5

Decision Trees and Random Forests

Learning Objectives

By the end of this chapter, you will be able to:

- Train a decision tree model in scikit-learn
- Use Graphviz to visualize a trained decision tree model
- Formulate the cost functions used to split nodes in a decision tree
- Perform a hyperparameter grid search using cross-validation with scikit-learn functions
- Train a random forest model in scikit-learn
- Evaluate the most important features in a random forest model

This chapter introduces decision trees and random forests in scikit-learn in addition to describing the method to perform hyperparameter grid search.

Introduction

In the last two chapters, we have gained a thorough understanding of the workings of logistic regression. We have also gotten a lot of experience with using the scikit-learn package in Python to create logistic regression models.

In this chapter, we will introduce a powerful type of predictive model that takes a completely different approach from the logistic regression model: **decision trees**. The concept of using a tree process to make decisions is simple, and therefore, decision tree models are easy to interpret. However, a common criticism of decision trees is that they overfit the training data. In order to remedy this issue, researchers have developed **ensemble methods**, such as **random forests**, that combine many decision trees to work together and make better predictions than any individual tree could.

We will see that decision trees and random forests can improve the quality of our predictive modeling of the case study data beyond what we achieved so far with logistic regression.

Decision trees

Decision trees and the machine learning models that are based on them, in particular **random forests** and **gradient boosted trees**, are fundamentally different types of models than generalized linear models, such as logistic regression. GLMs are rooted in the theories of classical statistics, which have a long history. The mathematics behind linear regression were originally developed at the beginning of the 19th century, by Legendre and Gauss. Because of this, the normal distribution is also called the Gaussian.

In contrast, while the idea of using a tree process to make decisions is relatively simple, the popularity of decision trees as mathematical models has come about more recently. The mathematical procedures that we currently use for formulating decision trees in the context of predictive modeling were published in the 1980s. The reason for this more recent development is that the methods used to grow decision trees rely on computational power – that is, the ability to crunch a lot of numbers quickly. We take such capabilities for granted nowadays, but they weren't widely available until more recently in the history of mathematics.

So, what is meant by a decision tree? We can illustrate the basic concept using a practical example. Imagine that you are considering whether or not to venture outdoors on a certain day. The only information you will base your decision on involves the weather and, in particular, whether the sun is shining and how warm it is. If it is sunny, your tolerance for cool temperatures is increased, and you will go outside if the temperature is at least 10 °C.

However, if it's cloudy, you require somewhat warmer temperatures and will only go outside if the temperature is 15 °C or more. Your decision-making process could be represented by the following tree:

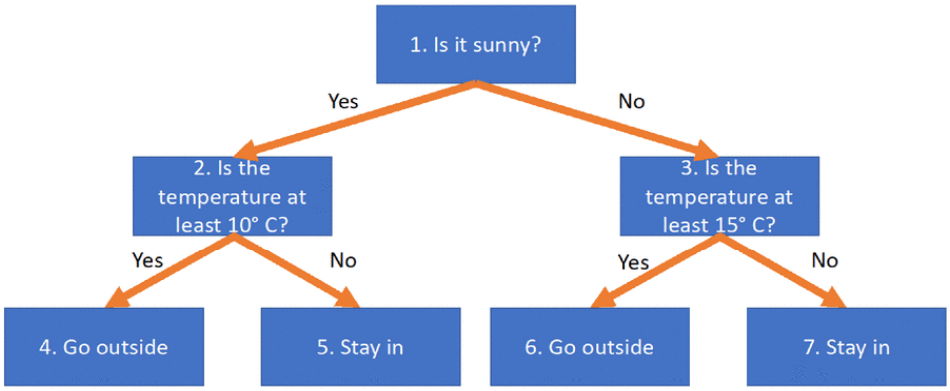

Figure 5.1: A decision tree for deciding whether to go outside given the weather

As you can see, decision trees have an intuitive structure and mimic the way that logical decisions might be made by humans. Therefore, they are a highly **interpretable** type of mathematical model, which can be a particularly desirable property depending on the audience. For example, the client for a data science project may be especially interested in a clear understanding of how a model works. Decision trees are a good way to deliver on this requirement, as long as their performance is sufficient.

The Terminology of Decision Trees and Connections to Machine Learning

Looking at the tree in *Figure* 5.1, we can begin to become familiar with some of the terminology of decision trees. Because there are two levels of decisions being made, based on cloud conditions at the first level, and temperature at the second level, we say that this decision tree has a **depth** of two. Here, both **nodes** at the second level are temperature-based decisions, but the kinds of decisions could be different within a level; for example, if we were to base our decision on whether or not it was raining, in the case that it was not sunny.

In the context of machine learning, the quantities that are used to make decisions at the nodes (in other words, to **split** the nodes) are the features. The features in the example in *Figure* 5.1 are a binary categorical feature for whether it's sunny, and a continuous feature of temperature. While we have only illustrated each feature being used once in a given branch of the tree, the same feature could be used multiple times in a branch. For example, if we were to go outside on a sunny day of at least 10 °C, but not if it were more than 40 °C – that's too hot! In this case, node 4 of *Figure* 5.1 would be split on the condition "Is the temperature at least 40 °C?", where "stay in" is the outcome if the answer is "yes", but "go outside" is the outcome if the answer is "no", meaning that the temperature is between 10 °C and 40 °C. Decision trees are, therefore, able to capture non-linear effects of the features, as opposed to a linear relationship which would assume that the hotter it was, the more likely we would be to go outside.

Consider the way that trees are typically represented, such as in *Figure* 5.1. The branches grow downward based on the binary decisions that can split the nodes into two more nodes. These binary decisions can be thought of as "if, then" rules. That is, if a certain criterion is met, do this, otherwise, do something else. The decision being made in our example tree is analogous to the concept of the response variable in machine learning. If we made a decision tree for the case study problem of credit default, the decisions would instead be predictions of the binary response values, which are "this account defaults" or "this account doesn't default". A tree that answers a binary yes/no type of question is a **classification tree**. However, decision trees are quite versatile and can also be used for multiclass classification and regression.

The terminal nodes at the bottom of the tree are called **leaves** or leaf nodes. In our example, the leaves are the final decisions of whether to go outside or stay in. There are four leaves on our tree, although you can imagine that if the tree only had a depth of one, where we made our decision based only on cloud conditions, there would be two leaves; and nodes 2 and 3 in *Figure* 5.1 would be leaf nodes with "go outside" and "stay in" as the decisions, for example.

In our example, every node at every level before the final level is split. This is not strictly necessary as you may go outside on any sunny day, regardless of the temperature. In this case, node 2 will not be split, so this branch of the tree will end on the first level with a "yes" decision. Your decision on cloudy days, however, may involve temperature, meaning this branch can extend to a further level. But, in the case that every node before the final level is split, consider how quickly the number of leaves grows with the number of levels.

Think about what would happen if we grew the decision tree in *Figure 5.1* down through an additional level, perhaps with a wind speed feature, to factor in a wind chill for the four combinations of cloud conditions and temperature. Each of the four nodes that are now leaves, nodes numbered from four to seven in *Figure 5.1*, would be split into two more leaf nodes, based on wind speed in each case. Then, there would be $4 \times 2 = 8$ leaf nodes. In general, it should be clear that in a tree with n levels, where every node before the final level is split, there will be 2^n leaf nodes. This is important to bear in mind as the **maximum depth** is one of the hyperparameters that you can set for a decision tree classifier in scikit-learn. We'll now explore this in the following exercise.

Exercise 19: A Decision Tree in scikit-learn

In this exercise, we will use the case study data to grow a decision tree, where we specify the maximum depth. We'll also use some handy functionality to visualize the decision tree. In order to do this, you'll need to run **`conda install graphviz`** and **`conda install python-graphviz`** at the command line to install the Python interface for Graphviz. Perform the following steps to complete the exercise:

> **Note**
>
> For Exercises 19 to 21 and Activity 5, the code and the resulting output have been loaded in a Jupyter Notebook that can be found at http://bit.ly/2GI7fJB. You can scroll to the appropriate section within the Jupyter Notebook to locate the exercise or activity of choice.

1. Start a new notebook for this chapter. To begin with, load all the packages that we've been using, and an additional one, **`graphviz`**, so that we can visualize decision trees:

```
import numpy as np #numerical computation
import pandas as pd #data wrangling
import matplotlib.pyplot as plt #plotting package
#Next line helps with rendering plots
%matplotlib inline
import matplotlib as mpl #add'l plotting functionality
mpl.rcParams['figure.dpi'] = 400 #high res figures
import graphviz #to visualize decision trees
```

2. Load the cleaned case study data:

```
df = pd.read_csv('../Data/Chapter_1_cleaned_data.csv')
```

> **Note**
>
> The location of the cleaned data may be different depending on where you saved it.

3. Get a list of column names of the **`DataFrame`**:

```
features_response = df.columns.tolist()
```

4. Make a list of columns to remove that aren't features or the response variable:

```
items_to_remove = ['ID', 'SEX', 'PAY_2', 'PAY_3', 'PAY_4', 'PAY_5',
'PAY_6',
                   'EDUCATION_CAT', 'graduate school', 'high school',
'none',
                   'others', 'university']
```

5. Use a list comprehension to remove these column names from our list of features and the response variable:

```
features_response = [item for item in features_response if item not in
items_to_remove]
features_response
```

This should output the list of features and the response variable:

```
['LIMIT_BAL',
 'EDUCATION',
 'MARRIAGE',
 'AGE',
 'PAY_1',
 'BILL_AMT1',
 'BILL_AMT2',
 'BILL_AMT3',
 'BILL_AMT4',
 'BILL_AMT5',
 'BILL_AMT6',
 'PAY_AMT1',
 'PAY_AMT2',
 'PAY_AMT3',
 'PAY_AMT4',
 'PAY_AMT5',
 'PAY_AMT6',
 'default payment next month']
```

Figure 5.2: A list of the feature and response variable names

Now the features are prepared for our usage. Next, we will make some imports from scikit-learn. We want to make a train/test split, which we are already familiar with. We also want to import the decision tree class.

6. Run this code to make imports from scikit-learn:

```
from sklearn.model_selection import train_test_split
from sklearn import tree
```

The **tree** library is the library of decision tree-related classes in scikit-learn.

7. Split the data into training and testing, using the same random seed we have throughout the book for the train/test split:

```
X_train, X_test, y_train, y_test = \
train_test_split(df[features_response[:-1]].values, df['default payment
next month'].values,
test_size=0.2, random_state=24)
```

Here, we use all but the last element of the list to get the names of the features, but not the response variable: **features_response[:-1]**. We use this to select columns from the **DataFrame**, and then retrieve their values using the **.values** method. We also do something similar for the response variable but directly specify the column name. In making the train/test split, we've used the same random seed as in previous work, and the same split size. This way, we can directly compare the work we will do in this chapter, with previous results. In particular, we have reserved the same "unseen test set" from the model development process so far.

Now we are ready to instantiate the decision tree class.

8. Instantiate the decision tree class by specifying the **max_depth** parameter to be 2:

```
dt = tree.DecisionTreeClassifier(max_depth=2)
```

We have used the **DecisionTreeClassifier** class because we have a classification problem. Since we specified **max_depth=2**, when we grow the decision tree using the case study data the tree will grow to a depth of at most 2. Let's now train this model.

9. Use this code to fit the decision tree model and grow the tree:

```
dt.fit(X_train, y_train)
```

This should display the following output:

```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=2,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best')
```

Figure 5.3: The output of fitting a decision tree classifier

In the output from model fitting, we can see all the options that are possible to set with a decision tree, most of which we have left at their default settings – we will discuss all of these shortly. For now, we have fit this decision tree model, so we can use the **graphviz** package to display a graphical representation of the tree.

10. Export the trained model in a format that can be read by the **graphviz** package using this code:

```
dot_data = tree.export_graphviz(dt, out_file=None, filled=True,
                                rounded=True, feature_names=features_
response[:-1],
                                proportion=True, class_names=['Not
defaulted', 'Defaulted'])
```

Here, we've specified a number of options to the **.export_graphviz** method. First, we need to say which trained model we'd like to graph, which we've got in the **dt** object. Next, we say we don't want an output file: **out_file=None**. Instead, we provide the **dot_data** variable to hold the output of this method. The rest of the options are used as follows:

filled=True: Each node will be filled with a color.

rounded=True: The nodes will appear with rounded edges as opposed to rectangles.

feature_names=features_response[:-1]: The names of the features from our list will be used as opposed to generic names such as **X[0]**.

proportion=True: The proportion of samples in each node will be displayed (we'll discuss this more later).

class_names=['Not defaulted', 'Defaulted']: The name of the predicted class will be displayed for each node.

What is the output of this method?

If you examine the contents of **dot_data**, you will see that it is a long text string. The **graphviz** package can interpret this text string to create a visualization.

11. Use the **.Source** method of the **graphviz** package to create an image from **dot_data** and display it:

```
graph = graphviz.Source(dot_data)
graph
```

The output should look like this:

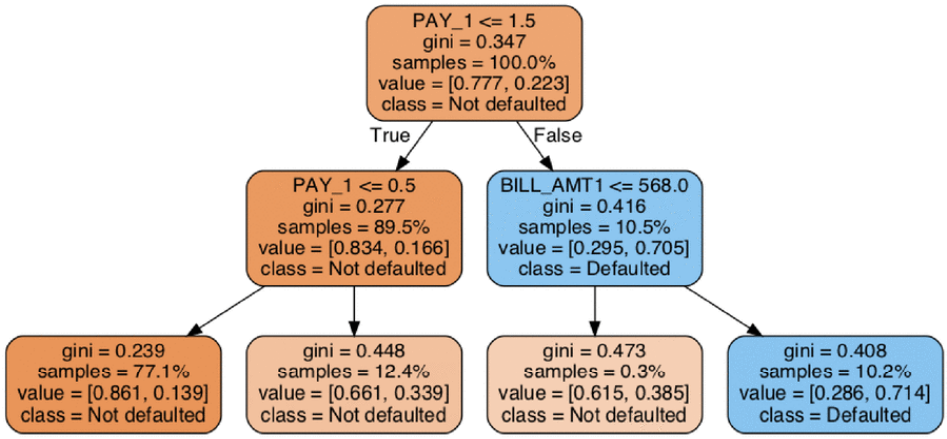

Figure 5.4: A decision tree plot from graphviz

The graphical representation of the decision tree in *Figure* 5.4 should be rendered directly in your Jupyter notebook.

> **Note**
>
> Alternatively, you could save the output of `.export_graphviz` to disk by providing a file path to the `out_file` keyword argument. To turn this output file into an image file, for example, a `.png` file that you could use in a presentation, you could run this code at the command line, substituting in the filenames as appropriate: `$ dot -Tpng <exported_file_name> -o <image_file_name_you_want>.png`
>
> For further details on the options to `.export_graphviz` you should consult the scikit-learn documentation (https://scikit-learn.org/stable/modules/generated/sklearn.tree.export_graphviz.html).

The visualization in *Figure* 5.4 contains a lot of information about how the decision tree was trained, and how it can be used to make predictions. We will discuss the training process in more detail later, but suffice to say that training a decision tree works by starting with all the training samples in the initial node at the top of the tree, and then splitting these into two groups based on a **threshold** in one of the features. The cut point is represented by a Boolean condition in the top `PAY_1 <= 1.5` node.

All the samples where the value of the `PAY_1` feature is less than or equal to the cut point of 1.5, will be represented as `True` under the Boolean condition. As shown in *Figure* 5.4, these samples get sorted into the left side of the tree, following the arrow that says "True" next to it.

As you can see in the graph, each node that is split contains the splitting criteria on the first line of text. The next line relates to "gini", which we will discuss shortly.

On the next line after the gini line, there is information about the proportion of samples in each node. In the top node, we are starting with all the samples ("samples = 100.0%"). Following the first split, 89.5% of the samples get sorted into the node on the left, while the remaining 10.5% go into the node on the right. This information is shown directly in the visualization and reflects how the training data was used to create the tree. Let's confirm this by examining the training data.

12. To confirm the proportion of training samples where the **PAY_1** feature is less than or equal to 1.5, first identify the index of this feature in the list of **features_response[:-1]** feature names:

    ```
    features_response[:-1].index('PAY_1')
    ```

    This code should output the following:

    ```
    4
    ```

    Figure 5.5: Index of the PAY_1 feature in the list of feature names

13. Now observe the shape of the training data:

    ```
    X_train.shape
    ```

    This should give you the following output:

    ```
    (21331, 17)
    ```

    Figure 5.6: The shape of the training data

    To confirm the fraction of samples after the first split of the decision tree, we need to know the proportion of samples, where the **PAY_1** feature meets the Boolean condition, that was used to make this split. To do this, we can use the index of the **PAY_1** feature in the training data, corresponding to the index in the list of feature names, and the number of samples in the training data, which is the number of rows we observed from **.shape**.

14. Use this code to confirm the proportion of samples after the first split of the decision tree:

    ```
    sum(X_train[:,4] <= 1.5)/X_train.shape[0]
    ```

    The output should be as follows:

    ```
    0.8946134733486475
    ```

    Figure 5.7: The proportion of training samples where PAY_1 >= 1.5

    By applying a logical condition to the column of the training data corresponding to the **PAY_1** feature, and then taking the sum of this, we calculated the number of samples meeting this condition. Then, by dividing by the total number of samples, we converted this to a proportion. We can see that the proportion we directly calculated from the training data is equal to the proportion displayed in the left node after the first split in *Figure* 5.4.

After the first split, the samples contained in each of the two nodes on the first level are split again. As further splits are made beyond the first split, smaller and smaller proportions of the training data will be assigned to any given node in the subsequent levels of a branch, as can be seen in *Figure* 5.4.

Now we want to interpret the remaining lines of text in the nodes in *Figure* 5.4. The lines starting with "value" give the class fractions of the response variable for the samples contained in each node. For example, in the top node, we see "value = [0.777, 0.223]". This is simply the class fraction for the overall training set, which you can confirm in the following step.

15. Calculate the class fraction in the training set with this code:

```
np.mean(y_train)
```

The output should be as follows:

```
0.223102526838873
```

Figure 5.8: The class fraction of positive samples in the training data

This is equal to the second member of the pair of numbers following "value" in the top node; the first number is just one minus this, in other words, the fraction of negative training samples. In each subsequent node, the class fraction of the samples that are contained in just that node are displayed. The class fractions are also how the nodes are colored: those with a higher proportion of the negative class than the positive class are orange, with darker orange signifying higher proportions, while those with a higher proportion of the positive class have a similar scheme using a blue color.

Finally, the line starting with "class" indicates how the decision tree will make predictions from a given node, if that node were a leaf node. Decision trees for classification make predictions by determining which leaf node a sample will be sorted in to, given the values of the features, and then predicting the class of the majority of the training samples in that leaf node. This strategy means that the class proportions in each node are the necessary information that is needed to make a prediction.

For example, if we've made no splits and we are forced to make a prediction knowing nothing but the class fractions for the overall training data, we will simply choose the majority class. Since most people don't default, the class on the top node is "Not defaulted." However, the class fractions in the nodes of deeper levels are different, leading to different predictions. We'll discuss the training process in the following section.

Importance of max_depth

Recall that the only hyperparameter we specified in this exercise was **max_depth**, that is, the maximum depth to which the decision tree can be grown during the model training process. It turns out that this is one of the most important hyperparameters. Without placing a limit on the depth, the tree will be grown until one of the other limitations, specified by other hyperparameters, takes effect. This can lead to very deep trees, with very many nodes. For example, consider how many leaf nodes there could be in a tree with a depth of 20. This would be 2^{20} leaf nodes, which is over 1 million! Do we even have 1 million training samples to sort into all these nodes? In this case, we do not. It would clearly be impossible to grow such a tree, with every node before the final level being split, using this training data. However, if we remove the **max_depth** limit and rerun the model training of this exercise, observe the effect:

Figure 5.9: A portion of the decision tree grown with no maximum depth

Here, we have shown a portion of the decision tree that is grown with the default options, which include **max_depth**=None, meaning no limitation on the depth of the tree. The entire tree is about twice as wide as the portion shown here. There are so many nodes that they only appear as very small orange or blue patches; the exact interpretation of each node is not important as we are just trying to illustrate how large trees can potentially be. It should be clear that without hyperparameters to govern the tree-growing process, extremely large and complex trees may result.

Training Decision Trees: Node Impurity

So far, we have treated the decision tree training process as a black box. At this point, you should have an understanding of how a decision tree makes predictions using features, and the class fractions of training samples in the leaf nodes.

But how are the splits decided during the training process?

Given that the method of prediction is to take the majority class of a leaf node, it makes intuitive sense that we'd like to find leaf nodes that are primarily from one class or the other. In the perfect case, the training data can be split so that every leaf node contains entirely positive or entirely negative samples. Then, we will have a high level of confidence that a new sample, once sorted into one of these nodes, will be either positive or negative. In practice, this rarely, if ever, happens. However, this illustrates the goal of training decision trees – that is, to make splits so that the next two nodes after the split have a higher **purity**, or, in other words, are closer to containing either only positive or only negative samples.

In practice, decision trees are actually trained using the inverse of purity, or **node impurity**. This is some measure of how far the node is from having 100% of the training samples belonging to one class and is analogous to the concept of a cost function, which signifies how far a given solution is from a theoretical perfect solution. The most intuitive concept of node impurity is the misclassification rate. Adopting a widely-used notation (for example, https://scikit-learn.org/stable/modules/tree.html) for the proportion of samples in each node belonging to a certain class, we can define p_{mk} as the proportion of samples belonging to the k^{th} class in the m^{th} node. In a binary classification problem, there are only two classes: $k = 0$ and $k = 1$. For a given node m, the **misclassification rate** is simply the proportion of the less common class in that node, since all these samples will be misclassified when the majority class in that node is taken as the prediction.

Let's visualize the misclassification rate as a way to start thinking about how decision trees are trained. Programmatically, we consider possible class fractions, p_{m0}, between 0.01 and 0.99 of the negative class, $k = 0$, in a node, m, using NumPy's **`linspace`** function:

```
pm0 = np.linspace(0.01,0.99,99)
```

Then, the fraction of the positive class for this node is simply the rest of the samples:

$$p_{m1} = 1 - p_{m0}$$

Figure 5.10: Equation to calculate the fraction of positive class for node m0

Now, the misclassification rate for this node will be whatever the smaller class fraction is, between **pm0** and **pm1**. We can find the smaller of the corresponding elements between two arrays with the same shape in NumPy by using the minimum function:

```
misclassification_rate = np.minimum(pm0, pm1)
```

What does the misclassification rate look like plotted against the possible class fractions of the negative class?

We can plot this using the following code:

```
mpl.rcParams['figure.dpi'] = 400
plt.plot(pm0, misclassification_rate, label='Misclassification rate')
plt.xlabel('$p_{m0}$')
plt.legend()
```

You should obtain this graph:

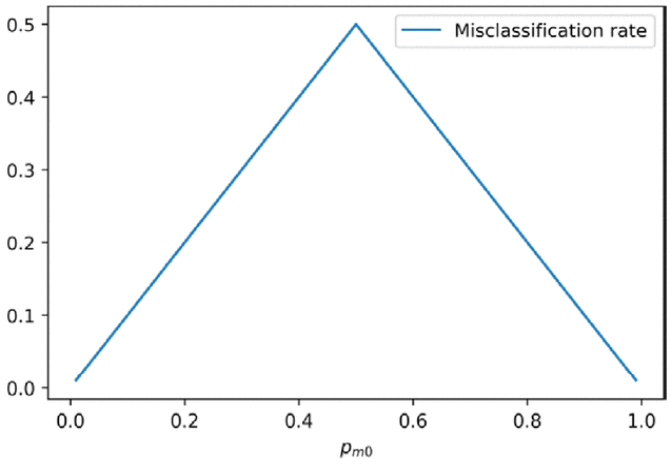

Figure 5.11: The misclassification rate for a node

Now, it's clear that the closer the class fraction of the negative class, p_{m0}, is to 0 or 1, the lower the misclassification rate will be. How is this information used when growing decision trees? Consider the process that might be followed.

Every time a node is split when growing a decision tree, two new nodes are created. Since the prediction from either of these new nodes is simply the majority class, an important goal will be to reduce the misclassification rate. Therefore, we will want to find a feature, from all the possible features, and a value of this feature at which to make a cut point, so that the misclassification rate in the two new nodes will be as low as possible when averaging over all the classes. This is very close to the actual process that is used to train decision trees.

Continuing for the moment with the idea of minimizing the misclassification rate, the decision tree training algorithm goes about node splitting by considering all the features. Although, the algorithm may possibly only consider a randomly-selected subset if you set the **`max_features`** hyperparameter to anything less than the total number of features. We'll discuss possible reasons for doing this later. In either case, the algorithm then considers each possible threshold for every candidate feature and chooses the one that results in the lowest impurity, calculated as the average impurity across the two possible new nodes, weighted by the number of samples in each node. The node splitting process is shown in *Figure 5.12*. This process is repeated until a stopping criterion of the tree, such as **`max_depth`**, is reached:

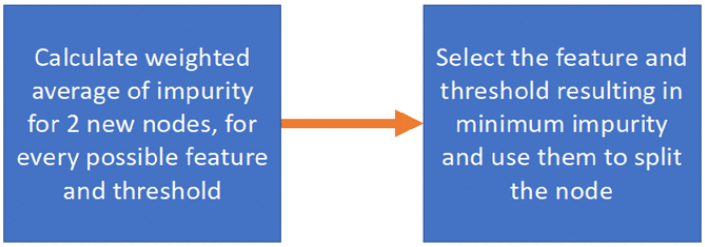

Figure 5.12: How to select a feature and threshold in order to split a node

While the misclassification rate is an intuitive measure of impurity, it happens that there are better measures that can be used to find splits during the model training process. The two options that are available in scikit-learn for the impurity calculation, which you can specify with the **`criterion`** keyword argument, are the **Gini impurity** and the **cross-entropy** options. Here, we will describe these mathematically and show how they compare with the misclassification rate.

The Gini impurity is calculated for a node m using the following formula:

$$\text{Gini} = \sum_{k} p_{mk}\,(1 - p_{mk})$$

Figure 5.13: Equation to calculate Gini impurity

Here, the summation is taken over all classes. In the case of a binary classification problem, there are only two classes, and we can write this programmatically as follows:

```
gini = (pm0*(1-pm0)) + (pm1*(1-pm1))
```

Cross entropy is calculated using this formula:

$$\text{cross entropy} = -\sum_{k} p_{mk} \log\left(p_{mk}\right)$$

Figure 5.14: Equation to calculate cross entropy

Using this code, we can calculate the cross entropy:

```
cross_ent = -1*((pm0*np.log(pm0)) + (mp1*np.log(mp1)))
```

In order to add the Gini impurity and cross entropy to our plot of misclassification rate and see how they compare, we just need to include the following lines of code, after we plot the misclassification rate:

```
plt.plot(pm0, gini, label='Gini impurity')
plt.plot(pm0, cross_ent, label='Cross entropy')
```

The final plot should appear as follows:

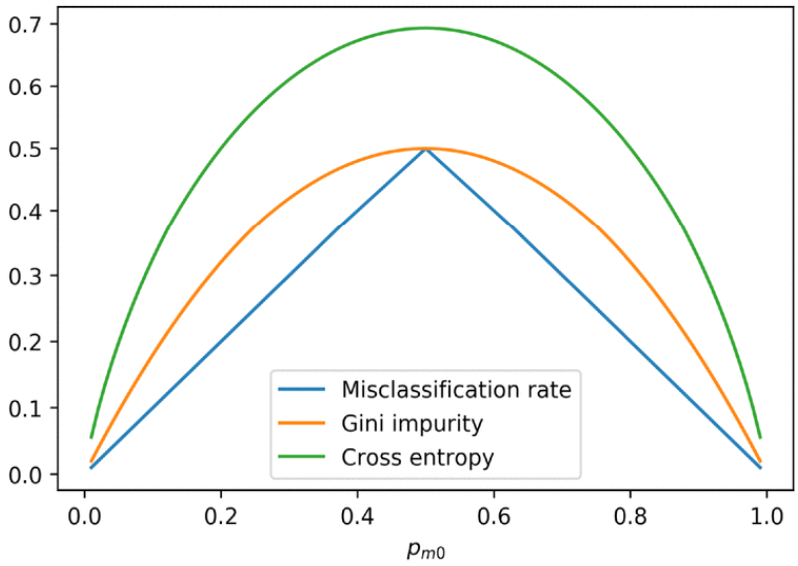

Figure 5.15: The misclassification rate, Gini impurity, and cross entropy

Like the misclassification rate, both the Gini impurity and the cross entropy are highest when the class fractions are equal at 0.5, and they decrease as the node becomes purer – in other words, when they contain a higher proportion of just one of the classes. However, the Gini impurity is somewhat steeper than the misclassification rate in certain regions of the class fraction, which enables it to more effectively find the best split. Cross-entropy looks yet steeper. So, which one is better for your work? This is the kind of question that does not have a concrete answer across all datasets. You should consider both impurity metrics in a cross-validation search for hyperparameters in order to determine the appropriate one. Note that in scikit-learn, Gini impurity can be specified with the `criterion` argument using the `'gini'` string, while cross entropy is just referred to as `'entropy'`.

Features Used for the First splits: Connections to Univariate Feature Selection and Interactions

We can begin to get an impression of how important various features are to decision tree models, based on the small tree shown in *Figure* 5.4. Notice that `PAY_1` was the feature chosen for the first split. This means that it was the best feature in terms of decreasing node impurity on the node containing all of the training samples. Recalling our experience with univariate feature selection in *Chapter* 3, *Details of Logistic Regression and Feature Exploration*, `PAY_1` was the top-selected feature from the `F-test`. So, the appearance of this feature in the first split of the decision tree makes sense given our previous analysis.

In the second level of the tree, there is another split on `PAY_1`, as well as a split on `BILL_AMT_1`. `BILL_AMT_1` was not listed among the top features in univariate feature selection. However, it may be that there is an important interaction between `BILL_AMT_1` and `PAY_1`, which could not be picked up by univariate methods. In particular, from the splits chosen by the decision tree, it seems that those accounts with both a value of 2 or greater for `PAY_1`, and a `BILL_AMT_1` of greater than 568, are especially at risk of default. This combined effect of `PAY_1` and `BILL_AMT_1` is an interaction and may also be why we were able to improve logistic regression performance by including interaction terms in the activity of the previous chapter.

Training Decision Trees: A Greedy Algorithm

There is no guarantee that a decision tree trained by the process described previously will be the best possible decision tree for finding leaf nodes with the lowest impurity. This is because the algorithm used to train decision trees is what is called a greedy algorithm. In this context, this means that, at each opportunity to split a node, the algorithm is looking for the best possible split at that point in time, without any regard to the fact that the opportunities for later splits are being affected.

For example, consider the following hypothetical scenario: the best initial split for the training data of the case study involves **`PAY_1`**, as we've seen in *Figure* 5.4. But what if we instead split on **`BILL_AMT_1`**, and then make subsequent splits on **`PAY_1`** in the next level. Will this mean that the impurity of the leaf nodes will be lower? Even though the initial split on **`BILL_AMT_1`** is not the best one available at first, the end result will be better if it is done this way. The algorithm has no way of finding solutions like this, since it only considers the best possible split at each node. Just to be clear, this is only a hypothetical scenario and we have no reason to suspect this of the case study data.

The reason why we still use the algorithm that we previously described is because it takes substantially longer to consider all possible splits in a way that enables finding the truly optimal tree. Despite this shortcoming of the decision tree training process, there are methods that you can use to reduce the possible harmful effects of the greedy algorithm. Instead of searching for the best split at each node, the **`splitter`** keyword argument to the decision tree class can be specified as **`random`** in order to choose a random feature to make a split on. However, the default is **`best`**, which searches all features for the best split. Another option, which we've already discussed, is to limit the number of features using the **`max_features`** keyword, which will be searched at each splitting opportunity. Finally, you can also use ensembles of decision trees such as random forests, which we will describe shortly. Note that all these options, in addition to possibly avoiding the ill effects of the greedy algorithm, are also options to address the overfitting that decisions trees are often criticized for.

Training Decision Trees: Different Stopping Criteria

We have already reviewed using the **`max_depth`** parameter as a limit to how deep a tree will grow. However, there are several other options available in scikit-learn as well. These are mainly related to how many samples are present in a leaf node, or how much the impurity can be decreased by further splitting nodes. As we have already mentioned, you may be limited by the size of your dataset in terms of how deep you can grow a tree. And it may not make sense to grow trees deeper, especially if the splitting process is no longer finding nodes with higher purity.

We summarize all of the keyword arguments that you can supply to the **`DecisionTreeClassifier`** class in scikit-learn here:

Parameter	Possible values	Notes
criterion	string, 'gini' or 'entropy'	This is the formula used to calculate node impurity.
splitter	string, 'best' or 'random'	This determines whether to search among all candidate features when making a split, or to choose one at random.
max_depth	int or None	A stopping criterion; None means there is no limit to the maximum depth for growing the tree, although the tree may be stopped for reasons specified in other hyperparameters. An int integer means to stop growing the tree after that many levels.
min_samples_split	int or float	A stopping criterion. If an integer, then a node must have at least this many samples in order to be split. If a float value, it is the fraction of the total number of samples that must be in a node to split it.

Parameter	Possible values	Notes
min_samples_leaf	int or float	A stopping criterion. Similar to min_samples_split, but this refers to the number of samples that will be in the nodes after the split. It refers to a node at any depth.
min_weight_fraction_leaf	float	Similar to min_samples_leaf, but uses the weight fraction of samples instead of the raw fraction of samples. It is useful only if sample weighting has been specified with the class_weight parameter.
max_features	int, float, string: 'auto', 'sqrt', 'log2', or None	This is a strategy for how many features to consider when trying to split a node. If None, all the features are considered. If an int or float, then that number or fraction of features are considered. 'auto' and 'sqrt' both use the square root of the number of features, and 'log2' uses the base 2 logarithm. If the number here is less than the total number of features, a random selection of features is taken. However, more than this number of features may be considered if none of the random selection meets other criteria, such as min_impurity_decrease.

Parameter	Possible values	Notes
random_state	int or None	If an integer is supplied, this will seed the random number generator for repeatable results between runs of the same code.
max_leaf_nodes	int or None	A stopping criterion. It allows the maximum number of leaf nodes if an integer is supplied, or no limit for None.
min_impurity_decrease	float	A stopping criterion. If larger than 0, the required decrease in impurity to split a node. This means the tree only keeps growing if the nodes are getting purer.
min_impurity_split	float	A stopping criterion but being phased out. Use min_impurity_decrease instead.

Parameter	Possible values	Notes
class_weight	dict, list of dicts, 'balanced' or None	If the response variable has imbalanced classes, 'balanced' will weight samples similar to the logistic regression. Here, this will be used to calculate the weighted average of node impurity. A dictionary can be used to manually supply class weights, for example, if you have a reason to be more confident in a portion of the data. The list of dictionaries can be used for multi-output problems, which is when there are multiple response variables (which is beyond the scope of this course). None does no sample weighting.
presort	bool	This determines whether to sort the samples before growing the tree, which may speed up training for smaller datasets.

Figure 5.16: The complete list of options for the decision tree classifier in Scikit-Learn

Using Decision Trees: Advantages and Predicted Probabilities

While decision trees are simple in concept, there are several practical advantages that they can present:

No need to scale features

Consider the reasons why we needed to scale features for logistic regression. One reason is that, for some of the solution algorithms based on gradient descent, it is necessary that features be on the same scale. Another is that when we are using L1 or L2 regularization to penalize coefficients, all the features must be on the same scale so they are penalized equally. With decision trees, the node splitting algorithm considers each feature individually and, therefore, it doesn't matter whether the features are on the same scale or not.

Non-linear relationships and interactions

Because each successive split in a decision tree is performed on a subset of the samples resulting from previous split(s), decision trees can describe complex non-linear relationships of a single feature, as well as interactions between features. Consider our previous discussion of why `BILL_AMT_1` was chosen at the second level of the tree in *Figure* 5.4. Also, as a hypothetical example with synthetic data, consider the following dataset for classification:

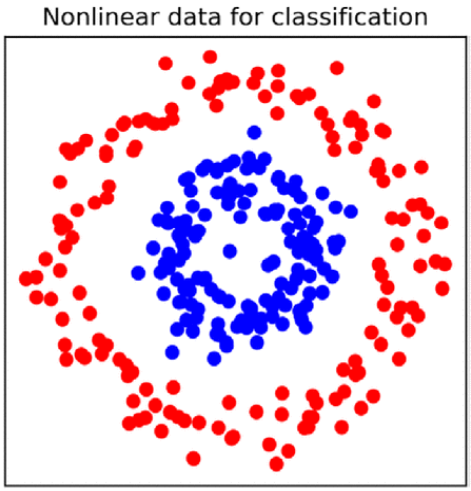

Figure 5.17: An example classification dataset, with the classes shown in red and blue.

We know from *Chapter* 3, *Details of Logistic Regression and Feature Exploration* that logistic regression has a linear decision boundary. So, how do you think logistic regression may cope with such a dataset as that shown in *Figure* 5.17? Where would you draw a line to separate the blue and red classes? It should be clear that without engineering additional features, a logistic regression is not likely to be a good classifier for these data. Now think about the set of "if, then" rules of a decision tree, which could be used with the features represented on the x and y axes of *Figure* 5.17. Do you think a decision tree will be effective with these data?

Here, we plot in the background the predicted probabilities of class membership using red and blue colors, for both of these models:

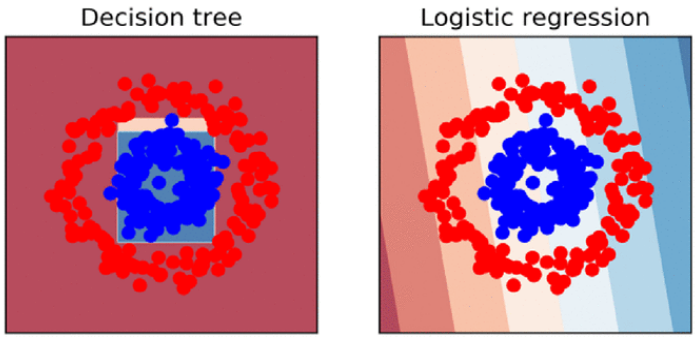

Figure 5.18: Decision tree and logistic regression predictions

In *Figure* 5.18, the predicted probabilities for both models are colored so that darker red corresponds to a higher predicted probability for the red class, while darker blue corresponds to a higher predicted probability for the blue class. We can see that the decision tree can isolate the blue class in the middle of the circle of red points. This is because, by using thresholds for the x and y coordinates in the node-splitting process, a decision tree can mathematically model the fact that the location of the blue and red classes depend on both the x and y coordinates together (interactions), and that the likelihood of either class is not a linearly increasing or decreasing function of x or y (non-linearities). Consequently, the decision tree approach is able to get most classifications right.

However, the logistic regression has a linear decision boundary, which will be the straight line between the lightest blue and red patches in the background. The logistic regression decision boundary goes right through the middle of the data and doesn't provide a useful classifier. This shows the power of decision trees "out of the box", without the need for engineering non-linear or interaction features.

Predicted probabilities

You may wonder how the predicted probabilities shown in *Figure 5.18* were obtained from the decision tree model. We know that logistic regression produces probabilities as raw output. However, a decision tree makes predictions based on the majority of class of the leaf nodes. So, where will this probability come from? In fact, decisions trees do offer the `.predict_proba` method in scikit-learn. The probability is based on the proportion of the majority class in the leaf node. If the leaf node consisted 75% of the positive class, for example, the prediction for that node will be the positive class and the predicted probability will be 0.75. The predicted probabilities from decision trees are not considered to be as statistically rigorous as those from generalized linear models, but they are still commonly used to measure the performance of models by methods that depend on varying the threshold for classification, such as the ROC curve or the precision-recall curve.

> **Note**
>
> We are focusing here on decision trees for classification because of the nature of the case study. However, decision trees can also be used for regression, making them a versatile method. The tree-growing process is similar for regression as it is for classification, except that instead of seeking to reduce node impurity, a regression tree seeks to minimize other metrics such as the mean squared error (MSE) or mean absolute error (MAE) of the predictions, where the prediction for a node may be the average or median of the samples in the node, respectively.

A More Convenient Approach to Cross-Validation

In *Chapter 4*, *The Bias-Variance Trade-off*, we gained a deep understanding of cross-validation by writing our own function to do it, using the `KFold` class to generate the training and testing indices. This was helpful to get a thorough understanding of how the process works. However, scikit-learn offers a convenient class that can do more of the heavy lifting for us: `GridSearchCV`. `GridSearchCV` can take as input a model that we want to find optimal hyperparameters for, such as a decision tree or a logistic regression, and a "grid" of hyperparameters that we want to perform cross-validation over. For example, in a logistic regression, we may want to get the average cross-validation score over all the folds for different values of the regularization parameter **C**. With decision trees, we may want to explore different depths of trees. You can also search multiple parameters at once, for example, if we wanted to try different depths of trees and different numbers of `max_features` to consider at each node split.

`GridSearchCV` does what is called an exhaustive grid search over all the possible combinations of parameters that we supply. This means that if we supplied five different values for each of the two hyperparameters, the cross-validation procedure would be run 5 x 5 = 25 times. If you are searching many values of many hyperparameters, the amount of cross-validation runs can grow very quickly. In these cases, you may wish to use `RandomizedSearchCV`, which searches a randomly selected number of hyperparameter combinations from the universe of all possibilities in the grid you supply.

`GridSearchCV` can speed up your work by streamlining the cross-validation process. You should be familiar with the concepts of cross-validation from the previous chapter, so we proceed directly to listing all the options available for `GridSearchCV`. In the following exercise, we will get hands-on practice using `GridSearchCV` with the case study data, to search hyperparameters for a decision tree classifier. Here are the options for `GridSearchCV`:

Parameter	Possible values	Notes
estimator	estimator object	This is a model object that you have instantiated from a model class. The hyperparameters will be updated as GridSearchCV does its work.
param_grid	dict or list of dicts	The dictionary has parameter names as keys and lists of parameters as values. These are the hyperparameter values for which you want to search all possible combinations. To do multiple grid searches, supply a list of dictionaries.
scoring	string	This represents the model assessment metric that you want to use to measure training and testing performance across the folds, for example, 'roc_auc'.
n_jobs	int or None	This determines the number of processing jobs to run in parallel. It may speed up cross-validation to run parallel jobs, but it is a good idea to experiment in order to be sure.

Parameter	Possible values	Notes
pre_dispatch	int or string	This determines the number of jobs or formula for the number of jobs to dispatch. It is relevant for parallel processing using n_jobs.
iid	bool	This indicates whether the data is independent and identically distributed (i.i.d.). True to compute a weighted average score across folds, using the number of samples in each fold for weighting. False to compute an unweighted average. It is not relevant if the number of samples is the same in each fold.
cv	int, cross-validation generator or iterable	If supplying an integer, this is the number of folds to use for cross validation.
refit	bool or string	After doing the cross-validation, the "best" hyperparameters according to the metric specified in scoring can be used directly with the fitted GridSearchCV object to make predictions. If refit=True, the model will be refit to all of the data (not just one of the folds) using the best hyperparameters. Use the string argument if multiple metrics are specified.

Parameter	Possible values	Notes
verbose	int	This controls how much output you will see from the cross-validation procedure.
error_score	'raise' or numeric	This determines what to do if an error happens during model fitting.
return_train_score	bool	This determines whether or not to compute and return training scores on the folds. It is not required for selecting the best hyperparameters based on testing fold scores and for some datasets and models, this can take substantially more time. However, it does give insight into possible overfitting.

Figure 5.19: The options for `GridSearchCV`

Exercise 20: Finding Optimal Hyperparameters for a Decision Tree

In this exercise, we will use `GridSearchCV` to tune the hyperparameters for a decision tree model. You will learn about a convenient way of searching different hyperparameters with scikit-learn. Perform the following steps to complete the exercise:

> **Note**
>
> The code and the resulting output for this exercise have been loaded in a Jupyter Notebook and can be found at http://bit.ly/2GI7fJB.

1. Import the **`GridSearchCV`** class with this code:

   ```
   from sklearn.model_selection import GridSearchCV
   ```

   The next step is to define the hyperparameters that we want to search using cross-validation. We will find the best maximum depth of tree, using the **`max_depth`** parameter. Deeper trees have more node splits, which partition the training set into smaller and smaller subspaces using the features. While we don't know the best maximum depth ahead of time, it is helpful to consider some limiting cases when considering the range of parameters to use for the grid search.

   We know that one is the minimum depth, consisting of a tree with just one split. As for the largest depth, you can consider how many samples you have in your training data, or more appropriately in this case, how many samples will be in the training fold for each split of the cross-validation. We will perform a 4-fold cross-validation like we did in the previous chapter. So, how many samples will be in each training fold and how does this relate to the depth of the tree?

2. Find the number of samples in the training data **using this code:**

   ```
   X_train.shape
   ```

   The output should be as follows:

   ```
   (21331, 17)
   ```

   Figure 5.20: The shape of the training data

   With 21,331 training samples and 4-fold cross-validation, there will be three-fourth of the samples, or about 16,000 samples, in each training fold.

   What does this mean for how deep we may wish to grow our tree?

   A theoretical limitation is that we need at least one sample in each leaf. From our discussion about how the depth of the tree relates to the number of leaves, we know a tree that splits at every node before the last level, with n levels, has 2^n leaf nodes. Therefore, a tree with L leaf nodes has a depth of approximately $\log_2(L)$. In the limiting case, if we grow the tree deep enough so that every leaf node has one training sample for a given fold, the depth will be $\log_2(16{,}000) \approx 14$. So, 14 is the theoretical limit to the depth of a tree that we could grow in this case.

Practically speaking, you will probably not want to grow a tree this deep, as the rules used to generate the decision tree will be very specific to the training data and the model is likely to be overfit. However, this gives you an idea of the range of values we may wish to consider for the **`max_depth`** hyperparameter. We will explore a range of depths from 1 up to 12.

3. Define a dictionary with the key being the hyperparameter name and the value being the list of values of this hyperparameter that we want to search in cross-validation:

```
params = {'max_depth':[1, 2, 4, 6, 8, 10, 12]}
```

In this case, we are only searching one hyperparameter. However, you could define a dictionary with multiple key-value pairs to search for multiple hyperparameters simultaneously.

Now we want to instantiate the **`GridSearchCV`** class.

4. Instantiate the **`GridSearchCV`** class using these options:

```
cv = GridSearchCV(dt, param_grid=params, scoring='roc_auc', fit_
params=None,
                  n_jobs=None, iid=False, refit=True, cv=4, verbose=1,
                  pre_dispatch=None, error_score=np.nan, return_train_
score=True)
```

Notice that we can reuse the decision tree object, **`dt`**, that we already instantiated earlier in this chapter. When creating **`dt`**, we used the default arguments for all options but **`max_depth`**. However, this hyperparameter will be reset here using the **`params`** dictionary that we defined in each iteration of the cross-validation loop. The other notable options here are that we use the ROC AUC metric (**`scoring='roc_auc'`**), that we do a 4-fold cross-validation **`(cv=4)`**, and that we calculate training scores (**`return_train_score=True`**) to assess the bias-variance trade-off.

Once the cross-validation object is defined, we can simply use the **`.fit`** method on it as we would with a model object. This encapsulates essentially all the functionality of the cross-validation loop we wrote in the previous chapter.

5. Perform a 4-fold cross-validation to search for the optimal maximum depth using this code:

```
cv.fit(X_train, y_train)
```

The output should be as follows:

```
Fitting 4 folds for each of 7 candidates, totalling 28 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  28 out of  28 | elapsed:    2.8s finished
GridSearchCV(cv=4, error_score=nan,
       estimator=DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best'),
       fit_params=None, iid=False, n_jobs=None,
       param_grid={'max_depth': [1, 2, 4, 6, 8, 10, 12]},
       pre_dispatch=None, refit=True, return_train_score=True,
       scoring='roc_auc', verbose=1)
```

Figure 5.18: The cross-validation fitting output

All the options that we specified are printed as output. Additionally, there is some output information about how many cross-validation fits were performed. We had 4 folds and 7 hyperparameters, meaning 4 x 7 = 28 fits are performed. The amount of time this took is also displayed. You can control how much output you get from this procedure with the **verbose** keyword argument; larger numbers mean more output.

Now it's time to examine the results of the cross-validation procedure. Among the methods that are available on the fitted **GridSearchCV** object is **.cv_results_**. This is a dictionary containing the names of results as keys and the results themselves as values. For example, the **mean_test_score** key holds the average testing score across the folds for each of the seven hyperparameters. You could directly examine this output by running **cv.cv_results_** in a code cell. However, this is not easy to read. Dictionaries with this kind of structure are immediately usable for creation of a pandas DataFrame, which makes looking at the results a little easier.

6. Run the following code to create and examine a pandas **DataFrame** of cross-validation results:

```
cv_results_df = pd.DataFrame(cv.cv_results_)
cv_results_df
```

The output should be as follows:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_max_depth	params	split0_test_score
0	0.023161	0.002917	0.002712	0.000827	1	{'max_depth': 1}	0.639514
1	0.040478	0.005298	0.003616	0.001432	2	{'max_depth': 2}	0.695134
2	0.062975	0.001703	0.002196	0.000017	4	{'max_depth': 4}	0.732720
3	0.094926	0.005329	0.002393	0.000150	6	{'max_depth': 6}	0.743836
4	0.123850	0.008124	0.003107	0.000768	8	{'max_depth': 8}	0.727948
5	0.142454	0.001005	0.002620	0.000221	10	{'max_depth': 10}	0.709049
6	0.178841	0.016180	0.002716	0.000238	12	{'max_depth': 12}	0.675597

Figure 5.19: Cross-validation results

The DataFrame has one row for each combination of hyperparameters in the grid. Since we are only searching one hyperparameter here, there is one row for each of the seven values that we searched for. You can see a lot of output for each row, such as the mean and standard deviation of the time in seconds that each of the four folds took for both training (fitting) and testing (scoring). The hyperparameter values that were searched for are also shown. In figure 5.19 we can see the ROC AUC score for the testing data of the first fold (index 0). So, what are the rest of the columns in the results DataFrame?

7. View the names of the remaining columns in the results DataFrame using this code:

```
cv_results_df.columns
```

The output should be as follows:

```
Index(['mean_fit_time', 'std_fit_time', 'mean_score_time', 'std_score_time',
       'param_max_depth', 'params', 'split0_test_score', 'split1_test_score',
       'split2_test_score', 'split3_test_score', 'mean_test_score',
       'std_test_score', 'rank_test_score', 'split0_train_score',
       'split1_train_score', 'split2_train_score', 'split3_train_score',
       'mean_train_score', 'std_train_score'],
      dtype='object')
```

Figure 5.20: Columns in the `DataFrame` of cross-validation results

The columns in the cross-validation results DataFrame include the testing scores for each fold, their average and standard deviation, and the same information for the training scores.

Generally speaking, the "best" combination of hyperparameters is that with the highest average testing score. This is an estimation of how well the model, fit using these hyperparameters, could perform when scored on new data. Let's make a plot showing how the average testing score varies with the **max_depth** hyperparameter. We will also show the average training scores on the same plot, to see how bias and variance change as we allow deeper and more complex trees to be grown during model fitting.

We include the standard deviations of the 4-fold training and testing scores as error bars, using the Matplotlib **errorbar** function. This gives you an indication of how variable the scores are across the folds.

8. Execute the following code to create an error bar plot of training and testing scores for each value of **max_depth** that was examined in cross-validation:

```
ax = plt.axes()
ax.errorbar(cv_results_df['param_max_depth'],
            cv_results_df['mean_train_score'],
            yerr=cv_results_df['std_train_score'],
            label='Mean $\pm$ 1 SD training scores')
ax.errorbar(cv_results_df['param_max_depth'],
            cv_results_df['mean_test_score'],
            yerr=cv_results_df['std_test_score'],
            label='Mean $\pm$ 1 SD testing scores')
ax.legend()
plt.xlabel('max_depth')
plt.ylabel('ROC AUC')
```

The plot should appear as follows:

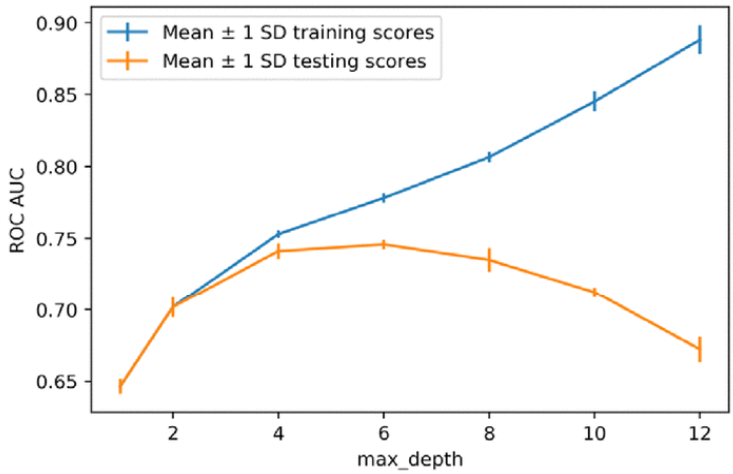

Figure 5.21: An error bar plot of training and testing scores across the four folds

The standard deviations of the training and testing scores are shown as vertical lines at each value of **`max_depth`** that was tried; the distance above and below the average score is 1 standard deviation. Whenever making error bar plots, it's best to ensure that the units of the error measurement are the same as the units of the y axis. In this case they are, since standard deviation has the same units as the underlying data, as opposed to variance, for example, which has squared units.

The error bars indicate how variable the scores are across folds. If there was a large amount of variation across the folds, it would indicate that the nature of the data across the folds was different in a way that affected the ability of our model to describe it. This could be concerning because it would indicate that we may not have enough data to train a model that would reliably perform on new data. However, in our case here, there is not much variability between the folds, so this is not an issue.

What about the general trends of the training and testing scores across the different values of **`max_depth`**? We can see that as we grow deeper and deeper trees, the model fits the training data better and better. As noted previously, if we grew trees deep enough so that each leaf node had just one training sample, we create a model that is very specific to the training data. In fact, it would fit the training data perfectly. We could say that such a model had extremely high **variance**.

But this performance on the training set does not necessarily translate over to the testing set. In *Figure* 5.21, it's apparent that increasing `max_depth` only increases testing scores up to a point, after which deeper trees in fact have lower testing performance. This is another example of how we can leverage the **bias-variance trade-off** to create a better predictive model – similar to how we use a regularized logistic regression. Shallower trees have more **bias**, since they are not fitting the training data as well. But this is fine because if we accept some bias, we will have better performance on the testing data, which is ultimately the metric we use to select model hyperparameters.

In this case, we would select `max_depth` = 6. You could also do a more thorough search, by trying every integer between 2 and 12, instead of going by 2's as we've done here. In general, it is a good idea to perform as thorough a search of parameter space as you can, up to the limits of the computational time that you have. In this case, it would lead to the same result.

Comparison between models

At this point, we've calculated a 4-fold cross-validation of several different machine learning models on the case study data. So, how are we doing? What's our best so far? In the last chapter, we got an average testing ROC AUC of 0.718 with logistic regression and 0.740 by engineering interaction features in a logistic regression. Here, with a decision tree, we can achieve 0.745. So, we are making gains in model performance. We will explore one more type of model to see if we can push performance even higher.

Random Forests: Ensembles of Decision Trees

As we saw in the previous exercise, decision trees are prone to overfitting. This is one of the principle criticisms of their usage, despite the fact that they are highly interpretable. We were able to limit this overfitting, to an extent, however, by limiting the maximum depth to which the tree could be grown.

It turns out that there are powerful and widely-used predictive models that use decision trees as the basis for more complex procedures. In particular, we will focus here on random forests of decision trees. Random forests are examples of what are called ensemble models, because they are formed by combining other models. By combining the predictions of many models, it is possible to improve upon the deficiencies of any given one of them.

Once you understand decision trees, the concept behind random forests is actually quite simple. That is because random forests are just ensembles of many decision trees; all the models in this kind of ensemble have the same mathematical form. So, how many decision tree models will be included in a random forest? This is one of the hyperparameters, `n_estimators`, that needs to be specified when building a random forest model. Generally speaking, the more trees, the better. As the number of trees increases, the variance of the overall ensemble will decrease. This should result in the random forest model having better generalization to new data, which will be reflected in increased testing scores. However, there will be a point of diminishing returns after which increasing the number of trees does not result in a substantial improvement in model performance.

So, how do random forests reduce the high variance (overfitting) issue that affects decision trees? The answer to this question lies in what is different about the different trees in the forest. There are two principle ways in which the trees are different, one of which we are already familiar with:

- The number of features considered at each split
- The samples used to grow different trees

The number of features considered at each split

We are already familiar with this option from the `DecisionTreeClassifier` class: `max_features`. In our previous usage of this class, we left `max_features` at its default value of `None`, which meant that all features were considered at each split. By using all the features to fit the training data, overfitting is possible. By limiting the number of features considered at each split, some of the decision trees in a random forest will potentially find better splits. This is because, although they are still greedily searching for the best split, they are doing it with a limited selection of features. This may make certain splits possible later in the tree that may not have been found if all features were being searched at each split.

There is a `max_features` option in the `RandomForestClassifier` class in scikit-learn just as there is for the `DecisionTreeClassifier` class and the options are similar. However, for the random forest, the default setting is 'auto', which means the algorithm will only search a random selection of the square root of the number of possible features at each split, for example a random selection of $\sqrt{9} = 3$ features from a total of 9 possible features. Because each tree in the forest will likely choose different random selections of features to split as the trees are being grown, the trees in the forest will not be the same.

The samples used to grow different trees

The other way that the trees in a random forest differ from each other is that they are usually grown with different training samples. To do this, a statistical procedure known as bootstrapping is used, which means to generate new synthetic datasets from the original data. The synthetic datasets are created by randomly selecting samples from the original dataset using replacement. Here, "replacement" means that if we select a certain sample, we will continue to consider it for selection, that is, it is "replaced" to the original dataset after we've sampled it. The number of samples in the synthetic datasets are the same as those in the original dataset, but some samples may be repeated because of replacement.

The procedure of using random sampling to create synthetic datasets, and training models on them separately, is called bagging, which is short for bootstrapped aggregation. Bagging can, in fact, be used with any machine learning model, not just decision trees, and scikit-learn offers functionality to do this for both classification (`BaggingClassifier`) and regression (`BaggingRegressor`) problems. In the case of random forest, bagging is turned on by default and the bootstrap option is set to `True`. But if you want all the trees in the forest to be grown using all of the training data, you can set this option to `False`.

Now you should have a good understanding of what a random forest is. As you can see, if you are already familiar with decision trees, understanding random forests does not involve much additional knowledge. A reflection of this fact is that the hyperparameters available for the `RandomForestClassifier` class in scikit-learn are mostly the same as those for the `DecisionTreeClassifier` class.

In addition to `n_estimators` and `bootstrap`, which we discussed previously, there are only two additional new options beyond what's available for decision trees:

- **`oob_score`, a bool**: This option controls whether or not to calculate an out of bag (OOB) score for each tree. This can be thought of as a testing score, where the samples that were not selected by the bagging procedure to grow a given tree are used to assess model performance of that tree. Here, use `True` to calculate the OOB score or `False` (the default) not to.
- **`warm_start`, a bool**: This is `False` by default – if you set this to `True`, then reusing the same random forest model object will cause additional trees to be added to the already-generated forest.

Other kinds of ensemble models

Random forest, as we now know, is an example of a bagging ensemble. Another kind of ensemble is a **boosting** ensemble. The general idea of boosting is to use successive new models of the same type and to train them on the errors of previous models. This way, successive models learn where earlier models didn't do well and correct these errors. Boosting has enjoyed successful application with decision trees and is available in scikit-learn and another popular Python package called `XGBoost`.

Stacking ensembles are a somewhat more advanced kind of ensemble, where the different models (estimators) within the ensemble do not need to be of the same type as they do in bagging and boosting. For example, you could build a stacking ensemble with a random forest and a logistic regression. The predictions of the different members of the ensemble are combined for a final prediction using yet another model (the **stacker**), which considers the predictions of the **stacked** models as features.

Random Forest: Predictions and Interpretability

Since a random forest is just a collection of decision trees, somehow the predictions of all those trees must be combined to create the prediction of the random forest.

After model training, classification trees will take an input sample and a produce a predicted class, for example, whether or not a credit account in our case study problem will default. One intuitive approach to combining the predictions of these trees into the ultimate prediction of the forest is to take a majority vote. That is, whatever the most common prediction of all the trees is becomes the prediction of the forest. This was, in fact, the approach taken in the publication first describing random forests (https://scikit-learn.org/stable/modules/ensemble.html#forest). However, scikit-learn uses a somewhat different approach: adding up the predicted probabilities for each class and then choosing the one with the highest probability. This captures more information from each tree than just the predicted class.

Interpretability of random forests

One of the main advantages of decision trees is that it is straightforward to see how any individual prediction is made. You can trace the decision path for any predicted sample through the series of "if then" rules used to make a prediction and know exactly how it came to have that prediction. By contrast, imagine that you have a random forest consisting of 1,000 trees. This would mean there are 1,000 sets of rules like this, which are much harder to communicate to human beings than 1 set of rules!

That being said, there are various methods that can be used to understand how random forests make predictions. There are advanced techniques for understanding how individual predictions are made, that are the subject of recent academic research and are beyond the scope of this book. However, a simple way to interpret the way a random forest works, that is available in scikit-learn, is to observe the feature importance. The feature importance of a random forest are a measure of how useful each of the features were when growing the trees in the forest. This usefulness is measured by using a combination of the fraction of training samples that were split using that feature, and the decrease in node impurity that resulted.

Because of the feature importance calculation, which can be used to rank features by how useful they are to the random forest model, random forests can also be used for feature selection.

Exercise 21: Fitting a Random Forest

In this exercise, we will extend our efforts with decision trees, by using the random forest model with cross-validation on the training data from the case study. We will observe the effect of increasing the number of trees in the forest and examine the feature importance that can be calculated using a random forest model. Perform the following steps to complete the exercise:

> **Note**
>
> The code and the resulting output for this exercise have been loaded in a Jupyter Notebook that can be found at https://bit.ly/2UpGDW2.

1. Import the random forest classifier model class as follows:

```
from sklearn.ensemble import RandomForestClassifier
```

2. Instantiate the class using these options:

```
rf = RandomForestClassifier(
n_estimators=10, criterion='gini', max_depth=3,
min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0,
max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.0,
min_impurity_split=None, bootstrap=True, oob_score=False, n_jobs=None,
random_state=4, verbose=0, warm_start=False, class_weight=None)
```

For this exercise, we'll use mainly the default options. However, note that we will set **max_depth** = 3. Here, we are only going to explore the effect of using different numbers of trees, which we will illustrate with relatively shallow trees for the sake of shorter runtimes. To find the best model performance, we'd typically try more and deeper depths of trees.

We also set **random_state** for predictable results across runs.

3. Create a parameter grid for this exercise in order to search the numbers of trees ranging from 10 to 100:

```
rf_params_ex = {'n_estimators':list(range(10,110,10))}
```

We use Python's **range()** function to create an iterator for the integer values we want, and then convert it to a **list** using **list()**.

4. Instantiate a grid search cross-validation object for the random forest model using the parameter grid from the previous step. Otherwise, you can use the same options that were used for the cross-validation of the decision tree:

```
cv_rf_ex = GridSearchCV(rf, param_grid=rf_params_ex, scoring='roc_auc',
fit_params=None,
                        n_jobs=None, iid=False, refit=True, cv=4,
verbose=1,
                        pre_dispatch=None, error_score=np.nan, return_
train_score=True)
```

5. Fit the cross-validation object as follows:

```
cv_rf_ex.fit(X_train, y_train)
```

The fitting procedure should output the following:

```
Fitting 4 folds for each of 10 candidates, totalling 40 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  40 out of  40 | elapsed:   22.9s finished
GridSearchCV(cv=4, error_score=nan,
       estimator=RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=3, max_features='auto', max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=None,
            oob_score=False, random_state=4, verbose=0, warm_start=False),
       fit_params=None, iid=False, n_jobs=None,
       param_grid={'n_estimators': [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]},
       pre_dispatch=None, refit=True, return_train_score=True,
       scoring='roc_auc', verbose=1)
```

Figure 5.22: The output from the cross-validation of the random forest across different numbers of trees

You probably noticed that, although we are only cross-validating over 10 hyperparameter values, comparable to the 7 values that we examined for the decision tree in the previous exercise, this cross-validation took noticeably longer. Consider how many trees we are growing in this case. For the last hyperparameter, **`n_estimators`** = 100, we have grown a total of 400 trees across all the cross-validation splits.

So, how long has model fitting taken across the various numbers of trees that we just tried? What gains in cross-validation testing performance have we made by using more trees? These are good things to examine using plots. First, we'll pull the cross-validation results out in to a pandas DataFrame as we've done before.

6. Put the cross-validation results into a pandas **`DataFrame`**:

```
cv_rf_ex_results_df = pd.DataFrame(cv_rf_ex.cv_results_)
```

You can examine the whole **`DataFrame`** in the accompanying Jupyter notebook. Here, we move directly to creating plots of the quantities of interest. We'll make a line plot, with symbols, of the mean fit time across the folds for each hyperparameter, contained in the column **`mean_fit_time`**, as well as an error bar plot of testing scores, which we've already done for decision trees. Both plots will be against **`max_depth`** on the x axis.

7. Create two subplots, of the mean time and mean testing scores with standard deviation:

```
fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(6, 3))
axs[0].plot(cv_rf_ex_results_df['param_n_estimators'],
            cv_rf_ex_results_df['mean_fit_time'],
            '-o')
axs[0].set_xlabel('Number of trees')
axs[0].set_ylabel('Mean fit time (seconds)')
axs[1].errorbar(cv_rf_ex_results_df['param_n_estimators'],
                cv_rf_ex_results_df['mean_test_score'],
                yerr=cv_rf_ex_results_df['std_test_score'])
axs[1].set_xlabel('Number of trees')
axs[1].set_ylabel('Mean testing ROC AUC $\pm$ 1 SD ')
plt.tight_layout()
```

Here, we've used **`plt.subplots`** to create two axes at once, within a figure, in a one-row-by-two-column configuration. We then access the axes objects by indexing the array of **`axs`** axes returned from this operation, to create plots.

The output should be this plot:

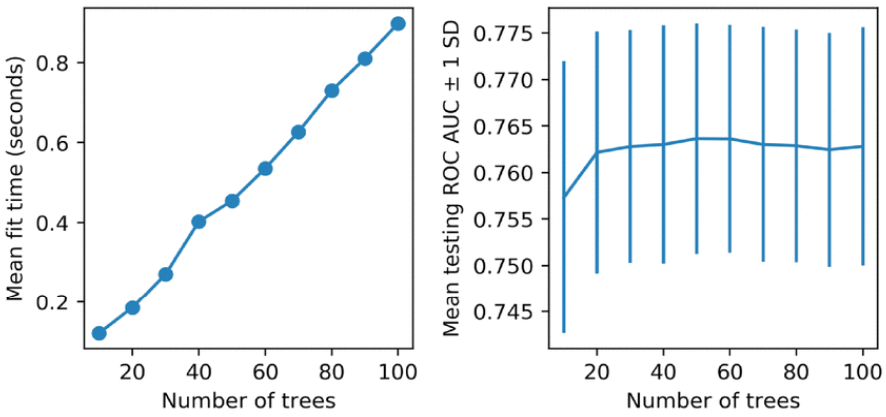

Figure 5.23: The mean fitting time and testing scores for different numbers of trees in the forest

> **Note**
>
> Your results may be different due to the differences in the platform or if you set a different random seed.

There are several things to note about these visualizations. First of all, we can see that by using a random forest, we have increased model performance on the cross-validation testing folds above that of any of our previous efforts. While we haven't made an attempt to tune the random forest hyperparameters to achieve the best model performance we can, this is a promising result and indicates that a random forest will be a valuable addition to our modeling efforts.

However, along with these higher model testing scores, notice that there is also more variability between the folds than what we saw with the decision tree; this variability is visible as larger standard deviations in model testing scores across the folds. While this indicates that there is a wider range in model performance that might be expected from using this model, you are encouraged to examine the model testing scores of the folds directly in the pandas DataFrame in the Jupyter notebook. You should see that even the lowest score from an individual fold is still higher than the average testing score from the decision tree, indicating that it will be better to use a random forest.

So, what about the other questions that we set out to explore with this visualization? We are interested in seeing how long it takes to fit random forest models with various numbers of trees, and what the gains in model performance are from using more trees. The subplot on the left of *Figure 5.23* shows that there is a linear increase in training time as more trees are added to the forest. This is probably to be expected; we are simply multiplying the amount of computation to be done in the training procedure by adding more trees.

But is this additional computational time worth it in terms of increased model performance? The subplot on the right of *Figure 5.23* shows that beyond about 20 trees, it's not clear that adding more trees reliably improves testing performance. While the model with 50 trees has the highest score, the fact that adding more trees actually decreases the testing score somewhat indicates that the gain in ROC AUC for 50 trees may just be due to randomness, as adding more trees theoretically should increase model performance. Based on this reasoning, if we were limited to **`max_depth`** = 3, we may choose a forest of 20 or perhaps 50 trees and proceed. However, we will explore the parameter space more fully in the activity at the end of this chapter.

Finally, note that we have not shown the training ROC AUC metrics here. If you were to plot these or look them up in the results DataFrame, you'd see that the training scores are higher than the testing scores, indicating that some amount of overfitting is happening. While this may be the case, it's still true that the testing scores for this random forest model are higher than those that we've observed for any other model. Based on this result, we would likely choose the random forest model.

As a few additional insights into what we can access using our fitted cross-validation object, let's take a look at the best hyperparameters and the feature importance.

8. Use this code to see the best hyperparameters from cross-validation:

```
cv_rf_ex.best_params_
```

This should output:

```
{'n_estimators': 50}
```

Figure 5.24: Best hyperparameters from cross-validation

Here, "best" just means the hyperparameters that resulted in the highest average model testing score.

9. Run this code to create a **`DataFrame`** of the feature names and importance, and then show it sorted by importance:

```
feat_imp_df = pd.DataFrame({
    'Feature name':features_response[:-1],
    'Importance':cv_rf_ex.best_estimator_.feature_importances_
})
feat_imp_df.sort_values('Importance', ascending=False)
```

The first few rows of output should be as follows:

	Feature name	Importance
4	PAY_1	0.609609
11	PAY_AMT1	0.094123
0	LIMIT_BAL	0.079265
13	PAY_AMT3	0.047067
12	PAY_AMT2	0.035393

Figure 5.25: Feature importance from a random forest

In this code, we've created a dictionary with feature names and importance. The feature importance came from the **`best_estimator_`** method of the fitted cross-validation object. This is a way to access the random forest model object, that was trained on all the training data, using the best hyperparameters we viewed in the previous step. **`feature_importances_`** is a method that can be used on fitted random forest models. After putting the feature names and importance in a dictionary, we create the DataFrame, and then show it sorted, descending by importance. Notice that the top 5 most important features from the random forest, are the same as the top 5 chosen by an ANOVA F-test in *Chapter 3*, *Details of Logistic Regression and Feature Exploration*, although they are in a somewhat different order. This is good confirmation between the different methods.

Checkerboard Graph

Before moving on to the Activity, we illustrate a visualization technique in Matplotlib. Plotting a two-dimensional grid with colored squares or other shapes on it, can be useful when you want to show three dimensions of data. Here, color illustrates the third dimension. For example, you may want to visualize model testing scores over a grid of two hyperparameters. This is, in fact, the use case in *Activity 5, Cross-Validation Grid Search with Random Forest*.

The first step in the process is to create grids of x and y coordinates. The NumPy **`meshgrid`** function can be used to do this. This function takes one-dimensional arrays of x and y coordinates and creates the mesh grid with all the possible pairs from both. The points in the mesh grid will be the corners of the checkerboard plot. Here is how the code looks for a 4 x 4 grid of colored patches. Since we are specifying the corners, we need a 5 x 5 grid of points:

```
xx_example, yy_example = np.meshgrid(range(5), range(5))
print(xx_example)
print(yy_example)
```

The output is as follows:

```
[[0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]]
[[0 0 0 0 0]
 [1 1 1 1 1]
 [2 2 2 2 2]
 [3 3 3 3 3]
 [4 4 4 4 4]]
```

Figure 5.26: The mesh grid output

The grid of data to plot on this mesh should also have a square shape. We take one-dimensional array of integers between 1 and 16, and reshape it to a two-dimensional, 4 x 4 grid:

```
z_example = np.arange(1,17).reshape(4,4)
z_example
```

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16]])
```

Figure 5.27: Data for the checkerboard plot

We can plot the **z_example** data on the **xx_example, yy_example** mesh grid with the following code. Notice that we use **pcolormesh** to make the plot with the **jet** colormap, which gives a rainbow color scale. We add a **colorbar**, which needs to be passed the object **pcolor_ex** returned by **pcolormesh** as an argument, so the interpretation of the color scale is clear:

```
ax = plt.axes()
pcolor_ex = ax.pcolormesh(xx_example, yy_example, z_example, cmap=plt.
cm.jet)
plt.colorbar(pcolor_ex, label='Color scale')
ax.set_xlabel('X coordinate')
ax.set_ylabel('Y coordinate')
```

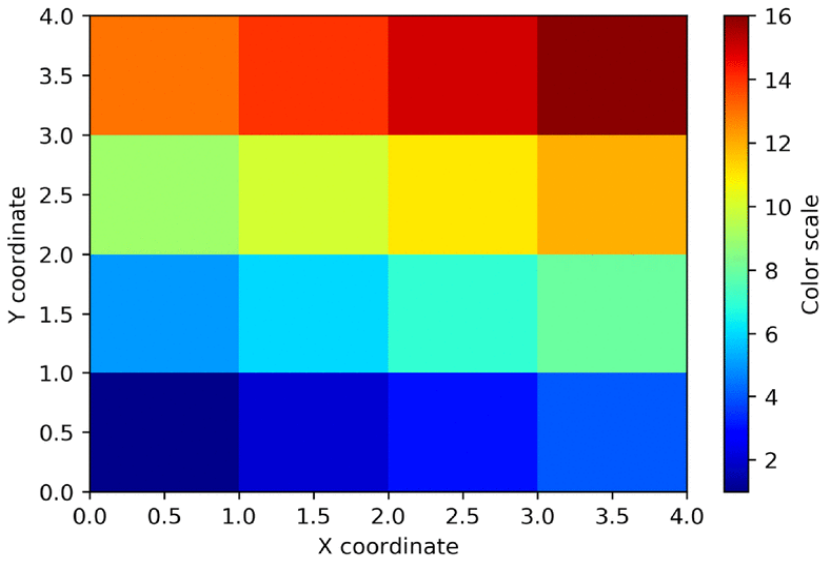

Figure 5.28: A **pcolormesh** plot of consecutive integers

Activity 5: Cross-Validation Grid Search with Random Forest

In this activity, you will conduct a grid search over the number of trees in the forest (**n_estimators**) and the maximum depth of a tree (**max_depth**) for a random forest model on the case study data. You will then create a visualization showing the average testing score for the grid of hyperparameters that you searched over. Perform the following steps to complete the activity:

> **Note**
>
> The code and the resulting output for this activity have been loaded in a Jupyter Notebook that can be found at http://bit.ly/2GI7fJB.

1. Create a dictionary representing the grid for the **`max_depth`** and **`n_estimators`** hyperparameters that will be searched. Include depths of 3, 6, 9, and 12, and 10, 50, 100, and 200 trees. Leave the other hyperparameters at their defaults.
2. Instantiate a **`GridSearchCV`** object using the same options that we have previously in this chapter, but with the dictionary of hyperparameters created in step 1 here. Set **`verbose=2`** to see the output for each fit performed. You can reuse the same random forest model object **`rf`** that we have been using.
3. Fit the **`GridSearchCV`** object on the training data.
4. Put the results of the grid search in a pandas DataFrame.
5. Create a **`pcolormesh`** visualization of the mean testing score for each combination of hyperparameters.
6. Conclude which set of hyperparameters to use.

> **Note**
>
> The solution to this activity can be found on page 344.

Summary

In this chapter, we've learned how to use decision trees and the ensemble models called random forests that are made up of many decision trees. Using these simply conceived models, we were able to make better predictions than we could with logistic regression, judging by the cross-validation ROC AUC score. This is often the case for many real-world problems. Decision trees are robust to a lot of the potential issues that can prevent logistic regression models from good performance, such as non-linear relationships between features and the response variable, and the presence of complicated interactions among features.

Although a single decision tree is prone to overfitting, the random forest ensemble method has been shown to reduce this high-variance issue. Random forests are built by training many trees. The decreased variance of the ensemble of trees is achieved by increasing the bias of the individual trees in the forest, by only training them on a portion of the available training set (bootstrapped aggregation or bagging) and only considering a reduced number of features at each node split.

Now we have tried several different machine learning approaches to modeling the case study data. We found that some work better than others; for example, a random forest with tuned hyperparameters provides the highest average cross-validation ROC AUC score of 0.776.

Ultimately, however, the ROC AUC score is an abstract metric by which to judge a model. The client may not have much context for what the ROC AUC score means for them in a practical sense, but they definitely need to know whether the model can meet their business needs. For this reason, a crucial final step for machine learning in a business context is to try to determine the financial, or business, value of a predictive model. We will show you how to go about this, as well as address other issues having to do with delivering models that will be used to guide business decisions, in the next chapter.