Chapter 4. Predicting Forest Cover with Decision Trees

Sean Owen

Prediction is very difficult, especially if it's about the future.

—Niels Bohr

In the late nineteenth century, the English scientist Sir Francis Galton was busy measuring things like peas and people. He found that large peas (and people) had larger-than-average offspring. This isn't surprising. However, the offspring were, on average, smaller than their parents. In terms of people: the child of a seven-foot-tall basketball player is likely to be taller than the global average but still more likely than not to be less than seven feet tall.

As almost a side effect of his study, Galton plotted child versus parent size and noticed there was a roughly linear relationship between the two. Large parent peas had large children, but slightly smaller than themselves; small parents had small children, but generally a bit larger than themselves. The line's slope was therefore positive but less than 1, and Galton described this phenomenon as we do today, as *regression to the mean*.

Although maybe not perceived this way at the time, this line was, to me, an early example of a predictive model. The line links the two values, and implies that the value of one suggests a lot about the value of the other. Given the size of a new pea, this relationship could lead to a more accurate estimate of its offspring's size than simply assuming the offspring would be like the parent or like every other pea.

Fast Forward to Regression

More than a century of statistics later, and since the advent of modern machine learning and data science, we still talk about the idea of predicting a value from other values as re-

gression, even though it has nothing to do with slipping back toward a mean value, or in-

The common thread linking regression and classification is that both involve predicting one (or more) values given one (or more) other values. To do so, both require a body of inputs and outputs to learn from. They need to be fed both questions and known answers. For this reason, they are known as types of *supervised learning*.

Classification and regression are the oldest and most well-studied types of predictive analytics. Most algorithms you will likely encounter in analytics packages and libraries are classification or regression techniques, like support vector machines, logistic regression, naïve Bayes, neural networks, and deep learning. Recommenders, the topic of Chapter 3, were comparatively more intuitive to introduce, but are also just a relatively recent and separate subtopic within machine learning.

This chapter will focus on a popular and flexible type of algorithm for both classification and regression: decision trees, and the algorithm's extension, random decision forests. The exciting thing about these algorithms is that, with all due respect to Mr. Bohr, they can help predict the future—or at least, predict the things we don't yet know for sure, like the likelihood you will buy a car based on your online behavior, whether an email is spam given its words, or which acres of land are likely to grow the most crops given their location and soil chemistry.

Vectors and Features

To explain the choice of the data set and algorithm featured in this chapter, and to begin to explain how regression and classification operate, it is necessary to briefly define the terms that describe their input and output.

Consider predicting tomorrow's high temperature given today's weather. There is nothing wrong with this idea, but "today's weather" is a casual concept that requires structuring before it can be fed into a learning algorithm.

Really, it is certain *features* of today's weather that may predict tomorrow's temperature, such as:

- Today's high temperature

- Today's low temperature

- The number of weather forecasters predicting a cold snap tomorrow

These features are also sometimes called *dimensions*, *predictors*, or just *variables*. Each of these features can be quantified. For example, high and low temperatures are measured in degrees Celsius, humidity can be measured as a fraction between 0 and 1, and weather type can be labeled `cloudy`, `rainy`, or `clear`. The number of forecasters is, of course, an integer count. Today's weather might therefore be reduced to a list of values like `13.1,19.0,0.73,cloudy,1`.

These five features together, in order, are known as a *feature vector*, and can describe any day's weather. This usage bears some resemblance to use of the term *vector* in linear algebra, except that a vector in this sense can conceptually contain nonnumeric values, and even lack some values.

These features are not all of the same type. The first two features are measured in degrees Celsius, but the third is unitless, a fraction. The fourth is not a number at all, and the fifth is a number that is always a nonnegative integer.

For purposes of discussion, this book will talk about features in two broad groups only: *categorical* features and *numeric* features. In this context, numeric features are those that can be quantified by a number and have a meaningful ordering. For example, it's meaningful to say that today's high was 23°C, and that this is higher than yesterday's high of 22°C. All of the features mentioned previously are numeric, except the weather type. Terms like `clear` are not numbers, and have no ordering. It is meaningless to say that `cloudy` is larger than `clear`. This is a categorical feature, which instead takes on one of several discrete values.

Training Examples

A learning algorithm needs to train on data in order to make predictions. It requires a large number of inputs, and known correct outputs, from historical data. For example, in this problem, the learning algorithm would be given that, one day, the weather was between 12° and 16°C, with 10% humidity, clear, with no forecast of a cold snap; and the following day, the high temperature was 17.2°C. With enough of these *examples*, a learning algorithm might learn to predict the following day's high temperature with some accuracy.

Feature vectors provide an organized way to describe input to a learning algorithm (here:

Note that regression problems are just those where the target is a numeric feature, and classification problems are those where the target is categorical. Not every regression or classification algorithm can handle categorical features or categorical targets; some are limited to numeric features.

Decision Trees and Forests

It turns out that the family of algorithms known as *decision trees* can naturally handle both categorical and numeric features. Building a single tree can be done in parallel, and many trees can be built in parallel at once. They are robust to outliers in the data, meaning that a few extreme and possibly erroneous data points might not affect predictions at all. They can consume data of different types and on different scales without the need for preprocessing or normalization, which is an issue that will reappear in Chapter 5.

Decision trees generalize into a more powerful algorithm, called *random decision forests*. Their flexibility makes these algorithms worthwhile to examine in this chapter, where Spark MLlib's `DecisionTree` and `RandomForest` implementation will be applied to a data set.

Decision tree–based algorithms have the further advantage of being comparatively intuitive to understand and reason about. In fact, we all probably use the same reasoning embodied in decision trees, implicitly, in everyday life. For example, I sit down to have morning coffee with milk. Before I commit to that milk and add it to my brew, I want to predict: is the milk spoiled? I don't know for sure. I might check if the use-by date has passed. If not, I predict `no`, it's not spoiled. If the date has passed, but that was three or fewer days ago, I take my chances and predict `no`, it's not spoiled. Otherwise, I sniff the milk. If it smells funny, I predict `yes`, and otherwise `no`.

This series of yes/no decisions that lead to a prediction are what decision trees embody. Each decision leads to one of two results, which is either a prediction or another decision, as shown in Figure 4-1. In this sense, it is natural to think of the process as a tree of decisions, where each internal node in the tree is a decision, and each leaf node is a final answer.

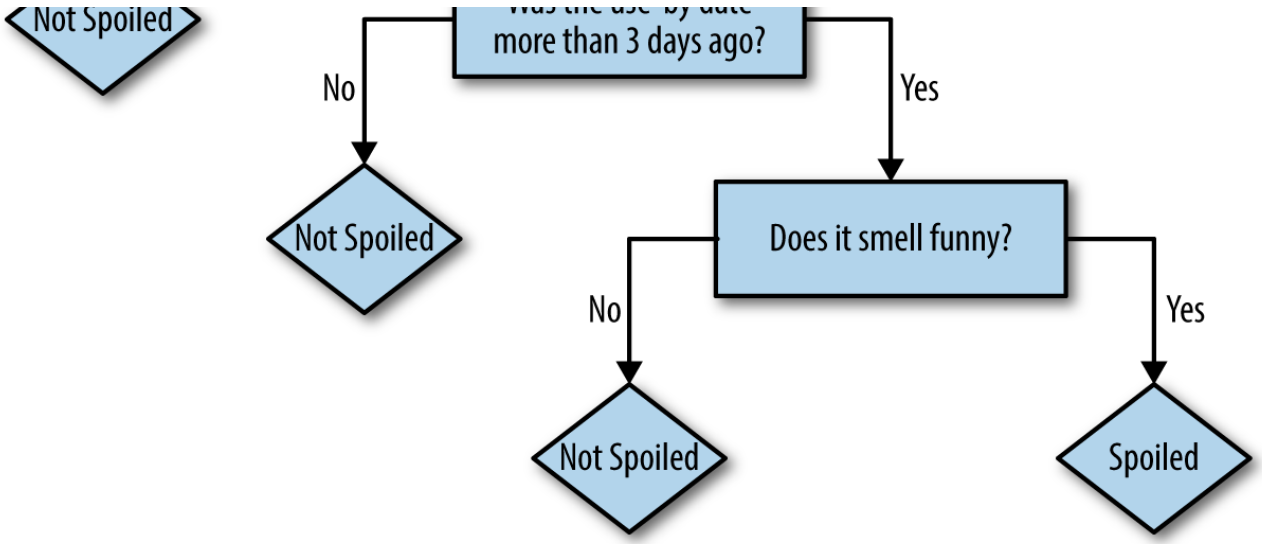

Figure 4-1. Decision tree: is it spoiled?

The preceding rules were ones I learned to apply intuitively over years of bachelor life—they seemed like rules that were both simple and also usefully differentiated cases of spoiled and nonspoiled milk. These are also properties of a good decision tree.

That is a simplistic decision tree, and was not built with any rigor. To elaborate, consider another example. A robot has taken a job in an exotic pet store. It wants to learn, before the shop opens, which animals in the shop would make a good pet for a child. The owner lists nine pets that would and wouldn't be suitable before hurrying off. The robot compiles the information found in Table 4-1 from examining the animals.

Table 4-1. Exotic pet store "feature vectors"

Mr. Slither	3.1	0	Green	No
Nemo	0.2	0	Tan	Yes
Dumbo	1390.8	4	Gray	No
Kitty	12.1	4	Gray	Yes
Jim	150.9	2	Tan	No
Millie	0.1	100	Brown	No
McPigeon	1.0	2	Gray	No
Spot	10.0	4	Brown	Yes

Although a name is given, it will not be included as a feature. There is little reason to believe the name alone is predictive; "Felix" could name a cat or a poisonous tarantula, for all the robot knows. So, there are two numeric features (weight, number of legs) and one categorical feature (color) predicting a categorical target (is/is not a good pet for a child).

The robot might try to fit a simple decision tree to this training data to start, consisting of a

Figure 4-2. Robot's first decision tree

The logic of the decision tree is easy to read and make some sense of: 500kg animals certainly sound unsuitable as pets. This rule predicts the correct value in five of nine cases. A quick glance suggests that we could improve the rule by lowering the weight threshold to 100kg. This gets six of nine examples correct. The heavy animals are now predicted correctly; the lighter animals are only partly correct.

So, a second decision can be constructed to further refine the prediction for examples with weights less than 100kg. It would be good to pick a feature that changes some of the incorrect `Yes` predictions to `No`. For example, there is one small green animal, sounding suspiciously like a snake, that the robot could predict correctly by deciding on color, as shown in Figure 4-3.

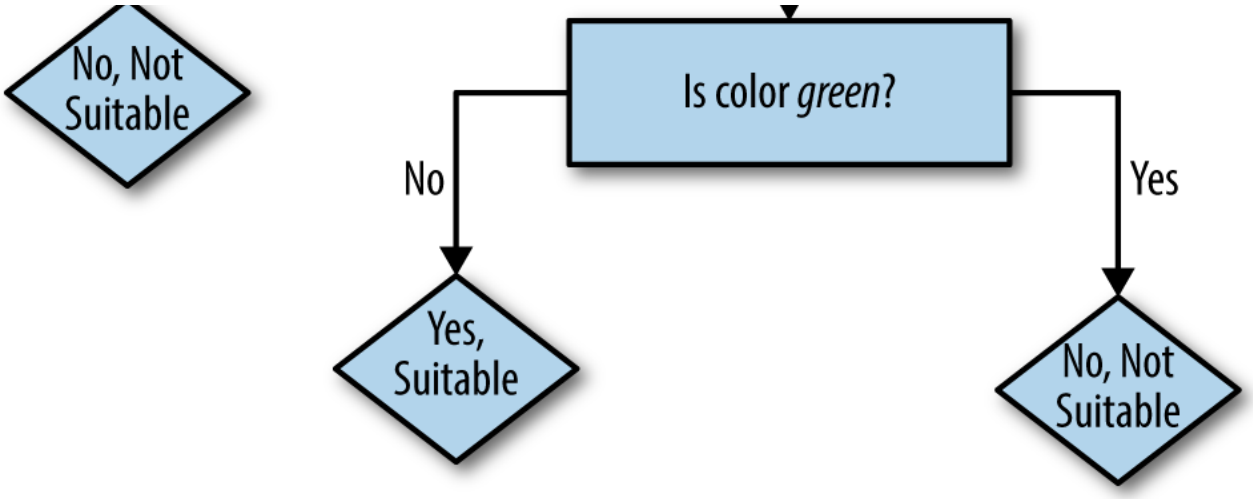

Figure 4-3. Robot's next decision tree

Now, seven of nine examples are correct. Of course, decision rules could be added until all nine were correctly predicted. The logic embodied in the resulting decision tree would probably sound implausible when translated into common speech: "If the animal's weight is less than 100kg, and its color is brown instead of green, and it has fewer than 10 legs, then yes it is a suitable pet." While perfectly fitting the given examples, a decision tree like this would fail to predict that a small, brown, four-legged wolverine is not a suitable pet. Some balance is needed to avoid this phenomenon, known as *overfitting*.

This is enough of an introduction to decision trees for us to begin using them with Spark. The remainder of the chapter will explore how to pick decision rules, how to know when to stop, and how to gain accuracy by creating a forest of trees.

Covtype Data Set

The data set used in this chapter is the well-known Covtype data set, available online as a compressed CSV-format data file, *covtype.data.gz*, and accompanying info file, *covtype.info*.

The data set records the types of forest-covering parcels of land in Colorado, USA. It's only a coincidence that the data set concerns real-world forests! Each example contains several features describing each parcel of land—like its elevation, slope, distance to water, shade,

and soil type—along with the known forest type covering the land. The forest cover type is

data but is large enough to be manageable as an example and still highlight some issues of scale.

Preparing the Data

Thankfully, the data is already in a simple CSV format and does not require much cleansing or other preparation to be used with Spark MLlib. Later, it will be of interest to explore some transformations of the data, but it can be used as is to start.

The *covtype.data* file should be extracted and copied into HDFS. This chapter will assume that the file is available at */user/ds/*. Start `spark-shell`. You may again find it helpful to give the shell a healthy amount of memory to work with, as building decision forests can be resource-intensive. If you have the memory, specify `--driver-memory 8g` or similar.

CSV files contain fundamentally tabular data, organized into rows of columns. Sometimes these columns are given names in a header line, although that's not the case here. The column names are given in the companion file, *covtype.info*. Conceptually, each column of a CSV file has a type as well—a number, a string—but a CSV file doesn't specify this.

It's natural to parse this data as a data frame because this is Spark's abstraction for tabular data, with a defined column schema, including column names and types. Spark has built-in support for reading CSV data, in fact:

```
val dataWithoutHeader = spark.read.
  option("inferSchema", true).
  option("header", false).
  csv("hdfs:///user/ds/covtype.data")

...
org.apache.spark.sql.DataFrame = [_c0: int, _c1: int ... 53 more fields]
```

This code reads the input as CSV and does not attempt to parse the first line as a header of column names. It also requests that the type of each column be inferred by examining the data. It correctly infers that all of the columns are numbers, and more specifically, integers. Unfortunately it can only name the columns "_c0" and so on.

Looking at the column names, it's clear that some features are indeed numeric. "Elevation"

ture that takes on *N* distinct values becomes *N* numeric features, each taking on the value 0 or 1. Exactly one of the *N* values has value 1, and the others are 0. For example, a categorical feature for weather that can be `cloudy`, `rainy`, or `clear` would become three numeric features, where `cloudy` is represented by `1,0,0`; `rainy` by `0,1,0`; and so on. These three numeric features might be thought of as `is_cloudy`, `is_rainy`, and `is_clear` features. Likewise, 40 of the columns are really one `Soil_Type` categorical feature.

This isn't the only possible way to encode a categorical feature as a number. Another possible encoding simply assigns a distinct numeric value to each possible value of the categorical feature. For example, `cloudy` may become 1.0, `rainy` 2.0, and so on. The target itself, "Cover_Type", is a categorical value encoded as a value 1 to 7.

> Be careful when encoding a categorical feature as a single numeric feature. The original categorical values have no ordering, but when encoded as a number, they appear to. Treating the encoded feature as numeric leads to meaningless results because the algorithm is effectively pretending that `rainy` is somehow greater than, and two times larger than, `cloudy`. It's OK as long as the encoding's numeric value is not used as a number.

So we see both types of encodings of categorical features. It would have, perhaps, been simpler and more straightforward to not encode such features (and in two ways, no less), and instead simply include their values directly like "Rawah Wilderness Area." This may be an artifact of history; the data set was released in 1998. For performance reasons or to match the format expected by libraries of the day, which were built more for regression problems, data sets often contain data encoded in these ways.

In any event, before proceeding, it is useful to add column names to this DataFrame in order to make it easier to work with:

```
val colNames = Seq(
    "Elevation", "Aspect", "Slope",
```

```
"Horizontal_Distance_To_Hydrology", "Vertical_Distance_To_Hydrology",
"Horizontal_Distance_To_Roadways"
```



```
  ) ++ Seq("Cover_Type")

val data = dataWithoutHeader.toDF(colNames:_*).
  withColumn("Cover_Type", $"Cover_Type".cast("double"))

data.head

...
org.apache.spark.sql.Row = [2596,51,3,258,0,510,221,232,148,6279,1,0,0,0,...
```

❶ ++ concatenates collections

The wilderness- and soil-related columns are named "Wilderness_Area_0", "Soil_Type_0", and a bit of Scala can generate these 44 names without having to type them all out. Finally, the target "Cover_Type" column is cast to a `double` value upfront, because it will actually be necessary to consume it as a `double` rather than `int` in all Spark MLlib APIs. This will become apparent later.

You can call `data.show()` to see some rows of the data set, but the display is so wide that it will be difficult to read at all. `data.head` displays it as a raw `Row` object, which will be more readable in this case.

A First Decision Tree

In Chapter 3, we built a recommender model right away on all of the available data. This created a recommender that could be sense-checked by anyone with some knowledge of music: looking at a user's listening habits and recommendations, we got some sense that it was producing good results. Here, that is not possible. We would have no idea how to make up a new 54-feature description of a new parcel of land in Colorado or what kind of forest cover to expect from such a parcel.

Instead, we must jump straight to holding out some data for purposes of evaluating the resulting model. Before, the AUC metric was used to assess the agreement between held-out listening data and predictions from recommendations. The principle is the same here, although the evaluation metric will be different: *accuracy*. The majority—90%—of the data will again be used for training, and later, we'll see that a subset of this training set will be

held out for cross-validation (the CV set). The other 10% held out here is actually a third

The data needs a little more preparation to be used with a classifier in Spark MLlib. The input DataFrame contains many columns, each holding one feature that could be used to predict the target column. Spark MLlib requires all of the inputs to be collected into *one* column, whose value is a vector. This class is an abstraction for vectors in the linear algebra sense, and contains only numbers. For most intents and purposes, they work like a simple array of `double` values (floating-point numbers). Of course, some of the input features are conceptually categorical, even if they're all represented with numbers in the input. For now, we'll overlook this point and return to it later.

Fortunately, the `VectorAssembler` class can do this work:

```
import org.apache.spark.ml.feature.VectorAssembler

val inputCols = trainData.columns.filter(_ != "Cover_Type")
val assembler = new VectorAssembler().
  setInputCols(inputCols).
  setOutputCol("featureVector")

val assembledTrainData = assembler.transform(trainData)
assembledTrainData.select("featureVector").show(truncate = false)

...
+-------------------------------------------------------------------- ...
|featureVector                                                       ...
+-------------------------------------------------------------------- ...
|(54,[0,1,2,3,4,5,6,7,8,9,13,15],[1863.0,37.0,17.0,120.0,18.0,90.0,2 ...
|(54,[0,1,2,5,6,7,8,9,13,18],[1874.0,18.0,14.0,90.0,208.0,209.0,135. ...
|(54,[0,1,2,3,4,5,6,7,8,9,13,18],[1879.0,28.0,19.0,30.0,12.0,95.0,20 ...
...
```

Its key parameters are the columns to combine into the feature vector, and the name of the new column containing the feature vector. Here, all columns—*except*—the target, of course—are included as input features. The resulting DataFrame has a new "featureVector" column, as shown.

The output doesn't look exactly like a sequence of numbers, but that's because this shows a raw representation of the vector, represented as a `SparseVector` instance to save stor-

age. Because most of the 54 values are 0, it only stores nonzero values and their indices.

be connected into an actual `Pipeline`. Here, the transformation is just invoked directly, which is sufficient to build a first decision tree classifier model.

```
import org.apache.spark.ml.classification.DecisionTreeClassifier
import scala.util.Random

val classifier = new DecisionTreeClassifier().
  setSeed(Random.nextLong()). ❶
  setLabelCol("Cover_Type").
  setFeaturesCol("featureVector").
  setPredictionCol("prediction")

val model = classifier.fit(assembledTrainData)
println(model.toDebugString)

...
DecisionTreeClassificationModel (uid=dtc_29cfe1281b30) of depth 5 with 63 nodes
  If (feature 0 <= 3039.0)
   If (feature 0 <= 2555.0)
    If (feature 10 <= 0.0)
     If (feature 0 <= 2453.0)
      If (feature 3 <= 0.0)
       Predict: 4.0
      Else (feature 3 > 0.0)
       Predict: 3.0
       ...
```

❶ Use random seed

Again, the essential configuration for the classifier consists of column names: the column containing the input feature vectors and the column containing the target value to predict. Because the model will later be used to predict new values of the target, it is given the name of a column to store predictions.

Printing a representation of the model shows some of its tree structure. It consists of a series of nested decisions about features, comparing feature values to thresholds. (Here, for historical reasons, the features are only referred to by number, not name, unfortunately.)

Decision trees are able to assess the importance of input features as part of their building


```
...
(0.7931809106979147,Elevation)
(0.050122380231328235,Horizontal_Distance_To_Hydrology)
(0.030609364695664505,Wilderness_Area_0)
(0.03052094489457567,Soil_Type_3)
(0.026170212644908816,Hillshade_Noon)
(0.024374024564392027,Soil_Type_1)
(0.01670006142176787,Soil_Type_31)
(0.012596990926899494,Horizontal_Distance_To_Roadways)
(0.011205482194428473,Wilderness_Area_2)
(0.0024194271152490235,Hillshade_3pm)
(0.0018551637821715788,Horizontal_Distance_To_Fire_Points)
(2.450368306995527E-4,Soil_Type_8)
(0.0,Wilderness_Area_3)
...
```

This pairs importance values (higher is better) with column names and prints them in order from most to least important. Elevation seems to dominate as the most important feature; most features are estimated to have virtually no importance when predicting the cover type!

The resulting `DecisionTreeClassificationModel` is itself a transformer because it can transform a data frame containing feature vectors into a data frame also containing predictions.

For example, it might be interesting to see what the model predicts on the *training* data, and compare its prediction with the known correct cover type.

```
val predictions = model.transform(assembledTrainData)
predictions.select("Cover_Type", "prediction", "probability").
  show(truncate = false)

...

+----------+----------+------------------------------------------------- ...
|Cover_Type|prediction|probability                                       ...
+----------+----------+------------------------------------------------- ...
|6.0       |3.0       |[0.0,0.0,0.03421818804589827,0.6318547696523378, ...
|6.0       |4.0       |[0.0,0.0,0.043440860215053764,0.283870967741935, ...
```

```
|6.0        |3.0        |[0.0,0.0,0.03421818804589827,0.6318547696523378, ...
```


stances, it's fairly sure the answer is 3 in several cases and quite sure the answer isn't 1.

Eagle-eyed readers might note that the probability vectors actually have eight values even though there are only seven possible outcomes. The vector's values at indices 1 to 7 do contain the probability of outcomes 1 to 7. However, there is also a value at index 0, which always shows as probability 0.0. This can be ignored, as 0 isn't even a valid outcome, as this says. It's a quirk of representing this information as a vector that's worth being aware of.

Based on this snippet, it looks like the model could use some work. Its predictions look like they are often wrong. As with the `ALS` implementation, the `DecisionTreeClassifier` implementation has several hyperparameters for which a value must be chosen, and they've all been left to defaults here. Here, the test set can be used to produce an unbiased evaluation of the expected accuracy of a model built with these default hyperparameters.

`MulticlassClassificationEvaluator` can compute accuracy and other metrics that evaluate the quality of the model's predictions. It's an example of an evaluator in Spark MLlib, which is responsible for assessing the quality of an output DataFrame in some way.

```
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator

val evaluator = new MulticlassClassificationEvaluator().
  setLabelCol("Cover_Type").
  setPredictionCol("prediction")

evaluator.setMetricName("accuracy").evaluate(predictions)
evaluator.setMetricName("f1").evaluate(predictions)

...
0.6976371385502989
0.6815943874214012
```

After being given the column containing the "label" (target, or known correct output value) and the name of the column containing the prediction, it finds that the two match about 70% of the time. This is the accuracy of this classifier. It can compute other related measures, like the F1 score. For purposes here, accuracy will be used to evaluate classifiers.

This single number gives a good summary of the quality of the classifier's output. Some-

rect predictions are the counts along the diagonal and the predictions are everything else.

Fortunately, Spark provides support code to compute the confusion matrix. Unfortunately, that implementation exists as part of the older MLlib APIs that operate on RDDs. However, that's no big deal, because data frames and data sets can freely be turned into RDDs and used with these older APIs. Here, `MulticlassMetrics` is appropriate for a data frame containing predictions.

```
import org.apache.spark.mllib.evaluation.MulticlassMetrics

val predictionRDD = predictions.
  select("prediction", "Cover_Type").
  as[(Double,Double)]. ❶
  rdd ❷
val multiclassMetrics = new MulticlassMetrics(predictionRDD)
multiclassMetrics.confusionMatrix

...
143125.0  41769.0   164.0    0.0    0.0   0.0  5396.0
65865.0   184360.0  3930.0   102.0  39.0  0.0  677.0
0.0       5680.0    25772.0  674.0  0.0   0.0  0.0
0.0       21.0      1481.0   973.0  0.0   0.0  0.0
87.0      7761.0    648.0    0.0    69.0  0.0  0.0
0.0       6175.0    8902.0   559.0  0.0   0.0  0.0
8058.0    24.0      50.0     0.0    0.0   0.0  10395.0
```

❶ Convert to data set.

❷ Convert to RDD.

> Your values will be a little different. The process of building a decision tree includes some random choices that can lead to slightly different classifications.

Counts are high along the diagonal, which is good. However, there are certainly a number


```
val confusionMatrix = predictions.
  groupBy("Cover_Type").
  pivot("prediction", (1 to 7)).
  count().
  na.fill(0.0). ❶
  orderBy("Cover_Type")

confusionMatrix.show()

...
+----------+------+------+-----+---+---+---+-----+
|Cover_Type|     1|     2|    3|  4|  5|  6|    7|
+----------+------+------+-----+---+---+---+-----+
|       1.0|143125| 41769|  164|  0|  0|  0| 5396|
|       2.0| 65865|184360| 3930|102| 39|  0|  677|
|       3.0|     0|  5680|25772|674|  0|  0|    0|
|       4.0|     0|    21| 1481|973|  0|  0|    0|
|       5.0|    87|  7761|  648|  0| 69|  0|    0|
|       6.0|     0|  6175| 8902|559|  0|  0|    0|
|       7.0|  8058|    24|   50|  0|  0|  0|10395|
+----------+------+------+-----+---+---+---+-----+
```

❶ Replace null with 0

Microsoft Excel users may have recognized the problem as just like that of computing a pivot table. A pivot table groups values by two dimensions whose values become rows and columns of the output, and compute some aggregation within those groupings, like a count here. This is also available as a PIVOT function in several databases, and is supported by Spark SQL. It's arguably more elegant and powerful to compute it this way.

Although 70% accuracy sounds decent, it's not immediately clear whether it is outstanding or poor. How well would a simplistic approach do to establish a baseline? Just as a broken clock is correct twice a day, randomly guessing a classification for each example would also occasionally produce the correct answer.

We could construct such a random "classifier" by picking a class at random in proportion to its prevalence in the training set. For example, if 30% of the training set were cover type

1, then the random classifier would guess "1" 33% of the time. Each classification would be


```
import org.apache.spark.sql.DataFrame

def classProbabilities(data: DataFrame): Array[Double] = {
  val total = data.count()
  data.groupBy("Cover_Type").count(). ❶
    orderBy("Cover_Type"). ❷
    select("count").as[Double]. ❸
    map(_ / total).
    collect()
}

val trainPriorProbabilities = classProbabilities(trainData)
val testPriorProbabilities = classProbabilities(testData)
trainPriorProbabilities.zip(testPriorProbabilities).map { ❹
  case (trainProb, cvProb) => trainProb * cvProb
}.sum

...
0.3771270477245849
```

❶ Count by category

❷ Order counts by category

❸ To data set

❹ Sum products of pairs in training, test sets

Random guessing achieves 37% accuracy then, which makes 70% seem like a good result after all. But this result was achieved with default hyperparameters. We can do even better by exploring what these actually mean for the tree-building process.

Decision Tree Hyperparameters

In Chapter 3, the ALS algorithm exposed several hyperparameters whose values we had to choose by building models with various combinations of values and then assessing the

quality of each result using some metric. The process is the same here, although the metric

number of chained decisions that the classifier will make to classify an example. It is useful to limit this to avoid overfitting the training data, as illustrated previously in the pet store example.

The decision tree algorithm is responsible for coming up with potential decision rules to try at each level, like the `weight >= 100` or `weight >= 500` decisions in the pet store example. Decisions are always of the same form: for numeric features, decisions are of the form `feature >= value`; and for categorical features, they are of the form `feature in (value1, value2, …)`. So, the set of decision rules to try is really a set of values to plug in to the decision rule. These are referred to as "bins" in the Spark MLlib implementation. A larger number of bins requires more processing time but might lead to finding a more optimal decision rule.

What makes a decision rule good? Intuitively, a good rule would meaningfully distinguish examples by target category value. For example, a rule that divides the Covtype data set into examples with only categories 1–3 on the one hand and 4–7 on the other would be excellent because it clearly separates some categories from others. A rule that resulted in about the same mix of all categories as are found in the whole data set doesn't seem helpful. Following either branch of such a decision leads to about the same distribution of possible target values, and so doesn't really make progress toward a confident classification.

Put another way, good rules divide the training data's target values into relatively homogeneous, or "pure," subsets. Picking a best rule means minimizing the impurity of the two subsets it induces. There are two commonly used measures of impurity: Gini impurity and entropy.

Gini impurity is directly related to the accuracy of the random-guess classifier. Within a subset, it is the probability that a randomly chosen classification of a randomly chosen example (both according to the distribution of classes in the subset) is *incorrect*. This is the sum of products of proportions of classes, but with themselves and subtracted from 1. If a subset has N classes and p_i is the proportion of examples of class i, then its Gini impurity is given in the Gini impurity equation:

N

$$i=1$$

If the subset contains only one class, this value is 0 because it is completely "pure." When there are N classes in the subset, this value is larger than 0 and is largest when the classes occur the same number of times—maximally impure.

Entropy is another measure of impurity, borrowed from information theory. Its nature is more difficult to explain, but it captures how much uncertainty the collection of target values in the subset implies about predictions for data that falls in that subset. A subset containing one class suggests that the outcome for the subset is completely certain and has 0 entropy—no uncertainty. A subset containing one of each possible class, on the other hand, suggests a lot of uncertainty about predictions for that subset because data have been observed with all kinds of target values. This has high entropy. Hence, low entropy, like low Gini impurity, is a good thing. Entropy is defined by the entropy equation:

$$I_E(p) = \sum_{i=1}^{N} p_i \log\left(\frac{1}{p}\right) = -\sum_{i=1}^{N} p_i \log\left(p_i\right)$$

> Interestingly, uncertainty has units. Because the logarithm is the natural log (base e), the units are *nats*, the base-e counterpart to more familiar *bits* (which we can obtain by using log base 2 instead). It really is measuring information, so it's also common to talk about the *information gain* of a decision rule when using entropy with decision trees.

One or the other measure may be a better metric for picking decision rules in a given data set. They are, in a way, similar. Both involve a weighted average: a sum over values weighted by p_i. The default in Spark's implementation is Gini impurity.

Finally, minimum information gain is a hyperparameter that imposes a minimum infor-

Tuning Decision Trees

It's not obvious from looking at the data which impurity measure leads to better accuracy, or what maximum depth or number of bins is enough without being excessive. Fortunately, as in Chapter 3, it's simple to let Spark try a number of combinations of these values and report the results.

First, it's necessary to set up a pipeline encapsulating the same two steps above. Creating the `VectorAssembler` and `DecisionTreeClassifier` and chaining these two `Transformers` together results in a single `Pipeline` object that represents these two operations together as one operation:

```
import org.apache.spark.ml.Pipeline

val inputCols = trainData.columns.filter(_ != "Cover_Type")
val assembler = new VectorAssembler().
  setInputCols(inputCols).
  setOutputCol("featureVector")

val classifier = new DecisionTreeClassifier().
  setSeed(Random.nextLong()).
  setLabelCol("Cover_Type").
  setFeaturesCol("featureVector").
  setPredictionCol("prediction")

val pipeline = new Pipeline().setStages(Array(assembler, classifier))
```

Naturally, pipelines can be much longer and more complex. This is about as simple as it gets. Now we can also define the combinations of hyperparameters that should be tested using the Spark ML API's built-in support, `ParamGridBuilder`. It's also time to define the evaluation metric that will be used to pick the "best" hyperparameters, and that is again `MulticlassClassificationEvaluator` here.

```
import org.apache.spark.ml.tuning.ParamGridBuilder

val paramGrid = new ParamGridBuilder().
  addGrid(classifier.impurity, Seq("gini", "entropy")).
  addGrid(classifier.maxDepth, Seq(1, 20)).
```

```
  addGrid(classifier.maxBins, Seq(40, 300)).
```



```
  setMetricName("accuracy")
```

This means that a model will be built and evaluated for two values of four hyperparameters. That's 16 models. They'll be evaluated by multiclass accuracy. Finally, `TrainValidationSplit` brings these components together—the pipeline that makes models, model evaluation metrics, and hyperparameters to try—and can run the evaluation on the training data. It's worth noting that `CrossValidator` could be used here as well to perform full k-fold cross-validation, but it is *k* times more expensive and doesn't add as much value in the presence of big data. So, `TrainValidationSplit` is used here.

```
import org.apache.spark.ml.tuning.TrainValidationSplit

val validator = new TrainValidationSplit().
  setSeed(Random.nextLong()).
  setEstimator(pipeline).
  setEvaluator(multiclassEval).
  setEstimatorParamMaps(paramGrid).
  setTrainRatio(0.9)

val validatorModel = validator.fit(trainData)
```

This will take minutes or more, depending on your hardware, because it's building and evaluating many models. Note the train ratio parameter is set to 0.9. This means that the training data is actually further subdivided by `TrainValidationSplit` into 90%/10% subsets. The former is used for training each model. The remaining 10% of the input is held out as a cross-validation set to evaluate the model. If it's already holding out some data for evaluation, then why did we hold out 10% of the original data as a test set?

If the purpose of the CV set was to evaluate *parameters* that fit to the *training* set, then the purpose of the test set is to evaluate *hyperparameters* that were "fit" to the CV set. That is, the test set ensures an unbiased estimate of the accuracy of the final, chosen model and its hyperparameters.

Say that the best model chosen by this process exhibits 90% accuracy on the CV set. It seems reasonable to expect it will exhibit 90% accuracy on future data. However, there's an element of randomness in how these models are built. By chance, this model and evalua-

tion could have turned out unusually well. The top model and evaluation result could have

ples in the CV set that were used to evaluate it. That is why a third subset, the test set, was held out.

The result of the validator contains the best model it found. This itself is a representation of the best overall *pipeline* it found, because we provided an instance of a pipeline to run. In order to query the parameters chosen by `DecisionTreeClassifier`, it's necessary to manually extract `DecisionTreeClassificationModel` from the resulting `PipelineModel`, which is the final stage in the pipeline.

```
import org.apache.spark.ml.PipelineModel

val bestModel = validatorModel.bestModel
bestModel.asInstanceOf[PipelineModel].stages.last.extractParamMap

...
{
        dtc_9136220619b4-cacheNodeIds: false,
        dtc_9136220619b4-checkpointInterval: 10,
        dtc_9136220619b4-featuresCol: featureVector,
        dtc_9136220619b4-impurity: entropy,
        dtc_9136220619b4-labelCol: Cover_Type,
        dtc_9136220619b4-maxBins: 40,
        dtc_9136220619b4-maxDepth: 20,
        dtc_9136220619b4-maxMemoryInMB: 256,
        dtc_9136220619b4-minInfoGain: 0.0,
        dtc_9136220619b4-minInstancesPerNode: 1,
        dtc_9136220619b4-predictionCol: prediction,
        dtc_9136220619b4-probabilityCol: probability,
        dtc_9136220619b4-rawPredictionCol: rawPrediction,
        dtc_9136220619b4-seed: 159147643
}
```

This contains a lot of information about the fitted model, but it also tells us that "entropy" apparently worked best as the impurity measure and that a max depth of 20 was not surprisingly better than 1. It might be surprising that the best model was fit with just 40 bins, but this is probably a sign that 40 was "plenty" rather than "better" than 300. Lastly, no minimum information gain was better than a small minimum, which could imply that the model is more prone to underfit than overfit.

You may wonder if it is possible to see the accuracy that each of the models achieved for


```
val validatorModel = validator.fit(trainData)

val paramsAndMetrics = validatorModel.validationMetrics.
  zip(validatorModel.getEstimatorParamMaps).sortBy(-_._1)

paramsAndMetrics.foreach { case (metric, params) =>
    println(metric)
    println(params)
    println()
}

...
0.9138483377774368
{
        dtc_3e3b8bb692d1-impurity: entropy,
        dtc_3e3b8bb692d1-maxBins: 40,
        dtc_3e3b8bb692d1-maxDepth: 20,
        dtc_3e3b8bb692d1-minInfoGain: 0.0
}

0.9122369506416774
{
        dtc_3e3b8bb692d1-impurity: entropy,
        dtc_3e3b8bb692d1-maxBins: 300,
        dtc_3e3b8bb692d1-maxDepth: 20,
        dtc_3e3b8bb692d1-minInfoGain: 0.0
}
...
```

What was the accuracy that this model achieved on the CV set? And finally, what accuracy does the model achieve on the test set?

```
validatorModel.validationMetrics.max
multiclassEval.evaluate(bestModel.transform(testData)) ❶

...
0.9138483377774368
0.9139978718291971
```

❶ bestModel is a complete pipeline.

The results are both about 91%. It happens that the estimate from the CV set was pretty

fit the idiosyncrasies and noise of the training data too closely. This is a problem common to most machine learning algorithms, not just decision trees.

When a decision tree has overfit, it will exhibit high accuracy when run on the same training data that it fit the model to, but low accuracy on other examples. Here, the final model's accuracy was about 91% on other, new examples. Accuracy can just as easily be evaluated over the same data that the model was trained on, `trainData`. This gives an accuracy of about 95%.

The difference is not large but suggests that the decision tree has overfit the training data to some extent. A lower maximum depth might be a better choice.

Categorical Features Revisited

So far, the code examples have implicitly treated all input features as if they're numeric (though "Cover_Type", despite being encoded as numeric, has actually been correctly treated as a categorical value.) This isn't exactly wrong, because the categorical features here are one-hot encoded as several binary 0/1 values. Treating these individual features as numeric turns out to be fine, because any decision rule on the "numeric" features will choose thresholds between 0 and 1, and all are equivalent since all values are 0 or 1.

Of course, this encoding forces the decision tree algorithm to consider the values of the underlying categorical features individually. Because features like soil type are broken down into many features, and because decision trees treat features individually, it is harder to relate information about related soil types.

For example, nine different soil types are actually part of the Leighcan family, and they may be related in ways that the decision tree can exploit. If soil type were encoded as a single categorical feature with 40 soil values, then the tree could express rules like "if the soil type is one of the nine Leighton family types" directly. However, when encoded as 40 features, the tree would have to learn a sequence of nine decisions on soil type to do the same, this expressiveness may lead to better decisions and more efficient trees.

However, having 40 numeric features represent one 40-valued categorical feature increases memory usage and slows things down.

What about undoing the one-hot encoding? This would replace, for example, the four col-


```
def unencodeOneHot(data: DataFrame): DataFrame = {
  val wildernessCols = (0 until 4).map(i => s"Wilderness_Area_$i").toArray

  val wildernessAssembler = new VectorAssembler().
    setInputCols(wildernessCols).
    setOutputCol("wilderness")

  val unhotUDF = udf((vec: Vector) => vec.toArray.indexOf(1.0).toDouble) ❶

  val withWilderness = wildernessAssembler.transform(data).
    drop(wildernessCols:_*). ❷
    withColumn("wilderness", unhotUDF($"wilderness")) ❸

  val soilCols = (0 until 40).map(i => s"Soil_Type_$i").toArray

  val soilAssembler = new VectorAssembler().
    setInputCols(soilCols).
    setOutputCol("soil")

  soilAssembler.transform(withWilderness).
    drop(soilCols:_*).
    withColumn("soil", unhotUDF($"soil"))
}
```

❶ Note UDF definition

❷ Drop one-hot columns; no longer needed

❸ Overwrite column with numeric one of same name

Here `VectorAssembler` is deployed to combine the 4 and 40 wilderness and soil type columns into two `Vector` columns. The values in these `Vectors` are all 0, except for one location that has a 1. There's no simple DataFrame function for this, so we have to define our own UDF that can be used to operate on columns. This turns these two new columns into numbers of just the type we need.

From here, nearly the same process as above can be used to tune the hyperparameters of a decision tree model built on this data and to choose and evaluate a best model. There's one

important difference, however. The two new numeric columns have nothing about them

this data are generally hidden from the caller, but includes information such as whether the column encodes a categorical value and how many distinct values it takes on. In order to add this metadata, it's necessary to put the data through `VectorIndexer`. Its job is to turn input into properly labeled categorical feature columns. Although we did much of the work already to turn the categorical features into 0-indexed values, `VectorIndexer` will take care of the metadata.

We need to add this stage to the `Pipeline`:

```
import org.apache.spark.ml.feature.VectorIndexer

val inputCols = unencTrainData.columns.filter(_ != "Cover_Type")
val assembler = new VectorAssembler().
  setInputCols(inputCols).
  setOutputCol("featureVector")

val indexer = new VectorIndexer().
  setMaxCategories(40). ❶
  setInputCol("featureVector").
  setOutputCol("indexedVector")

val classifier = new DecisionTreeClassifier().
  setSeed(Random.nextLong()).
  setLabelCol("Cover_Type").
  setFeaturesCol("indexedVector").
  setPredictionCol("prediction")

val pipeline = new Pipeline().setStages(Array(assembler, indexer, classifier))
```

❶ >= 40 because soil has 40 values

The approach assumes that the training set contains all possible values of each of the categorical features at least once. That is, it works correctly only if all 4 soil values and all 40 wilderness values appear in the training set so that all possible values get a mapping. Here, that happens to be true, but may not be for small training sets of data in which some labels appear very infrequently. In those cases, it could be necessary to manually create and add a `VectorIndexerModel` with the complete value mapping supplied manually.

Aside from that, the process is the same as before. You should find that it chose a similar

If you have been following along with the code examples, you may have noticed that your results differ slightly from those presented in the code listings in the book. That is because there is an element of randomness in building decision trees, and the randomness comes into play when you're deciding what data to use and what decision rules to explore.

The algorithm does not consider every possible decision rule at every level. To do so would take an incredible amount of time. For a categorical feature over N values, there are 2^N-2 possible decision rules (every subset except the empty set and entire set). For even moderately large N, this would create billions of candidate decision rules.

Instead, decision trees use several heuristics to determine which few rules to actually consider. The process of picking rules also involves some randomness; only a few features picked at random are looked at each time, and only values from a random subset of the training data. This trades a bit of accuracy for a lot of speed, but it also means that the decision tree algorithm won't build the same tree every time. This is a good thing.

It's good for the same reason that the "wisdom of the crowds" usually beats individual predictions. To illustrate, take this quick quiz: How many black taxis operate in London?

Don't peek at the answer; guess first.

I guessed 10,000, which is well off the correct answer of about 19,000. Because I guessed low, you're a bit more likely to have guessed higher than I did, and so the average of our answers will tend to be more accurate. There's that regression to the mean again. The average guess from an informal poll of 13 people in the office was indeed closer: 11,170.

A key to this effect is that the guesses were independent and didn't influence one another. (You didn't peek, did you?) The exercise would be useless if we had all agreed on and used the same methodology to make a guess, because the guesses would have been the same answer—the same potentially quite wrong answer. It would even have been different and worse if I'd merely influenced you by stating my guess upfront.

It would be great to have not one tree, but many trees, each producing reasonable but different and independent estimations of the right target value. Their collective average prediction should fall close to the true answer, more than any individual tree's does. It's the

randomness in the process of building that helps create this independence. This is the key

set, then most trees will not consider this problem feature most of the time. Most trees will not fit the noise and will tend to "outvote" the trees that have fit the noise in the forest.

The prediction of a random decision forest is simply a weighted average of the trees' predictions. For a categorical target, this can be a majority vote or the most probable value based on the average of probabilities produced by the trees. Random decision forests, like decision trees, also support regression, and the forest's prediction in this case is the average of the number predicted by each tree.

While random decision forests are a more powerful and complex classification technique, the good news is that it's virtually no different to use it in the pipeline that has been developed in this chapter. Simply drop in a `RandomForestClassifier` in place of `DecisionTreeClassifier` and proceed as before. There's really no more code or API to understand in order to use it.

```
import org.apache.spark.ml.classification.RandomForestClassifier

val classifier = new RandomForestClassifier().
  setSeed(Random.nextLong()).

  setLabelCol("Cover_Type").
  setFeaturesCol("indexedVector").
  setPredictionCol("prediction")
```

Note that this classifier has another hyperparameter: the number of trees to build. Like the max bins hyperparameter, higher values should give better results up to a point. The cost, however, is that building many trees of course takes many times longer than building one.

The accuracy of the best random decision forest model produced from a similar tuning process is 95% off the bat—about 2% better already, although viewed another way, that's a 28% reduction in the error rate over the best decision tree built previously, from 7% down to 5%. You may do better with further tuning.

Incidentally, at this point we have a more reliable picture of feature importance:

```
import org.apache.spark.ml.classification.RandomForestClassificationModel
```

```
val forestModel = bestModel.asInstanceOf[PipelineModel].
```



```
(0.28877055118903183,Elevation)
(0.17288279582959612,soil)
(0.12105056811661499,Horizontal_Distance_To_Roadways)
(0.1121550648692802,Horizontal_Distance_To_Fire_Points)
(0.08805270405239551,wilderness)
(0.04467393191338021,Vertical_Distance_To_Hydrology)
(0.04293099150373547,Horizontal_Distance_To_Hydrology)
(0.03149644050848614,Hillshade_Noon)
(0.028408483578137605,Hillshade_9am)
(0.027185325937200706,Aspect)
(0.027075578474331806,Hillshade_3pm)
(0.015317564027809389,Slope)
```

Random decision forests are appealing in the context of big data because trees are supposed to be built independently, and big data technologies like Spark and MapReduce inherently need *data-parallel* problems, where parts of the overall solution can be computed independently on parts of the data. The fact that trees can, and should, train on only a subset of features or input data makes it trivial to parallelize building the trees.

Making Predictions

Building a classifier, while an interesting and nuanced process, is not the end goal. The goal is to make predictions. This is the payoff, and it is comparatively quite easy.

The resulting "best model" is actually a whole pipeline of operations, which encapsulate how input is transformed for use with the model and includes the model itself, which can make predictions. It can operate on a data frame of new input. The only difference from the `data` DataFrame we started with is that it lacks the "Cover_Type" column. When we're making predictions—especially about the future, says Mr. Bohr—the output is of course not known.

To prove it, try dropping the "Cover_Type" from the test data input and obtaining a prediction:

```
bestModel.transform(unencTestData.drop("Cover_Type")).select("prediction").show(

...
```

```
+----------+
```


The result should be 6.0, which corresponds to class 7 (the original feature was 1-indexed) in the original Covtype data set. The predicted cover type for the land described in this example is Krummholz. Obviously.

Where to Go from Here

This chapter introduced two related and important types of machine learning, classification and regression, along with some foundational concepts in building and tuning models: features, vectors, training, and cross-validation. It demonstrated how to predict a type of forest cover from things like location and soil type using the Covtype data set, with decision trees and forests implemented in Spark MLlib.

As with recommenders in Chapter 3, it could be useful to continue exploring the effect of hyperparameters on accuracy. Most decision tree hyperparameters trade time for accuracy: more bins and trees generally produce better accuracy but hit a point of diminishing returns.

The classifier here turned out to be very accurate. It's unusual to achieve more than 95% accuracy. In general, you will achieve further improvements in accuracy by including more features or transforming existing features into a more predictive form. This is a common, repeated step in iteratively improving a classifier model. For example, for this data set, the two features encoding horizontal and vertical distance-to-surface-water features could produce a third feature: straight-line distance-to-surface-water features. This might turn out to be more useful than either original feature. Or, if it were possible to collect more data, we might try adding new information like soil moisture in order to improve classification.

Of course, not all prediction problems in the real world are exactly like the Covtype data set. For example, some problems require predicting a continuous numeric value, not a categorical value. Much of the same analysis and code applies to this type of *regression* problem; the `RandomForestRegressor` class will be of use in this case.

Furthermore, decision trees and forests are not the only classification or regression algorithms, and not the only ones implemented in Spark MLlib. For classification, it includes implementations of:

- Naïve Bayes

- Gradient boosting

Yes, logistic regression is a classification technique. Underneath the hood, it classifies by predicting a continuous function of a class probability. This detail is not necessary to understand.

Each of these algorithms operates quite differently from decision trees and forests. However, many elements are the same: they plug into a `Pipeline` and operate on columns in a data frame, and have hyperparameters that you must select using training, cross-validation, and test subsets of the input data. The same general principles, with these other algorithms, can also be deployed to model classification and regression problems.

These have been examples of supervised learning. What happens when some, or all, of the target values are unknown? The following chapter will explore what can be done in this situation.