  is misclassified by  $h_t$ ; hence its weight should increase by a factor of  $e^{\alpha_t}$ . The new weight will be  $D_2^{new} = D_2 \cdot e^{\alpha_t} = 0.75 \cdot e^{1.5} = 3.3613$ . This is illustrated in the figure below.



**Figure 4.8.** In iteration  $t$ , two training examples  $x_1$  and  $x_2$  have the same weights.  $x_1$  is correctly classified, while  $x_2$  is misclassified by the current base estimator  $h_t(x)$ . As the goal in the next iteration is to learn a classifier  $h_{t+1}(x)$  that fixes the mistakes of  $h_t(x)$ , AdaBoost increases the weight of the misclassified example  $x_2$ , while decreasing the weight of the correctly classified example  $x_1$ . This allows the base learning algorithm to prioritize  $x_2$  during training in iteration  $t+1$ .

### Training With AdaBoost

The AdaBoost algorithm is easy to implement. The listing below shows training for boosting.

#### Listing 4.1 Training an ensemble of weak learners using AdaBoost

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

def fit_boosting(X, y, n_estimators=10):
    n_samples, n_features = X.shape
    D = np.ones((n_samples, )) #A
    estimators = [] #B

    for t in range(n_estimators):
        D = D / np.sum(D) #C

        h = DecisionTreeClassifier(max_depth=1)
        h.fit(X, y, sample_weight=D) #D

        ypred = h.predict(X)
        e = 1 - accuracy_score(y, ypred, sample_weight=D) #D
        a = 0.5 * np.log((1 - e) / e)
```

```

    m = (y == ypred) * 1 + (y != ypred) * -1    #E
    D *= np.exp(-a * m)

    estimators.append((a, h)) #F

return estimators

```

**#A** non-negative weights, initialized to 1  
**#B** initialize an empty ensemble  
**#C** normalize the weights so they sum to 1  
**#D** train weak learner ( $h_t$ ) with weighted examples  
**#E** compute the training error ( $\epsilon_t$ ) and the weight ( $\alpha_t$ ) of the weak learner  
**#F** update the example weights: increase for misclassified examples, decrease for correctly classified examples  
**#G** save the weak learner and its weight

Once we have a trained `ensemble`, we can use it to make predictions. The listing below shows how to predict new test examples using the boosted ensemble. Observe that this is identical to making predictions with other weighted ensembles methods such as stacking.

#### Listing 4.2 Making predictions with AdaBoost

```

def predict_boosting(X, estimators):
    pred = np.zeros((X.shape[0], )) #A

    for a, h in estimators:
        pred += a * h.predict(X) #B

    y = np.sign(pred) #C

    return y

```

We can use these functions to fit and predict on a data set.

```

>>> from sklearn.datasets import make_moons
>>> X, y = make_moons(n_samples=200, noise=0.1)
>>> y = (2 * y) - 1 #D

>>> from sklearn.model_selection import train_test_split
>>> Xtrn, Xtst, ytrn, ytst = train_test_split(X, y, test_size=0.25)
>>> estimators = fit_boosting(Xtrn, ytrn)
>>> ypred = predict_boosting(Xtst, estimators)

>>> from sklearn.metrics import accuracy_score
>>> tst_err = 1 - accuracy_score(ytst, ypred)
>>> tst_err
0.0400000000000000036

```

**#A** initialize all the predictions to 0  
**#B** make weighted prediction for each example  
**#C** convert weighted predictions to  $-1/1$  labels  
**#D** convert labels from  $0/1$  to  $-1/1$

The test error of the ensemble learned by our implementation using 10 weak stumps is 4%.

**CAUTION** The boosting algorithm **we have implemented** requires negative examples and positive examples to be labeled  $-1$  and  $1$  respectively. In the example above, since the function `make_moons` returns labels  $y$  with negative examples and positive examples labeled  $0$  and  $1$  respectively, we convert them to  $-1$  and  $1$  with  $y = (2 * y) - 1$ .

Alternatively, if we stick to  $0$  and  $1$ , then the final classification can be obtained by applying a threshold (typically,  $0.5$ ) to the aggregated prediction. These steps are not necessary when using implementations provided by most machine-learning packages such as `scikit-learn` as they automatically preprocess a variety of training labels.

We visualize the performance of AdaBoost as the number of base estimators increases in the figure below. As we add more and more weak learners into the mix, the overall ensemble is increasingly boosted into a stronger, more complex and more nonlinear classifier.

While AdaBoost is generally more resistant to overfitting, like many other classifiers, overtraining a boosting algorithm can also result in overfitting, especially in the presence of noise. We will see how do deal with such situations in Section 4.3.

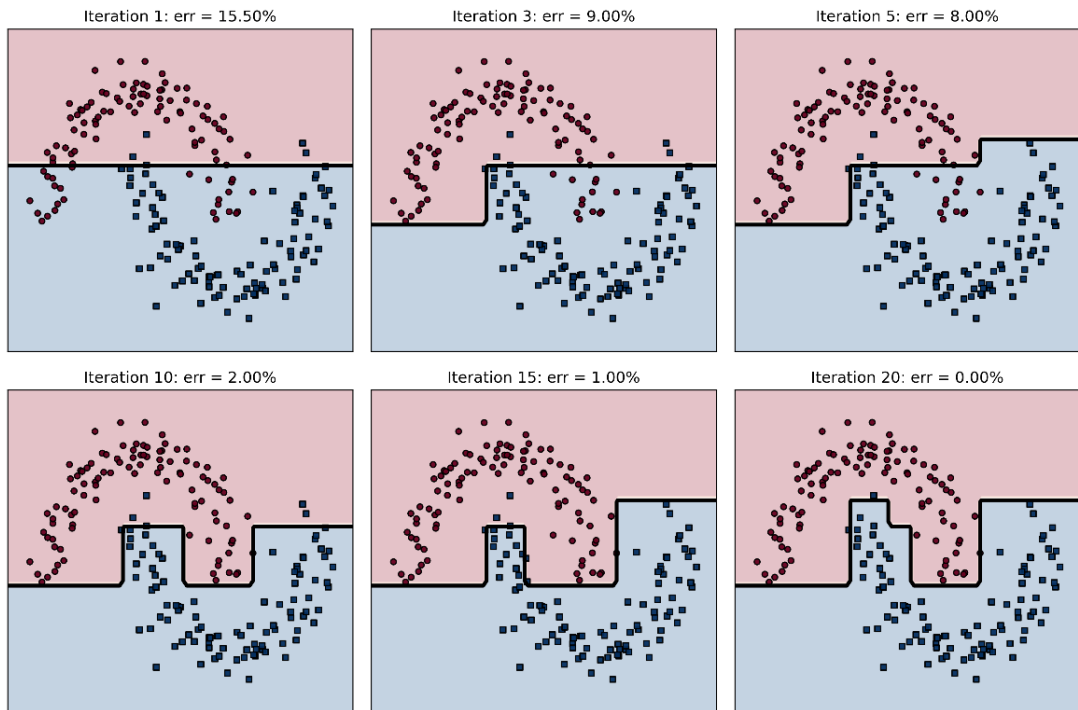


Figure 4.9 As the number of weak learners increases, the overall classifier is boosted into a strong model, which becomes increasingly nonlinear and is able to fit (and possibly overfit) the training data.

### 4.2.3 AdaBoost with scikit-learn

Now that we understand the intuition of how the AdaBoost classification algorithm works, we look at how to use `scikit-learn`'s `AdaBoostClassifier` package. `scikit-learn`'s implementation provides additional functionality including support for multi-class classification, as well as other base learning algorithms beyond decision trees.

There are three important arguments that the `AdaBoostClassifier` package takes for both binary as well as multi-class classification tasks:

1. `base_estimator`, the base learning algorithm AdaBoost uses to train weak learners. In our implementation, we used `DecisionTreeClassifier`. However, it is possible to use other weak learners such as shallow decision trees, shallow artificial neural networks and stochastic gradient-descent-based classifiers.
2. `n_estimators`, the number of weak learners that will be trained sequentially by AdaBoost, and
3. `learning_rate`, an additional parameter that progressively shrinks the contribution of each successive weak learner trained for the ensemble
4. smaller values of `learning_rate` make the weak learner weights  $\alpha$  smaller. Smaller  $\alpha$  means the variation in the example weights  $D_i$  decreases, and less diverse weak learners; larger values of `learning_rate` have the opposite effect and increase diversity in weak learners.

The `learning_rate` parameter has a natural interplay and tradeoff with `n_estimators`. Increasing `n_estimators` increases the number of iterations, which in turn, allows the training example weights  $D_i$  to keep growing. The unconstrained growth of example weights can be controlled by the `learning_rate`.

The example below illustrates `AdaBoostClassifier` in action on a binary classification data set. First, we load the breast cancer data and split into training and test sets.

```
>>> from sklearn.datasets import load_breast_cancer
>>> from sklearn.model_selection import train_test_split
>>> X, y = load_breast_cancer(return_X_y=True)
>>> Xtrn, Xtst, ytrn, ytst = train_test_split(X, y, test_size=0.25)
```

We will use shallow decision trees of depth 2 as base estimators for training.

```
>>> shallow_tree = DecisionTreeClassifier(max_depth=2)
>>> ensemble = AdaBoostClassifier(base_estimator=shallow_tree,
>>>                               n_estimators=20, learning_rate=0.75)
>>> ensemble.fit(Xtrn, ytrn)
```

After training, we can use the boosted ensemble to make predictions on the test set.

```
>>> ypred = ensemble.predict(Xtst)
>>> from sklearn.metrics import accuracy_score
>>> err = 1 - accuracy_score(ytst, ypred)
>>> err
0.07692307692307687
```

AdaBoost achieves test error rate of 7.69% on the breast cancer data set.

### **Multi-class Classification**

`scikit-learn`'s `AdaBoostClassifier` also supports multi-class classification, where data belongs to more than two classes. This is because `scikit-learn` contains the multi-class implementation of AdaBoost called Stagewise Additive Modeling using Multi-class Exponential loss, or SAMME.

SAMME is a generalization of Freund and Schapire's adaptive boosting algorithm (that we implemented in Section 4.2.2) from two to multiple classes. In addition to SAMME, `AdaBoostClassifier` also provides a variant called SAMME.R.

The key difference between these two algorithms is that SAMME.R handles real-valued predictions from base estimator algorithms, that is, class probabilities, while vanilla SAMME handles discrete predictions, that is, class labels.

Does this sound familiar? Recall from Chapter 3 that there are two types of combination functions: those that use the predicted class labels directly, and those that can use predicted class probabilities. This is precisely the difference between SAMME and SAMME.R as well.

The example below illustrates `AdaBoostClassifier` in action on a multi-class classification data set. First, we load the `iris` data and split into training and test sets.

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.utils.multiclass import unique_labels
>>> X, y = load_iris(return_X_y=True)
>>> Xtrn, Xtst, ytrn, ytst = train_test_split(X, y, test_size=0.25)
>>> unique_labels(y)
array([0, 1, 2])
```

As before, we can train and evaluate the AdaBoost on this multiclass data set.

```
>>> ensemble = AdaBoostClassifier(base_estimator=shallow_tree,
                                n_estimators=20,
                                learning_rate=0.75, algorithm='SAMME.R')
>>> ensemble.fit(Xtrn, ytrn)
>>> ypred = ensemble.predict(Xtst)
>>> err = 1 - accuracy_score(ytst, ypred)
>>> err
0.0699300699300699
```

AdaBoost achieves a test error of 6.99% on the (3-class) iris data set.

## **4.3 AdaBoost in Practice**

In this chapter, we look at some practical challenges we might expect to encounter when using AdaBoost and strategies we can use to ensure that we train robust models. AdaBoost's adaptive procedure makes it susceptible to outliers, or data points that are extremely noisy. In this section, we will see examples of how this problem can affect the robustness of AdaBoost and what we can do to mitigate it.



At the core of AdaBoost is its ability to adapt to mistakes made by previous weak learners. This adaptive property, however, can also be a disadvantage when *outliers are present*.

## OUTLIERS

Outliers are data points characterized by a high amount of noise. They are often the result of measurement or input errors and are prevalent in real data to varying degrees. Standard preprocessing techniques such as normalization often simply rescale the data and do not remove outliers, which allows them to continue to impact algorithm performance. This can be addressed by pre-processing the data to specifically detect and remove outliers.

For some tasks (for example, detecting network cyberattacks) the very thing we need to detect and classify (a cyberattack) will be an outlier, also called an anomaly, and extremely rare. In such situations, the goal of our learning task will be itself be anomaly detection.

AdaBoost is especially susceptible to outliers. Outliers are often misclassified by weak learners. Recall that AdaBoost increases the weight of misclassified examples. This means that the weight assigned to outliers continues to increase. When the next weak learner is trained

1. it continues to misclassify the outlier, in which case AdaBoost will increase its weight further, which in turn, causes succeeding weak learners to misclassify, fail and keep growing its weight, or
2. it correctly classifies the outlier, in which case AdaBoost has just overfit the data.

This is illustrated in the next figure.

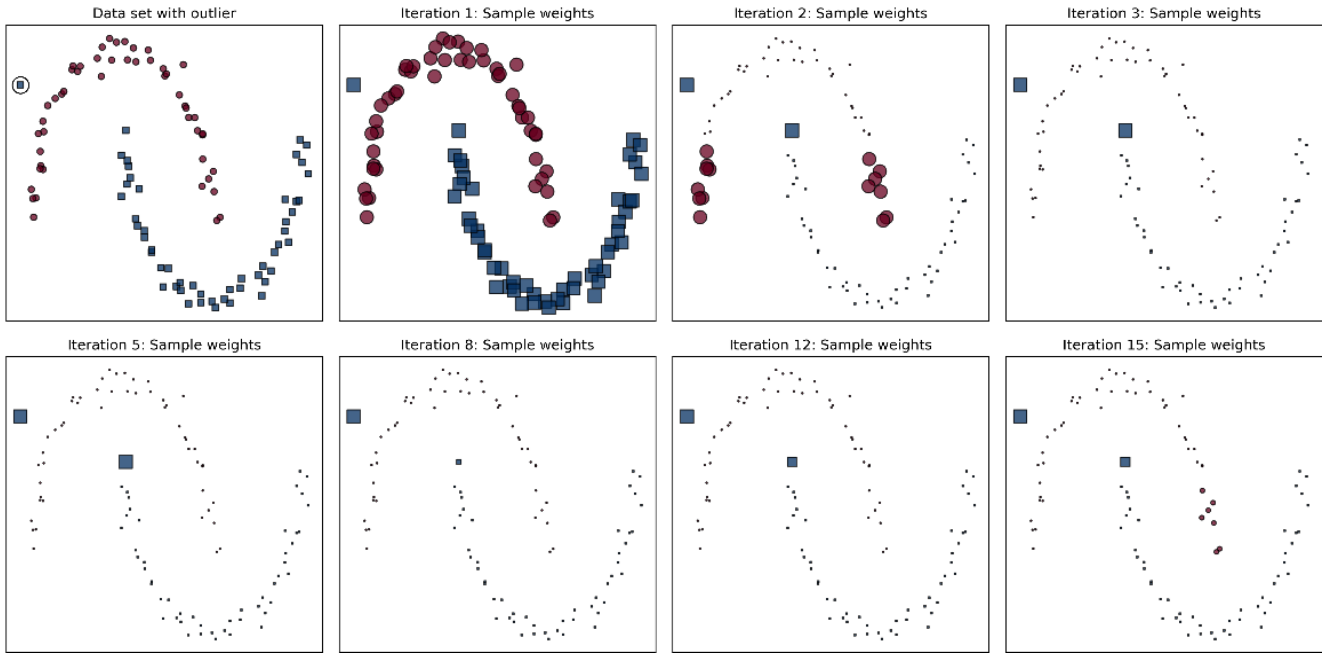


Figure 4.10 Consider a data set with an outlier (circled, top left). In iteration 1, it has the same weight as all the examples. As AdaBoost continues to sequentially train new weak learners, the weights of other data points eventually decrease as they are eventually correctly classified. The weight of the outlier continues to increase, ultimately resulting in overfitting.

Outliers force AdaBoost to spend a disproportionate amount of effort on training examples that are noisy. Put another way, outliers tend to confound AdaBoost and make it *less robust*.

### 4.3.1 Learning Rate

Now, we look at ways in which we can train robust models with AdaBoost. The first aspect we can control is *learning rate*, which adjusts the contribution of each estimator to the ensemble.

For example, a learning rate of 0.75 tells AdaBoost to decrease the overall contribution of each base estimator by a factor of 0.75. When there are outliers, a high learning rate will cause their influence to grow proportionally quickly, which can absolutely kill the performance of your model. Therefore, one way to mitigate the effect of outliers is to *lower the learning rate*.

As lowering the learning rate shrinks the contribution of each base estimator, controlling the learning rate is also known as shrinkage and is a form of model regularization to minimize overfitting. Concretely, at iteration  $t$ , the ensemble  $F_{t-1}$  is updated to  $F_t$  as

$$F_t(x) = F_{t-1}(x) + \eta \cdot \alpha_t \cdot h_t(x)$$

Here,  $\alpha_t$  is the weight of weak learner  $h_t$  (computed by AdaBoost) and  $\eta$  is the learning rate, a user-specified shrinkage hyperparameter in the range  $0 < \eta \leq 1$ .

A slower learning rate means that it will often take more iterations (and consequently, more base estimators) to build an effective ensemble. More iterations also mean more computational effort and longer training times. Often, however, slower learning rates may produce a robust model that generalizes better and may well be worth the effort.

An effective way to select the best learning rate is with a validation set or cross validation (CV). The listing below uses 10-fold cross validation to identify the best learning rate in the range [0.1,0.2,...,1.0]. We can observe the effectiveness of shrinkage on the breast cancer data:

```
>>> from sklearn.datasets import load_breast_cancer
>>> X, y = load_breast_cancer(return_X_y=True)
```

We use stratified  $k$ -fold CV, as we did with stacking. Recall that stratified means that the folds are created in such a way that the class distribution is preserved across the folds. This also helps with imbalanced data sets, as stratification ensures data from all classes is represented.

#### Listing 4.3 Cross validation to select the best learning rate

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import StratifiedKFold
import numpy as np

n_learning_rate_steps, n_folds = 10, 10
learning_rates = np.linspace(0.1, 1.0, num=n_learning_rate_steps) #A
splitter = StratifiedKFold(n_splits=n_folds, shuffle=True)
trn_err = np.zeros((n_learning_rate_steps, n_folds))
val_err = np.zeros((n_learning_rate_steps, n_folds))
stump = DecisionTreeClassifier(max_depth=1) #B

for i, rate in enumerate(learning_rates): #C
    for j, (trn, val) in enumerate(splitter.split(X, y)): #D
        model = AdaBoostClassifier(algorithm='SAMME', base_estimator=stump,
                                   n_estimators=10, learning_rate=rate)

        model.fit(X[trn, :], y[trn]) #E

        trn_err[i, j] = 1 - accuracy_score(y[trn],
                                           model.predict(X[trn, :]))
        val_err[i, j] = 1 - accuracy_score(y[val],
                                           model.predict(X[val, :])) #F

trn_err = np.mean(trn_err, axis=1)
val_err = np.mean(val_err, axis=1) #G
```

#A set up stratified 10-fold CV and initialize the search space

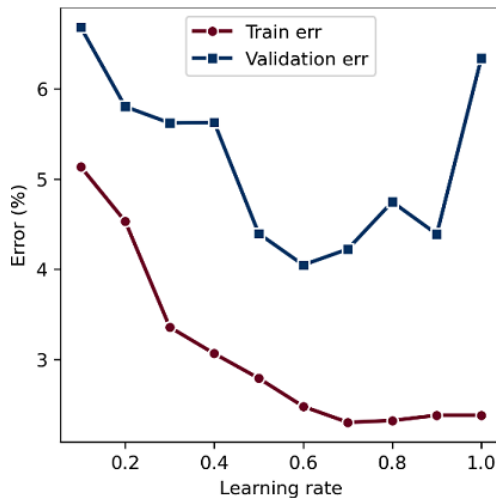
#B use decision stumps as weak learners

#C for all choices of learning rates

#D for train, validation sets

```
#E fit a model to training data in this fold
#F compute validation error for this fold
#G average the errors across the folds
```

We plot the results of this parameter search in Figure 4.11, which shows how the training and validation errors change as the learning rate increases. The number of base learners is fixed to 10. While the average training error continues to decrease with increasing learning rate, the best average validation error is achieved for `learning_rate=0.6`.



**Figure 4.11** Average training and validation errors for different learning rates. The validation error for `learning_rate=0.6` is lowest, and, in fact, lower than the default `learning_rate=1.0`.

### 4.3.2 Early Stopping and Pruning

Besides the `learning_rate`, the other important consideration for practical boosting is the number of base learners, `n_estimators`. It might be tempting to try to build an ensemble with a very large number of weak learners. However, this does not always translate to the best generalization performance.

In fact, it is often the case that we can achieve roughly the same performance with fewer base estimators than we think we might need. Identifying the least number of base estimators in order to build an effective ensemble is known as *early stopping*.

Maintaining fewer base estimators, can help control overfitting. Additionally, early stopping can also decrease training time as we end up having to train fewer base estimators.

The listing below uses a cross validation procedure identical to the one above to identify the best number of estimators. The learning rate here is fixed to 1.0.

**Listing 4.4 Cross validation to select the best number of weak learners**

```

n_estimator_steps, n_folds = 5, 10

number_of_stumps = np.arange(5, 50, n_estimator_steps) #A
splitter = StratifiedKFold(n_splits=n_folds, shuffle=True)

trn_err = np.zeros((len(number_of_stumps), n_folds))
val_err = np.zeros((len(number_of_stumps), n_folds))

stump = DecisionTreeClassifier(max_depth=1) #B
for i, n_stumps in enumerate(number_of_stumps): #C
    for j, (trn, val) in enumerate(splitter.split(X, y)): #D
        model = AdaBoostClassifier(algorithm='SAMME', base_estimator=stump,
                                   n_estimators=n_stumps, learning_rate=1.0)
        model.fit(X[trn, :], y[trn]) #E

        trn_err[i, j] = 1 - accuracy_score(y[trn],
                                           model.predict(X[trn, :]))
        val_err[i, j] = 1 - accuracy_score(y[val],
                                           model.predict(X[val, :])) #F

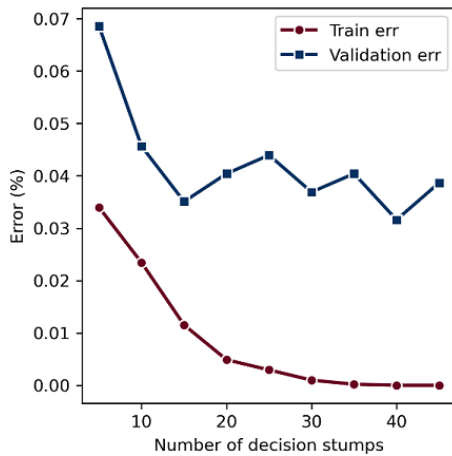
trn_err = np.mean(trn_err, axis=1)
val_err = np.mean(val_err, axis=1) #G

#A set up stratified 10-fold CV and initialize the search space
#B use decision stumps as weak learners
#C for all estimator sizes
#D for train, validation sets
#E fit a model to training data in this fold
#F compute validation error for this fold
#G average the errors across the folds

```

The results of this search for the best number of estimators is shown in Figure 4.12.

The average validation error suggests that it is sufficient to use as few as 20 decision trees to achieve comparable predictive performance on this data set. In practice, we can stop training early once the performance on the validation set reaches an acceptable level.



**Figure 4.12** Average training and validation errors for different numbers of base estimators (decision stumps, in this case). The validation error for `n_estimators=20` is lowest.

Early stopping is also known as *pre-pruning*, as we terminate training before exploring all possible values of `n_estimators`, and often leads to faster training times.

If we are not concerned about training time but want to be more judicious in selecting the number of base estimators, we can also consider *post-pruning*. Post-pruning means that we train a very large ensemble, and then drop the worst base estimators.

For AdaBoost, post-pruning drops all weak learners whose weights ( $\alpha_t$ ) are below a certain threshold. We can access the individual weak learners as well as their weights after training an `AdaBoostClassifier` through the fields `model.estimators_` and `model.estimator_weights_`. To prune the contribution of the least significant weak learners (those whose weight is below a certain `threshold`), we can simply set their weights to zero:

```
model.estimator_weights_[model.estimator_weights_ <= threshold] = 0.0
```

As before, a cross validation can be used to select a good threshold. Always remember that there is typically a tradeoff between AdaBoost's parameters, `learning_rate` and `n_estimators`. Lower learning rates typically require more iterations (hence more weak learners), while higher learning rates require fewer iterations (and fewer weak learners).

To be most effective, the best values of these parameters should be identified together through grid search combined with cross validation. An example of this is shown in the case study, which we look at next.

### Outlier Detection and Removal

While the procedures above are generally effective on noisy data sets, training examples with high amounts of noise (that is, outliers) can still cause significant issues. In such cases, it is often advisable to pre-process the data set to remove these outliers entirely.

## 4.4 Case Study: Handwritten Digit Classification

One of the earliest machine-learning applications is on handwritten digit classification. In fact, this task has been studied so extensively since the early 1990s that we might consider it the “Hello World!” of object recognition.

This task originated with the United States Postal Service’s attempts to automate digit recognition to accelerate mail processing by rapidly identifying zip codes. Since then, several different handwritten data sets have been created, and are widely used to benchmark and evaluate various machine-learning algorithms.

In this case study, we will use `scikit-learn`’s `digits` data set to illustrate the effectiveness of AdaBoost. The data set consists of 1797 scanned images of handwritten digits from 0 to 9. Each digit is associated with a unique label, which makes this a 10-class classification problem. There are roughly 180 digits per class.

The digits themselves are represented as 16 x 16 normalized greyscale bitmaps, which when flattened results in a 64-dimensional vector for each handwritten digit. The training set is of size 1797 examples x 64 features. We can load the data set directly from `scikit-learn`:

```
>>> from sklearn.datasets import load_digits
>>> X, y = load_digits(return_X_y=True)
```

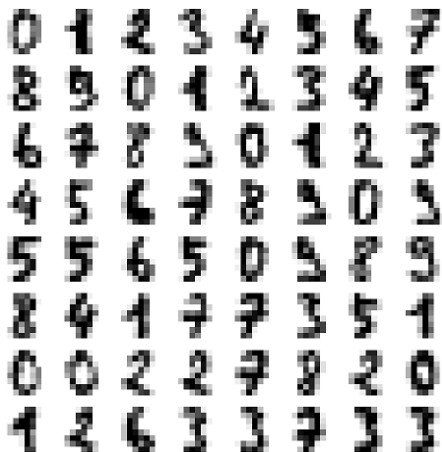


Figure 4.1. A snapshot of the digits data set used in this case study.

### 4.4.1 Dimensionality Reduction with t-SNE

While AdaBoost can effectively handle the dimensionality of the digits data set (64 features), we will (rather aggressively) look to reduce the dimensionality to 2. The main reason for this is to be able to visualize the data as well as the models learned by AdaBoost.

We'll use a nonlinear dimensionality reduction technique known as t-distributed stochastic neighbor embedding or t-SNE. t-SNE is a highly effective pre-processing technique for the digits data set and extracts an effective embedding in a two-dimensional space.

#### t-SNE

Stochastic neighbor embedding, as its name suggests, uses neighborhood information to construct a lower dimensional embedding. Specifically, it exploits the similarity between two examples  $x_i$  and  $x_j$ . In our case,  $x_i$  and  $x_j$  are two example digits from the data set and are 64-dimensional. The similarity between two digits can be measured as

$$\text{sim}(x_i, x_j) = \exp(-\|x_i - x_j\|^2 / (2 \sigma_i^2))$$

where  $\|x_i - x_j\|^2$  is the squared distance between  $x_i$  and  $x_j$  and  $\sigma_i^2$  is a similarity parameter.

You may have seen this form of similarity function in other areas of machine learning, especially in the context of support vector machines, where it is known as the radial-basis function kernel or the Gaussian kernel.

The similarity between,  $x_i$  and  $x_j$  can be converted to a probability  $p_{j|i}$  that  $x_j$  is a neighbor of  $x_i$ . The probability is just a normalized similarity measure, where we normalize by the sum of similarities of all points in the data set  $x_k$  with  $x_i$ :

$$p_{j|i} = \frac{\text{sim}(x_i, x_j)}{\text{sum of sim}(x_i, x_k) \text{ for all data points}} = \frac{\exp(-\|x_i - x_j\|^2 / (2 \sigma_i^2))}{\sum_{k \neq i} \exp(-\|x_i - x_k\|^2 / (2 \sigma_i^2))}$$

Let's say that the 2-dimensional embedding of these two digits is given by  $z_i$  and  $z_j$ . Then it is natural to expect that two similar digits  $x_i$  and  $x_j$  will continue to be neighbors even embedding into  $z_i$  and  $z_j$  respectively. The probability of  $z_j$  is a neighbor of  $z_i$  can be measured similarly:

$$q_{j|i} = \frac{\text{sim}(z_i, z_j)}{\text{sum of sim}(z_i, z_k) \text{ for all data points}} = \frac{\exp(-\|z_i - z_j\|^2)}{\sum_{k \neq i} \exp(-\|z_i - z_k\|^2)}$$

where we assume that the variance in the 2-dimensional ( $z$ -space) is  $1/\sqrt{2}$ .

Then, we can identify the embeddings of all the points by ensuring that  $q_{j|i}$ , the probabilities in the 2-dimensional embedding space ( $z$ -space) are well-aligned with  $p_{j|i}$  in the 64-dimensional original digit space ( $x$ -space). Mathematically, this is achieved by minimizing the KL-divergence between the distributions  $q_{j|i}$  and  $p_{j|i}$ .



With `scikit-learn`, the embeddings can be computed very easily:

```
>>> from sklearn.manifold import TSNE
>>> Xemb = TSNE(n_components=2).fit_transform(X)
```

So, what does this data set look like when embedded into a two-dimensional space?

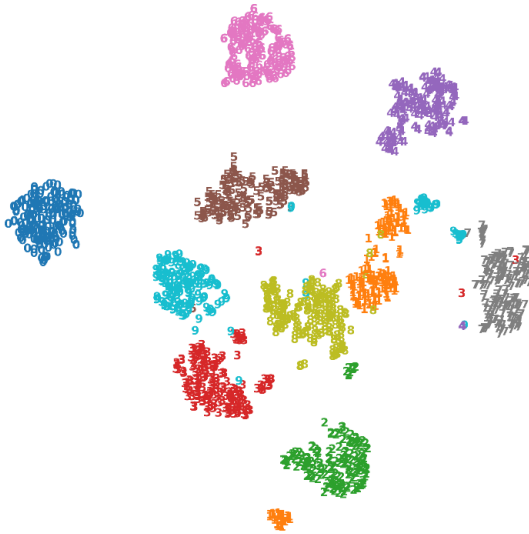


Figure 4.14 Visualization of the 2d embedding of digits data set produced by t-SNE, which is able to embed and separate the digits, effectively clustering them.

### ***Train-Test Split***

As always, it is important to hold aside a part of the training data for evaluation and to quantify the predictive performance of our models on future data. We split the lower-dimensional data `Xemb` and the labels into training and test sets.

```
>>> from sklearn.model_selection import train_test_split
>>> Xtrn, Xtst, ytrn, ytst = train_test_split(Xemb, y,
                                           test_size=0.2, stratify=y)
```

Observe the use of `stratify=y`, to ensure that the ratios of the different digits in train and test sets are identical.

## **4.4.2 Boosting**

We will now train an AdaBoost model for this digit classification task. Recall from our earlier discussion that AdaBoost requires us to first decide the type of base estimator. We continue to use decision stumps.

```
>>> stump = DecisionTreeClassifier(max_depth=2)
>>> ensemble = AdaBoostClassifier(algorithm='SAMME', base_estimator=stump)
```

In the previous section, we saw how to use cross validation for selecting the best value of `learning_rate` and `n_estimators` individually. In practice, we have to identify the *best combination* of `learning_rate` and `n_estimators`. For this, we will employ a combination of *k*-fold cross validation and grid search.

The basic idea is to consider different combinations of `learning_rate` and `n_estimators` and evaluate what their performance would be like via cross validation. First, we select various parameter values we want to explore.

```
>>> parameters_to_search = {'n_estimators': [200, 300, 400, 500],
                             'learning_rate': [0.6, 0.8, 1.0]}
```

Next, we make a scoring function to evaluate the performance of each parameter combination. For this task, we use the *balanced accuracy score*, which is essentially just the accuracy score weighted by each class. This scoring criterion is effective for multi-class classification problems such as this one, and also for imbalanced data sets.

```
>>> from sklearn.metrics import balanced_accuracy_score, make_scorer
>>> scorer = make_scorer(balanced_accuracy_score, greater_is_better=True)
```

Now, we set up and run the grid search to identify the best parameter combination with the `GridSearchCV` class. Several arguments to `GridSearchCV` are of interest to us. The parameter `cv=5` specifies 5-fold cross validation and `n_jobs=-1` specifies that the job should use all available cores for parallel processing (see Chapter 2).

```
>>> from sklearn.model_selection import GridSearchCV
>>> search = GridSearchCV(ensemble, param_grid=parameters_to_search,
                          scoring=scorer,
                          cv=5, n_jobs=-1, refit=True)
>>> search.fit(Xtrn, ytrn)
```

The final parameter in `GridSearchCV` is set to `refit=True`. This tells `GridSearchCV` to train a final model using all the available training data using the best parameter combination it has identified.

**TIP** For many data sets, it may not be computationally efficient to exhaustively explore and validate all possible hyperparameter choices with `GridSearchCV`. For such cases, it may be more computationally efficient to use `RandomizedSearchCV`, which samples a much smaller subset of hyperparameter combinations to validate.

After training, we can look up the scores for every parameter combination and even pull out the best results.

```
>>> best_combo = search.cv_results_['params'][search.best_index_]
>>> best_score = search.best_score_
>>> print('The best parameter settings are {0}, with score = {1}.'.format(
        best_combo, best_score))
The best parameter settings are {'learning_rate': 0.6, 'n_estimators': 200}, with
score = 0. 0.9826321839080459.
```

The best model is also available (because we set `refit=True`). Note that this model is trained using the `best_combo` parameters using the entire training data (`Xtrn`, `ytrn`) by `GridSearchCV`. This model is available in `search.best_estimator_` and can be used for making predictions on the *test data*:

```
>>> ypred = search.best_estimator_.predict(Xtst)
```

How well did this model do? We can first look at the classification report:

```
>>> from sklearn.metrics import classification_report
>>> print('Classification report:\n{0}\n'.format(classification_report(ytst, ypred)))
```

Classification report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	36
1	0.97	0.94	0.96	36
2	1.00	1.00	1.00	35
3	0.97	0.78	0.87	37
4	1.00	1.00	1.00	36
5	0.97	0.97	0.97	37
6	1.00	1.00	1.00	36
7	1.00	1.00	1.00	36
8	0.94	0.97	0.96	35
9	0.77	0.92	0.84	36
accuracy			0.96	360
macro avg	0.96	0.96	0.96	360
weighted avg	0.96	0.96	0.96	360

AdaBoost does quite well on most digits. It seems that it struggles most with 3s and 9s, which both have lower F1-scores. We can also look at the *confusion matrix*, which will give us a good idea which letters are being confounded with others.

```
>>> from sklearn.metrics import confusion_matrix
>>> print("Confusion matrix: \n {0}".format(confusion_matrix(ytst, ypred)))
```

Confusion matrix:

```
[[36 0 0 0 0 0 0 0 0 0]
 [ 0 34 0 0 0 0 0 0 1 1]
 [ 0 0 35 0 0 0 0 0 0 0]
 [ 0 0 0 29 0 0 0 0 0 8]
 [ 0 0 0 0 36 0 0 0 0 0]
 [ 0 0 0 0 0 36 0 0 0 1]
 [ 0 0 0 0 0 0 36 0 0 0]
 [ 0 0 0 0 0 0 0 36 0 0]
 [ 0 1 0 0 0 0 0 0 34 0]
 [ 0 0 0 1 0 1 0 0 1 33]]
```

Each row of the confusion matrix corresponds to the true labels (digits to 0 to 9) and each column corresponds to the predicted labels. The (3, 9) entry in the confusion matrix (last row, sixth column) indicates that several 3s are misclassified as 9s by AdaBoost.

Finally, we can plot the decision boundaries of the trained AdaBoost model, shown below.

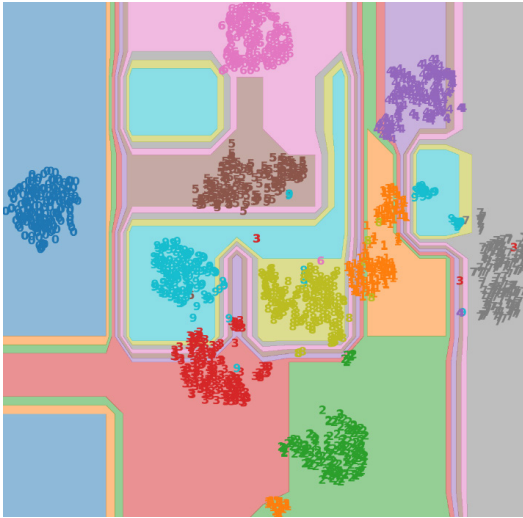


Figure 4.15 The decision boundaries learned by AdaBoost on the embeddings of the digits data set.

This case study illustrates how AdaBoost can boost the performance of weak learners into a powerful strong learner that can achieve good performance on a complex task. Before we end the chapter, we look at another boosting algorithm, LogitBoost.

## 4.5 LogitBoost: Boosting with the Logistic Loss

We now move on to a second boosting algorithm called *LogitBoost*. The development of LogitBoost was motivated by the desire to bring loss functions from established classification models (such as logistic regression) into the AdaBoost framework.

In this manner, the general boosting framework can be applied to specific classification settings in order to train boosted ensembles with properties similar to those classifiers.

### *Logistic vs. Exponential Loss Functions*

Recall from Section 4.2.2, that AdaBoost updates weights  $\alpha_t$  of weak learners with

$$\alpha_t = \frac{1}{2} \log \left( \frac{1 - \epsilon_t}{\epsilon_t} \right)$$

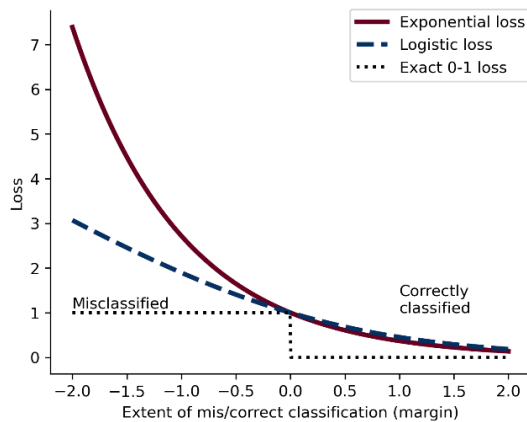
Where does this come from? This expression is a consequence of the fact that AdaBoost optimizes the exponential loss. In particular, the exponential loss of an example  $(x, y)$  with respect to a weak learner  $h_t(x)$  is given by

$$L(x; \alpha_t) = \exp(-\alpha_t y \cdot h_t(x)),$$

where  $y$  is the true label and  $h_t(x)$  is the prediction made by  $h_t$ . LogitBoost, or Logistic Boosting, differs from AdaBoost in three important ways. First, LogitBoost optimizes the logistic loss,

$$L(x; a_t) = \log(1 + \exp(-a_t \cdot y \cdot h_t(x)))$$

The logistic loss penalizes mistakes differently than the exponential loss (see figure below).



**Figure 4.16** Comparing the exponential loss and the logistic loss functions.

You may have seen the logistic loss in other machine-learning formulations, most notably logistic regression. The exact 0–1 loss (also known as the misclassification loss) is an idealized loss function that returns 0 for correctly classified examples and 1 for misclassified examples. However, this loss is difficult to optimize as it is not continuous. In order to build feasible machine learning algorithms, different methods use different surrogates.

The exponential loss function and the logistic loss function both penalize correctly classified examples similarly. Training examples which are correctly classified with greater confidence have corresponding losses close to zero.

The exponential loss penalizes misclassified examples far more harshly than the logistic loss, which makes it more susceptible to outliers and noise. The logistic loss is more measured.

### ***Regression As A Weak Learning Algorithm For Classification***

The second key difference is that AdaBoost works with predictions, while LogitBoost works with prediction probabilities. More precisely, AdaBoost works with the predictions of the overall ensemble  $F(x)$ , while LogitBoost works with prediction probabilities,  $P(x)$ .

The probability of predicting a training example  $x$  as a positive example is given by

$$P(y = 1 | \mathbf{x}) = \frac{1}{1 + e^{-F(\mathbf{x})}}$$

while the probability of predicting  $x$  as a negative example is given by  $P(y = 0 | \mathbf{x}) = 1 - P(y = 1 | \mathbf{x})$ . This fact directly influences our choice of base estimator.

The third key difference is, since AdaBoost works directly with discrete predictions ( $-1$  or  $1$ , for negative and positive examples), it uses any classification algorithm as the base learning algorithm. LogitBoost, instead, works with continuous prediction probabilities. Consequently, *it uses any regression algorithm as the base learning algorithm*.

### Implementing LogitBoost

Putting all of these together, the LogitBoost algorithm performs the following steps within each iteration. The probability  $P(y_j = 1 | x_i)$  is abbreviated  $P_i$ .

1. Compute the working response, or how much the prediction probability differs from the true label,

$$z_i = \frac{y_i - P_i}{P_i(1 - P_i)}$$

2. Update the example weights,  $D_i = P_i(1 - P_i)$
3. Train a weak regression stump  $h_t(x)$  on the weighted examples  $(x_i, z_i, D_i)$

Update the ensemble,  $F_{t+1}(x) = F_t(x) + h_t(x)$

4. Update the example probabilities

$$P_i = \frac{1}{1 + e^{-F_{t+1}(x)}}$$

As we can see from Step 3 above, LogitBoost, like AdaBoost, is an *additive ensemble*. This means that it ensembles base estimators and combines their predictions additively. The LogitBoost algorithm is also easy to implement, as the listing below shows.

#### Listing 4.5 LogitBoost for classification

```
import numpy as np
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import accuracy_score
from scipy.special import expit

def fit_logitboosting(X, y, n_estimators=10):
    n_samples, n_features = X.shape
    D = np.ones((n_samples, )) / n_samples
    p = np.full((n_samples, ), 0.5) #A
    estimators = []

    for t in range(n_estimators):
        z = (y - p) / (p * (1 - p)) #B
        D = p * (1 - p) #C
```

```

    h = DecisionTreeRegressor(max_depth=1) #D
    h.fit(X, z, sample_weight=D)
    estimators.append(h) #E

    if t == 0:
        margin = np.array([h.predict(X)
                           for h in estimators]).reshape(-1, )
    else:
        margin = np.sum(np.array([h.predict(X)
                                   for h in estimators]), axis=0)

    p = expit(margin) #F

    return estimators

```

#A initialize example weights, pred probabilities

#B compute working responses

#C compute new example weights

#D use decision tree regression as base estimators for classification problem

#E Append weak learner to ensemble  $F_{t+1}(x) = F_t(x) + h_t(x)$

#F update prediction probabilities, 
$$P_i = \frac{1}{1 + e^{-F_{t+1}(x)}}$$

The `predict_boosting` function described in Listing 4.2 can be used to make predictions with the LogitBoost ensembles as well. However, LogitBoost requires training labels to be in 0/1 form while AdaBoost requires them to be in  $-1/1$  form. Thus, we modify that function slightly to return 0/1 labels.

#### Listing 4.5 LogitBoost for prediction

```

def predict_logit_boosting(X, estimators):
    pred = np.zeros((X.shape[0], ))

    for h in estimators:
        pred += h.predict(X)

    y = (np.sign(pred) + 1) / 2 #A

    return y

```

#A convert  $-1/1$  predictions to 0/1

As with AdaBoost, we can visualize how the ensemble trained by LogitBoost evolves over several iterations in Figure 4.17. Contrast this figure with Figure 4.9, which shows the evolution of the ensemble trained by AdaBoost over several iterations.

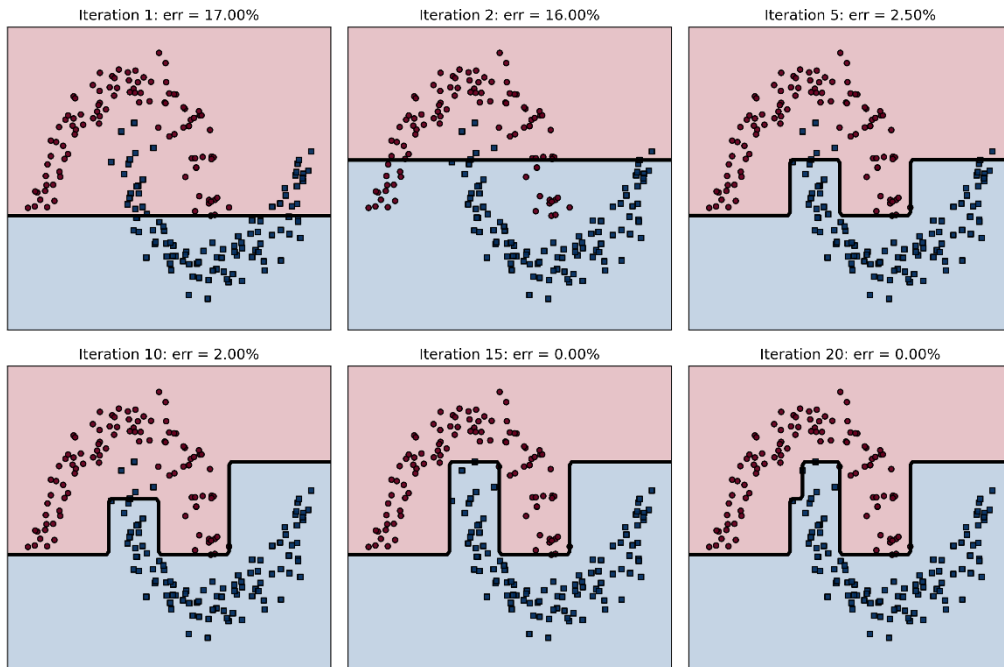


Figure 4.17 LogitBoost uses decision tree regression to train regression stumps as weak learners to sequentially optimize the logistic loss.

## 4.6 Summary

In this chapter, we were introduced to sequential ensemble methods of weak learners.

1. AdaBoost, or Adaptive Boosting is a sequential ensemble algorithm that uses weak learners as base estimators.
2. In classification, a weak learner is a simple model that performs only slightly better than random guessing, that is 50% accuracy. Decision stumps and shallow decision trees are examples of weak learners.
3. AdaBoost maintains and updates weights over training examples. It uses *reweighting* both to prioritize misclassified examples and to promote ensemble diversity.
4. AdaBoost is also an *additive ensemble* in that it makes final predictions through weighted additive (linear) combinations of the predictions of its base estimators.
5. AdaBoost is generally robust to overfitting as it ensembles several weak learners. However, it is sensitive to outliers owing to its adaptive reweighting strategy, which repeatedly increases the weight of outliers over iterations.
6. The performance of AdaBoost can be improved by finding a good tradeoff between the learning rate and number of base estimators
7. Cross validation with grid search is commonly deployed to identify the best parameter tradeoff between learning rate and number of estimators.



8. Under the hood, AdaBoost ultimately optimizes the exponential loss function.
9. LogitBoost is another boosting algorithm that optimizes the logistic loss function. It differs from AdaBoost in two other ways

We have now seen two boosting algorithms that handle two different loss functions. Is there a way to generalize boosting to different loss functions and for different tasks such as regression?

It turns out that the answer to this question is an emphatic “Yes!”, as long as the loss function is differentiable (and you can compute its gradient). This is the intuition behind *gradient boosting*, which we will look into in the next two chapters.

# 5

## *Sequential Ensembles: Gradient Boosting*

### **This chapter covers**

- Using gradient descent to optimize loss functions for training models
- Implementing and understanding how gradient boosting works
- Training fast gradient-boosting models with histogram-based splitting for tree learning
- Introducing LightGBM: a powerful framework for gradient boosting
- Avoiding overfitting with LightGBM in practice
- Using custom loss function with LightGBM

The last chapter introduced boosting: where we train weak learners sequentially and “boost” them into a strong ensemble model. An important sequential ensemble method introduced in the last chapter is adaptive boosting, or AdaBoost.

AdaBoost is a foundational boosting model that trains a new weak learner to fix the misclassifications of the previous weak learner. It does this by maintaining and adaptively updating weights on training examples. These weights reflect the extent of misclassification and indicate priority training examples to the base learning algorithm.

In this chapter we look at an alternative to weights on training examples to convey misclassification information to a base learning algorithm for boosting: loss function gradients.

Recall that we use loss function to measure how well a model is fitting each training example in the data set. The gradient of the loss function for a single example is called the residual and, as we will see shortly, captures the error between true and predicted labels. This error or residual, of course, measures the amount of misclassification.

In contrast to AdaBoost, which uses weights as a surrogate for residuals, gradient boosting uses these residuals directly! Thus, gradient boosting is another sequential ensemble method that aims to train weak learners over residuals (that is, gradients).

The framework of gradient boosting can be applied to any loss function, which means that any classification, regression or ranking problem can be “boosted” using weak learners. This flexibility has been a key reason for the emergence and ubiquity of gradient boosting as a **state-of-the-art** ensemble approach.

Several powerful packages and implementations of gradient boosting are available (`LightGBM`, `CatBoost`, `XGBoost`) and provide the ability to train models on big data efficiently via parallel computing and GPUs.

This chapter is organized as follows. To gain a deeper understanding of gradient boosting, we will need a deeper understanding of gradient descent. So, we kick off the chapter with an example of gradient descent can be used to train a machine learning model (Section 5.1).

Section 5.2 aims to provide intuition for learning with residuals, which is at the heart of gradient boosting. Then, we implement our own version of gradient boosting and step through it to understand how it combines gradient descent and boosting at every step to train a sequential ensemble.

Section 5.2 also introduces histogram-based gradient boosting, which essentially bins the training data for tree learning in order to significantly accelerate learning and scaling to larger data sets.

Section 5.3 introduces `LightGBM`, a free and open-source gradient boosting package and important tool for building and deploying real-world ML applications. In Section 5.4, we further see how we can avoid overfitting with strategies such as early stopping and adapting the learning rate to train effective models with `LightGBM` and how to extend `LightGBM` to custom loss functions.

All of this leads us into a demonstration of how to use gradient boosting on a real-world task: document retrieval, which will be the focus of our chapter-concluding case study (Section 5.5). Document retrieval, a form of information retrieval is a key task in many applications, one we have all used at some time or another (for example, web search engines).

To understand gradient boosting, we will first have to understand gradient descent, a simple yet effective approach that is widely used for training many machine-learning algorithms. This will help us contextualize the role gradient descent plays inside gradient boosting, both conceptually and algorithmically.

## 5.1 Gradient Descent for Minimization

We now delve into gradient descent, an optimization approach at the heart of many training algorithms. Understanding gradient descent will allow us to understand how the gradient boosting framework ingeniously combines this optimization procedure with ensemble learning.

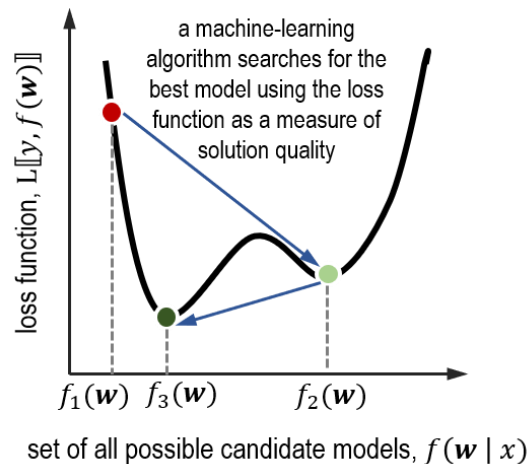
Optimization, or the search for the “best”, is at the heart of many applications. Indeed, at the heart of all machine learning, is our search for the best model.

**NOTE** Learning problems are often cast as optimization problems. For example, training is essentially finding the “best” fitting model given the data. If the notion of “best” is characterized by a loss function, then training is cast as a minimization problem as the best model corresponds to the lowest loss. Alternately, if the notion of “best” is

characterized by a likelihood function, then training is cast as a maximization problem as the best model corresponds to the highest likelihood (or probability). Unless specified, we will characterize model quality or fit using loss functions, which will require us to perform minimization.

Loss functions explicitly measure the fit of a model on a data set. Most often, we measure loss with respect to the true labels, by quantifying the error between the predicted and true labels. Thus, the “best” model will have the lowest error, or loss.

You may be familiar with loss functions such as cross entropy (for classification) or mean squared error (for regression). We will revisit cross entropy in Section 5.4.2 and mean squared error in Section 7. Given a loss function, training is the search for the “optimal” model that minimizes the loss. This is illustrated in the figure below.



**Figure 5.1.** An optimization procedure for finding the “best” model. Machine learning algorithms search for the best model among a set of all possible candidate models. The notion of “best” is quantified by the loss function, which evaluates the quality of a selected candidate using the labels and the data. Thus, machine-learning algorithms are essentially optimization procedures. Here, the optimization procedure sequentially identifies increasingly better models  $f_1$ ,  $f_2$  and the final model:  $f_3$ .

One example of such a search we may be familiar with is grid search for parameter selection during training of, say decision trees. With grid search we choose among many modeling choices: number of leaves, maximum tree depth, etc. systematically and exhaustively over a grid of parameters.

Another, more effective optimization technique is gradient descent, which uses first derivative information, or gradients to guide our search. In this section, we look at two examples of gradient descent. The first is a simple illustrative example to understand and visualize the basics of how gradient descent works.

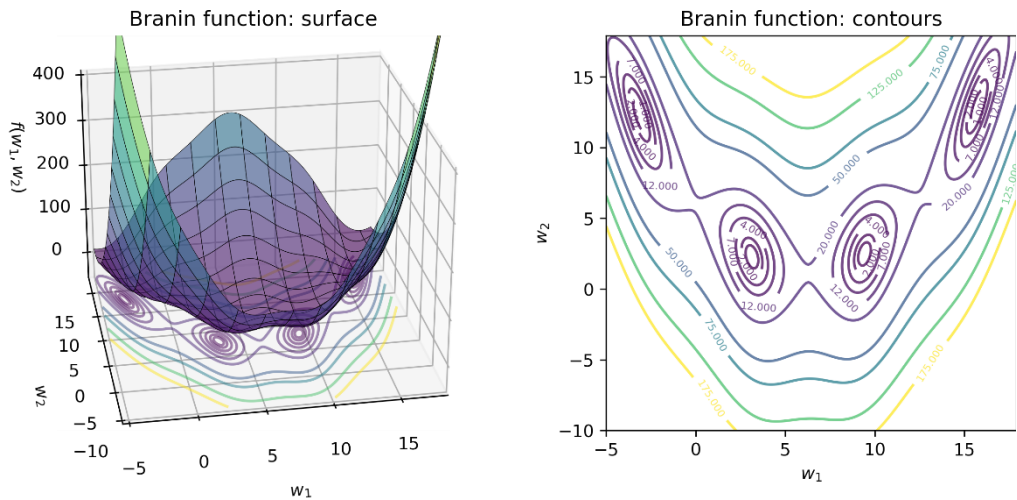
The second example demonstrates how gradient descent can be used on an actual loss function with data to train a machine-learning model.

### 5.1.1 Gradient Descent with an Illustrative Example

We will use the Branin function, a commonly used example function, to illustrate how gradient descent works, before moving on to a more concrete case grounded in machine learning (Sec 5.1.2). The Branin function is a function of two variables ( $\mathbf{w}_1$  and  $\mathbf{w}_2$ ), defined as follows:

$$f(\mathbf{w}_1, \mathbf{w}_2) = a(\mathbf{w}_2 - b\mathbf{w}_1^2 - c\mathbf{w}_1 - r)^2 + s(1 - t) \cos(\mathbf{w}_1) + s,$$

where  $a = 1$ ,  $b = 5.1 / 4\pi^2$ ,  $c = 5\pi$ ,  $r = 6$ ,  $s = 10$  and  $t = 1 - 8\pi$  are fixed constants, which we will not worry about. We can visualize this function by plotting a three-dimensional plot of  $\mathbf{w}_1$  vs.  $\mathbf{w}_2$  vs.  $f(\mathbf{w}_1, \mathbf{w}_2)$ . The figures below illustrate the three-dimensional surface plot as well as the contour plot (the top view).



**Figure 5.2.** The surface plot (left) and contour plot (right) of the Branin function. We can visually verify that this function has four minima, which are the centers of the elliptical regions in the contour plot.

Visualization of the Branin function shows us that it takes the smallest values at four different locations, which are called local minimizers or minima. So how can we identify these local minima?

There is always the brute force approach: we can make a grid over the variables  $\mathbf{w}_1$  and  $\mathbf{w}_2$  and evaluate  $f(\mathbf{w}_1, \mathbf{w}_2)$  at every possible combination exhaustively. There are several problems with this. First, how coarse or fine should our grid be? If our grid is too coarse, we may miss the minimizer in our search. If our grid is too fine, then we will have a very large number of grid points to search over, making our optimization procedure very slow.

Second, and more worryingly, this approach ignores all the extra information inherent in the function itself, which could be quite helpful in guiding our search. For instance, the first derivatives, or the rates of change of  $f(\mathbf{w}_1, \mathbf{w}_2)$  with respect to  $\mathbf{w}_1$  and  $\mathbf{w}_2$  can be very helpful.

### UNDERSTANDING AND IMPLEMENTING GRADIENT DESCENT

The first derivative information is known as the gradient of  $f(\mathbf{w}_1, \mathbf{w}_2)$  and is a measure of the (local) slope of the function surface. More importantly, the gradient points in the direction of steepest ascent, that is, moving in the direction of steepest ascent will lead us to bigger values of  $f(\mathbf{w}_1, \mathbf{w}_2)$ .

If we want to use gradient information to find the minimizers, then we have to travel in the opposite direction of the gradient! This is precisely the simple, yet highly effective principle behind gradient descent: keep going in the direction of the negative gradient, and you will end up at a (local) minimizer.

We can formalize this intuition in the pseudo-code below, which describes the steps of gradient descent. As we see below, gradient descent is an iterative procedure that steadily moves towards a local minimizer by moving in the direction of steepest descent: the negative gradient.

```

initialize:  $\mathbf{w}_{old}$  = some initial guess, converged=False
while not converged:
1. compute the negative gradient at  $\mathbf{w}_{old}$  and normalize to unit length (direction,  $\mathbf{g}$ )
2. compute the step length using line search (distance,  $\alpha$ )
3. update the solution:  $\mathbf{w}_{new} = \mathbf{w}_{old} - \text{distance} \cdot \text{direction}$ 
4. if change between  $\mathbf{w}_{new}$  and  $\mathbf{w}_{old}$  is below some specified tolerance:
    converged=True, so break
5.  $\mathbf{w}_{old} = \mathbf{w}_{new}$ , get ready for the next iteration

```

The gradient descent procedure is fairly straightforward. First, we initialize our solution (and call it  $\mathbf{w}_{old}$ ); this can be a random initialization or perhaps a more sophisticated guess. Starting from this initial guess, we compute the negative gradient, which tells us which direction we want to go.

Next, we compute a step length, which tells us the distance or how far we want to go in the direction of the negative gradient. Computing the step length is an important step, as it ensures that we don't overshoot our solution.

The step length computation is another optimization problem, where we wish to identify a scalar  $\alpha \geq 0$  such that traveling along the gradient  $\mathbf{g}$  for a distance of  $\alpha$  produces the biggest decrease in the loss function. Formally, this is known as a *line search problem* and is often used to efficiently select step lengths during optimization.

**NOTE** Many optimization packages and tools (such as `scipy.optimize` used in this chapter) provide exact and approximate line search functions which can be used to identify step lengths. Alternately, step length can also be set according to some pre-determined strategy, often for efficiency.

In machine learning, the step length is often called the learning rate, and is represented by the Greek letter eta ( $\eta$ ).

With a direction and distance, we can take this step and update our solution guess to  $\mathbf{w}_{new}$ . Once we get there, we check for convergence. There are several tests for convergence; here, we assume convergence if the solution doesn't change much between consecutive iterations.

If converged, then we've found a local minimizer. If not, then we iterate again from  $\mathbf{w}_{new}$ .

The listing below shows how to perform gradient descent.

### Listing 5.1. Gradient Descent

```

import numpy as np
from scipy.optimize import line_search

def gradient_descent(f, g, x_init, max_iter=100, args=()):  #A
    converged = False  #B
    n_iter = 0

    x_old, x_new = np.array(x_init), None
    descent_path = np.full((max_iter + 1, 2), fill_value=np.nan)
    descent_path[n_iter] = x_old

    while not converged:
        n_iter += 1
        gradient = -g(x_old, *args)  #C
        direction = gradient / np.linalg.norm(gradient)  #D

        step = line_search(f, g, x_old, direction, args=args)  #E

        if step[0] is None:  #F
            distance = 1.0
        else:
            distance = step[0]

        x_new = x_old + distance * direction  #G
        descent_path[n_iter] = x_new

        err = np.linalg.norm(x_new - x_old)  #H
        if err <= 1e-3 or n_iter >= max_iter:
            converged = True  #I

        x_old = x_new  #J

    return x_new, descent_path

```

#A gradient descent requires a function  $f$  and its gradient  $g$

#B initialize gradient descent to not converged

#C compute the negative gradient

#D normalize gradient to unit length

#E compute step length using line search

#F if line search fails, make it 1.0

#G compute the update

#H compute change from previous iteration

#I converged if change is small or max. iterations reached

#J get ready for the next iteration

We can test drive this gradient descent procedure on the Branin function. To do this, in addition to the function itself, we will also need its gradient. We can compute the gradient explicitly by dredging up the basics of calculus (if not the memories of it).

The gradient is a vector with two components: the gradient of  $f$  with respect to  $w_1$  and  $w_2$  respectively. With this gradient we can compute the direction of steepest increase everywhere:

$$g(w_1, w_2) = \begin{bmatrix} \frac{\partial f(w_1, w_2)}{\partial w_1} \\ \frac{\partial f(w_1, w_2)}{\partial w_2} \end{bmatrix} = \begin{bmatrix} 2a(w_2 - b w_1^2 + c w_1 - r) \cdot (-2 b w_1 + c) - s(1 - t) \sin(w_1) \\ 2a(w_2 - b w_1^2 + c w_1 - r) \end{bmatrix}$$

We can implement the Branin function and its gradient as shown below:

```
def branin(w, a, b, c, r, s, t):
    return a * (w[1] - b * w[0]**2 + c * w[0] - r)**2 +
           s * (1 - t) * np.cos(w[0]) + s

def branin_gradient(w, a, b, c, r, s, t):
    return np.array([2 * a * (w[1] - b * w[0]**2 + c * w[0] - r) *
                    (-2 * b * w[0] + c) - s * (1 - t) * np.sin(w[0]),
                    2 * a * (w[1] - b * w[0]**2 + c * w[0] - r)])
```

In addition to the function and the gradient, Listing 5.1 also requires an initial guess `x_init`. Here, we will initialize gradient descent with `w_init = [-4, -5]`. Now, we can call the gradient descent procedure.

```
a, b, c, r, s, t = 1, 5.1/(4 * np.pi**2), 5/np.pi, 6, 10, 1/(8 * np.pi)
w_init = np.array([-4, -5])
w_optimal, w_path = gradient_descent(branin, branin_gradient,
                                    w_init, args=(a, b, c, r, s, t))
```

Gradient descent returns an optimal solution `w_optimal` and the optimization path `w_path`, the sequence of intermediate solutions that the procedure iterated through on its way to the optimal solution.

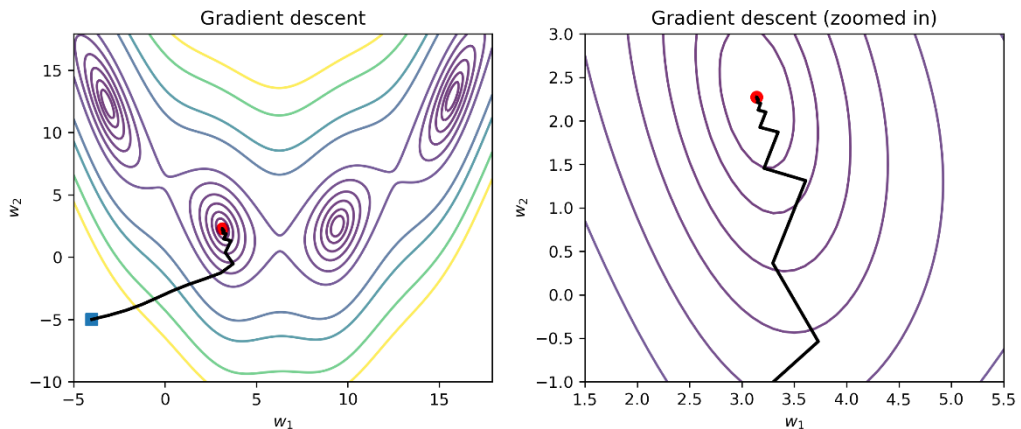


Figure 5.3. The figure on the left shows the full descent path of gradient descent, starting from `[-4, -5]` (blue square) and converging to one of the local minima (red circle). The figure on the right shows the zoomed-in version of the same descent path as gradient descent approaches the solution. Note that the gradient steps become smaller and the descent algorithm tends to zig-zag as it approaches the solution.



And voila! In the figure above, we see that gradient descent is able to reach one of the four local minimizers of the Branin function. There are several important things to note about gradient descent.

#### PROPERTIES OF GRADIENT DESCENT

First, observe that the gradient steps become smaller and smaller as we approach one of the minimizers. This is because gradients vanish at minimizers. More importantly, gradient descent exhibits zig-zagging behavior. This is because the gradient doesn't point at the local minimizer itself, it points in the direction of steepest ascent (or descent, if negative).

The gradient at a point essentially captures local information, that is, the nature of the function close to that point. Gradient descent chains several such gradient steps to get to a minimizer. When the gradient descent has to pass through steep valleys, it's tendency to use local information causes it to bounce around the valley walls as it moves towards the minimum.

Second, gradient descent converged to one of the four local minimizers of the Branin function. Can you get it to converge to a different minimizer? Yes! By changing the initialization. The figure below illustrates various gradient descent paths for different initializations.

The sensitivity of gradient descent to initialization is illustrated in Figure 5.4, where different random initializations cause gradient descent to converge to different local minimizers. This behavior may be familiar to those of you who have used k-means clustering: different initializations will often produce different clusterings, each of which is a different local solution.

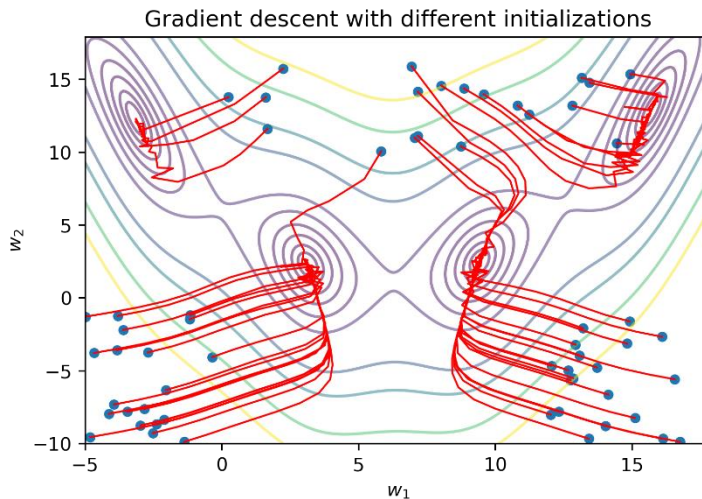


Figure 5.4. Different initializations will cause gradient descent to reach different local minima.

An interesting challenge with gradient descent is in identifying the appropriate initialization as different initializations lead gradient descent to different local minimizers. From an optimization perspective, it is not always easy to identify the correct initialization beforehand.

However, from a machine-learning perspective, it may be the case that the different local solutions demonstrate the same generalization behavior. That is, the locally optimal learned models all have similar predictive performance.

This situation is commonly encountered with neural networks and deep learning, which is why training procedures for many deep models are initialized from pre-trained solutions.

**TIP** The sensitivity of gradient descent to initialization depends on the type of function being optimized. If the function is convex or cup-shaped everywhere, then any local minimizer that gradient descent identifies will always be a global minimizer too! This is the case with models learned by support vector machine (SVM) optimizers. However, a good initial guess is still important as it may cause the algorithm to converge faster.

Many real-world problems are typically non-convex and have several local minima. Gradient descent will converge to one of them, depending on initialization and shape of the function in the locality of the initial guess. The objective function of k-means clustering is non-convex, which is why different initializations produce different clusterings.

See *Algorithms for Optimization* by Kochenderfer and Wheeler (MIT Press, 2019) for a solid and hand-on introduction to optimization.

### 5.1.2 Gradient Descent over Loss Functions for Training

Now that we understand the basics of how gradient descent works on a simple example (the Branin function), let's build a classification task from scratch using a loss function of our own. Then, we will use gradient descent to train a model. First, we create a 2d classification problem:

```
from sklearn.datasets import make_blobs
X, y = make_blobs(n_samples=200, n_features=2,
                  centers=[[-1.5, -1.5], [1.5, 1.5]])
```

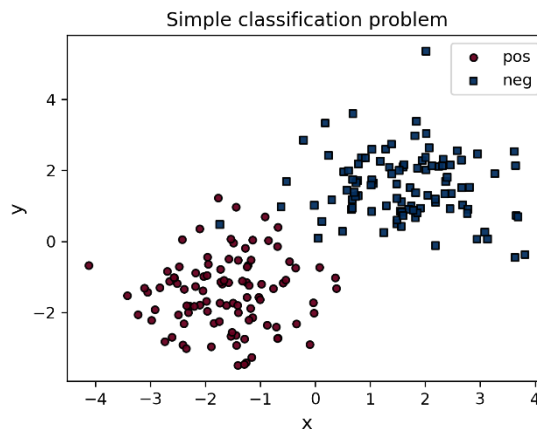


Figure 5.5. A (nearly) linearly separable two-class data set over which we will learn a classifier. The positive examples have labels  $y = 1$  and the negative examples have labels  $y = 0$ .

We specifically create a linearly separable data set (with some noise, of course) so that we can train a linear separator or classification function. This will keep our loss function formulation simple, and our gradients easy to calculate.

The classifier we wish to train,  $h_w(\mathbf{x})$ , takes two-dimensional data points  $\mathbf{x} = [x_1, x_2]^T$  and returns a prediction using a linear function

$$h_w(\mathbf{x}) = w_1 x_1 + w_2 x_2$$

The classifier is parameterized by  $\mathbf{w} = [w_1, w_2]^T$ , which we have to learn using the training examples. In order to learn, we will need a loss function over the true label and predicted label. We will use the familiar squared loss (or squared error) that measures the cost as for an individual, labeled training example  $(\mathbf{x}, y)$

$$f_{\text{loss}}(y, \mathbf{x}) = \frac{1}{2} (y - h_w(\mathbf{x}))^2 = \frac{1}{2} (y - w_1 x_1 - w_2 x_2)^2$$

The squared loss function computes the loss between the prediction of the current candidate model ( $h_w$ ) on a single training example ( $\mathbf{x}$ ) and its true label ( $y$ ). For the  $n$  training examples in the data set, the overall loss can be written as

$$f_{\text{loss}}(y, \mathbf{x}) = \frac{1}{2} \sum_{i=1}^n (y - h_w(\mathbf{x}))^2 = \frac{1}{2} \sum_{i=1}^n (y - w_1 x_1 - w_2 x_2)^2 = \frac{1}{2} (\mathbf{y} - \mathbf{X}\mathbf{w})^T (\mathbf{y} - \mathbf{X}\mathbf{w})$$

The expression for the overall loss is just the sum of the individual losses of the  $n$  training examples in the data set.

The expression to the far right above is simply the vectorized version of the overall loss, which uses dot products instead of loops. In the vectorized version, the boldface  $\mathbf{y}$  is an  $n \times 1$  vector of true labels,  $\mathbf{X}$  is a  $n \times 2$  data matrix, where each row is a two-dimensional training example, and  $\mathbf{w}$  is a  $2 \times 1$  model vector that we want to learn.

As before, we will need the gradient of the loss function:

$$\mathbf{g}(w_1, w_2) = \begin{bmatrix} \frac{\partial f_{\text{loss}}(w_1, w_2)}{\partial w_1} \\ \frac{\partial f_{\text{loss}}(w_1, w_2)}{\partial w_2} \end{bmatrix} = \begin{bmatrix} -\sum_{i=1}^n (y - w_1 x_1 - w_2 x_2) x_1 \\ -\sum_{i=1}^n (y - w_1 x_1 - w_2 x_2) x_2 \end{bmatrix} = -\mathbf{X}^T (\mathbf{y} - \mathbf{X}\mathbf{w}).$$

We can implement the vectorized versions as they are more compact and more efficient as they avoid explicit loops for summation.

```
def squared_loss(w, X, y):
    return 0.5 * np.sum((y - np.dot(X, w))**2)

def squared_loss_gradient(w, X, y):
    return -np.dot(X.T, (y - np.dot(X, w)))
```

**TIP** If you are alarmed at the prospect of hand-computing gradients, despair not, for there are other alternatives that can numerically approximate the gradients and are used for training many machine-learning models including deep learning and gradient boosting.

These alternatives rely on auto-differentiation, which is based on first principles of numerical calculus and linear algebra, to compute approximate gradients. An easy-to-use auto-gradient tool is the function `scipy.optimize.approx_fprime` available in the `scipy` scientific package.

A far more powerful auto-differentiation tool is JAX (<https://github.com/google/jax>), which is free, open source and in research project status as of the time of writing this book. JAX is intended for computing gradients of complex functions representing deep neural networks with many layers. It can differentiate through loops, branches and even recursion, and has GPU support for large-scale gradient computations.

What does our loss function look like? We can visualize it as before.

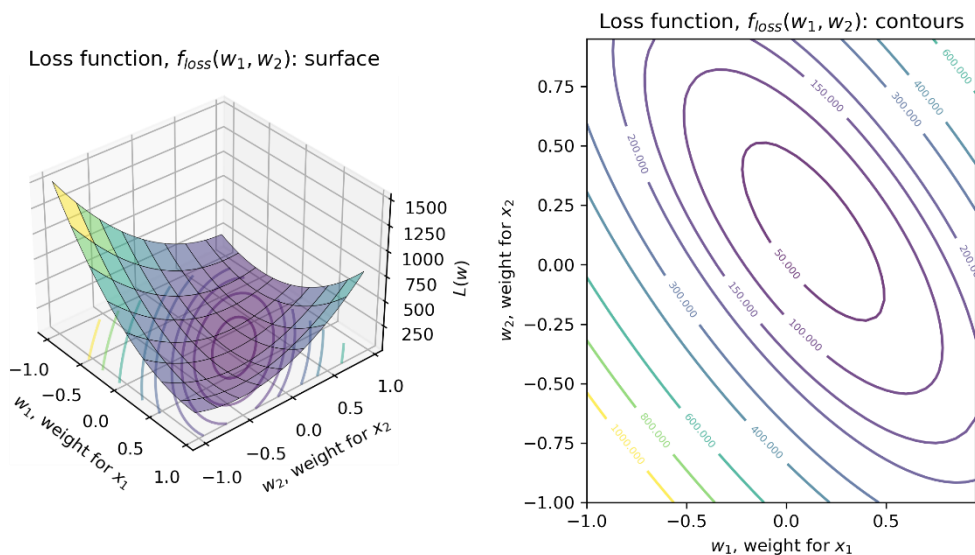


Figure 5.6. The overall squared loss over the entire training set, visualized.

As before, we perform gradient descent, this time initializing at  $\mathbf{w} = [0.0, -0.99]$  using the code snippet below, with the gradient descent path shown in Figure 5.7.

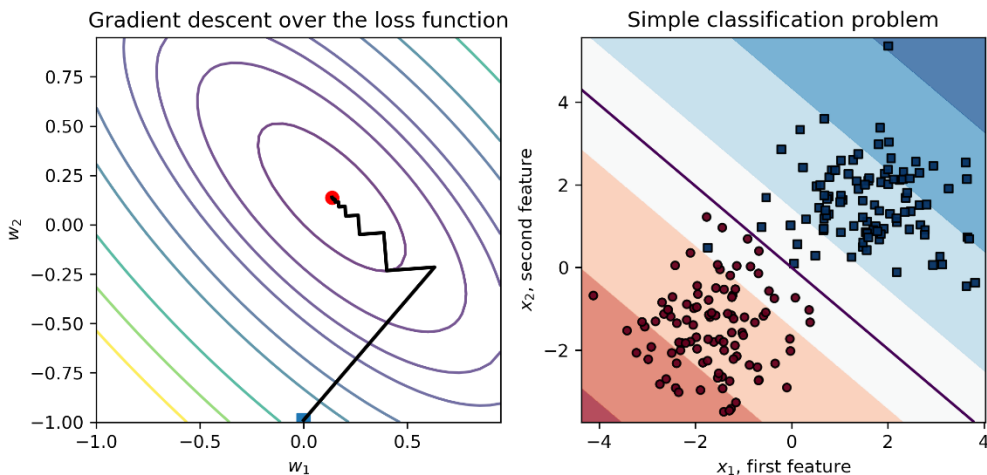
```
w_init = np.array([0.0, -0.99])
w, path = gradient_descent(squared_loss, squared_loss_gradient,
                          w_init, args=(X, y))
print(w)
[0.17390066 0.11937649]
```

Gradient descent has learned a final learned model  $w^* = [0.174, 0.119]$ . We visualize the linear classifier learned by our gradient descent procedure in Figure 5.7. In addition to visually confirming that the gradient descent procedure learned a useful model, we can also compute training accuracy.

Recall that a linear classifier  $h_w(x) = w_1x_1 + w_2x_2$  returns real-valued predictions, which we need to convert to 0 or 1. This is straightforward: we simply assign all positive predictions (examples above the line) to the class  $y_{\text{pred}} = 1$  and negative predictions (examples below the line) to the class  $y_{\text{pred}} = 0$ .

```
ypred = (np.dot(X, w) >= 0).astype(int)
from sklearn.metrics import accuracy_score
accuracy_score(y, ypred)
0.995
```

Success! The training accuracy learned by our implementation of gradient descent is 99.5%.



**Figure 5.7.** (left) Gradient descent over our squared loss function starting at  $w_{\text{init}}$  (blue square) and converging at the optimal solution (red circle). (right) The learned model  $w = [0.174, 0.119]$  is a linear classifier that fits the training data quite well as it separates both the classes.

Now that we understand how gradient descent uses gradient information sequentially to minimize a loss function during training, let's see how we can extend it with boosting to train a sequential ensemble.

## 5.2 Gradient Boosting: Gradient Descent + Boosting

In gradient boosting, we aim to train a sequence of weak learners that approximate the gradient at each iteration. Gradient boosting and its successor Newton boosting are currently considered state-of-the-art ensemble methods and are widely implemented and deployed for several tasks in diverse application areas.

We will first look at the intuition of gradient boosting and contrast it with another familiar boosting method: AdaBoost. Armed with this intuition, as before, we will implement our own version of gradient boosting to visualize what is really going on under the hood.

Then, we will look at two gradient boosting approaches available in `scikit-learn`: the `GradientBoostingClassifier`, and its more scalable counterpart `HistogramGradientBoostingClassifier`. This will set us up nicely for LightGBM, a powerful and flexible implementation of gradient boosting widely used for practical applications.

### 5.2.1 Intuition: Learning with Residuals

The key component of sequential ensemble methods, such as AdaBoost and gradient boosting, is that they aim to train a new weak estimator at each iteration to fix the errors made by the weak estimator at the previous iteration. However, AdaBoost and gradient boosting train new weak estimators on poorly classified examples in rather different ways.

#### ADABOOST VS. GRADIENT BOOSTING

AdaBoost identifies high-priority training examples by weighting them such that misclassified examples have higher weights than correctly classified ones. In this way, AdaBoost can tell the base learning algorithm which training examples it should focus on in the current iteration.

In contrast, gradient boosting uses residuals or errors (between the true and predicted labels) to tell the base learning algorithm which training examples it should focus on in the next iteration.

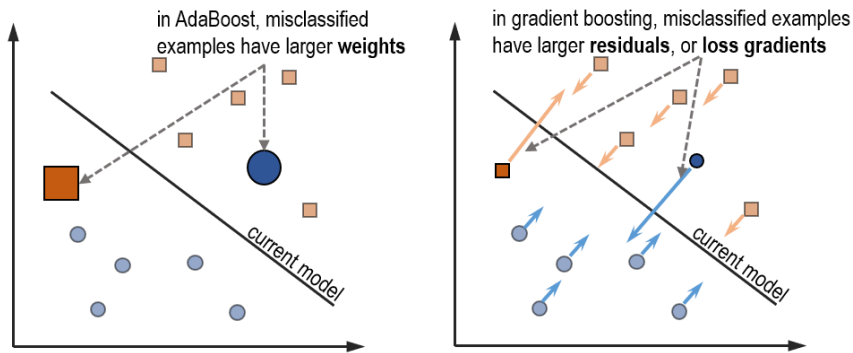
What exactly is a residual? For a training example, it is simply the error between the true label and the corresponding prediction. Intuitively, a correctly classified example must have a small residual and a misclassified example must have a large residual.

More concretely, if a classifier  $h$  makes a prediction  $h(\mathbf{x})$  on a training example  $\mathbf{x}$ , a naïve way of computing the residual would be to directly measure the difference between them

$$\text{residual}(\text{true}, \text{predicted}) = \text{residual}(y, h(\mathbf{x})) = y - h(\mathbf{x}).$$

Recall the squared loss function we were using previously:  $f_{\text{loss}}(\mathbf{y}, \mathbf{x}) = \sum (\mathbf{y} - \mathbf{h}(\mathbf{x}))^2$ . The gradient of this loss  $f$  with respect to our model  $h$  is

$$\text{gradient}(\text{true}, \text{predicted}) = \frac{\partial f_{\text{loss}}}{\partial h}(y, h(\mathbf{x})) = -(y - h(\mathbf{x}))$$



**Figure 5.8. Comparing AdaBoost (left) to gradient boosting (right).** Both approaches train weak estimators that improve classification performance on misclassified examples. AdaBoost uses weights, with misclassified examples being assigned higher weights. Gradient boosting uses residuals, with misclassified examples having higher residuals. The residuals are nothing but negative loss gradients.

The negative gradient of the squared loss is exactly the same as our residual! This means that the gradient of the loss function is a measure of the misclassification and is the residual.

Training examples that are badly misclassified will have large gradients (or residuals, or errors) as the gap between the true and predicted labels will be large. Training examples that are correctly classified will have small gradients.

Thus, analogous to AdaBoost, we have a measure of how badly each training example is misclassified. How can we use this information to train a weak learner?

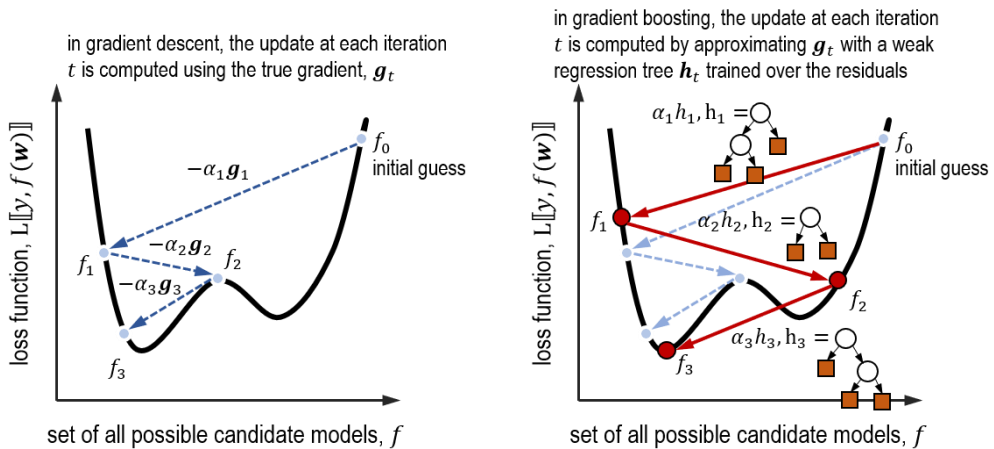
#### WEAK LEARNERS TO APPROXIMATE GRADIENTS

Continuing our analogy with AdaBoost, recall that once it assigns weights to all the training examples, we have a weight-augmented data set  $(\mathbf{x}_i, y_i, D_i)$  with  $i = 1, \dots, n$ , of weighted examples. Thus, training a weak learner in AdaBoost is an instance of a weighted classification problem. With an appropriate base classification algorithm, AdaBoost trains a weak classifier.

In gradient boosting, we no longer have weights  $D_i$ . Instead, we have residuals (or negative loss gradients),  $\mathbf{r}_i$  and a residual-augmented data set  $(\mathbf{x}_i, -\mathbf{r}_i)$ . Instead of classification labels ( $y_i = 0$  or  $1$ ) and example weights ( $D_i$ ), each training example now has an associated residual, which can be viewed as a real-valued label.

Thus, training a weak learner in gradient boosting is an instance of a regression problem, which requires a base learning algorithm such as decision tree regression. When trained, weak estimators in gradient boosting can be viewed as approximate gradients.

The figure below illustrates how gradient descent differs from gradient boosting and how gradient boosting is conceptually similar to gradient descent. The key difference between the two is that gradient descent directly uses the negative gradient, while gradient boosting trains a weak regressor to approximate the negative gradient.



**Figure 5.9.** Comparing gradient descent (left) to gradient boosting (right). At iteration  $t$ , gradient descent updates the model using the negative gradient,  $-\mathbf{g}_t$ . At iteration  $t$ , gradient boosting approximates the negative gradient by training a weak regressor,  $h_t$ , on the negative residuals  $-r_t^i$ . The step length  $\alpha_t$  in gradient descent is equivalent to the hypothesis weight of each base estimator in a sequential ensemble.

**NOTE** Gradient boosting aims to fit a weak estimator to residuals, which are real-valued. Thus, gradient boosting will **always** need to use a regression algorithm as a base learning algorithm and learn regressors as weak estimators. This will be the case even when the loss function corresponds to binary or multi-class classification, regression or ranking.

We now have all the ingredients to formalize the algorithmic steps of gradient boosting.

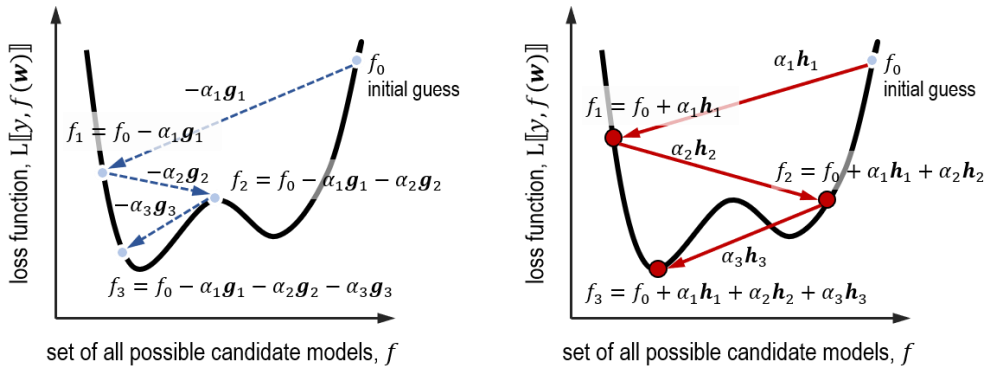
#### GRADIENT BOOSTING IS GRADIENT DESCENT + BOOSTING

To summarize, gradient boosting combines gradient descent and boosting.

- Like AdaBoost, gradient boosting trains a weak learner to fix the mistakes made by the previous weak learner. Adaboost uses example weights to focus learning on misclassified examples, while gradient boosting uses example residuals to do the same.
- Like gradient descent, gradient boosting updates the current model with gradient information. Gradient descent uses the negative gradient directly, while gradient boosting trains a weak regressor over the negative residuals to approximate the gradient.

Finally, both gradient descent and gradient boosting are additive algorithms, that is, they generate sequences of intermediate terms that are additively combined to produce the final model. This is apparent in the figure below.





**Figure 5.10.** Both gradient descent (left) and gradient boosting (right) produce a sequence of updates. In gradient descent, each iteration additively updates the current model with the new negative gradient ( $-g_i$ ). In gradient boosting, each iteration additively updates the current model with the new approximate weak gradient estimate (the regression tree,  $h_i$ ).

At each iteration, AdaBoost, gradient descent and gradient boosting all update the current model using an additive expression of the form:

$$\text{new model} = \text{old model} + (\text{step length}) * (\text{update direction}),$$

or more formally,

$$\mathbf{F}_{(t)}(\mathbf{x}) \square \mathbf{F}_t(\mathbf{x}) \square \alpha_t \cdot \mathbf{h}_t(\mathbf{x})$$

We can unravel the expression above for iterations  $t, t-1, t-2, \dots, 0$  to obtain the overall update sequence AdaBoost, gradient descent and gradient boosting produce:

$$\mathbf{F}_{(t)}(\mathbf{x}) \square \mathbf{F}_0(\mathbf{x}) + \alpha_1 \cdot \mathbf{h}_1(\mathbf{x}) + \alpha_2 \cdot \mathbf{h}_2(\mathbf{x}) \square \dots + \alpha_{(t-1)} \cdot \mathbf{h}_{(t-1)}(\mathbf{x}) + \alpha_t \cdot \mathbf{h}_t(\mathbf{x}).$$

The key differences between the three algorithms are in how we compute the updates  $\mathbf{h}_t$  and the hypothesis weights (also known as step lengths)  $\alpha_t$ . We can summarize the update steps of all three algorithms in Table 5.1.

**Table 5.1. Comparing AdaBoost, gradient descent and gradient boosting.**

Algorithm	Loss function	Base learning algo.	Update dir $\mathbf{h}_t(\mathbf{x})$ .	Step length $\alpha_t$
AdaBoost for classification	Exponential	Classification with weighted examples	Weak classifier	Computed in closed form
Gradient descent	User-specified	None	Gradient vector	Line search
Gradient	User-specified	Regression with	Weak regressor	Line search

Why is Gradient Boosting = Gradient Descent + Boosting? Because it *generalizes* the boosting procedure from the exponential loss function used by AdaBoost to any user-specified loss function. In order for gradient boosting to flexibly adapt to wide variety of loss functions, it adopts two general procedures: (1) approximate gradients using weak regressors and, (2) compute the hypothesis weights (or step lengths) using line search.

## 5.2.2 Implementing Gradient Boosting

As before, we will put our intuition to practice by implementing our own version of gradient boosting. The basic algorithm can be outlined with the following pseudocode:

```
initialize:  $F \leftarrow f_0$ , some constant value
for  $t = 1$  to  $T$ :
1. compute the negative residuals for each example,  $r_i^t = -\frac{\partial L}{\partial F}(\mathbf{x}_i)$ 
2. fit a weak decision tree regressor  $h_t(\mathbf{x})$  using the training set  $(\mathbf{x}_i, r_i)_{i=1}^n$ 
3. compute the step length ( $\alpha_t$ ) using line search
4. update the model:  $F \leftarrow F + \alpha_t \cdot h_t(\mathbf{x})$ 
```

This training procedure is almost the same as that of gradient descent except for a couple of differences: (1) instead of using the negative gradient, we use an approximate gradient trained on the negative residuals, and (2) instead of checking for convergence, the algorithm terminates after a finite number of iterations  $T$ . The listing below implements this pseudo-code specifically for the squared loss.

### Listing 5.2 Gradient Boosting for the squared loss

```
def fit_gradient_boosting(X, y, n_estimators=10):
    n_samples, n_features = X.shape #A
    n_estimators = 10
    estimators = [] #B
    F = np.full((n_samples, ), 0.0) #C

    for t in range(n_estimators):
        residuals = y - F #D
        h = DecisionTreeRegressor(max_depth=1)
        h.fit(X, residuals) #E

        hreg = h.predict(X) #F
        loss = lambda a: np.linalg.norm(y - (F + a * hreg))**2 #G
        step = minimize_scalar(loss, method='golden') #H
        a = step.x

        F += a * hreg #I
        estimators.append((a, h)) #J

    return estimators
```

#A get dimensions of the data set  
#B initialize an empty ensemble

```

#C predictions of the ensemble on the training set
#D compute residuals as negative gradients of the squared loss
#E fit weak regression tree ( $h_i$ ) to the examples and residuals
#F get predictions of the weak learner,  $h_i$ 
#G set up the loss function as a line search problem
#H find the best step length using the golden section search
#I update the ensemble predictions
#J update the ensemble

```

Once the model is learned, we can make predictions as with the AdaBoost ensemble. Note that, just like our AdaBoost implementation previously, this model returns predictions of -1/1 rather than 0/1

### Listing 5.3: Predictions using gradient boosted model

```

def predict_gradient_boosting(X, estimators):
    pred = np.zeros((X.shape[0], )) #A

    for a, h in estimators:
        pred += a * h.predict(X) #B

    y = np.sign(pred) #C

    return y

```

```

#A initialize all the predictions to 0
#B aggregate individual predictions from each regressor
#C convert weighted predictions to -1/1 labels

```

We can test drive this implementation on a simple two-moons classification example. Note that we convert the training labels from 0/1 to -1/1 to ensure that we learn and predict correctly.

```

from sklearn.datasets import make_moons
X, y = make_moons(n_samples=200, noise=0.15, random_state=13)
y = 2 * y - 1 #A
from sklearn.model_selection import train_test_split
Xtrn, Xtst, ytrn, ytst = train_test_split(X, y, #B
                                         test_size=0.25, random_state=11)

estimators = fit_gradient_boosting(Xtrn, ytrn)
ypred = predict_gradient_boosting(Xtst, estimators)

from sklearn.metrics import accuracy_score
tst_err = 1 - accuracy_score(ytst, ypred) #C
tst_err
0.060000000000000005

```

```

#A convert training labels to -1/1
#B split into train and test sets
#C train and get test error

```

The prediction of this model is 6%, which is pretty good.

### VISUALIZING GRADIENT BOOSTING ITERATIONS

Finally, to comprehensively nail down our understanding of gradient boosting, let us step through the first few iterations to see how gradient boosting uses residuals to boost classification.

In our implementation, we initialize our predictions to be  $F(x_i) = 0$ . This means that in the first iteration, the negative residuals for examples in Class 1 will be  $r_i = 1 - 0 = 1$ , and the residuals for the examples in Class 0 will be  $r_i = 0 - 0 = 0$ . This is evident in the figure below.

In the first iteration, all the training examples have high residuals (either +1 or -1), and the base learning algorithm (decision tree regression) has to train a weak regressor taking all these residuals into account. The trained regression tree ( $h_1$ ) is shown in Figure 5.11 (right).

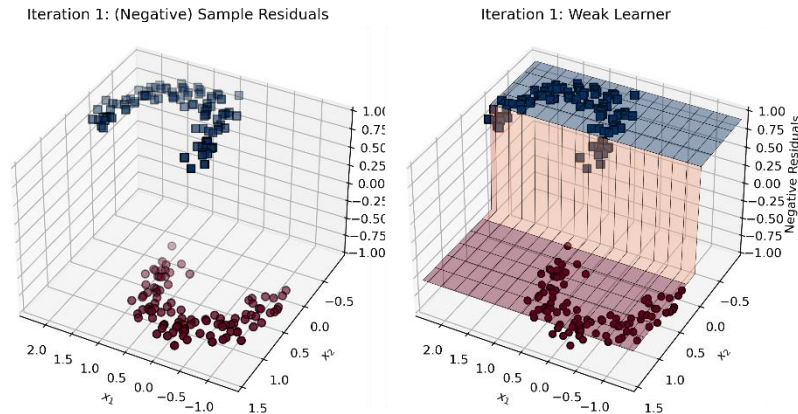


Figure 5.11. Iteration 1: residuals (left) and the weak regressor trained over the residuals (right).

The current ensemble consists of only one regression tree:  $F = \alpha_1 h_1$ . We can also visualize the classification predictions of  $h_1$  and the ensemble  $F$ . The resulting classifications achieve an overall error rate of 16%, as shown in the following figure.

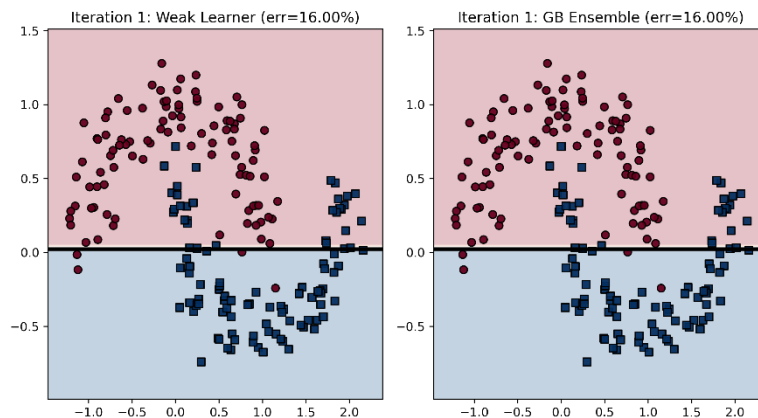


Figure 5.12. Iteration 1: Predictions of the weak learner ( $h_1$ ) and the whole ensemble ( $F$ ). Since this is the first iteration, the ensemble consists of only one weak regressor.

In iteration 2, we compute the residuals again. Now, the residuals begin to show more separation, which reflects how well they are classified by the current ensemble. The decision tree regressor attempts to fit the residuals again (Figure 5.13, right), though this time, it focuses on examples that have been misclassified previously.

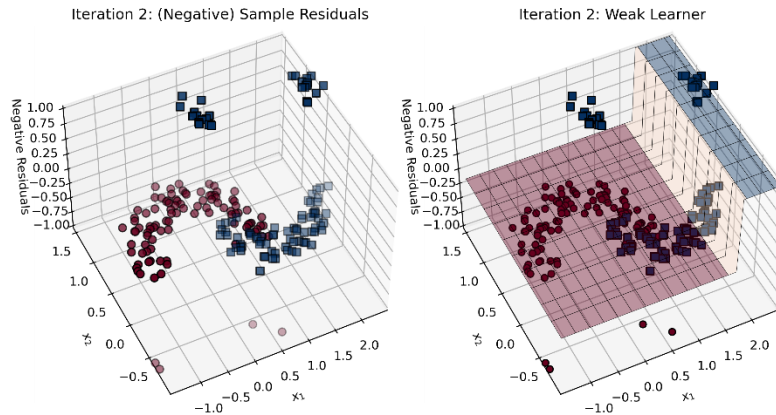


Figure 5.13. Iteration 2: residuals (left) and the weak regressor trained over the residuals (right).

The ensemble now consists of two regression trees:  $F = \alpha_1 h_1 + \alpha_2 h_2$ . We can now visualize the classification predictions of the newly trained regressor  $h_2$  and the overall ensemble  $F$ .

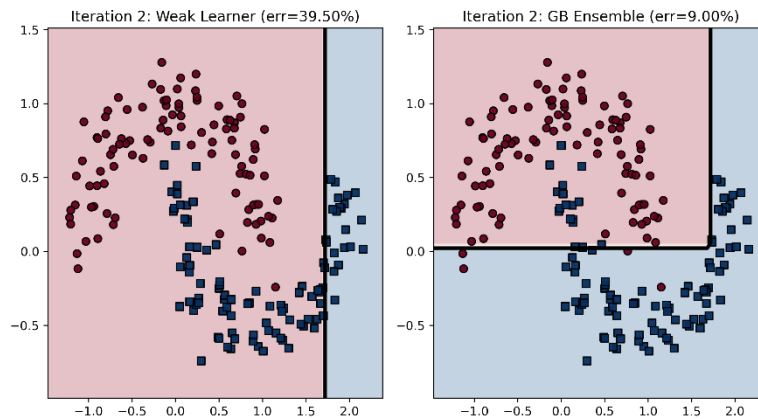


Figure 5.14. Iteration 2: Predictions of the weak learner ( $h_2$ ) and the overall ensemble ( $F$ ).

The weak learner trained in iteration 2 has an overall error rate of 39.5%. Yet the first two weak learners have already boosted the ensemble performance up to 91% accuracy, that is 9% error. This process continues in iteration 3.

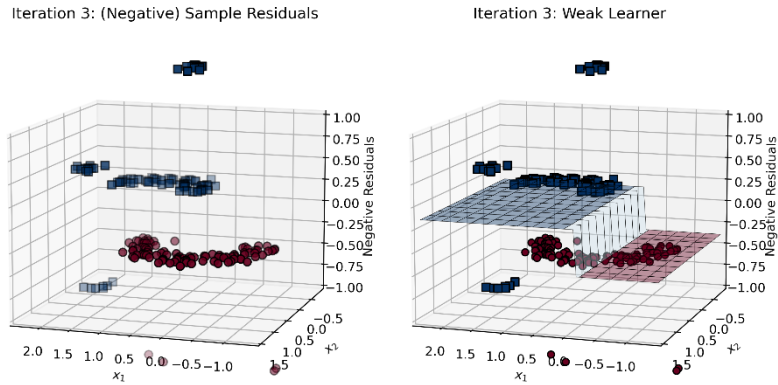


Figure 5.15. Iteration 3: residuals (left) and the weak regressor trained over the residuals (right).

In this manner, gradient boosting continues to sequentially train and add base regressors to the ensemble. The figure below shows the model learned after 10 iterations; the ensemble consists of 10 weak regressor estimators and has boosted overall training accuracy to 97.5%!

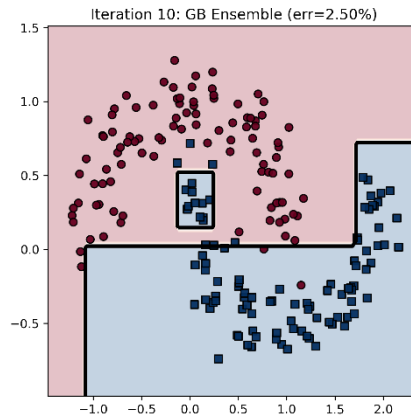


Figure 5.16. Final gradient boosting ensemble after 10 iterations.

There are several publicly available and efficient implementations of gradient boosting that you can use for your machine-learning tasks. For the rest of this section, we will focus on the most familiar: `scikit-learn`.

### 5.2.3 Gradient Boosting with `scikit-learn`

We will now look at how to use two `scikit-learn` classes: `GradientBoostingClassifier` and a new (currently experimental) version called `HistogramGradientBoostingClassifier`, which

trade exactness for speed to train models significantly faster than `GradientBoostingClassifier`. This makes it ideally suited for larger data sets.

`scikit-learn`'s `GradientBoostingClassifier` essentially implements the same gradient-boosting algorithm that we have ourselves implemented in this section. Its usage is similar to other `scikit-learn` classifiers such as `AdaBoostClassifier`. There are two key differences from `AdaBoostClassifier`, however:

- Unlike `AdaBoostClassifier`, which supports several different types of base estimators, `GradientBoostingClassifier` only supports tree-based ensembles. This means that it will always use decision trees as base estimators, and there is no mechanism to specify other types of base learning algorithms.
- `AdaBoostClassifier` optimizes the exponential loss (by design). `GradientBoostingClassifier` allows the user to select either the logistic or exponential loss functions. The logistic loss (also known as cross entropy) is a commonly used loss function for binary classification (and also has a multi-class variant).

**NOTE** Training a `GradientBoostingClassifier` with the exponential loss is essentially equivalent to training an `AdaBoostClassifier`.

In addition to selecting the loss function, we can also set additional learning parameters. These parameters are often selected by cross validation, much like any other machine-learning algorithm (see Chapter 4.3 for parameter selection in `AdaBoostClassifier`).

- We directly can control the complexity of the base tree estimators with `max_depth` and `max_leaf_nodes`. Higher values mean that the base tree-learning algorithm has greater flexibility in training more complex trees. The caveat here, of course, is that deeper trees or trees with more leaf nodes tend to overfit the training data.
- `n_estimators` caps the number of weak learners that will be trained sequentially by `GradientBoostingClassifier` and is essentially the number of algorithm iterations.
- Like AdaBoost, gradient boosting also trains weak learners ( $h_t$  in iteration  $t$ ) sequentially and constructs an ensemble incrementally and additively:  $F_t(x) = F_{t-1}(x) + \eta \cdot \alpha_t \cdot h_t(x)$ . Here,  $\alpha_t$  is the weight of weak learner  $h_t$  (or the step length) and  $\eta$  is the learning rate. The learning rate is a user-defined learning parameter that lies in the range  $0 < \eta \leq 1$ . Recall that a slower learning rate means that it will often take more iterations to train an ensemble. It may be necessary to opt for slower learning rates in order to make successive weak learners more robust to outliers and noise. Learning rate is controlled by the `learning_rate` parameter.

Let's look at an example of gradient boosting in action on the breast cancer data set. We train and evaluate a `GradientBoostingClassifier` model using the breast cancer data set.

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
X, y = load_breast_cancer(return_X_y=True)
Xtrn, Xtst, ytrn, ytst = train_test_split(X, y, test_size=0.25) #A

from sklearn.ensemble import GradientBoostingClassifier
```

```
ensemble = GradientBoostingClassifier(max_depth=1, #B
                                     n_estimators=20,
                                     learning_rate=0.75)
ensemble.fit(Xtrn, ytrn)
```

```
#A load the data set and split into training and test sets
#B train a GB model with these learning parameters
```

And how well did this model do?

```
ypred = ensemble.predict(Xtst)
err = 1 - accuracy_score(ytst, ypred)
print(err)
0.020979020979020935
```

This gradient boosting classifier achieves 2.1% test error, which is pretty good. A key limitation of `GradientBoostingClassifier`, however, is its speed; while effective, it does tend to be rather slow on large data sets.

The efficiency bottleneck, as it turns out, is in tree learning. Recall that gradient boosting has to learn a regression tree at each iteration as a base estimator. For large data sets, the number of splits a tree-learner has to consider becomes prohibitively large.

This has led to the emergence of histogram-based gradient boosting, which aims to speed up base estimator tree learning, allowing gradient boosting to scale up to large data sets.

## 5.2.4 Histogram-based Gradient Boosting

To understand the need for histogram-based tree learning, we have to revisit how a decision tree algorithm learns a regression tree. In tree-learning, we learn a tree in a top-down fashion, one decision node at a time.

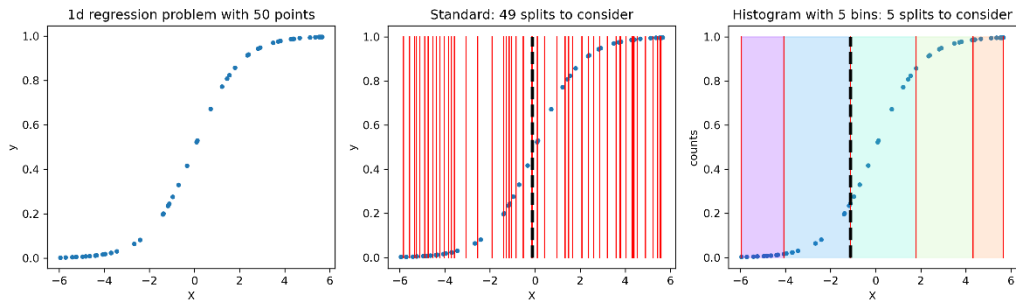
The standard way to do this is by pre-sorting the feature values, enumerating all possible splits and then evaluating all of them in order to find the best split. Let's say we have 1 million ( $10^6$ ) training examples each of dimension **100**.

Standard tree learning will enumerate and evaluate 100 million splits ( $10^6 \times 100 \times 10^8$ ) to identify a decision node! This is clearly untenable.

One alternative is to re-organize the feature values into a small number of bins. In the hypothetical example above, suppose we binned each feature column into 100 bins. Now, to find the best split, we have to only search over 10 thousand splits ( $100 \times 100 \times 10^4$ ), which can speed up training rather dramatically!

Of course, this means that we are trading-off exactness for speed. However, it is usually the case that there is a large amount of redundancy or repeated information in many (big) data sets, which we "compress" by binning the data into smaller buckets. The figure below illustrates this tradeoff.





**Figure 5.17.** (left) A simple 1-dimensional regression problem with 50 data points. (center) Standard tree learning evaluates every possible split, which is illustrated a line between each pair of data points. The best split is the one with the lowest split criterion (here, squared loss). (right) Histogram-based binning first puts the data into 5 buckets, and then evaluates the splits between each pair of data buckets. Again, the best split is the one with the lowest split criterion (also squared loss).

In the figure above, we contrast the behaviors of standard decision tree learning and histogram-based tree learning. In standard tree learning, each split considered is between two successive data points (figure center); for 50 data points, we have to evaluate 49 splits.

In histogram-based splitting, we first bin the data (figure right, into 5 bins). Now, each split considered is between two successive data buckets; for 5 bins, we only have to evaluate 4 splits! Now imagine how this would scale to millions of data points.

`scikit-learn` 0.21 introduced an experimental version of gradient boosting called `HistogramGradientBoostingClassifier` that implements histogram-based gradient boosting such that its training time is significantly improved.

The snippet below shows how to train and evaluate `Histogram GradientBoostingClassifier` on the breast cancer data set.

```
from sklearn.experimental import enable_hist_gradient_boosting
from sklearn.ensemble import HistGradientBoostingClassifier #A

ensemble = HistGradientBoostingClassifier(max_depth=2,
                                         max_iter=20,
                                         learning_rate=0.75)

ensemble.fit(Xtrn, ytrn) #B

ypred = ensemble.predict(Xtst)
err = 1 - accuracy_score(ytst, ypred)
print(err)
0.013986013986013957
```

**#A** enable experimental mode (this is current as of v.0.21)

**#B** train a histogram-based GB model with these learning parameters

On the breast cancer data set, `HistGradientBoostingClassifier` achieved a test error of 1.4%. `scikit-learn`'s histogram-based boosting implementation itself is inspired by another popular gradient boosting package: `LightGBM`.

## 5.3 LightGBM: A Framework for Gradient Boosting

LightGBM, or Light Gradient Boosted Machines, is an open source gradient boosting framework that was originally developed and released by Microsoft.

At its core, LightGBM is essentially a histogram-based gradient boosting approach. However, it also has several modeling and algorithmic features that enable it to handle large-scale data. In particular, LightGBM offers the following advantages:

- Algorithmic speedups such as gradient based one-sided sampling and exclusive feature bundling that result in faster training and lower memory usage; these are described in more detail in Section 5.3.1;
- Support for a large number of loss functions for classification, regression and ranking as well as application-specific custom loss functions (Section 5.3.2);
- Support for parallel and GPU learning, which enables it handle large-scale data sets (parallel/GPU-based machine learning is out-of-scope for this book).

We will also delve into how to apply LightGBM to some practical learning situations to avoid overfitting (Section 5.4.1), and ultimately a case study on a real-world data set (Section 5.4).

It will be impossible to detail all the features available in LightGBM in this limited space, of course. Instead, this section and the next introduce LightGBM and illustrate its usage and applications in practical settings. This should enable readers to springboard further into advanced use cases of LightGBM for their applications through its documentation.

### 5.3.1 What Makes LightGBM “Light”?

Recall from our earlier discussion that the biggest computational bottleneck in scaling gradient boosting to large (with many training examples) or high-dimensional (with many features) data sets is tree learning, specifically, identifying optimal splits in the regression tree base estimators.

As we saw in the previous section, histogram-based gradient boosting attempts to address this computational bottleneck. This works reasonably well for medium-sized data sets. However, histogram-bin construction can itself be slow if we have a very large number of data points or a large number of features, or both.

In this section, we will look at two key conceptual improvements that LightGBM implements that often lead to significant speedups in training times in practice. The first, Gradient-based One-Side Sampling (GOSS), aims to reduce the number of training examples, while the second, Exclusive Feature Bundling (EFB), aims to reduce the number of features.

#### GRADIENT-BASED ONE-SIDE SAMPLING (GOSS)

A well-known approach to dealing with a very large number of training examples is to downsample the data set, that is, randomly sample a smaller subset of the data set. We have already seen examples of this in other ensemble approaches such as pasting (which is bagging without replacement, see Chapter 2).

There are two problems with randomly downsampling the data set. First, not all examples are equally important; as in AdaBoost, some training examples are more important than others depending on the extent of their misclassification. Thus, it is imperative that downsampling not throw away high-importance training examples.

Second, sampling should also ensure that some fraction of correctly classified examples is also included. This is important in order to not overwhelm the base learning algorithm with just misclassified examples, which will inevitably lead it to overfit.

This is addressed by downsampling the data smartly using a procedure called Gradient-based One-Side Sampling or GOSS. Briefly, GOSS performs the following steps:

1. Similar to AdaBoost that uses sample weights, GOSS uses the gradient magnitude. Remember that the gradient indicates how much more the prediction can be improved: well-trained examples have small gradients, while under-trained (typically, misclassified or confusing) examples have large gradients.
2. Select the top  $a\%$  of examples with the largest gradients; call this subset  $top$
3. Randomly sample  $b\%$  of the remaining examples; call this subset  $rand$
4. Assign weights to examples in both sets:  $w^{top} = 1, w^{rand} = \frac{100-a}{b}$ ,
5. Train a base regressor over this sampled data: (data, -gradients,  $\mathbf{w}$ )

The weights computed in the Step 4 ensure that there is a good balance between undertrained and well-trained samples. Overall, such sampling also fosters ensemble diversity, which ultimately leads to better ensembles.

#### EXCLUSIVE FEATURE BUNDLING (EFB)

Aside from a large number of training examples, big data also often provides the challenge of very high dimensionality, which can adversely affect histogram construction and slow down the overall training process.

Similar to downsampling training examples, if we are able to downsample the features as well, it is possible to gain (sometimes very big) improvements in training speed. This is especially so when feature space is sparse, and features are mutually exclusive.

One common example of such a feature space is when we apply one-hot vectorization to categorical variables. For instance, consider a categorical variable that takes 10 unique values. When one-hot vectorized, this variable is expanded to 10 binary variables, of which only one is non-zero and all others are zero. This makes the 10 columns corresponding to this feature highly sparse.

Exclusive feature bundling (EFB) exploits this sparsity and aims to merge mutually exclusive columns into one column to reduce the number of effective features. At a high-level, EFB performs two steps:

1. Identify features that can be bundled together by measuring conflicts or the number of times both features are *non-zero*. The intuition here is that if two features are often simultaneously zero, they are low conflict and can be bundled together.
2. Merge the identified low conflict features into a feature bundle. The idea here is to preserve information carefully when merging non-zero values, which is typically done by adding offsets to feature values to prevent overlaps.

By merging features in this manner, EFB effectively reduces the overall number of features base estimator algorithms have to consider, which often makes training much faster.

### 5.3.2 Gradient Boosting with LightGBM

LightGBM is available for various platforms including Windows, Linux and MacOS, and can either be built from scratch or installed using tools such as `pip`. Its usage syntax is quite similar to `scikit-learn`'s.

Continuing with the breast cancer data set from Section 5.2.3, we can learn a gradient boosting model using LightGBM as follows:

```
from lightgbm import LGBMClassifier
gbm = LGBMClassifier(boosting_type='gbdt', n_estimators=20, max_depth=1)
gbm.fit(Xtrn, ytrn)
```

Here, we instantiate an instance of `LGBMClassifier` and set it to train an ensemble of 20 regression stumps (that is, the base estimators will be regression trees of depth 1). The other important specification here is `boosting_type`. LightGBM can be trained in four modes:

- `boosting_type='rf'` trains traditional random forest ensembles (see Section 2.4)
- `boosting_type='gbdt'` trains an ensemble using traditional gradient boosting (see Section 5.2)
- `boosting_type='goss'` trains an ensemble using Gradient One-Side Sampling (GOSS, see Section 5.3.1)
- `boosting_type='dart'` trains an ensemble using DART, or Dropout meets Multiple Additive Regression Trees (which will be described in Section 5.4)

The last three gradient boosting modes essentially tradeoff between training speed and predictive performance, and we will explore this in our case study. For now, how well does the model we just trained using `boosting_type='gbdt'` do?

```
from sklearn.metrics import accuracy_score
ypred = gbm.predict(Xtst)
accuracy_score(ytst, ypred)
0.9473684210526315
```

Our first LightGBM classifier achieves 94.7% accuracy on the test set held out from the breast cancer data set. Now that we've familiarized ourselves with the basic functionality of LightGBM, let us look at how we can train models for real-world use cases with LightGBM.

## 5.4 LightGBM in Practice

In this section, we describe how to train models in practice using LightGBM. As always, this means ensuring that LightGBM models do not overfit and generalize well. As with AdaBoost, we look to set the learning rate (Section 5.4.1) or employ early stopping (Section 5.4.2) as a means to control overfitting. Specifically,

- by selecting an effective learning rate, we try to control the rate at which the model learns so that it doesn't rapidly fit, and then overfit the training data. We can think of this a proactive modeling approach, where we try to identify a good training strategy so that it leads to a good model.
- by enforcing early stopping, we try to stop training as soon as we observe that the model is starting to overfit. We can think of this as a reactive modeling approach, where we

contemplate terminating training as soon as we think we have a good model.

Finally, we also explore one of the most powerful functionalities of LightGBM: its support for custom loss functions. Recall that one of the major benefits of gradient boosting is that it is a general procedure, widely applicable to many loss functions.

While LightGBM provides support for many standard loss functions for classification, regression and ranking, sometimes it may be necessary to train with application-specific loss functions. In Section 5.4.3, we will see precisely how we can do this with LightGBM.

### 5.4.1 Learning Rate

When using gradient boosting, as with other machine-learning algorithms, it is possible to overfit on the training data. This means that, while we achieve very good training set performance, this doesn't result in a similar test set performance. That is, the model we've trained fails to generalize well. LightGBM, like `scikit-learn`, provides us with the means to control model complexity before overfitting.

#### LEARNING RATE VIA CROSS VALIDATION

LightGBM allows us to control the learning rate through the `learning_rate` training parameter (a positive number that has a default value of 0.1).

This parameter also has a couple of aliases: `shrinkage_rate` and `eta`, which are other terms for the learning rate commonly used in machine-learning literature. Though all of these parameters have the same effect, care must be taken to set only one of them.

How can we figure out an effective learning rate for our problem? As with any other learning parameter, we can use cross validation. Recall that we also used cross validation to select the learning rate for AdaBoost in the previous chapter.

LightGBM plays nicely with `scikit-learn`, and we can combine the relevant functionalities from both packages to perform effective model learning.

In the listing below, we combine `scikit-learn`'s `StratifiedKFold` class to split the training data into 10 folds of training and validation sets. `StratifiedKFold` ensures that we preserve class distributions, that is, the fractions of different classes across the folds.

Once the cross validation folds are set up, we can train and validate models on these 10 folds for different choices of learning rates: 0.1, 0.2, ..., 1.0.

#### Listing 5.4. Cross Validation with LightGBM and `scikit-learn`

```
from sklearn.model_selection import StratifiedKFold
import numpy as np

n_learning_rate_steps, n_folds = 10, 10 #A
learning_rates = np.linspace(0.1, 1.0, num=n_learning_rate_steps)

splitter = StratifiedKFold(n_splits=n_folds, shuffle=True, random_state=42) #B

trn_err = np.zeros((n_learning_rate_steps, n_folds)) #C
val_err = np.zeros((n_learning_rate_steps, n_folds)) #C
```

```

for i, rate in enumerate(learning_rates): #D
    for j, (trn, val) in enumerate(splitter.split(X, y)):
        gbm = LGBMClassifier(boosting_type='gbdt', n_estimators=10,
                             max_depth=1, learning_rate=rate)
        gbm.fit(X[trn, :], y[trn])
        trn_err[i, j] = (1 - accuracy_score(y[trn], #D
                                           gbm.predict(X[trn, :]))) * 100
        val_err[i, j] = (1 - accuracy_score(y[val],
                                           gbm.predict(X[val, :]))) * 100

trn_err = np.mean(trn_err, axis=1) #E
val_err = np.mean(val_err, axis=1) #E

```

```

#A initialize learning rates & num. CV folds
#B split data into training and validation folds
#C to save training & validation errors
#D Train a LightGBM classifier for each fold with different learning rates
#E save training and validation errors
#F average training & validation
errors across folds

```

We can visualize the training and validation errors for different learning rates as shown below.

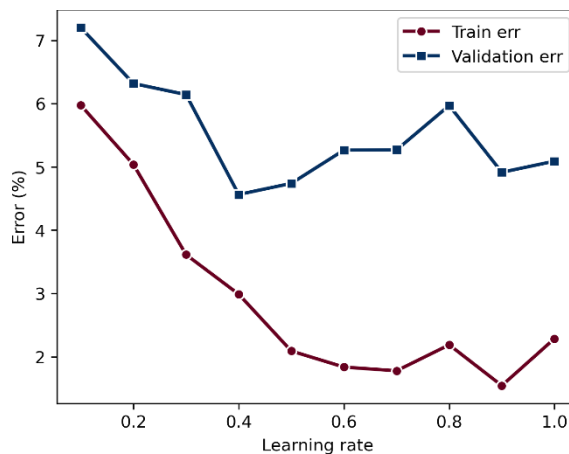


Figure 5.18. Averaged training and validation errors of LightGBM across 10 folds of the breast cancer data set.

Unsurprisingly, as the learning rate increases, the training error continues to decrease, suggesting that the model first fits and then begins to overfit the training data. The validation error does not show the same trend.

It decreases initially, and then increases; a learning rate of 0.4 produces the lowest validation error. This, then, is the best choice of learning rate.

#### CROSS VALIDATION WITH LIGHTGBM

LightGBM provides its own functionality to perform cross validation (CV) with given parameter choices through a function called `cv`.

**Listing 5.5. Cross Validation with LightGBM**

```

from lightgbm import cv, Dataset

trn_data = Dataset(Xtrn, label=ytrn) #A
params = {'boosting_type': 'gbdt', 'objective': 'cross_entropy'
          'learning_rate': 0.25, #B
          'max_depth': 1}
cv_results = cv(params, trn_data,
                num_boost_round=100,
                nfold=5,
                stratified=True, shuffle=True)

```

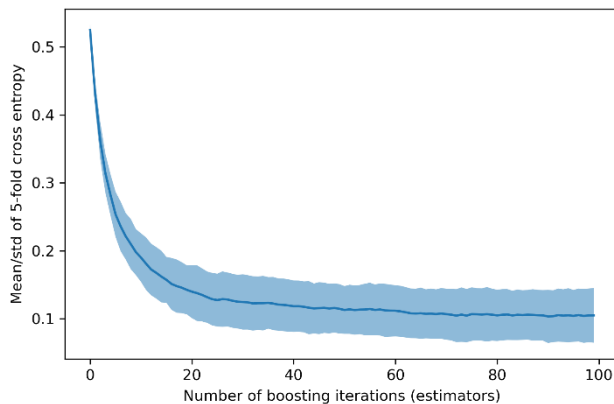
**#A** put data into a LightGBM Dataset object

**#B** specify learning parameters

**#C** perform 5-fold CV, each with 100 estimators

In the listing above, we perform 5-fold CV over 100 boosting rounds (thus eventually training 100 base estimators). Setting `stratified=True` ensures that we preserve class distributions, that is, the fractions of different classes across the folds. Setting `shuffle=True` randomly shuffles the training data before splitting into folds.

We can visualize the training objective as training progresses. In the listing above, we train our classification model by optimizing cross entropy, set via `'objective': 'cross_entropy'`. This is shown in the figure below: as we add more base estimators to our sequential ensemble, the average 5-fold cross entropy objective decreases.



**Figure 5.19.** The average cross entropy across the folds decreases with increasing iterations, as we add more base estimators to the ensemble.

### 5.4.2 Early Stopping

Another way of reining in overfitting behavior is through early stopping. As we've seen with AdaBoost, the idea of early stopping is pretty straightforward. As we train sequential ensembles, we train one base estimator at each iteration.

This process continues until we reach the user-specified ensemble size (in LightGBM, there are several aliases to specify this: `n_estimators`, `num_trees`, `num_rounds`).

As the number of base estimators in the ensemble increases, the complexity of the ensemble also increases, which eventually leads to overfitting. To avoid this, what if, instead of training the model, we stopped before we reached the limit of ensemble size?

This is precisely early stopping. We keep track of overfitting behavior by means of a validation set. Then, we train until we see no improvement in validation performance for a certain pre-specified number of iterations.

For example, let's say that we have started training an ensemble of 500 base estimators. We keep a close eye on the validation error as we grow our ensemble, resolving that if the validation error does not improve over a window of `m` iterations or early stopping rounds, we will terminate training.

In LightGBM, we can incorporate early stopping if we specify a value for the parameter `early_stopping_rounds`. As long as the overall validation score (say accuracy) improves over the last `early_stopping_rounds`, LightGBM will continue to train. However, if the score has not improved after `early_stopping_rounds`, LightGBM terminates.

As with AdaBoost, LightGBM also needs us to explicitly specify a validation set as well as a scoring metric for early stopping. In the listing below, we use the AUC (area under the receiver-operator curve) as the scoring metric to determine early stopping.

The AUC is an important evaluation metric for classification problems and can be interpreted as the probability that the model will rank a randomly chosen positive example higher than a randomly chosen negative example. Thus, high values of AUC are preferred as it means that the model is more discriminative.

#### Listing 5.6. Early Stopping with LightGBM

```
from sklearn.model_selection import train_test_split
Xtrn, Xval, ytrn, yval = train_test_split(X, y, test_size=0.2, #A
                                       shuffle=True, random_state=42)

gbm = LGBMClassifier(boosting_type='gbdt', n_estimators=50,
                    max_depth=1, early_stopping=5) #B

gbm.fit(Xtrn, ytrn, eval_set=[(Xval, yval)], eval_metric='auc') #C
```

```
#A split data into train & validation sets
#B early stopping if no change in val. score after 5 rounds
#C use AUC as the validation scoring metric for early stopping
```

Let's look at the output produced by LightGBM. In the listing above, we set `n_estimators=50`, which means training will add one base estimator per iteration.

```
Training until validation scores don't improve for 5 rounds
[1] valid_0's auc: 0.885522    valid_0's binary_logloss: 0.602321
[2] valid_0's auc: 0.961022    valid_0's binary_logloss: 0.542925
...
[27] valid_0's auc: 0.996069    valid_0's binary_logloss: 0.156152
[28] valid_0's auc: 0.996069    valid_0's binary_logloss: 0.153942
[29] valid_0's auc: 0.996069    valid_0's binary_logloss: 0.15031
[30] valid_0's auc: 0.996069    valid_0's binary_logloss: 0.145113
```



```

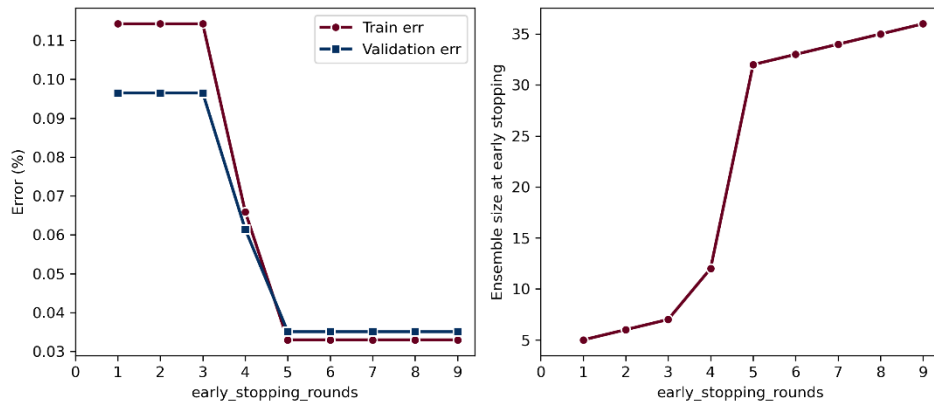
[31] valid_0's auc: 0.995742    valid_0's binary_logloss: 0.143901
[32] valid_0's auc: 0.996069    valid_0's binary_logloss: 0.139801
Early stopping, best iteration is:
[27] valid_0's auc: 0.996069    valid_0's binary_logloss: 0.156152

```

First, observe that training terminates after 32 iterations, meaning that LightGBM did indeed terminate before going all the way to training a full set of 50 base estimators. Next, note that the best iteration was 27, which had a score (in this case, AUC) of 0.996069.

Over the next 5 (`early_stopping_rounds`) iterations, from 28 to 32, LightGBM observed that adding additional estimators did not improve the validation score significantly. This triggers the early stopping criterion, causing LightGBM to terminate and return an ensemble with 32 base estimators.

We also visualize the training and validation errors as well as the ensemble size for different choices of `early_stopping_rounds`.



**Figure 5.20.** (left) Training and validation errors for different values of `early_stopping_rounds`. (right) Ensemble sizes for different values of `early_stopping_rounds`.

Small values of `early_stopping_rounds` make LightGBM very "impatient" and aggressive in that it does not wait too long to see if there is any improvement before stopping learning early. This leads to underfitting; in the figure above, for instance, setting `early_stopping_rounds` to 1 leads to an ensemble of just 5 base estimators, hardly enough to even fit the training data properly!

Large values of `early_stopping_rounds` make LightGBM too passive in that it waits for longer periods to see if there is any improvement. The choice of `early_stopping_rounds` ultimately depends on your problem: how big it is, what your performance metric is, and the complexity of the models you are willing to tolerate.

### 5.4.3 Custom Loss Functions

Recall that one of the most powerful features of gradient boosting is that it is applicable to a wide variety of loss functions. This means that it is also possible for us to design our own, problem-specific loss functions to handle specific properties of our data set and task.

Perhaps our data set is imbalanced, meaning that different classes have different amounts of data. In such situations, rather than high accuracy, we might require high recall (fewer false negatives, for example, in medical diagnosis) or high precision (fewer false positives, for example, in spam detection). In many such scenarios, it is often necessary to design our own problem-specific loss functions.

**NOTE** For more details on evaluation metrics such as precision and recall, as well metrics for other machine learning tasks such as regression and ranking, see *Evaluating Machine Learning Models* by Alice Zheng (O'Reilly, 2015).

With gradient boosting generally, and LightGBM specifically, once we have a loss function, we can rapidly train and evaluate models that are targeted towards our problem. In this section, we will explore how to use LightGBM for a custom loss function called the *focal loss*.

#### THE FOCAL LOSS

The focal loss was introduced for dense object detection, or the problem of object detection at a large number of densely packed windows in an image. Ultimately, such object detection tasks come down to a foreground vs. background classification problem, which is highly imbalanced as there are often many more windows with background than foreground objects of interest.

The focal loss, in general, was designed for, and well-suited for classification problems with such class imbalances. It is a modification of the classical cross-entropy loss that puts more focus on harder-to-classify examples, while ignoring the easier examples.

More formally, recall that the standard cross entropy loss between a true label and a predicted label can be computed as

$$L_{ce}(y_{true}, y_{pred}) = y_{true} \log(p_{pred}) - (1 - y_{true}) \log(1 - p_{pred})$$

where  $p_{pred}$  is the probability of positive prediction, that is,  $\text{prob}(y_{pred} = 1) = p_{pred}$ . Note that, for a binary classification problem, since the only other label is  $y = 0$ , the probability of negative prediction will be  $\text{prob}(y_{pred} = 0) = 1 - p_{pred}$ .

The focal loss introduces a *modulating factor* to each term in the cross-entropy loss

$$L_{fo}(y_{true}, y_{pred}) = y_{true} \log(p_{pred}) \cdot (1 - p_{pred})^\gamma - (1 - y_{true}) \log(1 - p_{pred}) \cdot (p_{pred})^\gamma$$

The modulating factor suppresses the contribution of well-classified examples, forcing a learning algorithm to focus on poorly classified examples. The extent of this "focus" is determined by a user-controllable parameter,  $\gamma > 0$ . To see how modulation works, let's compare the cross-entropy loss with the focal loss with  $\gamma = 2$ :

- *well-classified example*: let's say the true label:  $y_{true} = 1$ , with high predicted label probability,

$p_{pred} = 0.95$ . The cross-entropy loss is  $L_{ce} = -1 \log 0.95 - 0 \log 0.05 = 0.0513$ , while the focal loss is  $L_{fo} = -1 \log 0.95 \cdot 0.05^2 - 0 \log 0.05 \cdot 0.95^2 = 0.0001$ . The modulating factor in the focal loss, thus, downweights the loss if an example is well-classified.

- *poorly classified example*: let's say the true label:  $y_{true} = 1$ , with low predicted label probability,  $p_{pred} = 0.05$ . The cross-entropy loss is  $L_{ce} = -1 \log 0.05 - 0 \log 0.95 = 2.9957$ , while the focal loss is  $L_{fo} = -1 \log 0.05 \cdot 0.95^2 - 0 \log 0.95 \cdot 0.05^2 = 2.7036$ . The modulating factor affects the loss for this example far less as it is poorly classified.

This effect can be seen below, where the focal loss is plotted for different values of  $\gamma$ . For bigger values of  $\gamma$ , well-classified examples (with high probability of  $y=1$ ) have lower losses, while poorly classified examples have higher losses.

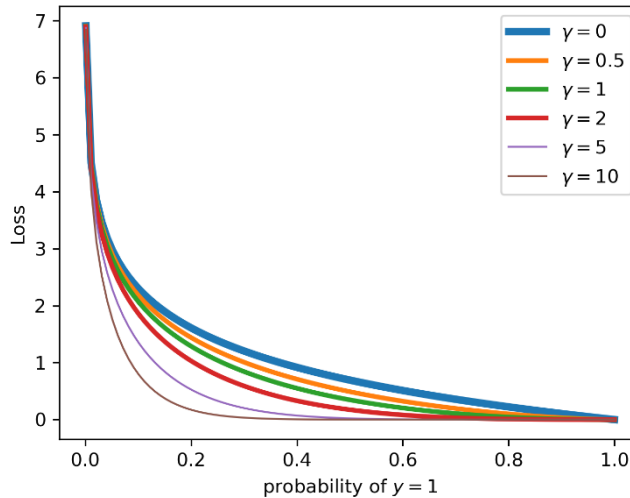


Figure 5.21. The focal loss visualized for various values of  $\gamma$ . When  $\gamma = 0$ , the original cross-entropy loss is recovered. As  $\gamma$  increases, the part of the curve corresponding to "well-classified" examples becomes longer, reflecting the loss function's focus on poor classification.

### GRADIENT BOOSTING WITH THE FOCAL LOSS

In order to use the focal loss to train gradient boosted decision trees, we have to provide LightGBM with two functions:

- the actual loss function itself, which will be used for function evaluations and scoring during learning and,
- the first derivative (gradient) and second derivative (Hessian) of the loss function, which will be used for learning the constituent base estimator trees.

LightGBM uses the Hessian information for learning at leaf nodes. For the moment, let us put this small detail out of our mind, and revisit it in the next chapter.

The listing below shows how we can define custom loss functions. The function, `focal_loss` is the loss itself, defined as described above. The function, `focal_loss_metric`, turns `focal_loss` into a scoring metric for use with LightGBM.

The function `focal_loss_objective` returns the gradient and the Hessian of the loss function for LightGBM to use in tree learning. This function is rather unintuitively suffixed with `'objective'` to be consistent with LightGBM's usage, as will become apparent below.

### Listing 5.7. Defining Custom Loss Functions

```
from scipy.misc import derivative

def focal_loss(ytrue, ypred, gamma=2.0): #A
    p = 1 / (1 + np.exp(-ypred))
    loss = -(1 - ytrue) * p**gamma * np.log(1 - p)
           - ytrue * (1 - p)**gamma * np.log(p)
    return loss

def focal_loss_metric(ytrue, ypred): #B
    return 'focal_loss_metric', np.mean(focal_loss(ytrue, ypred)), False

def focal_loss_objective(ytrue, ypred):
    func = lambda z: focal_loss(ytrue, z)
    grad = derivative(func, ypred, n=1, dx=1e-6) #C
    hess = derivative(func, ypred, n=2, dx=1e-6) #C
    return grad, hess
```

#A define the focal loss function

#B wrapper function that returns a LightGBM-compatible scoring metric

#C auto-differentiation is used to compute gradient & Hessian

Care must be taken to ensure that the loss function, metric and objective are all vector-compatible, that is, that they can take array-like objects `ytrue` and `ypred` as inputs.

In Listing 5.7, we have used `scipy`'s `derivative` functionality to approximate the first and second derivatives. It is also possible to explicitly analytically derive and implement the first and second derivatives for some loss functions. Once we have defined our custom loss function, it is straightforward to use it with LightGBM:

```
gbm_focal_loss = LGBMClassifier(objective=focal_loss_objective, #A
                               learning_rate=0.25,
                               n_estimators=20, max_depth=1)

gbm_focal_loss.fit(Xtrn, ytrn,
                  eval_set=[(Xval, yval)], eval_metric=focal_loss_metric) #B

accuracy_score(yval, gbm_focal_loss.predict(Xval))
0.9736842105263158
```

#A set objective to ensure that LightGBM uses the gradients of the focal loss for learning

#B set metric to ensure that LightGBM uses the focal loss for evaluation

GBDT with focal loss achieves a validation score of 97.37% on the breast cancer data set.

## 5.5 Case Study: Document Retrieval

Document retrieval is the task of retrieving documents from a database to match a user’s query. For example, a paralegal at a law firm might need to search for information about previous cases from legal archives in order to establish precedent and research case law.

Or perhaps a graduate student might need to search for articles from a journal’s database during the course of a literature survey of work in a specific area.

You may also have seen a feature called “related articles” on many websites that list articles that may be related to the article you’re currently reading. There are many such use cases for document retrieval in a wide range of domains, where a user searches for specific terms and the system must return a list of documents relevant to the search.

This challenging problem has two key components: first, find the documents that match the user’s query, and second, rank the documents according to some notion of “relevance” to present to the user.

In this case study, the problem is set up as a 3-class classification problem of identifying the relevance rank/class (least, moderately or highly relevant) given a query-document pair. We explore the performance of different LightGBM classifiers for this task.

### 5.5.1 The LETOR Data Set

The data set we will use for this case study is called the LEarning TO Rank (LETOR) ver. 4.0, which was itself created from a large corpus of webpages called GOV2. The GOV2<sup>1</sup> data set is a collection of about 25 million webpages extracted from the .gov domain.

The LETOR 4.0<sup>2</sup> data collection is derived from the GOV2 corpus and is made freely available by Microsoft Research. The collection contains several data sets, and we will use the data set that was originally developed for the Million Query track of the 2008 Text Retrieval Conference (TREC), specifically, `MQ2008.rar`.

Each training example in the MQ2008 data set corresponds to a query-document *pair*. The data itself is in LIBSVM format, and several examples are shown below. Each row in the data set is a labeled training example in the format:

```
<relevance label> qid:<query id> 1:<feature 1 value> 2:<feature 2 value> 3:<feature 3 value> ...
46:<feature 46 value> # meta-information
```

Every example has 46 features extracted from a query-document pair, and a relevance label. The features include:

- low-level content features extracted from the body, anchor, title, and URL; these include features commonly used in text mining such as term frequency, inverse document frequency, document length and various combinations,
- high-level content features extracted from the body, anchor and title; these features are extracted using two well-known retrieval systems: Okapi BM25 and language-model approaches for information retrieval (LMIR),
- hyperlink features extracted from hyperlinks using several different tools such as Google PageRank and variations,

<sup>1</sup>[http://ir.dcs.gla.ac.uk/test\\_collections/access\\_to\\_data.html](http://ir.dcs.gla.ac.uk/test_collections/access_to_data.html)

<sup>2</sup><https://www.microsoft.com/en-us/research/project/letor-learning-rank-information-retrieval/#!letor-4-0>

- hybrid features containing both content and hyperlink information.

The label for each query-document example is a relevance rank that takes 3 unique values: 0 (least relevant), 1 (moderately relevant), 2 (highly relevant). In our case study, these are treated as class labels, making this an instance of a 3-class classification problem. Some examples of the data are shown below.

```
0 qid:10032 1:0.130742 2:0.000000 3:0.333333 4:0.000000 5:0.134276 ... 45:0.750000 46:1.000000
  #docid = GX140-98-13566007 inc = 1 prob = 0.0701303

1 qid:10032 1:0.593640 2:1.000000 3:0.000000 4:0.000000 5:0.600707 ... 45:0.500000 46:0.000000
  #docid = GX256-43-0740276 inc = 0.0136292023050293 prob = 0.400738

2 qid:10032 1:0.056537 2:0.000000 3:0.666667 4:1.000000 5:0.067138 ... 45:0.000000 46:0.076923
  #docid = GX029-35-5894638 inc = 0.0119881192468859 prob = 0.139842
```

Much more detail can be found in the documentation and references provided with the LETOR 4.0 data collection. We first load this data set and split into training and test sets:

```
from sklearn.datasets import load_svmlight_file
from sklearn.model_selection import train_test_split

query_data_file = './data/ch05/MQ2008/Querylevelnorm.txt'
X, y = load_svmlight_file(query_data_file)

Xtrn, Xtst, ytrn, ytst = train_test_split(X, y,
                                         test_size=0.2, random_state=42)

print(Xtrn.shape, Xtst.shape)
(12168, 46) (3043, 46)
```

We now have a training set with 12K examples and a test set with 3K examples.

## 5.5.2 Document Retrieval with LightGBM

We will learn four models using LightGBM. Each of these models represents a tradeoff between speed and accuracy:

- random forest: our now familiar parallel homogeneous ensemble of randomized decision trees; this method will serve as a baseline approach;
- gradient boosted decision trees (GBDT): this is the standard approach to gradient boosting and represents a balance between models with good generalization performance and training speed;
- gradient boosting with gradient one-side sampling (GOSS): this variant of gradient boosting downsamples the training data and is ideally suited for large data sets; due to downsampling, it may lose out on generalization, but is typically very fast to train;
- Dropout meets Multiple Additive Regression Trees (DART): this variant incorporates the notion of dropout from deep learning, where neural units are randomly and temporarily dropped during backpropagation iterations to mitigate overfitting. Similarly, DART randomly and temporarily drops base estimators from the overall ensemble during gradient fitting iterations to mitigate overfitting. DART is often the slowest of all the gradient boosting options available in LightGBM.

We will train a model using each of these four approaches. Each of the four models share the following learning parameters. Specifically, observe that all the models are trained using the multi-class logistic loss, a generalization of the binary logistic loss function that is used in logistic regression. The number of `early_stopping_rounds` is set to 25.

```
fixed_params = {'early_stopping_rounds': 25,
                'eval_metric': 'multi_logloss',
                'eval_set': [(Xtst, ytst)],
                'eval_names': ['test set'],
                'verbose': 100}
```

Beyond these parameters that are common to all models, we will also need to identify other learning parameters such as learning rate (to control the rate of learning) or the number of leaf nodes (to control the complexity of the base estimator trees).

These parameters are selected using `scikit-learn`'s randomized cross validation module: `RandomizedSearchCV`. Specifically, we perform 5-fold cross validation over a grid of various parameter choices; however, instead of exhaustively evaluating all possible learning parameter combinations the way `GridSearchCV` does, `RandomizedSearchCV` samples a smaller number of model combinations for faster parameter selection.

```
num_random_iters = 20
num_cv_folds = 5
```

The listing below is used to train random forests using `LightGBM`

```
rf_params = {'bagging_fraction': [0.4, 0.5, 0.6, 0.7, 0.8],
             'bagging_freq': [5, 6, 7, 8],
             'num_leaves': randint(5, 50)}

ens = lgb.LGBMClassifier(boosting='rf', n_estimators=1000,
                        max_depth=-1, metric='multi_logloss',
                        random_state=42)
cv = RandomizedSearchCV(estimator=ens,
                       param_distributions=rf_params,
                       n_iter=num_random_iters,
                       cv=num_cv_folds,
                       refit=True,
                       random_state=42, verbose=True)
cv.fit(Xtrn, ytrn, **fixed_params)
```

Similarly, `LightGBM` is also trained with `boosting='gbdt'`, `boosting='goss'` and `boosting='dart'` with code similar to the listing below.

```
gbdt_params = {'num_leaves': randint(5, 50),
               'learning_rate': [0.25, 0.5, 1, 2, 4, 8, 16],
               'min_child_samples': randint(100, 500),
               'min_child_weight': [1e-2, 1e-1, 1, 1e1, 1e2],
               'subsample': uniform(loc=0.2, scale=0.8),
               'colsample_bytree': uniform(loc=0.4, scale=0.6),
               'reg_alpha': [0, 1e-1, 1, 10, 100],
               'reg_lambda': [0, 1e-1, 1, 10, 100]}

ens = lgb.LGBMClassifier(boosting='gbdt', n_estimators=1000,
                        max_depth=-1, metric='multi_logloss',
                        random_state=42)
```

```

cv = RandomizedSearchCV(estimator=ens,
                        param_distributions=gbdt_params,
                        n_iter=num_random_iters,
                        cv=num_cv_folds,
                        refit=True,
                        random_state=42, verbose=True)

cv.fit(Xtrn, ytrn, **fixed_params)

```

The CV-based learning parameter selection procedure explores several different values for the following parameters:

- `num_leaves`, which limits the number of leaf nodes and hence base estimator complexity to control overfitting,
- `min_child_samples` and `min_child_weight`, which limits each leaf node either by size or by the sum of Hessian values to control overfitting,
- `subsample` and `colsample_bytree`, which specify the fractions of training examples and features to sample from the training data respectively, to accelerate training,
- `reg_alpha` and `reg_lambda`, which specify the amount of regularization of the leaf node values, to control overfitting as well.

For each of these approaches, we are interested in looking at two performance measures: the test set accuracy and overall model development time, which includes parameter selection and training time. These are shown in the figure below. The key takeaways are:

- GOSS and GBDT perform similarly. However, GOSS runs faster than GBDT. This will be much more pronounced for increasingly larger data sets, especially those with hundreds of thousands of training examples.
- DART achieves best training performance. However, this comes at a cost: significantly increased training time. Here, for instance, DART has a running time of close to 20 minutes, compared to random forest (2 min.), GBDT and GOSS (under half a minute).
- It should be noted that LightGBM supports both multi-CPU as well as GPU processing, which may be able to significantly improve running times.



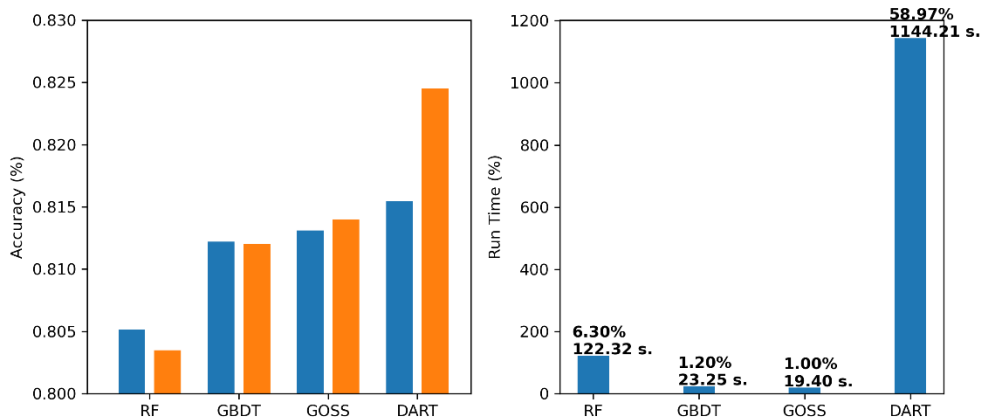


Figure 5.22. (left) Comparing test set accuracy of random forest, GBDT, GOSS and DART; (right) comparing the overall training times of random forest, GBDT, GOSS and DART. All algorithms trained using LightGBM.

## 5.6 Summary

In this chapter, we were introduced to sequential ensemble methods of weak learners.

- Gradient descent is often used to minimize a loss function to train a machine learning model.
- Residuals, or errors between the true labels and model predictions, can be used to characterize correctly classified and poorly classified training examples. This is analogous to how Unlike AdaBoost uses weights.
- Gradient boosting combines gradient descent and boosting to learn a sequential ensemble of weak learners.
- Weak learners in gradient boosting are regression trees that are trained over the residuals of the training examples and approximate the gradient.
- Gradient boosting can be applied to a wide variety of loss functions arising from classification, regression or ranking tasks.
- Histogram-based tree learning trades off exactness for efficiency, allowing us to train gradient boosting models very rapidly and scaling up to larger data sets.
- Learning can be sped up even further by smartly sampling training examples (Gradient One-Side Sampling, GOSS) or smartly bundling features (Exclusive Feature Bundling, EFB).
- LightGBM is a powerful, publicly available framework for gradient boosting that incorporates both GOSS and EFB.
- As with AdaBoost, we can avoid overfitting in gradient boosting by choosing an effective learning rate or via early stopping. LightGBM provides support for both.
- In addition to a wide variety of loss functions for classification, regression and ranking, LightGBM also provides support for incorporation of our own custom, problem-specific loss functions for training.

In this chapter we have dived into the deep end of the current state-of-the-art in ensemble methods with gradient boosting. In the next chapter, we will move on to another powerful state-of-the-art: *Newton boosting*.

# 6

## Sequential Ensembles: Newton Boosting

### This chapter covers

- Using Newton descent to optimize loss functions for training models
- Implementing and understanding how Newton boosting works
- Learning with regularized loss functions
- Introducing XGBoost: a powerful framework for Newton boosting
- Avoiding overfitting with XGBoost in practice

In Chapters 4 and 5 we saw two approaches to constructing sequential ensembles. In Chapter 4, we introduced a new ensemble method called adaptive boosting (AdaBoost), which uses weights to identify the most misclassified examples. In Chapter 5 we introduced another ensemble method called “gradient boosting,” which uses gradients (residuals) to identify the most misclassified examples.

The fundamental intuition behind both of these boosting methods is to target the most misclassified (essentially, the “worst” behaving) examples at every iteration and improve classification by doing better with them.

In this chapter, we introduce a third boosting approach: Newton boosting, that combines the advantages of both and uses *weighted gradients* (or weighted residuals) to identify the most misclassified examples.

As with gradient boosting, the framework of Newton Boosting can be applied to any loss function, which means that any classification, regression, or ranking problem can be “boosted” using weak learners. In addition to this flexibility, packages such as `XGBoost` are now available that can scale Newton Boosting to big data through parallelization.

Unsurprisingly, today, Newton boosting is considered by many practitioners to be a **state-of-the-art** ensemble approach.

As Newton boosting builds upon Newton descent, we kick off the chapter with examples of Newton descent, and how can be used to train a machine learning model (Section 6.1).

Section 6.2 aims to provide intuition for learning with weighted residuals, the key intuition behind Newton boosting. As always, we implement our own version of Newton boosting to understand how it combines gradient descent and boosting to train a sequential ensemble.

Section 6.3 introduces XGBoost, a free and open-source gradient and Newton boosting package, a widely used tool for building and deploying real-world ML applications. In Section 6.4, we further see how we can avoid overfitting with strategies such as early stopping and adapting the learning rate with XGBoost.

Finally, we will reuse the real-world study from Chapter 5, document retrieval, in Section 6.5 to compare the performance of XGBoost to LightGBM, its variants and random forests.

The origins and motivation for devising Newton boosting are analogous to those of the gradient boosting algorithm: the optimization of loss functions. Gradient descent, which gradient boosting is based on, is a first-order optimization method, in that it uses first derivatives during optimization.

Newton's method, or Newton descent, is a second-order optimization method, in that it uses both first and second derivative information during optimization. That is, unlike gradient descent that uses only gradient information to do compute the next (gradient) step, Newton's method uses gradient and second derivative information together to compute a Newton step. When combined with boosting, *we obtain the ensemble method of Newton boosting.*

We begin this chapter by trying to understand how Newton's method inspires a powerful and widely used ensemble method.

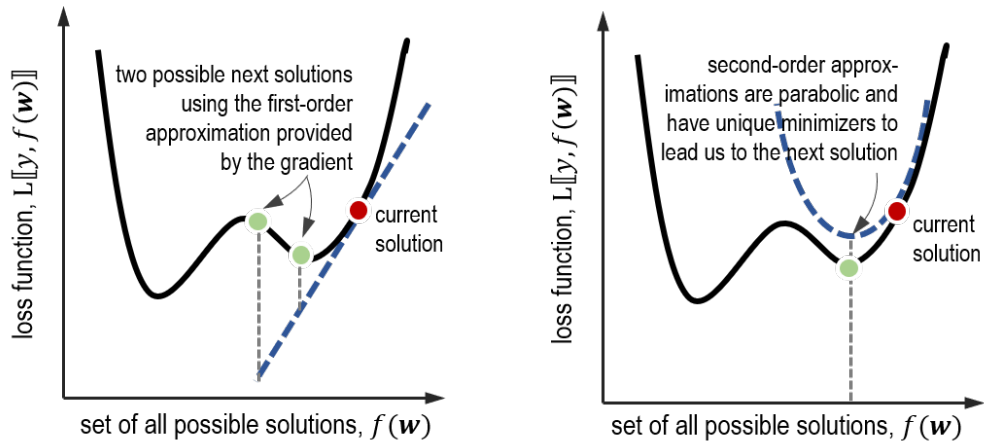
## 6.1 Newton's Method for Minimization

Iterative optimization methods such as gradient descent and Newton's method perform an update within each iteration:  $\text{next} = \text{current} + \text{step} * \text{direction}$ .

In gradient descent (Figure 6.1, left), first derivative information only allows us to construct a local linear approximation at best. While this gives us a descent direction, different step lengths can give us vastly different estimates, and ultimately slow down convergence.

Incorporating second derivative information, as Newton descent does, allows us to construct a local quadratic approximation! This extra information leads to a better local approximation, resulting in better steps and faster convergence.

**NOTE** The approach described in this chapter, Newton's method for optimization, is derived from a more general root-finding method, also called Newton's method. We will often use the term Newton descent to refer to Newton's method for minimization.



**Figure 6.1.** Comparing gradient descent (left) and Newton's method (right). Gradient descent only uses local first-order information near the current solution, which leads to a linear approximation of the function being optimized. Newton's method uses both local first- and second-order information near the current solution, leading to a quadratic (parabolic) approximation of the function being optimized. This provides a better estimate of the next step.

More formally, gradient descent computes the next update as

$$w_{t+1} = w_t + \alpha_t \cdot (-f'(w_t)),$$

Where  $\alpha_t$  is the step length and  $(-f'(w_t))$  is the negative gradient, or the negative of the first derivative. Newton's method computes the next update as

$$w_{t+1} = w_t + \alpha_t \cdot \left( -\frac{f'(w_t)}{f''(w_t)} \right),$$

Where  $f''(w_t)$  is the second derivative, and the step length  $\alpha_t$  is typically set to 1.

#### THE SECOND DERIVATIVE AND THE HESSIAN MATRIX

For univariate functions (i.e., functions in one variable), the second derivative is easy to compute: we simply differentiate the function twice. For instance, for the function  $f(w) = x^5$ , the first derivative is  $f'(x) = \partial f / \partial x = 5x^4$  and the second derivative is  $f''(x) = \partial f / \partial x \partial y = 20x^3$ .

For multivariate functions, or functions in many variables, the calculation of the second derivative is a little more involved. This is because we now have to consider differentiating the multivariate function with respect to pairs of variables.

To see this, consider a function in three variables:  $f(x, y, z)$ . The gradient of this function is straightforward to compute: we differentiate the function  $f$  with respect to each of the variables  $x, y$  and  $z$  (where w.r.t. is with respect to):

$$\nabla f = \begin{bmatrix} \text{derivative of } f \text{ w.r.t. } x \\ \text{derivative of } f \text{ w.r.t. } y \\ \text{derivative of } f \text{ w.r.t. } z \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \\ \frac{\partial f}{\partial z} \end{bmatrix}.$$

To compute the second derivative, we have to further differentiate each entry of the gradient with respect to  $x, y$  and  $z$  again. This produces a matrix known as the Hessian:

$$\begin{aligned} \nabla^2 f &= \begin{bmatrix} \text{deriv. of } \frac{df}{dx} \text{ w.r.t. } x & \text{deriv. of } \frac{df}{dx} \text{ w.r.t. } y & \text{deriv. of } \frac{df}{dx} \text{ w.r.t. } z \\ \text{deriv. of } \frac{df}{dy} \text{ w.r.t. } x & \text{deriv. of } \frac{df}{dy} \text{ w.r.t. } y & \text{deriv. of } \frac{df}{dy} \text{ w.r.t. } z \\ \text{deriv. of } \frac{df}{dz} \text{ w.r.t. } x & \text{deriv. of } \frac{df}{dz} \text{ w.r.t. } y & \text{deriv. of } \frac{df}{dz} \text{ w.r.t. } z \end{bmatrix} \\ &= \begin{bmatrix} \frac{\partial}{\partial x} \left( \frac{\partial f}{\partial x} \right) & \frac{\partial}{\partial x} \left( \frac{\partial f}{\partial y} \right) & \frac{\partial}{\partial x} \left( \frac{\partial f}{\partial z} \right) \\ \frac{\partial}{\partial y} \left( \frac{\partial f}{\partial x} \right) & \frac{\partial}{\partial y} \left( \frac{\partial f}{\partial y} \right) & \frac{\partial}{\partial y} \left( \frac{\partial f}{\partial z} \right) \\ \frac{\partial}{\partial z} \left( \frac{\partial f}{\partial x} \right) & \frac{\partial}{\partial z} \left( \frac{\partial f}{\partial y} \right) & \frac{\partial}{\partial z} \left( \frac{\partial f}{\partial z} \right) \end{bmatrix}. \end{aligned}$$

The Hessian matrix is a symmetric matrix, because the order of differentiation does not change the result meaning that

$$\frac{\partial}{\partial x} \left( \frac{\partial f}{\partial y} \right) = \frac{\partial}{\partial y} \left( \frac{\partial f}{\partial x} \right),$$

and so on, for all pairs of variables in  $f$ . In the multivariate case, the extension Newton's method is given by,

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha_t \cdot (-\nabla^2 f(\mathbf{w}_t)^{-1} \nabla f(\mathbf{w}_t)),$$

Where  $\nabla f(\mathbf{w}_t)$  is the gradient vector of the multivariate function  $f$  and  $-\nabla^2 f(\mathbf{w}_t)^{-1}$  is the inverse of the Hessian matrix. Inverting the second-derivative Hessian matrix is the multivariate equivalent of dividing by the term  $f''(\mathbf{w}_t)$ .

For large problems with many variables, inverting the Hessian matrix can become quite computationally expensive, slowing down overall optimization. As we will see in Section 6.2, Newton boosting circumvents this issue by computing second derivatives for individual examples and avoids inverting the Hessian.

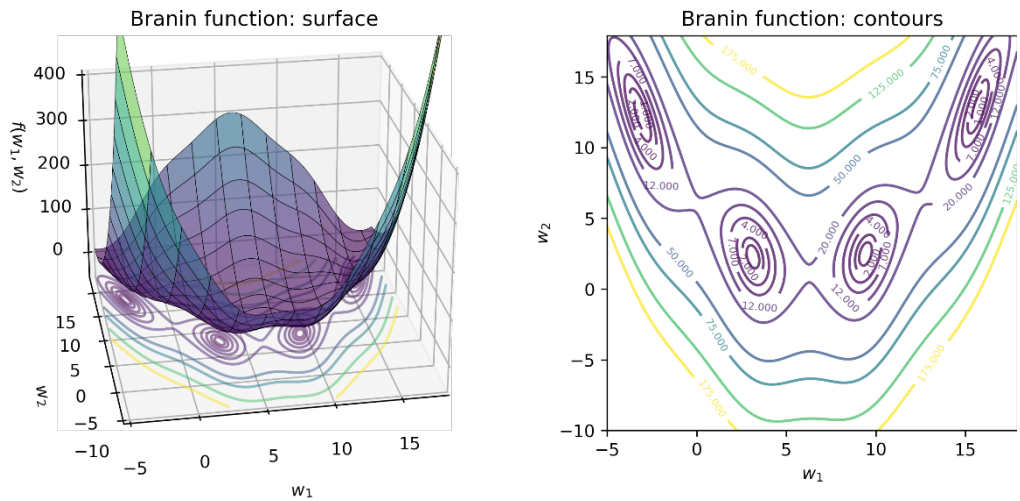
For now, let us continue to explore the differences between gradient descent and Newton's method. We return to the two examples we used in Section 5.1: the simple illustrative Branin function and the squared-loss function. We will use these examples to illustrate the differences between gradient descent and Newton descent.

### 6.1.1 Newton's Method with an Illustrative Example

Recall from Chapter 5 that the Branin function is a function of two variables ( $w_1$  and  $w_2$ ), defined as follows:

$$f(w_1, w_2) = a(w_2 - b w_1^2 + c w_1 - r)^2 + s(1 - t) \cos(w_1) + s,$$

Where  $a=1$ ,  $b=5.1 / 4\pi^2$ ,  $c=5 / \pi$ ,  $r=6$ ,  $s=10$ , and  $t=1 / 8\pi$  are fixed constants. This function is show below and has four minimizers at the centers of the elliptical regions.



**Figure 6.2.** The surface plot (left) and contour plot (right) of the Branin function. We can visually verify that this function has four minima, which are the centers of the elliptical regions in the contour plot.

We will take our gradient descent implementation from the previous section and modify it to implement Newton's method. The modified pseudo-code is shown below.

There are two key differences: 1. we compute the descent direction using the gradient and the Hessian, that is, using both the first and second-derivative information, and 2. we drop the computation of the step length, that is, we assume that the step length is 1.

```

initialize:  $w_{old}$  = some initial guess, converged=False
while not converged:
1. compute the gradient vector  $g$  and Hessian matrix  $H$  at the current estimate,  $w_{old}$ 
2. compute the descent direction  $d = -H^{-1}g$ 
3. set step length  $\theta < \alpha \leq 1$ 
4. update the solution:  $w_{new} = w_{old} + \text{distance} * \text{direction} = w_{old} + \alpha * d$ 
5. if change between  $w_{new}$  and  $w_{old}$  is below some specified tolerance:
    converged=True, so break
6.  $w_{old} = w_{new}$ , get ready for the next iteration

```

The key steps in this pseudo-code are Steps 1 and 2, where the descent direction is computed using the inverse Hessian matrix (second derivatives) and the gradient (first derivatives). Note that, as with gradient descent, the Newton descent direction is negated.

Step 3 is simply included to explicitly illustrate that, unlike gradient descent, Newton's method does not require the computation of a step length. Instead the step length can be set ahead of time, much like a learning rate. Once the descent direction is identified, Step 4 implements the Newton update:  $w_{t+1} = w_t + (-\nabla^2 f(w_t)^{-1} \nabla f(w_t))$ .

After we compute each update, similar to gradient descent, we check for convergence; here, our convergence test is to see how close  $w_{new}$  and  $w_{old}$  are to each other. If they are close enough, we terminate, and if not, we continue on to the next iteration. The listing below implements Newton's method.

### Listing 6.1 Newton descent

```

def newton_descent(f, g, h, x_init, max_iter=100, args=()): #A
    converged = False #B
    n_iter = 0

    x_old, x_new = np.array(x_init), None
    descent_path = np.full((max_iter + 1, 2), fill_value=np.nan)
    descent_path[n_iter] = x_old

    while not converged:
        n_iter += 1
        gradient, hessian = g(x_old, *args), h(x_old, *args) #C
        direction = -np.dot(np.linalg.inv(hessian), gradient) #D

        distance = 0.5 #E
        x_new = x_old + distance * direction #F
        descent_path[n_iter] = x_new

        err = np.linalg.norm(x_new - x_old) #G
        if err <= 1e-3 or n_iter >= max_iter:
            converged = True #H

        x_old = x_new #I

    return x_new, descent_path

```

#A Newton descent requires a function  $f$ , its gradient  $g$ , and its Hessian  $h$

#B initialize Newton descent to not converged

#C compute the gradient and the Hessian



```

#D compute the Newton direction
#E set step length to 0.5
#F compute the update
#G compute change from previous iteration
#H converged if change is small or max. iterations reached
#I get ready for the next iteration

```

Note that the step length is set to 0.5 to control the rate of descent (more on this below).

Let's take our implementation of Newton descent for a spin. We have already implemented the Branin function and its gradient in the previous section. This implementation is shown here again:

```

def branin(w, a, b, c, r, s, t):
    return a * (w[1] - b * w[0] ** 2 + c * w[0] - r) ** 2 +
           s * (1 - t) * np.cos(w[0]) + s

def branin_gradient(w, a, b, c, r, s, t):
    return np.array([2 * a * (w[1] - b * w[0] ** 2 + c * w[0] - r) *
                    (-2 * b * w[0] + c) - s * (1 - t) * np.sin(w[0]),
                    2 * a * (w[1] - b * w[0] ** 2 + c * w[0] - r)])

```

We also need the Hessian (second derivative) matrix for Newton descent. We can compute it by analytically differentiating the gradient (first derivative) vector:

$$h(w_1, w_2) = \begin{pmatrix} \frac{\partial}{\partial w_1} \left( \frac{\partial f}{\partial w_1} \right) & \frac{\partial}{\partial w_2} \left( \frac{\partial f}{\partial w_1} \right) \\ \frac{\partial}{\partial w_1} \left( \frac{\partial f}{\partial w_2} \right) & \frac{\partial}{\partial w_2} \left( \frac{\partial f}{\partial w_2} \right) \end{pmatrix}$$

$$= \begin{bmatrix} 2a(-2bw_1 + c)^2 - 4ab(w_2 - bw_1^2 + cw_1 - r) - s(1-t)\cos w_1 & 2a(-2bw_1 + c) \\ 2a(-2bw_1 + c) & 2a \end{bmatrix}$$

This can also be implemented as shown below:

```

def branin_hessian(w, a, b, c, r, s, t):
    return np.array([[2 * a * (-2 * b * w[0] + c)** 2 -
                    4 * a * b * (w[1] - b * w[0] ** 2 + c * w[0] - r) -
                    s * (1 - t) * np.cos(w[0]),
                    2 * a * (-2 * b * w[0] + c)],
                    [2 * a * (-2 * b * w[0] + c),
                    2 * a]])

```

As with gradient descent, Newton descent (Listing 6.1) also requires an initial guess `x_init`. Here, we will initialize gradient descent with `w_init=[2, -5]`. Now, we can call the Newton descent procedure.

```

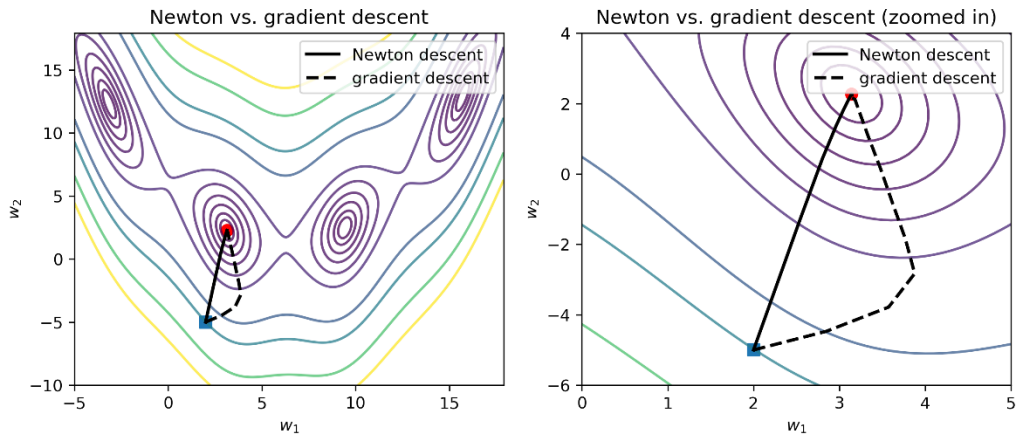
a, b, c, r, s, t = 1, 5.1/(4 * np.pi**2), 5/np.pi, 6, 10, 1/(8 * np.pi)
w_init = np.array([2, -5])
w_optimal, w_path = newton_descent(branin, branin_gradient,
                                   w_init, args=(a, b, c, r, s, t))

```

Newton descent returns an optimal solution `w_optimal` and the optimization path `w_path`. So how does Newton descent compare to gradient descent? In the figure below, we plot both the solution paths of both optimization algorithms together.

The result of this comparison is pretty striking: Newton descent is able to exploit the additional local information about the curvature of the function provided by the Hessian matrix to take a more direct path to the solution.

In contrast, gradient descent only has the first-order gradient information to work with and takes a roundabout path to the same solution.



**Figure 6.3.** We compare the solution paths of Newton descent and gradient descent starting from  $[2, -5]$  (blue square) and both converging to one of the local minima (red circle). Newton descent (solid line) progresses towards the local minimum in a more direct path compared to gradient descent (dotted line). This is because Newton descent uses a more informative second-order local approximation with each update, while gradient descent only uses a first-order local approximation.

#### PROPERTIES OF NEWTON DESCENT

We note a couple of important things about Newton descent and its similarities to gradient descent. First, unlike gradient descent, Newton's method computes the descent step exactly and does not require a step length.

In our example, we set the step length  $\alpha=0.5$  for Newton's method; this allows us to control the rate of descent and ensure convergence to a local minimum.

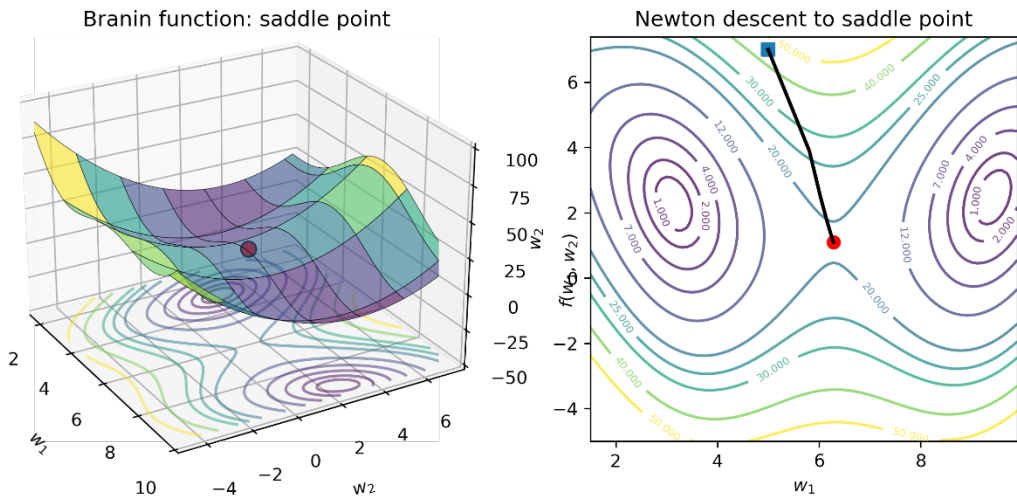
Keep in mind that our purpose is to extend Newton descent to Newton boosting. From this perspective, the step length can be interpreted as a learning rate. Choosing an effective learning rate (say, using cross validation like we did for AdaBoost or gradient boosting) is very much akin to choosing a good step length. Instead of selecting a learning rate to accelerate convergence, in boosting algorithms, we select the learning rate to help us avoid overfitting and to generalize better to the test set and future data.

Another important point to keep in mind is that, like gradient descent, Newton descent is also sensitive to our choice of initial point. Different initializations will lead Newton descent to different local minimizers.

A bigger issue is that, in addition to local minimizers, our choice of initial point can also lead Newton descent to converge to *saddle points*. This is a problem faced by all descent algorithms and is illustrated in Figure 6.4.

A saddle point mimics a local minimizer: at both locations, the gradient of the function becomes zero. However, saddle points are not true local minimizers: the saddle-like shape means that it is curving upwards in one direction and curving downwards in another. This is in contrast to local minimizers which are bowl-shaped.

However, both local minimizers and saddle points have zero gradients. This means that descent algorithms cannot distinguish between them and sometimes converge to saddle points instead of minimizers.



**Figure 6.4.** A saddle point of the Branin function lies between two minimizers, and like the minimizers, it too has a zero gradient at its location. This causes all descent methods to converge to saddle points.

The existence of saddle points and local minimizers depends on the functions being optimized, of course. For our purposes, most common loss functions are often convex and “well-shaped”, meaning that we can safely use Newton descent and Newton boosting.

Care should be taken, however, to ensure convexity when creating and using custom loss functions. Handling such non-convex loss functions is an active and ongoing research area.

### 6.1.2 Newton Descent over Loss Functions for Training

So how does Newton descent fare on a machine learning task? To see this, we can revisit the simple 2d classification problem from Section 5.1.2 on which we have previously trained a model using gradient descent. The task is a binary classification problem, with data generated as shown below.

```
from sklearn.datasets import make_blobs
X, y = make_blobs(n_samples=200, n_features=2,
                 centers=[[-1.5, -1.5], [1.5, 1.5]])
```

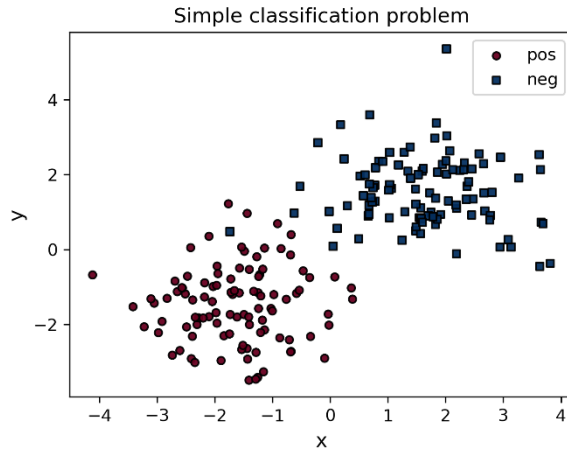


Figure 6.5. A (nearly) linearly separable two-class data set over which we will learn a classifier. The positive examples have labels  $y=1$  and the negative examples have labels  $y=0$ .

Recall that we would like to train a linear classifier  $h_w(x) = w_1x_1 + w_2x_2$ , takes two-dimensional data points  $x = [x_1, x_2]^T$  and returns a prediction. As in Section 5.1.2, we will use the squared loss function for this task.

The linear classifier is parameterized by weights  $w = [w_1, w_2]^T$ , that we have learned. The weights, of course, have to be learned such that they minimize some loss over the data to achieve the best training fit.

The squared loss measures the error between true labels  $y_i$  and their corresponding predictions  $h_w(x_i)$  as shown below,

$$f_{\text{loss}}(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^n (y_i - h_w(\mathbf{x}_i))^2 = \frac{1}{2} \sum_{i=1}^n (y_i - w_1x_1^i - w_2x_2^i)^2 = \frac{1}{2} (\mathbf{y} - \mathbf{X}\mathbf{w})'(\mathbf{y} - \mathbf{X}\mathbf{w}).$$

Here,  $X$  is an  $n \times d$  data matrix of  $n$  training examples with  $d$  features each and  $y$  is a  $d \times 1$  vector of true labels.

The expression on the far right is a compact way of representing the loss over the entire data set using vector and matrix notation.

For Newton descent, we will need the gradient and Hessian of this loss function. These can be obtained by differentiating the loss function analytically, just as with the Branin function. In vector-matrix notation, these can also be compactly written as:

$$\begin{aligned} g(\mathbf{w}) &= -X'(\mathbf{y} - \mathbf{X}\mathbf{w}) \\ \mathbf{He}(\mathbf{w}) &= X'X \end{aligned}$$

The implementations of the loss function, its gradient and Hessian are shown below:

```
def squared_loss(w, X, y):
    return 0.5 * np.sum((y - np.dot(X, w))**2)

def squared_loss_gradient(w, X, y):
    return -np.dot(X.T, (y - np.dot(X, w)))

def squared_loss_hessian(w, X, y):
    return np.dot(X.T, X)
```

Now that we have all the components of the loss function, we can use Newton descent to compute an optimal solution, i.e., “learn a model”.

We initialize both gradient descent and Newton descent with  $w=[0.0, 0.99]'$ .

```
w_init = np.array([0.0, -0.99])
w_gradient, path_gradient = gradient_descent(squared_loss,
                                             squared_loss_gradient,
                                             w_init, args=(X, y))
w_newton, path_newton = newton_descent(squared_loss,
                                       squared_loss_gradient,
                                       squared_loss_hessian,
                                       w_init, args=(X, y))

print(w_gradient)
[0.13643511 0.13862275]

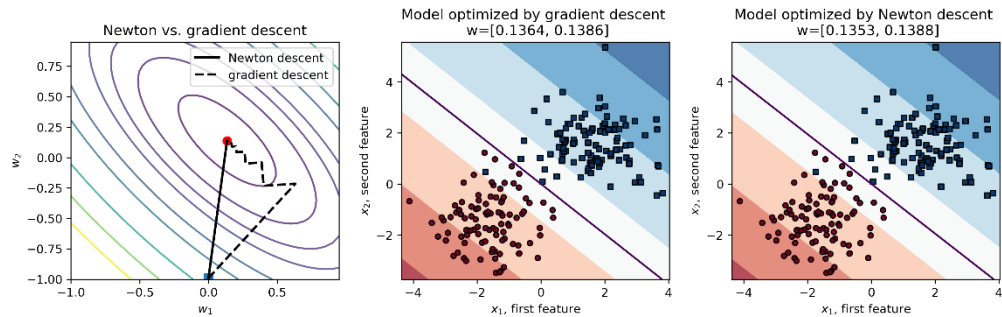
print(w_newton)
[0.13528094 0.13884772]
```

The squared loss function we are optimizing is convex and has *only one minimizer*. Both gradient descent and Newton descent essentially learn the same model, though they terminate as soon as they reach the threshold  $10^{-3}$ , roughly the third decimal place.

We can easily verify that this learned model achieves a training accuracy of 99.5%.

```
ypred = (np.dot(X, w_newton) >= 0).astype(int)
from sklearn.metrics import accuracy_score
accuracy_score(y, ypred)
0.995
```

While both gradient descent and Newton descent learn the same model, they arrive there in decidedly different ways, as the figure below illustrates.



**Figure 6.6.** The solution paths of Newton descent (solid line) vs. gradient descent (dotted line) as well as the models produced by Newton and gradient descent. Gradient descent takes 20 iterations to learn this model, while Newton descent takes 12 iterations.

The key takeaway is that Newton descent is a powerful optimization method in the family of descent methods. It converges to solutions much faster as it takes local second-order derivative information (essentially curvature) into account in constructing descent directions.

This additional information about the shape of the objective (or loss) function being optimized greatly aids convergence. This comes at a computational cost however: with more variables, the second derivative, or the Hessian, which holds the second-order information becomes increasingly difficult to manage, especially as it has to be inverted.

As we see in the next section, Newton boosting avoids computing or inverting the entire Hessian matrix by using an approximation with *pointwise second derivatives*, essentially second derivatives computed and inverted per training example, which keeps training efficient.

## 6.2 Newton Boosting: Newton's Method + Boosting

We begin our deep dive into Newton boosting by gaining an intuitive understanding of how Newton boosting differs from gradient boosting. We'll compare the two methods side by side to see exactly what Newton boosting adds to each iteration.

### 6.2.1 Intuition: Learning with Weighted Residuals

As with other boosting methods, Newton boosting learns a new weak estimator every iteration such that it fixes the misclassifications or errors made by the previous iteration.

AdaBoost identifies and characterizes misclassified examples that need attention by assigning *weights* to them: badly misclassified examples are assigned higher weights. A weak classifier trained on such weighted examples will focus on them more during learning.

Gradient boosting characterizes misclassified examples that need attention through *residuals*. A residual is simply another means to measure the extent of misclassification and is computed as the gradient of the loss function.

Newton boosting does both and uses *weighted residuals*! The residuals in Newton boosting are computed in exactly the same way as in gradient boosting: using the gradient of the loss

function (the first derivative). The weights, on the other hand, are computed using the Hessian of the loss function (the second derivative).

#### NEWTON BOOSTING IS NEWTON DESCENT + BOOSTING

As we saw in Chapter 5, each gradient boosting iteration mimics gradient descent. At iteration  $t$ , gradient descent updates the model  $f_t$  using the gradient of the loss function ( $\nabla L(f_t) = g_t$ ):

$$f_{t+1} = f_t + \alpha_t \cdot \nabla L(f_t) = f_t + \alpha_t \cdot g_t.$$

Rather than compute the overall gradient directly  $g_t$ , gradient boosting learns a weak estimator ( $h_t^{\text{GB}}$ ) over the individual gradients, which are also residuals. That is, a weak estimator is trained over the data and corresponding residuals  $(x_i, g_t(x_i))_{i=1}^n$ . The model is then updated as

$$f_{t+1} = f_t + \alpha_t \cdot h_t^{\text{GB}}.$$

Similarly, Newton boosting mimics Newton descent. At iteration  $t$ , Newton descent updates the model  $f_t$  using the gradient of the loss function ( $(\nabla^2 L(f_t) = H_t$ , as with gradient descent) and the Hessian of the loss function ( $\nabla^2 L(f_t) = H_t$ ):

$$f_{t+1} = f_t + \alpha_t \cdot \nabla L(f_t)^{-1} \cdot \nabla L(f_t) = f_t + \alpha_t \cdot H_t^{-1} g_t.$$

Computing the Hessian can often be very computationally expensive. Newton boosting avoids the expense of computing the gradient or the Hessian by learning a weak estimator over the individual gradients and Hessians.

For each training example, in addition to the gradient residual, we have to incorporate the Hessian information as well, all the while ensuring that the overall weak estimator we want to train approximates Newton descent.

How do we do this? Observe that the Hessian matrix is inverted in the Newton update ( $H_t^{-1}$ ). For a single training example, the second (functional) derivative will be a scalar (a single number instead of a matrix).

This means that the term  $H_t^{-1} g_t$  becomes  $g_t(x_i)/H_t(x_i)$ ; these are simply the residuals  $g_t(x_i)$  weighted by the Hessians  $g_t(x_i)/H_t(x_i)$ .

Thus, for Newton boosting, we train a weak estimator ( $h_t^{\text{NB}}$ ) using Hessian-weighted gradient residuals:  $(x_i, g_t(x_i)/H_t(x_i))_{i=1}^n$ . And voila, we can update our ensemble in exactly the same way as gradient boosting:

$$f_{t+1} = f_t + \alpha_t \cdot h_t^{\text{NB}}.$$

In summary, Newton boosting uses Hessian-weighted residuals, while gradient boosting uses unweighted residuals.

### WHAT DO THE HESSIANS ADD?

So, what kind of additional information do these Hessian-based weights add to boosting?

Mathematically, Hessians, or second derivatives, correspond to the curvature or how “curvy” a function is. In Newton boosting, we weight gradients by second-derivative information for each training example  $x_i$ ,

$$g_t(x_i)/He_t(x_i).$$

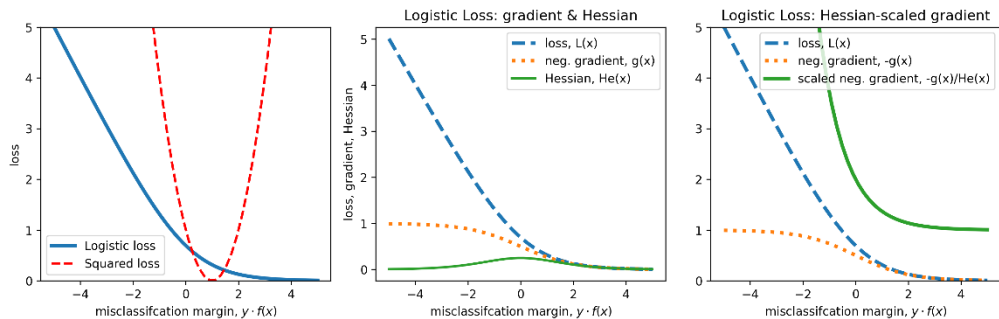
A large value of the second derivative  $He_t(x_i)$  implies that the curvature of the function is large at  $x_i$ . At these curvy regions, the Hessian weight decreases the gradient, which, in turn, leads Newton boosting to take smaller, more conservative steps.

Conversely, if the second derivative  $He_t(x_i)$  is small, then the curvature at  $x_i$  is small, meaning that the function is rather flat. In such situations, the Hessian weight allows Newton descent to take large, bolder steps so it can traverse the flat area faster.

Thus, second derivatives combined with first-derivative residuals can capture the notion of “misclassification” very effectively. Let’s see this in action over a commonly used loss function: the logistic loss, which measures the extent of the misclassification:

$$L(y, f(x)) = \log(1 + e^{-y \cdot f(x)}).$$

The logistic loss is compared to the squared loss function in Figure 6.7 (left).



**Figure 6.7.** (left) Logistic loss vs. squared loss function; (center) negative gradient and Hessian of the logistic loss; (right) Hessian-scaled scale negative gradient of the logistic loss.

In Figure 6.7 (center), we look at the logistic loss function and its corresponding gradient (first derivative) and Hessian (second derivative). All of these are functions of the misclassification margin: the product of the true label ( $y$ ) and the prediction ( $f(x)$ ).

If  $y$  and  $f(x)$  have opposite signs, then we have that  $y \cdot f(x) < 0$ . In this case, the true label does not match the predicted label and we have a misclassification.



Thus, the left part of the logistic loss measures the extent of the misclassification, which the right part of the loss function corresponds to correctly classified examples, whose loss is nearly zero, as we expect.

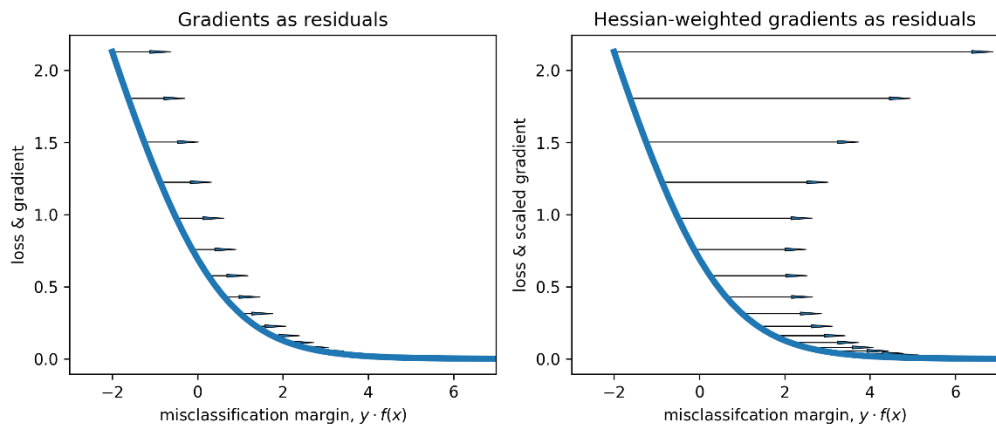
The second derivative achieves its highest values around 0, which corresponds to the elbow of the logistic loss function. This is not surprising since we can see that the logistic loss function is curviest around the elbow and flat to the left and right of the elbow.

In Figure 6.7 (right), we can see the effect of weighting the gradients. For well classified examples ( $y \cdot f(x) > 0$ ), the overall gradient as well as weighted gradient are zero. This means that these examples will not participate in the boosting iteration.

On the other hand, for misclassified examples ( $y \cdot f(x) < 0$ ), the overall weighed gradient  $g(x_i) / \text{He}(x_i)$  increases steeply with misclassification. In general, it increases far more steeply than the unweighted gradient.

Now, we can answer the question of what the Hessians do. They incorporate local curvature information to ensure that training examples that are badly misclassified get higher weights.

This is illustrated in the figure below.



**Figure 6.8.** Unweighted residuals (left) used by gradient boosting compared to Hessian-weighted residuals (right) used by Newton boosting. Positive values of misclassification margin ( $y \cdot f(x) > 0$ ) indicate correct classification. For misclassifications we have  $y \cdot f(x) < 0$ . For badly misclassified examples, the Hessian-weighted gradients capture the notion “badly classified” more effectively than unweighted gradients.

The more badly misclassified a training example is, the further to the left it will be in Figure 6.8. Hessian-weighting of residuals ensures that training examples that are further to the left will get higher weights.

This is in sharp contrast to gradient boosting, which is unable to differentiate training examples as effectively as it only uses unweighted residuals.

To summarize: Newton boosting aims to use both first derivative (gradient) information as well as second derivative (Hessian) information to ensure that misclassified training examples receive focus based on the extent of the misclassification.

### 6.2.2 Intuition: Learning with Regularized Loss Functions

Before proceeding, let's introduce the notion of *regularized loss functions*. A regularized loss function contains an additional smoothing term along with the loss function, making it more convex, or bowl-like.

Regularizing a loss function introduces additional structure to the learning problem, which often stabilizes and accelerates learning algorithms. Regularization also allows us to control the complexity of the model being learned and improves overall robustness and generalization capabilities of the model.

Essentially, a regularized loss function explicitly captures the fit vs. complexity tradeoff inherent in most machine-learning models (see Chapter 1.3).

A regularized loss function is of the form:

$$\lambda \cdot \underbrace{\text{regularization}(f(x))}_{\text{measures model complexity}} + \underbrace{\text{loss}(f(x), \text{data})}_{\text{measures model fit}}$$

The regularization term measures the flatness (the opposite of "curviness") of the model: the more it is minimized, the less complex the learned model is.

The loss term measures the fit to the training data through a *loss function*: the more it is minimized, the better the fit to the training data. The *regularization parameter*  $\lambda$  trades-off between these two competing objectives:

- a large value of  $\lambda$  means the model will focus more on regularization and simplicity and less on training error, which causes the model to have higher training error and underfit,
- a small value of  $\lambda$  means the model will focus more on training error, learn more complex models, which causes the model to have lower training errors and possibly overfit.

Thus, a regularized loss function allows us to tradeoff between fit and complexity during learning, ultimately leading to models that generalize well in practice.

As we saw in Section 1.3, there are several ways to introduce regularization and control model complexity during learning. For example, limiting the maximum depth of trees or the number of nodes prevents trees from overfitting.

Another common approach is through L2 regularization, which amounts to introducing a penalty over the model directly. That is, if we have a model  $f(x)$ , L2 regularization introduces a penalty over model by  $f(x)^2$ :

$$\lambda \cdot \underbrace{f(\mathbf{x})^2}_{\text{penalizes model complexity}} + \underbrace{\text{loss}(f(\mathbf{x}), \text{data})}_{\text{measures model fit}}$$

The loss functions of many common ML approaches can be expressed in this form. In Chapter 5, we implemented the gradient boosting algorithm for the *unregularized* squared loss function

$$0 \cdot \underbrace{f(\mathbf{x})^2}_{\text{model complexity}} + \underbrace{\frac{1}{2} (y - f(\mathbf{x}))^2}_{\text{squared loss}},$$

between the true label  $y$  and the predicted label  $f(\mathbf{x})$ . In the above setting, unregularized loss functions simply have the regularization parameter  $\lambda=0.1$ .

We have already seen an example of a regularized loss function (Section 1.3.2): support vector machines (SVMs), which use the regularized hinge loss function.

$$L(y, f(\mathbf{x})) = \lambda \underbrace{f(\mathbf{x})^2}_{\text{L2 regularization}} + \underbrace{\max(0, 1 - y \cdot f(\mathbf{x}))}_{\text{hinge loss}}.$$

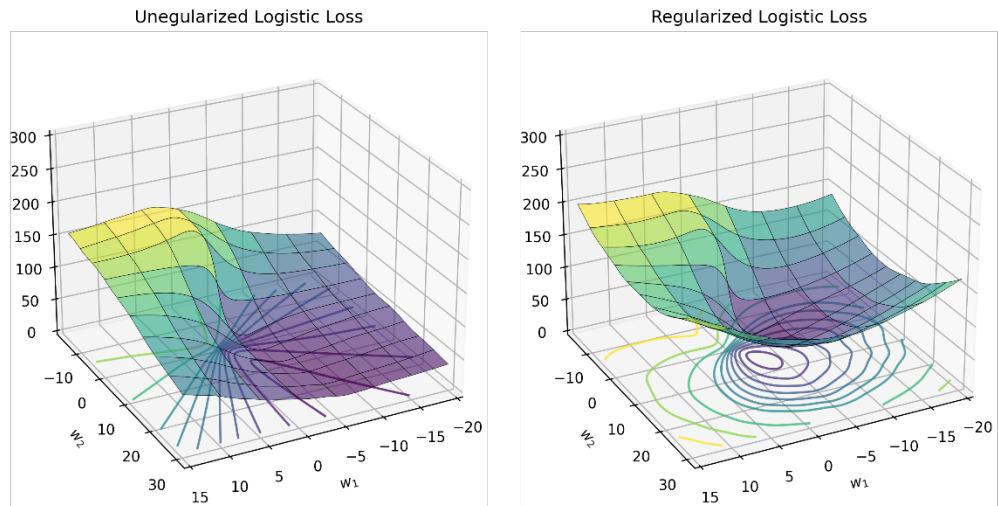
In this chapter, we consider the *regularized logistic loss* function, which is commonly used in logistic regression:

$$L(y, f(\mathbf{x})) = \lambda \underbrace{f(\mathbf{x})^2}_{\text{L2 regularization}} + \underbrace{\log(1 + e^{-y \cdot f(\mathbf{x})})}_{\text{logistic loss}},$$

which augments the standard logistic loss  $\log(1 + e^{-y \cdot f(\mathbf{x})})$ , with a regularization term  $\lambda f(\mathbf{x})^2$ .

The figure below illustrates the regularized logistic loss for  $\lambda=0.1$ . Observe how the regularization term makes the loss function's profile curvier and more bowl-like.

The regularization parameter  $\lambda$  trades off between fit vs. complexity: as  $\lambda$  is increased, the regularization effect will increase, making the overall surface more convex and ignore the contributions of the loss function. Since the loss function affects the fit, over-regularizing the model will lead to underfitting.



**Figure 6.9.** (left) The standard logistic loss function vs. (right) the regularized logistic loss function, which is curvier and has a better-defined minimum.

The gradient and Hessian of the regularized logistic loss function can be computed as the first and second derivatives with respect to the model's prediction ( $\hat{f}(x)$ )

$$g(y, f(x)) = \frac{-y}{(1 + e^{-y f(x)})} + 2 \lambda f(x),$$

$$He(y, f(x)) = \frac{e^{y f(x)}}{(1 + e^{y f(x)})^2} + 2 \lambda.$$

The listing below implements functions to compute the regularized logistic loss, with the value of the parameter  $\lambda=0.1$ .

#### Listing 6.2 Regularized logistic loss, its gradient and Hessian with $\lambda=0.1$

```
def log_loss_func(y, F):
    return np.log(1 + np.exp(-y * F)) + 0.1 * F**2

def log_loss_grad(y, F):
    return -y / (1 + np.exp(y * F)) + 0.2 * F

def log_loss_hess(y, F):
    return np.exp(y * F) / (1 + np.exp(y * F))**2 + 0.2
```

These functions can now be used compute the residuals and corresponding Hessian weights that we will need for Newton boosting.

### 6.2.3 Implementing Newton Boosting

In this section, we will develop our own implementation of Newton boosting. The basic algorithm can be outlined with the following pseudocode:

```
initialize:  $F = f_0$ , some constant value
for  $t = 1$  to  $T$ :
1. compute first and second derivatives for each example,

$$g_i^t = -\frac{\partial L}{\partial F}(x_i), He_i^t = \frac{\partial^2 L}{\partial F^2}(x_i)$$

2. compute the weighted residuals for each example  $r_i^t = -g_i^t / He_i^t$ 
3. fit a weak decision tree regressor  $h_t(x)$  using the training set  $(x_i, r_i^t)_{i=1}^n$ 
4. compute the step length ( $\alpha_t$ ) using line search
5. update the model:  $F_{t+1} = F_t + \alpha_t h_t(x)$ 
```

Unsurprisingly, this training procedure is the same as gradient boosting, with the only change being the computation of Hessian-weighted residuals in steps 1 and 2.

Since the general algorithmic framework for gradient and Newton boosting is the same, we can combine implement them together. The listing below extends Listing 5.2 to incorporate Newton boosting, which it uses for training only when the flag `use_Newton=True`.

#### Listing 6.3 Newton boosting for the regularized logistic loss

```
def fit_gradient_boosting(X, y, n_estimators=10, use_newton=True):
    n_samples, n_features = X.shape #A
    estimators = [] #B
    F = np.full((n_samples, ), 0.0) #C

    for t in range(n_estimators):
        if use_newton: #D
            residuals = -log_loss_grad(y, F) / log_loss_hess(y, F)
        else:
            residuals = -log_loss_grad(y, F) #E

        h = DecisionTreeRegressor(max_depth=1)
        h.fit(X, residuals) #F

        hreg = h.predict(X) #G
        loss = lambda a: np.linalg.norm(y - (F + a * hreg))**2 #H
        step = minimize_scalar(loss, method='golden') #I
        a = step.x

        F += a * hreg #J

        estimators.append((a, h)) #K

    return estimators
```

```
#A get dimensions of the data set
#B initialize an empty ensemble
#C predictions of the ensemble on the training set
#D if Newton boosting, compute Hessian-weighted residuals
#E else compute unweighted residuals for gradient boosting
```

```

#F fit weak regression tree (h_t) to the examples and residuals
#G get predictions of the weak learner, h_t
#H set up the loss function as a line search problem
#I find the best step length using the golden section search
#J update the ensemble predictions
#K update the ensemble

```

Once the model is learned, we can make predictions exactly as with AdaBoost or gradient boosting, since the ensemble learned is a sequential ensemble.

The listing below is the same prediction function used by these previously introduced methods, repeated here for convenience.

#### Listing 6.4: Predictions of Newton boosting

```

def predict_gradient_boosting(X, estimators):
    pred = np.zeros((X.shape[0], )) #A

    for a, h in estimators:
        pred += a * h.predict(X) #B

    y = np.sign(pred) #C

    return y

```

```

#A initialize all the predictions to 0
#B aggregate individual predictions from each regressor
#C convert weighted predictions to -1/1 labels

```

Let's compare the performance our implementations of gradient boosting (from the previous chapter) and Newton boosting (from above).

```

from sklearn.datasets import make_moons
X, y = make_moons(n_samples=200, noise=0.15, random_state=13)
y = 2 * y - 1 #A

from sklearn.model_selection import train_test_split
Xtrn, Xtst, ytrn, ytst = train_test_split(X, y, test_size=0.25,
                                         random_state=11) #B

estimators_nb = fit_gradient_boosting(Xtrn, ytrn, n_estimators=25,
                                     use_newton=True) #C
ypred_nb = predict_gradient_boosting(Xtst, estimators_nb)
print('Newton boosting test error = {}'.format(1 - accuracy_score(ypred_nb, ytst)))

estimators_gb = fit_gradient_boosting(Xtrn, ytrn, n_estimators=25,
                                     use_newton=False) #D
ypred_gb = predict_gradient_boosting(Xtst, estimators_gb)
print('Gradient boosting test error = {}'.format(1 - accuracy_score(ypred_gb, ytst)))

```

```

#A convert training labels to -1/1
#B split into train and test sets
#C Newton boosting
#D gradient boosting

```

We see that Newton boosting produces a test error of around 8% compared to gradient boosting, which achieves 12%.

```
Newton boosting test error = 0.07999999999999996
Gradient boosting test error = 0.12
```

### VISUALIZING GRADIENT BOOSTING ITERATIONS

Now that we have our joint gradient-and-Newton boosting implementation (Listing 6.3), we can compare the behaviors of both these algorithms. First, note that they both train and grow their ensembles in roughly the same way.

The key difference between them is in the residuals they use for ensemble training. To summarize, gradient boosting uses the (negative) gradients directly as residuals, which Newton boosting uses the (negative) Hessian-weighted gradients.

Let's step through the first few iterations to see what the impact of Hessian-weighting is. In the first iteration, both gradient and Newton boosting are initialized with  $F(x_i)=\theta$ .

Both gradient and Newton boosting use residuals as a means to measure the extent of misclassification so that the most misclassified training examples can get more attention in the current iteration.

In Figure 6.10, the very first iteration, the impact of Hessian weighting is immediately observable. Using second derivative information to weight the residuals increases the separation between the two classes, making them easier to discriminate between.

This behavior can also be seen in the second (Figure 6.11) and third (Figure 6.12) iterations, where Hessian weighting enables greater stratification of misclassifications, enabling the weak learning algorithm to construct more effective weak learners.

In summary, Newton boosting aims to use both first derivative (gradient) information as well as second derivative (Hessian) information to ensure that misclassified training examples receive increased attention dependent on the extent of the misclassification.

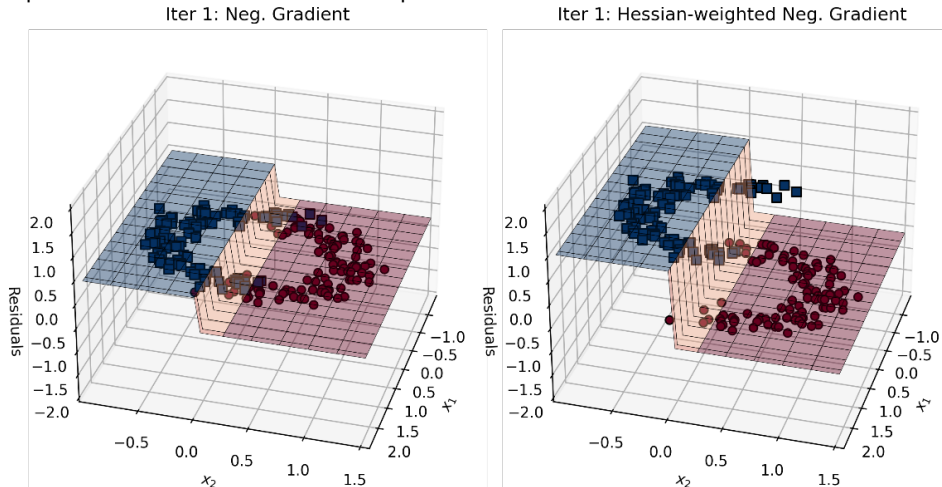
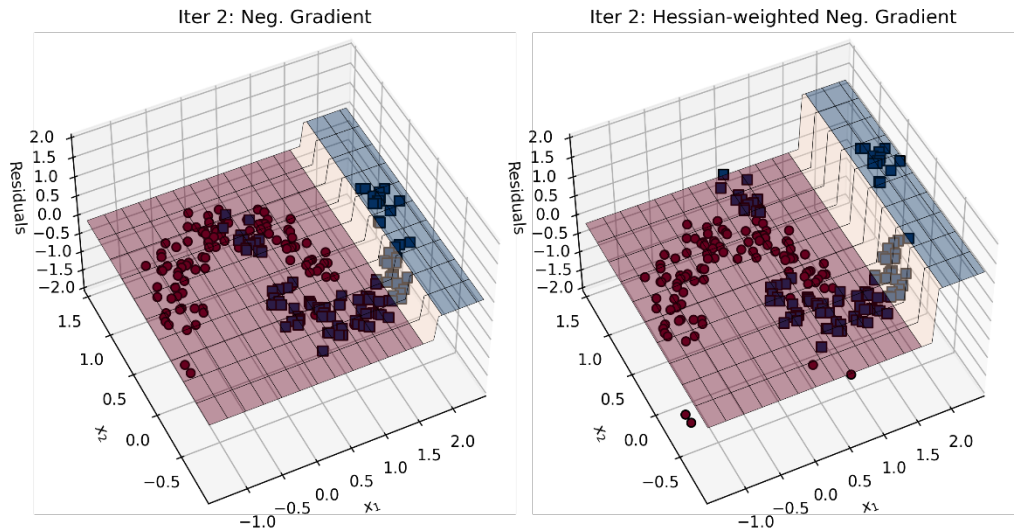
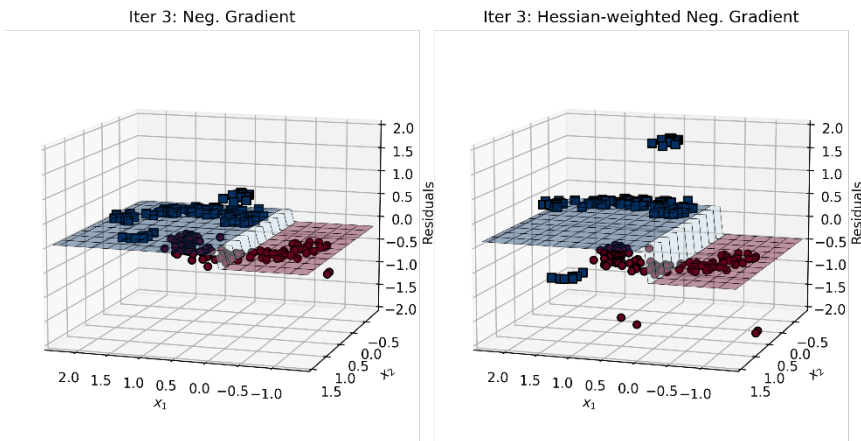


Figure 6.10. Iteration 1: (left) negative gradients as residuals in gradient boosting vs. (right) Hessian-weighted negative gradients as residuals in Newton boosting.



**Figure 6.11.** Iteration 2: (left) negative gradients as residuals in gradient boosting vs. (right) Hessian-weighted negative gradients as residuals in Newton boosting.

The figure below illustrates how Newton boosting grows the ensemble and decreases the error steadily over successive iterations.



**Figure 6.12.** Iteration 3: (left) negative gradients as residuals in gradient boosting vs. (right) Hessian-weighted negative gradients as residuals in Newton boosting.



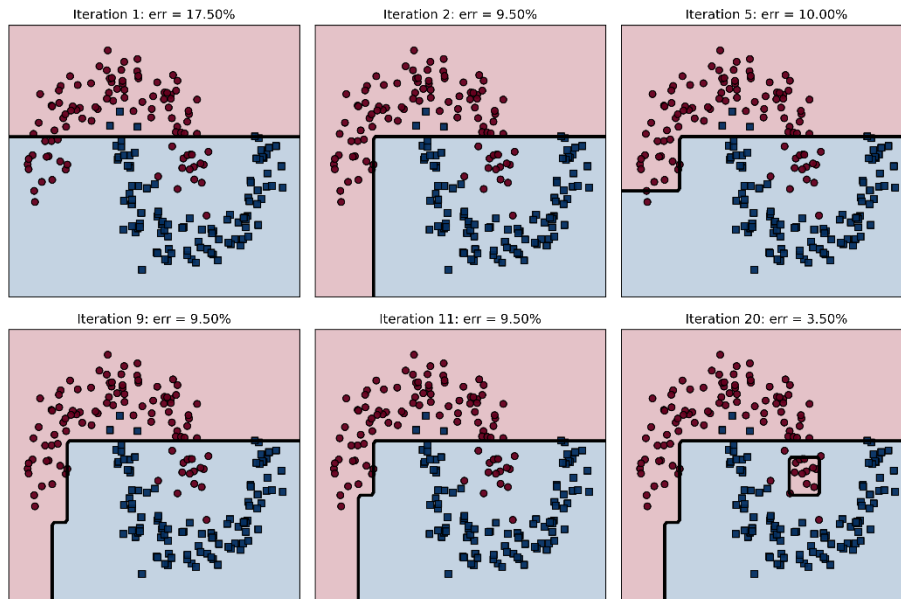


Figure 6.13. Newton boosting over 20 iterations.

### 6.3 XGBoost: A Framework for Newton Boosting

XGBoost, or eXtreme Gradient Boosting, is an open source gradient boosting framework that originated as a research project by Tianqi Chen. It gained widespread recognition and adoption, especially in the data science competition community, after its success in the Higgs Machine Learning Challenge.

It has, since, evolved into a powerful boosting framework that provides parallelization and distributed processing capabilities allowing it to scale to very large data sets. Today, XGBoost is available in many languages including Python, R and C/C++ and is deployed on several data-science platforms such as Apache Spark and H2O.

XGBoost has several key features that make it applicable in a wide variety of domains as well as for large-scale data:

- Newton boosting on regularized loss functions to directly control the complexity of the regression tree functions (weak learners) that constitute the ensemble (Section 6.3.1);
- Algorithmic speedups such as weighted quantile sketch, a variant of the histogram-based split finding algorithm (that LightGBM uses) for faster training (Section 6.3.1);
- Support for a large number of loss functions for classification, regression, and ranking, as well as application-specific custom loss functions, similar to LightGBM;
- Block-based system design that stores data in memory in smaller units called blocks; this allows for parallel learning, better caching, and efficient multi-threading (these details are out-of-scope for this book).

It will be impossible to detail all the features available in XGBoost in this limited space. Instead, this section and the next introduce XGBoost, its usage and applications in practical settings.

This should enable readers to springboard further into advanced use cases of XGBoost for their applications through its documentation.

### 6.3.1 What Makes XGBoost “Extreme”?

In a nutshell, XGBoost is extreme because of Newton boosting with regularized loss functions, efficient tree learning and a parallelizable implementation.

In particular, the success of XGBoost lies in the fact that its algorithms feature conceptual and algorithmic improvements *specifically designed for tree-based learning*.

In this section, we’ll focus on how XGBoost improves robustness and generalizability of tree-based ensembles and how it does so efficiently.

#### REGULARIZED LOSS FUNCTIONS FOR LEARNING

In Section 6.2.2, we saw several examples of L2-regularized loss functions of the form:

$$\lambda \cdot \underbrace{f(x)^2}_{\text{measures model complexity}} + \overbrace{\text{loss}(f(x), \text{data})}^{\text{measures model fit}}.$$

If we only consider tree-based learners for weak models in our ensemble, there are other ways to *directly control the complexity of the trees during learning*. XGBoost does this by introducing a further regularization term to limit the number of leaf nodes:

$$\lambda \cdot \underbrace{f(x)^2}_{\text{measures model complexity}} + \gamma \cdot \underbrace{T}_{\text{number of leaf nodes}} + \overbrace{\text{loss}(f(x), \text{data})}^{\text{measures model fit}}.$$

How does this control the complexity of a tree? Recall that each split in a binary decision tree has two children. It’s easy to check that a binary tree of depth  $d$  will have  $(d+1)/2$  leaf nodes.

Thus, by limiting the number of leaf nodes, this additional term will force tree learning to train shallower trees, which in turn makes the trees weaker and less complex.

XGBoost uses this regularized objective function in many ways. For instance, during tree-learning, instead of using scoring function such as Gini criterion or entropy for split-finding, XGBoost uses the regularized learning objective described above. Thus, this criterion is used to determine the *structure* of the individual trees, the weak learners in the ensemble.

XGBoost also uses this objective to compute the leaf values themselves, which are essentially the regression values that gradient boosting ensembles. Thus, this criterion is used to determine the *parameters* of the individual trees as well.

An important caveat before we move on: the additional regularization term allows direct control over model complexity and downstream generalization. This comes as a price, however, in that we now have an extra parameter  $\gamma$  to worry about.

Since  $\gamma$  is a user-defined parameter, we have to set this value, along with  $\lambda$  and many others. These will often have to be selected by cross validation and can add to the overall model development time and effort.

#### **(WEIGHTED) QUANTILE-BASED NEWTON BOOSTING**

Even with a regularized learning objective, the biggest computational bottleneck is in scaling learning to large data sets, specifically, in identifying optimal splits for during learning of the regression tree base estimators.

The standard approach to tree learning exhaustively enumerates all possible splits in the data. As we've seen in Section 5.2.4, this is not a good idea for large data sets. Efficient modifications such as histogram-based splitting bin the data instead so that we evaluate far fewer splits.

Implementations such as LightGBM incorporate further improvements such as sampling and feature bundling to speed up tree learning. XGBoost also aims to bring these notions into its implementation. However, there is one key consideration unique to XGBoost.

Packages such as LightGBM implement gradient boosting, while XGBoost implements Newton boosting. This means that XGBoost's tree learning has to consider Hessian-weighted training examples, unlike LightGBM where all the examples are weighted equally!

XGBoost's approximate split finding algorithm, called weighted quantile sketch, aims to find ideal split points using quantiles in the features. This is analogous to histogram-based splitting, which uses bins, employed by gradient boosting algorithms.

The details of weighted quantile sketch and its implementation are considerable and cannot be covered here owing to limited space. However, our key takeaways are as follow:

- Conceptually, XGBoost also uses approximate split-finding algorithms; these algorithms consider additional information unique to Newton boosting (such as Hessian weights). Ultimately they are similar to histogram-based algorithms and aim to bin the data.

Unlike other histogram-based algorithms that bucket data into evenly sized bins, XGBoost bins data into feature-dependent buckets. At the end of the day, XGBoost trades off exactness for efficiency by adapting clever strategies for split finding.

- From an implementation standpoint, XGBoost pre-sorts and organizes data into blocks both in memory and on disk. Once this is done, XGBoost further exploits this organization by caching access patterns, block compression and chunking the data into easily accessible shards. These features significantly improve the efficiency of Newton boosting, allowing it to scale to very large data sets.

### **6.3.2 Newton Boosting with XGBoost**

We kick off our explorations of XGBoost with the breast cancer data set, which we have used several times in the past as a pedagogical data set.

```

from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
X, y = load_breast_cancer(return_X_y=True)
Xtrn, Xtst, ytrn, ytst = train_test_split(X, y, test_size=0.2,
                                         shuffle=True, random_state=42)

```

For Python users, especially those who are familiar with `scikit-learn`, XGBoost provides a familiar interface that is designed to look and feel like `scikit-learn`. Using this interface, it is very easy to setup and train an XGBoost model:

```

from xgboost import XGBClassifier
ens = XGBClassifier(n_estimators=20, max_depth=1,
                   objective='binary:logistic')
ens.fit(Xtrn, ytrn)

```

We set the loss function to be the logistic loss, the number of iterations (with one estimator trained per iteration) to 20, and maximum tree depth to be 1. This results in an ensemble of 20 decision stumps (trees of depth 1).

It is also similarly easy to predict labels on test data and evaluate model performance:

```

from sklearn.metrics import accuracy_score
ypred = ens.predict(Xtst)
accuracy_score(ytst, ypred)
0.9649122807017544

```

Alternately, it is also possible to use XGBoost's native interface, which was originally designed to read data in the LibSVM format, which is well-suited for storing sparse data with lots of zeros efficiently.

In this format (which was introduced in the case study in Section 5.4), each line of the data file contains a single training example represented as:

```

<label> qid:<example id> 1:<feature 1 value> 2:<feature 2 value> ... k:<feature k value> ...
# other information as comments

```

XGBoost uses a data object called `DMatrix` to group data and corresponding labels together. `DMatrix` objects can be created by reading data directly from files or from other array-like objects. Here, we create two `DMatrix` objects called `trn` and `tst` to represent the train and test data matrices.

```

import xgboost as xgb
trn = xgb.DMatrix(Xtrn, label=ytrn)
tst = xgb.DMatrix(Xtst, label=ytst)

```

We also set up the training parameters using a dictionary and train an XGBoost model using the `DMatrix` `trn` and the parameters.

```

params = {'max_depth': 1, 'objective': 'binary:logistic'}
ens2 = xgb.train(params, trn, num_boost_round=20)

```

Care must be taken while using this model for prediction, however. Models trained with certain loss functions will return prediction probabilities rather than the predictions directly. The logistic loss function is one such case.

These prediction probabilities can be converted to binary classification labels 0/1 by thresholding at 0.5. That is, all test examples with prediction probability  $\geq 0.5$  are classified into Class 1 and the rest into Class 0:

```
ypred_proba = ens2.predict(tst)
ypred = (ypred_proba >= 0.5).astype(int)
accuracy_score(ytst, ypred)
0.9649122807017544
```

Finally, XGBoost supports three different types of boosting approaches, which can be set through the `booster` parameter:

- `booster='gbtree'` is the default setting and implements Newton boosting using trees as weak learners trained using tree-based regression
- `booster='gblinear'` implements Newton boosting using linear functions as weak learners trained using linear regression
- `booster='dart'` trains an ensemble using DART, or Dropout meets Multiple Additive Regression Trees (this was previously described in Section 5.4)
- it is possible to train (parallel) random forest ensembles using XGBoost. By carefully setting the training parameters to ensure training example and feature subsampling, tree boosting can learn a random forest. This is generally only useful when you wish to leverage XGBoost's parallel and distributed training architecture to explicitly train parallel ensembles.

## 6.4 XGBoost in Practice

In this section, we describe how to train models in practice using XGBoost. As with AdaBoost and gradient boosting, we look to set the learning rate (Section 6.4.1) or employ early stopping (Section 6.4.2) as a means to control overfitting. To recap,

- by selecting an effective learning rate, we try to control the rate at which the model learns so that it doesn't rapidly fit, and then overfit the training data. We can think of this a proactive modeling approach, where we try to identify a good training strategy so that it leads to a good model.
- by enforcing early stopping, we try to stop training as soon as we observe that the model is starting to overfit. We can think of this as a reactive modeling approach, where we contemplate terminating training as soon as we think we have a good model.

### 6.4.1 Learning Rate

Recall from Section 6.1 that the step length is analogous to the learning rate and is a measure of each weak learner's contribution to the entire ensemble. The learning rate allows greater control over how quickly the complexity of the ensemble grows.

Therefore, it is essential that we identify the best learning rate for our data set in practice so that we can avoid overfitting and generalize well after training.

### LEARNING RATE VIA CROSS VALIDATION

As we've seen in the last section, XGBoost provides an interface that plays nicely with `scikit-learn`. In this subsection, we see how we can combine functionalities of both packages to effectively perform parameter selection using cross validation.

While we use cross validation to set the learning rate here, cross validation can be used to select other learning parameters such as maximum tree depth, number of leaf nodes and even loss function specific parameters.

We combine `scikit-learn`'s `StratifiedKFold` class to split the training data into 10 folds of training and validation sets. `StratifiedKFold` ensures that we preserve class distributions, that is, the fractions of different classes across the folds.

First we initialize the learning rates we are interested in exploring:

```
import numpy as np
learning_rates = np.concatenate([np.linspace(0.02, 0.1, num=5),
                                np.linspace(0.2, 1.8, num=9)])
n_learning_rate_steps = len(learning_rates)
print(learning_rates)
[0.02 0.04 0.06 0.08 0.1 0.2 0.4 0.6 0.8 1. 1.2 1.4 1.6 1.8 ]
```

Next, we setup `StratifiedKFold` to split the training data into 10 folds:

```
from sklearn.model_selection import StratifiedKFold
n_folds = 10
splitter = StratifiedKFold(n_splits=n_folds, shuffle=True, random_state=42)
```

In the listing below, we perform cross validation by training and evaluating models on each of the 10 folds with XGBoost.

#### Listing 6.5. Cross Validation with XGBoost and `scikit-learn`

```
trn_err = np.zeros((n_learning_rate_steps, n_folds))
val_err = np.zeros((n_learning_rate_steps, n_folds)) #A

for i, rate in enumerate(learning_rates): #B
    for j, (trn, val) in enumerate(splitter.split(X, y)):
        gbm = XGBClassifier(n_estimators=10, max_depth=1,
                            learning_rate=rate, verbosity=0)
        gbm.fit(X[trn, :], y[trn])

        trn_err[i, j] = (1 - accuracy_score(y[trn], #C
                                           gbm.predict(X[trn, :]))) * 100
        val_err[i, j] = (1 - accuracy_score(y[val],
                                           gbm.predict(X[val, :]))) * 100

trn_err = np.mean(trn_err, axis=1) #D
val_err = np.mean(val_err, axis=1) #D
```

#A to save training & validation errors

#B Train a XGBoost classifier for each fold with different learning rates

#C save training and validation errors

#D average training & validation errors across folds

When applied to the breast cancer data set (see Section 6.3.2), we obtain the averaged training and validation errors for this data set. We visualize these for different learning rates below.

As learning rate decreases, XGBoost's performance degrades as the boosting process becomes increasingly more conservative and exhibits underfitting behavior.

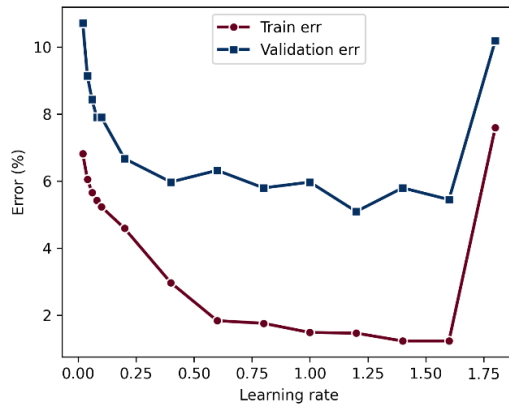


Figure 6.14. Averaged training and validation errors of XGBoost across 10 folds of the breast cancer data set.

As learning rate increases, XGBoost's performance, once again, degrades as the boosting process becomes increasingly more aggressive and exhibits overfitting behavior. The best value among our parameter choices appears to be `learning_rate=1.2`, and generally in the region between 1.0 and 1.5.

#### CROSS VALIDATION WITH XGBOOST

Beyond parameter selection, cross validation can also be useful to characterize model performance. In the listing below, we use XGBoost's built-in CV functionality to characterize how XGBoost's performance changes as we increase the number of estimators in the ensemble.

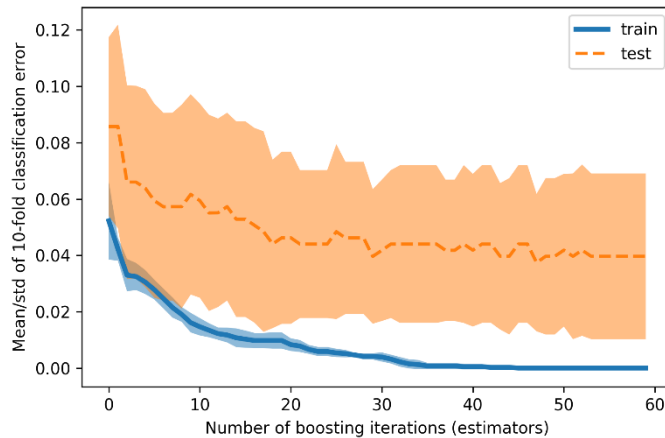
We use the `XGBoost.cv` function to perform 10-fold cross validation. Observe that `XGBoost.cv` is called in nearly the same way as `XGBoost.train` from the previous section.

#### Listing 6.6. Cross Validation with XGBoost

```
import xgboost as xgb
trn = xgb.DMatrix(Xtrn, label=ytrn)
tst = xgb.DMatrix(Xtst, label=ytst)

params = {'learning_rate': 0.25, 'max_depth': 2,
          'objective': 'binary:logistic'}
cv_results = xgb.cv(params, trn, num_boost_round=60,
                    nfold=10, metrics={'error'}, seed=42)
```

In this listing, the model performance is characterized by error, which is passed to `XGBoost.cv` using the argument `metrics={'error'}`.



**Figure 6.15.** The average error across the folds decreases with increasing iterations, as we add more and more base estimators into the ensemble

Another interesting observation from the plot above is that training as well as validation performance stop improving meaningfully at around 35 iterations. This suggests that there is no significant performance improvement to be gained by prolonging training beyond this point.

This brings us, rather neatly, to the notion of early stopping, which we have encountered before with both AdaBoost and gradient boosting.

### 6.4.2 Early Stopping

As the number of base estimators in the ensemble increases, the complexity of the ensemble also increases, which eventually leads to overfitting. To avoid this, what if, instead of training the model, we stopped before we reached the limit of ensemble size?

Early stopping with XGBoost works pretty similarly to LightGBM, where we specify a value for the parameter `early_stopping_rounds`. The performance of the ensemble is scored after each iteration on a *validation set*, which is split from the training set for the purpose of identifying a good early stopping point.

As long as the overall score (say accuracy) improves over the last `early_stopping_rounds`, XGBoost will continue to train. However, when the score does not improve after `early_stopping_rounds`, XGBoost terminates training.

The listing below illustrates early stopping using XGBoost. Note that `train_test_split` is used to create an independent validation set that is used by XGBoost to identify an early stopping point.



**Listing 6.7: Early Stopping with XGBoost**

```

from sklearn.model_selection import train_test_split
Xtrn, Xval, ytrn, yval = train_test_split(X, y, test_size=0.2,
                                       shuffle=True, random_state=42)
ens = XGBClassifier(n_estimators=50, max_depth=2,
                   objective='binary:logistic')
ens.fit(Xtrn, ytrn, early_stopping_rounds=5,
        eval_set=[(Xval, yval)], eval_metric='auc')

```

The three key parameters above for early stopping are the evaluation set that is used to determine the early stopping point are `early_stopping_rounds=5`, `eval_set=[(Xval, yval)]` and the evaluation metric: `eval_metric='auc'`.

With these parameters, training terminates after 11 rounds even though `n_estimators` was initialized to 50 in `XGBClassifier`. Thus, early stopping can greatly improve training times, while ensuring that model performance does not degrade excessively.

```

[0] validation_0-auc:0.95480
[1] validation_0-auc:0.96725
[2] validation_0-auc:0.96757
[3] validation_0-auc:0.99017
[4] validation_0-auc:0.99099
[5] validation_0-auc:0.99181
[6] validation_0-auc:0.99410
[7] validation_0-auc:0.99640
[8] validation_0-auc:0.99476
[9] validation_0-auc:0.99148
[10] validation_0-auc:0.99050
[11] validation_0-auc:0.99050

```

## 6.5 Case Study Redux: Document Retrieval

To conclude this chapter, we revisit the case study from Chapter 5 that addressed the task of document retrieval, which identifies and retrieves documents from a database to match a user's query. In Chapter 5, we compared several gradient boosting approaches available in LightGBM.

In this chapter, we will train Newton boosting models using XGBoost on the document retrieval task and compare the performance of XGBoost and LightGBM. In addition to this comparison, this case study also illustrates how to set up randomized cross validation for effective parameter selection in XGBoost over large data sets.

### 6.5.1 The LETOR Data Set

We use the LEarning TO Rank (LETOR) ver. 4.0 data set, which is made freely available by Microsoft Research. Each training example corresponds to a query-document pair, with features describing the query, the document, and the matches between them. Each training label is a relevance rank: least relevant, moderately relevant, or highly relevant.

This problem is set up as a 3-class classification problem of identifying the relevance class (least, moderately, or highly relevant) given a training example: a query-document pair.

For purposes of convenience and consistency, we will use the functionalities provided by XGBoost's `scikit-learn` wrapper along with modules from `scikit-learn` itself.

First let's load the LETOR data set.

```
from sklearn.datasets import load_svmlight_file
from sklearn.model_selection import train_test_split
import numpy as np

query_data_file = './data/ch05/MQ2008/Querylevelnorm.txt'
X, y = load_svmlight_file(query_data_file)
```

Next, let's split this into train and test sets.

```
Xtrn, Xtst, ytrn, ytst = train_test_split(X, y,
                                         test_size=0.2, random_state=42)
```

## 6.5.2 Document Retrieval with XGBoost

As we have a 3-class (multiclass) classification problem, we train a tree based XGBoost classifier using the softmax loss function.

The softmax loss is a generalization of the logistic loss function to multiclass classification and is commonly used in many multi-class learning algorithms including multinomial logistic regression and in deep neural networks.

```
xgb = XGBClassifier(booster='gbtree', objective='multi:softmax')
```

As with LightGBM, XGBoost also requires that we set several learning parameters such as learning rate (to control the rate of learning) or the number of leaf nodes (to control the complexity of the base estimator trees).

These parameters are selected using `scikit-learn`'s randomized cross validation module: `RandomizedSearchCV`. Specifically, we perform 5-fold cross validation over a grid of various parameter choices; however, instead of exhaustively evaluating all possible learning parameter combinations the way `GridSearchCV` does, `RandomizedSearchCV` samples a smaller number of model combinations for faster parameter selection.

```
num_random_iters = 20
num_cv_folds = 5
```

We can explore several different values of some key parameters described below:

- `learning_rate`, which controls the overall contribution of each tree to the overall ensemble,
- `max_depth`, which limits tree depth to accelerate training and decrease complexity,
- `min_child_weight`, which limits each leaf node by the sum of Hessian values to control overfitting,
- `colsample_bytree`, which specifies the fraction of features to sample from the training data respectively, to accelerate training,
- `reg_alpha` and `reg_lambda`, which specify the amount of regularization of the leaf node values, to control overfitting as well.

The code below specifies the ranges of values for these parameters we are interested in searching over to identify an effective training parameter combination.

```
from scipy.stats import randint, uniform
xgb_params = {'max_depth': randint(2, 10),
              'learning_rate': 2**np.linspace(-6, 2, num=5),
              'min_child_weight': [1e-2, 1e-1, 1, 1e1, 1e2],
              'colsample_bytree': uniform(loc=0.4, scale=0.6),
              'reg_alpha': [0, 1e-1, 1, 10, 100],
              'reg_lambda': [0, 1e-1, 1, 10, 100]}
```

As mentioned above, the grid over these parameters produces too many combinations to evaluate efficiently. Thus, we adopt randomized search with cross validation and randomly sample a much smaller number of parameter combinations.

```
cv = RandomizedSearchCV(estimator=xgb,
                       param_distributions=xgb_params,
                       n_iter=num_random_iters,
                       cv=num_cv_folds,
                       refit=True,
                       random_state=42, verbose=1)
cv.fit(Xtrn, ytrn, eval_set=[(Xtst, ytst)],
      eval_metric='merror', verbose=False)
```

Observe that we have set `refit=True` in `RandomizedSearchCV`, which enables the training of one final model using the optimal parameter combination identified by `RandomizedSearchCV`.

After training, we compare the performance of XGBoost with four models trained by LightGBM in Section 5.5:

- random forest: parallel homogeneous ensemble of randomized decision trees;
- gradient boosted decision trees (GBDT): this is the standard approach to gradient boosting and represents a balance between models with good generalization performance and training speed;
- gradient boosting with gradient one-side sampling (GOSS): this variant of gradient boosting downsamples the training data and is ideally suited for large data sets; due to downsampling, it may lose out on generalization, but is typically very fast to train;
- Dropout meets Multiple Additive Regression Trees (DART): this variant incorporates the notion of dropout from deep learning, where neural units are randomly and temporarily dropped during backpropagation iterations to mitigate overfitting. DART is often the slowest of all the gradient boosting options available in LightGBM.

XGBoost uses regularized loss functions and Newton boosting. In contrast, random forest does not use any gradient information, while GBDT, GOSS and DART use gradient boosting.

As before, we compare performance of all algorithms using test set accuracy (left) and overall model development time (right), which includes cross-validation-based parameter selection as well as training time.

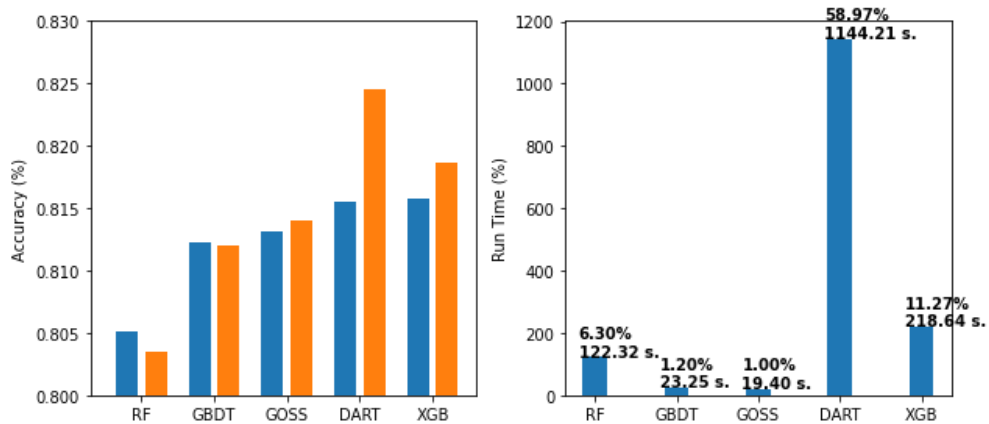


Figure 6.16. (left) Comparing test set accuracy of random forest, GBDT, GOSS and DART; (right) comparing the overall training times of random forest, GBDT, GOSS and DART (all trained using LightGBM).

From the figure above, the key takeaways from this experiment are:

- On training performance, XGBoost performs comparably to DART, GOSS and GBDT and outperforms random forest. On test set performance XGBoost is second only to DART.
- On training time, the overall model development time of XGBoost is significantly smaller than DART. This suggests that there is an application-dependent tradeoff to be made here between the need for the additional performance improvement and the accompanying computational overhead incurred.
- Finally, it should be noted that these results are dependent on various choices made during modeling such as learning parameter ranges and randomization. Further performance gains are possible with careful feature engineering, loss function selection and leveraging distributed processing for efficiency.

## 6.6 Summary

In this chapter, we were introduced to a new sequential ensemble approach: Newton boosting. Some key takeaways:

- Newton descent is another optimization algorithm, similar to gradient descent.
- Newton descent uses second-order (Hessian) information to accelerate optimization in comparison to gradient descent, which only uses first-order (gradient information).
- Newton boosting combines Newton descent and boosting to learn a sequential ensemble of weak learners.
- Newton boosting uses weighted residuals to characterize correctly classified and poorly classified training examples. This is analogous to both how AdaBoost uses weights and gradient boosting uses residuals.
- Weak learners in Newton boosting are regression trees that are trained over the

weighted residuals of the training examples and approximate the Newton step.

- Like gradient boosting, Newton boosting can be applied to a wide variety of loss functions arising from classification, regression, or ranking tasks.
- Optimizing a regularized loss function helps control the complexity of the weak learners in the learned ensemble, prevent overfitting and improve generalization.
- XGBoost is a powerful, publicly available framework for tree-based Newton boosting that incorporates Newton boosting, efficient split finding, and distributed learning.
- XGBoost optimizes a regularized learning objective consisting of the loss function (to fit the data) and two regularization functions: L2-regularization and number of leaf nodes.
- As with AdaBoost and gradient boosting, we can avoid overfitting in Newton boosting by choosing an effective learning rate or via early stopping. XGBoost supports both.
- XGBoost implements an approximate split-finding algorithm called weighted quantile sketch, which is similar to histogram-based split finding, but adapted and optimized for efficient Newton boosting.
- In addition to a wide variety of loss functions for classification, regression, and ranking, XGBoost also provides support for incorporation of our own custom, problem-specific loss functions for training.

# 7

## Learning with Continuous and Count Labels

### This chapter covers

- An introduction to regression in machine learning
- Understanding loss and likelihood functions for regression
- Understanding when to use different loss and likelihood functions
- Adapting parallel and sequential ensembles for regression problems
- Using ensembles for regression in practical settings

Many real-world modeling, prediction and forecasting problems are best framed and solved as regression problems. Regression has a rich history predating the advent of machine learning and has long been a part of the standard statistician's toolkit.

Regression techniques have been developed and widely applied in many areas. Here are just a few examples:

- Weather forecasting: to predict the precipitation tomorrow using data from today, including temperature, humidity, cloud cover, wind and more.
- Insurance analytics: to predict the number of automobile insurance claims over a period of time, given various vehicle and driver attributes.
- Financial forecasting: to predict stock prices using historical stock data and trends.
- Demand forecasting: predict the residential energy load for the next three months using historical, demographic and weather data.

Whereas Chapters 2-6 introduced ensembling techniques for classification problems, in this chapter, we will see how to adapt ensembling techniques to regression problems.

Consider the task of detecting fraudulent credit card transactions. This is a *classification problem* because we're aiming to distinguish between two types of transactions: fraudulent

(with class label, say 1) and not fraudulent (with class label, say 0). The labels (or targets) we want to predict in classification are *categorical* (0, 1, ...) and represent different categories.

On the other hand, consider the task of predicting a cardholder's monthly credit card balance. This is an instance of a *regression task*. Unlike classification, the labels (or targets) we want to predict take *continuous* values (e.g., \$650.35).

Consider yet another task of predicting the number of times a cardholder uses their card every week. This is also an instance of a regression task, though with a subtle difference. The labels, or targets we want to predict are *counts*.

We typically distinguish between *continuous regression* and *count regression* as it doesn't always make sense to model counts as continuous values (for instance, what does it even mean to predict that a cardholder used their card 7.62 times?)

In this chapter, we will learn about these types of problems and others that can be modeled with regression, and how we can train regression ensembles.

Section 7.1 introduces regression formally, shows some commonly used regression models, and how regression can be used to model continuous and count-valued labels (and even categorical labels) under a single framework called the Generalized Linear Model.

Sections 7.2 (parallel ensembles for regression) and 7.3 (sequential ensembles for regression) show how we can adapt ensemble methods to regression problems.

Section 7.3 introduces loss and likelihood functions for continuous and count-valued targets and provides guidelines on when and how to use them. We conclude with a case study in Section 7.4, this time from the realm of demand forecasting.

## 7.1 A Brief Review of Regression

This section reviews the terminology and background material for regression. We begin with the more familiar and traditional framing of regression as learning with continuous labels.

We will then learn about Poisson regression, an important technique for learning with count labels, and logistic regression, another important technique for learning with categorical labels.

In particular, we will see that linear, Poisson and logistic regression are all individual variations within a framework called Generalized Linear Models or GLMs.

We also briefly review two important nonlinear regression methods: decision tree regression and artificial neural networks, as they are both often used as base estimators or meta-estimators in ensemble methods.

### 7.1.1 Linear Regression for Continuous Labels

The most fundamental regression method is *linear regression*, where the model to be trained is a linear, weighted combination of the input features, that is

$$f(\mathbf{x}) = w_0 + w_1x_1 + \dots + w_dx_d = w_0 + \mathbf{w}'\mathbf{x}.$$

The linear regression model  $f(\mathbf{x})$  takes an example  $\mathbf{x}$  as input and is parameterized by

feature weights,  $\mathbf{w}$  and the intercept (sometimes also known as the bias)  $w_0$ . This model is trained by identifying the weights that minimize the (mean) squared error between the true labels ( $\mathbf{w}$ ) and predicted labels ( $f(\mathbf{x}_i)$ ) over all  $n$  training examples, where

$$\text{squared error/loss} = \frac{1}{2n} \sum_{i=1}^n (y_i - f(\mathbf{x}_i))^2 = \frac{1}{2n} \sum_{i=1}^n (y_i - \mathbf{w}'\mathbf{x}_i - w_0)^2.$$

The (mean) squared error is nothing but the (mean) squared loss. Since we minimize the loss function to learn the model, linear regression is also known by another name that may be familiar to you: *ordinary least squares (OLS) regression*.

Recall from Section 6.2 (and also from Chapter 1) that most machine-learning problems can be cast as combinations of regularization and loss functions, where the regularization function controls model complexity, and the loss function controls model fit:

$$\text{learning objective} = \lambda \cdot \underbrace{\text{regularization}(f)}_{\text{measures model complexity}} + \underbrace{\text{loss}(f, \text{data})}_{\text{measures model fit}}.$$

$\lambda$ , of course, is the regularization parameter that trades off between fit and complexity and must be determined and set by the user, typically through practices such as cross validation.

Optimizing (specifically, minimizing) this learning objective essentially amounts to training a model. From this perspective, *ordinary least squares regression can be framed as an unregularized learning problem* where only the squared-loss function is optimized,

$$\text{OLS learning objective} = \lambda \cdot \underbrace{0}_{\text{measures model complexity}} + \underbrace{\text{squared loss}}_{\text{measures model fit}}.$$

Is it possible to use different regularization functions to come up with other linear regression methods? Absolutely, and this is precisely what the statistics community has been up to for the better part of the last century.

#### COMMON LINEAR REGRESSION METHODS

Let's see some common linear regression methods in practice through `scikit-learn's` `linear_model` subpackage, which implements several linear regression models.

We'll use a synthetic data set, where the true underlying function is given by  $f(x) = -25x + 3.2$ . This is a univariate function, or a function of one variable (for our purposes, one feature). In practice, we will often not know the true underlying function, of course. The following code snippet generates a small, noisy data set of 100 training examples.

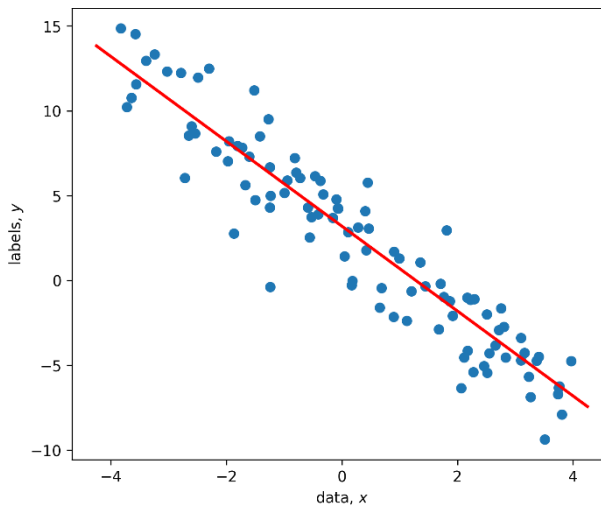


```

n = 100
X = np.random.uniform(low=-4.0, high=4.0, size=(n, 1))
f = lambda x: -2.5 * x + 3.2
y = f(X) + np.random.normal(scale=0.15 * np.max(y), size=(n, 1))

```

We can visualize this data set in the figure below.



**Figure 7.1.** Data for a synthetic regression problem to which we fit several linear regression models, generated by the univariate (one-dimensional), noisy function  $f(x) = -2.5x + 3.2$  (shown by the red line)

Different regularization methods serve different modeling needs and can handle different types of data issues. The most common data issue that linear regression models must contend with is that of *multicollinearity*.

Multicollinearity in data arises when one feature depends on others, that is, when the features are *correlated with each other*. For example, in medical data, patient weight and blood pressure are often highly correlated. In practical terms, this means that both features convey nearly the *same information*, and it should be possible to train a less complex model by selecting and using only one of them.

To understand the impact of different regularization methods, we'll explicitly create a data set with multicollinearity using our recently generated univariate data. Specifically, we will create a data set with two features, where one feature is dependent on the other:

```

X = np.concatenate([X, 3*X + 0.25*np.random.uniform(size=(n, 1))], axis=1)

```

This produces a data set of two features, where the second feature is 3 times the first (with some added random noise, to keep it more realistic). We now have a two-dimensional data set, where the second feature is highly correlated with the first. As before, we'll split the data set into training (75%) and test (25%) sets:

```
from sklearn.model_selection import train_test_split
Xtrn, Xtst, ytrn, ytst = train_test_split(X, y, test_size=0.25)
```

We now train four commonly used linear regression models:

- Ordinary least squares (OLS) regression, with no regularization (as seen above)
- Ridge regression, which uses  $L_2$ -regularization
- LASSO, which uses  $L_1$ -regularization, and
- Elastic net, which uses a combination of  $L_1$  and  $L_2$ -regularization

The listing below initializes and trains all 4 models.

### Listing 7.1 Linear regression models

```
from sklearn.linear_model import LinearRegression, Ridge, Lasso, ElasticNet
from sklearn.metrics import mean_squared_error, mean_absolute_error

models = ['OLS Regression', 'Ridge Regression', 'LASSO', 'Elastic Net']
regressors = [LinearRegression(),      #A
              Ridge(alpha=2.0),
              Lasso(alpha=2.0),
              ElasticNet(alpha=2.0, l1_ratio=0.5)]

for (model, regressor) in zip(models, regressors):
    regressor.fit(Xtrn, ytrn)      #B
    ypred = regressor.predict(Xtst)  #C
    mse = mean_squared_error(ytst, ypred) #D
    mad = mean_absolute_error(ytst, ypred)

    print('{0}\n's test set performance: MSE = {1:4.3f}, MAD={2:4.3f}'
          .format(model, mse, mad))
    print('{0} model: {1} * x + {2}\n'      #E
          .format(model, regressor.coef_, regressor.intercept_))
```

#A initialize four common linear regression models

#B train regression model

#C get predictions on the test set

#D compute the test error using MSE and MAD

#E print the regression weights

The unregularized OLS model will serve as our baseline for comparing the others.

```
OLS Regression's test set performance: MSE = 2.622, MAD=1.230
OLS Regression model: [[-15.97039936  4.49467661]] * x + [2.50949892]
```

We'll use two metrics to evaluate the performance of each model: mean squared error (MSE) and mean absolute deviation (MAD). This model has MSE of 2.622 and MAD of 1.23.

The next linear regression model, *ridge regression*, uses  $L_2$ -regularization, which is just the sum-of-squares of the weights, that is,

$$\text{ridge regression learning objective} = \lambda \cdot \underbrace{\frac{1}{2}(w_1^2 + \dots + w_d^2)}_{\text{measures model complexity}} + \overbrace{\text{squared loss.}}^{\text{measures model fit}}$$

So, what does  $L_2$ -regularization do? Learning involves minimizing the learning objective; when the regularization term, or sum of squares is minimized, it pushes individual weights to zero. This is known as *shrinkage* of the model weights, which reduces model complexity.

The squared loss term in the objective is critical because, without it, we would learn a degenerate model with all-zero weights. Thus, a ridge regression model trades off complexity to fit, the balance of which is controlled by appropriately setting the parameter  $\lambda > 0$ .

Listing 7.1 produces the following ridge regression model (with  $\lambda = 2.0$ ):

```
Ridge Regression's test set performance: MSE = 2.364, MAD=1.139
Ridge Regression model: [[-0.55652638 -0.64978444]] * x + [3.17809807]
```

The effect of regularization and the resultant shrinkage is immediately evident when we compare the weights learned by  $L_2$ -regularized ridge regression:  $[-0.557, -0.65]$  to those learned by unregularized OLS regression:  $[-15.97, 4.494]$ .

What's more, the ridge regression model improves test set performance as evidenced by the improvement in MSE (2.622 to 2.364) and MAD (1.230 to 1.139) between the two.

Another popular linear regression method is *Least Absolute Shrinkage and Selection (LASSO)*, which is rather similar to ridge regression except that it uses  $L_1$ -regularization to control model complexity. That is, the learning objective with  $L_1$ -regression becomes

$$\text{LASSO learning objective} = \lambda \cdot \underbrace{(|w_1| + \dots + |w_d|)}_{\text{measures model complexity}} + \overbrace{\text{squared loss.}}^{\text{measures model fit}}$$

$L_1$ -regularization is the sum of absolute values of the weights, rather than the sum of squares in  $L_2$ -regularization. The effect, overall, is similar to  $L_2$ -regularization, except that  $L_1$ -regularization shrinks the weights for less predictive features. In contrast,  $L_2$ -regularization shrinks the weights for all the features uniformly.

Put another way,  $L_1$ -regularization pushes the weights of less informative features down to zero, which makes it well suited for feature selection.  $L_2$ -regularization pushes the weights of all features down together, which makes it well suited for handling correlated and covariant features.

Listing 7.1 produces the following LASSO model (with  $\lambda = 20$ ):

```
LASSO's test set performance: MSE = 2.327, MAD=1.135
LASSO model: [-0.          -0.79667616] * x + [3.19674495]
```

Contrast the LASSO model's weights:  $[0, -0.797]$ , to those learned by ridge regression:  $[-0.557, -0.65]$ : LASSO has actually learned a zero-weight for the first feature!

We can see that  $L_1$ -regularization induces *model sparsity*. That is, LASSO performs implicit feature selection during learning to identify a small set of features needed to build a less complex model, while maintaining or even improving performance.

Put another way, there are two clear benefits of the model trained by LASSO compared to the one trained by OLS. First, LASSO achieves a smaller test error (2.327 vs. 2.622 MSE, 1.135 vs. 1.23 MAE), which means it will generalize and perform better on future data.

Second, the LASSO model only depends on one feature, while the OLS model requires two. This makes the LASS model less complex than the OLS model. While this may not mean much for this toy data set, this has significant scalability implications when deployed for a data set that has thousands of features.

Recall that our synthetic data set was carefully constructed to have two highly correlated features. LASSO has identified this and determined that it does not require both and hence learned a zero weight for one, effectively, zeroing out its contribution to the final model.

The final linear regression model we will look at is called the *Elastic Net*, a celebrated, widely used, and well-studied model. Elastic net regression uses a combination of both  $L_1$  and  $L_2$ -regularization:

$$\text{elastic net objective} = \lambda \cdot \underbrace{\alpha (|w_1| + \dots + |w_d|)}_{\text{measures model complexity}} + \underbrace{\frac{b}{2} (w_1^2 + \dots + w_d^2)}_{\text{measures model fit}} + \text{squared loss.}$$

The proportions of  $L_1$  and  $L_2$ -regularizers in the overall regularization are controlled by parameters,  $a, b \geq 0$ , while the parameter  $\lambda > 0$  still controls the tradeoff between the overall regularization and the loss function.

Note that in elastic net, rather than set  $a$  and  $b$  directly, we instead need to set  $\text{alpha} = a + b$  and  $\text{l1\_ratio} = a / (a + b)$ . Listing 7.1 produces the following elastic net model:

```
Elastic Net's test set performance: MSE = 2.326, MAD=1.135
Elastic Net model: [-0.          -0.80058507] * x + [3.19729763]
```

As we see from the results, the elastic net model still has the sparsity inducing characteristics of LASSO (observe the first learned weight is zero), while incorporating the robustness of ridge regression to correlations in the data (compare the test set performances of ridge regression and elastic net).

The table below summarizes several common linear regression models, all of which can be cast into the squared loss + regularization framework above.

Model	Loss function	Regularization	Comments
Ordinary Least Squares Regression	Squared loss $(y - f(x))^2$	None	Classical linear regression; becomes unstable with highly correlated features
Ridge Regression	Squared loss $(y - f(x))^2$	$L_2$ -penalty $\frac{1}{2}(w_1^2 + \dots + w_d^2)$	Shrinks the weights to control model complexity, encourages robustness to highly correlated features
LASSO	Squared loss $(y - f(x))^2$	$L_1$ -penalty $ w_1  + \dots +  w_d $	Shrinks the weights even more and encourages sparse models and performs implicit feature selection
Elastic Net	Squared loss $(y - f(x))^2$	$\alpha L_1 + (1 - \alpha)L_2$ $0 \leq \alpha \leq 1$	Weighted combination of both regularizers to balance between sparsity and robustness

**Table 7-1.** Four popular linear regression methods: they all use the squared-loss function, though different approaches to regularization that contributes to model robustness and sparsity.

During model training, these (regularized) loss functions are often optimized through +gradient descent, Newton descent or their variants as seen in Chapters 5.1 and 6.1.

All the linear regression methods in Table 1 use the squared loss. Other regression methods can be derived using different loss functions. We'll see examples in Section 7.3, and again in the Chapter case study in 7.4.

### 7.1.2 Poisson Regression for Count Labels

The previous section introduced regression as a machine-learning approach suited for modeling problems with continuous-valued targets (labels). There are often situations, however, where we have to develop models where the labels are *counts*.

In health informatics, for instance, we may wish to build a model to predict the number (essentially, the count) of doctor visits given patient data. In insurance pricing, a common problem is that of modeling claim frequency, to predict the count of how many insurance claims we can expect for different types of insurance policies.

In urban planning, we may want to model different count variables for census regions, such as household size, number of crimes, number of births and deaths, and many more.

In all of these problems, we are still interested in building a regression model of the form  $y = f(x)$ , but the target label  $y$  is no longer a continuous value, but a count.

#### ASSUMPTIONS OF CONTINUOUS-VALUED REGRESSION MODELS

One approach is to simply treat counts as continuous values, but this does not always work.

For one, continuous-valued predictions of count variables cannot always be interpreted meaningfully. Consider that we were predicting the number of doctor visits per patient: a prediction of 2.3 visits is not really helpful: is it two visits or three?

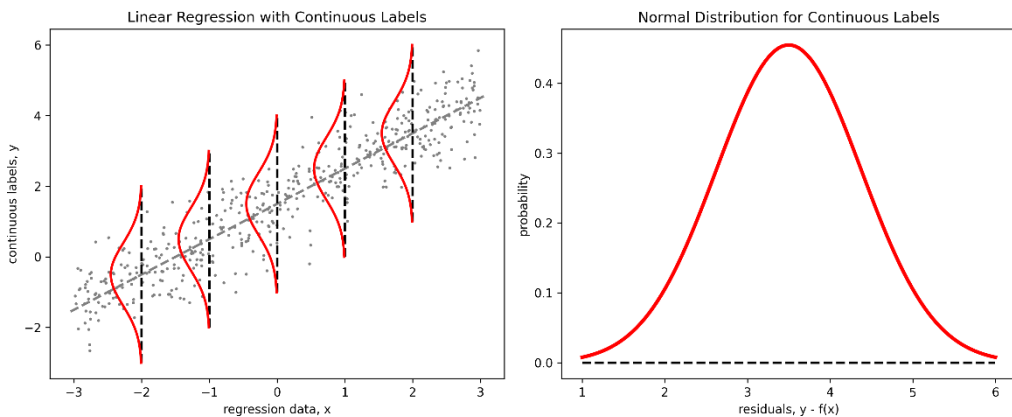
What is worse, a continuous-valued predictor may even predict negative values that may be completely meaningless. What does  $-4.7$  visits to a doctor even mean? This discussion shows that continuous and count-valued targets mean completely different things and should be treated differently.

First, let's look at how linear regression fits continuous-valued targets. Figure 7.2 (left) shows a (noisy) univariate data set, where the continuous-valued label ( $y$ ) depends on a single feature ( $x$ ).

A linear regression model assumes that for an input  $x$ , the *prediction errors or residuals* ( $y - f(x)$ ) are distributed according to the *normal distribution*. In Figure 7.2 (left), we overlay several such normal distributions on the data and labels and the linear regression model (the dotted line).

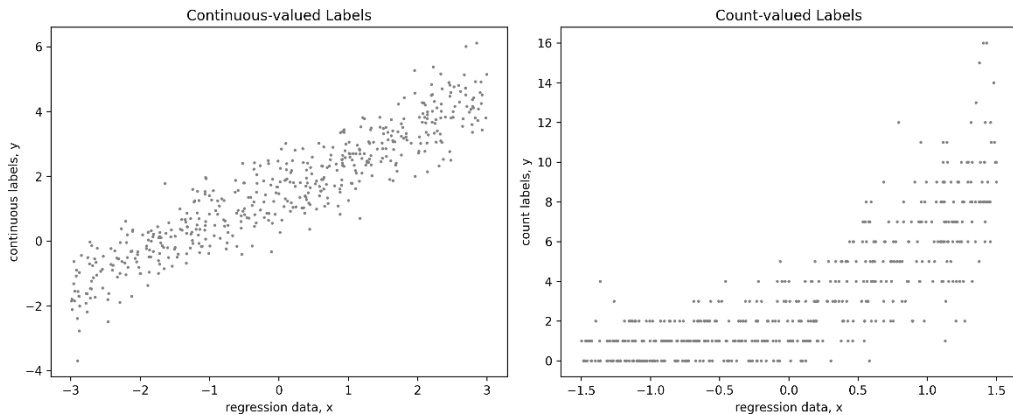
To put it simply, linear regression tries to fit a linear model such the errors have a normal distribution. The normal distribution, also called the Gaussian distribution, is a *probability distribution*, or a mathematical description of the spread and shape of the possible values a (random) variable can take.

As we can see in Figure 7.2 (right), the normal distribution is a continuous-valued distribution and reasonable choice for continuous-valued labels.



**Figure 7.2.** Linear regression (left) fits continuous-valued targets by assuming that the *spread of the targets* can be modeled by the continuous-valued normal distribution (right). More precisely, linear regression assumes that the predictions  $f(x)$  for an example  $x$  are distributed according to the normal distribution.

But what of count data? In Figure 7.3, we visualize the difference between our data set from Figure 7.2 with continuous-valued targets (left) and count-valued targets (right).



**Figure 7.3.** Visualizing the differences between continuous-valued targets (left) and count-valued targets (right) shows us that linear regression will not work well as the distribution (spread and shape) of the count labels is quite different from that of continuous labels.

We begin to see some rather stark differences between continuous-valued and count-valued labels. Intuitively, a regression model designed for continuous targets would struggle to build a viable model with count-valued targets.

This is because regression models for continuous targets assume that the residuals have a certain shape: the normal distribution. As we see below, count-valued targets are *not* normally distributed, but, in fact, often follow a Poisson distribution.

Because count-valued labels are fundamentally different from continuous-valued labels, a regression approach designed for continuous-valued labels will not generally work well on count-valued labels.

#### NEW ASSUMPTIONS FOR COUNT-VALUED REGRESSION MODELS

Can we keep the general framework of linear regression, then, but extend it to be able to handle count-valued data? We can indeed:

- We will have to change how we *link* the label (prediction target) to the input features. Linear regression relates labels to features through a linear function:  $y = \beta_0 + \beta'x$ . For count labels, we will introduce a *link function*  $g(y)$  into the model:  $g(y) = \beta_0 + \beta'x$ , in particular, the log-link function:  $\ln(y) = \beta_0 + \beta'x$ , or by inverting the log,  $y = e^{\beta_0 + \beta'x}$
- We will have to change our assumptions on how we think the predictions  $f(x)$  are distributed. Linear regression assumes the normal distribution for continuous-valued labels. For count-valued labels, we will need the *Poisson distribution*.

The Poisson distribution is a discrete probability distribution, so it is well suited to handle discrete count-valued labels and is expressed as the probability of how many events can occur in a fixed interval of time.

Figure 7.4 illustrates both the need for a log-link function as well as the Poisson distribution for developing regression models for count-valued data.

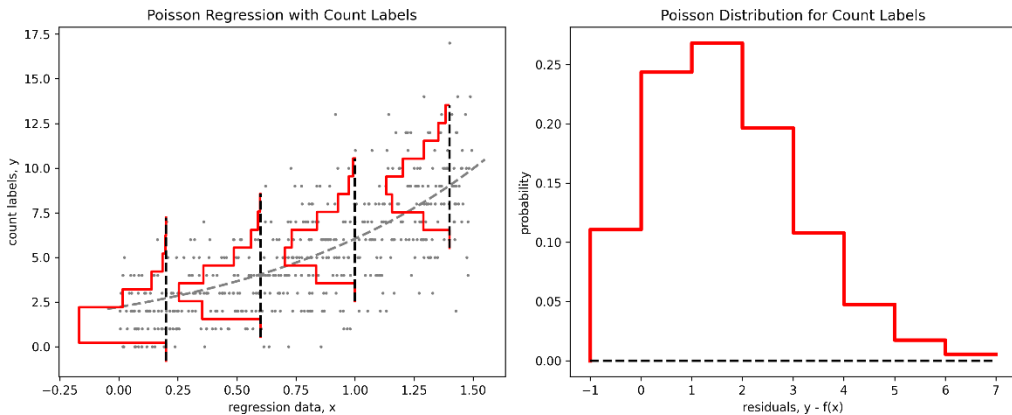
First, observe the mean (average) trend of the count labels ( $y$ ) in relation to the regression data ( $x$ ) in Figure 7.4 (left), illustrated by the dashed line. Intuitively, this is a gentle exponential trend, and shows how the features ( $x$ ) can be linked to the labels ( $y$ ).

Second, observe how the Poisson distributions overlaid on the visualization model the nature of counts (discrete) as well as their spread far better than the normal distribution.

A regression model with these underlying changes is allowed to model count-valued targets and is, appropriately called, *Poisson regression*.

To recap, Poisson regression still uses a linear model to capture the impact of the various input features from the examples. However, it introduces a log-link function and the Poisson distribution assumption to effectively model count-labeled data.

The Poisson regression approach described above is an extension of ordinary linear regression, meaning that it has no regularization. Unsurprisingly, however, we can add different regularization terms to induce robustness or sparsity, as we saw in Section 7.1.



**Figure 7.4.** Poisson regression (left) fits count-valued targets by assuming that the *spread* of the targets can be modeled by the discrete-valued Poisson distribution (right). More precisely, Poisson regression assumes that the predictions  $f(x)$  for an example  $x$  are distributed according to the Poisson distribution.

`scikit-learn`'s implementation of Poisson regression is part of the `sklearn.linear_model` subpackage. It implements Poisson regression with  $L_2$ -regularization, where the impact of regularization can be controlled through the argument `alpha`.

Thus, the parameter `alpha` is the regularization parameter, analogous to the parameter in ridge regression. Setting `alpha=0` causes the model to learn an unregularized Poisson regressor, which, as with unregularized linear regression, cannot handle feature correlations as effectively.

In the following example, we call Poisson regression with `alpha=0.01`, which trains a regression model for count labels, that is also robust to feature correlations in the data!



```

from sklearn.linear_model import PoissonRegressor
poiss_reg = PoissonRegressor(alpha=0.01)
poiss_reg.fit(Xtrn, ytrn)
ypred = poiss_reg.predict(Xtst)
mse = mean_squared_error(ytst, ypred)
mad = mean_absolute_error(ytst, ypred)
print('Poisson regression test set performance:
      MSE={0:4.3f}, MAD={1:4.3f}'.format(mse, mad))

```

This snippet, executed on the data in Figure 7.4, results in the following output:

```
Poisson regression test set performance: MSE = 3.963, MAD=1.594
```

We can train a ridge regression model on this synthetic data set with count-valued features. Remember that ridge regression uses the mean-squared error as the loss function, which is unsuited for count variables. We see that this is indeed the case:

```
Ridge regression test set performance: MSE = 4.219, MAD=1.610
```

### 7.1.3 Logistic Regression for Classification Labels

In the previous section, we saw that it is possible to extend linear regression to count-valued labels with an appropriate choice of link function and target distribution.

What other label types can we handle? Can this idea (of adding link functions and introducing other types of distributions) be extended to categorical labels?

Categorical (or class) labels are used to describe classes in binary classification problems (0 or 1) or multiclass classification problems (0, 1, 2)?

The question, then, is can we apply a regression framework to a classification problem? Amazingly, yes! For simplicity, let's focus on binary classification, where labels can take only two values: 0 or 1.

- We will have to change how we *link* the target label to the input features. For class/categorical labels, we use the *logit link function*  $g(y)=\ln(y-1)$ .

Thus, the model we will learn will be  $\ln\left(\frac{y}{1-y}\right) = \beta_0 + \beta'x$ . This may seem like a rather arbitrary choice at first, but a slightly deeper look demystifies this choice.

First, by inverting the logit function, we have the equivalent link  $y = 1/(1 + e^{-(\beta_0 + \beta'x)})$  between the labels  $y$  and the data  $x$ . That is,  $y$  is modeled with the sigmoid function, also known as the *logistic function*! Thus, using the logit-link function in a regression model turns it into *logistic regression*, a well-known *classification algorithm*!

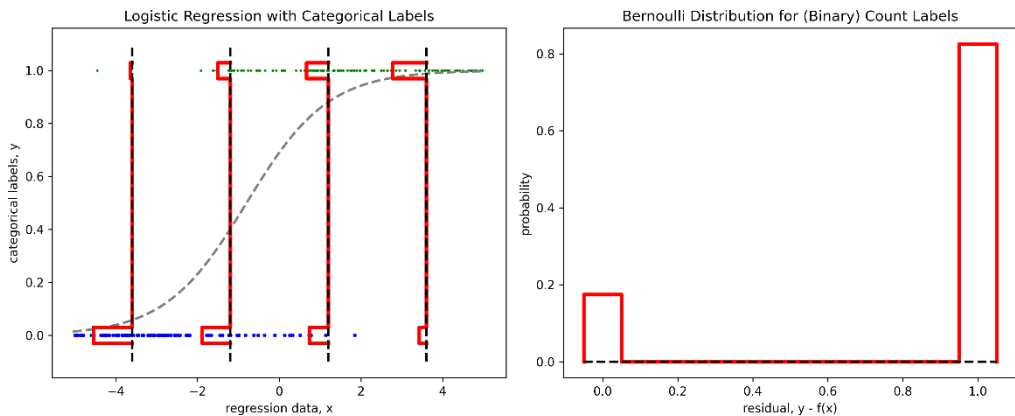
Second, let's rewrite the ratio  $y/(1-y)$  as  $y : (1-y)$ , which we interpret as the *odds of  $y$  being Class 0 to being Class 1*. These odds are exactly the same as the odds offered in gambling and betting. The logit-link function is simply the logarithm of the odds, or *log-odds*. This link function, essentially, is providing a measure of likelihood of the class being 0 or 1.

- Linear regression assumed the normal distribution for continuous-valued labels, Poisson regression assumed the Poisson distribution for count-valued labels. Logistic regression assumes the Bernoulli distribution for binary class labels.

The Bernoulli, like the Poisson distribution, is another discrete probability distribution. However, rather than describing counts of events, the Bernoulli distribution models the outcomes of yes/no questions. This is ideally suited for the binary classification case, where we ask the question, “Does this example belong to Class 0 or Class 1?”

Putting all this together, we visualize logistic regression analogously to linear regression or Poisson regression in Figure 7.5.

Figure 7.5 (left) shows a binary classification data set, where the data has only one feature and the targets belong to one of two categories. In this case, the binary labels follow the Bernoulli distribution, and the sigmoid link function (dotted line) allows us to relate the data ( $x$ ) to the labels ( $y$ ) nicely. Figure 7.5 (right) shows us a closer look at the Bernoulli distribution.



**Figure 7.5.** Logistic regression (left) fits 0/1-valued targets by assuming that the *spread* of the targets can be modeled by the discrete-valued Bernoulli distribution (right). Observe how the prediction probabilities (the heights of the bars) of Class 0 and Class 1 change with the data.

Logistic regression, of course, is one of many different classification algorithms, though one with a close connection to regression. This segue into classification problems is only intended to highlight the various types of problems the general regression framework can handle.

### 7.1.4 Generalized Linear Models

The *Generalized Linear Model* (GLM) framework includes different combinations of link functions and label probability distributions (and many other models) to create problem-specific regression variants.

Linear regression, Poisson regression, logistic regression and many other models are all different GLM variants. A (regularized) GLM regression model has four components:

- a probability distribution (formally, from the exponential family of distributions),
- a linear model  $\eta = \beta_0 + \beta'x$
- a link function  $g(\eta) = \eta$ , and

- a regularization function,  $R(\beta)$ .

Why do we care about GLMs? First, they're obviously a cool modeling approach that allows us to handle several different types of regression problems in one unified framework.

Second, and more importantly, GLMs are often used as weak learners in sequential models, especially in many gradient boosting packages such as XGBoost.

Third, and most important, GLMs allow us to think about problems in a principled manner; in practice, this means that during data set analysis, as we begin to get a good sense of the labels and their distribution, we can see which GLM variant best suits the problem at hand.

The table below shows different GLM variants, link function-distribution combinations and the types of labels they're best suited for. Some of these approaches such as Tweedie regression may be new to you, and we will get into them more in Sections 7.3 and 7.4.

Model	Link function	Distribution	Type of label
Linear Regression	Identity $g(y) = y$	Normal	Real-valued
Gamma Regression	Negative inverse $g(y) = -\frac{1}{y}$	Gamma	Positive real-valued
Poisson Regression	Log $g(y) = \ln(y)$	Poisson	Counts/occurrences; integer-valued
Logistic Regression	Logit $g(y) = \frac{y}{1-y}$	Bernoulli	0-1; binary class ids; yes/no outcomes
Multiclass Logistic Regression	Multiclass Logit $g(y) = \frac{y}{K-y}$	Binomial	0... K; multiclass ids; multichoice outcomes
Tweedie Regression	Log $g(y) = \ln(y)$	Tweedie	Labels with many zeros, right-skewed targets

**Table 7-2. Generalized Linear Models for different types of labels.**

The last method, Tweedie regression, is a particularly important GLM variant that is widely used for regression modeling in agriculture, insurance, weather, and many other areas.

### 7.1.5 Nonlinear Regression

Unlike linear regression, where the model to be learned is cast as a weighted sum of the features,  $f(x) = w_0 + w_1x_1 + \dots + w_dx_d$ , in nonlinear regression, the model to be learned can be made up of any combination of features and functions of features.

For example, a polynomial regression model of three features can be constructed from weighted combinations of all possible feature interactions:

$$f(x_1, x_2, x_3) = w_0 + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_1x_2 + w_5x_2x_3 + w_6x_3x_1 + w_7x_1x_2x_3.$$

From a modeling perspective, nonlinear regression poses two challenges:

1. Which feature combinations should we use? In the example above, with 3 features, we have  $2^3 = 8$  feature combinations, each with its own weight. In general, with  $d$  features, we would have  $2^d$  feature combinations to consider and as many weights to learn. Doing this exhaustively can be extremely computationally expensive. And the above example doesn't even include any higher order terms (e.g.,  $x_2^2 x_3$ ).
2. Which nonlinear functions should we use? All sorts of functions and combinations beyond polynomials are admissible: trigonometric, exponential, logarithmic, and many other types of functions, and even many more combinations. Searching through this space of functions exhaustively is simply computationally infeasible.

While many different nonlinear regression techniques have been proposed, studied, and used, two approaches are especially relevant in the modern context: decision trees and neural networks. We'll discuss them both briefly below, though we will focus more on decision trees as they are the building blocks of most ensemble methods.

*Tree-based methods* use decision trees to define the space of nonlinear functions to explore. During learning, decision trees are grown using the same loss functions as described previously, such as the squared loss. Each time a new decision node is added, it introduces a new feature interaction/combination into the tree.

Thus, decision trees induce feature combinations greedily and recursively during learning via the loss function as a scoring metric. As the tree grows, its nonlinearity (or, complexity) also increases. The learning objective of decision trees, then, can be written as:

$$\text{decision tree learning objective} = \lambda \cdot \underbrace{\text{tree depth}}_{\text{measures model complexity}} + \overbrace{\text{squared loss.}}^{\text{measures model fit}}$$

On the other hand, artificial neural networks use layers of neurons to successively induce increasingly complex feature combinations at each layer. The nonlinearity of a neural network increases with network depth, which directly influences the number of network weights that must be learned.

$$\text{neural network learning objective} = \lambda \cdot \underbrace{\text{number of nodes}}_{\text{measures model complexity}} + \overbrace{\text{squared loss.}}^{\text{measures model fit}}$$

The scikit-learn package provides many nonlinear regression approaches. Let's take a quick look at how we can train decision tree and neural network regressors for a simple problem.

As before, let's generate a simple, univariate data set to visualize these two regression approaches. The data are generated according to the function  $f(x) = e^{-0.5x} \sin(1.25\pi x - 1.44)$ , which is the true nonlinear relationship between the data  $x$  and the continuous labels  $y$ :

```
n = 150
X = np.random.uniform(low=-1.0, high=5.0, size=(n, 1))
g = lambda x: np.exp(-0.5*x) * np.sin(1.25 * np.pi * x - 1.414)
y = g(X)
y += np.random.normal(scale=0.08 * np.max(y), size=(n, 1))
y = y.reshape(-1, )
```

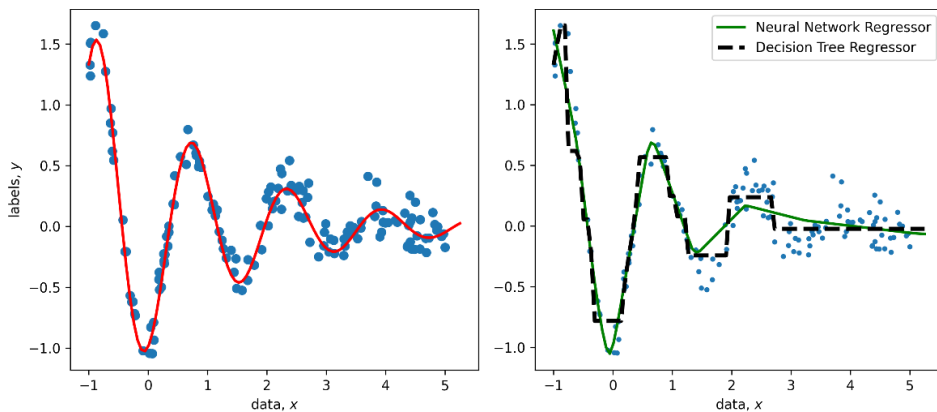


Figure 7.6. (left) The true function relating labels to data (red curve) and the generated data samples; (right) two nonlinear regression models fit to this synthetic data set: decision tree and neural network regressors.

Split into train and test sets:

```
Xtrn, Xtst, ytrn, ytst = train_test_split(X, y, test_size=0.25)
```

Now train a decision tree regressor of maximum depth 5:

```
from sklearn.tree import DecisionTreeRegressor
dt = DecisionTreeRegressor(max_depth=5)
dt.fit(Xtrn, ytrn)

ypred_dt = dt.predict(Xtst)
mse = mean_squared_error(ytst, ypred_dt)
mad = mean_absolute_error(ytst, ypred_dt)
print('Decision Tree's test set performance: MSE = {0:4.3f}, MAD={1:4.3f}'.format(mse,
    mad))
```

The learned decision tree is shown in Figure 7.6 (right). A decision tree with a univariate (single-variable) split function learns axis-parallel fits, which is reflected in the decision-tree model in the figure: the model is made up of segments that are parallel to the  $x$ - or  $y$ -axes.

In similar fashion, we can train an artificial neural network for regression, also known as a multilayer perceptron (MLP) regressor.

```

from sklearn.neural_network import MLPRegressor
ann = MLPRegressor(hidden_layer_sizes=(50, 50, 50),
                   alpha=0.001, max_iter=1000)
ann.fit(Xtrn, ytrn.reshape(-1, ))
ypred_ann = ann.predict(Xtst)
mse = mean_squared_error(ytst, ypred_ann)
mad = mean_absolute_error(ytst, ypred_ann)

print('Neural Network''s test set performance: MSE = {0:4.3f},
      MAD={1:4.3f}'.format(mse, mad))

```

This neural network is made up three hidden layers, each containing 50 neurons, which is specified during network initialization through `hidden_layer_sizes=(50, 50, 50)`.

MLPRegressor uses the piecewise-linear rectifier function ( $\text{relu}(x) = \max(x, 0)$ ) as the activation for each neuron. The regression function learned by the neural network is in Figure 7.6 (right). Since the neural network activation functions were piecewise linear, the final learned neural network model is nonlinear, though made up of several linear components, hence, piecewise.

Comparing the performance of both networks, we see that they are quite similar:

```

Decision Trees test set performance: MSE = 0.043, MAD=0.156
Neural Networks test set performance: MSE = 0.047, MAD=0.177

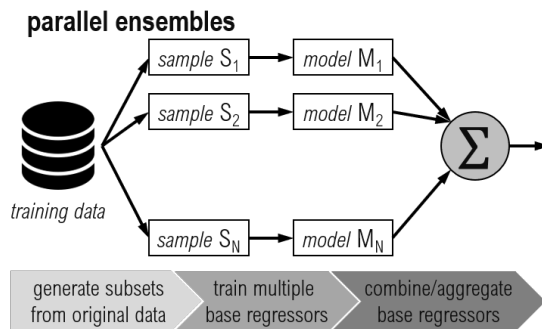
```

Finally, ensemble methods for regression are typically train nonlinear regression models (except with specific choices of base estimators), much like the ones discussed in this subsection.

## 7.2 Parallel Ensembles for Regression

In this section, we revisit parallel ensembles, both homogeneous (Chapter 2) and heterogeneous (Chapter 3) and see how they can be applied to regression problems.

Before we dive into how, let us refresh ourselves on how parallel ensembles work. The figure below illustrates a generic parallel ensemble, where base estimators are regressors.



**Figure 7.7.** Parallel ensembles train multiple base estimators *independently* of each other, and then combine their predictions into a joint ensemble prediction. Parallel regression ensembles simply use regression algorithms such as decision tree regression as base learning algorithms.

Parallel ensemble methods train each component estimator independently of the others, which means that they can be trained in parallel. Parallel ensembles typically use strong learners, or high-complexity, high-fit learners as base learners. This is in contrast to sequential ensembles, which typically use weak learners, or low-complexity, low-fit learners as base learners.

As with all ensemble methods, ensemble diversity among the component base estimators is the key. Parallel ensembles achieve this in two ways:

- With homogeneous ensembles, where the base learning algorithm is fixed, but the training data is randomly subsampled to induce ensemble diversity. In Section 7.2.1, we look at two such approaches: random forest and ExtraTrees.
- With heterogeneous ensembles, where the base learning algorithm is changed for diversity, while the training data is fixed. In Section 7.2.2, we look at two such approaches: fusing base estimator predictions with a combining function (or aggregator) and stacking base estimator predictions by learning a second-level estimator (or meta-estimator).

We focus on a problem with continuous-valued labels called AutoMPG, which is a popular regression data set, often used as a benchmark to evaluate regression methods.

The regression task is to predict the *fuel efficiency of various car models* or MPG (miles per gallon). The features consist of various engine-related attributes such as number of cylinders, displacement, horsepower, weight and acceleration.

The data set is available from the UCI Repository; it is also available with the source code. The listing below shows how to load the data and split into training and test sets.

It also includes a pre-processing step, where the data is centered and rescaled such that each feature has a mean of 0 and standard deviation of 1. This step, called normalization or standardization, ensures that all the features are in the same range of values and improves the performance of downstream learning algorithms.

### Listing 7.2. Loading and pre-processing the AutoMPG data set

```
import pandas as pd
data = pd.read_csv('./data/ch07/autompg.csv')    #A

labels = data.columns.get_loc('MPG')
features = np.setdiff1d(np.arange(0, len(data.columns), 1), labels)    #B

from sklearn.model_selection import train_test_split
trn, tst = train_test_split(data, test_size=0.2, random_state=42)    #C

from sklearn.preprocessing import StandardScaler
preprocessor = StandardScaler().fit(trn)    #D
trn, tst = preprocessor.transform(trn), preprocessor.transform(tst)

Xtrn, ytrn = trn[:, features], trn[:, labels]    #E
Xtst, ytst = tst[:, features], tst[:, labels]
```

**#A** load the data set using Pandas

**#B** get column indices for labels and features

**#C** split the data set into train and test sets

**#D** data preprocessing: normalize both training and test data and labels

#E further split train and test data into X (features) and y (labels)

We'll be using this data set as a running example for this and the next section.

### 7.2.1 Random Forest and ExtraTrees

Homogeneous parallel ensembles are some of the oldest ensemble methods and are generally variants of *bagging*. Chapter 2 introduced homogeneous ensemble methods in the context of classification. To recap, each base estimator in the bagging ensemble can be trained *independently* using the following steps:

1. Generate a bootstrap sample (or sampling with replacement, which means an example can be sampled multiple times) from the original data set
2. Fit a base estimator to the bootstrap sample; since each bootstrap sample will be different, the base estimators will be diverse.

We can follow the same for regression ensembles. The only difference is in how the individual base estimator predictions are aggregated. For classification, we use majority voting; for regression, we use the mean (essentially, the average prediction), though others such as the median can also be used.

**NOTE** Each base estimator in bagging is a fully trained strong estimator. This means that, if the bagging ensemble contains 10 base regressors, it will take 10 times as long to train. Of course, this training procedure can be parallelized over multiple CPU cores; however, the overall computational resources needed for full-blown bagging is often prohibitive.

As bagging can be rather computationally expensive to train, two important tree-based and randomized variants are used:

- *Random forest* is essentially bagging with *randomized decision trees* as base estimators. That is to say: random forests perform bootstrap sampling to generate a training subset (exactly like bagging), and then use randomized decision trees as base estimators.

Randomized decision trees are trained using a modified decision-tree learning algorithm, which introduces randomness when growing trees. Specifically, instead of considering all the features to identify the best split, a *random subset of features* is evaluated to identify the best feature to split on.

- *ExtraTrees*, or extremely randomized trees, randomized trees take the idea of randomized decision trees to the extreme by selecting not just the splitting variable from a random subset of features but also the splitting threshold. This extreme randomization is so effective, in fact, that we can construct an ensemble of extremely randomized trees directly from the original data set *without bootstrap sampling*!

Randomization has two important and beneficial consequences. Once, as we expect, is that it improves training efficiency and reduces the computational requirements. The other is that it *improves ensemble diversity*!



Random forest and ExtraTrees can be adapted to regression by modifying the underlying learning algorithm to train *regression trees* to make continuous-valued predictions rather than classification trees.

Regression trees use different splitting criteria during training compared to classification trees. In principle, any *loss function for regression* can be used as the splitting criterion. However, two commonly implemented splitting criteria are mean-squared error (MSE) and mean-absolute error (MAE). We will look at other loss functions for regression in Section 7.3.

The listing below shows how we can use `scikit-learn`'s `RandomForestRegressor` and `ExtraTreesRegressor` to train regression ensembles for the AutoMPG data set. Two versions of each method are trained: one using MSE and one using MAE as the train criterion.

### Listing 7.3. Random Forest and ExtraTrees for Regression

```
from sklearn.ensemble import RandomForestRegressor, ExtraTreesRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error

ensembles = { #A
    'Random Forest MSE': RandomForestRegressor(criterion='mse'),
    'Random Forest MAE': RandomForestRegressor(criterion='mae'),
    'ExtraTrees MSE': ExtraTreesRegressor(criterion='mse'),
    'ExtraTrees MAE': ExtraTreesRegressor(criterion='mae')}

results = pd.DataFrame() #B
ypred_trn = {}
ypred_tst = {}

for method, ensemble in ensembles.items():
    ensemble.fit(Xtrn, ytrn) #C

    ypred_trn[method] = ensemble.predict(Xtrn) #D
    ypred_tst[method] = ensemble.predict(Xtst)

    res = {'Method-Loss': method, #E
          'Train MSE': mean_squared_error(ytrn, ypred_trn[method]),
          'Train MAE': mean_absolute_error(ytrn, ypred_trn[method]),
          'Test MSE': mean_squared_error(ytst, ypred_tst[method]),
          'Test MAE': mean_absolute_error(ytst, ypred_tst[method])}

    results = results.append(res, ignore_index=True) #F
```

**#A** initialize Random Forest and ExtraTrees with different loss functions  
**#B** create data structures to store model predictions & evaluation results  
**#C** train the ensemble  
**#D** get ensemble predictions on both train & test sets  
**#E** evaluate train & test set performance with MAE and MAE  
**#F** save results

All models are also evaluated using MSE and MAE as the evaluation criterion. These evaluation metrics are added to the `results` variable:

	Method-Loss	Test MAE	Test MSE	Train MAE	Train MSE
0	Random Forest MSE	0.2085	0.0953	0.0914	0.0163
1	Random Forest MAE	0.2272	0.0982	0.0979	0.0191
2	ExtraTrees MSE	0.1960	0.0765	0.0000	0.0000
3	ExtraTrees MAE	0.2014	0.0758	0.0000	0.0000

In the example above, we used the default parameter settings for both `RandomForestRegressor` and `ExtraTreesRegressor`. For instance, each trained ensemble is of size 100 as `n_estimators=100`, by default.

As with any other machine-learning algorithm, we have to identify the best model hyperparameters (such as `n_estimators`), through a grid search or randomized search. There are several examples of this in the case study in Section 7.4.

## 7.2.2 Combining Regression Models

Another classical ensembling approach, especially when we have multiple models is to simply combine their predictions. This is essentially one of the simplest heterogeneous parallel ensembling approaches

Why combine regression models? It is quite common, during the data exploration phase, to experiment with different ML algorithms. This means that we often have several different models available to us for ensembling.

For example, in Section 7.2.1, we trained four different regression models. Since we have the predictions of four different models, can we not combine them into one ensemble prediction?

Happily, yes! But what combination functions should we use?

- For continuous-valued targets, use combining functions/aggregators such as weighted mean, median, min or max. In particular, the median is especially effective when combining heterogeneous predictions where the models are in greater disagreement.

For example, if we have 5 models in the ensemble predicting [0.29, 0.3, 0.32, 0.35, 0.85]. Most of the models agree, though there is one outlier, 0.85. The mean of these predictions is 0.42, while the median is 0.32. Thus, the median tends to discard the influence of the outliers (and behaves similarly to majority voting), while the mean tends to include them. This is because the median is simply (and literally) the middle value, while the mean is the averaged value.

- For count-valued targets, use combining functions/aggregators such as mode and median. We can think of the mode, in particular, as the generalization of majority voting to counts. The mode is simply the most common answer.

For example, if we have 5 models in the ensemble predicting [12, 15, 15, 15, 16], the mode is 15. In case of conflicts, where there are equal counts, we can use random selection to break ties.

The listing below illustrates the use of four simple aggregators for continuous-valued data. In this listing, we use the four regressors trained in Listing 7.3 as the (heterogeneous) base estimators whose values we'll combine: `RandomForestRegressor` and `ExtraTreesRegressor`, each trained with MSE and MAE as the loss function/split criterion.

**Listing 7.4. Aggregators for Continuous-Valued Labels**

```

import numpy as np
agg_methods = ['Mean', 'Median', 'Max', 'Min']
aggregators = [np.mean, np.median, np.max, np.min]    #A

results = pd.DataFrame()    #B
ypred_trn_values = np.array(list(ypred_trn.values()))    #C
ypred_tst_values = np.array(list(ypred_tst.values()))

for method, aggregate in zip(agg_methods, aggregators):
    yagg_trn = aggregate(ypred_trn_values, axis=0)    #D
    yagg_tst = aggregate(ypred_tst_values, axis=0)

    res = {'Aggregator': method,    #E
          'Train MSE': mean_squared_error(ytrn, yagg_trn),
          'Train MAE': mean_absolute_error(ytrn, yagg_trn),
          'Test MSE': mean_squared_error(ytst, yagg_tst),
          'Test MAE': mean_absolute_error(ytst, yagg_tst)}
    results = results.append(res, ignore_index=True)

```

#A different combining functions for continuous-valued predictions

#B data structure model predictions & evaluation results

#C collect predictions of the four ensembles trained in Listing 7.3

#D aggregate predictions of the four ensembles trained in Listing 7.3

#E collect and save results

Again, all models are also evaluated using MSE and MAE as the evaluation criterion. These evaluation metrics are added to the `results` variable:

	Aggregator	Test MAE	Test MSE	Train MAE	Train MSE
0	Mean	0.2055	0.0826	0.0469	0.0043
1	Median	0.2026	0.0804	0.0396	0.0034
2	Max	0.2280	0.1063	0.0573	0.0101
3	Min	0.1985	0.0763	0.0527	0.0119

### 7.2.3 Stacking Regression Models

Another way to combine the predictions of different (heterogeneous) regressors is through *stacking* or meta-learning.

Instead of making up a function ourselves (such as the mean or median), we train a second-level model to *learn how to combine the predictions of the base estimators*. This second-level regressor is known as the *meta-learner* or the meta-estimator.

The meta-estimator is often a nonlinear model that can effectively combine the predictions of the base estimators in a nonlinear manner. The price we pay for this added complexity is that stacking can often overfit, especially in the presence of noisy data.

To guard against overfitting, stacking is often combined with *k-fold cross validation* such that each base estimator is not trained on the exact same data set. This often leads to more diversity and robustness, while decreasing the chances of overfitting.

In Listing 3.1, we implemented a stacking model for classification from scratch. An alternate implementation uses scikit-learn's `StackingClassifier` and `StackingRegressor`. This is illustrated for regression problems in Listing 7.5 below.

Here, we train four nonlinear regressors: kernel ridge regression (a nonlinear extension

of ridge regression), support vector regression, k-nearest neighbor regression and ExtraTrees.

We use an *artificial neural network* as a meta-learner, which allows us to combine predictions of various heterogeneous regression models in a learnable and highly nonlinear fashion.

#### Listing 7.5. Stacking Regression Models

```
from sklearn.ensemble import StackingRegressor
from sklearn.neural_network import MLPRegressor
from sklearn.kernel_ridge import KernelRidge
from sklearn.svm import SVR
from sklearn.tree import DecisionTreeRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.gaussian_process import GaussianProcessRegressor

estimators = [('Kernel Ridge', KernelRidge(kernel='rbf', gamma=0.1)), #A
             ('Support Vector Machine', SVR(kernel='rbf', gamma=0.1)),
             ('K-Nearest Neighbors', KNeighborsRegressor(n_neighbors=3)),
             ('ExtraTrees', ExtraTreesRegressor(criterion='mae'))]

meta_learner = MLPRegressor(hidden_layer_sizes=(50, 50, 50), #B
                           max_iter=1000)

stack = StackingRegressor(estimators, final_estimator=meta_learner, cv=3)
stack.fit(Xtrn, ytrn) #C

ypred_trn = stack.predict(Xtrn) #D
ypred_tst = stack.predict(Xtst)
print('Train MSE = {0:5.4f}, Train MAE = {1:5.4f}\n' \
      'Test MSE = {2:5.4f}, Test MAE = {3:5.4f}'.format(
      mean_squared_error(ytrn, ypred_trn),
      mean_absolute_error(ytrn, ypred_trn),
      mean_squared_error(ytst, ypred_tst),
      mean_absolute_error(ytst, ypred_tst)))
```

#A initialize first level (base) regressors  
 #B initialize second level (meta) regressor  
 #C train a stacking regressor with 3-fold cross validation  
 #D compute train & test errors

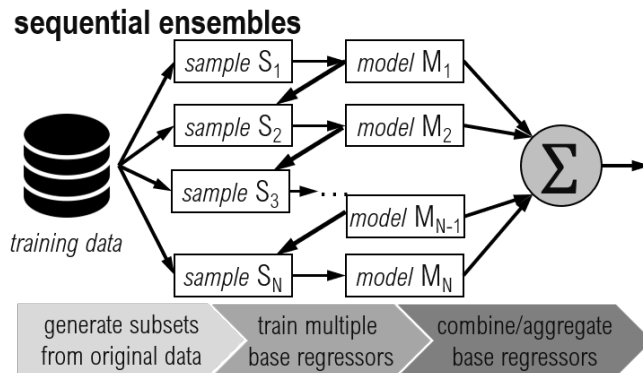
The stacking regression produces the following output:

```
Train MSE = 0.0464, Train MAE = 0.1547
Test MSE = 0.0969, Test MAE = 0.2297
```

It should be noted here that default parameters were used with the individual base regressors. The performance of this stacking ensemble can further be improved with effective hyperparameter tuning of the base estimator models, which improves the performance of each ensemble component, and hence the ensemble overall.

### 7.3 Sequential Ensembles for Regression

In this section, we revisit sequential ensembles, specifically gradient boosting (with LightGBM, Chapter 5) and Newton boosting (with XGBoost, Chapter 6) and see how they can be adapted to regression problems.



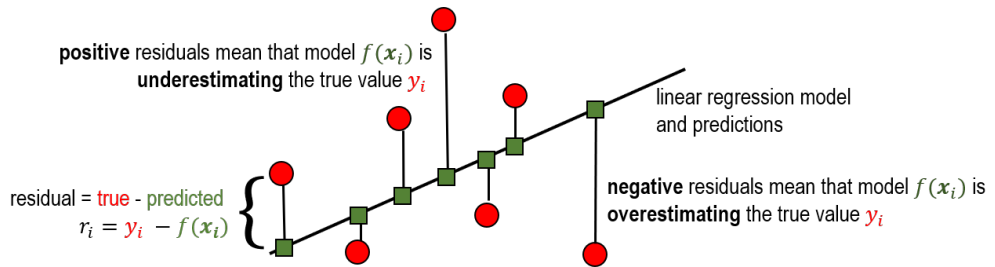
**Figure 7.8.** Unlike parallel ensembles which train base estimators *independently* of each other, sequential ensembles such as boosting train successive base estimators to identify and minimize the errors made by the previous base estimator.

Both these approaches are very general in that they can be trained on a wide variety of loss functions. This means that they can easily be adapted to different types of problem settings, allowing for problem-specific modeling of continuous-valued and count-valued labels.

Before we dive into how, let us refresh ourselves on how parallel ensembles work. The figure below illustrates a generic sequential ensemble, where base estimators are regressors.

Unlike parallel ensembles, sequential ensembles grow the ensemble one estimator at a time, where successive estimators aim to improve upon the predictions of the previous ones.

Each successive base estimator uses the *residual* as a means of identifying which training examples need attention in the current iteration. In regression problems, the residual tells the base estimator how much the model is underestimating or overestimating the prediction. This is illustrated in the figure below.



**Figure 7.9.** A linear regression model and its predictions (green squares) fit to a data set (red circles). The residuals are a measure of the *error* between the true label ( $y_i$ ) and predicted label  $f(x_i)$ . The size of the residual of each training example indicates the extent of the error in fitting, while the sign of the residual indicates whether the model is under or overestimating.

Residuals convey two important pieces of information to the base learners. For each training example, the magnitude of the residual can be interpreted in a straightforward manner: bigger residuals mean more errors.

The sign of the residual also conveys important information. A positive residual suggests that the current model's prediction is *underestimating* the true value, that is, the model has to increase its prediction. A negative residual suggests that the current model's prediction is *overestimating* the true value, that is, the model has to decrease its prediction.

The *loss function and, more importantly, its derivatives* allow us to measure the *residual* between the current model's prediction and the true label. By changing the loss function, we are essentially changing how we prioritize different examples.

Both gradient boosting and Newton boosting use shallow regression trees as *weak base learners*. Weak learners (contrast with bagging and its variants, which use strong learners) are essentially low-complexity, low-fit models.

By training a sequence of weak learners to fix the mistakes of the previously learned weak learners, both methods *boost* the performance of the ensemble in stages:

- *Gradient boosting* uses the negative gradient of the loss function as the residual to identify training examples to focus on.
- *Newton boosting* uses Hessian-weighted gradients of the loss function as the residual to identify training examples to focus on. The Hessians (second derivatives) of the loss functions incorporate local "curvature" information, to increase the weight on training examples with higher loss values.

Loss functions, then, are a key ingredient in developing effective sequential ensembles.

### 7.3.1 Loss and Likelihood Functions for Regression

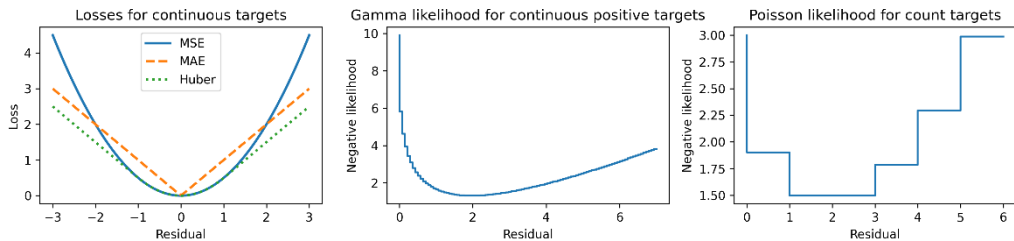
In this section, we'll take a look at some common (and uncommon) loss functions for different types of labels: continuous valued, continuous valued but positive, and count valued.

Each of these loss functions penalizes errors differently, and will result in learning models with different properties, much like how different regularization functions produced models with different properties (in Section 7.1).

Many loss functions are ultimately derived from how we assume the *residuals* are distributed. We have already seen this in Section 7.1, where we assume that the residuals of continuous-valued targets can be modeled using the Gaussian distribution, count-valued targets can be modeled using the Poisson distribution and so on.

Here, we formalize that notion. Note that some loss functions do not have a closed-form expression. In such cases, it is useful to visualize the *negative log of the underlying distribution*. This term, called the *negative log-likelihood*, is sometimes optimized instead of the loss function, and ultimately has the same effect in the final model.

We consider three types of labels and their corresponding loss functions. These are visualized in the figure below.



**Figure 7.10.** Loss and log-likelihood functions for three different types of targets: continuous-valued (left), positive continuous valued (center) and count-valued (right).

### CONTINUOUS-VALUED LABELS

There are several well-known loss functions for *continuous-valued targets*. Two of the most common are the *mean squared error* (MSE):  $\frac{1}{2} (y-f(x))^2$  and the *mean absolute error* (MAE):  $|y-f(x)|$ . The MSE directly corresponds to assuming a Gaussian distribution over the residuals (the MAE corresponds to assuming the Laplacian distribution over the residuals).

The MSE penalizes errors far more heavily than the MAE, as is evident from the loss values at the extremes in the figure above. This makes the MSE highly sensitive to outliers.

The MSE is also a doubly differentiable loss function, which means that we can compute both the first and second derivatives. Thus, we can use it for both gradient boosting (which uses residuals) and Newton boosting (which uses Hessian-boosted residuals). The MAE is not doubly differentiable, meaning it cannot be used in Newton boosting.

The Huber loss is a hybrid of the MSE and the MAE and switches its behavior between the two at some user-specified threshold  $\alpha$ :

$$L(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2, & \text{for } |y - f(x)| \leq \alpha, \\ \alpha|y - f(x)| - \frac{\alpha^2}{2}, & \text{otherwise.} \end{cases}$$

For residuals smaller than  $\alpha$ , the Huber loss behaves like the MSE, and beyond the threshold, like the (scaled) MAE (see Figure 7.10). This makes the Huber loss ideal in situations where we desire to limit the *influence of outliers*.

Note that the Huber loss cannot be directly used with Newton boosting as it contains the MAE as one of its components. For this reason, Newton boosting implementations use a smooth approximation called the *pseudo-Huber loss*:

$$L(y, f(x)) = \alpha^2 \left( \sqrt{1 + \frac{1}{\alpha^2} (y - f(x))^2} - 1 \right)$$

The pseudo-Huber loss, for all intents and purposes, behaves like the Huber loss, though is an approximate version that is outputs  $\frac{1}{2} (y-f(x))^2$  for residuals  $(y-f(x))$  close to zero.

#### CONTINUOUS-VALUED POSITIVE LABELS

In some domains such as insurance claims analytics, the target values that we wish to predict only take positive values. For example, the claim amount is continuous-valued, but can only be positive.

In such situations, where the Gaussian distribution is not appropriate, we can use the *gamma distribution*. The gamma distribution is a highly flexible distribution that can fit many target distribution shapes. This makes it ideally suited for modeling problems where the target distributions have long tails: that is outliers that cannot be ignored.

The gamma distribution does not correspond to a closed-form loss function. In Figure 7.10 (center), we plot the negative log-likelihood instead, which functions as a surrogate loss function. First, observe that the loss function is only defined for positive real values (x-axis).

Next, observe how the log-likelihood function only gently penalizes errors to the further right. This allows the underlying models to fit to right-skewed data.

#### COUNT-VALUED LABELS

Beyond continuous-valued labels, some regression problems require us to fit count-valued targets. We have already seen examples of this in Section 7.1, where we learned that counts, which are discrete-valued, can be modeled using the *Poisson distribution*.

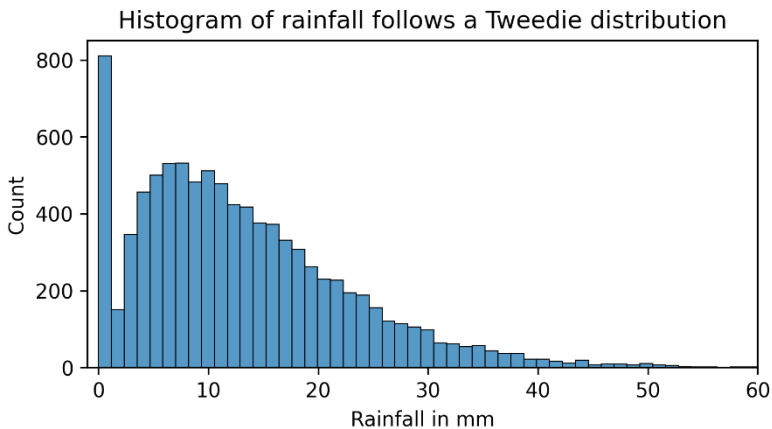
Like the gamma distribution, the Poisson distribution also doesn't correspond to a closed-form loss function. Figure 7.10 (right) illustrates the negative log-likelihood of the Poisson distribution, which can be used to build regression models (called Poisson regression).



### HYBRID LABELS

In some problems, the underlying labels cannot be modeled by a single distribution. For example, in weather analytics, if we want to model rainfall we can expect that: (a) on most days we will have no rain at all, (b) on some days, we will have varying degrees of rainfall, and (c) on some rare occasions, we will have very heavy rainfall.

The figure below illustrates the distribution of rainfall data, where we have a big “point mass” or spike at 0 (corresponding to most days that receive no rainfall). In addition, this distribution is also right skewed as there are a small number of days with very high rainfall.



**Figure 7.11.** Modeling some types of labels require combinations of distributions, called compound distributions, to effectively model them. One such compound distribution is the Tweedie distribution.

To model this problem, we need a loss function corresponding to a hybrid distribution, specifically a Poisson-gamma distribution: the Poisson distribution to model the big point mass at 0, and the gamma distribution to model the right-skewed, positive continuous data.

For such labels, we can use a powerful family of probability distributions called the Tweedie distributions, which are parameterized by a parameter  $p$ . Different values of  $p$ , give rise to different distributions:

- $p = 0$ , Gaussian (normal) distribution
- $p = 1$ , Poisson distribution
- $1 < p < 2$ , Poisson-gamma distributions for different  $p$
- $p = 2$ , gamma distribution
- other distributions are possible with other values of  $p$ .

For our purposes, we are mostly interested in using  $1 < p < 2$ , to create a hybrid Poisson-gamma loss functions.

Both LightGBM and XGBoost come with support for the Tweedie distribution, and has led to their widespread adoption in domains such as weather analytics, insurance analytics and health informatics. We will see how to use this in our case study in Section 7.4.

### 7.3.2 Gradient Boosting with LightGBM and XGBoost

Now, armed with the knowledge of various loss functions, let's see how we can apply gradient boosting regressors to the AutoMPG data set.

#### GRADIENT BOOSTING WITH LIGHTGBM

First, let's apply standard gradient boosting: LightGBM's `LGBMRegressor` with the Huber loss function. There are several LightGBM hyperparameters that we further have to select. These parameters control various components of LightGBM:

- *loss function parameters*: `alpha` is the Huber loss parameter, the threshold where it switches from behaving like the MSE to behaving like the MAE loss.
- *learning control parameters*: such as `learning_rate` to control the rate at which the model learns so that it doesn't rapidly fit, and then overfit the training data; such as `subsample` to randomly sample a smaller fraction of the data during training to induce additional ensemble diversity and improve training efficiency.
- *regularization parameters*: `lambda_l1` and `lambda_l2` are the weights on the  $L_1$  and  $L_2$  regularization functions respectively; these correspond to  $a$  and  $b$  in the elastic net objective (see Table 7.1)
- *tree learning parameters*: `max_depth`, which limits the maximum depth of each weak tree in the ensemble

There are other hyperparameters in each category, that further allow for finer-grained control over training. We select hyperparameters using a combination of randomized search (since an exhaustive grid search would be too slow) and cross validation. The listing below shows an example of this with LightGBM.

In addition to hyperparameter selection, the listing below also implements *early stopping*, where training is terminated if no performance improvement is observed on an evaluation set.

**Listing 7.6. LightGBM with Huber loss**

```

from lightgbm import LGBMRegressor
from sklearn.model_selection import RandomizedSearchCV

parameters = {'alpha': [0.3, 0.9, 1.8],      #A
              'max_depth': np.arange(2, 5, step=1),
              'learning_rate': 2**np.arange(-8., 2., step=2),
              'subsample': [0.6, 0.7, 0.8],
              'lambda_l1': [0.01, 0.1, 1],
              'lambda_l2': [0.01, 0.1, 1e-1, 1]}

lgb = LGBMRegressor(objective='huber', n_estimators=100)  #B
param_tuner = RandomizedSearchCV(lgb, parameters, n_iter=20, cv=5,      #C
                                refit=True, verbose=1)

param_tuner.fit(Xtrn, ytrn,      #D
                eval_set=[(Xtst, ytst)], eval_metric='mse', verbose=False)

ypred_trn = param_tuner.best_estimator_.predict(Xtrn)      #E
ypred_tst = param_tuner.best_estimator_.predict(Xtst)
print('Train MSE = {0:5.4f}, Train MAE = {1:5.4f}\n' \
      'Test MSE = {2:5.4f}, Test MAE = {3:5.4f}'.format(
    mean_squared_error(ytrn, ypred_trn),
    mean_absolute_error(ytrn, ypred_trn),
    mean_squared_error(ytst, ypred_tst),
    mean_absolute_error(ytst, ypred_tst)))

```

**#A** ranges of hyperparameters that we would like to search over

**#B** initialize a LightGBM regressor

**#C** since GridSearchCV will be slow, search over 20 random parameter combinations with 5-fold CV

**#D** fit the regressor with early stopping

**#E** compute train and test errors

This produces the following output

```

Fitting 5 folds for each of 20 candidates, totalling 100 fits
Train MSE = 0.0476, Train MAE = 0.1497
Test MSE = 0.0951, Test MAE = 0.2250

```

**NEWTON BOOSTING WITH XGBOOST**

We can repeat this training and evaluation with XGBoost's `XGBRegressor`. Since Newton boosting requires second derivatives, which cannot be computed for the Huber loss, XGBoost does not provide this loss directly.

Instead, XGBoost provides a pseudo-Huber loss, which is a differentiable approximation of the Huber loss, which we will use here. Again, as with LightGBM, we have to set several different hyperparameters. Many of XGBoost's parameters correspond exactly to LightGBM's parameters, though they have different names:

- *learning control parameters*: such as `learning_rate` to control the rate at which the model learns so that it doesn't rapidly fit, and then overfit the training data; such as `colsample_bytree` to randomly sample a smaller fraction of the data during training to induce additional ensemble diversity and improve training efficiency.
- *regularization parameters*: `reg_alpha` and `reg_lambda` are the weights on the  $L_1$  and

$L_2$  regularizations functions respectively; these correspond to  $a$  and  $b$  in the elastic net objective (see Table 7.1)

- *tree learning parameters*: `max_depth`, which limits the maximum depth of each weak tree in the ensemble

The listing below shows how we can train an `XGBRegressor`, including a randomized hyperparameter search.

#### Listing 7.7. Using XGBoost with pseudo-Huber Loss

```
from xgboost import XGBRegressor
parameters = {'max_depth': np.arange(2, 5, step=1),      #A
              'learning_rate': 2**np.arange(-8., 2., step=2),
              'colsample_bytree': [0.6, 0.7, 0.8],
              'reg_alpha': [0.01, 0.1, 1],
              'reg_lambda': [0.01, 0.1, 1e-1, 1]}

xgb = XGBRegressor(objective='reg:pseudohubererror')    #B
param_tuner = RandomizedSearchCV(xgb, parameters, n_iter=20, #C
                                 cv=5, refit=True, verbose=1)

param_tuner.fit(Xtrn, ytrn, eval_set=[(Xtst, ytst)],    #D
                eval_metric='rmse', verbose=False)

ypred_trn = param_tuner.best_estimator_.predict(Xtrn)  #E
ypred_tst = param_tuner.best_estimator_.predict(Xtst)
print('Train MSE = {0:5.4f}, Train MAE = {1:5.4f}\n' \
      'Test MSE = {2:5.4f}, Test MAE = {3:5.4f}'.format(
    mean_squared_error(ytrn, ypred_trn),
    mean_absolute_error(ytrn, ypred_trn),
    mean_squared_error(ytst, ypred_tst),
    mean_absolute_error(ytst, ypred_tst)))
```

**#A** ranges of hyperparameters that we would like to search over

**#B** initialize a XGBoost regressor

**#C** since `GridSearchCV` will be slow, search over 20 random parameter combinations with 5-fold CV

**#D** fit the regressor with early stopping

**#E** compute train and test errors

This produces the following output:

```
Fitting 5 folds for each of 20 candidates, totalling 100 fits
Train MSE = 0.0499, Train MAE = 0.1597
Test MSE = 0.0956, Test MAE = 0.2242
```

The LightGBM (gradient boosting) model trained with the Huber loss achieves test MSE of 0.0951, while the XGBoost (Newton boosting) model trained with the pseudo-Huber achieves a similar test MSE of 0.0956.

This illustrates that the pseudo-Huber loss is a reasonable substitute for the Huber loss when the situation calls for it. We will shortly see how we can use LightGBM and XGBoost with other loss functions discussed in this Section on the task of bike demand prediction, the Chapter case study.

## 7.4 Case Study: Demand Forecasting

Demand forecasting is an important problem that arises in many business contexts, where the goal is to predict the demand for a certain product or commodity. Accurately predicting demand is critical for downstream supply chain management and optimization: to ensure that there is enough supply to meet needs and not too much that there is waste.

Demand forecasting is often cast as a regression problem of using historical data and trends to build a model to predict future demand. The target labels can be continuous or count-valued.

For example, in energy demand forecasting, the label to predict (energy demand in gigawatt hours) is continuous-valued. Alternately, in product demand forecasting, the label to predict (number of items to be shipped) is count-valued.

In this section, we study the problem of **Bike Rental Forecasting**. As we see below, the nature of the problem (and especially, the targets/labels) is quite similar to those arising in the areas of weather prediction and analytics, insurance and risk analytics, health informatics, energy demand forecasting, business intelligence and many others.

We analyze the data set and then build progressively more complex models, beginning with single linear models, then moving on to ensemble nonlinear models. At each stage, we will perform hyperparameter tuning to select the best hyperparameter combinations.

### 7.4.1 The UCI Bike Rental Data Set

The Bike Rental data set<sup>4</sup> was the first of several similar publicly available data sets that tracks the usage of bicycle sharing services in major metropolitan areas. These data sets are made publicly available through the UCI Machine Learning Repository.

This data set, first made available in 2013, tracks hourly and daily bicycle rentals of Capital Bike Sharing in Washington DC. In addition, the data set also contains several features describing the weather as well as the time of day and day of the year.

The overall goal of the problem is to predict the bike rental demand depending on the time of day, the season, and the weather. The demand is measured in *total number of users*, a count! The total number of users is further composed of casual and registered users.

The number of registered users appears to be fairly consistent across the year, since these are users who presumably use bike sharing as a regular transportation option rather than a recreational activity. This is akin to commuters who have a monthly/annual bus pass for their daily commutes as opposed to, say tourists, who only buy bus tickets as needed.

Keeping this in mind, we construct a derived data set for our case study that can be used to build a model to *forecast the rental bike demand of casual users*.

**Problem:** *Predict the number of casual bike rental users depending on the time of day, season and weather.*

The (modified) data set for this case study is available with the code and can be loaded thus:

---

<sup>4</sup> Fanaee-T, H., Gama, J. Event labeling combining ensemble detectors and background knowledge. *Progress in Artificial Intelligence* **2**, 113–127 (2014).

```
import pandas as pd
data = pd.read_csv('./data/ch07/bikesharing.csv')
```

We can look at the statistics of the data set with

```
data.describe()
```

	season	mnth	hr	holiday	weekday	workingday	weathersit	temp	atemp	hum	windspeed	casual
count	17,379.000	17,379.000	17,379.000	17,379.000	17,379.000	17,379.000	17,379.000	17,379.000	17,379.000	17,379.000	17,379.000	17,379.000
mean	2.502	6.538	11.547	0.029	3.004	0.683	1.425	0.497	0.476	0.627	0.190	35.676
std	1.107	3.439	6.914	0.167	2.006	0.465	0.639	0.193	0.172	0.193	0.122	49.305
min	1.000	1.000	0.000	0.000	0.000	0.000	1.000	0.020	0.000	0.000	0.000	0.000
25%	2.000	4.000	6.000	0.000	1.000	0.000	1.000	0.340	0.333	0.480	0.104	4.000
50%	3.000	7.000	12.000	0.000	3.000	1.000	1.000	0.500	0.485	0.630	0.194	17.000
75%	3.000	10.000	18.000	0.000	5.000	1.000	2.000	0.660	0.621	0.780	0.254	48.000
max	4.000	12.000	23.000	1.000	6.000	1.000	4.000	1.000	1.000	1.000	0.851	367.000

Figure 7.12. Statistics of the bike rental data set. The column 'casual' is the prediction target (label).

The data set contains several continuous weather features: `temp` (normalized temperature), `atemp` (normalized 'feels like' temperature), `hum` (humidity) and `windspeed`. The categorical feature `weathersit` describes the type of weather seen at that time with four categories

- 1: Clear, Few clouds, Partly cloudy, Partly cloudy
- 2: Mist + Cloudy, Mist + Broken clouds, Mist + Few clouds, Mist
- 3: Light Snow, Light Rain + Thunderstorm + Scattered clouds, Rain + Scattered clouds
- 4: Heavy Rain + Ice Pallets + Thunderstorm + Mist, Snow + Fog

The data set also contains discrete features: `season` (1:winter, 2:spring, 3:summer, 4:fall), `mnth` (1 to 12 for Jan through Dec) and `hr` (hour from 0 to 23) to describe the time. In addition, the binary features `holiday`, `weekday` and `workingday` encode whether the day in question is a holiday or a weekday or working day.

#### PREPROCESSING THE FEATURES

Let's preprocess this data set by normalizing the features, that is, we ensure that each feature is zero mean, unit standard deviation.

Normalization is not always the best approach to deal with discrete features. For now though, let's use this simple preprocessing and keep our focus on ensembles for regression. In Chapter 8, we delve more into preprocessing strategies for these types of features.

The listing below shows our preprocessing steps: it splits the data into training (80% of the data) and test sets (remaining 20% of the data) and applies normalization to the features.

As always, we'll hold out the test set from the training process so that we can evaluate the performance of each of our trained models on the test set.

### Listing 7.8. Preprocessing the Bike Rental Data Set

```

labels = data.columns.get_loc('casual') #A
features = np.setdiff1d(np.arange(0, len(data.columns), 1), labels) #B

from sklearn.model_selection import train_test_split
trn, tst = train_test_split(data, test_size=0.2, random_state=42)
Xtrn, ytrn = trn.values[:, features], trn.values[:, labels] #C
Xtst, ytst = tst.values[:, features], tst.values[:, labels]

from sklearn.preprocessing import StandardScaler
preprocessor = StandardScaler().fit(Xtrn) #D
Xtrn, Xtst = preprocessor.transform(Xtrn), preprocessor.transform(Xtst)

```

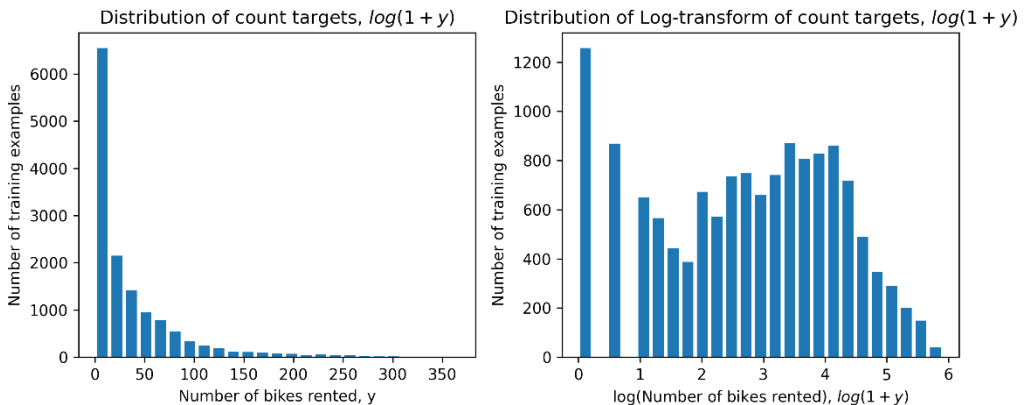
**#A** get column index for the label  
**#B** get column indices for the features  
**#C** split into train & test sets  
**#D** pre-process features by normalizing

#### COUNT-VALUED TARGETS

The target label we want to predict is `casual`, the number of casual users, which is count-valued, ranging from 0 to 367.

We plot the histogram of these targets below (left). This data set has a large point mass at 0, indicating that on many days, there are no casual users. Further, we can see that this distribution has a *long tail*, which makes it *right skewed*.

We can further analyze these labels by applying a log transformation, that is, we transform each count label  $y$  to  $\log(1+y)$ , where we add 1 to avoid taking the logarithm of zero count data. This is shown in the figure below (right).



**Figure 7.13.** Histogram of count-valued targets, the number of casual users (left); histogram of the count targets after log transformation (right).

This gives us two great insights regarding how we might want to model the problem:

- the distribution of the log-transformed count target looks very similar to the histogram of rainfall in Figure 7.11, which suggests that a *Tweedie distribution might be appropriate for modeling this problem*. Recall that a Tweedie distribution with parameter  $1 < p < 2$  can model a compound Poisson-gamma distribution: the Poisson distribution to model the big point mass at 0, and the gamma distribution to model the right-skewed, positive continuous data.
- the log transformation itself suggests a connection between the target and the features. If we were to model this regression task as a Generalized Linear Model (GLM), we would have to use the log-link function. We would like to extend this notion to ensemble methods (which are usually nonlinear).

As we will see shortly, LightGBM and XGBoost provide support for modeling both the log link (and other link functions) and distributions such as Poisson, gamma, and Tweedie. This allows them to emulate the intuition of GLMs to capture the nuances of the data set, while going beyond the limitations of GLMs of being restricted to learning only linear models.

#### 7.4.2 Generalized Linear Models and Stacking

Let's first train *individual* general linear regression models that capture the intuitions gleaned above. In addition, we will also stack these individual models to combine their predictions. We will train three individual regressors:

- Tweedie regression with the log link function, which uses the Tweedie distribution to model the positive, right-skewed targets. We use `scikit-learn's TweedieRegressor`, which requires that we choose two parameters: `alpha`, parameter for the L2-regularization term and `power`, which should be between 1 and 2.
- Poisson regression with the log link function, which uses the Poisson distribution to model count variables. We use `scikit-learn's PoissonRegressor`, which requires that we choose only one parameter: `alpha`, parameter for the L2-regularization term. It should be noted that setting `power=1` in `TweedieRegressor` is equivalent to using `PoissonRegressor`.
- Ridge regression, which uses the normal distribution to model continuous variables. This, in general, is not well-suited for this data and is included as a baseline, as it is one of the most common methods we will encounter in the wild.

The listing below demonstrates how we can train these regressors, with the hyperparameter search through exhaustive grid search combined with cross validation.



**Listing 7.9. Training GLMs for Bike Rental Prediction**

```

from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error,
                             mean_absolute_error, r2_score
from sklearn.linear_model import Ridge, PoissonRegressor, TweedieRegressor

parameters = {'GLM: Linear': {'alpha': 10 ** np.arange(-4., 1.)},      #A
              'GLM: Poisson': {'alpha': 10 ** np.arange(-4., 1.)},
              'GLM: Tweedie': {'alpha': 10 ** np.arange(-4., 1.),
                               'power': np.linspace(1.1, 1.9, num=5)}} #B

glms = {'GLM: Linear': Ridge(),      #C
        'GLM: Poisson': PoissonRegressor(max_iter=1000),
        'GLM: Tweedie': TweedieRegressor(max_iter=1000)}

best_glms = {}      #D
results = pd.DataFrame()

for glm_type, glm in glms.items():
    param_tuner = GridSearchCV(glm, parameters[glm_type],      #E
                               cv=5, refit=True, verbose=2)
    param_tuner.fit(Xtrn, ytrn)

    best_glms[glm_type] = param_tuner.best_estimator_      #F
    ypred_trn = best_glms[glm_type].predict(Xtrn)
    ypred_tst = best_glms[glm_type].predict(Xtst)

    res = {'Method': glm_type,      #G
           'Train MSE': mean_squared_error(ytrn, ypred_trn),
           'Train MAE': mean_absolute_error(ytrn, ypred_trn),
           'Train R2': r2_score(ytrn, ypred_trn),
           'Test MSE': mean_squared_error(ytst, ypred_tst),
           'Test MAE': mean_absolute_error(ytst, ypred_tst),
           'Test R2': r2_score(ytst, ypred_tst)}
    results = results.append(res, ignore_index=True)

```

**#A** ranges of hyperparameters for ridge, Poisson & Tweedie regressors

**#B** Tweedie regression has an additional parameter: power

**#C** initialize GLMs

**#D** to save individual GLMs after cross validation

**#E** perform grid search for each GLM with 5-fold CV

**#F** get the final refit GLM model and compute train & test predictions

**#G** compute and save 3 metrics for each GLM: MAE, MSE, R2 score

If we `print(results)`, we will see what the three models have learned. We evaluate the train and test set performance using metrics: MSE, MAE and R2 score. Recall that the R2 score (or the coefficient of determination) is the proportion of the target variance that is explainable from the data.

R2 score ranges from negative infinity to 1, with higher scores indicating better performance. MSE and MAE range from 0 to infinity, with lower errors indicating better performance.

Method	Test MAE	Test MSE	Test R2	Train MAE	Train MSE	Train R2
GLM: Linear	23.981	1,270.218	0.447	24.959	1,368.679	0.444
GLM: Poisson	20.622	1,227.512	0.466	21.716	1,353.968	0.450
GLM: Tweedie	20.643	1,253.274	0.455	21.747	1,384.024	0.438

The test set performance immediately confirms one of our intuitions: classical regression approaches, which assume a normal distribution over the data fare the worst. Poisson or Tweedie distributions, however, show promise.

We now have trained our first 3 ML models: let us ensemble them by stacking them. The code below shows how, using artificial neural network (ANN) regression. While the GLMs we trained are linear, this stacked model will be nonlinear!

### Listing 7.10. Stacking GLMs for Bike Rental Prediction

```

from sklearn.neural_network import MLPRegressor
from sklearn.ensemble import StackingRegressor

base_estimators = list(best_glms.items()) #A
meta_learner = MLPRegressor(hidden_layer_sizes=(25, 25, 25), #B
                             max_iter=1000, activation='relu')

stack = StackingRegressor(base_estimators, final_estimator=meta_learner)
stack.fit(Xtrn, ytrn) #C

ypred_trn = stack.predict(Xtrn) #D
ypred_tst = stack.predict(Xtst)

res = {'Method': 'GLM Stack', #E
       'Train MSE': mean_squared_error(ytrn, ypred_trn),
       'Train MAE': mean_absolute_error(ytrn, ypred_trn),
       'Train R2': r2_score(ytrn, ypred_trn),
       'Test MSE': mean_squared_error(ytst, ypred_tst),
       'Test MAE': mean_absolute_error(ytst, ypred_tst),
       'Test R2': r2_score(ytst, ypred_tst)}
results = results.append(res, ignore_index=True)

```

#A GLMs with the best parameter settings from Listing 7.9, are base estimators

#B 3-layer neural network is the meta estimator

#C train the stacking ensemble

#D make train and test predictions

#E compute and save 3 metrics for this model: MAE, MSE, R2 score

Now, we can compare the results of stacking with the individual models

Method	Test MAE	Test MSE	Test R2	Train MAE	Train MSE	Train R2
GLM Stack	18.476	944.036	0.589	19.129	975.831	0.604

The stacked GLM ensemble already improves test set performance noticeably, indicating that nonlinear models are the way to go.

### 7.4.3 Random Forest and ExtraTrees

Now, let's train some more parallel ensembles for the bike rental prediction task using `scikit-learn`'s `RandomForestRegressor` and `ExtraTreesRegressor`.

Both modules only support MSE and MAE as the loss function, of which we will stick to the more standard MSE loss. We will train two individual regressors, with similar hyperparameter sweeps for each of them.

**NOTE** Mean Squared Error is not always a good choice of loss function for count-valued targets. The example below only serves to demonstrate how to use scikit-learn's ensemble modules for regression.

What we really want is to train random forests to emulate Poisson regression for the count-valued targets we have in this case study. scikit-learn (currently) does not have this functionality. It is still possible to use other packages such as LightGBM to train such random forest regressors. Recall from Section 5.5 that LightGBM has random forest training mode! As we will see below, LightGBM also supports Poisson regression with a log link function. This means that it is possible to use LightGBM to train a more effective random forest ensemble for this problem by changing the loss function. Try it!

For random forests and ExtraTrees, we are looking to identify the best choice of two hyperparameters: ensemble size (`n_estimators`) and the maximum depth of each base estimator (`max_depth`). The listing below demonstrates how we can train these regressors, with hyperparameter search through exhaustive grid search combined with cross validation, similar to what we did for GLMs.

#### Listing 7.11: Random Forest and ExtraTrees for Bike Rental Prediction

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import ExtraTreesRegressor

parameters = {'n_estimators': np.arange(200, 600, step=100),    #A
              'max_depth': np.arange(4, 7, step=1)}

ensembles = {'Random Forest': RandomForestRegressor(criterion='mse'),    #B
             'ExtraTrees': ExtraTreesRegressor(criterion='mse')}

for ens_type, ensemble in ensembles.items():
    param_tuner = GridSearchCV(ensemble, parameters,    #C
                               cv=5, refit=True, verbose=2)
    param_tuner.fit(Xtrn, ytrn)

    ypred_trn = param_tuner.best_estimator_.predict(Xtrn)    #D
    ypred_tst = param_tuner.best_estimator_.predict(Xtst)

    res = {'Method': ens_type,    #E
           'Train MSE': mean_squared_error(ytrn, ypred_trn),
           'Train MAE': mean_absolute_error(ytrn, ypred_trn),
           'Train R2': r2_score(ytrn, ypred_trn),
           'Test MSE': mean_squared_error(ytst, ypred_tst),
           'Test MAE': mean_absolute_error(ytst, ypred_tst),
           'Test R2': r2_score(ytst, ypred_tst)}
    results = results.append(res, ignore_index=True)
```

**#A** ranges of hyperparameters for both random forest and ExtraTrees

**#B** both ensembles use MSE as the training criterion

**#C** hyperparameter tuning with grid search and 5-fold CV

```
#D get train and test predictions for each ensemble
#E compute and save 3 metrics for each ensemble: MAE, MSE and R2 score
```

Compare the results of these parallel ensemble models with stacking and individual GLM models. In particular, observe the sharp improvement in performance compared to single models, which demonstrates the power of ensemble methods, even when trained on suboptimal loss functions.

Method	Test MAE	Test MSE	Test R2	Train MAE	Train MSE	Train R2
Random Forest	12.279	488.183	0.788	12.529	496.213	0.799
ExtraTrees	13.657	556.447	0.758	13.865	563.810	0.771

Can we get similar or better performance with gradient and Newton boosting methods? Let's find out.

#### 7.4.4 XGBoost and LightGBM

Finally, let's train sequential ensembles using both XGBoost and LightGBM on this data set. Both packages have support for a wide variety of loss functions:

- Some of the loss & likelihood functions that XGBoost supports are the MSE, pseudo-Huber loss and the Gamma, Poisson and Tweedie losses with the log link function. Note again that XGBoost implements Newton boosting, which requires computing second derivatives; this means that XGBoost cannot implement the MAE or Huber losses directly. Instead, XGBoost provides support for the pseudo-Huber loss.
- Like XGBoost, LightGBM supports the MSE and the Gamma, Poisson and Tweedie losses with the log link function. However, since it implements gradient boosting which only requires first derivatives, it directly supports the MAE and the Huber loss.

For both models, we will need to tune for several hyperparameters that control various aspects of ensembling (such as learning rate and early stopping), regularization (such as weights on the  $L_1$  and  $L_2$  regularizations) and tree learning (such as maximum tree depth).

Many of the previous models we trained only require tuning of a small number of hyperparameters, which allowed us to identify them through a grid search procedure.

Grid search becomes prohibitively computationally expensive and time consuming and should be avoided in instances such as this. Instead of an exhaustive grid search, randomized search can be an efficient alternative.

In randomized hyperparameter search, we sample a smaller number of random hyperparameter combinations from the full list. If necessary, we can further fine tune once we've identified a good combination to see if we can refine and improve our results further.

**Listing 7.12: XGBoost for Bike Rental Prediction**

```

from xgboost import XGBRegressor
from sklearn.model_selection import RandomizedSearchCV

parameters = {'max_depth': np.arange(2, 7, step=1),      #A
              'learning_rate': 2**np.arange(-8., 2., step=2),
              'colsample_bytree': [0.4, 0.5, 0.6, 0.7, 0.8],
              'reg_alpha': [0, 0.01, 0.1, 1, 10],
              'reg_lambda': [0, 0.01, 0.1, 1e-1, 1, 10]}
ensembles =      #B
    {'XGB: Squared Error': XGBRegressor(objective='reg:squarederror'),
     'XGB: Pseudo Huber': XGBRegressor(objective='reg:pseudohubererror'),
     'XGB: Gamma': XGBRegressor(objective='reg:gamma'),
     'XGB: Poisson': XGBRegressor(objective='count:poisson'),
     'XGB: Tweedie': XGBRegressor(objective='reg:tweedie')}

for ens_type, ensemble in ensembles.items():
    if ens_type == 'XGB: Tweedie':      #C
        parameters['tweedie_variance_power'] = np.linspace(1.1, 1.9, num=9)

    param_tuner = RandomizedSearchCV(ensemble, parameters, n_iter=50,      #D
                                    cv=5, refit=True, verbose=2)
    param_tuner.fit(Xtrn, ytrn, eval_set=[(Xtst, ytst)],      #E
                  eval_metric='poisson-nloglik', verbose=False)

    ypred_trn = param_tuner.best_estimator_.predict(Xtrn)      #F
    ypred_tst = param_tuner.best_estimator_.predict(Xtst)

    res = {'Method': ens_type,      #G
           'Train MSE': mean_squared_error(ytrn, ypred_trn),
           'Train MAE': mean_absolute_error(ytrn, ypred_trn),
           'Train R2': r2_score(ytrn, ypred_trn),
           'Test MSE': mean_squared_error(ytst, ypred_tst),
           'Test MAE': mean_absolute_error(ytst, ypred_tst),
           'Test R2': r2_score(ytst, ypred_tst)}
    results = results.append(res, ignore_index=True)

```

**#A** ranges of hyperparameters for all XGBoost loss functions  
**#B** initialize XGBoost models, each with a different loss function  
**#C** for the Tweedie loss, we have an additional hyperparameter: power  
**#D** hyperparameter tuning using randomized search with 5-fold CV  
**#E** select the best model using negative Poisson log-likelihood  
**#F** train & test predictions  
**#G** compute and save 3 metrics for each XGBoost ensemble: MAE, MSE, R2 score

**NOTE** The listing above uses early stopping to terminate training early if there is no noticeable performance improvement on an evaluation set. When we last employed early stopping on the AutoMPG data set (Listing 7.6), we used the MSE as the evaluation metric to track performance improvement.

Here, we use the negative Poisson log-likelihood (`eval_metric='poisson-nloglik'`). Recall from our discussion in Section 7.3.1 that negative log-likelihood is often used as a surrogate for loss functions without a closed form. In this case, since we are modeling count targets (which follow a Poisson distribution) it may be more appropriate to measure model performance with negative Poisson log-likelihood.

It would've also been appropriate to compare test set performances of different models with this metric alongside MSE, MAE and R2 as we've been doing. However, this metric is not always available or exposed in most packages.

The performance of XGBoost with different loss functions is shown below.

Method	Test MAE	Test MSE	Test R2	Train MAE	Train MSE	Train R2
XGB: Sq. Error	9.789	264.919	0.885	8.138	172.371	0.930
XGB: Ps. Huber	11.469	397.092	0.827	10.355	365.967	0.851
XGB: Gamma	9.625	306.216	0.867	9.254	287.669	0.883
XGB: Poisson	9.125	259.739	0.887	8.333	199.904	0.919
XGB: Tweedie	8.760	243.668	0.894	7.135	149.316	0.939

These results are dramatically improved, with XGBoost trained with Poisson and Tweedie losses performing the best.

We can repeat a similar experiment with LightGBM. The implementation for this (which can be found in the companion code) is quite similar to how we trained LightGBM models for the AutoMPG data set in Listing 7.6 and XGBoost models for the Bike Rental data set in Listing 7.11 above. The performance of LightGBM with MSE, MAE, Huber, Poisson and Tweedie losses is shown below.

Method	Test MAE	Test MSE	Test R2	Train MAE	Train MSE	Train R2
LGBM: Sq Error	9.476	256.803	0.888	8.243	181.150	0.926
LGBM: Abs Error	9.756	321.206	0.860	9.071	302.753	0.877
LGBM: Huber	12.235	707.681	0.692	12.561	752.741	0.694
LGBM: Poisson	9.120	252.658	0.890	8.706	218.423	0.911
LGBM: Tweedie	9.094	258.478	0.888	8.419	212.384	0.914

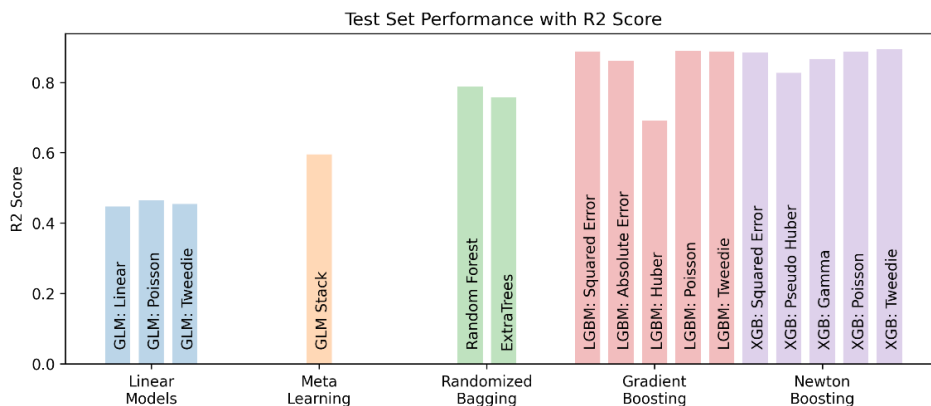
LightGBM's performance is similar to that of XGBoost, with Poisson and Tweedie losses, again, performing the best, with XGBoost edging LightGBM out slightly.

Figure 7.14 summarizes the test set performance (with R2 score) of all the models we have trained for the bike rental demand prediction tasks. We note the following:

- Individual GLMs perform far worse than any ensemble method. This is unsurprising since ensemble methods combine the power of many individual models into a final prediction. Furthermore, many of the ensemble regressors are nonlinear and fit the data better, while all GLMs are linear and limited.
- The appropriate choice of loss functions is critical to training a good model. In this case, LightGBM and XGBoost models trained with Tweedie fit and generalize best. This is because the Tweedie loss captures the distribution of the bike demand, which are

count-valued targets.

- Packages such as LightGBM and XGBoost provide loss functions such as the Tweedie, while scikit-learn's ensemble method implementations (random forest, XtraTrees) only support MSE and MAE losses (at the time of this writing). It's possible to push the performance of these methods up further by adopting losses such as the Tweedie, but this would require custom loss implementations.



**Figure 7.14.** The test set performance (with the R2 score metric) of the various ensemble methods for regression as we progressed through our analysis and modeling. Gradient and Newton boosting ensembles are the current state-of-the-art. Among these methods, performance can further be improved through a judicious choice of loss function and systematic parameter selection.

## 7.5 Summary

In this chapter, we learned about the types of problems that can be modeled with regression, and how we can train regression ensembles. Some key takeaways:

- Regression can be used to model continuous-valued, count-valued and even discrete valued targets.
- Classical linear models such as ordinary least squares, ridge regression, LASSO and elastic net all use the squared loss function, but different regularization functions.
- Poisson regression uses a linear model with a log-link function and the Poisson distribution assumption on the targets to effectively model count-labeled data
- Gamma regression uses a linear model with a log-link function and the gamma distribution assumption on the targets to effectively model continuous, but positively valued and right-skewed data
- Tweedie regression uses a linear model with a log-link function and the Tweedie distribution assumption to model compound distributions on data arising in many practical applications such as insurance, weather and health analytics.
- Classical mean-squared regression, Poisson regression, gamma regression, Tweedie regression (and even logistic regression) are all different variants of Generalized Linear Models (GLMs).

- Random Forest and ExtraTrees use randomized regression tree learning to induce ensemble diversity.
- Common statistical measures such as mean and median can be used to combine the predictions of continuous targets and mode and median to combine predictions of count targets.
- Artificial neural network regressors are good choices for meta-estimators when learning stacking ensembles.
- Loss functions such as mean-squared error, mean average deviation and Huber loss are well-suited for continuous valued labels
- The gamma likelihood function is well-suited for continuous valued, but positive labels (that is, they don't take negative values)
- The Poisson likelihood function is well-suited for count valued labels.
- Some problems contain a mix of these labels and can be modeled with a Tweedie likelihood function.
- LightGBM and XGBoost provide support for modeling both the log link (and other link functions) and distributions such as Poisson, gamma, and Tweedie.
- Hyperparameter selection, either through exhaustive grid search (slow, but thorough) or randomized search (fast, but approximate) is essential for good ensemble development in practice.



## 8

# Learning with Categorical Features

## This chapter covers

- An introduction to categorical features in machine learning
- Preprocessing categorical features using supervised and unsupervised encoding
- Understanding how ordered boosting works
- Introducing CatBoost: a powerful ordered boosting framework for categorical variables
- Handling high-cardinality categorical features

Data sets for supervised machine learning consist of features that describe objects, and labels that describe the targets we are interested in modeling. At a high level, features, also known as attributes or variables, are usually classified into two types: continuous and *categorical*.

A categorical feature is one that takes a discrete value from a set of finite, *non-numeric* values, called categories. Categorical features are ubiquitous and appear in nearly every data set and in every domain. For example,

- Demographic features such as `gender` or `race` are common attributes in many modeling problems in medicine, insurance, finance, advertising, recommendation systems and many more. For instance, the United States Census Bureau's `race` attribute is a categorical feature that admits 5 choices or categories: (1) American Indian or Alaska Native, (2) Asian, (3) Black or African American, (4) Native Hawaiian or Other Pacific Islander, (5) White.
- Geographical features such as `US State` or `ZIP code` are also categorical features. The feature `US State` is a categorical variable with 50 categories. The feature `ZIP code` is also a categorical variable, with 41,692 unique categories (!) in the United States, from 00501, belonging to the Internal Revenue Service in Holtsville, NY, to 99950 in Ketchikan, AK.

Categorical features are usually represented as strings or in specific formats (such as ZIP codes, which have to be exactly 5 digits long and can start with zeroes).

Since most machine-learning algorithms require numeric inputs, categorical features must be *encoded* or converted to numeric form before training. The nature of this encoding must be carefully chosen to capture the true underlying nature of the categorical features.

In the ensemble setting, there are two approaches to handling categorical features:

- Approach 1: *Pre-process* categorical features using one of several “standard” or general-purpose encoding techniques available in libraries such as `scikit-learn`, and then train ensemble models with packages such as LightGBM or XGBoost with the pre-processed features.
- Approach 2: Use an ensemble method, such as CatBoost, that is designed to handle categorical features while training ensembles directly and carefully.

Section 8.1 covers Approach 1. It introduces commonly used preprocessing methods for categorical features, and how we can use them in practice (using the `category_encoders` package) with any machine learning algorithm, including ensemble methods.

Section 8.1 also discusses two common problems: training-to-test-set *leakage* and *train-to-test distribution shift*, or *prediction shift*, which affects our ability to accurately evaluate the generalization ability of our models to future, unseen data.

Section 8.2 covers Approach 2 and introduces a new ensemble approach called *Ordered Boosting*, which is an extension of boosting approaches we’ve already seen but specially modified to address leakage and shift and designed for categorical features. This section also introduces the CatBoost package and shows how we can use it to train ensemble methods on data sets with categorical features.

We explore both approaches in a real-world case study in Section 8.3, where we compare random forest, LightGBM, XGBoost and CatBoost on an income prediction task.

Finally, many general-purpose approaches do not scale well to *high-cardinality categorical features* (where the number of categories is very high, such as ZIP code) or in the presence of noise, or so-called “dirty” categorical variables. Section 8.4 shows how we can effectively handle such high-cardinality categories with the `dirty-cat` package.

## 8.1 Encoding Categorical Features

This section reviews the different *types* of categorical features introduces two classes of standard approaches to handling them: unsupervised (specifically, ordinal and one-hot encoding) and supervised encoding (specifically, with target statistics).

This section reviews the different types of categorical features and introduces some standard methods of handling them.

Encoding techniques, like machine-learning methods, are either *unsupervised* and *supervised*. Unsupervised encoding methods use only the features to encode categories, while supervised encoding methods use both features and targets.

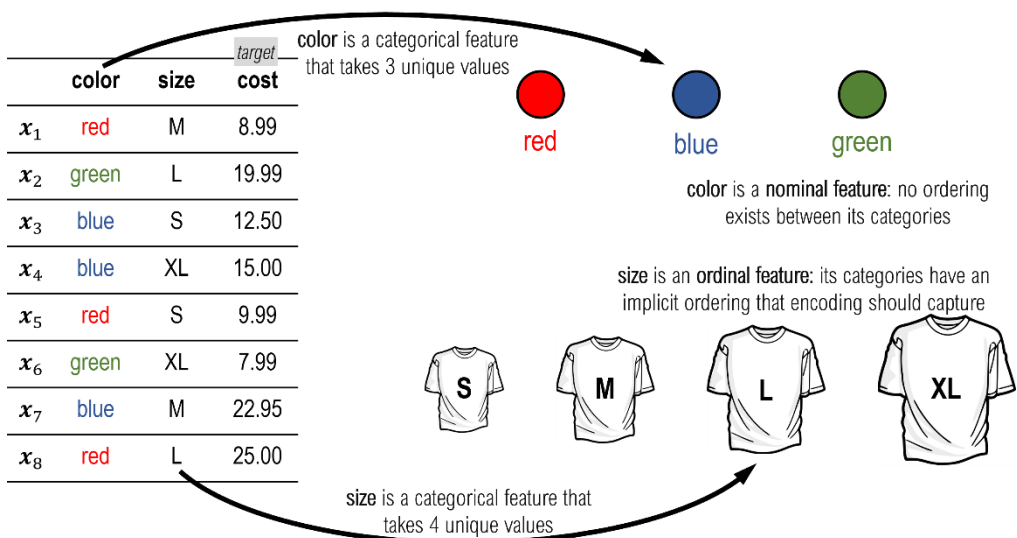
We will also see how supervised encoding techniques can lead to degraded performance in practice owing to a phenomenon called *target leakage*. This will help us understand the motivations behind the development of the ordinal boosting approach, which we will explore in Section 8.3.

### 8.1.1 Types of Categorical Features

A categorical feature contains information about a category or group that a training example belongs to. The values, or categories that make up such variables are often represented using strings or other non-numeric tags.

Broadly, categorical features are of two types: *ordinal*, where an ordering exists between the categories and *nominal*, where no ordering exists between the categories.

Let's look closely at nominal and ordinal categorical features in the context of a hypothetical fashion task, where the goal is to train a machine-learning algorithm to predict the `cost` of a t-shirts. Each t-shirt is described by two attributes: `color` and `size` (Figure 8.1.).



**Figure 8.1.** Training examples, in this case, t-shirts are described using two categorical features: `color` and `size`. Categorical features can be either (1) nominal, where there is no ordering between the various categories, or (2) ordinal, where there exists an ordering between the categories. The third feature in this data set, `cost`, is a continuous, numeric variable.

The feature `color` takes three discrete values: red, blue, and green. No ordering exists between these categories, which makes `color` a nominal feature. Since it doesn't matter how we order `color`'s values, the ordering red-blue-green is equivalent to other ordering permutations such as blue-red-green or green-red-blue.

The feature `size` takes four discrete values: S, M, L, and XL. Unlike, `color`, however, there is an implicit ordering between the sizes:  $S < M < L < XL$ . This makes `size` an ordinal feature.

While we can order sizes any way we want, ordering them in increasing order, S-M-L-XL or in decreasing order of size, XL-L-M-S is most sensible.

Understanding the domain and the nature of each categorical feature is an important component of deciding how to encode them.

## 8.1.2 Ordinal and One-Hot Encoding

Categorical variables such as color and size have to be encoded, that is, converted to some sort of numeric representation prior to training a machine-learning model.

Encoding is a type of feature engineering and must be done with care as an inappropriate choice of encoding can affect model performance and interpretability.

In this section, we will look at two commonly used unsupervised methods of encoding categorical variables are: *ordinal encoding* and *one-hot encoding*. They are unsupervised as they do not use the targets (labels) for encoding.

### ORDINAL ENCODING

Ordinal encoding simply assigns each category a number. For example, the nominal feature `color` can be encoded by assigning `{'red': 0, 'blue': 1, 'green': 2}`. Since the categories don't have any implicit ordering, we could have also encoded by assigning other permutations such as `{'red': 2, 'blue': 0, 'green': 1}`.

On the other hand, since `size` is already an ordinal variable, it makes sense to assign numeric values to preserve this ordering. For `size`, either encoding with `{'S': 0, 'M': 1, 'L': 2, 'XL': 3}` (increasing) or `{'S': 3, 'M': 2, 'L': 1, 'XL': 0}` (decreasing) preserves the inherent relationship between the `size` categories.

`scikit-learn`'s `OrdinalEncoder` can be used to create ordinal encodings. Let's encode the two categorical features (`color` and `size`) in the data set from Figure 8.1 (denoted by `X` below).

```
import numpy as np
X = np.array([[ 'red', 'M'],
              [ 'green', 'L'],
              [ 'red', 'S'],
              [ 'blue', 'XL'],
              [ 'blue', 'S'],
              [ 'green', 'XL'],
              [ 'blue', 'M'],
              [ 'red', 'L']])
```

We will specify our encoding for `color` assuming it can take four values: red, yellow, green, blue (even though we only see red, green, and blue in our data). We will also specify the ordering for `size`: XL, L, M, S.

```
from sklearn.preprocessing import OrdinalEncoder
encoder = OrdinalEncoder(categories=[['red', 'yellow', 'green', 'blue'], #A
                                   ['XL', 'L', 'M', 'S']])           #B
Xenc = encoder.fit_transform(X)                                     #C
```

**#A** specify that there are four possible colors  
**#B** specify that size should be organized in decreasing order  
**#C** encode categorical features only using this specification

Now, we can look at the encodings for these features:

```
encoder.categories_
[array(['red', 'yellow', 'green', 'blue'], dtype='<U5'),
 array(['XL', 'L', 'M', 'S'], dtype='<U5')]
```

This encoding assigns numeric values to `color` as {'red': 0, 'yellow': 1, 'blue': 2, 'green': 3} and to `size` as {'XL': 0, 'L': 1, 'M': 2, 'S': 3}. This encoding transforms these categorical features to numeric values:

```
Xenc
array([[0., 2.],
       [2., 1.],
       [0., 3.],
       [3., 0.],
       [3., 3.],
       [2., 0.],
       [3., 2.],
       [0., 1.]])
```

Compare the encoded `color` (the first column of `Xenc`) with the raw data (the first column of `X`). All the entries `red` are encoded as 0, `green` as 2 and `blue` as 3. As there are no entries, `yellow`, we have no encodings of value 1 in this column.

Note that ordinal encoding imposes an inherent ordering between variables. While this is ideal for ordinal categorical features, it may not always make sense for nominal categorical features.

#### ONE-HOT ENCODING

One-hot encoding is a way to encode a categorical feature without imposing any ordering among its values and is more suited for nominal features.

Why use one-hot encoding? If we use ordinal encoding for nominal features, it would introduce an ordering that does not exist between the categories in the real-world, thus misleading the learning algorithm into thinking there was one. Unlike ordinal encoding, which encodes each category using a single number, one-hot encoding encodes each category using a vector of 0s and 1s. The size of the vector depends on the number of categories.

For example, if we assume that `color` is a 3-valued category (red, blue, green), it will be encoded as a length-3 vector. One such one-hot encoding can be {'red': [1, 0, 0], 'blue': [0, 1, 0], 'green': [0, 0, 1]}.

Observe the position of the 1s: red corresponds to the first encoding entry, blue corresponds to the second and green to the third.

If we assume that `color` is 4-valued category (red, yellow, blue, green), one-hot encoding will produce length-4 vectors for each category.

Since `size` takes 4 unique values, one-hot encoding produces length-4 vectors for each `size` category as well. One such one-hot encoding can be {'S': [1, 0, 0, 0], 'M': [0, 1, 0, 0], 'L': [0, 0, 1, 0], 'XL': [0, 0, 0, 1]}.

`scikit-learn`'s `OneHotEncoder` can be used to create one-hot encodings. As before, let's encode the two categorical features (`color` and `size`) in the data set from Figure 8.1.

```

from sklearn.preprocessing import OneHotEncoder
encoder = OneHotEncoder(categories=[[ 'red', 'green', 'blue' ],      #A
                                  [ 'XL', 'L', 'M', 'S' ]])      #B
Xenc = encoder.fit_transform(X)      #C

```

#A specify that there are three possible colors

#B specify that there are four possible sizes

#C encode categorical features only using this specification

Now, we can look at the encodings for these features:

```

encoder.categories_
[array([ 'red', 'green', 'blue' ], dtype='<U5'),
 array([ 'S', 'M', 'L', 'XL' ], dtype='<U5')]

```

This encoding will introduce 3 one-hot features (first 3 columns in `Xenc`) to replace the `color` feature (first column in `X`) and 4 one-hot feature (last 4 columns in `Xenc`) to replace the `size` feature (last column in `X`).

```

Xenc.toarray()
array([[1., 0., 0., 0., 1., 0., 0.],
       [0., 1., 0., 0., 0., 1., 0.],
       [1., 0., 0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0., 1.],
       [0., 0., 1., 1., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 1.],
       [0., 0., 1., 0., 1., 0., 0.],
       [1., 0., 0., 0., 0., 1., 0.]])

```

Each individual category has its own column now (3 for each `color` category and 4 for each `size` category), and any ordering between them has been lost.

Since one-hot encoding removes any inherent ordering between categories, it is an ideal choice to encode nominal features.

This choice, however, comes with a cost: we have blown up the size of our data set. The original data set was 8 examples x 2 features. With ordinal encoding, it remained 8 x 2, though a forced ordering was imposed upon the nominal feature, `color`. With one-hot encoding, the size became 8 x 7, and the inherent ordering in the ordinal feature, `size`, was removed.

### 8.1.3 Encoding with Target Statistics

We now shift our focus to *encoding with target statistics*, or *target encoding*, which is an example of a supervised encoding technique. In contrast to unsupervised encoding methods, supervised encoding methods use labels to encode categorical features.

The idea behind encoding with target statistics is fairly straightforward: for each category, we compute a statistic such as the mean over the targets (that is, labels) and replace the category with this newly-computed numerical statistic. Encoding with label information often helps overcome the drawbacks of unsupervised encoding methods.

Unlike one-hot encoding, target encoding doesn't create any additional columns meaning the dimensionality of the overall data set remains the same after encoding. Unlike ordinal encoding, target encoding doesn't introduce spurious relationships between the categories.

### GREEDY TARGET ENCODING

In the example fashion data set from the previous section, recall that each training example is a t-shirt with two attributes: `color` and `size` and the target to predict is `cost`. Let's say that we would like to encode the `color` feature with target statistics. This feature has three categories: `red`, `blue`, and `green` that need to be encoded.

Figure 8.2 illustrates how encoding with target statistics works for the category `red`.

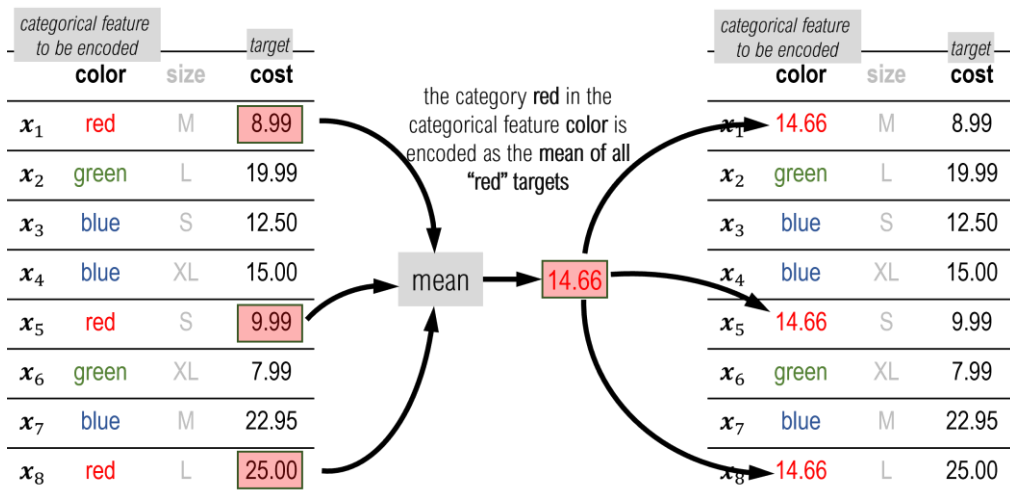


Figure 8.2. The category `red` of the feature `color` is replaced by its target statistic, the average (mean) of all the target values (`cost`) corresponding to the examples whose `color` is `red`. This is called greedy target encoding as all the training labels have been used for encoding.

There are three t-shirts  $x_1$ ,  $x_5$  and  $x_8$  whose `color` is `red`. Their corresponding target values (`cost`) are 8.99, 9.99 and 25.00. The target statistic is computed as the mean of these values:  $(8.99 + 9.99 + 25.00) / 3 = 14.66$ .

Thus, each instance of `red` is replaced by its corresponding target statistic: 14.66. The other two categories, `blue` and `green`, can similarly be encoded with their corresponding target statistics, 16.82 and 13.99.

More formally, the target statistic for the  $k$ -th category of the  $j$ -th feature can be computed using the formula

$$s_k = \frac{\sum_{i=1}^n \mathbf{I}(x_i^j = k) \cdot y_i + a p}{\sum_{i=1}^n \mathbf{I}(x_i^j = k) + a}$$

Here, the notation  $\mathbf{I}(x_i^j = k)$  denotes an indicator function, which returns 1 if true and 0 if false. For example, in our fashion data set,  $\mathbf{I}(x_1^{\text{color}} = \text{red}) = 1$ , whereas  $\mathbf{I}(x_4^{\text{color}} = \text{red}) = 0$ .

This formula for computing target statistics actually computes a *smoothed average* rather than just the average. Smoothing is performed by adding a parameter  $\alpha$  to the denominator. This is to ensure that categories with a small number of values (and hence small denominators) do not end up with target statistics that are scaled differently to other categories.

The constant  $p$  in the numerator is typically the average target value of the entire data set, and serves as a *prior*, or as a means of regularizing the target statistic.

This target encoding approach is called *greedy target encoding*, as it uses all the available training data to compute the encodings. As we see below, a greedy encoding approach leaks information from the training to the test set. This is problematic because a model identified as high-performing during training and testing will often actually perform poorly in deployment and production.

#### INFORMATION LEAKAGE AND DISTRIBUTION SHIFT

Many pre-processing approaches are affected by one or both of two common practical issues: *training-to-test-set information leakage* and *train-test distribution shift*. Both issues affect our ability to evaluate our trained model and accurately estimate how it will behave on future, unseen data, that is, how it will generalize.

A key step in machine-learning model development is the creation of a *hold-out test set*, which is used to evaluate trained models. The test set is completely held-out from every stage of modeling (including pre-processing, training, and validation) and used purely for evaluating model performance.

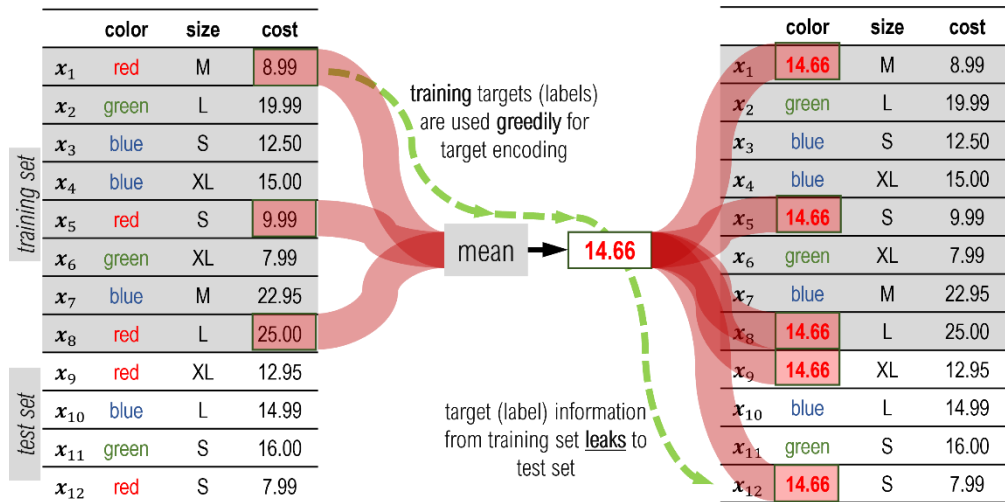
The reason for this is to simulate model performance on unseen data; to do this effectively, we have to ensure that no part of the training data makes its way into test data. When this happens during modeling it is called *information leakage from the training to test set*.

Data leakage occurs when information about the features leaks into the test set, while target leakage occurs when information about the targets (label) leaks into the test set.

Greedy target encoding leads to target leakage, as illustrated in Figure 8.3. In this example, a data set of 12 data points is partitioned into a training and test set. The training set is used to perform greedy target encoding of the category `red` of the feature `color`.

More specifically, the target encoding from the training set is used to transform *both* the training and the test set, leading to information leakage about the targets from the training set to the test set, making this an instance of target leakage.





**Figure 8.3. Target leakage from the training to test set illustrated. All the targets (labels) of in the training set are greedily used to create an encoding for red, which is used to encode this category in both the training and test sets, leading to target leakage.**

Another requirement of the train-test split is to ensure that the training and test sets have similar distributions, that is, they have similar statistical properties. This is often achieved by randomly sampling the hold-out test set from the overall set.

However, pre-processing techniques such as greedy target encoding can introduce disparities between the training and test sets, leading to a *prediction shift* between the training and test sets.

This is illustrated in Figure 8.4. As before, the category `red` for the feature `color` is encoded using greedy target statistics. This encoding is computed as the mean of the targets corresponding to `color == red` in the training data and is 14.66.

However, if we compute the mean of the targets corresponding to `color == red` in the test data only, the mean is 10.47. This discrepancy between the training and test sets is a by-product of greedy target encoding, which causes the test set distributions to become shifted from the training set distribution.

Put another way, the statistical properties of the test set are now no longer similar to that of the training set, which has an inevitable and cascading influence on our model evaluation.

Both target leakage and prediction shift introduce a statistical bias into the performance metrics we use to evaluate the generalization performance of our trained models. Often, they overestimate generalization performance and make the trained model look better than it actually is, which causes a problem when this model is deployed and fails to perform according to expectations.

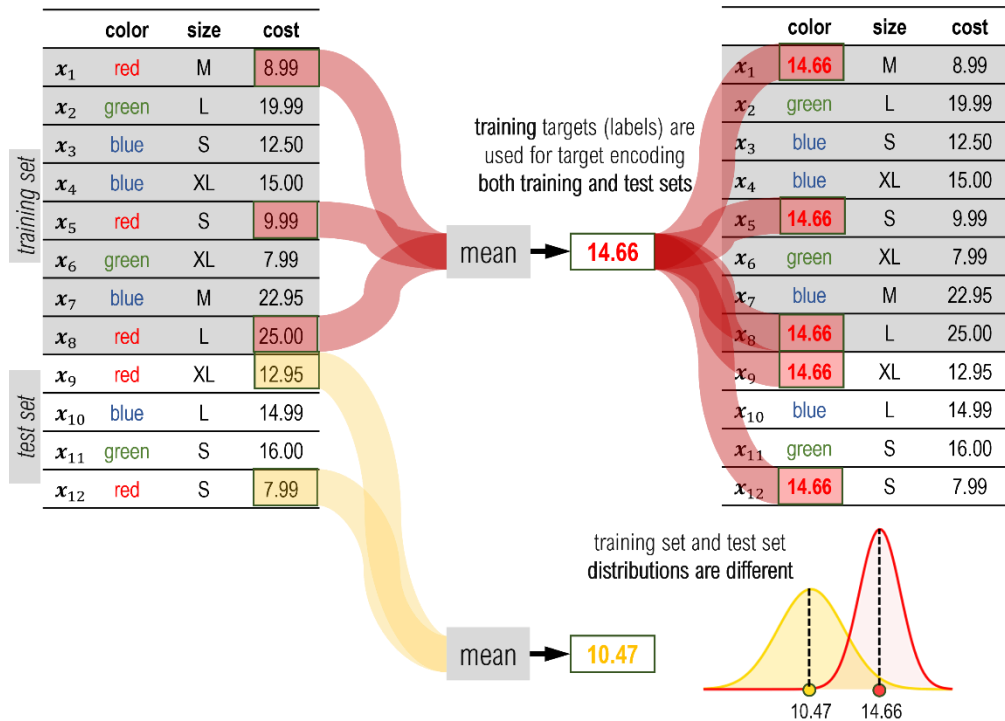
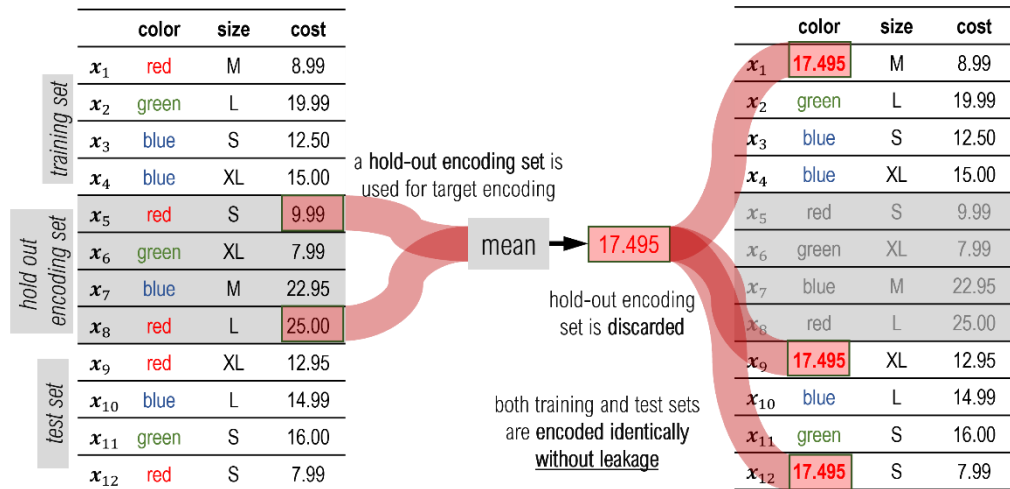


Figure 8.4. Distribution shift between the training and test sets illustrated. Since the target encoding for the test set is computed using the training set, it can lead to a shift in the distribution and statistical properties of the test set (yellow) compared to the training set (red).

#### HOLD-OUT & LEAVE-ONE-OUT TARGET ENCODING

The best (and simplest) way to eliminate both target leakage and prediction shift is to hold-out a part of the training data for encoding. Thus, in addition to the training and hold-out test sets, we would also need to create a hold-out encoding set!

This approach, called *hold-out target encoding*, is illustrated in Figure 8.5. Here, our data set from Figure 8.3 and Figure 8.4 is split into three sets, each with four data points: a training set, a hold-out encoding set and a hold-out test set.



**Figure 8.5. Hold-out encoding partitions the available data into three sets: training and test, as usual, and a third hold-out test set to be used exclusively for encoding with target statistics. This avoids both target leakage and distribution shift.**

The hold-out encoding set is used to compute the target encoding for both the training and test sets. This ensures that the independence of training and test sets and eliminates target leakage. Further, since the same target statistic is used for both training and test sets, it also avoids prediction shift.

A key drawback of hold-out target encoding is its *data inefficiency*. In order to avoid leakage, once the hold-out encoding set is used to compute the encoding, it needs to be discarded, which means that a good chunk of the total data available for modeling can be wasted.

One (imperfect) alternative is to use *leave-one-out* (LOO) target encoding, which is illustrated in Figure 8.6. LOO encoding works similarly to LOO cross-validation, except that the left-out example is being encoded rather than being validated.

In Figure 8.6, we see that to perform LOO target encoding for the red example  $x_5$ , we compute the target statistic using the other two red training examples  $x_1$  and  $x_8$ , while leaving out  $x_5$ . This procedure is repeated for the other two red training examples  $x_1$  and  $x_8$  in turn.

Since the test set and the training set do not overlap, it's not possible to leave out test examples while encoding with training examples. Thus, we can apply greedy target encoding as before for the test set.

As we can see, the LOO target encoding procedure aims to emulate hold-out target encoding, while being significantly more data efficient. However, it should be noted that this overall procedure does not fully eliminate target leakage and prediction shift issues.

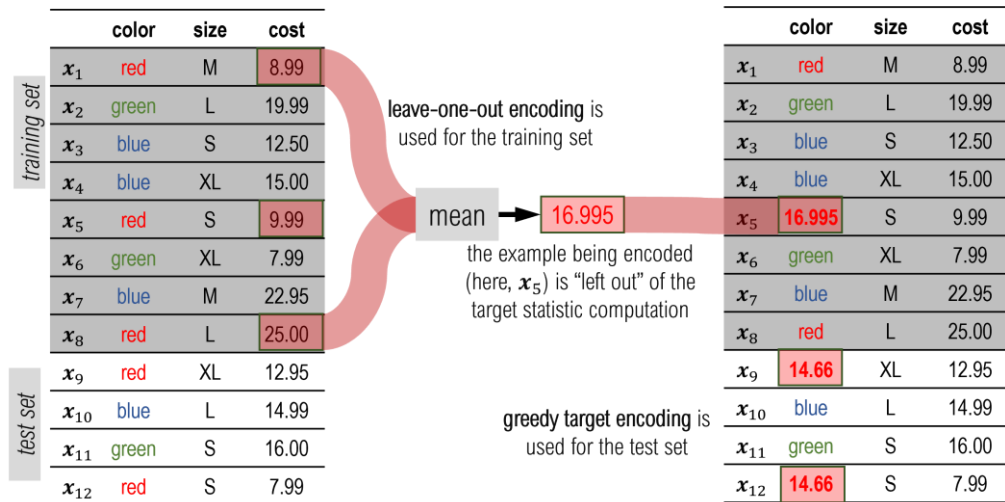


Figure 8.6. Leave-one-out target encoding is applied to the training data to avoid creating a wasteful hold-out encoding set. Instead of holding out a subset of the data, only the example being encoded is held out. Test data is encoded using greedy target encoding as before.

As we will see in Section 8.2, another encoding strategy called *ordered target statistics* aims to further mitigate the issues of target leakage and prediction shift, while ensuring both data as well as computational efficiency.

### 8.1.4 The `category_encoders` Package

This section provides examples of how to put together end-to-end encoding and training pipelines for data sets with categorical features. The subpackage `sklearn.preprocessing` provides some common encoders such as `OneHotEncoder` and `OrdinalEncoder`.

We will, instead, use the `category_encoders`<sup>1</sup> package, which provides many more encoding strategies including for greedy and leave-one-out target encoding.

`category_encoders` is scikit-learn compatible, which means that it can be used with other ensemble method implementations that provide sklearn-compatible interfaces (such as LightGBM and XGBoost) discussed in this book.

We will use the Australian Credit Approval data set from the UCI Machine Learning repository<sup>2</sup>. A clean version of this data set is available along with the source code, which we will use to demonstrate category encoding in practice.

The data set contains 6 continuous, 4 binary features and 4 categorical features, and the task is to determine whether to approve or deny a credit card application, that is, binary classification.

First, let's load the data set and look at the feature names and the first few rows.

<sup>1</sup> [https://contrib.scikit-learn.org/category\\_encoders/](https://contrib.scikit-learn.org/category_encoders/)

<sup>2</sup> [https://archive.ics.uci.edu/ml/datasets/statlog+\(australian+credit+approval\)](https://archive.ics.uci.edu/ml/datasets/statlog+(australian+credit+approval))

```
import pandas as pd
df = pd.read_csv('./data/ch08/australian-credit.csv')
df.head()
```

	f1-bin	f2-cont	f3-cont	f4-cat	f5-cat	f6-cat	f7-cont	f8-bin	f9-bin	f10-cont	f11-bin	f12-cat	f13-cont	f14-cont	target
0	1	22.08	11.46	2	4	4	1.585	0	0	0	1	2	100	1213	0
1	0	22.67	7.00	2	8	4	0.165	0	0	0	0	2	160	1	0
2	0	29.58	1.75	1	4	4	1.250	0	0	0	1	2	280	1	0
3	0	21.67	11.50	1	5	3	0.000	1	1	11	1	2	0	1	1
4	1	20.17	8.17	2	6	4	1.960	1	1	14	0	2	60	159	1

**Figure 8.7.** The Australian credit data set from the UCI repository. Attribute names have been changed to protect confidentiality of the individuals represented in the data set.

The feature names are of the form f1-bin, f2-cont or f5-cat, indicating the column index and whether the feature is binary, continuous, or categorical.

To protect applicant confidentiality, the category strings and names have been replaced with integer values. That is, the *categorical features have already been processed with ordinal encoding!*

Let's separate the columns into the data set and the targets and further split into training and test sets as usual.

```
X, y = df.drop('target', axis=1), df['target']
from sklearn.model_selection import train_test_split
Xtrn, Xtst, ytrn, ytst = train_test_split(X, y, test_size=0.2)
```

Furthermore, let's explicitly identify the categorical and continuous features we're interested in pre-processing:

```
cat_features = ['f4-cat', 'f5-cat', 'f6-cat', 'f12-cat']
cont_features = ['f2-cont', 'f3-cont', 'f7-cont', 'f10-cont',
                 'f13-cont', 'f14-cont']
```

We will pre-process the continuous and categorical features in different ways. The continuous features will be standardized: that is, each column of continuous features is rescaled to have zero mean and unit standard deviation. This rescaling ensures that different columns do not have drastically different scales, which can mess up downstream learning algorithms.

The categorical features will be pre-processed using one-hot encoding. For this, we will use the `OneHotEncoder` from the `category_encoders` package.

We will create two separate pre-processing pipelines, one for continuous features and one for categorical features.

```
import category_encoders as ce
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

preprocess_continuous = Pipeline(steps=[('scaler', StandardScaler())])
preprocess_categorical = Pipeline(steps=[('encoder',
                                         ce.OneHotEncoder(cols=cat_features))])
```

Note that `ce.OneHotEncoder` requires us to explicitly specify the categorical features, without which it will apply encoding to *all* the columns.

Now that we have two separate pipelines, we need to put these together to ensure that the correct pre-processing is applied to the correct feature type. We can do this with `scikit-learn`'s `ColumnTransformer`, which allows us to apply different steps to different columns.

```
from sklearn.compose import ColumnTransformer
ct = ColumnTransformer(
    transformers=[('continuous', #A
                 preprocess_continuous, cont_features),
                 ('categorical', #B
                 preprocess_categorical, cat_features)],
    remainder='passthrough') #C
```

**#A preprocess continuous features here**

**#B preprocess categorical features here**

**#C keep the remaining features as-is**

Now, we can fit a pre-processor on the training set and apply the transformation to both the training and test sets

```
Xtrn_one_hot = ct.fit_transform(Xtrn, ytrn)
Xtst_one_hot = ct.transform(Xtst)
```

Observe how the test set is not used to fit the pre-processor pipeline. This is a subtle but important practical step to ensure that the test set is *held-out* and there is no inadvertent data or target leakage due to pre-processing. Now, let's see what one-hot encoding has done to our feature set size:

```
print('Num features after ONE HOT encoding = {}'.format(
    Xtst_one_hot.shape[1]))
Num features after ONE HOT encoding = 38
```

Since one-hot encoding introduces one new column for each category of a categorical feature, the overall number of columns has increased from 14 to 38!

Now let's train and evaluate a `RandomForestClassifier` on this pre-processed data set

```

from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier(n_estimators=200,
                             max_depth=6, criterion='entropy')
model.fit(Xtrn_one_hot, ytrn)

from sklearn.metrics import accuracy_score
ypred = model.predict(Xtst_one_hot)
print('Model Accuracy using ONE HOT encoding = {0:5.3f}%'.
      format(100 * accuracy_score(ypred, ytst)))

Model Accuracy using ONE HOT encoding = 84.058%

```

Our one-hot encoding strategy learned a model whose hold-out test accuracy is 84%.

In addition to `OneHotEncoder` and `OrdinalEncoder`, the `category_encoders` package also provides many other encoders.

Two of encoders of interest to us are the greedy `TargetEncoder` and the `LeaveOneOutEncoder`, which can be used in exactly the same way as `OneHotEncoder` above. Specifically, we simply replace `OneHotEncoder` with `TargetEncoder` in

```

preprocess_categorical = Pipeline(steps=[('encoder',
                                         ce.TargetEncoder(cols=cat_features,
                                                           smoothing=10.0))])

```

`TargetEncoder` takes one additional parameter, `smoothing`, a positive value that combines the effect of smoothing and prior (see Section 8.1.2). Higher values force higher smoothing and can counter overfitting. After pre-processing and training, we have

```

Num features after GREEDY TARGET encoding = 14
Model Accuracy using GREEDY TARGET encoding = 82.609%

```

Unlike, one-hot encoding, greedy target encoding does not add any new columns, which means that the overall dimensions of the data set remain unchanged.

We can use `LeaveOneOutEncoder` in a similar way:

```

preprocess_categorical = Pipeline(steps=[('encoder',
                                         ce.LeaveOneOutEncoder(cols=cat_features,
                                                                    sigma=0.4))])

```

The `sigma` parameter is a noise parameter that aims to decrease overfitting. The user manual recommends using values between 0.05 to 0.6. After pre-processing and training, we again have

```

Num features after LEAVE-ONE-OUT TARGET encoding = 14
Model Accuracy using LEAVE-ONE-OUT TARGET encoding = 84.058%

```

As with `TargetEncoding`, the number of features remain unchanged due to preprocessing.

## 8.2 CatBoost: A Framework for Ordered Boosting

*CatBoost* is another open source gradient boosting framework developed by Yandex. *CatBoost* introduces three major modifications to classical Newton boosting approach.

First, it is specialized to categorical features, unlike other boosting approaches that are more general. Second, it uses ordered boosting as its underlying ensemble learning approach,

which allows it to address data leakage and prediction shift implicitly during training. Third, it uses *oblivious decision trees* as base estimators, which often leads to faster training times.

### 8.2.1 Ordered Target Statistics and Ordered Boosting

CatBoost handles categorical features in two ways: (a) by encoding categorical features as described before with target statistics, and (b) by cleverly creating categorical combinations of features (and encoding them with target statistics as well). While these features enable CatBoost to seamlessly handle categorical features, they do introduce other downsides that must be addressed.

As we've seen before, encoding with target statistics introduces target leakage and more importantly, a *prediction shift* in the test set. The most ideal way to handle this by creating a hold-out encoding set.

Holding out training examples for just encoding and nothing else is rather wasteful of data, meaning that this approach is rarely used in practice. The alternative, leave-one-out encoding, is more data-efficient, but does not completely mitigate prediction shift.

In addition to issues with encoding features, gradient and Newton boosting both reuse data between iterations, leading to a gradient distribution shift, which ultimately causes a *further prediction shift*. That is to say, even if we didn't have categorical features, we would still have a prediction shift problem, which would bias our estimates of model generalization!

CatBoost addresses this central issue of prediction shift by using permutation for ordering training examples to (1) compute target statistics for encoding categorical variables (called ordered target statistics), and (2) train its weak estimators (called ordered boosting).

#### ORDERED TARGET STATISTICS

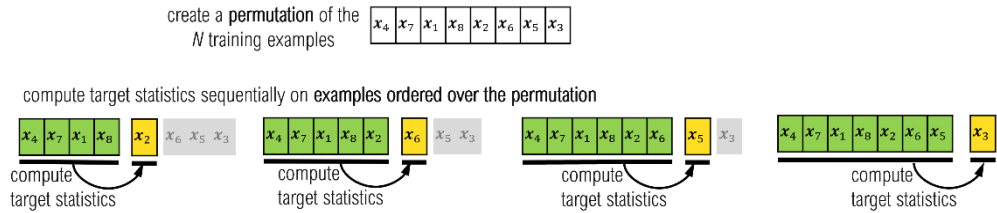
At its heart, the ordering principle is simple and elegant and consists of two steps:

1. reorder the training examples according to a random permutation
2. to compute target statistics for the  $i$ -th training example, use the previous  $i-1$  training examples according to this random permutation

This is illustrated in Figure 8.8 for eight training examples. First, the examples are permuted into a random ordering: 4, 7, 1, 8, 2, 6, 5, 3. Now, to compute target statistics for each training example, we assume that we can only see these examples sequentially.

For example, to compute the target statistics for example 2, we can only use examples in the sequence that we have "previously seen": 4, 7, 1 and 8. Then, to compute the target statistics for example 6, we can only use examples in the sequence that we have previously seen, now: 4, 7, 1, 8, and 2. And so on.





**Figure 8.8.** Ordered target statistics first permutes the examples into a random sequence, using only the previous examples in the ordered sequence to compute the target statistics.

Thus, to compute the encoding for the  $i$ -th training example, ordered target statistics never uses its own target value; this behavior is similar to leave-one-out target encoding. The key difference between the two is that ordered target statistics uses the notion of a “history”.

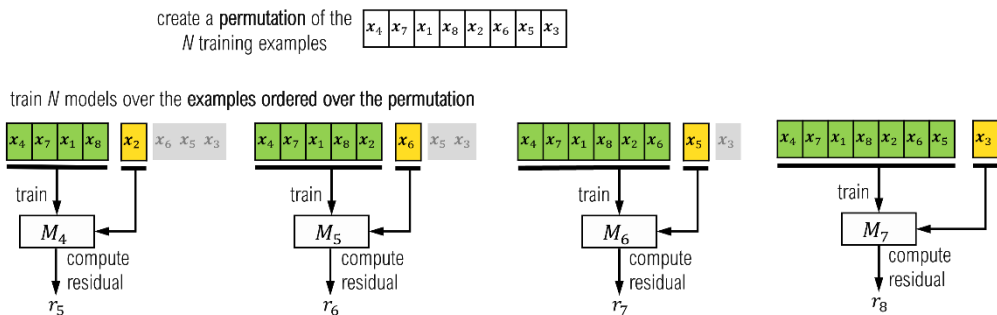
One downside to this approach is that training examples that occur early in a randomized sequence are encoded with far fewer examples.

To compensate for this in practice and increase robustness, CatBoost maintains several sequences, or “histories” which are, in turn, randomly chosen. This means that CatBoost recomputes target statistics for categorical variables at each iteration.

#### ORDERED BOOSTING

CatBoost is fundamentally a Newton boosting algorithm (see Chapter 6), that is, it uses both the first and second gradient of the loss function to train its constituent weak estimators.

As mentioned previously, there are two sources of prediction shift: variable encoding and gradient computations themselves. To avoid prediction shift due to gradients, CatBoost extends the idea of ordering to training its weak learners. Another way to think about this is: Newton boosting + ordering = CatBoost.



**Figure 8.9.** Ordered boosting also permutes the examples into a random sequence and uses only the previous examples in the ordered sequence to compute the gradients (residuals).

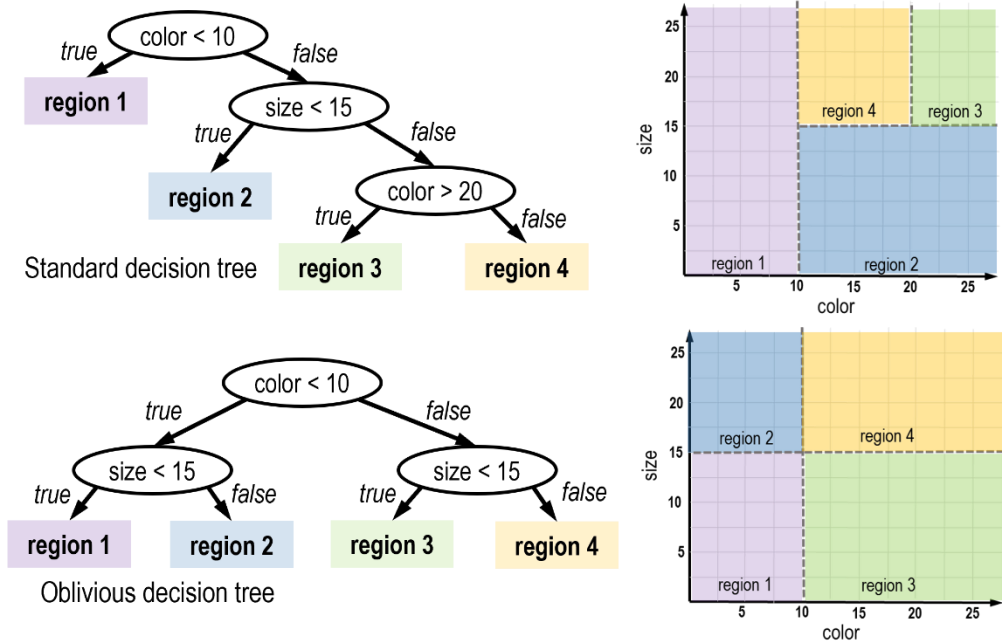
Figure 8.9 illustrates ordered boosting, analogous to ordered target statistics. For example, to compute the residuals and gradients for example 2, ordered boosting uses a model only trained on the examples in the sequence that it has “previously seen”: 4, 7, 1 and 8.

As with ordered target statistics, CatBoost uses multiple permutations to increase robustness. These residuals are now used to train its weak estimators.

## 8.2.2 Oblivious Decision Trees

Another key difference between Newton boosting implementations such as XGBoost and CatBoost are the base estimators. XGBoost uses standard decision trees as weak estimators, while CatBoost uses *oblivious decision trees*.

Oblivious decision trees use the same splitting criterion in all the nodes across an entire level/depth of the tree. This is illustrated in the figure above, which compares a standard decision tree with 4 leaf nodes with an oblivious decision tree with 4 leaf nodes.



**Figure 8.10.** Comparing standard and oblivious decision trees, each with 4 leaf nodes. Observe that the decision nodes at depth 2 of the oblivious decision tree are both the same ( $size < 15$ ). This is a key feature of oblivious decision trees: only one split criterion is learned for each depth.

In this example, observe that the second level of the oblivious tree (bottom right), uses the same decision criterion,  $size < 15$ , at each node in the second level.

While this is a simple example, note already that we only need to learn 2 split criteria for the oblivious tree, as opposed to the standard decision tree. This makes oblivious trees easier and more efficient to learn, which has the effect of speeding up overall training.

In addition, oblivious trees are balanced and symmetric, making them less complex and thus, less prone to overfitting.

### 8.2.3 CatBoost in Practice

In this section, we will see how we can create a learning pipeline with CatBoost. We will also look at an example of how to set the learning rate and employ early stopping as a means to control overfitting. To recap,

- by selecting an effective learning rate, we try to control the rate at which the model learns so that it doesn't rapidly fit, and then overfit the training data. We can think of this a proactive modeling approach, where we try to identify a good training strategy so that it leads to a good model.
- by enforcing early stopping, we try to stop training as soon as we observe that the model is starting to overfit. We can think of this as a reactive modeling approach, where we contemplate terminating training as soon as we think we have a good model.

We will use the Australian Credit Approval data that we used in Section 8.1.4. The listing below provides a simple illustration of how to use CatBoost.

#### Listing 8.1. Using CatBoost

```
import pandas as pd
df = pd.read_csv('./data/ch08/australian-credit.csv') #A
cat_features = ['f4-cat', 'f5-cat', 'f6-cat', 'f12-cat'] #B

X, y = df.drop('target', axis=1), df['target']

from sklearn.model_selection import train_test_split
Xtrn, Xtst, ytrn, ytst = train_test_split(X, y, test_size=0.2) #C

from catboost import CatBoostClassifier
ens = CatBoostClassifier(iterations=5, depth=3, #D
                        cat_features=cat_features) #E
ens.fit(Xtrn, ytrn)
ypred = ens.predict(Xtst)
print('Model Accuracy using CATBOOST = {0:5.3f}%'.
      format(100 * accuracy_score(ypred, ytst)))
```

#A Load the data set as a pandas dataframe

#B Explicitly identify the categorical features

#C Prepare data for training & evaluation

#D Train an ensemble of 5 oblivious trees, each of depth 3

#E Make sure CatBoost knows which features are categorical

This listing trains and evaluates a CatBoost model:

```
Model Accuracy using CATBOOST = 83.333%
```

### CROSS VALIDATION WITH CATBOOST

CatBoost provides support for many different loss functions for regression and classification tasks, and many features to control various aspects of training.

This includes hyperparameters to control overfitting by controlling the complexity of the ensemble (`iterations`, with one tree trained per iteration) and the complexity of the base estimators (`depth` of the oblivious decision trees).

In addition to these, another key parameter is the `learning_rate`. Recall that the learning rate allows greater control over how quickly the complexity of the ensemble grows. Therefore, identifying an “optimal” learning rate for our data set in practice can help avoid overfitting and generalize well after training.

As with previous ensemble approaches, we will use 5-fold cross validation to search over several different hyper-parameter combinations to identify the best model. The listing below illustrates how to perform cross validation with CatBoost

#### Listing 8.2. Cross Validation with CatBoost

```
params = {'depth': [1, 3],
          'iterations': [5, 10, 15],
          'learning_rate': [0.01, 0.1]} #A

ens = CatBoostClassifier(cat_features=cat_features) #B
grid_search = ens.grid_search(params, Xtrn, ytrn, #C
                              cv=5, refit=True) #D

print('Best parameters: ', grid_search['params'])
ypred = ens.predict(Xtst)
print('Model Accuracy using CATBOOST = {0:5.3f}%'.
      format(100 * accuracy_score(ypred, ytst)))
```

**#A Create a grid of possible parameter combinations**

**#B Explicitly identify the categorical features**

**#C Use CatBoost's built-in grid search functionality**

**#D Perform 5-fold cross validation and then refit a model using the best parameters identified after grid search**

This listing evaluates the ( $2 \times 3 \times 2 = 12$ ) parameter combinations specified in `params` using 5-fold cross validation to identify the best parameter combination and (re)fits trains a final model with them.

```
Best parameters: {'depth': 3, 'iterations': 15, 'learning_rate': 0.1}
Model Accuracy using CATBOOST = 82.609%
```

### EARLY STOPPING WITH CATBOOST

As with other ensemble methods, with each successive iteration, CatBoost adds a new base estimator to the ensemble. This causes the complexity of overall ensemble to steadily increase during training until the model begins to overfit the training data.

As with other ensemble methods, it is possible to employ early stopping with CatBoost, where we monitor the performance of CatBoost with the help of an evaluation set to stop training as soon as there is no significant improvement in performance.

In the listing below, we initialize CatBoost to train 100 trees. As we will see below, with early stopping, it is possible to terminate training early, thus ensuring a good model as well as training efficiency.

### Listing 8.3. Early Stopping with CatBoost

```
ens = CatBoostClassifier(iterations=100, depth=3, #A
                        cat_features=cat_features,
                        loss_function='Logloss')

from catboost import Pool
eval_set = Pool(Xtst, ytst, cat_features=cat_features) #B

ens.fit(Xtrn, ytrn, eval_set=eval_set,
        early_stopping_rounds=5, #C
        verbose=False, plot=True) #D

ypred = ens.predict(Xtst)
print('Model Accuracy using CATBOOST = {0:5.3f}%'.
      format(100 * accuracy_score(ypred, ytst)))
```

**#A** Initialize a CatBoostClassifier with ensemble size 100

**#B** Create an evaluation set by pooling Xtst and ytst

**#C** Stop training if no improvement detected after 5 rounds

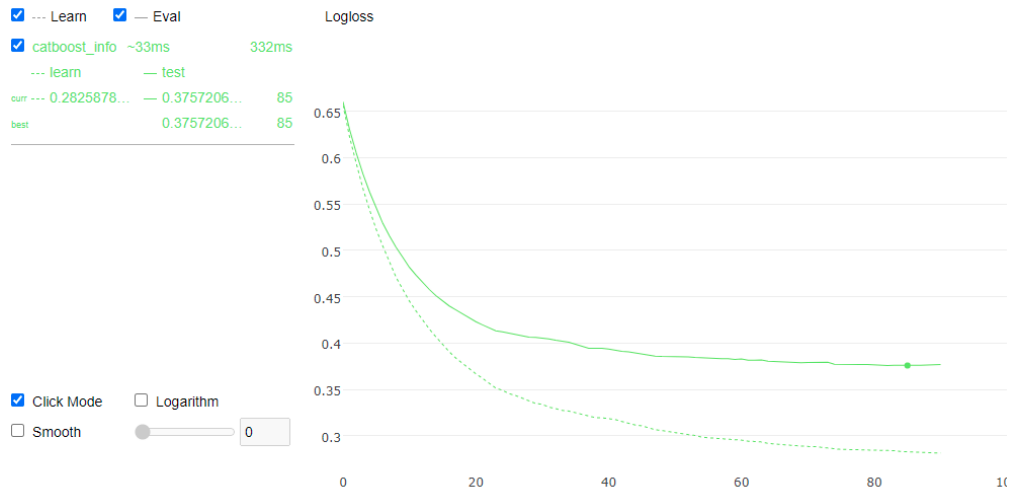
**#D** Set plotting to true for CatBoost to plot training and evaluation curves

This code generates training and curves as shown in Figure 8.11, where the effect of overfitting is observable. Around the 80-th iteration, the training curve (dashed) is continuing to decrease, while the evaluation curve has begun to flatten.

This means that the training error is continuing to decrease without an equivalent decrease in our validation set, indicating overfitting. CatBoost observes this behavior for 5 more iterations (as `early_stopping_rounds=5`), and then terminates training.

The final model reports a test set performance of 82.61%, achieved after 85 rounds, with early stopping avoiding training all the way to 100 iterations as originally specified.

```
Model Accuracy using CATBOOST = 82.609%
```



**Figure 8.11.** Training (dashed) and evaluation (solid) curves generated by CatBoost. The dot at the 85-th iteration indicates the early stopping point.

## 8.3 Case Study: Income Prediction

In this section, we study the problem of *income prediction* from demographic data. Demographic data typically contains many different types of features, including categorical and continuous features.

We will explore two approaches to training ensemble methods

- Approach 1 (Sections 8.3.2 and 8.3.3): *Pre-process* categorical features using the `category_encoders` package and then train ensembles using `scikit-learn`'s Random Forest, LightGBM and XGBoost with the pre-preprocessed features.
- Approach 2 (Section 8.3.4): Use CatBoost to directly handle categorical features during training through ordered target statistics and ordered boosting.

### 8.3.1 The Adult Census Data Set

This case study uses the Adult Census dataset (originally created in 1995) from the UCI Repository. The task is to predict whether an individual will earn more or less than \$50,000 per year based on several demographic indicators such as education, marital status, race, and gender.

This data set contains a nice mix of categorical and continuous features, which makes it an ideal choice for this case study. The data set is available along with the source code. Let's load the data set and visualize it.

```
import pandas as pd
df = pd.read_csv('./data/ch08/adult.csv')
df.head()
```

age	workclass	fnlwgt	education	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country	salary
50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	13	United-States	<=50K
38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0	0	40	United-States	<=50K
53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0	0	40	United-States	<=50K
28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0	0	40	Cuba	<=50K
37	Private	284582	Masters	14	Married-civ-spouse	Exec-managerial	Wife	White	Female	0	0	40	United-States	<=50K

**Figure 8.12.** The Adult Data set contains categorical and continuous features.

This data set contains several categorical features:

- `workclass`, which describes the classification of the type of employment and contains 8 categories: Private, Self-emp-not-inc, Self-emp-inc, Federal-gov, Local-gov, State-gov, Without-pay, Never-worked
- `education`, which describes the highest education level attained and contains 16 categories: Bachelors, Some-college, 11th, HS-grad, Prof-school, Assoc-acdm, Assoc-voc, 9th, 7th-8th, 12th, Masters, 1st-4th, 10th, Doctorate, 5th-6th, Preschool
- `marital-status`, which has 7 categories: Married-civ-spouse, Divorced, Never-married, Separated, Widowed, Married-spouse-absent, Married-AF-spouse
- `occupation`, which describes the classification of the occupation area and contains 15 categories: Tech-support, Craft-repair, Other-service, Sales, Exec-managerial, Prof-specialty, Handlers-cleaners, Machine-op-inspct, Adm-clerical, Farming-fishing, Transport-moving, Priv-house-serv, Protective-serv, Armed-Forces
- `relationship` status: Wife, Own-child, Husband, Not-in-family, Other-relative, Unmarried,
- `sex`: male or female,
- `native-country`, which is a high(ish)-cardinality categorical variable that contains 30 unique countries in this data set

In addition, the data set also contains several continuous features such as age, number of years of education, number of hours worked per week, capital gains and losses etc.

In the listing below, we explore some of the categorical features using the `seaborn` package, which provide some neat functions for quickly exploring and visualizing data sets.

**Listing 8.4. Visualize some categorical features in the Adult Census data set**

```
import matplotlib.pyplot as plt
import seaborn as sns

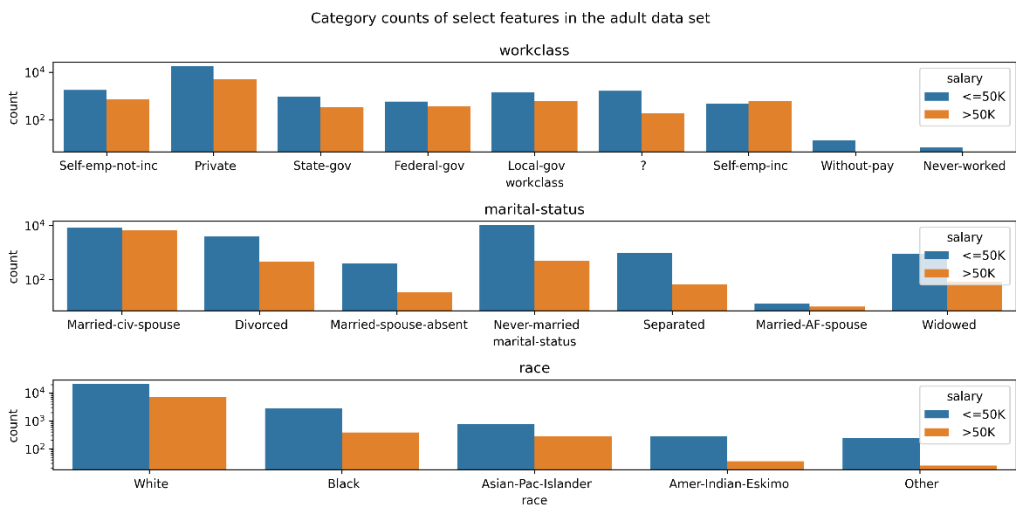
fig, ax = plt.subplots(nrows=3, ncols=1, figsize=((12, 6)))
fig.suptitle('Category counts of select features in the adult data set')

sns.countplot(x='workclass', hue='salary', data=df, ax=ax[0])
ax[0].set(yscale='log')

sns.countplot(x='marital-status', hue='salary', data=df, ax=ax[1])
ax[1].set(yscale='log')

sns.countplot(x='race', hue='salary', data=df, ax=ax[2])
ax[2].set(yscale='log')
fig.tight_layout()
```

This listing produces the following output.



**Figure 8.13.** Visualizing the category counts of three categorical features in the Adult Census data set: workclass, marital-status, race. Note that the y-axis is in log scale.

### 8.3.2 Creating Preprocessing and Modeling Pipelines

The listing below describes how to prepare the data. In particular, we use `sklearn.preprocessing.LabelEncoder` to convert the target labels from string (<=50k, >50k) to numeric (0/1). `LabelEncoder` is identical to `OrdinalEncoder`, except that it is specifically designed to work with 1D data (targets).



**Listing 8.5. Prepare the Adult Census Data**

```
X, y = df.drop('salary', axis=1), df['salary']    #A

from sklearn.preprocessing import LabelEncoder
y = LabelEncoder().fit_transform(y)            #B

from sklearn.model_selection import train_test_split
Xtrn, Xtst, ytrn, ytst = train_test_split(X, y, test_size=0.2)    #C

features = X.columns
cat_features = ['workclass', 'education', 'marital-status',
                'occupation', 'relationship', 'race', 'sex',
                'native-country']              #D
cont_features = features.drop(cat_features).tolist()
```

**#A Split the data into features and targets**

**#B Encode the labels**

**#C Split into train and test sets**

**#D Explicitly identify categorical and continuous features**

Recall that the task is to predict if the income is greater than \$50,000 (with labels  $y=1$ ) or less than \$50,000 (with labels  $y=0$ ). One thing to note about this data set is that it is *imbalanced*, that is, it contains different proportions of the two classes:

```
import numpy as np
n_pos, n_neg = np.sum(y > 0)/len(y), np.sum(y <= 0)/len(y)
print(n_pos, n_neg)
0.24081695331695332  0.7591830466830467
```

Here, we see that the positive-negative distribution is 24.1% to 75.9%, rather than 50% to 50%. This means that evaluation metrics such as accuracy can unintentionally skew our view of model performance as they assume a balanced data set.

Next, we define a preprocessing function that can be reused with different types of category encoders. This function has two preprocessing pipelines, one to be applied to continuous features only, and the other for categorical features. The continuous features are preprocessed using `StandardScaler`, which normalizes each feature column to have zero mean and unit standard deviation.

In addition, both pipelines have a `SimpleImputer`, to impute missing values. Missing continuous values are imputed with their corresponding median feature value, while missing categorical features are imputed as a new category called `'missing'` prior to encoding.

For example, the feature `workclass` has missing values (indicated by `'?'` in Figure 8.12), which are treated as a separate category for modeling purposes.

**Listing 8.6. Preprocessing pipelines**

```

from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer

import category_encoders as ce

def create_preprocessor(encoder):
    preprocess_continuous = Pipeline(steps=[      #A
('impute_missing', SimpleImputer(strategy='median')),
('normalize', StandardScaler())])
    preprocess_categorical = Pipeline(steps=[      #B
('impute_missing', SimpleImputer(strategy='constant',
    fill_value='missing')),
    ('encode', encoder())])

    transformations = ColumnTransformer(transformers=[      #C
('continuous', preprocess_continuous, cont_features),
('categorical', preprocess_categorical,
    cat_features)])

    return transformations

```

**#A preprocessing pipeline for continuous features**

**#B preprocessing pipeline for categorical features**

**#C ColumnTransformer object is used to combine the pipelines**

This listing will create and return a `scikit-learn` `ColumnTransformer` object, which can apply a similar pre-processing strategy to training and test sets, ensuring consistency and minimizing data leakage.

Finally, we define a function to train and evaluate different types of ensembles, combining them with different types of category encoding. This will enable us to create different ensemble models by combining different ensemble learning packages with different types of category encoders.

The function below allows us to pass an `ensemble` as well as a grid of `ensemble parameters` for ensemble parameter selection. It uses n-fold cross validation combined with randomized search to identify the best ensemble parameters before training a final model with these best parameters.

Once trained, the function evaluates final model performance on the test set using three metrics: accuracy, balanced accuracy and F1 score.

Balanced accuracy and F1 score are especially useful metrics when the data set is imbalanced, as they take label imbalance into account by weighting model performance on each class based on how often they appear in the labels.

**Listing 8.7. Training and evaluating combinations of encoders and ensembles**

```

from sklearn.model_selection import RandomizedSearchCV
from sklearn.metrics import accuracy_score, f1_score,
    balanced_accuracy_score

def train_and_evaluate_models(ensemble, parameters,    #A
                              n_iter=25,           #B
                              cv=5):              #C
    results = pd.DataFrame()

    for encoder in [ce.OneHotEncoder,
                   ce.OrdinalEncoder,
                   ce.TargetEncoder]:             #D
        preprocess_pipeline = create_preprocessor(encoder)    #E

        model = Pipeline(steps=[
            ('preprocess', preprocess_pipeline),
            ('crossvalidate',
             RandomizedSearchCV(
                 ensemble, parameters,
                 n_iter=n_iter, cv=cv,          #F
                 refit=True,                   #G
                 verbose=2))])

        model.fit(Xtrn, ytrn)

        ypred_trn = model.predict(Xtrn)
        ypred_tst = model.predict(Xtst)

        res = {'Encoder': encoder.__name__,        #H
              'Ensemble': ensemble.__class__.__name__,
              'Train Acc': accuracy_score(ytrn, ypred_trn),
              'Train B Acc': balanced_accuracy_score(ytrn,
                                                      ypred_trn),
              'Train F1': f1_score(ytrn, ypred_trn),
              'Test Acc': accuracy_score(ytst, ypred_tst),
              'Test B Acc': balanced_accuracy_score(ytst,
                                                      ypred_tst),
              'Test F1': f1_score(ytst, ypred_tst)}
        results = results.append(res, ignore_index=True)

    return results

```

**#A** specify ensemble and parameter grid

**#B** maximum number of parameter combinations for randomized grid search

**#C** number of cross validation folds for parameter selection

**#D** different categorical encoding strategies to try

**#E** initialize preprocessor pipeline (Listing 8.6)

**#F** parameter selection using randomized grid search

**#G** refit a final ensemble using the best parameters

**#H** evaluate final ensemble performance and save the results

### 8.3.3 Category Encoding and Ensembling

In this Section, we will train various combinations of encoders and ensemble methods. In particular, we consider

- encoders: one hot, ordinal, and greedy target encoding (from the `category_encoders` package)
- ensembles: scikit-learn's random forest, gradient boosting with LightGBM and Newton boosting with XGBoost

For each combination of encoder and ensemble, we follow the same steps implemented in Listings 8.6 and 8.7: pre-process the features, perform ensemble parameter selection to get best ensemble parameters, refit a final ensemble model with the best parameter combination and, finally, evaluate the final model.

### RANDOM FOREST

The listing below trains and evaluates the best combination of categorical encoding (one hot, ordinal, and greedy target) and random forest.

#### Listing 8.8. Category encoding followed by ensembling with random forest

```
from sklearn.ensemble import RandomForestClassifier

ensemble = RandomForestClassifier(n_jobs=-1)
parameters = {'n_estimators': [25, 50, 100, 200],      #A
              'max_depth': [3, 5, 7, 10],           #B
              'max_features': [0.2, 0.4, 0.6, 0.8]}   #C

rf_results = train_and_evaluate_models(ensemble, parameters,
                                       n_iter=25, cv=5) #D
```

#A number of trees in the random forest ensemble

#B maximum depth of individual trees in the ensemble

#C fraction of features/columns during tree learning

#D randomized grid search with 25 parameter combinations and 5-fold cross validation

This listing returns of results, shown below (edited to fit into the page).

Encoder	Test Acc	Test B Acc	Test F1	Train Acc	Train B Acc	Train F1
OneHot	0.862	0.766	0.669	0.875	0.783	0.7
Ordinal	0.861	0.756	0.657	0.874	0.773	0.688
Target	0.864	0.774	0.679	0.881	0.797	0.72

Observe the difference between plain accuracy (`Acc`) and balanced accuracy (`B Acc`) or F1-score (`F1`) for both the training and test sets. This illustrates the importance of using the right metric to evaluate our models.

While all encoding methods appear equally effective using plain accuracy as the evaluation metric, encoding with target statistics seems to be most effective in classifying between the positive and negative examples.

### LIGHTGBM

Next, we repeat this training and evaluation procedure with LightGBM, where we train an ensemble with 200 trees. Several other ensemble hyperparameters will be selected using 5-fold cross validation: maximum tree depth, learning rate, bagging fraction and regularization parameters.

**Listing 8.9. Category encoding followed by ensembling with LightGBM**

```

from lightgbm import LGBMClassifier

ensemble = LGBMClassifier(n_estimators=200, n_jobs=-1)
parameters = {'max_depth': np.arange(3, 10, step=1),      #A
              'learning_rate': 2**np.arange(-8., 2., step=2),  #B
              'bagging_fraction': [0.4, 0.5, 0.6, 0.7, 0.8],  #C
              'lambda_l1': [0, 0.01, 0.1, 1, 10],          #D
              'lambda_l2': [0, 0.01, 0.1, 1e-1, 1, 10]}

lgbm_results = train_and_evaluate_models(ensemble, parameters,
                                         n_iter=50, cv=5)

```

**#A** maximum depth of individual trees in the ensemble

**#B** learning rate for gradient boosting

**#C** fraction of examples used during tree learning

**#D** parameters for weight regularization

This listing returns of results, shown below (edited to fit into the page).

Encoder	Test Acc	Test B Acc	Test F1	Train Acc	Train B Acc	Train F1
OneHot	0.874	0.802	0.716	0.891	0.824	0.754
Ordinal	0.874	0.802	0.717	0.892	0.825	0.757
Target	0.873	0.796	0.71	0.886	0.815	0.741

With LightGBM, all three encoding methods lead to ensembles with roughly similar generalization performance as evidenced by test balanced accuracy and F1 scores. The overall performance is also better than random forest.

**XGBoost**

Finally, we repeat this training and evaluation procedure with XGBoost as well, where we again train an ensemble of 200 trees.

**Listing 8.10. Category encoding followed by ensembling with XGBoost**

```

from xgboost import XGBClassifier

ensemble = XGBClassifier(n_estimators=200, n_jobs=-1)
parameters = {'max_depth': np.arange(3, 10, step=1),      #A
              'learning_rate': 2**np.arange(-8., 2., step=2),  #B
              'colsample_bytree': [0.4, 0.5, 0.6, 0.7, 0.8],  #C
              'reg_alpha': [0, 0.01, 0.1, 1, 10],          #D
              'reg_lambda': [0, 0.01, 0.1, 1e-1, 1, 10]}

xgb_results = train_and_evaluate_models(ensemble, parameters,
                                         n_iter=50, cv=5)

```

**#A** maximum depth of individual trees in the ensemble

**#B** learning rate for Newton boosting

**#C** fraction of features/columns during tree learning

**#D** parameters for weight regularization

This listing returns of results, shown below (edited to fit into the page).

Encoder	Test Acc	Test B Acc	Test F1	Train Acc	Train B Acc	Train F1
OneHot	0.875	0.799	0.715	0.896	0.829	0.764
Ordinal	0.873	0.799	0.712	0.891	0.823	0.753
Target	0.875	0.802	0.717	0.898	0.834	0.771

As with LightGBM, all three encoding methods lead to XGBoost ensembles with roughly similar generalization performance. The overall performance of XGBoost is similar to that of LightGBM, but better than Random Forest.

### 8.3.4 Ordered Encoding and Boosting with CatBoost

Finally, we explore the performance of CatBoost on this data set. Unlike the previous approaches in Section 8.3.4, we will *not* use the `category_encoders` package. This is because, CatBoost uses ordered target statistics along with ordered boosting.

This means that, as long as clearly identify the categorical features that need encoding with ordered target statistics, CatBoost will take care of the rest and we do not need to incorporate any additional pre-processing!

#### List 8.11. Ordered target encoding and ordered boosting with CatBoost

```
from catboost import CatBoostClassifier

ensemble = CatBoostClassifier(cat_features=cat_features)
parameters = {'iterations': [25, 50, 100, 200], #A
              'depth': np.arange(3, 10, step=1), #B
              'learning_rate': 2*np.arange(-5., 0., step=1), #C
              'l2_leaf_reg': [0, 0.01, 0.1, 1e-1, 1, 10]} #D

search = ensemble.randomized_search(parameters, Xtrn, ytrn,
                                    n_iter=50, cv=5, refit=True,
                                    verbose=False) #E

ypred_trn = ensemble.predict(Xtrn)
ypred_tst = ensemble.predict(Xtst)

res = {'Encoder': '',
       'Ensemble': ensemble.__class__.__name__,
       'Train Acc': accuracy_score(ytrn, ypred_trn),
       'Train B Acc': balanced_accuracy_score(ytrn, ypred_trn),
       'Train F1': f1_score(ytrn, ypred_trn),
       'Test Acc': accuracy_score(ytst, ypred_tst),
       'Test B Acc': balanced_accuracy_score(ytst, ypred_tst),
       'Test F1': f1_score(ytst, ypred_tst)}

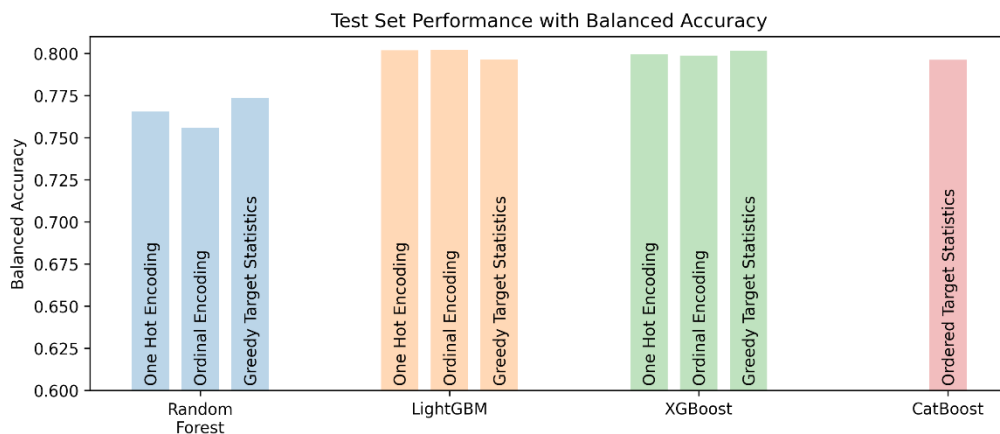
cat_results = pd.DataFrame()
cat_results = cat_results.append(res, ignore_index=True)
```

**#A** number of trees in the random forest ensemble  
**#B** maximum depth of individual trees in the ensemble  
**#C** learning rate for Newton boosting  
**#D** parameters for weight regularization  
**#E** use CatBoost's randomized search functionality

CatBoost provides its own `randomized_search` feature, which can be initialized and invoked similarly to scikit-learn's `RandomizedGridCV`, which we used in the previous section.

Ensemble	Test Acc	Test B Acc	Test F1	Train Acc	Train B Acc	Train F1
CatBoost	0.87	0.796	0.708	0.888	0.82	0.747

CatBoost's performance on this data set is comparable to that of LightGBM and XGBoost, and better than Random Forest.



**Figure 8.14.** The test set performance (with the balanced accuracy metric) of various encoding and ensemble method combinations.

Now, let's put the results of all the approaches side-by-side; in the figure above, we look at how each approach performed with respect to balanced accuracy evaluated on the test set.

In analyzing these results, keep in mind, always, that there is no free lunch, and no method is best performing all the time. However, CatBoost does enjoy two key benefits:

- it allows for a consolidated approach to encoding and handling categorical features, unlike other ensemble approaches which necessarily use a two-step encode+ensemble approach, and
- by design, it mitigates data and target leakage and distribution shift issues, which often need more care with other ensembling approaches.

## 8.4 Encoding High-Cardinality String Features

We wrap up this chapter by exploring encoding techniques for *high-cardinality categorical features*. The cardinality of a categorical feature is simply the number of unique categories in it. The number of categories is an important consideration in categorical encoding.

Real-world data sets often contain categorical string features, where feature values are strings. For example, consider a categorical feature of job titles at an organization. This feature can contain dozens to hundreds of job titles from 'Intern' to 'President and CEO', each with their own unique roles and responsibilities.

Such features contain a large number of categories and are inherently high-cardinality. This disqualifies encoding approaches such as one-hot encoding (because it increases feature dimension significantly), or ordinal encoding (because no natural ordering typically exists).

What's more, in real-world data sets, such high-cardinality are also 'dirty', in that there are several variations of the same category:

- Natural variations can arise because data is compiled from different sources. For example, two departments in the same organization may have different titles for the exact same role: "Lead Data Scientist" and "Senior Data Scientist".
- Many such data sets are manually entered into databases, which introduces noise due to typos and other errors. For example, "Data Sciencsit" [sic] and "Data Scientist".

Because two (or more!) such variants do not match exactly, they are treated as their own unique categories, even though common sense suggests that they should be cleaned and/or merged. This causes additional problems with high-cardinality string features by adding new categories to an already large set of categories.

To address this issue, we will need to determine categories (and how to encode them) by *string similarity* rather than by exact matching! The intuition behind this approach is to encode similar categories together in a way that a human might, to ensure that the downstream learning algorithm treats them similarly (as it should).

For example, similarity-based encoding would encode "Data Sciencsit" [sic] and "Data Scientist" with similar features so that they appear nearly identical to a learning algorithm. Similarity-based encoding methods use the notion of *string similarity* to identify similar categories.

Such *string similarity metrics*, or measures, are widely used in natural language and text applications, for example, in autocorrect applications, database retrieval or in language translation.



## String similarity metrics

A similarity metric is a function that takes two objects and returns a numeric similarity measure between them. Higher values mean that the two objects are more similar to each other. A string similarity metric operates on strings.

Measuring the similarity between strings is challenging as strings can be of different lengths and can have similar substrings in different locations. To identify if two strings are similar potentially requires matching characters and subsequences of all possible lengths and locations. This combinatorial complexity means that computing string similarity can be computationally expensive.

Several approaches to computing string similarity between strings of different lengths exist. Two common types are *character-based* string similarity and *token-based* string similarity, depending on the *granularity* of the string components being compared.

Character-based approaches measure string similarity by the number of operations at the character-level (insertion, deletion, or substitution) needed to transform one string to another. These approaches are well-suited for short strings.

Longer strings are often decomposed into tokens, typically substrings or words, called n-grams. Token-based approaches measure string similarity at the token-level.

Irrespective of which string similarity metric you use, the similarity score can be used to encode both high-cardinality features (by grouping similar string categories together) and dirty features (by ‘cleaning’ typos).

## THE DIRTY-CAT PACKAGE

The `dirty-cat`<sup>3</sup> package provides such functionality off-the-shelf and can be used seamlessly in modeling pipelines. The package provides three specialized encoders to handle so called “dirty categories”, which are essentially noisy and/or high-cardinality string categories.

- `SimilarityEncoder`, a version of one-hot encoding constructed using string similarities,
- `GapEncoder`, that encodes categories by considering frequently co-occurring substring combinations, and
- `MinHashEncoder`, that encodes categories by applying hashing techniques to substrings.

We use another salary data set to see how we can use the `dirty_cat` package in practice. This data set is a modified version of a publicly-available employee salaries data set from data.gov, with the goal being to predict an individual’s salary given their job title and department.

First, we load the data set and visualize the first few rows:

<sup>3</sup> <https://dirty-cat.github.io/stable/index.html>

```
import pandas as pd
df = pd.read_csv('./data/ch08/employee_salaries.csv')
df.head()
```

gender	department_name	assignment_category	employee_position_title	underfilled_job_title	year_first_hired	salary
F	Department of Environmental Protection	Fulltime-Regular	Program Specialist II	NaN	2013	75362.93
F	Department of Recreation	Fulltime-Regular	Recreation Supervisor	NaN	1997	79522.62
F	Department of Transportation	Fulltime-Regular	Bus Operator	NaN	2014	42053.83
M	Fire and Rescue Services	Fulltime-Regular	Fire/Rescue Captain	NaN	1995	114587.02
F	Department of Public Libraries	Fulltime-Regular	Library Assistant I	NaN	1996	55139.67

**Figure 8.15.** The Employee Salaries data set mostly contains string categories.

The column `salary` is the target variable, making this a regression problem. We split this dataframe into features and labels.

```
X, y = df.drop('salary', axis=1), df['salary']
print(X.shape)
(9211, 6)
```

We can get a sense of which features are high-cardinality by counting the number of unique categories or values per column.

```
for col in X.columns:
    print('{0}: {1} categories'.format(col, df[col].nunique()))

gender: 2 categories
department_name: 37 categories
assignment_category: 2 categories
employee_position_title: 385 categories
underfilled_job_title: 83 categories
year_first_hired: 51 categories
```

We see that the feature `employee_position_title` has 385 unique string categories, making this an instance of a high-cardinality feature. Directly encoding this using one-hot encoding, say, would introduce 385 new columns into our data set, thus increasing the number of columns greatly!

Instead, let's see how we can use the `dirty-cat` package to train an XGBoost ensemble on this data set. First, let's identify the different types of features in our data set explicitly:

```
lo_card = ['gender', 'department_name', 'assignment_category']
hi_card = ['employee_position_title']
continuous = ['year_first_hired']
```

Next, let's initialize the different `dirty-cat` encoders we want to use:

```
encoders = [SimilarityEncoder(similarity='ngram'), #A
             MinHashEncoder(n_components=100), #B
             GapEncoder(n_components=100)]
```

**#A** specify the string similarity measure to use

**#B** encoding dimension

The most important parameter in `SimilarityEncoder` is the `similarity` metric, which compares two strings and returns a value indicating how similar they are. The most important parameter for `MinHashEncoder` and `GapEncoder` is `n_components`, which is also known as the *encoding dimension*.

`SimilarityEncoder` accepts four different types of string similarity metrics: `ngram` (token-based), `levenshtein-ratio`, `jaro` and `jaro-winkler` (character-based). Character-based string similarity metrics are better suited for short strings with a few words at most, while token-based metrics are better suited for longer strings.

To understand encoding dimension, consider that we are one-hot encoding the feature `employee_position_title`, which contains 385 unique categories. This encoding converts each categorical value to a 385-dimensional vector, making the encoding dimension 385.

`MinHashEncoder` and `GapEncoder`, on the other hand, can take in a user-specified encoding dimension, and create an encoding of the specified size. In this case, the encoding dimension is specified to be 100 for both, which is much smaller than what one-hot encoding would be forced to use.

Practically, the encoding dimension (`n_components`) is a modeling choice, and the best value should be determined through k-fold cross validation depending on the tradeoff between model training time vs. model performance.

We put all this together into the listing below, which trains three different XGBoost models, one for each type of `dirty-cat` encoding.

**Listing 8.12. Encoding and ensembling with high-cardinality features**

```

from sklearn.preprocessing import OneHotEncoder, MinMaxScaler
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from dirty_cat import SimilarityEncoder, MinHashEncoder, GapEncoder
from xgboost import XGBRegressor
from sklearn.metrics import r2_score

lo_card = ['gender', 'department_name', 'assignment_category'] #A
hi_card = ['employee_position_title'] #B
continuous = ['year_first_hired'] #C

encoders = [SimilarityEncoder(similarity='ngram'),
             MinHashEncoder(n_components=100),
             GapEncoder(n_components=100)]

for encoder in encoders:
    ensemble = XGBRegressor(objective='reg:mse', learning_rate=0.1,
                           n_estimators=100, max_depth=3) #D

    preprocess = ColumnTransformer(
        transformers=[
            ('continuous', MinMaxScaler(), continuous), #E
            ('onehot', OneHotEncoder(sparse=False), lo_card), #F
            ('dirty', encoder, hi_card)], #G
        remainder='drop')

    pipe = Pipeline(steps=[('preprocess', preprocess),
                           ('train', ensemble)]) #H
    pipe.fit(Xtrn, ytrn)

    ypred = pipe.predict(Xtst)
    print('{0}: {1}'.format(encoder.__class__.__name__,
                            r2_score(ytst, ypred))) #I

```

```

#A identify low-cardinality features
#B identify high-cardinality features
#C identify continuous features
#D use XGBoost as the ensemble method
#E rescale continuous features to [0, 1] range
#F one-hot encode the low-cardinality features
#G encode high-cardinality features using dirty-cat encoding
#H create a pre-processing and training pipeline
#I use R2 score to evaluate overall performance

```

In this example, we identify three different types of features, each of which we pre-process differently:

- low-cardinality features such as `gender` (2 categories) and `department_name` (37 categories) are one-hot encoded,
- high-cardinality features such as `employee_position_title` are encoded using `dirty_cat` encoders, and
- continuous features such as `year_first_hired` are rescaled using `MinMaxScaler` to be in the range 0 to 1.

After encoding, we train an XGBoost regressor with 100 trees each of maximum depth 3 using the fairly standard mean-squared-error loss function. The trained models are evaluated using the regression metric R2 score (see Chapter 1 for details), which ranges from  $-\infty$  to 1, with values closer to 1 indicating better-performing regressors.

```
SimilarityEncoder: 0.8406447781562516
MinHashEncoder: 0.8162021132963543
GapEncoder: 0.8087771002785784
```

In this simple use case, similarity encoding combined with XGBoost works best, compared to XGBoost models trained using the `GapEncoder` or the `MinHashEncoder`. As with the other supervised methods, it is often necessary to use cross validation determine which encoding parameters produce the best results for the data set at hand.

## 8.5 Summary

- A categorical feature is a type of data attribute that takes discrete values called classes or categories. For this reason, categorical features are also called discrete features.
- A nominal feature is a categorical variable whose values have no relationship between them (e.g., cat, dog, pig, cow)
- An ordinal feature is a categorical variable whose values are ordered, either increasing or decreasing (e.g, freshman, sophomore, junior, senior)
- One-hot vectorization/encoding and ordinal encoding are commonly used unsupervised encoding methods.
- One-hot encoding introduces a binary (0-1) columns for each category into the data set and can be inefficient when a feature has a large number of categories. Ordinal encoding introduces integer values sequentially for each category.
- Target statistics (TS) are a supervised encoding approach for categorical features; rather than a predetermined or learned encoding step, categorical features are replaced with a statistic that describes the category (such as mean)
- Greedy target statistics use all the training data for encoding which leads to issues of train-to-test target leakage and distribution shift, which affects how we evaluate model generalization performance.
- Hold-out target statistics uses a special hold-out encoding set in addition to a hold-out test set; it eliminates leakage and shift but is wasteful of data.
- Leave-one-out target statistics and ordered target statistics are data-efficient ways to mitigate leakage and shift.
- Gradient boosting techniques use training data for both residual computation *and* model training, which causes a prediction shift and overfitting.
- Ordered boosting is a modification of Newton boosting that uses a permutation-based approach to ensembling to further reduce prediction shift. Ordered boosting tackles prediction shift by training a sequence of models on different permutations and subsets of the data.
- CatBoost is a publicly available boosting library that implements ordered target statistics *and* ordered boosting.
- While CatBoost is well-suited for categorical features, it can also be applied to regular

features.

- CatBoost uses oblivious decision trees as weak learners. Oblivious decision trees use the same splitting criterion in all the nodes across an entire level/depth of the tree. Oblivious trees are balanced, less prone to overfitting, and allow speeding up execution at testing time significantly
- High-cardinality features contain many, many unique categories; one-hot encoding high-cardinality features can introduce a large number of new data columns, most of them sparse (with many zeros), which leads to inefficient learning
- dirty-cat is a package that produces more compact encodings for discrete-valued features and uses string and substring similarity and hashing to create effective encodings.

## 9

## Explaining Your Ensembles

### This chapter covers

- Understanding glass-box vs. black-box and global vs. local interpretability
- Reviewing the basics of glass-box interpretability of decision trees and generalized linear models
- Using global black-box methods to understand the behavior of pre-trained ensembles
- Using local black-box methods to explain predictions of pre-trained ensembles
- Training and using explainable global & local glass-box ensembles from scratch

When training and deploying models, we are usually concerned about *WHAT the model prediction* is. Equally important, however, is *WHY the model made the prediction that it did*.

Understanding a model's predictions is a critical component of building robust machine-learning pipelines. This is especially true when machine-learning models are used in high-stakes applications such as in healthcare or finance.

For example, in a medical diagnosis task such as diabetes diagnosis, understanding why the model made a specific diagnosis can provide users (in this case, doctors) with additional insights that can guide them towards better prescriptions, preventative or palliative care.

This increased *transparency*, in turn, increases *trust* in the machine-learning system, and allows the users for whom the models have been developed to use them with confidence.

Understanding the reasons behind a model's predictions is also extremely useful in model debugging, identification of failure cases and in finding ways to improve model performance. Furthermore, model debugging can also help pinpoint biases and problems with the data itself.

Machine-learning models are typically characterized as black-box models and glass-box models. Blackbox models are typically challenging to understand owing to their complexity (for example, deep neural networks). The predictions of such models require specialized tools to be *explainable*. Many of the ensembles covered in this book, such as random forests and gradient boosting are black-box machine-learning models.

Glass-box models are more intuitive and easier to understand (for example, decision trees). The structure of such models makes them inherently *interpretable*. In this chapter, we explore the concepts of explainability and interpretability from the perspective of ensemble methods.

Interpretability methods are also characterized as global or local. Global methods attempt to broadly explain a model's features and relevance to decision-making across different types of examples. Local methods attempt to specifically explain a model's decision-making process with respect to individual examples and predictions.

Section 9.1 introduces the basics of black-box and glass-box machine-learning models. This section also reintroduces two well-known machine learning models from the perspective of interpretability: decision trees and generalized linear models.

Section 9.2 introduces this chapter's case study: data-driven marketing. This application is used in the rest of the section to illustrate techniques for interpretability and explainability.

Section 9.3 introduces three techniques for global black-box explainability: permutation feature importance, partial dependence plots and global surrogate models. Section 9.4 introduces two methods for local black-box explainability: *LIME* and *SHAP*.

The black-box methods introduced in Sections 9.3 and 9.4 are model-agnostic, that is, can be used for any machine-learning black-box. In these sections, we specifically focus on how they can be used for ensemble methods.

Section 9.5 introduces a glass-box method called *explainable boosting machines*, a new ensemble method that is designed to directly interpretable and provides *both* global and local interpretability.

## 9.1 What is Interpretability?

We first introduce the basics of interpretability and explainability for machine-learning models generally, before moving to how these concepts apply to ensemble methods specifically.

The notions of interpretability and explainability of a machine-learning model are related to its *structure* (e.g., is it a tree, a network, a linear model?) and its *parameters* (e.g., split and leaf values in trees, layer weights in neural networks, feature weights in linear models).

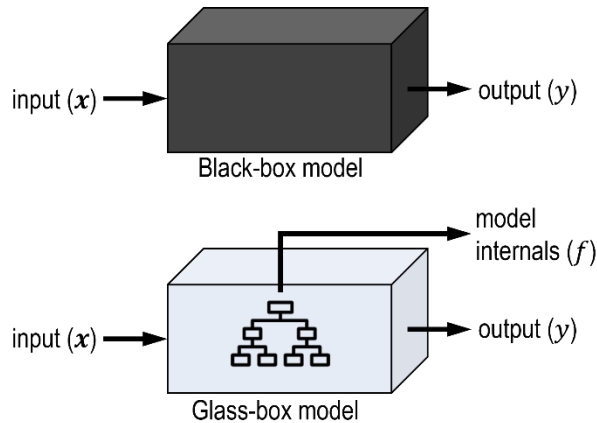
Our goal is to understand a model's behavior in terms of its input features, output predictions and the *model internals* (that is, structure and parameters).

### 9.1.1 Black-Box vs. Glass-Box Models

*Black-box machine-learning models* are models that are difficult to describe in terms of their model internals. This can be because we do not have access to the internal model structure and parameters (for example, if it was trained by someone else).

Even in cases where we do have access to the model internals, the model itself may be sufficiently complex that it is not easy to analyze and establish an intuitive understanding of the relationship between its inputs and outputs (see Figure 9.1).





**Figure 9.1.** With black-box machine-learning models, we can only use the input-output pairs to analyze and explain model behavior. The model internals in a black box are either unavailable or are not directly interpretable. With glass-box machine-learning models, in addition to input-output pairs, the model internals are also intuitively interpretable.

Neural networks and deep learning models are often cited as examples of black-box models, owing to the considerable complexity arising from their multilayered structure and large number of network parameters.

These models essentially function as black boxes: given an input example, they provide a prediction, but their inner workings are opaque to us. This makes interpreting model behavior pretty hard.

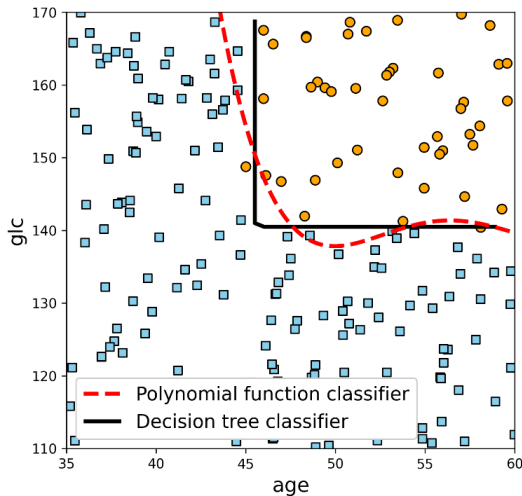
Many of the ensemble methods we've seen so far: random forests, AdaBoost, gradient and Newton boosting are all effectively black-box models to us. This is because, even though the individual base estimators themselves may be intuitive and interpretable, the process of ensembling introduces complex interactions between the features, which in turn, makes it hard to interpret the ensemble and its predictions.

Black-box models typically require *black-box explainers*, which are explanation models that aim to explain model behavior using only a model's inputs and outputs, but not its internals.

*Glass-box machine-learning models*, on the other hand, are easier to understand. This is often because their model structures are immediately intuitive or comprehensible to humans.

For example, consider a simple task of diabetes diagnosis from only two features: *age* and blood-glucose test result (*glc*). Let's say that we have learned two machine-learning models that have identical predictive performance, a 4-th degree polynomial classifier and a decision tree classifier

The data set for this example is shown in Figure 9.2, where patients who do not have diabetes (*class=-1*) are denoted by squares and patients who have diabetes (*class=+1*) are denoted by circles. The two classification models are also shown.



**Figure 9.2.** The problem space of diabetes patients who have to be classified as having diabetes (circles) or not having diabetes (squares) based on two features: *age* and *glc*. Two machine-learning models: a 4-th degree polynomial classifier and a decision-tree classifier are trained to have roughly similar predictive performance. However, the nature of their model internals (structure and parameters) means that decision lists are more intuitive for explanations and for understanding model behavior (see text).

The first model is a 4-th degree polynomial classifier. This classifier has an additive structure made up of weighted feature powers, and the weights are the model parameters:

$$f(\text{age}, \text{glc}) = \text{sign}(0.0021 \text{age}^4 - 0.497 \text{age}^3 + 41.734 \text{age}^2 - 1550.251 \text{age} + 21645.647 - \text{glc}).$$

This function returns either +1 (diabetes = TRUE) or -1 (diabetes = FALSE). Even with the full model available to us, given a new patient and resulting diagnostic prediction (say, diabetes = TRUE), it is not immediately clear why the model made the decision it did.

Was it because of the patient's age? Their blood-glucose test result? Both of these factors? This information is buried within complex mathematical calculations that are not easy for us to infer by simply looking at the model, its structure, and parameters.

Now, let's consider a second model, a decision tree with a single decision node of the form:

$$f(\text{age}, \text{glc}) = \begin{cases} \text{if age} > 45 \text{ AND } \text{glc} > 140, & +1, \\ \text{otherwise,} & -1. \end{cases}$$

This function also returns either +1 (diabetes = TRUE) or -1 (diabetes = FALSE). However, the structure of this decision tree is easily interpretable as:

```
if age > 45 AND glc > 140 then diabetes = TRUE else diabetes = FALSE.
```

This model's interpretation is pretty straightforward: any patient who is over the age of 45 and has a blood glucose test result of over 140 will be diagnosed as having diabetes.

In summary, even though the full model internals of the polynomial classifier are available to us, the model might as well be a black-box since the model internals are not intuitive or interpretable. On the other hand, the inherent nature of how the decision tree represents the knowledge it has learned allows for easier interpretation, making it a glass-box model.

In the rest of this section, we will explore two familiar machine-learning models that are also glass-box models: decision trees (and decision lists) and *generalized linear models* (GLMs). This will set us up to better understand the notions of interpretability and explainability for ensembles as both GLMs and decision trees are commonly used as base estimators in many different ensemble methods.

### 9.1.2 Decision Trees (and Decision Rules)

Decision trees are arguably the most interpretable of machine-learning models as they implement decision-making as a sequential process of asking and answering questions. The tree structure of a decision tree and its feature-based splitting functions are easy to interpret, as we will see below. This makes decision trees glass-box models.

Let's begin by training a decision tree on a classification data set called *Iris*. The task is a 3-way classification of irises into three species: *Iris setosa*, *Iris versicolour* and *Iris virginica* based on four features: sepal height, sepal width, petal height and petal width. This exceedingly simple data set only has 150 training examples and will serve as a good teaching example for the notions of visualization.

#### INTERPRETING DECISION TREES IN PRACTICE

Listing 9.1 loads the data set, trains a decision tree classifier, and visualizes it. Once visualized, we can interpret the learned decision tree model.

### Listing 9.1. Training and interpreting decision trees

```

from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
iris = load_iris() # A
Xtrn, Xtst, ytrn, ytst = train_test_split(iris.data, iris.target,
                                         test_size=0.15)

from sklearn import tree
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
model = DecisionTreeClassifier(min_samples_leaf=40, criterion='entropy') #B
model.fit(Xtrn, ytrn) # C
ypred = model.predict(Xtst)
print('Accuracy = {0:4.3}%'.format(accuracy_score(ytst, ypred) * 100))

import graphviz, re, pydotplus
dot = tree.export_graphviz(model, feature_names=iris.feature_names,
                           class_names=['Iris-Setosa',
                                         'Iris-Versicolour',
                                         'Iris-Virginica'],
                           filled=True, impurity=False)
graphviz.Source(dot, format="png") # D

```

#A Load the Iris data set and split into training and test sets

#B Use entropy as the criterion to measure quality of splits during learning

#C Train a decision tree classifier and evaluate its test set performance

#D Export the tree internals to dot format and then render using graphviz

The resulting decision tree achieves 91.3% accuracy on the iris data set. We visualize it using the open-source graph visualization software `graphviz` package, which is used to render lists, trees, graphs, and networks.

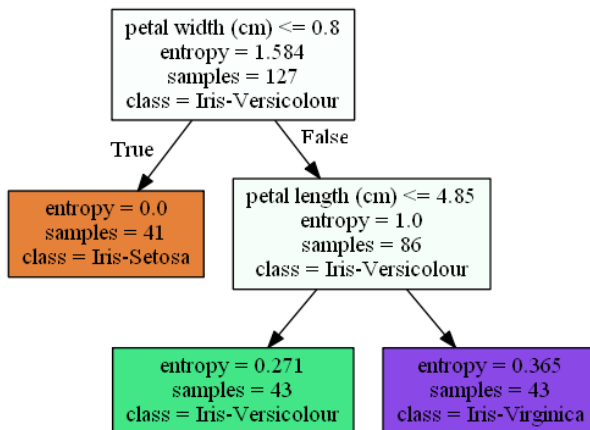


Figure 9.3. Decision tree learned on the Iris data set for classification of irises into three species: *Iris setosa*, *Iris versicolour* and *Iris virginica*. The standard convention for splits is followed here: if the split function evaluates to true, we proceed to the right branch and if it evaluates to false, the left branch.

The first thing we notice is that only two of the four features: petal width and length are enough to achieve over 90% accuracy. Thus, this decision tree has learned a *sparse model* by using only a subset of the features. But we can glean far more than that from this.

A nice property of decision trees is that each path from root node to leaf node represents a decision rule. At every split, since an example can either go left or right only, the example can only end up at one of the three leaf nodes. This means that each leaf node (and by extension, each path from root to leaf, that is each rule) partitions the overall population into a sub-population. Let's actually see this in action.

Since there are three leaf nodes, there are three decision rules, which we can write in Python syntax to understand them easier:

```
if petal_width <= 0.8:
    class = 'Iris-Setosa'
elif (petal_width > 0.8) and (petal_length <= 4.85):
    class = 'Iris-Versicolour'
elif (petal_width > 0.8] and (petal_length > 4.85):
    class = 'Iris-Virginica'
else:
    'Can never reach here'
```

In general, every decision tree can be expressed as a set of decision rules, which are more easily comprehensible to humans owing to their if-then structure.

**NOTE** The interpretability of decision trees can be subjective and depends on the depth of the tree and the number of leaf nodes. Trees of small-to-medium depth (say, up to 3-4) and approximately 8-15 nodes are generally more intuitive and easier to understand. As the tree depth and number of leaf nodes increase, the number and length of decision rules we will have to contend with and interpret also increases. This makes deep and complex decision trees also rather difficult to interpret and black-box-like.

Finally, observe that every example that passes through a decision tree *must end up at one and only one of the leaf nodes*. Thus, the set of paths from root to the leaves will fully cover all the examples.

What's more, this tree/rules will partition the space of all irises into three non-overlapping subpopulations, each corresponding to one of the three species. This is very helpful for visualization and interpretation as shown in Figure 9.4.

#### FEATURE IMPORTANCES

We know from the tree that two features are used: petal length and petal width. But how much did each feature contribute to the model? This is the notion of feature importance: where we ascribe a score to each feature depending on how much it influences overall decision-making in a model. In a decision tree, feature importances can be computed very easily!

Let's compute the feature importances for each feature in the tree in Figure 9.3, keeping in mind a couple of important details. First, the training set consisted of 127 training examples (`samples = 127` in the root node). Next, this tree was learned using entropy as the split-quality criterion (see Listing 9.1).

Thus, to measure feature importance, we simply compute how much each feature decreases entropy overall after the split. In order to avoid skewing our perception of splits with very small or very large proportion of examples, we will also weight the entropy decrease.

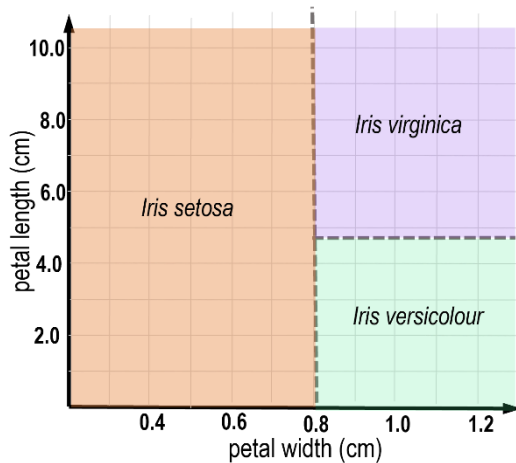


Figure 9.4. Decision trees partition the feature space into non-overlapping subspaces, where each subspace denotes a subpopulation of the examples.

More precisely, for each split node, compute how much its (weighted) entropy decreases with respect to its child nodes after the split:

$$\text{Importance}(\text{node}) \propto n_{\text{node}} H(\text{node}) - (n_{\text{left}} H(\text{left}) \propto n_{\text{right}} H(\text{right})).$$

For the node `[petal_width <= 0.8]`, `Importance(petal_width)  $\propto 127 \cdot 1.584 - (41 \cdot 0.0 \propto 86 \cdot 1.0) \propto 115.244$` . For the node `[petal_length <= 4.85]`, `Importance(petal_length)  $\propto 86 \cdot 1.0 - (43 \cdot 0.271 \propto 43 \cdot 0.365) \propto 58.599$` . Since the other two features are not used in the model, their feature importances will be zero.

The final step is to normalize the feature importances so that they sum to one. This gives us: `Importance(petal_width) = 115.244 / (115.244 + 58.599) = 0.663` and `Importance(petal_length)  $\propto 58.599 / (115.244 \propto 58.599) \propto 0.337$` .

In practice, we don't have to compute feature importances ourselves as most implementations of decision-tree learning do so. For example, the feature importances of the decision tree we just learned from Listing 9.1 can be obtained directly from the model (compare with our computation above):

```
model.feature_importances_
array([0.          , 0.          , 0.33708016, 0.66291984])
```

Finally, the example above showed the interpretability of decision trees for classification problems. Decision tree regressors can also be interpreted in the same way; the only difference is that the leaf nodes will be regression values instead of class labels.

### 9.1.3 Generalized Linear Models

We now revisit generalized linear models (GLMs), which were originally introduced in Section 7.1.4. Recall that generalized linear models extend linear models through a (nonlinear) link function,  $g(y)$ . For example, linear regression uses the identity link to relate the regression values  $y$  to the data  $x$ :

$$y = \beta_0 + \beta_1 x_1 + \dots + \beta_d x_d.$$

Here, the data point  $x = [x_1, \dots, x_d]^T$  is described by  $d$  features, and the linear model is parameterized by the linear coefficients  $\beta_1, \dots, \beta_d$  and the intercept (sometimes called the bias)  $\beta_0$ . Another example of a GLM is logistic regression, which uses the logit link to relate class probabilities  $p$  to the data  $x$ :

$$\ln\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x_1 + \dots + \beta_d x_d.$$

Generalized linear models are interpretable due to their linear and additive structure. The linear parameters themselves gives us an intuitive sense of each feature's contribution to the overall prediction. The additive structure ensures that the overall prediction depends on the individual contributions from each feature.

For example, consider that we've trained a logistic regression model for the diabetes diagnosis task discussed earlier, to classify if a patient has diabetes using two features: `age` and blood-glucose test result (`glc`). Let's say the learned model is (with  $p(y = 1)$  as  $p$ )

$$\ln\left(\frac{p}{1-p}\right) = 0.1 + 0.5 \text{age} - 0.29 \text{glc}.$$

Recall that if  $p$  is the probability of a positive diagnosis, then  $p/(1-p)$  are the *odds* that the patient has diagnosis. Thus, logistic regression represents the log odds of a positive diabetes diagnosis as a weighted combination of the features `age` and `glc`.

#### FEATURE WEIGHTS

How can we interpret the feature weights? If `age` is increased by 1,  $\ln(p/(1-p))$  will increase by 0.5 (because the model is linear and additive). Thus, for a patient who's a year older, their log odds of a positive diabetes diagnosis are  $\ln(p/(1-p)) + 0.5$ . Consequently, their odds of a positive diabetes diagnosis are  $p/(1-p) \times e^{0.5} \approx 1.65$ , or 65% more.

In a similar vein, if `glc` is increased by 1,  $\ln(p/(1-p))$  will decrease by 0.29 (note the minus in the weight, indicating a decrease). Thus, for a patient whose `glc` increases by 1, their odds of a positive diabetes diagnosis are  $p/(1-p) \times e^{-0.29} \approx 0.75$ , or 25% less.

Let's take this intuition and see how we can interpret a more realistic logistic regression model. We begin by training a logistic regression model on a data set called *Breast Cancer*.

The task is binary classification for breast cancer diagnosis. Each example in the data set is characterized by 30 features extracted from an image of the breast mass. These features characterize properties such as the radius, perimeter, area, concavity etc. of the breast mass.

### INTERPRETING GLMS IN PRACTICE

Listing 9.2 loads the data set, trains a logistic regression classifier, and visualizes the increase or decrease in the odds of a positive breast cancer diagnosis of each feature.

#### Listing 9.2. Training and interpreting logistic regression

```
from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
import matplotlib.pyplot as plt

bc = load_breast_cancer() #A
X, y = StandardScaler().fit_transform(bc.data), bc.target #B

Xtrn, Xtst, ytrn, ytst = train_test_split(X, y, test_size=0.15)
model = LogisticRegression(max_iter=1000, solver='saga', penalty='l1')
model.fit(Xtrn, ytrn) #C
ypred = model.predict(Xtst)
print('Accuracy = {0:5.3}%'.format(accuracy_score(ytst, ypred) * 100))

fig, ax = plt.subplots(figsize=(12, 4))

odds = np.exp(model.coef_[0]) - 1. #D
colors = np.full(odds.shape, fill_value='#0081ff')
colors[model.coef_[0] < 0] = '#ff0141'
ax.bar(height=odds, x=np.arange(0, Xtrn.shape[1]), color=colors) #E
```

#A Load the Breast Cancer data set and split into training and test sets

#B Preprocess the features ensure they are all the same scale

#C Train a logistic regression classifier and evaluate its test set performance

#D Compute the increase or decrease in odds as  $\exp(\text{weight}) - 1$

#E Visualize the change in odds as a bar chart

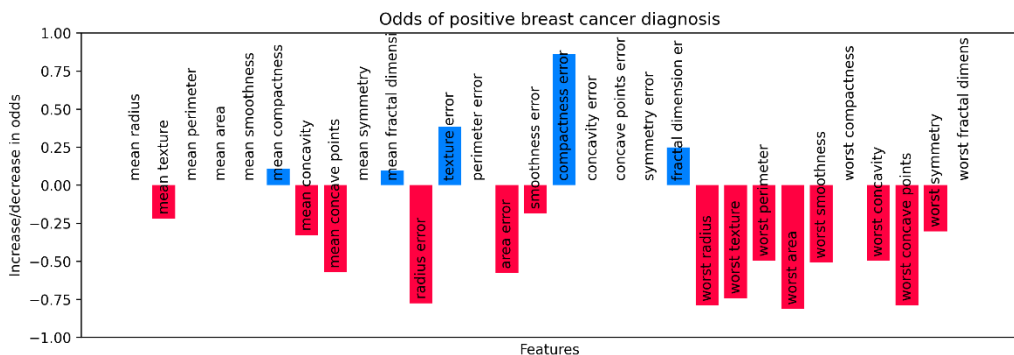


Figure 9.5. Interpreting a logistic regression, a linear model for classification, for breast cancer diagnosis. Positive feature weights lead to increased odds of breast cancer, negative feature weights lead to decreased odds of breast cancer, and zero feature weights do not affect the odds of breast cancer.



The odds of each feature  $i$  are calculated from the weights as  $odds_i = e^{w_i}$ . The change in odds is calculated as  $change_i = odds_i - 1 = e^{w_i} - 1$ , and visualized in Figure 9.5.

If the feature weight  $w_i > 0$ , then  $odds_i > 1$  and it will increase the odds of a positive diagnosis ( $change_i > 0$ ). If a feature weight  $w_i < 0$ , then  $odds_i < 1$  and it will decrease the odds of a positive diagnosis ( $change_i < 0$ ). If a feature weight  $w_i = 0$ , then  $odds_i = 1$  and that feature does not affect the diagnosis ( $change_i = 0$ ).

This last part is an important component of learning *sparse linear models*, where we learn a model as a mixture of zero and non-zero feature weights. A zero feature weight means that that feature does not contribute to the model and can be effectively dropped. This, in turn, allows for a sparser feature set and leaner, more interpretable models!

**NOTE** The interpretability of linear models is dependent on the relative scaling between the features. For example, age might be in the range 18-65, while salary might be in the range \$30,000-\$90,000. This disparity in features affects the underlying weight learning, and the feature with the higher weight range (in this case, salary) will dominate the models. When we interpret such models, we might incorrectly ascribe greater significance to such features. In order to train a robust model that considers all the features equally *during learning*, care must be taken to properly pre-process the data to ensure all features are in the same numerical range.

Linear regression models can also be interpreted similarly. In this case, rather than compute the odds, we can compute the contribution of each feature to the regression value directly, since the regression value  $y = \beta_0 + \beta_1 x_1 + \dots + \beta_d x_d$ .

## 9.2 Case Study: Data-driven Marketing

In the rest of this chapter, we will explore how we can train both black-box and glass-box ensembles in the context of a machine-learning task from the domain of data-driven marketing.

Data-driven marketing aims to use customer and socio-economic information to identify customers who will be most receptive to certain types of marketing strategies. This allows businesses to target specific customers with advertisements, offers and sales in an optimal and personalized way.

### 9.2.1 The Bank Telemarketing Data Set

The data set we will consider is the Bank Marketing data set<sup>4</sup> from the UCI repository and is a part of a phone-based direct marketing campaign of a Portuguese bank. The task is to predict if a customer will subscribe to a fixed-term deposit.

For each customer in the data set, there are four types of features: demographic attributes, details of the last phone contact, overall campaign information pertaining to this customer and general socio-economic indicators. The details are illustrated in Table 9.1.

<sup>4</sup> S. Moro, P. Cortez and P. Rita. A Data-Driven Approach to Predict the Success of Bank Telemarketing. *Decision Support Systems*, Elsevier, 62:22-31, June 2014.

**Table 9.1 Features and target of the bank marketing data set, grouped by the feature type and source.**

Feature	Type	Feature Description
<b>client demographic attributes and financial indicators</b>		
age	continuous	age of the customer
job	categorical	type of job (12 categories: e.g., blue-collar, retired, self-employed, student, services, etc., and unknown)
marital	categorical	marital status (divorced, married, single, unknown)
education	categorical	highest education (8 categories: e.g., high school, university degree, professional course, and unknown)
default	categorical	does customer have credit in default? (yes, no, unknown)
housing	categorical	does customer have a housing loan? (yes, no, unknown)
loan	categorical	does customer have a personal loan? (yes, no, unknown)
<b>date and time conditions of last marketing contact</b>		
contact	binary	contact communication type (cellphone, telephone)
month	categorical	last contact month (12 categories: jan-dec)
day-of-week	categorical	last contact weekday (5 categories: mon-fri)
<b>marketing campaign details from current and previous campaigns</b>		
campaign	continuous	total number of contacts during this campaign
pdays	continuous	number of days since the last contact in previous campaign
previous	continuous	number of contacts performed before this campaign
poutcome	categorical	outcome of previous marketing campaign (3 categories: failure, nonexistent, success)
<b>general social and economic indicators</b>		
emp.var.rate	continuous	employment variation rate - quarterly indicator
cons.price.idx	continuous	consumer price index - monthly indicator
cons.conf.idx	continuous	consumer confidence index - monthly indicator
euribor3m	continuous	euribor 3 month rate - daily indicator
nr.employed	continuous	number of employees - quarterly indicator
<b>prediction target</b>		
subscribed?	binary	has the customer subscribed to a term deposit?

An important thing to note is that this data set is extremely imbalanced: only 10% of the customers in the data set subscribed to a term deposit as a result of this marketing campaign.

The listing below loads the data set, splits into training and test sets and pre-processes them. The continuous features are scaled to between 0 and 1 using scikit-learn's `MinMaxEncoder` and the categorical features are encoded with `OrdinalEncoder`.

**Listing 9.3. Load and preprocess the Bank Marketing Data Set**

```

data_file = './data/ch09/bank-additional-full.csv'
df = pd.read_csv(data_file, sep=';') #A
df = df.drop('duration', axis=1) #B

from sklearn.model_selection import train_test_split
y = df['y']
X = df.drop('y', axis=1) #C
Xtrn, Xtst, ytrn, ytst = train_test_split(X, y, stratify=y, test_size=0.25)
#D

from sklearn.preprocessing import LabelEncoder #E
preprocess_labels = LabelEncoder()
ytrn = preprocess_labels.fit_transform(ytrn).astype(float)
ytst = preprocess_labels.transform(ytst)

from sklearn.preprocessing import MinMaxScaler, OrdinalEncoder #F
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer

cat_features = ['default', 'housing', 'loan', 'contact', 'poutcome',
                'job', 'marital', 'education', 'month', 'day_of_week']
cntnous_features = ['age', 'campaign', 'pdays', 'previous', 'emp.var.rate',
                    'cons.price.idx', 'cons.conf.idx', 'nr.employed',
                    'euribor3m']

preprocess_categorical = Pipeline(steps=[('encoder', OrdinalEncoder())])
preprocess_numerical = Pipeline(steps=[('scaler', MinMaxScaler())])
data_transformer =
    ColumnTransformer(transformers=[
        ('categorical', preprocess_categorical, cat_features),
        ('numerical', preprocess_numerical, cntnous_features)])
all_features = cat_features + cntnous_features

Xtrn = pd.DataFrame(data_transformer.fit_transform(Xtrn),
                    columns=all_features)
Xtst = pd.DataFrame(data_transformer.transform(Xtst), columns=all_features)

```

**#A** Load data set

**#B** Drop 'duration' column (see NOTE for a more detailed explanation)

**#C** Split the dataframe into features and labels

**#D** Split into train and test with stratified sampling to preserve class balances

**#E** Preprocess labels using LabelEncoder

**#F** Preprocess continuous features with MinMaxEncoder and categorical features with OrdinalEncoder

In order to prevent data and target leakage (see Chapter 8), we ensure that scaling and encoding functions are only fit to the training set before applying to the test set.

**NOTE** The original data set contains a feature called 'duration': the duration of the last phone call. Longer calls are highly correlated with the outcome of the call, because longer calls indicate more engaged customers who are likelier to subscribe. However, unlike other features which are known before making the call, we cannot possibly know a call's duration ahead of time. In this way, 'duration' essentially behaves like a target variable since both 'duration' and 'subscribed?' will immediately be known after a call. In order to build a realistic predictive model that can be deployed in practice with all features available *before* calling, we drop this feature from our modeling.

## 9.2.2 Training Ensembles

We will now train two ensembles (from two different packages) on this data set: `xgboost.XGBoostClassifier` and `sklearn.RandomForestClassifier`. Both these models will be complex ensembles of 200 decision trees (weighted ensembles, in the case of XGBoost) and are effectively black boxes. Once trained, we will explore how to make these black boxes explainable in Section 9.3.

Listing 9.4 shows how we can train an XGBoost ensemble over this data set. We use randomized grid search combined with 5-fold cross validation and early stopping (see Chapter 6 for additional details) to select among various hyperparameters such as learning rate and regularization parameters.

### Listing 9.4. Training XGBoost on the bank marketing data set

```
from xgboost import XGBClassifier
from sklearn.model_selection import RandomizedSearchCV

xgb_params = {'learning_rate': [0.001, 0.01, 0.1],
              'n_estimators': [100],
              'max_depth': [3, 5, 7, 9],
              'lambda': [0.001, 0.01, 0.1, 1],
              'alpha': [0, 0.001, 0.01, 0.1],
              'subsample': [0.6, 0.7, 0.8, 0.9],
              'colsample_bytree': [0.5, 0.6, 0.7],
              'scale_pos_weight': [5, 10, 50, 100]} #A

fit_params = {'early_stopping_rounds': 15, #B
              'eval_metric': 'aucpr',
              'eval_set': [(Xtst, ytst)],
              'verbose': 0}

xgb = XGBClassifier(objective='binary:logistic', use_label_encoder=False)#C
xgb_search = RandomizedSearchCV(xgb, xgb_params, cv=5, n_iter=40,
                               verbose=2, n_jobs=-1)
xgb_search.fit(X=Xtrn, y=ytrn.ravel(), **fit_params)
xgb = xgb_search.best_estimator_ #D
```

#A Create a grid of hyperparameters for XGBoost

#B Initialize early stopping and set early stopping rounds to 15

#C Set the classification loss for XGBoost to the logistic loss

#D Save the best XGBoost model after cross validation

Note also that one of the hyperparameters is `scale_pos_weight`, which allows us to weight positive and negative training examples differently. This is necessary since the bank marketing

data set is unbalanced (10%:90% positive-to-negative example ratio). By weighting the positive examples more, we can ensure that their contribution is not drowned out by the larger proportion of negative examples.

This listing trains an `XGBoostClassifier` that achieves around 87.24% test set accuracy and 74.67% balanced accuracy. We can use a similar procedure to train a random forest over this data set. The main difference is that we set the class weights for positive examples to 10.

#### Listing 9.5. Training a random forest on the bank marketing data set

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV

rf_params = {'max_depth': [3, 5, 7],
             'max_samples': [0.5, 0.6, 0.7, 0.8],
             'max_features': [0.5, 0.6, 0.7, 0.8]} #A

rf = RandomForestClassifier(class_weight={0: 1, 1: 10}, n_estimators=200) #B
rf_search = RandomizedSearchCV(rf, rf_params, cv=5, n_iter=30,
                              verbose=2, n_jobs=-1)
rf_search.fit(X=Xtrn, y=ytrn)
rf = rf_search.best_estimator_ #C
```

**#A** Create a grid of hyperparameters for `RandomForestClassifier`

**#B** Set the weights for negative-to-positive examples to be 1:10

**#C** Save the best random forest after cross validation

This listing trains a `RandomForestClassifier` that achieves around 84% test set accuracy.

### 9.2.3 Feature Importances in Tree Ensembles

Most of the ensembles in this book (including `XGBoostClassifier` and `RandomForestClassifier` trained in the previous subsection) are tree ensembles as they use decision trees as their base estimators.

This means that we can simply average the feature importances from the individual decision trees to give us feature importances for the learned ensembles!

In fact, the implementations of random forest (in scikit-learn) and XGBoost do this already, and we can obtain the ensemble feature importances using

```
xbg_search.best_estimator_.feature_importances_
rf_search.best_estimator_.feature_importances_
```

We visualize the feature importances of both these ensembles in Figure 9.6 to interpret and understand their decision making.

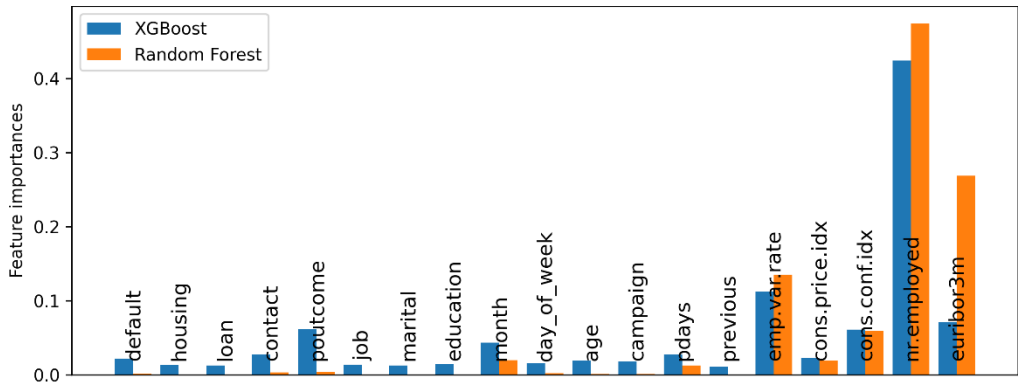


Figure 9.6. Feature importances of the ensembles learned by XGBoost and Random Forest classifiers.

Both ensembles ascribe significant importance to the socio-economic indicator variables, in particular `nr.employed` and `emp.var.rate` (which indicate unemployment rates), `euribor3m` (interbanking interest rates, which indicate financial stability) and `cons.conf.idx` (which consumer optimism regarding their expected financial situation).

The XGBoost model, however, is strongly reliant on just one of the variables over the others: `nr.employed`. The overall takeaway from is that people are more likely to subscribe to a fixed-term deposit account when the overall economic picture is optimistic.

Feature importances allow us to understand what a model is doing overall and over different types of examples. That is, feature importances are a type of global explainability method.

### 9.3 Black-Box Methods for Global Explainability

Methods for machine-learning model explainability can be categorized into two types:

- global methods attempt to *generally explain* a model's decision-making process, and what factors are broadly relevant, while
- local methods attempt to *specifically explain* a model's decision-making process with respect to individual examples and predictions.

Global explainability speaks to a model's sensible behavior over a large number of examples when deployed or used in practice, while local explainability speaks to a model's individual predictions on single examples that allow the user to make decisions on what to do next.

In this section, we look at some methods for the global explainability for black-box models. These approaches only consider a model's inputs and outputs and do not use the model internals (hence black box) to explain model behavior.

For this reason, they can be used for global explainability of *any* machine-learning method and are also called *model-agnostic methods*.

### 9.3.1 Permutation Feature Importance

Feature importance in a machine-learning model refers to a score that indicates how good a feature is in a model, that is, how effective it is in a model's decision-making process.

We've already seen how we can compute feature importances for decision trees, and by aggregation, for tree-based ensembles that use decision trees as base estimators.

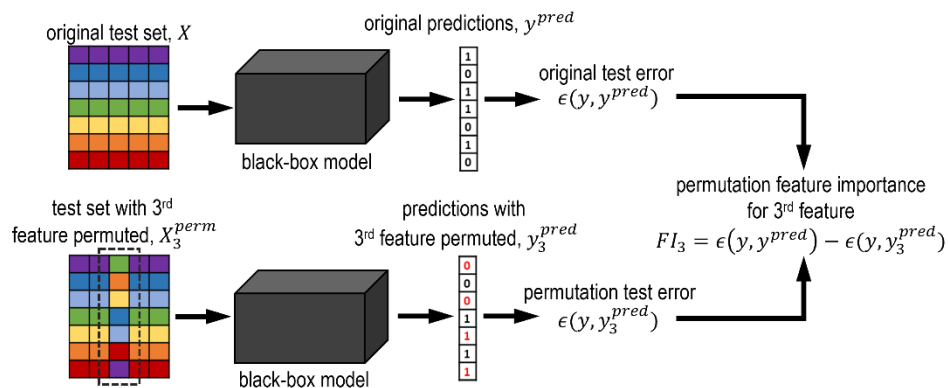
For tree-based methods, the feature importance calculation uses model internals such as the tree structure and split parameters. But what if these model internals are not available? Is there a black-box equivalent method for obtaining feature importances in such situations?

There is indeed: *permutation feature importance*. Recall that decision-tree feature importance scores each feature by how much it decreases the split criterion (such as gini impurity or entropy for classification, squared error for regression).

In contrast, permutation feature importance scores each feature by how much it increases the test error after we *permute (shuffle) that feature's values*.

The intuition here is straightforward: if a feature is more important, then "messing with it" affects its ability to contribute to predictions and will increase the test error. If a feature is less important, then "messing with it" will not have much of an impact on the model's predictions and will not affect the test error.

We "mess with a feature" by randomly permuting its values. This effectively snaps any relationship between that feature and its prediction. The procedure of permutation feature importance is illustrated in Figure 9.7:



**Figure 9.7.** The procedure for computing permutation feature importance illustrated for the 3<sup>rd</sup> feature. This procedure is repeated for all features. Permutation feature importance uses only inputs and outputs to estimate feature importance and does not use model internals (making this a model-agnostic approach).

The permutation feature importance is elegant and simple in how it scores features without access to model internals. Here are some important technical details to keep in mind, though:

- Permutation feature importance is a before-and-after score. It tries to estimate how the model's predictive performance changes from before to after we shuffle/permute features. In order to get a robust and unbiased estimate of the before and after model

performance, it is essential that we *use a hold-out test set!*

- There are many ways to evaluate a model's predictive performance depending on the task (classification or regression), the data set and our own modeling goals. For this task, for instance, consider the following performance metrics:
- balanced accuracy: Since this is a classification task, accuracy is a natural choice for a model evaluation metric. However, this data set is imbalanced with a 1:10 ratio of positive-to-negative examples. To account for this, we can use balanced accuracy, which ensures this skew is considered by weighting predictions by class size.
- recall: the purpose of this model is to identify "high-value" customers, who will subscribe to fixed-term deposits. From this perspective, we would like to minimize false negatives, or customers that our model thinks will not subscribe, but who actually will! This type of wrong prediction costs us customers, and recall is a good metric to minimize such false negatives.
- This procedure *randomly* shuffles feature values. As with any randomized approach, it's a good idea to repeat the process several times and average the result.

#### PERMUTATION FEATURE IMPORTANCE IN PRACTICE

The code snippet in Listing 9.6 computes permutation feature importances for the `XGBClassifier` trained in the previous section using `balanced_accuracy`.

#### Listing 9.6 Computing Permutation Feature Importance

```
from sklearn.inspection import permutation_importance
pfi = permutation_importance(xgb,
                             Xtst, ytst,      #A
                             scoring='balanced_accuracy', #B
                             n_repeats=30)    #C
```

**#A** Use a hold-out test set to compute feature importances

**#B** Different metrics can be used to evaluate model performance & feature importance

**#C** Repeat randomized shuffling of features

Figure 9.8 compares feature importance of the XGBoost model with the permutation importance computed using balanced accuracy and recall and visualizes the top 10 features identified by each approach.





Unlike permutation feature importance, which uses randomization to elicit the importance of a feature, the partial dependence relationship is identified using *marginalization*, or summing out.

Let's say that we are interested in computing the partial dependence between the target  $y$  and the  $k$ -th feature,  $X_k$ . Let the data set with the remaining features be  $X_{rest}$ . We have a black-box model  $y = f(X_k, X_{rest})$

To obtain the partial dependence function  $\hat{f}(X_k)$  from this black box, we simply have to sum over all possible values of all the other features  $X_{rest}$ , that is, we marginalize the other features. Mathematically, summing over all possible values of the other features is equivalent to integrating over them:

$$\hat{f}(X_k) = \int_{X_{rest}} f([X_k, X_{rest}]) dX_{rest}.$$

However, since computing this integral is not really feasible, we will need to approximate it. We can do so very easily using a set of  $n$  examples:

$$\hat{f}(X_k = a) = \frac{1}{n} \sum_{i=1}^n f([a, X_{rest}^i]).$$

The equation above gives us a straightforward way of computing the partial dependence function for a feature  $X_k$ .

For different values of  $a$ , we simply replace the entire column with  $a$ . Thus, for each  $a$  we create a new data set  $X^{[a]}$ , where the  $k$ -th feature takes the value  $a$  for every example.

The predictions of this modified data set using our model black-box will be  $y^{[a]} = f(X^{[a]})$ . The prediction vector  $y^{[a]}$  is a length- $n$  vector, containing the predictions of each test example in the modified data set. We can now average over these predictions to give us *one pair of points*

$$\left( X_k = a, \quad \hat{f}(X_k = a) = \frac{1}{n} \sum_{i=1}^n y_i^{[a]} \right).$$

We repeat this procedure for different values of  $a$  to generate the full partial dependence plot. This is illustrated in Figure 9.9 for two values,  $a = 0.1$  and  $a = 0.4$ .

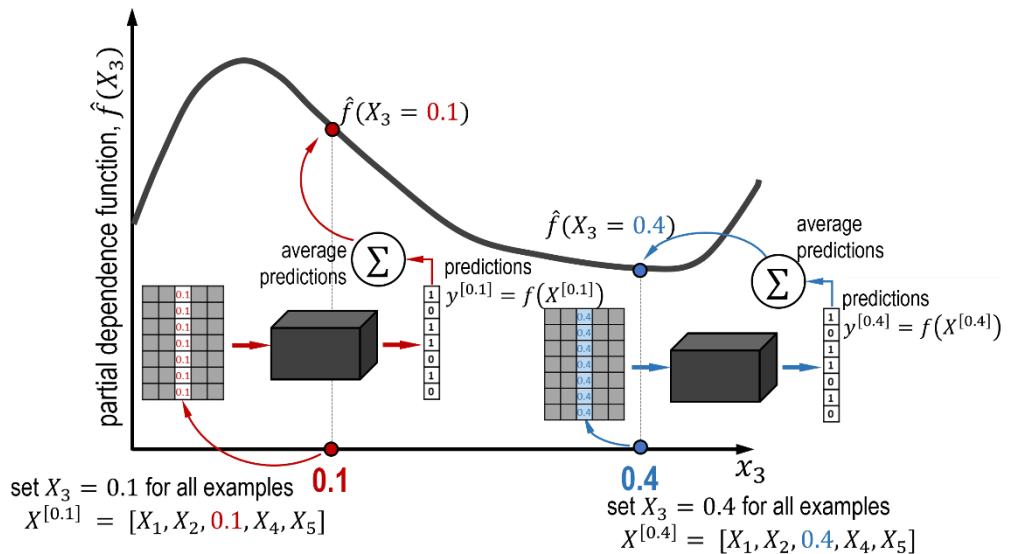


Figure 9.9. Two points in the partial dependence plot for the 3<sup>rd</sup> feature computed at  $X_3 = 0.1$  and  $X_3 = 0.4$ .

Partial dependence plots are intuitive to create and use, though can be somewhat time-consuming as new modified versions of the data set have to be created and evaluated for each point in the dependence plot. Here are some important technical details to keep in mind:

- Partial dependence tries to relate a model's output to input features, that is, model behavior in terms of what it has learned. For this reason, it is best to create and visualize a partial dependence plot with the training set.
- Remember that a partial dependence function is created by averaging across  $n$  examples. That is to say, each training example results in an example-specific partial dependence function. The partial dependence between a specific example and its output is called the *individual conditional expectation (ICE)*.

#### PARTIAL DEPENDENCE PLOTS IN PRACTICE

Listing 9.7 illustrates how to construct partial dependence plots for the `XGBoostClassifier` trained in Section 9.2 on the bank marketing data set.

**Listing 9.7. Creating partial dependence plots**

```

from sklearn.inspection import PartialDependenceDisplay as pdp
fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(10, 6))
pdp.from_estimator(xgb, Xtrn,
                  features=['euribor3m', 'nr.employed',
                           'contact', 'emp.var.rate'], #A
                  feature_names=list(Xtrn.columns), #B
                  kind='average', #C
                  response_method='predict_proba', #D
                  ax=ax)

```

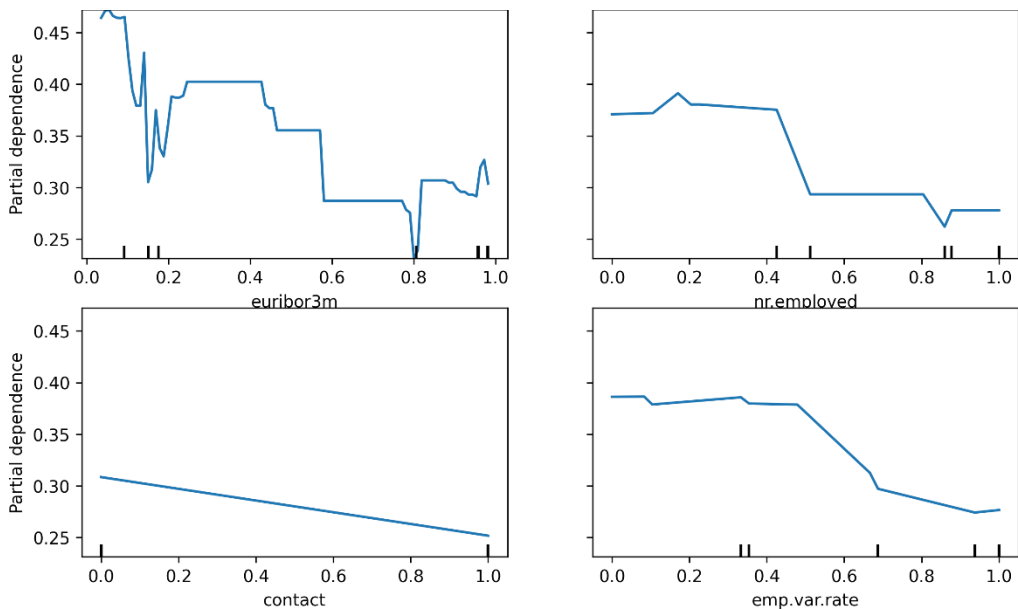
**#A** Features that we want to compute PDPs for

**#B** List of all the features in the data set

**#C** Plot individual conditional expectations for each example or the average PDP

**#D** Set whether we want partial dependence with predictions or prediction probabilities

Figure 9.10 shows the partial dependence function of four high-scoring variables: `euribor3m`, `nr.employed`, `contact` and `emp.var.rate` from the marketing data set.



**Figure 9.10.** Partial dependence plots of four variables in the bank-marketing data set.

The partial dependence plots give us further insight into how different variables belong and how they influence predictions. Note that, in Listing 9.6, we set `response_method` to `'predict_proba'`.

Thus, the plots above show how each variable (partially) influences the prediction probability of a customer subscribing to a fixed-deposit account. Higher prediction probabilities indicate that those attributes are more helpful in identifying 'high-value' customers.

For example, low values of `euribor3m` (say, in the range 0-0.5) generally correspond to higher subscription likelihoods. As discussed previously, this makes sense as lower bank borrowing rates typically mean lower customer interest rates, which would be attractive to a potential customer.

A similar conclusion can be drawn from the variables `emp.var.rate` and `nr.employed`, that lower unemployment rates are also likely to influence potential customers into opening fixed-deposit accounts.

**NOTE** A key assumption in the procedure for partial dependence plots is that the feature we are interested in  $X_k$  is not correlated with the remaining features,  $X_{rest}$ . This independence assumption is what allows us to marginalize the remaining features by summing over them. If the features  $X_{rest}$  are correlated, then marginalizing over them destroys some component of  $X_k$  as well, and we no longer have an accurate view of how much  $X_k$  contributes to the predictions.

One key limitation of PDPs is that it is only possible to create plots of partial dependence functions of one variable (curves), two variables (contours) or three variables (surface plots). Beyond 3 variables, it becomes impossible to visualize multi-variable partial dependence without breaking features down into smaller groups of 2 or 3.

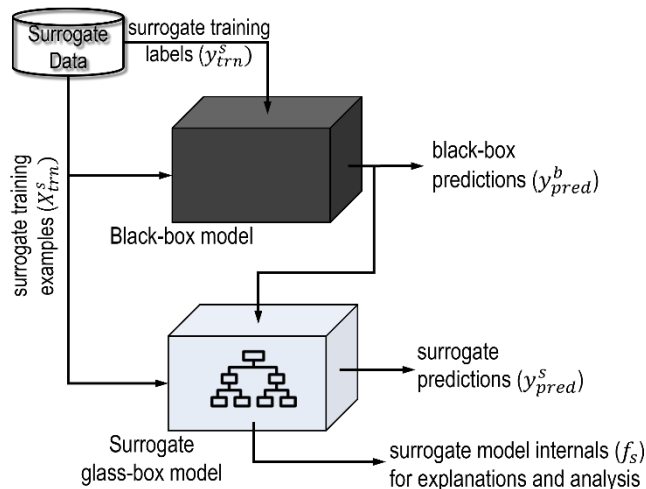
### 9.3.3 Global Surrogate Models

Black-box explanations such as feature importance and partial dependence aim to identify the impact of an individual feature or groups of features on predictions. In this section, we explore a more holistic approach that aims to approximate the behavior of the black-box model in an interpretable way.

The idea of a surrogate model is extremely simple: we train a second model that mimics the behavior of the black-box model. However, the surrogate model itself is a glass box and inherently explainable.

Once trained, we can use the surrogate glass-box model to explain the predictions of the black-box model. This is illustrated in Figure 9.11.

- A “surrogate data set” ( $X_{irm}^s, y_{irm}^s$ ) is used to train the surrogate model. The original data that was used to train the black-box model can also be used to train the surrogate model, if it is available. If not, an alternate data sample from the original problem space is used. The key is to ensure that the surrogate data set has the same distribution as the original data set that was used to train the black-box model.
- The surrogate model is trained *on the predictions* of the original black-box model. This is because the idea is to fit a surrogate model to *mimic* the behavior of the black-box model so that we can explain the black-box using the surrogate. Once trained, if the surrogate predictions ( $y_{pred}^s$ ) match the black-box predictions ( $y_{pred}^b$ ) then the surrogate model can be used to explain the predictions.
- Any glass-box model can be used as a surrogate model. This includes decision trees and generalized linear models, which can then be interpreted as shown in Section 9.1.



**Figure 9.11.** The procedure to train a global surrogate model from the predictions of a black-box model is illustrated. Both models are trained on the same surrogate training examples. However, the surrogate model is trained on the predictions of the black-box model, so that it can learn to mimic its predictions. If the black-box and surrogate make the same prediction, then the surrogate can be used to explain the black-box model's prediction.

#### THE FIDELITY-INTERPRETABILITY TRADEOFF

Let's train a surrogate decision tree to explain the behavior of the XGBoost model that was originally trained on the bank marketing data set. The original training set is used as the surrogate training set.

Keep in mind that we would like to tradeoff between two criteria while training the model: the surrogate's fidelity to the black-box model and the surrogate's explainability.

The surrogate's fidelity measures how well it can mimic the black-box's predictive behavior. More precisely, we measure how similar the surrogate model's predictions ( $y_{pred}^s$ ) are to the black-box model's predictions ( $y_{pred}^b$ ).

For binary classification problems, we can do this using metrics such as accuracy or  $R^2$  score (see Chapter 1). For regression problems, we can do this with metrics such as mean squared error, or  $R^2$  again. Higher  $R^2$  scores indicate better fidelity between the black-box and its surrogate.

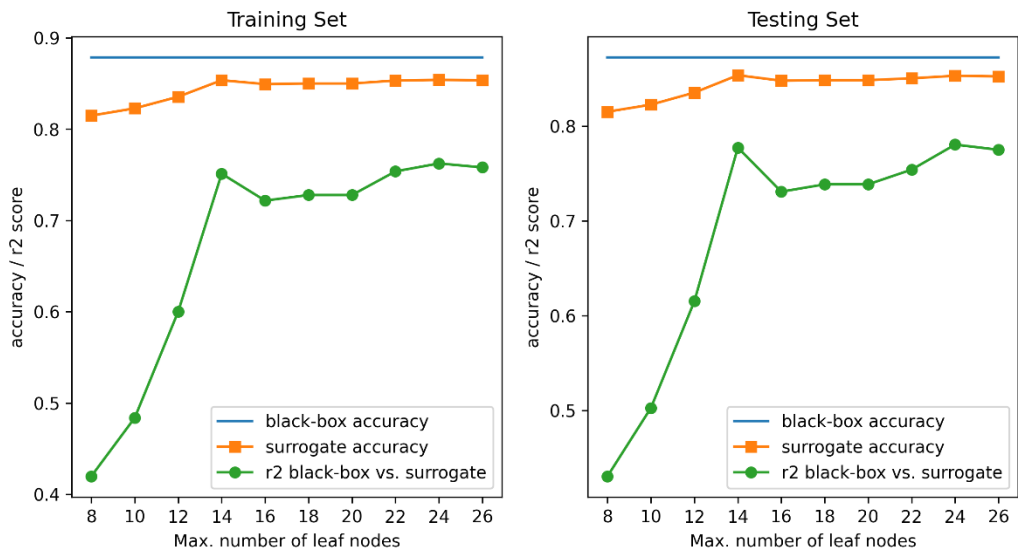
The surrogate's explainability depends on its complexity. Let's say that we want to train a decision-tree surrogate. Recall from our discussion in Section 9.1 that we need to limit the number of leaf nodes in the surrogate model for it to be human-interpretable as too many leaf nodes might lead to model complexity that might overwhelm the interpreter.

#### TRAINING GLOBAL SURROGATE MODELS IN PRACTICE

To train a useful surrogate model, we will need to find the sweet spot in the *fidelity-interpretability tradeoff*. This sweet spot will be a surrogate model that approximates the black-

box's predictions pretty well but is also not so complex that it defies any interpretation (possibly by inspection).

Figure 9.12 shows the fidelity-explainability tradeoff for a decision-tree surrogate trained for the XGBoost model. The surrogate is trained on the same bank marketing training set that was used to train the XGBoost model in Section 9.1.



**Figure 9.12.** The fidelity-explainability tradeoff for the bank marketing data set. The black-box model is an XGBoost ensemble, while the surrogate is a decision-tree trained on the black-box predictions.

We increase the surrogate's complexity (characterized by the number of leaf nodes), while keeping an eye on the fidelity (R2 score) between the black-box and surrogate predictions.

A decision-tree surrogate with 14 leaf nodes seems to achieve the ideal tradeoff between fidelity and complexity for explainability. Listing 9.8 trains a surrogate decision-tree model with these specifications.

#### Listing 9.8. Training a surrogate model

```
from sklearn.tree import DecisionTreeClassifier
surrogate = DecisionTreeClassifier(criterion='gini',
                                  max_leaf_nodes=14,      #A
                                  min_samples_leaf=20,    #B
                                  class_weight={0: 1, 1: 10}) #C
surrogate.fit(Xtrn, yb_trn_pred)
```

**#A** set maximum possible leaf nodes to 14

**#B** set minimum samples in a leaf node to 20 to avoid overfitting

**#C** set class weights to 1 for negative examples and 10 for positive examples to account for the class imbalance

Figure 9.13 shows the decision-tree surrogate for the XGBoost model.

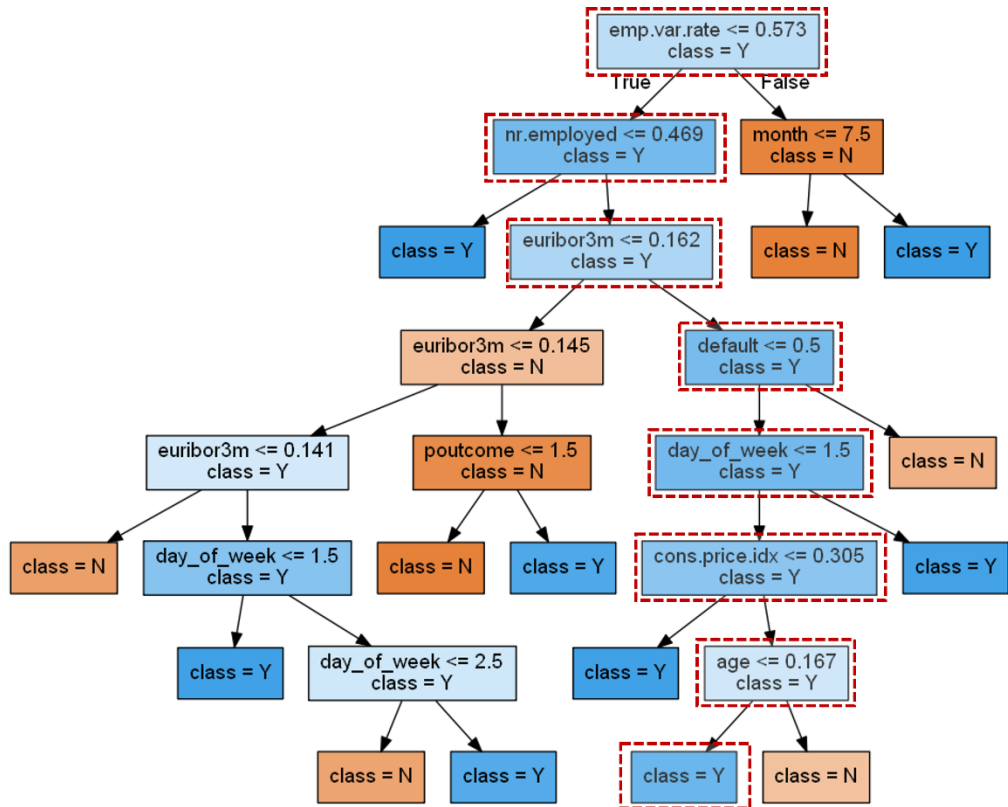


Figure 9.13. Surrogate model trained from the predictions of the XGBoost model, which was originally trained on the bank marketing data set. This tree has 14 leaf nodes. Inspecting and analyzing this tree can yield many insights. One such insight for the highlighted path from root to leaf is discussed in the text.

Several variables appear in the highlighted path from leaf to a tree node. These variables describe a 'high-value' subpopulation and provide insights into potentially successful strategies.

First, the socio-economic variables such as `emp.var.rate`, `nr.employed`, `cons.price.idx` and `euribor3m` identify favorable societal circumstances during which to launch a successful campaign.

The node `age <= 0.167` is obtained on the pre-processed data, where 0.167 corresponds to 42 before rescaling. Thus, the nodes `[default <= 0.5]` and `[age <= 0.167]` suggest that customers who are under 42 years of age and have no previous defaults are 'high value'.

Finally, `[day_of_week <= 1.5]` suggests that calling these high-value customers on Monday (`day_of_week = 0`) or Tuesday (`day_of_week = 1`) is a good strategy.

Can you see if you can identify any other viable strategies for identifying high-value customers and strategies?



## 9.4 Black-Box Methods for Local Explainability

The previous section introduced methods for global explainability, which aim to explain a model's global behavioral trends across different types of input examples and subpopulations.

In this section, we will explore methods for *local explainability*, which aim to explain a model's individual predictions. The explanations allow users (such as doctor's using a diagnostic system) to trust the predictions and take actions based on them. This is tied to the user's ability to understand *why* a model made a particular decision.

### 9.4.1 Local Surrogate Models with LIME

The first method we'll look at is called *Locally Interpretable Model-Agnostic Explanation*, or *LIME*. As the name rather transparently suggests, LIME is (1) a model-agnostic method, which means it can be used with any machine-learning method as it treats it like a black-box, (2) a local interpretability method that is used to explain a model's individual predictions.

LIME is, in fact, a *local surrogate method*. It uses a linear model to approximate a black-box model's predictions in the locality of the example whose predictions we are interested in explaining. This intuition is shown in Figure 9.14, which shows the complex surface of a (black-box) model, and an interpretable linear surrogate model that approximates black-box behavior around a single example of interest.

#### THE FIDELITY-INTERPRETABILITY TRADEOFF AGAIN

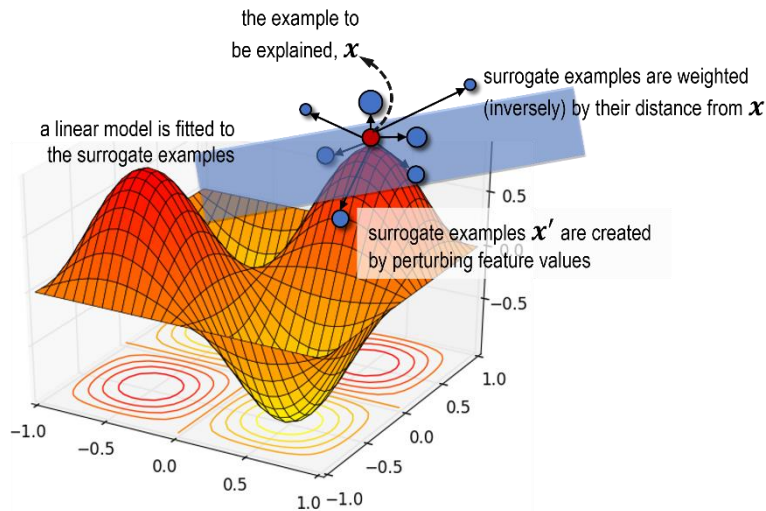
Given a training example whose predictions we want to explain, LIME trains a local surrogate to be the model with the best tradeoff between fidelity and interpretability.

In the previous section, we trained a decision-tree surrogate to optimize the fidelity-interpretability tradeoff.

Let's write this down more formally. First, we denote the black box model by  $f_b(\mathbf{x})$  and the surrogate model by  $f_s(\mathbf{x})$ . We measure fidelity between the predictions of the black box ( $f_b$ ) and the surrogate ( $f_s$ ) using the  $R^2$  score. We measure interpretability of the surrogate model by using the number of leaf nodes in the tree: fewer leaf nodes generally leading to better interpretability.

Let's say that we want to explain the predictions of the black box on example  $x$ . For decision-tree surrogate training, we tried to find a decision tree that optimized the following

$$\text{surrogate training criteria for trees} = \underbrace{R^2(f_b(\mathbf{x}), f_s(\mathbf{x}'))}_{\text{fidelity}} + \frac{\text{interpretability}}{n_{leaf}(f_s)}.$$



**Figure 9.14.** LIME creates a surrogate training set of examples (visualized in blue) in the locality of the example whose prediction needs to be explained (visualized in red). These examples further weighted by their distance. This is indicated by the sizes of the surrogate examples, with closer examples getting higher weights (and visualized larger). A weighted loss function is used to fit a linear surrogate model, which provides local explanations.

In a similar vein, LIME trains a linear surrogate by optimizing

$$\text{surrogate training criteria for LIME} = \underbrace{L(f_b(\mathbf{x}), f_s(\mathbf{x}'), \pi_x)}_{\text{fidelity}} + \frac{\text{interpretability}}{n_{non-zeros}(f_s)}.$$

Here, the examples  $\mathbf{x}'$ , called surrogate training examples, will be used to train the surrogate model. The loss function that is used to measure fidelity is a simple weighted mean-squared error that measures the disparity in the predictions of the black box and the surrogate:

$$L(f_b(\mathbf{x}), f_s(\mathbf{x}'), \pi_x) = \sum_z \underbrace{\pi_x(\mathbf{x}')}_{\text{local weight}} \cdot (f_b(\mathbf{x}) - f_s(\mathbf{x}'))^2.$$

The surrogate is a linear model of the form  $f_s(\mathbf{x}) = \beta_0 + \beta_1 x_1 + \dots + \beta_d x_d$ . As we've seen in Section 9.1, the interpretability of linear models depends on the number of features. Fewer

features make analyzing their corresponding parameters  $\beta_k$  easier. Thus, LIME seeks to train sparser linear models with more zero parameters to promote interpretability.

A good linear surrogate will trade-off between fidelity and interpretability. But what makes LIME local? How can we train a local surrogate model? How do we obtain surrogate examples  $x'$ ? And what are these local weights ( $\pi_{x'}$ ) in the equation above?

#### SAMPLING SURROGATE EXAMPLES FOR LOCAL EXPLAINABILITY

We now have a well-defined fidelity-interpretability criterion to train our surrogate model. If we used the entire training set, we would obtain a global surrogate model.

To train a local surrogate model we need data points that are “close” or “similar” to our example of interest. LIME creates a local surrogate training set by sampling and smoothing.

Let’s say that we are interested in explaining the prediction of the black-box on an example with five features  $x = [x_1, x_2, x_3, x_4, x_5]$ . LIME samples data in a neighborhood of  $x$  as follows:

- *Perturb*: Randomly generate perturbations for each feature. For continuous features, perturbations are randomly sampled drawn from the normal distribution,  $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{1})$ . For categorical features, these are randomly sampled from the multivariate distribution over  $K$  category values,  $\epsilon \sim \text{Cat}(K)$ . This generates one surrogate example  $x' = [x_1 + \epsilon_1, x_2 + \epsilon_2, x_3 + \epsilon_3, x_4 + \epsilon_4, x_5 + \epsilon_5]$ . This example can now also be labeled using the black box,  $y' = f_b(x')$ . And so on, until we obtain a surrogate set  $Z$  in the locality of  $x$ .
- *Smooth*: Each surrogate training example is also assigned a weight using the exponential smoothing kernel:  $\pi_{x'}(x) = \exp(-\gamma \cdot D(x, x'))$ . Here,  $D(x, x')$  is the distance between our example that needs to be explained  $x$  and a perturbed sample  $z$ . Samples that are further from  $x$  get smaller weights and those that are closer to  $x$  get higher weights. Thus, this function encourages the surrogate model to prioritize surrogate examples that are ‘more local’ when training a linear approximation.

The smoothing parameter  $\gamma \geq 0$  controls the width of the kernel. Increasing  $\gamma$  allows LIME to consider larger neighborhoods, making the model less local.

Now that we have a surrogate training set in the locality of the example  $x$ , we can train a linear model. The goal is to train it to induce sparsity (as many zero parameters) as possible. LIME supports training of sparse linear models with L1 regularization, such as LASSO or elastic net.

These models are covered in Chapter 7 for linear regression and can be easily extended to logistic regression for classification as well.

**NOTE** Keen observers may have noticed that the exponential kernel is the same as a radial-basis function (RBF) kernel that is used in support vector machines and other kernel methods. From that perspective, the exponential smoothing kernel is essentially a similarity function. Points that are closer are considered more similar and will have higher weights.

#### LIME IN PRACTICE

LIME is available as a package available through Python’s two most popular package managers: pip and anaconda. The package’s GitHub page (<https://github.com/marcotcr/lime>)

also contains additional documentation and a number of examples illustrating how to use it for classification, regression and applications in text and image analytics.

In Listing 9.9, we use LIME to explain the predictions of a test set example from the bank marketing data set. Test example 3104 is a customer who did subscribe, which the XGBoost model identified with 64% confidence, a true positive example.

### Listing 9.9. Using LIME to explain XGBoost predictions

```
cat_features = ['default', 'housing', 'loan', 'contact', 'poutcome',
               'job', 'marital', 'education', 'month', 'day_of_week']
cat_idx = np.array([cat_features.index(f) for f in cat_features]) #A

from lime import lime_tabular
explainer = lime_tabular.LimeTabularExplainer(
    Xtrn.values, #B
    feature_names=list(Xtrn.columns), #C
    class_names=['Sub?=NO', 'Sub?=YES'],
    categorical_names=cat_features,
    categorical_features=cat_idx, #A
    kernel_width=75.0, #D
    discretize_continuous=False)

exp = explainer.explain_instance(Xtst.iloc[3104], xgb.predict_proba) #E
fig = exp.as_pyplot_figure() #F
```

#A Identify the categorical features and their indices explicitly (for visualization)

#B Pass the training set, which is sometimes used for sampling, especially continuous features

#C Identify the feature names and class names explicitly (for visualization)

#D Set the kernel width for this data set (identified here by trial-and-error)

#E Explain the predictions of test example 3104

#F Visualize the explanation as a bar chart

Figure 9.15 visualizes the local weights identified by LIME to explain this example.

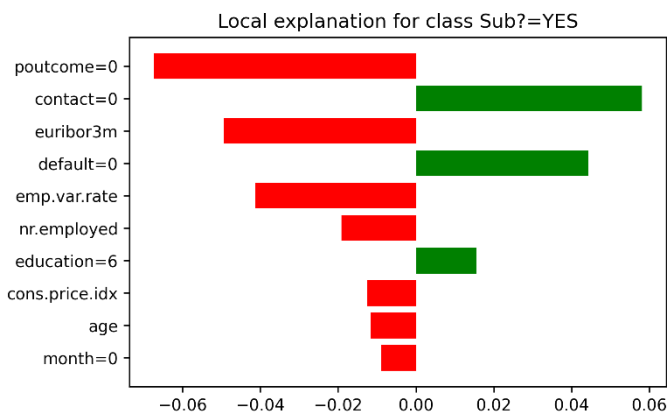


Figure 9.15. Explanations generated by LIME for test example 3104 (a true positive prediction). Features that contributed to a negative prediction (will not subscribe) are in red and those to a positive prediction (will subscribe) are in green.

The features and feature values (of the example being explained) are shown in the y-axis. The x-axis shows LIME feature importances.

Aside from socioeconomic trends, let's look at the personalized features of this customer. The variables with the biggest impact are `poutcome(=0)`, whether the previous marketing campaign was successful (it wasn't), `contact(=0)`, whether they were contacted by cellular or landline (here, 0=cellular) and `default`, whether they have prior banking defaults in their prior history (they don't).

These interpretations are intuitive to even non-technical people, such as sales and marketing, who might further analyze them to fine tune future marketing campaigns.

### 9.4.2 Local Interpretability with SHAP

In this section, we will cover another widely used local interpretability approach: *SHapley Additive exPlanations*, or SHAP. SHAP is a model-agnostic black-box explainer, which is used to explain individual predictions (hence, local interpretability) through feature importance.

SHAP is a feature attribution technique that computes feature importance based on each feature's contribution to the overall prediction. SHAP is built upon the concept of Shapley values, which comes from the field of co-operative game theory.

In this section, we will learn what Shapley values are, how they can be applied to computing feature importances and how we can compute them efficiently in practice.

#### UNDERSTANDING SHAPLEY VALUES

Let's say a group of 4 data scientists (Ava, Ben, Cam, and Dev) work collaboratively on a Kaggle Challenge and win first place with total prizemoney of \$20,000. Being a fair-minded group, they decide to split the prizemoney based on their contributions.

They do this by trying to figure out how well they work in various combinations. Since they've worked together a lot in the past, they write down how well they work individually, and also in groups of 2 and in groups of 3. These values are shown in Figure 9.16.

This table lists every possible combination of Ava, Ben, Cam, and Dev, also known as a *coalition*. Associated with each coalition is its value (prizemoney in \$1000 units), that indicates how much each coalition might have been worth had they only been working on this project.

For example, the coalition of Ava alone has value of \$7,000, while the coalition of Ava, Ben and Dev has a value of \$13,000. The last coalition of all four of them, called the *grand coalition*, has a value of \$20,000, the overall prizemoney.

The *Shapley value* allows us to attribute the overall prizemoney to each of these four team members across all the coalitions possible. It essentially helps us determine team member importance to the overall collaboration and helps us determine a fair way to split the overall value of the collaboration (in this case, the prizemoney).

The Shapley value of each team member  $p$  is computed in a very intuitive manner: we look at how the value of *each coalition changes*, with and without that team member. More formally:

$$\phi_p = \sum_{S \in \text{all coalitions without } p} \frac{(n - n_S - 1)! n_S!}{\text{weight, } \pi_x} \cdot \left( \frac{\text{value of the coalition } S \text{ with } p}{\text{value of the coalition } S} - \frac{\text{value of the coalition } S \text{ without } p}{\text{val}(S)} \right).$$

Coalition	Val	Coalition	Val
	0		8
	7		11
	5		10
	4		12
	3		13
	8		16
	9		18
	12		20

Figure 9.16. All possible coalitions of Ava, Ben, Cam and Dev, and their corresponding values (in units of \$1000). The last coalition contains all four friends and has value \$20,000, the total prizemoney. There is one coalition of size 0, 4 coalitions of size 1, 6 coalitions of size 2, 4 coalitions of size 3 and 1 coalition of size 4. This table is called the characteristic function. (Icons credit: freepix.com).

This equation might look intimidating at first, but it's actually quite simple. Figure 9.17 illustrates the components of this equation when computing the Shapley values for Dev (team member 4): (1) coalitions *with* Dev on the first row, (2) corresponding coalitions *without* Dev on the second row, and (3) the weighted difference between the two on the third row.

The weights are computed using  $n$ , the total number of team members (in this case, 4) and  $n_S$ , the coalition size. For example, for the coalition  $S = \{Ava, Cam\}$ ,  $n_S = 2$ . The weights for the coalition without **Dev** ( $S$ ) and with **Dev**  $S \cup \{Dev\}$  will both be  $1!2!/4! = 1/12$ . Other weights can be computed similarly.

Summing all the weighted differences in the last row in Figure 9.17 gives us the Shapley value for Dev,  $\phi_{Dev} = 6$ . Similarly, we can also obtain  $\phi_{Ava} = 4.667$ ,  $\phi_{Ben} = 4.333$  and  $\phi_{Cam} = 5$ . This suggests that an equitable way to attribute the prize money based on contribution is \$4667, \$4333, \$5000, and \$6000 respectively between Ava, Ben, Cam, and Dev.

















1 coal. of size 1	3 coalitions of size 2				3 coalitions of size 3			1 coal. of size 4
 3	 12	 11	 10	 13	 16	 18	 20	
 0	 7	 5	 4	 8	 9	 8	 12	
$\frac{1}{4}(3-0)$	$\frac{1}{12}(12-7)$	$\frac{1}{12}(11-5)$	$\frac{1}{12}(10-4)$	$\frac{1}{12}(13-8)$	$\frac{1}{12}(16-9)$	$\frac{1}{12}(18-8)$	$\frac{1}{4}(20-12)$	

Figure 9.17. Computing the Shapley values for Dev. The top row is all the coalitions with Dev. The middle row are the corresponding coalitions without Dev. The last row shows the individual weighted differences in the values of the coalitions. Summing across the last row gives us the Shapley values for Dev:  $\phi_{Dev} = 6$ .

The Shapley value has some interesting theoretical properties. First, observe that  $\phi_{Ava} = 4.667$ ,  $\phi_{Ben} = 4.333$ ,  $\phi_{Cam} = 5$ ,  $\phi_{Dev} = 6$ . That is, the Shapley values sum to the value of the grand coalition:

$$\sum_p \phi_p = val(\{1, 2, \dots, n\}).$$

This property of the Shapley value, called *efficiency*, ensures that the value of the overall collaboration is exactly broken down and attributed to each team member in the collaboration.

Another important property is *additivity*, which ensures that if we have two value functions, the overall Shapley value computed using a joint value function is equal to the sum of the individual Shapley values. This has some important implications for ensemble methods since it allows us to add Shapley values across individual base estimators to obtain the Shapley values across the entire ensemble.

So, what does the Shapley value have to do with explainability? Analogous to the case of the four data scientist friends above, features in a machine learning problem collaborate together to make predictions. The Shapley value allows us to attribute how much each feature contributed to the overall prediction.

#### SHAPLEY VALUES AS FEATURE IMPORTANCE

Let's say that we want to explain the predictions of a black-box model  $f$  on an example  $x$ . The Shapley value of a feature  $j$  is computed as:

$$\phi_j = \sum_{S \in \text{all feature coalitions without feature } j} \frac{(n - n_S - 1)! n_S!}{\text{weight}, \pi_x} \cdot \left( \underbrace{f(\mathbf{x}_{S \cup j})}_{\text{model using features } S \text{ with } j} - \underbrace{\widehat{f}(\mathbf{x}_S)}_{\text{model using features } S \text{ without } j} \right).$$

We use the black-box as the characteristic / value function. As before, we consider all possible coalitions with and without the feature  $j$ .

Now, we can compute the Shapley values for all the features. As before, the Shapley value for feature importance estimation is efficient and attributes a part of the overall prediction to each feature:

$$\sum_j \phi_j = f(\mathbf{x}).$$

The Shapley value is theoretically well-motivated and has some very attractive properties that make it a robust measure of feature importance. There is one significant limitation to using this procedure directly in practice: scalability.

The Shapley computation uses trained models to score feature importance. In fact, it will need to *use one trained model for each coalition of features*. For example, for our diabetes diagnosis model from earlier with 2 features: `age` and `glc`, we will have to train 3 models, one for each coalition:  $f_1(\text{age})$ ,  $f_2(\text{glc})$  and  $f_3(\text{age}, \text{glc})$ .

In general, if we have  $d$  features, we will have  $2^d$  total coalitions and will have to train  $2^d - 1$  models (we don't train a model for the null coalition). For instance, the bank marketing data set has 19 features and will require the training of  $2^{19} - 1 \approx 524,287$  models! This is simply absurd in practice.

#### SHAPLEY ADDITIVE EXPLANATIONS (SHAP)

What can we do in the face of such combinatorial infeasibility? What we always do: approximate and sample. Inspired by LIME, the SHAP method aims to learn a linear surrogate function whose parameters are the Shapley values for each feature.

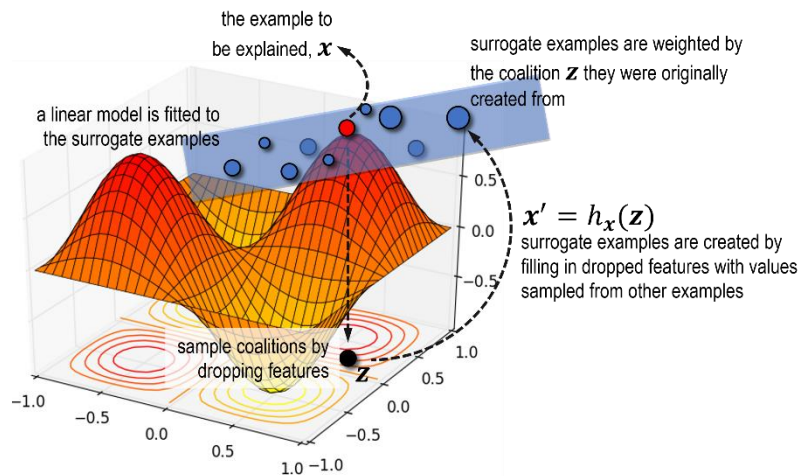
Analogous to LIME, given a black-box model  $f_b(\mathbf{x})$ , SHAP also learns a surrogate model  $f_s(\mathbf{x})$  using a loss function that has an identical form to LIME's. Unlike LIME, however, we have to accommodate the notion of coalitions in the loss function:

$$L(f_b(\mathbf{x}), f_s(\mathbf{x}'|\mathbf{z}), \pi_x) = \sum_{\mathbf{z}} \underbrace{\pi_x(\mathbf{z})}_{\text{local weight}} \cdot (f_b(\mathbf{x}) - f_s(h_x(\mathbf{z})))^2$$

Let's understand this loss function and SHAP by seeing what it does similarly and differently from LIME (also see Figure 9.18). As before, let's say that we are interested in explaining the prediction of the black-box on an example with five features  $\mathbf{x} \approx [x_1, x_2, x_3, x_4, x_5]$



- LIME creates surrogate examples  $x'$  by randomly perturbing the original example  $x$ . SHAP uses a slightly more involved two-step approach to create surrogate examples
- First, SHAP generates a random coalition vector,  $z$ , which is a 0-1 vector indicating if a feature is in the coalition or not. For example,  $z = [1, 1, 0, 1]$  represents a coalition of the first, second and fifth features.
- Next, SHAP creates a surrogate example from  $z$  by using a mapping function  $h_x(z)$ . Wherever  $z_j = 1$ ,  $h_x = x_{i_j}$ , the original feature value. Wherever  $z_j = 0$ ,  $h_x = x'_{i_j}$ , a feature value from a random example.
- Thus, each surrogate example is a patchwork of features from the original training example we want to explain and another random training example. The idea is that features belonging to the coalition get “good values” from and features not belonging to the coalition get random “garbage values”.
- LIME weights surrogate examples  $x'$  inversely by their distance from  $x$  using the RBF/exponential kernel. SHAP weights surrogate examples  $x'$  using the Shapley kernel, which is simply the weight from the Shapley computation,  $\pi_x(z) = \binom{d - n_z - 1}{n_z} / d!$ , where  $d$  is the number of features and  $n_z$  is the coalition size (number of 1s in  $z$ ). Intuitively, this weight reflects the number of other similar coalitions, with a similar number of zero and non-zero features.



**Figure 9.18.** SHAP creates a surrogate training set of examples (visualized in blue) in the locality of the example whose prediction needs to be explained (visualized in red).

Now that we have a surrogate training set in the locality of the example  $x$ , we can train a linear model. The weights of this linear model will be the approximate Shapley values for each feature.

**SHAP IN PRACTICE**

SHAP is available as a package available through Python's two most popular package managers: `pip` and `anaconda`. The package's GitHub page (<https://github.com/slundberg/shap>) also contains additional documentation and a number of examples illustrating how to use it for classification, regression and applications in text and image analytics.

In this section, we will use a version of SHAP called TreeSHAP that is specifically designed to be used for tree-based models, including individual decision trees and ensembles. TreeSHAP is a special variant of SHAP that exploits the unique structure of decision trees in the function  $h_x(\mathbf{z})$  to calculate the Shapley values *efficiently*.

As mentioned before, Shapley values have a nice property called additivity. For us, this means that if we have a model that is an additive combination of trees, that is, tree ensembles (such as bagging, random forests, gradient and Newton boosting among others), then the Shapley value of the ensemble is simply the sum of the Shapley values of the individual trees.

Since TreeSHAP can efficiently compute the Shapley values of each feature in each individual tree in an ensemble, we can efficiently get the Shapley values of the entire ensemble.

Finally, unlike LIME, TreeSHAP does not require us to furnish a surrogate data set, as the trees themselves contain all the information (feature splits, leaf values/predictions, example counts etc.) needed.

TreeSHAP supports many of the ensemble methods discussed in this book, including XGBoost. Listing 9.10 shows how to compute and interpret the Shapley values for test example 3104 of the bank marketing data set using an XGBoost model.

**Listing 9.10. Using TreeSHAP to explain XGBoost predictions**

```
import shap
explainer = shap.TreeExplainer(xgb, feature_names=list(Xtrn.columns))
shap_values = explainer(Xtst.iloc[3104].values.reshape(1, -1)) #A
shap.plots.waterfall(shap_values[0]) #B
shap.initjs()
shap.plots.force(shap_values[0]) #C
```

```
#A Explain the predictions of test example 3104
#B Visualize Shapley values using a waterfall plot
#C Visualize Shapley values using a force plot
```

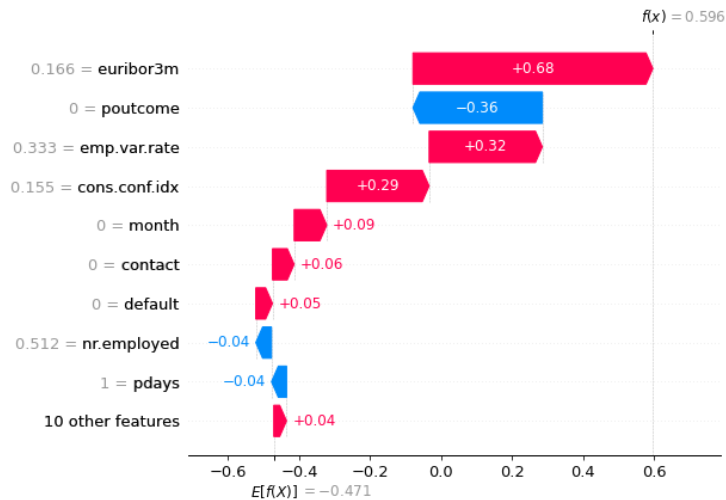


Figure 9.19. A waterfall plot to visualize Shapley values. The gray text before the feature names shows the feature values for test example 3104, while the text in the bars shows their Shapley values.

This snippet visualizes Shapley values in two ways: as a waterfall plot and as a force plot.

SHAP explains classifier models in terms of their prediction probabilities (confidence). For a classifier, the x-axis values will be the log-odds, with 0.0 representing even odds (1:1) of classification, or 50% prediction probability as a positive example.

Force plots allow for a more intuitive view of how the features contribute to a prediction. The plot is centered around the prediction (0.596 for this example) and visualizes how much the features contribute towards a positive or negative explanation.

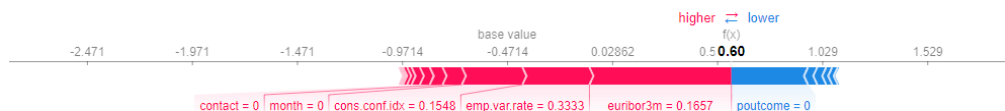


Figure 9.20. A force plot to visualize Shapley values. The features to the left (in red) contribute to a positive decision (Sub? = YES), while the features to the right (in blue) contribute to a negative decision. The feature values of the example being explained are shown along with the features under the force plot.

**NOTE** LIME and SHAP are both *additive* local explainability methods. This means that they can be extended to global explainability in a rather straightforward manner: global feature importances from either method can be obtained by averaging over local feature importances computed over a task-relevant data set.

One drawback of LIME and SHAP is that they are fundamentally designed only to compute and evaluate individual feature importances, and not *feature interactions*. SHAP offers some support for visualizing feature interactions in a manner similar to partial dependence plots.

However, like partial dependence plots, SHAP does not have any mechanism to automatically identify important interacting feature groups and forces us to visualize all pairs, which can be overwhelming. For example, with 19 features in the bank marketing data set, we will have 171 pairwise feature interactions.

In real-world applications, since many features depend on each other, it is important to also understand how feature interactions come into play in decision making. In the next section, we will learn about one such method: *explainable boosting machines*.

## 9.5 Glass-Box Ensembles: Training for Interpretability

We have learned about model-agnostic explainability methods. These methods can take a model that was already trained (for example, by an ensemble learner such as XGBoost) and attempt to explain the model itself (global) or its predictions (local).

But instead of treating our ensembles as a black box, can we learn an explainable ensemble from scratch? Can this ensemble method still perform well *and* be explainable?

These are the types of questions that motivated the development of *Explainable Boosting Machines*, or EBMs, a type of glass-box ensemble method. Some key highlights of EBMs are:

- EBMs can be used for *both* global explainability and local explainability of individual examples!
- EBMs learn a fully factorized model, that is, the model components only depend on individual features or pairs of features. These components provide interpretability directly and EBMs need no additional computations (like SHAP or LIME) to generate explanations.
- EBMs are a type of *generalized additive model* (GAM), which are nonlinear extensions of generalized linear models (GLMs) discussed in this chapter and elsewhere in the book. Similar to GLMs, each component of a GAM only depends on one feature.
- EBMs can also detect important *pairwise feature interactions*. Thus, EBMs extend the GAMs to include components of two features.
- EBMs use a *cyclic* training approach, where a very large number of base estimators are trained by repeated passes through all the features. This approach is also *parallelizable*, which makes EBMs an efficient training approach.

In the next two sections, we will see how EBMs work conceptually, and how we can train and use them in practice.

### 9.5.1 Explainable Boosting Machines (EBMs)

EBMs have two key components: they are generalized additive models with feature interactions. This allows the model representation to be broken down into smaller components which allow for better interpretation.

### GENERALIZED ADDITIVE MODELS WITH FEATURE INTERACTIONS

We are familiar with the concept of the generalized linear model (GLM), which use link functions  $g(y)$  to relate targets to linear models over features:

$$g(y) = \beta_0 + \beta_1 x_1 + \dots + \beta_d x_d.$$

Each component of the GLM  $\beta_j x_j$  only depends on one feature  $x_j$ . The generalized additive model (GAM) further extends this to be nonlinear:

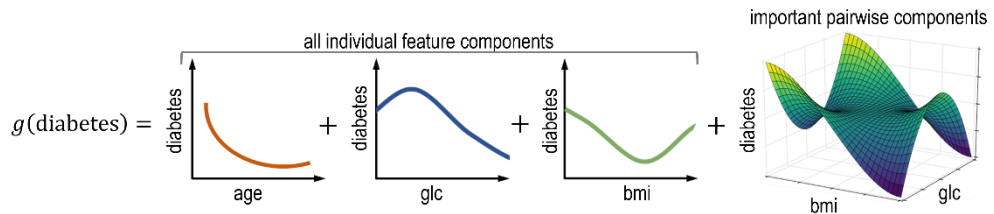
$$g(y) = \beta_0 + f_1(x_1) + \dots + f_d(x_d).$$

As with the GLM, each component of a GAM  $f(x_j)$  also depends on only one feature  $x_j$ . Keep in mind that both GLMs and GAMs can be viewed as ensembles, with each component of the ensemble depending on only one feature! This has important implications for training.

Explainable boosting machines further extend GAMs to include pairwise components as well. However, since the number of feature pairs can be very large, EBMs only include a small number of important feature pairs:

$$g(y) = \beta_0 + \underbrace{f_1(x_1) + \dots + f_d(x_d)}_{\text{all individual features}} + \overbrace{f_{ab}(x_a, x_b) + \dots + f_{uv}(x_u, x_v)}^{\text{important feature pairs}}.$$

This is also visualized in Figure 9.21 for the diabetes diagnosis problem from earlier, but with three variables: `age`, `blood glucose level (glc)` and `body mass index (bmi)`. This example EBM contains components for all three features individually, and one pairwise component.



**Figure 9.21.** An explainable boosting machine is a generalized additive model consisting of nonlinear components that depend on only one feature and nonlinear components that depend on pairs of features. This example shows an EBM for diabetes diagnosis dependent on three variables: `age`, `glc` and `bmi`. Though there are 3 pairs of variables (`age-glc`, `glc-bmi`, `age-bmi`), this EBM includes only one of them that it has deemed significant. The explainable boosting model is also an ensemble.

Since each component is a function of only one or two variables, once learned, we can immediately visualize the dependence between each variable or pair of variables and the target.

In addition, the EBM avoids incorporating all pairwise components, and only selects the most impactful ones. This avoids model bloat and improves explainability. By carefully choosing

the structure of the EBM, we can learn an explainable ensemble, which makes this a glass-box method.

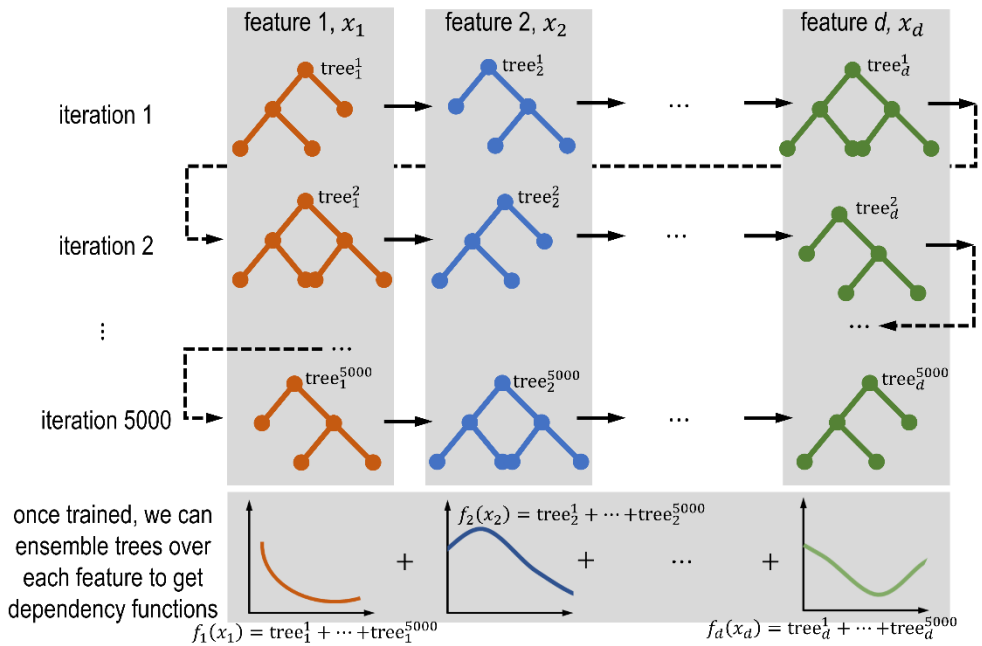
But what about model performance? Is it possible to train an EBM effectively to perform as well as existing ensemble methods?

### TRAINING EBMS

As with GLMs and GAMs, the EBM is also an ensemble of base components over individual features as well as feature pairs. Why is this important?

Because it allows us to train EBMs sequentially using simple modifications of our favorite ensemble learner: gradient boosting. EBMs are trained using a two-stage procedure:

- In the first stage, the EBM fits components for each feature  $f_j(x_j)$ . This is done through a cyclical and sequential training process over several thousand iterations, one feature at a time. In iteration  $t$ , for feature  $j$ , we fit a very shallow tree  $t_j$  using gradient boosting. Once we cycle through all the features within an iteration, we move on to the next iteration. This procedure is illustrated in Figure 9.22.
- The partially trained EBM  $g(y) \approx f_1(x_1) + \dots + f_d(x_d)$  is now frozen and used to evaluate and score all possible feature pairs  $(x_i, x_j)$ . This enables EBM to determine critically important feature interaction pairs  $(x_a, x_b) \dots (x_w, x_v)$  in the data. A small number of relevant feature pairs are selected.
- In the second stage, the EBM fits components for each feature pair  $f_{jk}(x_j, x_k)$  in a manner identical to the first stage. This produces a fully trained EBM:  $g(y) \approx f_1(x_1) + \dots + f_d(x_d) + f_{ab}(x_a, x_b) + \dots + f_{uv}(x_u, x_v)$ .



**Figure 9.22.** The first stage of the training procedure for EBMs, where models for each feature are trained sequentially and cyclically, with one model per feature per iteration. The trees trained are shallow and the learning rate is very low. However, over a very large number of iterations, a sufficiently complex nonlinear model for each feature can be learned. A similar procedure is also followed for the second stage of training EBMs, where models for pairwise feature interactions are trained.

From Figure 9.22, we can see that each individual component  $f_j(x_j)$  is actually an ensemble of thousands of shallow trees:

$$f_j(x_j) \approx \text{tree}_j^1(x_j) + \dots + \text{tree}_j^{5000}(x_j),$$

and similarly, each feature interaction component is also an ensemble:

$$f_{jk}(x_{jk}) \approx \text{tree}_{jk}^1(x_j, x_k) + \dots + \text{tree}_{jk}^{5000}(x_j, x_k).$$

So how exactly is this EBM a glass box? In two ways!

- *Local interpretability:* For a classification problem, given a specific example we want to explain  $x$ , we can get the log-odds of prediction from the EBM as:  $f_1(x_1) + \dots + f_d(x_d) + f_{ab}(x_a, x_b) + \dots + f_{uv}(x_u, x_v)$ . By construction, the EBM is already a fully decomposed and additive model, allowing us to simply grab the contribution of each feature  $f_j(x_j)$  or feature pair  $f_{jk}(x_j, x_k)$ . For regression, we can get the contribution to the overall regression value similarly. In both cases, there is no additional procedure like LIME or

SHAP and there is no need to approximate using linear models!

- *Global interpretability*: Since we have each component  $f_j(x_j)$  or  $f_{jk}(x_j, x_k)$ , we can also plot this over the feature ranges of  $x_j$  and/or  $x_k$ . This will produce a *dependency plot* for the features  $x_j$  and/or  $x_k$  over all possible values they can take. This tells us how the model behaves on aggregate.
- *Feature interactions*: Unlike SHAP or LIME, the model also inherently identifies key feature interactions, by design. This provides additional insights into model behavior and helps explain predictions better.

### 9.5.2 EBMs in Practice

Explainable Boosting Machines (EBMs) are available as part of the InterpretML package. In addition to EBMs, the InterpretML package also provides wrappers for LIME and SHAP, allowing us to use them in one framework. InterpretML also provides some nice functionalities for visualization.

In this section, though, we will only explore how to train, visualize, and interpret EBMs with InterpretML. InterpretML can be installed through pip and anaconda. The package's documentation page (<https://interpret.ml/>) contains additional information on how to use various glass-box and black-box models.

Listing 9.11 shows how we can train EBMs on the bank marketing data set. Like random forests and XGBoost models trained in Section 9.2, we will have to account for the class imbalance in the data. We do this by weighting positive examples by 5.0 and negative examples by 1.0 during training.

The listing also creates two visualizations: one for local explainability (of test example 3104) and another for global explainability (using feature importances and dependency plots).



**Listing 9.11. Training and visualizing EBMs using InterpretML**

```

from interpret.glassbox import ExplainableBoostingClassifier

wts = np.full_like(ytrn, fill_value=1.0)
wts[ytrn > 0] = 5.0      #A

feature_names = list(Xtrn.columns)
feature_types = np.full_like(feature_names, fill_value='continuous')
cat_features = ['default', 'housing', 'loan', 'contact', 'poutcome',
                'job', 'marital', 'education', 'month', 'day_of_week']
feature_types = ['categorical' if f in cat_features else 'continuous'
                 for f in feature_names]      #B

ebm = ExplainableBoostingClassifier()
ebm.fit(Xtrn, ytrn, sample_weight=wts)      #C

from interpret import set_visualize_provider      #D
from interpret.provider import InlineProvider
set_visualize_provider(InlineProvider())

from interpret import show      #E
x = Xtst.iloc[3104, :].values.reshape(1, -1)
y = ytst[3104].astype(float).reshape((1, 1))

local_explainer = ebm.explain_local(x, y)      #F
show(local_explainer)

ebm_global = ebm.explain_global()      #G
show(ebm_global)

```

**#A** Weight examples 1:5 to account for class imbalance

**#B** Identify the feature type for EBM: categorical, continuous

**#C** Initialize and train an EBM with these weights

**#D** Initialize the InterpretML visualizers

**#E** Test example 3104 will be explained

**#F** Local explanations (for test example 3104)

**#G** Global explanations (feature importances and dependency plots)

`ExplainableBoostingClassifier` trains for 5000 rounds by default, with support for early stopping. `ExplainableBoostingClassifier` also limits the number of pairwise interactions to 10 (by default, though this can be set by the user). Since this data set has 19 feature, there will be 171 total pairwise interactions, of which the model will pick the top 10.

The trained EBM model has an overall accuracy of 86.69% and balanced accuracy 74.59%. The XGBoost model trained in Section 9.2 has an overall accuracy of 87.24% and balanced accuracy of 74.67%. The EBM model is pretty comparable to the XGBoost model! The key difference is that the XGBoost model is a black box, while the EBM is a glass box.

So, what can we get out of this glass box? Figure 9.23 shows the local explanations of test example 3104. The local explanations show how much each feature and feature interaction pair in the model contributes to the overall positive or negative prediction.

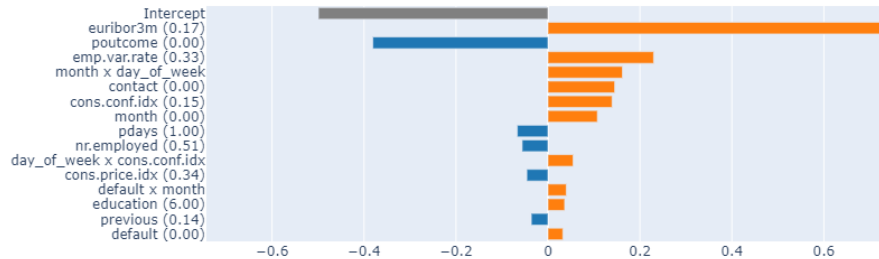
Test example 3104 is a positive example (i.e., `Sub?=YES`, meaning that the customer did subscribe to a fixed-term deposit account). The EBM model has correctly classified this example, with confidence (prediction probability) 66.1%.

This trained EBM uses three pairwise features to make a prediction for 3104: `month x day_of_week`, `day_of_week x cons.conf.idx`, `default x month`.

The highest pairwise feature interaction is `month x day_of_week`, which contributes a positive amount to the overall prediction. Contrast this to LIME and SHAP explanations of the XGBoost black box, which could only identify `month` since they do not support feature interactions explicitly. The EBM model is able to learn use a finer-grained feature and also explain its importance!

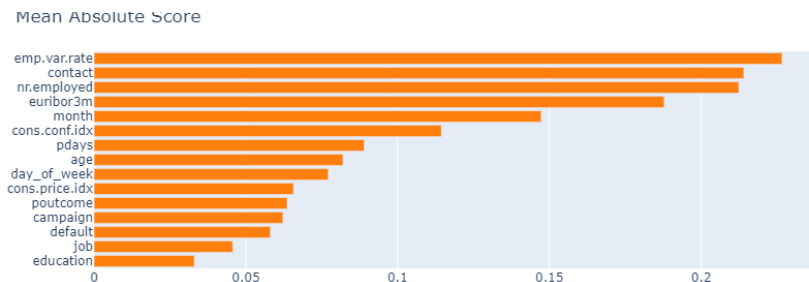
The takeaway here is that the EBM model is explicitly structured to incorporate feature interactions and to be able to explain them.

Predicted (1.0): 0.661 | Actual (1.0): 0.661



**Figure 9.23.** Local explainability of test example 3104, with individual features (such as `euribor3m` and `poutcome`) and pairwise features (such as `month x day_of_week`). The value of each EBM component and their contribution to the overall prediction (Sub? = YES) is shown.

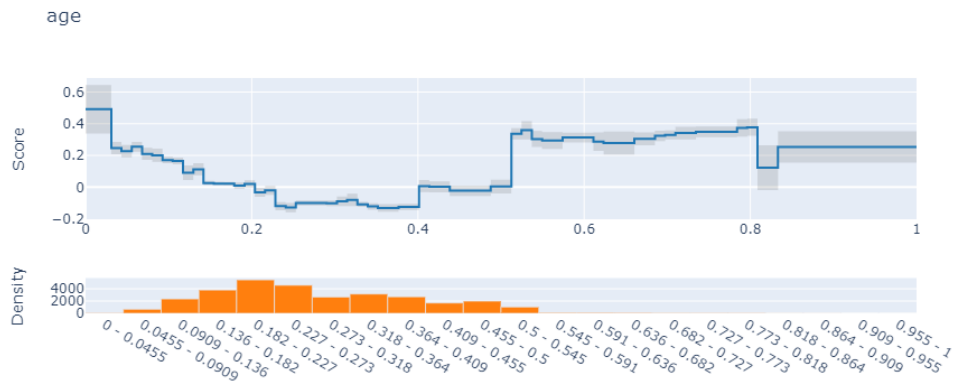
EBMs can also provide global interpretability in terms of feature importances. The overall importance is obtained by averaging over (the absolute values) of individual feature importances over the entire training set.



**Figure 9.24.** Global explainability of the trained EBM model, showing feature importance scores.

The overall model contains 30 components: 19 individual feature components, 10 pairwise feature components and 1 intercept. The top 15 feature and pairwise feature importances are visualized in Figure 9.24. These results are in general agreement with previous feature importance measures computed using other methods such as SHAP and LIME.

Finally, we can also obtain dependency plots directly from the EBM (as described in Figure 9.22). Figure 9.25 shows the dependency plot for age and how it influences whether someone will subscribe to a fixed-deposit account.



**Figure 9.25.** Dependency plot for age. The x-axis bins representing age, scaled to the range 0-1 during pre-processing. The raw ages are in the range 17-98. Scores are negative for people in the range 0.2-0.4, which corresponds to ages 33-49. This suggests that, absent any other information, people in this age range are typically not likely to subscribe to a fixed-deposit account.

## 9.6 Summary

- Blackbox models are typically challenging to understand owing to their complexity. The predictions of such models require specialized tools to be *explainable*. Glass-box models are more intuitive and easier to understand. The structure of such models makes them inherently *interpretable*.
- Most ensemble methods are typically black-box methods.
- Global methods attempt to generally explain a model's overall decision-making process, and what factors are broadly relevant. Local methods attempt to specifically explain a model's decision-making process with respect to individual examples and predictions.
- Feature importance is an interpretability method that assigns score to features based on their contribution to correct prediction of a target variable.
- Decision trees are commonly used glass box models and can be expressed as a set of decision rules, which are easily interpretable by humans.
- The interpretability of decision trees depends on their complexity (depth and number of leaf nodes). More complex trees are less intuitive and harder to understand.
- Generalized linear models are another commonly used glass box model. Their feature

weights can be interpreted as feature importances as they determine how much each feature contributes to the overall decision.

- Permutation feature importance is a black-box method for global interpretability. It tries to estimate how the model's predictive performance changes from before to after we shuffle/permute features.
- Partial dependence plots are another black-box method for global interpretability. Partial dependences are identified using marginalization or summing out other variables.
- Surrogate models are often used to mimic or approximate the behavior of a black-box model. Surrogate models are glass boxes and inherently explainable.
- Global surrogate models such as decision trees train models to optimize the fidelity-interpretability tradeoff.
- Locally Interpretable Model-Agnostic Explanation, or LIME is a local surrogate model that trains a linear model in the neighborhood of the example we want to explain.
- LIME also optimizes the fidelity-interpretability tradeoff and does so with a surrogate training set generated by perturbing features in the local neighborhood of the example to be explained.
- Shapley values are a tool that allows us to attribute the overall contribution of individual features (feature importances) by considering their contributions across all possible combinations of features.
- Shapley values are infeasible to compute directly for real world data sets with many features and examples.
- SHapley Additive exPlanations, or SHAP is a local surrogate model that trains a local linear model to approximate Shapley values.
- For tree-based models, a specially designed variant called TreeSHAP is used to compute the Shapley values efficiently.
- Shapley values and SHAP both have the additivity property, which allows us to aggregate Shapley values when ensembling individual models.
- One drawback of LIME and SHAP is that they are fundamentally designed only to compute and evaluate individual feature importances, and not feature interactions.
- Explainable Boosting Machines are a type of glass-box model and can be used for both global explainability and local explainability of individual examples.
- EBMs learn a fully factorized model, that is, the model components only depend on individual features or pairs of features. These components provide interpretability directly and EBMs need no additional computations (like SHAP or LIME) to generate explanations.
- EBMs are a type of generalized additive model (GAM), which are nonlinear extensions of generalized linear models (GLMs).
- EBMs can also detect important pairwise feature interactions. Thus, EBMs extend the GAMs to include components of two features.
- EBMs use a cyclic training approach, where a very large number of base estimators are trained by repeated passes through all the features. This approach is also parallelizable, which makes EBMs an efficient training approach.