Chapter 10. Machine Learning with MLlib

Up until this point, we have focused on data engineering workloads with Apache Spark. Data engineering is often a precursory step to preparing your data for machine learning (ML) tasks, which will be the focus of this chapter. We live in an era in which machine learning and artificial intelligence applications are an integral part of our lives. Chances are that whether we realize it or not, every day we come into contact with ML models for purposes such as online shopping recommendations and advertisements, fraud detection, classification, image recognition, pattern matching, and more. These ML models drive important business decisions for many companies. According to this McKinsey study, 35% of what consumers purchase on Amazon and 75% of what they watch on Netflix is driven by machine learning–based product recommendations. Building a model that performs well can make or break companies.

In this chapter we will get you started building ML models using MLlib, the de facto machine learning library in Apache Spark. We'll begin with a brief introduction to machine learning, then cover best practices for distributed ML and feature engineering at scale (if you're already familiar with machine learning fundamentals, you can skip straight to "Designing Machine Learning Pipelines"). Through the short code snippets presented here and the notebooks available in the book's GitHub repo, you'll learn how to build basic ML models and use MLlib.

NOTE

This chapter covers the Scala and Python APIs; if you're interested in using R (`sparklyr`) with Spark for machine learning, we invite you to check out *Mastering Spark with R* by Javier Luraschi, Kevin Kuo, and Edgar Ruiz (O'Reilly).

What Is Machine Learning?

Machine learning is getting a lot of hype these days—but what is it, exactly? Broadly speaking, machine learning is a process for extracting patterns from your data, using statistics, linear algebra, and numerical optimization. Machine learning can be applied to problems such as predicting power consumption, determining whether or not there is a cat in your video, or clustering items with similar characteristics.

There are a few types of machine learning, including supervised, semi-supervised, unsupervised, and reinforcement learning. This chapter will mainly focus on supervised machine learning and just touch upon unsupervised learning. Before we dive in, let's briefly discuss the differences between supervised and unsupervised ML.

Supervised Learning

In supervised machine learning, your data consists of a set of input records, each of which has associated labels, and the goal is to predict the output label(s) given a new unlabeled input. These output labels can either be *discrete* or *continuous*, which brings us to the two types of supervised machine learning: *classification* and *regression*.

In a classification problem, the aim is to separate the inputs into a discrete set of classes or labels. With *binary* classification, there are two discrete labels you want to predict, such as "dog" or "not dog," as Figure 10-1 depicts.

Figure 10-1. Binary classification example: dog or not dog

With *multiclass*, also known as *multinomial*, classification, there can be three or more discrete labels, such as predicting the breed of a dog (e.g., Australian shepherd, golden retriever, or poodle, as shown in Figure 10-2).

Figure 10-2. Multinomial classification example: Australian shepherd, golden retriever, or poodle

In regression problems, the value to predict is a continuous number, not a label. This means you might predict values that your model hasn't seen during training, as shown in Figure 10-3. For example, you might build a model to predict the daily ice cream sales given the temperature. Your model might predict the value $77.67, even if none of the input/output pairs it was trained on contained that value.

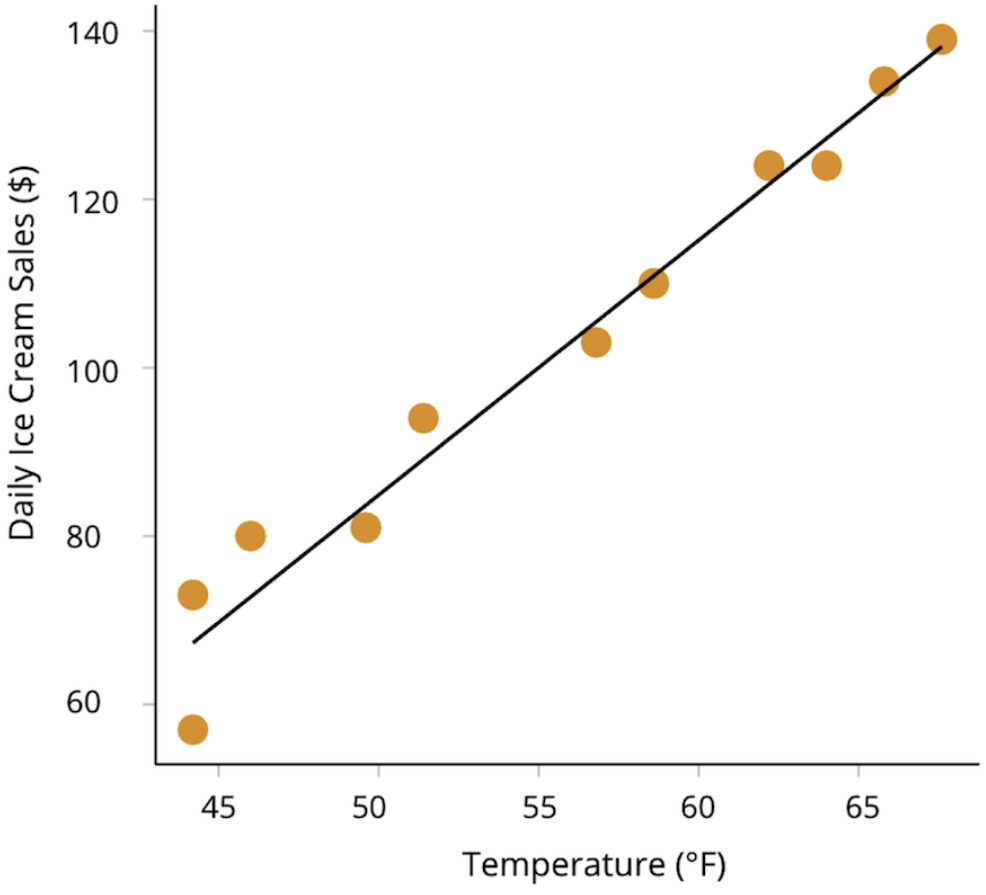

Figure 10-3. Regression example: predicting ice cream sales based on temperature

Table 10-1 lists some commonly used supervised ML algorithms that are available in Spark MLlib, with a note as to whether they can be used for regression, classification, or both.

Table 10-1. Popular classification and regression algorithms

Algorithm	Typical usage
Linear regression	Regression
Logistic regression	Classification (we know, it has regression in the name!)
Decision trees	Both
Gradient boosted trees	Both
Random forests	Both
Naive Bayes	Classification
Support vector machines (SVMs)	Classification

Unsupervised Learning

Obtaining the labeled data required by supervised machine learning can be very expensive and/or infeasible. This is where unsupervised machine learning comes into play. Instead of predicting a label, unsupervised ML helps you to better understand the structure of your data.

As an example, consider the original unclustered data on the left in Figure 10-4. There is no known true label for each of these data points (x_1, x_2), but by applying unsupervised machine learning to our data we can find the clusters that naturally form, as shown on the right.

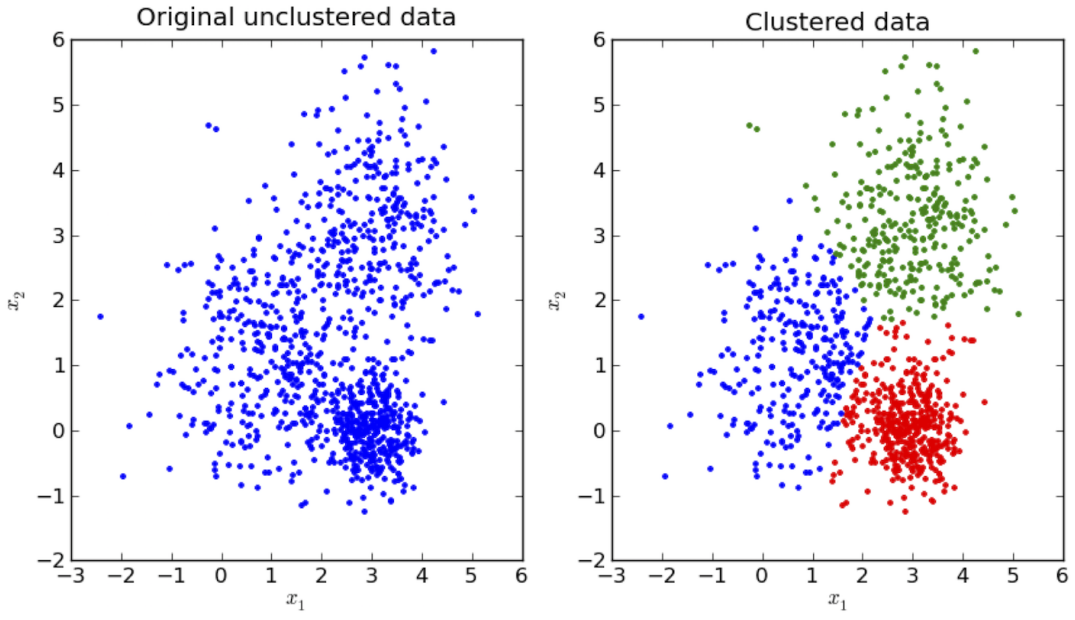

Figure 10-4. Clustering example

Unsupervised machine learning can be used for outlier detection or as a preprocessing step for supervised machine learning—for example, to reduce the dimensionality (i.e., number of dimensions per datum) of the data set, which is useful for reducing storage requirements or simplifying downstream tasks. Some unsupervised machine learning algorithms in MLlib include *k*-means, Latent Dirichlet Allocation (LDA), and Gaussian mixture models.

Why Spark for Machine Learning?

Spark is a unified analytics engine that provides an ecosystem for data ingestion, feature engineering, model training, and deployment. Without Spark, developers would need many disparate tools to accomplish this set of tasks, and might still struggle with scalability.

Spark has two machine learning packages: `spark.mllib` and `spark.ml`. `spark.mllib` is the original machine learning API, based on the RDD API (which has been in maintenance mode since Spark 2.0), while `spark.ml` is the newer API, based on DataFrames. The rest of this chapter will focus on using the `spark.ml` package and how to design machine learning pipelines in Spark. However, we use "MLlib" as an umbrella term to refer to both machine learning library packages in Apache Spark.

With `spark.ml`, data scientists can use one ecosystem for their data preparation and model building, without the need to downsample their data to fit on a single machine. `spark.ml` focuses on O(n) scale-out, where the model scales linearly with the number of data points you have, so it can scale to massive amounts of data. In the following chapter, we will discuss some of the trade-offs involved in choosing between a distributed framework such as `spark.ml` and a single-node framework like `scikit-learn` (`sklearn`). If you have previously used `scikit-learn`, many of the APIs in `spark.ml` will feel quite familiar, but there are some subtle differences that we will discuss.

Designing Machine Learning Pipelines

In this section, we will cover how to create and tune ML pipelines. The concept of pipelines is common across many ML frameworks as a way to organize a series of operations to apply to your data. In MLlib, the Pipeline API provides a high-level API built on top of DataFrames to organize your machine learning workflow. The Pipeline API is composed of a series of transformers and estimators, which we will discuss in-depth later.

Throughout this chapter, we will use the San Francisco housing data set from Inside Airbnb. It contains information about Airbnb rentals in San Francisco, such as the number of bedrooms, location, review scores, etc., and our goal is to build a model to predict the nightly rental prices for listings in that city. This is a regression problem, because price is a continuous variable. We will guide you through the workflow a data scientist would go through to approach this problem, including feature engineering, building models, hyperparameter tuning, and evaluating model performance. This data set is quite messy and can be difficult to model (like most real-world data sets!), so if you are experimenting on your own, don't feel bad if your early models aren't great.

The intent of this chapter is not to show you every API in MLlib, but rather to equip you with the skills and knowledge to get started with using MLlib to build end-to-end pipelines. Before going into the details, let's define some MLlib terminology:

Transformer

Accepts a DataFrame as input, and returns a new DataFrame with one or more columns appended to it. Transformers do not learn any parameters from your data and simply apply rule-based transformations to either prepare data for model training or generate predictions using a trained MLlib model. They have a `.transform()` method.

Estimator

Learns (or "fits") parameters from your DataFrame via a `.fit()` method and returns a `Model`, which is a transformer.

Pipeline

Organizes a series of transformers and estimators into a single model. While pipelines themselves are estimators, the output of `pipeline.fit()` returns a `PipelineModel`, a transformer.

While these concepts may seem rather abstract right now, the code snippets and examples in this chapter will help you understand how they all come together. But before we can build our ML models and use transformers, estimators, and pipelines, we need to load in our data and perform some data preparation.

Data Ingestion and Exploration

We have slightly preprocessed the data in our example data set to remove outliers (e.g., Airbnbs posted for $0/night), converted all integers to doubles, and selected an informative subset of the more than one hundred fields. Further, for any missing numerical values in our data columns, we have imputed the median value and added an indicator column (the column name followed by `_na`, such as `bedrooms_na`). This way the ML model or human analyst can interpret any value in that column as an imputed value, not a true value. You can see the data preparation notebook in the book's GitHub repo. Note there are many other ways to handle missing values, which are outside the scope of this book.

Let's take a quick peek at the data set and the corresponding schema (with the output showing just a subset of the columns):

```
# In Python
filePath = """/databricks-datasets/learning-spark-v2/sf-airbnb/
sf-airbnb-clean.parquet/"""
```

```
airbnbDF = spark.read.parquet(filePath)
airbnbDF.select("neighbourhood_cleansed", "room_type", "bedrooms", "bathrooms",
                "number_of_reviews", "price").show(5)
```

```
// In Scala
val filePath =
  "/databricks-datasets/learning-spark-v2/sf-airbnb/sf-airbnb-clean.parquet/"
val airbnbDF = spark.read.parquet(filePath)
airbnbDF.select("neighbourhood_cleansed", "room_type", "bedrooms", "bathrooms",
                "number_of_reviews", "price").show(5)
```

```
+----------------------+---------------+--------+---------+----------+-----+
|neighbourhood_cleansed|      room_type|bedrooms|bathrooms|number_...|price|
+----------------------+---------------+--------+---------+----------+-----+
|      Western Addition|Entire home/apt|     1.0|      1.0|     180.0|170.0|
|        Bernal Heights|Entire home/apt|     2.0|      1.0|     111.0|235.0|
|        Haight Ashbury|   Private room|     1.0|      4.0|      17.0| 65.0|
|        Haight Ashbury|   Private room|     1.0|      4.0|       8.0| 65.0|
|      Western Addition|Entire home/apt|     2.0|      1.5|      27.0|785.0|
+----------------------+---------------+--------+---------+----------+-----+
```

Our goal is to predict the price per night for a rental property, given our features.

NOTE

Before data scientists can get to model building, they need to explore and understand their data. They will often use Spark to group their data, then use data visualization libraries such as matplotlib to visualize the data. We will leave data exploration as an exercise for the reader.

Creating Training and Test Data Sets

Before we begin feature engineering and modeling, we will divide our data set into two groups: *train* and *test*. Depending on the size of your data set, your train/test ratio may vary, but many data scientists use 80/20 as a standard train/test split. You might be wondering, "Why not use the entire data set to train the model?" The problem is that if we built a model on the entire data set, it's possible that the model would memorize

or “overfit” to the training data we provided, and we would have no more data with which to evaluate how well it generalizes to previously unseen data. The model’s performance on the test set is a proxy for how well it will perform on unseen data (i.e., in the wild or in production), assuming that data follows similar distributions. This split is depicted in Figure 10-5.

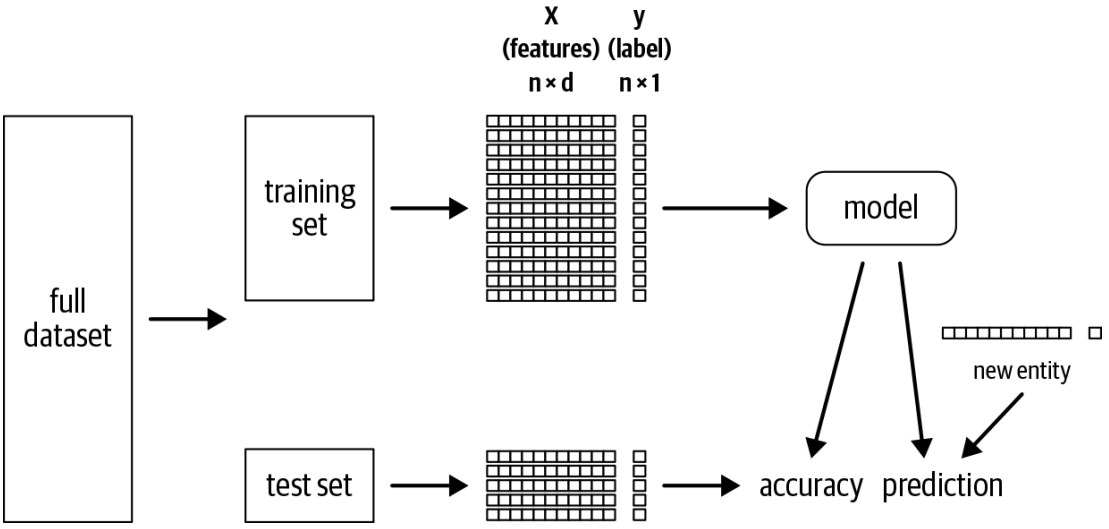

Figure 10-5. Train/test split

Our training set consists of a set of features, X, and a label, y. Here we use capital X to denote a matrix with dimensions *n* x *d*, where *n* is the number of data points (or examples) and *d* is the number of features (this is what we call the fields or columns in our DataFrame). We use lowercase y to denote a vector, with dimensions *n* x 1; for every example, there is one label.

Different metrics are used to measure the performance of the model. For classification problems, a standard metric is the *accuracy*, or percentage, of correct predictions. Once the model has satisfactory performance on the training set using that metric, we will apply the model to our test set. If it performs well on our test set according to our evaluation metrics, then we can feel confident that we have built a model that will “generalize” to unseen data.

For our Airbnb data set, we will keep 80% for the training set and set aside 20% of our data for the test set. Further, we will set a random seed for reproducibility, such that if we rerun this code we will get the same data points going to our train and test data sets, respectively. The value of

the seed itself *shouldn't* matter, but data scientists often like setting it to 42 as that is the answer to the Ultimate Question of Life:

```
# In Python
trainDF, testDF = airbnbDF.randomSplit([.8, .2], seed=42)
print(f"""There are {trainDF.count()} rows in the training set,
and {testDF.count()} in the test set""")


// In Scala
val Array(trainDF, testDF) = airbnbDF.randomSplit(Array(.8, .2), seed=42)
println(f"""There are ${trainDF.count} rows in the training set, and
${testDF.count} in the test set""")
```

This produces the following output:

```
There are 5780 rows in the training set, and 1366 in the test set
```

But what happens if we change the number of executors in our Spark cluster? The Catalyst optimizer determines the optimal way to partition your data as a function of your cluster resources and size of your data set. Given that data in a Spark DataFrame is row-partitioned and each worker performs its split independently of the other workers, if the data in the partitions changes, then the result of the split (by `randomSplit()`) won't be the same.

While you could fix your cluster configuration and your seed to ensure that you get consistent results, our recommendation is to split your data once, then write it out to its own train/test folder so you don't have these reproducibility issues.

NOTE

During your exploratory analysis, you should cache the training data set because you will be accessing it many times throughout the machine learning process. Please reference the section on "Caching and Persistence of Data" in Chapter 7.

Preparing Features with Transformers

Now that we have split our data into training and test sets, let's prepare the data to build a linear regression model predicting price given the number of bedrooms. In a later example, we will include all of the relevant features, but for now let's make sure we have the mechanics in place. Linear regression (like many other algorithms in Spark) requires that all the input features are contained within a single vector in your DataFrame. Thus, we need to *transform* our data.

Transformers in Spark accept a DataFrame as input and return a new DataFrame with one or more columns appended to it. They do not learn from your data, but apply rule-based transformations using the `transform()` method.

For the task of putting all of our features into a single vector, we will use the `VectorAssembler` transformer. `VectorAssembler` takes a list of input columns and creates a new DataFrame with an additional column, which we will call `features`. It combines the values of those input columns into a single vector:

```
# In Python
from pyspark.ml.feature import VectorAssembler
vecAssembler = VectorAssembler(inputCols=["bedrooms"], outputCol="features")
vecTrainDF = vecAssembler.transform(trainDF)
vecTrainDF.select("bedrooms", "features", "price").show(10)
```

```
// In Scala
import org.apache.spark.ml.feature.VectorAssembler
val vecAssembler = new VectorAssembler()
  .setInputCols(Array("bedrooms"))
  .setOutputCol("features")
val vecTrainDF = vecAssembler.transform(trainDF)
vecTrainDF.select("bedrooms", "features", "price").show(10)
```

```
+--------+--------+-----+
|bedrooms|features|price|
+--------+--------+-----+
|     1.0|   [1.0]|200.0|
|     1.0|   [1.0]|130.0|
```

```
|     1.0|   [1.0]| 95.0|
|     1.0|   [1.0]|250.0|
|     3.0|   [3.0]|250.0|
|     1.0|   [1.0]|115.0|
|     1.0|   [1.0]|105.0|
|     1.0|   [1.0]| 86.0|
|     1.0|   [1.0]|100.0|
|     2.0|   [2.0]|220.0|
+--------+--------+-----+
```

You'll notice that in the Scala code, we had to instantiate the new `VectorAssembler` object as well as using setter methods to change the input and output columns. In Python, you have the option to pass the parameters directly to the constructor of `VectorAssembler` or to use the setter methods, but in Scala you can only use the setter methods.

We cover the fundamentals of linear regression next, but if you are already familiar with the algorithm, please skip to "Using Estimators to Build Models".

Understanding Linear Regression

Linear regression models a linear relationship between your dependent variable (or label) and one or more independent variables (or features). In our case, we want to fit a linear regression model to predict the price of an Airbnb rental given the number of bedrooms.

In Figure 10-6, we have a single feature x and an output y (this is our dependent variable). Linear regression seeks to fit an equation for a line to x and y, which for scalar variables can be expressed as $y = mx + b$, where m is the slope and b is the offset or intercept.

The dots indicate the true (x, y) pairs from our data set, and the solid line indicates the line of best fit for this data set. The data points do not perfectly line up, so we usually think of linear regression as fitting a model to $y \approx mx + b + \varepsilon$, where ε (epsilon) is an error drawn independently per record x from some distribution. These are the errors between our model predictions and the true values. Often we think of ε as being Gaussian, or normally distributed. The vertical lines above the regression line indicate

positive ε (or residuals), where your true values are above the predicted values, and the vertical lines below the regression line indicate negative residuals. The goal of linear regression is to find a line that minimizes the square of these residuals. You'll notice that the line can extrapolate predictions for data points it hasn't seen.

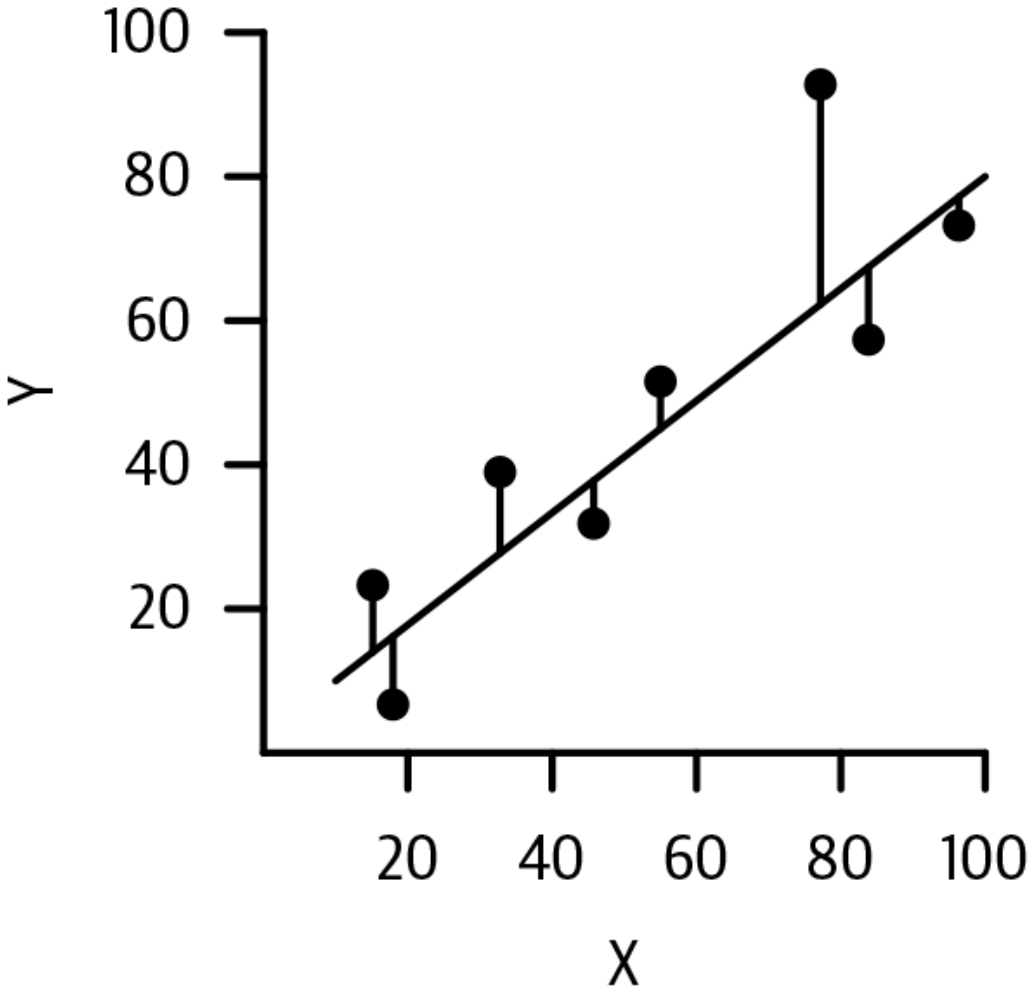

Figure 10-6. Univariate linear regression

Linear regression can also be extended to handle multiple independent variables. If we had three features as input, $x = [x_1, x_2, x_3]$, then we could model y as $y \approx w_0 + w_1x_1 + w_2x_2 + w_3x_3 + \varepsilon$. In this case, there is a separate coefficient (or weight) for each feature and a single intercept (w_0 instead of b here). The process of estimating the coefficients and intercept for our model is called *learning* (or *fitting*) the parameters for the model. For right now, we'll focus on the univariate regression example of predicting price given the number of bedrooms, and we'll get back to multivariate linear regression in a bit.

Using Estimators to Build Models

After setting up our `vectorAssembler`, we have our data prepared and transformed into a format that our linear regression model expects. In Spark, `LinearRegression` is a type of estimator—it takes in a DataFrame and returns a `Model`. Estimators learn parameters from your data, have an `estimator_name.fit()` method, and are eagerly evaluated (i.e., kick off Spark jobs), whereas transformers are lazily evaluated. Some other examples of estimators include `Imputer`, `DecisionTreeClassifier`, and `RandomForestRegressor`.

You'll notice that our input column for linear regression (`features`) is the output from our `vectorAssembler`:

```
# In Python
from pyspark.ml.regression import LinearRegression
lr = LinearRegression(featuresCol="features", labelCol="price")
lrModel = lr.fit(vecTrainDF)
```

```
// In Scala
import org.apache.spark.ml.regression.LinearRegression
val lr = new LinearRegression()
  .setFeaturesCol("features")
  .setLabelCol("price")

val lrModel = lr.fit(vecTrainDF)
```

`lr.fit()` returns a `LinearRegressionModel` (`lrModel`), which is a transformer. In other words, the output of an estimator's `fit()` method is a transformer. Once the estimator has learned the parameters, the transformer can apply these parameters to new data points to generate predictions. Let's inspect the parameters it learned:

```
# In Python
m = round(lrModel.coefficients[0], 2)
b = round(lrModel.intercept, 2)
print(f"""The formula for the linear regression line is
price = {m}*bedrooms + {b}""")
```

```
// In Scala
val m = lrModel.coefficients(0)
val b = lrModel.intercept
println(f"""The formula for the linear regression line is
price = $m%1.2f*bedrooms + $b%1.2f""")
```

This prints:

```
The formula for the linear regression line is price = 123.68*bedrooms + 47.51
```

Creating a Pipeline

If we want to apply our model to our test set, then we need to prepare that data in the same way as the training set (i.e., pass it through the vector assembler). Oftentimes data preparation pipelines will have multiple steps, and it becomes cumbersome to remember not only which steps to apply, but also the ordering of the steps. This is the motivation for the Pipeline API: you simply specify the stages you want your data to pass through, in order, and Spark takes care of the processing for you. They provide the user with better code reusability and organization. In Spark, `Pipeline`s are estimators, whereas `PipelineModel`s—fitted `Pipeline`s—are transformers.

Let's build our pipeline now:

```
# In Python
from pyspark.ml import Pipeline
pipeline = Pipeline(stages=[vecAssembler, lr])
pipelineModel = pipeline.fit(trainDF)
```

```
// In Scala
import org.apache.spark.ml.Pipeline
val pipeline = new Pipeline().setStages(Array(vecAssembler, lr))
val pipelineModel = pipeline.fit(trainDF)
```

Another advantage of using the Pipeline API is that it determines which stages are estimators/transformers for you, so you don't have to worry

about specifying *name*`.fit()` versus *name*`.transform()` for each of the stages.

Since `pipelineModel` is a transformer, it is straightforward to apply it to our test data set too:

```
# In Python
predDF = pipelineModel.transform(testDF)
predDF.select("bedrooms", "features", "price", "prediction").show(10)
```

```
// In Scala
val predDF = pipelineModel.transform(testDF)
predDF.select("bedrooms", "features", "price", "prediction").show(10)
```

```
+--------+--------+------+------------------+
|bedrooms|features| price|        prediction|
+--------+--------+------+------------------+
|     1.0|   [1.0]|  85.0|171.18598011578285|
|     1.0|   [1.0]|  45.0|171.18598011578285|
|     1.0|   [1.0]|  70.0|171.18598011578285|
|     1.0|   [1.0]| 128.0|171.18598011578285|
|     1.0|   [1.0]| 159.0|171.18598011578285|
|     2.0|   [2.0]| 250.0|294.86172649777757|
|     1.0|   [1.0]|  99.0|171.18598011578285|
|     1.0|   [1.0]|  95.0|171.18598011578285|
|     1.0|   [1.0]| 100.0|171.18598011578285|
|     1.0|   [1.0]|2010.0|171.18598011578285|
+--------+--------+------+------------------+
```

In this code we built a model using only a single feature, `bedrooms` (you can find the notebook for this chapter in the book's GitHub repo). However, you may want to build a model using all of your features, some of which may be categorical, such as `host_is_superhost`. Categorical features take on discrete values and have no intrinsic ordering—examples include occupations or country names. In the next section we'll consider a solution for how to treat these kinds of variables, known as *one-hot encoding*.

One-hot encoding

In the pipeline we just created, we only had two stages, and our linear regression model only used one feature. Let's take a look at how to build a slightly more complex pipeline that incorporates all of our numeric and categorical features.

Most machine learning models in MLlib expect numerical values as input, represented as vectors. To convert categorical values into numeric values, we can use a technique called one-hot encoding (OHE). Suppose we have a column called `Animal` and we have three types of animals: `Dog`, `Cat`, and `Fish`. We can't pass the string types into our ML model directly, so we need to assign a numeric mapping, such as this:

```
Animal = {"Dog", "Cat", "Fish"}
"Dog" = 1, "Cat" = 2, "Fish" = 3
```

However, using this approach we've introduced some spurious relationships into our data set that weren't there before. For example, why did we assign `Cat` twice the value of `Dog`? The numeric values we use should not introduce any relationships into our data set. Instead, we want to create a separate column for every distinct value in our `Animal` column:

```
"Dog" = [ 1, 0, 0]
"Cat" = [ 0, 1, 0]
"Fish" = [0, 0, 1]
```

If the animal is a dog, it has a one in the first column and zeros elsewhere. If it is a cat, it has a one in the second column and zeros elsewhere. The ordering of the columns is irrelevant. If you've used pandas before, you'll note that this does the same thing as `pandas.get_dummies()`.

If we had a zoo of 300 animals, would OHE massively increase consumption of memory/compute resources? Not with Spark! Spark internally uses a `SparseVector` when the majority of the entries are `0`, as is often the case after OHE, so it does not waste space storing `0` values. Let's take a look at an example to better understand how `SparseVector`s work:

```
DenseVector(0, 0, 0, 7, 0, 2, 0, 0, 0, 0)
SparseVector(10, [3, 5], [7, 2])
```

The `DenseVector` in this example contains 10 values, all but 2 of which are `0`. To create a `SparseVector`, we need to keep track of the size of the vector, the indices of the nonzero elements, and the corresponding values at those indices. In this example the size of the vector is 10, there are two nonzero values at indices 3 and 5, and the corresponding values at those indices are 7 and 2.

There are a few ways to one-hot encode your data with Spark. A common approach is to use the `StringIndexer` and `OneHotEncoder`. With this approach, the first step is to apply the `StringIndexer` estimator to convert categorical values into category indices. These category indices are ordered by label frequencies, so the most frequent label gets index 0, which provides us with reproducible results across various runs of the same data.

Once you have created your category indices, you can pass those as input to the `OneHotEncoder` (`OneHotEncoderEstimator` if using Spark 2.3/2.4). The `OneHotEncoder` maps a column of category indices to a column of binary vectors. Take a look at Table 10-2 to see the differences in the `StringIndexer` and `OneHotEncoder` APIs from Spark 2.3/2.4 to 3.0.

Table 10-2. StringIndexer and OneHotEncoder changes in Spark 3.0

	Spark 2.3 and 2.4	**Spark 3.0**
`StringIndexer`	Single column as input/output	Multiple columns as input/output
`OneHotEncoder`	Deprecated	Multiple columns as input/output
`OneHotEncoderEstimator`	Multiple columns as input/output	N/A

The following code demonstrates how to one-hot encode our categorical features. In our data set, any column of type `string` is treated as a categorical feature, but sometimes you might have numeric features you want treated as categorical or vice versa. You'll need to carefully identify which columns are numeric and which are categorical:

```
# In Python
from pyspark.ml.feature import OneHotEncoder, StringIndexer

categoricalCols = [field for (field, dataType) in trainDF.dtypes
                   if dataType == "string"]
indexOutputCols = [x + "Index" for x in categoricalCols]
oheOutputCols = [x + "OHE" for x in categoricalCols]

stringIndexer = StringIndexer(inputCols=categoricalCols,
                              outputCols=indexOutputCols,
                              handleInvalid="skip")
oheEncoder = OneHotEncoder(inputCols=indexOutputCols,
                           outputCols=oheOutputCols)

numericCols = [field for (field, dataType) in trainDF.dtypes
               if ((dataType == "double") & (field != "price"))]
assemblerInputs = oheOutputCols + numericCols
vecAssembler = VectorAssembler(inputCols=assemblerInputs,
                               outputCol="features")


// In Scala
import org.apache.spark.ml.feature.{OneHotEncoder, StringIndexer}

val categoricalCols = trainDF.dtypes.filter(_._2 == "StringType").map(_._1)
val indexOutputCols = categoricalCols.map(_ + "Index")
val oheOutputCols = categoricalCols.map(_ + "OHE")

val stringIndexer = new StringIndexer()
  .setInputCols(categoricalCols)
  .setOutputCols(indexOutputCols)
  .setHandleInvalid("skip")

val oheEncoder = new OneHotEncoder()
  .setInputCols(indexOutputCols)
  .setOutputCols(oheOutputCols)
```

```
val numericCols = trainDF.dtypes.filter{ case (field, dataType) =>
  dataType == "DoubleType" && field != "price"}.map(_._1)
val assemblerInputs = oheOutputCols ++ numericCols
val vecAssembler = new VectorAssembler()
  .setInputCols(assemblerInputs)
  .setOutputCol("features")
```

Now you might be wondering, "How does the `StringIndexer` handle new categories that appear in the test data set, but not in the training data set?" There is a `handleInvalid` parameter that specifies how you want to handle them. The options are `skip` (filter out rows with invalid data), `error` (throw an error), or `keep` (put invalid data in a special additional bucket, at index `numLabels`). For this example, we just skipped the invalid records.

One difficulty with this approach is that you need to tell `StringIndexer` explicitly which features should be treated as categorical features. You could use `VectorIndexer` to automatically detect all the categorical variables, but it is computationally expensive as it has to iterate over every single column and detect if it has fewer than `maxCategories` distinct values. `maxCategories` is a parameter the user specifies, and determining this value can also be difficult.

Another approach is to use `RFormula`. The syntax for this is inspired by the R programming language. With `RFormula`, you provide your label and which features you want to include. It supports a limited subset of the R operators, including `~`, `.`, `:`, `+`, and `-`. For example, you might specify `formula = "y ~ bedrooms + bathrooms"`, which means to predict `y` given just `bedrooms` and `bathrooms`, or `formula = "y ~ ."`, which means to use all of the available features (and automatically excludes `y` from the features). `RFormula` will automatically `StringIndex` and OHE all of your `string` columns, convert your numeric columns to `double` type, and combine all of these into a single vector using `VectorAssembler` under the hood. Thus, we can replace all of the preceding code with a single line, and we will get the same result:

```
# In Python
from pyspark.ml.feature import RFormula
```

```
rFormula = RFormula(formula="price ~ .",
                    featuresCol="features",
                    labelCol="price",
                    handleInvalid="skip")
```

```
// In Scala
import org.apache.spark.ml.feature.RFormula

val rFormula = new RFormula()
  .setFormula("price ~ .")
  .setFeaturesCol("features")
  .setLabelCol("price")
  .setHandleInvalid("skip")
```

The downside of `RFormula` automatically combining the `StringIndexer` and `OneHotEncoder` is that one-hot encoding is not required or recommended for all algorithms. For example, tree-based algorithms can handle categorical variables directly if you just use the `StringIndexer` for the categorical features. You do not need to one-hot encode categorical features for tree-based methods, and it will often make your tree-based models worse. Unfortunately, there is no one-size-fits-all solution for feature engineering, and the ideal approach is closely related to the downstream algorithms you plan to apply to your data set.

NOTE

If someone else performs the feature engineering for you, make sure they document how they generated those features.

Once you've written the code to transform your data set, you can add a linear regression model using all of the features as input.

Here, we put all the feature preparation and model building into the pipeline, and apply it to our data set:

```
# In Python
lr = LinearRegression(labelCol="price", featuresCol="features")
pipeline = Pipeline(stages = [stringIndexer, oheEncoder, vecAssembler, lr])
```

```
# Or use RFormula
# pipeline = Pipeline(stages = [rFormula, lr])

pipelineModel = pipeline.fit(trainDF)
predDF = pipelineModel.transform(testDF)
predDF.select("features", "price", "prediction").show(5)
```

```
// In Scala
val lr = new LinearRegression()
  .setLabelCol("price")
  .setFeaturesCol("features")
val pipeline = new Pipeline()
  .setStages(Array(stringIndexer, oheEncoder, vecAssembler, lr))
// Or use RFormula
// val pipeline = new Pipeline().setStages(Array(rFormula, lr))

val pipelineModel = pipeline.fit(trainDF)
val predDF = pipelineModel.transform(testDF)
predDF.select("features", "price", "prediction").show(5)
```

```
+--------------------+-----+------------------+
|            features|price|        prediction|
+--------------------+-----+------------------+
|(98,[0,3,6,7,23,4...| 85.0| 55.80250714362137|
|(98,[0,3,6,7,23,4...| 45.0| 22.74720286761658|
|(98,[0,3,6,7,23,4...| 70.0|27.115811183814913|
|(98,[0,3,6,7,13,4...|128.0|-91.60763412465076|
|(98,[0,3,6,7,13,4...|159.0| 94.70374072351933|
+--------------------+-----+------------------+
```

As you can see, the features column is represented as a `SparseVector`. There are 98 features after one-hot encoding, followed by the nonzero indices and then the values themselves. You can see the whole output if you pass in `truncate=False` to `show()`.

How is our model performing? You can see that while some of the predictions might be considered "close," others are far off (a negative price for a rental!?). Next, we'll numerically evaluate how well our model performs across our entire test set.

Evaluating Models

Now that we have built a model, we need to evaluate how well it performs. In `spark.ml` there are classification, regression, clustering, and ranking evaluators (introduced in Spark 3.0). Given that this is a regression problem, we will use root-mean-square error (RMSE) and R^2 (pronounced "R-squared") to evaluate our model's performance.

RMSE

RMSE is a metric that ranges from zero to infinity. The closer it is to zero, the better.

Let's walk through the mathematical formula step by step:

1. Compute the difference (or error) between the true value y_i and the predicted value $\hat{y}_i$ (pronounced *y*-hat, where the "hat" indicates that it is a predicted value of the quantity under the hat):

$$\text{Error} = (y_i - \hat{y}_i)$$

2. Square the difference between y_i and $\hat{y}_i$ so that our positive and negative residuals do not cancel out. This is known as the squared error:

$$\text{Square Error (SE)} = (y_i - \hat{y}_i)^2$$

3. Then we sum up the squared error for all n of our data points, known as the sum of squared errors (SSE) or sum of squared residuals:

$$\text{Sum of Squared Errors (SSE)} = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

4. However, the SSE grows with the number of records n in the data set, so we want to normalize it by the number of records. The gives us the mean-squared error (MSE), a very commonly used regression metric:

$$\text{Mean Squared Error (MSE)} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

5. If we stop at MSE, then our error term is on the scale of $unit^2$. We'll often take the square root of the MSE to get the error back on the

scale of the original unit, which gives us the root-mean-square error (RMSE):

$$\text{Root Mean Squared Error (RMSE)} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

Let's evaluate our model using RMSE:

```
# In Python
from pyspark.ml.evaluation import RegressionEvaluator
regressionEvaluator = RegressionEvaluator(
  predictionCol="prediction",
  labelCol="price",
  metricName="rmse")
rmse = regressionEvaluator.evaluate(predDF)
print(f"RMSE is {rmse:.1f}")
```

```
// In Scala
import org.apache.spark.ml.evaluation.RegressionEvaluator
val regressionEvaluator = new RegressionEvaluator()
  .setPredictionCol("prediction")
  .setLabelCol("price")
  .setMetricName("rmse")
val rmse = regressionEvaluator.evaluate(predDF)
println(f"RMSE is $rmse%.1f")
```

This produces the following output:

```
RMSE is 220.6
```

Interpreting the value of RMSE

So how do we know if 220.6 is a good value for the RMSE? There are various ways to interpret this value, one of which is to build a simple baseline model and compute its RMSE to compare against. A common baseline model for regression tasks is to compute the average value of the label on the training set $\bar{y}$ (pronounced y-bar), then predict $\bar{y}$ for every record in the test data set and compute the resulting RMSE (example code is available in the book's GitHub repo). If you try this, you will see that our base-

line model has an RMSE of 240.7, so we beat our baseline. If you don't beat the baseline, then something probably went wrong in your model building process.

NOTE

If this were a classification problem, you might want to predict the most prevalent class as your baseline model.

Keep in mind that the unit of your label directly impacts your RMSE. For example, if your label is height, then your RMSE will be higher if you use centimeters rather than meters as your unit of measurement. You could arbitrarily decrease the RMSE by using a different unit, which is why it is important to compare your RMSE against a baseline.

There are also some metrics that naturally give you an intuition of how you are performing against a baseline, such as R^2, which we discuss next.

R^2

Despite the name R^2 containing "squared," R^2 values range from negative infinity to 1. Let's take a look at the math behind this metric. R^2 is computed as follows:

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

where SS_{tot} is the total sum of squares if you always predict $\bar{y}$:

$$SS_{tot} = \sum_{i=1}^{n} (y_i - \overline{y})^2$$

and SS_{res} is the sum of residuals squared from your model predictions (also known as the sum of squared errors, which we used to compute the RMSE):

$$SS_{res} = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

If your model perfectly predicts every data point, then your SS_{res} = 0, making your R^2 = 1. And if your SS_{res} = SS_{tot}, then the fraction is 1/1, so

your R^2 is 0. This is what happens if your model performs the same as always predicting the average value, $\bar{y}$.

But what if your model performs worse than always predicting $\bar{y}$ and your SS_{res} is really large? Then your R^2 can actually be negative! If your R^2 is negative, you should reevaluate your modeling process. The nice thing about using R^2 is that you don't necessarily need to define a baseline model to compare against.

If we want to change our regression evaluator to use R^2, instead of redefining the regression evaluator, we can set the metric name using the setter property:

```
# In Python
r2 = regressionEvaluator.setMetricName("r2").evaluate(predDF)
print(f"R2 is {r2}")
```

```
// In Scala
val r2 = regressionEvaluator.setMetricName("r2").evaluate(predDF)
println(s"R2 is $r2")
```

The output is:

```
R2 is 0.159854
```

Our R^2 is positive, but it's very close to 0. One of the reasons why our model is not performing too well is because our label, `price`, appears to be log-normally distributed. If a distribution is log-normal, it means that if we take the logarithm of the value, the result looks like a normal distribution. Price is often log-normally distributed. If you think about rental prices in San Francisco, most cost around $200 per night, but there are some that rent for thousands of dollars a night! You can see the distribution of our Airbnb prices for our training Dataset in Figure 10-7.

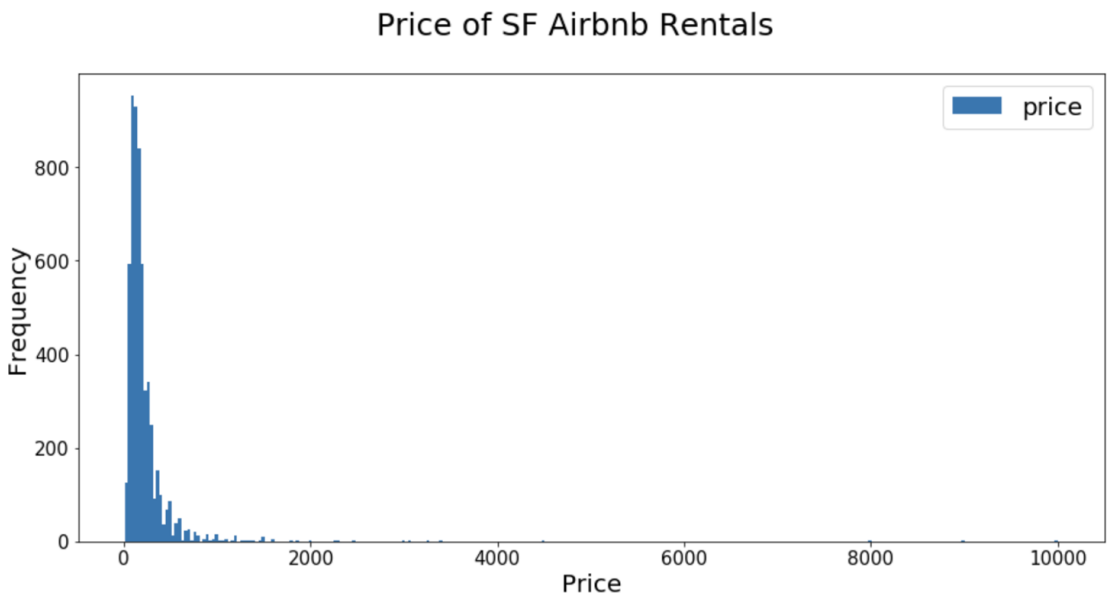

Figure 10-7. San Francisco housing price distribution

Let's take a look at the resulting distribution if we instead look at the log of the price (Figure 10-8).

Figure 10-8. San Francisco housing log-price distribution

You can see here that our log-price distribution looks a bit more like a normal distribution. As an exercise, try building a model to predict price on the log scale, then exponentiate the prediction to get it out of log scale and evaluate your model. The code can also be found in this chapter's notebook in the book's GitHub repo. You should see that your RMSE decreases and your R^2 increases for this data set.

Saving and Loading Models

Now that we have built and evaluated a model, let's save it to persistent storage for reuse later (or in the event that our cluster goes down, we don't have to recompute the model). Saving models is very similar to writing DataFrames—the API is `model.write().save(path)`. You can optionally provide the `overwrite()` command to overwrite any data contained in that path:

```
# In Python
pipelinePath = "/tmp/lr-pipeline-model"
pipelineModel.write().overwrite().save(pipelinePath)
```

```
// In Scala
val pipelinePath = "/tmp/lr-pipeline-model"
pipelineModel.write.overwrite().save(pipelinePath)
```

When you load your saved models, you need to specify the type of model you are loading back in (e.g., was it a `LinearRegressionModel` or a `LogisticRegressionModel`?). For this reason, we recommend you always put your transformers/estimators into a `Pipeline`, so that for all your models you load a `PipelineModel` and only need to change the file path to the model:

```
# In Python
from pyspark.ml import PipelineModel
savedPipelineModel = PipelineModel.load(pipelinePath)
```

```
// In Scala
import org.apache.spark.ml.PipelineModel
val savedPipelineModel = PipelineModel.load(pipelinePath)
```

After loading, you can apply it to new data points. However, you can't use the weights from this model as initialization parameters for training a new model (as opposed to starting with random weights), as Spark has no concept of "warm starts." If your data set changes slightly, you'll have to retrain the entire linear regression model from scratch.

With our linear regression model built and evaluated, let's explore how a few other models perform on our data set. In the next section, we will explore tree-based models and look at some common hyperparameters to tune in order to improve model performance.

Hyperparameter Tuning

When data scientists talk about tuning their models, they often discuss tuning hyperparameters to improve the model's predictive power. A *hyperparameter* is an attribute that you define about the model prior to training, and it is not learned during the training process (not to be confused with parameters, which *are* learned in the training process). The number of trees in your random forest is an example of a hyperparameter.

In this section, we will focus on using tree-based models as an example for hyperparameter tuning procedures, but the same concepts apply to other models as well. Once we set up the mechanics to do hyperparameter tuning with `spark.ml`, we will discuss ways to optimize the pipeline. Let's get started with a brief introduction to decision trees, followed by how we can use them in `spark.ml`.

Tree-Based Models

Tree-based models such as decision trees, gradient boosted trees, and random forests are relatively simple yet powerful models that are easy to interpret (meaning, it is easy to explain the predictions they make). Hence, they're quite popular for machine learning tasks. We'll get to random forests shortly, but first we need to cover the fundamentals of decision trees.

Decision trees

As an off-the-shelf solution, decision trees are well suited to data mining. They are relatively fast to build, highly interpretable, and scale-invariant (i.e., standardizing or scaling the numeric features does not change the performance of the tree). So what is a decision tree?

A decision tree is a series of if-then-else rules learned from your data for classification or regression tasks. Suppose we are trying to build a model to predict whether or not someone will accept a job offer, and the features comprise salary, commute time, free coffee, etc. If we fit a decision tree to this data set, we might get a model that looks like Figure 10-9.

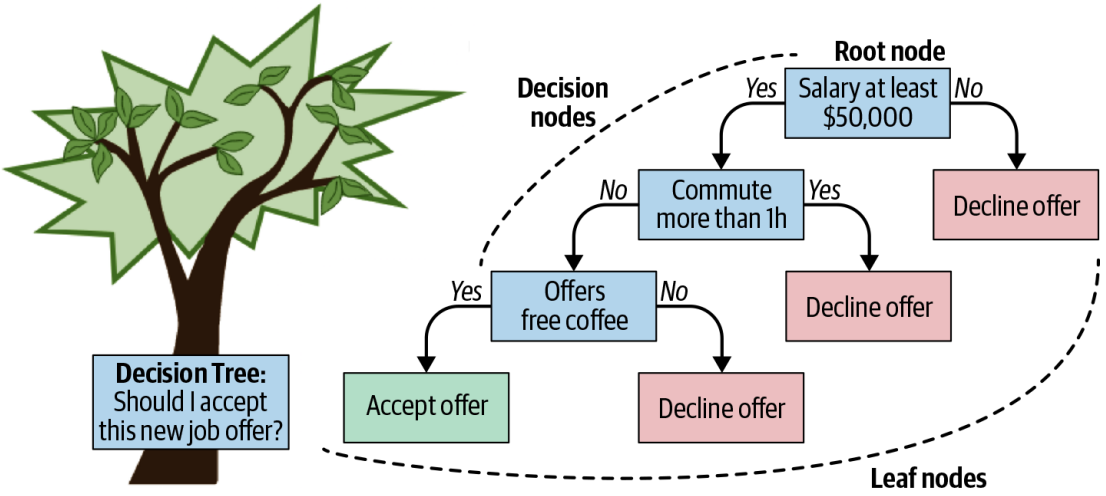

Figure 10-9. Decision tree example

The node at the top of the tree is called the "root" of the tree because it's the first feature that we "split" on. This feature should give the most informative split—in this case, if the salary is less than $50,000, then the majority of candidates will decline the job offer. The "Decline offer" node is known as a "leaf node" as there are no other splits coming out of that node; it's at the end of a branch. (Yes, it's a bit funny that we call it a decision "tree" but draw the root of the tree at the top and the leaves at the bottom!)

However, if the salary offered is greater than $50,000, we proceed to the next most informative feature in the decision tree, which in this case is the commute time. Even if the salary is over $50,000, if the commute is longer than one hour, then the majority of people will decline the job offer.

NOTE

We won't get into the details of how to determine which features will give you the highest information gain here, but if you're interested, check out Chapter 9 of *The Elements of Statistical Learning*, by Trevor Hastie, Robert Tibshirani, and Jerome Friedman (Springer).

The final feature in our model is free coffee. In this case the decision tree shows that if the salary is greater than $50,000, the commute is less than an hour, and there is free coffee, then the majority of people will accept our job offer (if only it were that simple!). As a follow-up resource, R2D3 has a great visualization of how decision trees work.

NOTE

It is possible to split on the same feature multiple times in a single decision tree, but each split will occur at a different value.

The *depth* of a decision tree is the longest path from the root node to any given leaf node. In Figure 10-9, the depth is three. Trees that are very deep are prone to overfitting, or memorizing noise in your training data set, but trees that are too shallow will underfit to your data set (i.e., could have picked up more signal from the data).

With the essence of a decision tree explained, let's resume the topic of feature preparation for decision trees. For decision trees, you don't have to worry about standardizing or scaling your input features, because this has no impact on the splits—but you do have to be careful about how you prepare your categorical features.

Tree-based methods can naturally handle categorical variables. In `spark.ml`, you just need to pass the categorical columns to the `StringIndexer`, and the decision tree can take care of the rest. Let's fit a decision tree to our data set:

```
# In Python
from pyspark.ml.regression import DecisionTreeRegressor

dt = DecisionTreeRegressor(labelCol="price")

# Filter for just numeric columns (and exclude price, our label)
numericCols = [field for (field, dataType) in trainDF.dtypes
               if ((dataType == "double") & (field != "price"))]

# Combine output of StringIndexer defined above and numeric columns
assemblerInputs = indexOutputCols + numericCols
```

```
vecAssembler = VectorAssembler(inputCols=assemblerInputs, outputCol="features")

# Combine stages into pipeline
stages = [stringIndexer, vecAssembler, dt]
pipeline = Pipeline(stages=stages)
pipelineModel = pipeline.fit(trainDF) # This line should error
```

```
// In Scala
import org.apache.spark.ml.regression.DecisionTreeRegressor

val dt = new DecisionTreeRegressor()
  .setLabelCol("price")

// Filter for just numeric columns (and exclude price, our label)
val numericCols = trainDF.dtypes.filter{ case (field, dataType) =>
  dataType == "DoubleType" && field != "price"}.map(_._1)

// Combine output of StringIndexer defined above and numeric columns
val assemblerInputs = indexOutputCols ++ numericCols
val vecAssembler = new VectorAssembler()
  .setInputCols(assemblerInputs)
  .setOutputCol("features")

// Combine stages into pipeline
val stages = Array(stringIndexer, vecAssembler, dt)
val pipeline = new Pipeline()
  .setStages(stages)

val pipelineModel = pipeline.fit(trainDF) // This line should error
```

This produces the following error:

```
java.lang.IllegalArgumentException: requirement failed: DecisionTree requires
maxBins (= 32) to be at least as large as the number of values in each
categorical feature, but categorical feature 3 has 36 values. Consider removing
this and other categorical features with a large number of values, or add more
training examples.
```

We can see that there is an issue with the `maxBins` parameter. What does that parameter do? `maxBins` determines the number of bins into which your continuous features are discretized, or split. This discretization step

is crucial for performing distributed training. There is no `maxBins` parameter in `scikit-learn` because all of the data and the model reside on a single machine. In Spark, however, workers have all the columns of the data, but only a subset of the rows. Thus, when communicating about which features and values to split on, we need to be sure they're all talking about the same split values, which we get from the common discretization set up at training time. Let's take a look at Figure 10-10, which shows the PLANET implementation of distributed decision trees, to get a better understanding of distributed machine learning and illustrate the `maxBins` parameter.

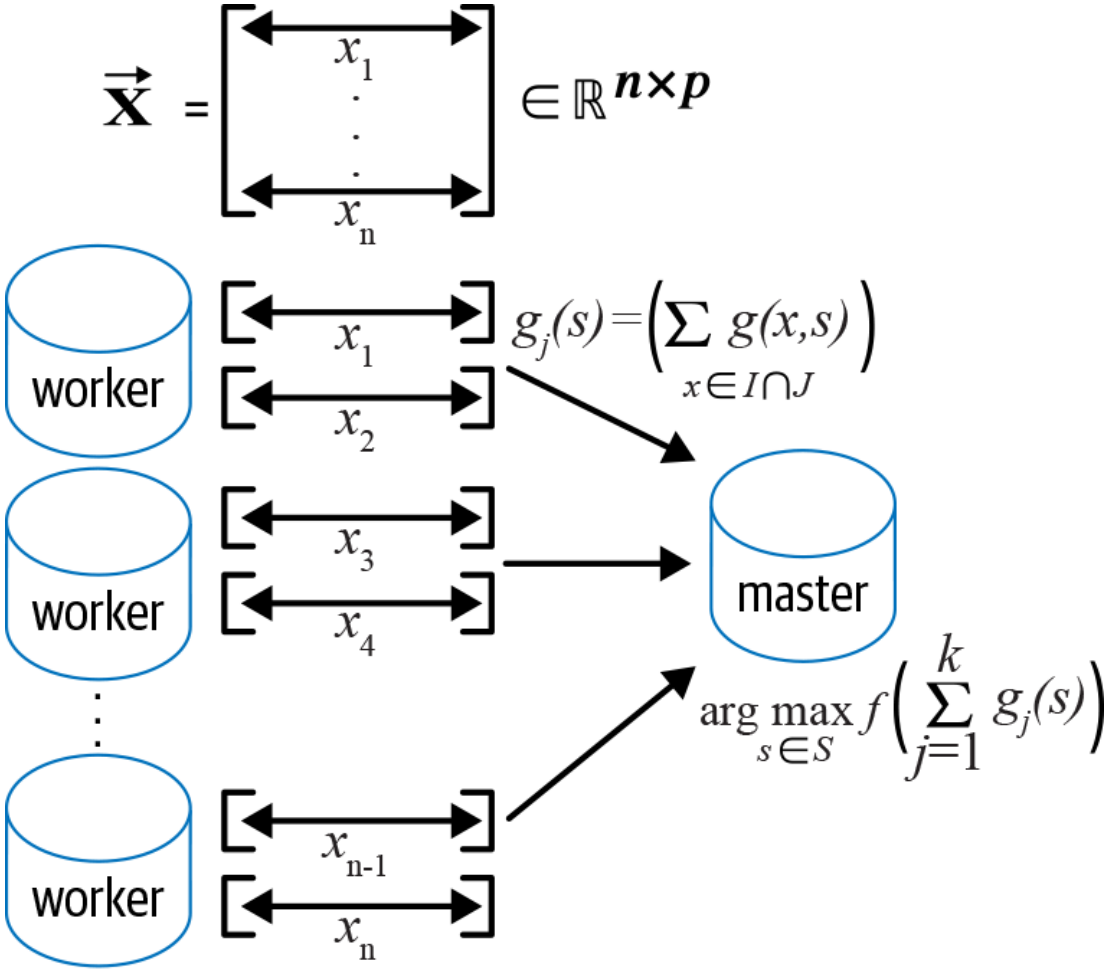

Figure 10-10. PLANET implementation of distributed decision trees (source: *https://oreil.ly/RAvvP*)

Every worker has to compute summary statistics for every feature and every possible split point, and those statistics will be aggregated across the workers. MLlib requires `maxBins` to be large enough to handle the discretization of the categorical columns. The default value for `maxBins` is `32`, and we had a categorical column with 36 distinct values, which is why we got the error earlier. While we could increase `maxBins` to `64` to more accurately represent our continuous features, that would double

the number of possible splits for continuous variables, greatly increasing our computation time. Let's instead set `maxBins` to be `40` and retrain the pipeline. You'll notice here that we are using the setter method `setMaxBins()` to modify the decision tree rather than redefining it completely:

```
# In Python
dt.setMaxBins(40)
pipelineModel = pipeline.fit(trainDF)
```

```
// In Scala
dt.setMaxBins(40)
val pipelineModel = pipeline.fit(trainDF)
```

NOTE

Due to differences in implementation, oftentimes you won't get exactly the same results when building a model with `scikit-learn` versus MLlib. However, that's OK. The key is to understand why they are different, and to see what parameters are in your control to get them to perform the way you need them to. If you are porting workloads over from `scikit-learn` to MLlib, we encourage you to take a look at the `spark.ml` and `scikit-learn` documentation to see what parameters differ, and to tweak those parameters to get comparable results for the same data. Once the values are close enough, you can scale up your MLlib model to larger data sizes that `scikit-learn` can't handle.

Now that we have successfully built our model, we can extract the if-then-else rules learned by the decision tree:

```
# In Python
dtModel = pipelineModel.stages[-1]
print(dtModel.toDebugString)
```

```
// In Scala
val dtModel = pipelineModel.stages.last
  .asInstanceOf[org.apache.spark.ml.regression.DecisionTreeRegressionModel]
println(dtModel.toDebugString)
```

```
DecisionTreeRegressionModel: uid=dtr_005040f1efac, depth=5, numNodes=47,...
  If (feature 12 <= 2.5)
   If (feature 12 <= 1.5)
    If (feature 5 in {1.0,2.0})
     If (feature 4 in {0.0,1.0,3.0,5.0,9.0,10.0,11.0,13.0,14.0,16.0,18.0,24.0})
      If (feature 3 in
{0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0,11.0,12.0,13.0,14.0,...})
       Predict: 104.23992784125075
      Else (feature 3 not in {0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0,...}
       Predict: 250.7111111111111
...
```

This is just a subset of the printout, but you'll notice that it's possible to split on the same feature more than once (e.g., feature 12), but at different split values. Also notice the difference between how the decision tree splits on numeric features versus categorical features: for numeric features it checks if the value is less than or equal to the threshold, and for categorical features it checks if the value is in that set or not.

We can also extract the feature importance scores from our model to see the most important features:

```
# In Python
import pandas as pd

featureImp = pd.DataFrame(
  list(zip(vecAssembler.getInputCols(), dtModel.featureImportances)),
  columns=["feature", "importance"])
featureImp.sort_values(by="importance", ascending=False)
```

```
// In Scala
val featureImp = vecAssembler
  .getInputCols.zip(dtModel.featureImportances.toArray)
val columns = Array("feature", "Importance")
val featureImpDF = spark.createDataFrame(featureImp).toDF(columns: _*)

featureImpDF.orderBy($"Importance".desc).show()
```

Feature	Importance
bedrooms	0.283406
cancellation_policyIndex	0.167893
instant_bookableIndex	0.140081
property_typeIndex	0.128179
number_of_reviews	0.126233
neighbourhood_cleansedIndex	0.056200
longitude	0.038810
minimum_nights	0.029473
beds	0.015218
room_typeIndex	0.010905
accommodates	0.003603

While decision trees are very flexible and easy to use, they are not always the most accurate model. If we were to compute our R^2 on the test data set, we would actually get a negative score! That's worse than just predicting the average. (You can see this in this chapter's notebook in the book's GitHub repo.)

Let's look at improving this model by using an *ensemble* approach that combines different models to achieve a better result: random forests.

Random forests

Ensembles work by taking a democratic approach. Imagine there are many M&Ms in a jar. You ask one hundred people to guess the number of

M&Ms, and then take the average of all the guesses. The average is probably closer to the true value than most of the individual guesses. That same concept applies to machine learning models. If you build many models and combine/average their predictions, they will be more robust than those produced by any individual model.

Random forests are an ensemble of decision trees with two key tweaks:

Bootstrapping samples by rows

> *Bootstrapping* is a technique for simulating new data by sampling with replacement from your original data. Each decision tree is trained on a different bootstrap sample of your data set, which produces slightly different decision trees, and then you aggregate their predictions. This technique is known as *bootstrap aggregating*, or *bagging*. In a typical random forest implementation, each tree samples the same number of data points with replacement from the original data set, and that number can be controlled through the `subsamplingRate` parameter.

Random feature selection by columns

> The main drawback with bagging is that the trees are all highly correlated, and thus learn similar patterns in your data. To mitigate this problem, each time you want to make a split you only consider a random subset of the columns (1/3 of the features for `RandomForestRegressor` and $\sqrt{\#\text{features}}$ for `RandomForestClassifier`). Due to this randomness you introduce, you typically want each tree to be quite shallow. You might be thinking: each of these trees will perform worse than any single decision tree, so how could this approach possibly be better? It turns out that each of the trees learns something different about your data set, and combining this collection of "weak" learners into an ensemble makes the forest much more robust than a single decision tree.

Figure 10-11 illustrates a random forest at training time. At each split, it considers 3 of the 10 original features to split on; finally, it picks the best from among those.

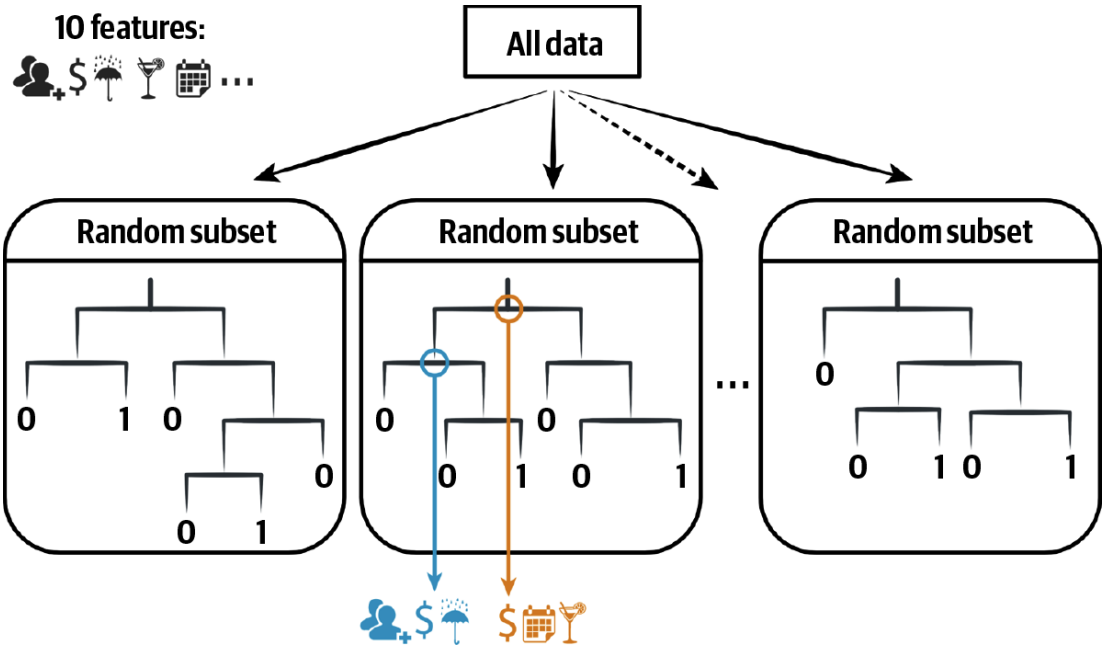

Figure 10-11. Random forest training

The APIs for random forests and decision trees are similar, and both can be applied to regression or classification tasks:

```
# In Python
from pyspark.ml.regression import RandomForestRegressor
rf = RandomForestRegressor(labelCol="price", maxBins=40, seed=42)
```

```
// In Scala
import org.apache.spark.ml.regression.RandomForestRegressor
val rf = new RandomForestRegressor()
  .setLabelCol("price")
  .setMaxBins(40)
  .setSeed(42)
```

Once you've trained your random forest, you can pass new data points through the different trees trained in the ensemble.

As Figure 10-12 shows, if you build a random forest for classification, it passes the test point through each of the trees in the forest and takes a majority vote among the predictions of the individual trees. (By contrast, in regression, the random forest simply averages those predictions.) Even though each of these trees is less performant than any individual decision tree, the collection (or ensemble) actually provides a more robust model.

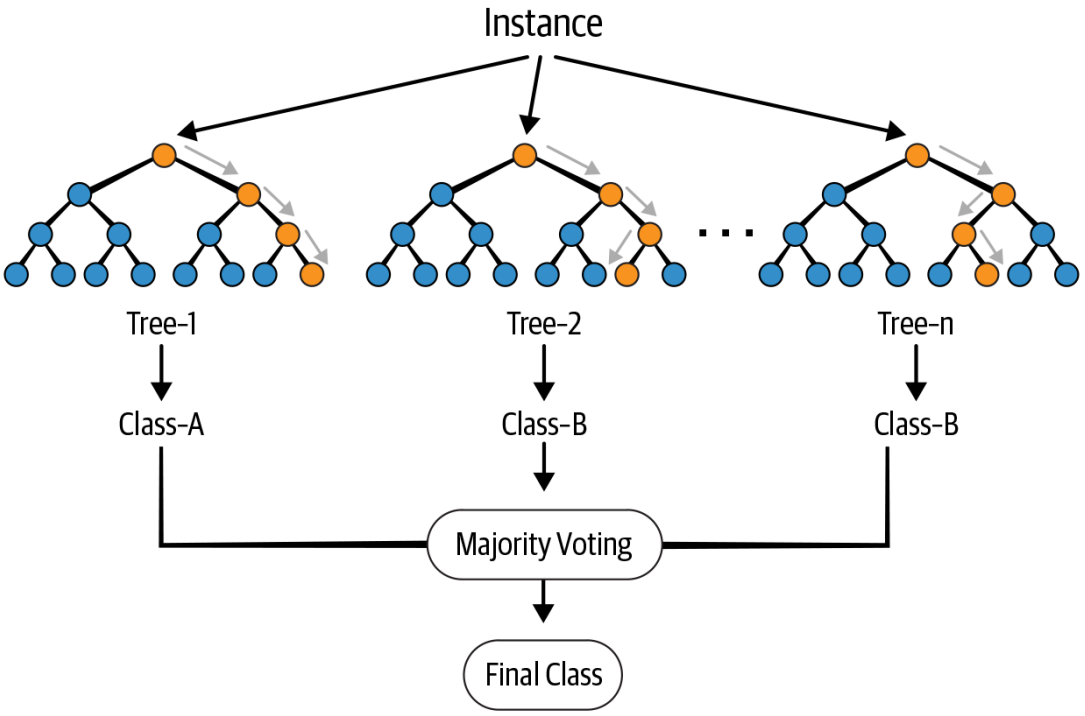

Figure 10-12. Random forest predictions

Random forests truly demonstrate the power of distributed machine learning with Spark, as each tree can be built independently of the other trees (e.g., you do not need to build tree 3 before you build tree 10). Furthermore, within each level of the tree, you can parallelize the work to find the optimal splits.

So how do we determine what the optimal number of trees in our random forest or the max depth of those trees should be? This process is called *hyperparameter tuning*. In contrast to a parameter, a hyperparameter is a value that controls the learning process or structure of your model, and it is not learned during training. Both the number of trees and the max depth are examples of hyperparameters you can tune for random forests. Let's now shift our focus to how we can discover and evaluate the best random forest model by tuning some hyperparameters.

k-Fold Cross-Validation

Which data set should we use to determine the optimal hyperparameter values? If we use the training set, then the model is likely to overfit, or memorize the nuances of our training data. This means it will be less likely to generalize to unseen data. But if we use the test set, then that will

no longer represent “unseen” data, so we won’t be able to use it to verify how well our model generalizes. Thus, we need another data set to help us determine the optimal hyperparameters: the *validation* data set.

For example, instead of splitting our data into an 80/20 train/test split, as we did earlier, we can do a 60/20/20 split to generate training, validation, and test data sets, respectively. We can then build our model on the training set, evaluate performance on the validation set to select the best hyperparameter configuration, and apply the model to the test set to see how well it performs on new data. However, one of the downsides of this approach is that we lose 25% of our training data (80% -> 60%), which could have been used to help improve the model. This motivates the use of the *k-fold cross-validation* technique to solve this problem.

With this approach, instead of splitting the data set into separate training, validation, and test sets, we split it into training and test sets as before—but we use the training data for both training and validation. To accomplish this, we split our training data into k subsets, or “folds” (e.g., three). Then, for a given hyperparameter configuration, we train our model on k–1 folds and evaluate on the remaining fold, repeating this process k times. Figure 10-13 illustrates this approach.

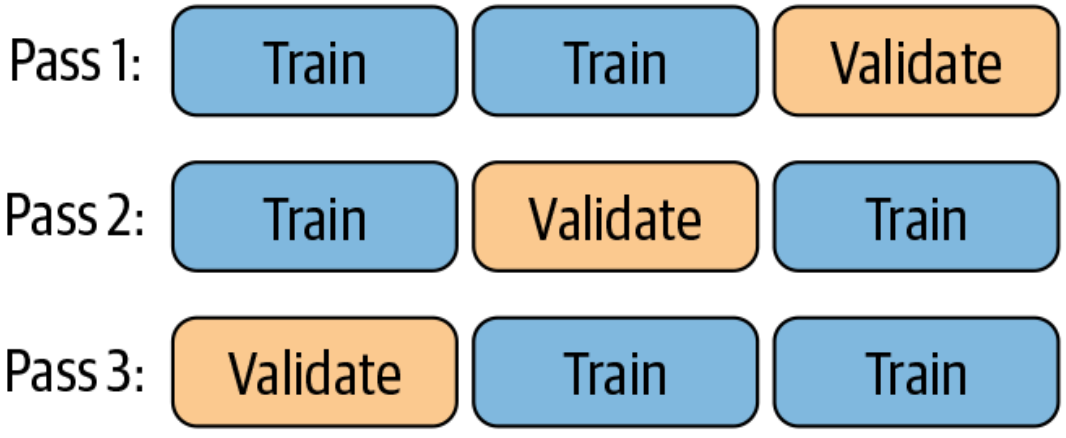

Figure 10-13. k-fold cross-validation

As this figure shows, if we split our data into three folds, our model is first trained on the first and second folds (or splits) of the data, and evaluated on the third fold. We then build the same model with the same hyperparameters on the first and third folds of the data, and evaluate its performance on the second fold. Lastly, we build the model on the second and

third folds and evaluate it on the first fold. We then average the performance of those three (or *k*) validation data sets as a proxy of how well this model will perform on unseen data, as every data point had the chance to be part of the validation data set exactly once. Next, we repeat this process for all of our different hyperparameter configurations to identify the optimal one.

Determining the search space of your hyperparameters can be difficult, and often doing a random search of hyperparameters outperforms a structured grid search. There are specialized libraries, such as Hyperopt, to help you identify the optimal hyperparameter configurations, which we touch upon in Chapter 11.

To perform a hyperparameter search in Spark, take the following steps :

1. Define the `estimator` you want to evaluate.
2. Specify which hyperparameters you want to vary, as well as their respective values, using the `ParamGridBuilder` .
3. Define an `evaluator` to specify which metric to use to compare the various models.
4. Use the `CrossValidator` to perform cross-validation, evaluating each of the various models.

Let's start by defining our pipeline estimator:

```
# In Python
pipeline = Pipeline(stages = [stringIndexer, vecAssembler, rf])
```

```
// In Scala
val pipeline = new Pipeline()
  .setStages(Array(stringIndexer, vecAssembler, rf))
```

For our `ParamGridBuilder` , we'll vary our `maxDepth` to be 2, 4, or 6 and `numTrees` (the number of trees in our random forest) to be 10 or 100. This will give us a grid of 6 (3 x 2) different hyperparameter configurations in total:

```
(maxDepth=2, numTrees=10)
(maxDepth=2, numTrees=100)
(maxDepth=4, numTrees=10)
(maxDepth=4, numTrees=100)
(maxDepth=6, numTrees=10)
(maxDepth=6, numTrees=100)
```

```
# In Python
from pyspark.ml.tuning import ParamGridBuilder
paramGrid = (ParamGridBuilder()
             .addGrid(rf.maxDepth, [2, 4, 6])
             .addGrid(rf.numTrees, [10, 100])
             .build())
```

```
// In Scala
import org.apache.spark.ml.tuning.ParamGridBuilder
val paramGrid = new ParamGridBuilder()
  .addGrid(rf.maxDepth, Array(2, 4, 6))
  .addGrid(rf.numTrees, Array(10, 100))
  .build()
```

Now that we have set up our hyperparameter grid, we need to define how to evaluate each of the models to determine which one performed best. For this task we will use the `RegressionEvaluator`, and we'll use RMSE as our metric of interest:

```
# In Python
evaluator = RegressionEvaluator(labelCol="price",
                                predictionCol="prediction",
                                metricName="rmse")
```

```
// In Scala
val evaluator = new RegressionEvaluator()
  .setLabelCol("price")
  .setPredictionCol("prediction")
  .setMetricName("rmse")
```

We will perform our *k*-fold cross-validation using the `CrossValidator`, which accepts an `estimator`, `evaluator`, and `estimatorParamMaps` so that it knows which model to use, how to evaluate the model, and which hyperparameters to set for the model. We can also set the number of folds we want to split our data into (`numFolds=3`), as well as setting a seed so we have reproducible splits across the folds (`seed=42`). Let's then fit this cross-validator to our training data set:

```
# In Python
from pyspark.ml.tuning import CrossValidator

cv = CrossValidator(estimator=pipeline,
                    evaluator=evaluator,
                    estimatorParamMaps=paramGrid,
                    numFolds=3,
                    seed=42)
cvModel = cv.fit(trainDF)


// In Scala
import org.apache.spark.ml.tuning.CrossValidator

val cv = new CrossValidator()
 .setEstimator(pipeline)
 .setEvaluator(evaluator)
 .setEstimatorParamMaps(paramGrid)
 .setNumFolds(3)
 .setSeed(42)
val cvModel = cv.fit(trainDF)
```

The output tells us how long the operation took:

```
Command took 1.07 minutes
```

So, how many models did we just train? If you answered 18 (6 hyperparameter configurations x 3-fold cross-validation), you're close. Once you've identified the optimal hyperparameter configuration, how do you combine those three (or *k*) models together? While some models might be easy enough to average together, some are not. Therefore, Spark retrains your model on the entire training data set once it has identified the opti-

mal hyperparameter configuration, so in the end we trained 19 models. If you want to retain the intermediate models trained, you can set `collectSubModels=True` in the `CrossValidator`.

To inspect the results of the cross-validator, you can take a look at the `avgMetrics`:

```
# In Python
list(zip(cvModel.getEstimatorParamMaps(), cvModel.avgMetrics))
```

```
// In Scala
cvModel.getEstimatorParamMaps.zip(cvModel.avgMetrics)
```

Here's the output:

```
res1: Array[(org.apache.spark.ml.param.ParamMap, Double)] =
Array(({
    rfr_a132fb1ab6c8-maxDepth: 2,
    rfr_a132fb1ab6c8-numTrees: 10
},303.99522869739343), ({
    rfr_a132fb1ab6c8-maxDepth: 2,
    rfr_a132fb1ab6c8-numTrees: 100
},299.56501993529474), ({
    rfr_a132fb1ab6c8-maxDepth: 4,
    rfr_a132fb1ab6c8-numTrees: 10
},310.63687030886894), ({
    rfr_a132fb1ab6c8-maxDepth: 4,
    rfr_a132fb1ab6c8-numTrees: 100
},294.7369599168999), ({
    rfr_a132fb1ab6c8-maxDepth: 6,
    rfr_a132fb1ab6c8-numTrees: 10
},312.6678169109293), ({
    rfr_a132fb1ab6c8-maxDepth: 6,
    rfr_a132fb1ab6c8-numTrees: 100
},292.101039874209))
```

We can see that the best model from our `CrossValidator` (the one with the lowest RMSE) had `maxDepth=6` and `numTrees=100`. However, this took a long time to run. In the next section, we will look at how we can

decrease the time to train our model while maintaining the same model performance.

Optimizing Pipelines

If your code takes long enough for you to think about improving it, then you should optimize it. In the preceding code, even though each of the models in the cross-validator is technically independent, `spark.ml` actually trains the collection of models sequentially rather than in parallel. In Spark 2.3, a `parallelism` parameter was introduced to solve this problem. This parameter determines the number of models to train in parallel, which themselves are fit in parallel. From the Spark Tuning Guide:

> *The value of `parallelism` should be chosen carefully to maximize parallelism without exceeding cluster resources, and larger values may not always lead to improved performance. Generally speaking, a value up to `10` should be sufficient for most clusters.*

Let's set this value to `4` and see if we can train any faster:

```
# In Python
cvModel = cv.setParallelism(4).fit(trainDF)
```

```
// In Scala
val cvModel = cv.setParallelism(4).fit(trainDF)
```

The answer is yes:

```
Command took 31.45 seconds
```

We've cut the training time in half (from 1.07 minutes to 31.45 seconds), but we can still improve it further! There's another trick we can use to speed up model training: putting the cross-validator inside the pipeline (e.g., `Pipeline(stages=[..., cv])` instead of putting the pipeline inside the cross-validator (e.g., `CrossValidator(estimator=pipeline, ...)`). Every time the cross-validator evaluates the pipeline, it runs through every step of the pipeline for each model, even if some of the steps don't

change, such as the `StringIndexer`. By reevaluating every step in the pipeline, we are learning the same `StringIndexer` mapping over and over again, even though it's not changing.

If instead we put our cross-validator inside our pipeline, then we won't be reevaluating the `StringIndexer` (or any other estimator) each time we try a different model:

```
# In Python
cv = CrossValidator(estimator=rf,
                    evaluator=evaluator,
                    estimatorParamMaps=paramGrid,
                    numFolds=3,
                    parallelism=4,
                    seed=42)

pipeline = Pipeline(stages=[stringIndexer, vecAssembler, cv])
pipelineModel = pipeline.fit(trainDF)
```

```
// In Scala
val cv = new CrossValidator()
  .setEstimator(rf)
  .setEvaluator(evaluator)
  .setEstimatorParamMaps(paramGrid)
  .setNumFolds(3)
  .setParallelism(4)
  .setSeed(42)

val pipeline = new Pipeline()
                   .setStages(Array(stringIndexer, vecAssembler, cv))
val pipelineModel = pipeline.fit(trainDF)
```

This trims five seconds off our training time:

```
Command took 26.21 seconds
```

Thanks to the `parallelism` parameter and rearranging the ordering of our pipeline, that last run was the fastest—and if you apply it to the test data set you'll see that you get the same results. Although these gains

were on the order of seconds, the same techniques apply to much larger data sets and models, with correspondingly larger time savings. You can try running this code yourself by accessing the notebook in the book's GitHub repo.

Summary

In this chapter we covered how to build pipelines using Spark MLlib—in particular, its DataFrame-based API package, `spark.ml`. We discussed the differences between transformers and estimators, how to compose them using the Pipeline API, and some different metrics for evaluating models. We then explored how to use cross-validation to perform hyperparameter tuning to deliver the best model, as well as tips for optimizing cross-validation and model training in Spark.

All this sets the context for the next chapter, in which we will discuss deployment strategies and ways to manage and scale machine learning pipelines with Spark.