**Chapter 24. Advanced Analytics and Machine Learning Overview**

Thus far, we have covered fairly general data flow APIs. This part of the book will dive deeper into some of the more specific advanced analytics APIs available in Spark. Beyond large-scale SQL analysis and streaming, Spark also provides support for statistics, machine learning, and graph analytics. These encompass a set of workloads that we will refer to as advanced analytics. This part of the book will cover advanced analytics tools in Spark, including:

* Preprocessing your data (cleaning data and feature engineering)
* Supervised learning
* Recommendation learning
* Unsupervised engines
* Graph analytics
* Deep learning

This chapter offers a basic overview of advanced analytics, some example use cases, and a basic advanced analytics workflow. Then we’ll cover the analytics tools just listed and teach you how to apply them.

**WARNING**

This book is not intended to teach you everything you need to know about machine learning from scratch. We won’t go into strict mathematical definitions and formulations—not for lack of importance but simply because it’s too much information to include. This part of the book is *not* an algorithm guide that will teach you the mathematical underpinnings of every available algorithm nor the in-depth implementation strategies used. The chapters included here serve as a guide for *users*, with the purpose of outlining what you need to know to use Spark’s advanced analytics APIs.

**A Short Primer on Advanced Analytics**

Advanced analytics refers to a variety of techniques aimed at solving the core problem of deriving insights and making predictions or recommendations based on data. The best ontology for machine learning is structured based on the task that you’d like to perform. The most common tasks include:

* Supervised learning, including classification and regression, where the goal is to predict a label for each data point based on various features.
* Recommendation engines to suggest products to users based on behavior.
* Unsupervised learning, including clustering, anomaly detection, and topic modeling, where the goal is to discover structure in the data.
* Graph analytics tasks such as searching for patterns in a social network.

Before discussing Spark’s APIs in detail, let’s review each of these tasks along with some common machine learning and advanced analytics use cases. While we have certainly tried to make this introduction as accessible as possible, at times you may need to consult other resources in order to fully understand the material. O’Reilly should we link to or mention any specific ones? Additionally, we will cite the following books throughout the next few chapters because they are great resources for learning more about the individual analytics (and, as a bonus, they are freely available on the web):

* *[An Introduction to Statistical Learning](http://www-bcf.usc.edu/~gareth/ISL/)* by Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. We refer to this book as “ISL.”
* *[Elements of Statistical Learning](https://web.stanford.edu/~hastie/ElemStatLearn/)* by Trevor Hastie, Robert Tibshirani, and Jerome Friedman. We refer to this book as “ESL.”
* *[Deep Learning](http://www.deeplearningbook.org/)* by Ian Goodfellow, Yoshua Bengio, and Aaron Courville. We refer to this book as “DLB.”

**Supervised Learning**

*Supervised learning* is probably the most common type of machine learning. The goal is simple: using historical data that already has labels (often called the dependent variables), train a model to predict the values of those labels based on various features of the data points. One example would be to predict a person’s income (the dependent variable) based on age (a feature). This training process usually proceeds through an iterative optimization algorithm such as gradient descent. The training algorithm starts with a basic model and gradually improves it by adjusting various internal parameters (coefficients) during each training iteration. The result of this process is a trained model that you can use to make predictions on new data. There are a number of different tasks we’ll need to complete as part of the process of training and making predictions, such as measuring the success of trained models before using them in the field, but the fundamental principle is simple: train on historical data, ensure that it generalizes to data we didn’t train on, and then make predictions on new data.

We can further organize supervised learning based on the type of variable we’re looking to predict. We’ll get to that next.

**CLASSIFICATION**

One common type of supervised learning is classification. Classification is the act of training an algorithm to predict a dependent variable that is *categorical* (belonging to a discrete, finite set of values). The most common case is *binary classification*, where our resulting model will make a prediction that a given item belongs to one of two groups. The canonical example is classifying email spam. Using a set of historical emails that are organized into groups of spam emails and not spam emails, we train an algorithm to analyze the words in, and any number of properties of, the historical emails and make predictions about them. Once we are satisfied with the algorithm’s performance, we use that model to make predictions about future emails the model has never seen before.

When we classify items into more than just two categories, we call this *multiclass classification*. For example, we may have four different categories of email (as opposed to the two categories in the previous paragraph): spam, personal, work related, and other. There are many use cases for classification, including:

Predicting disease

A doctor or hospital might have a historical dataset of behavioral and physiological attributes of a set of patients. They could use this dataset to train a model on this historical data (and evaluate its success and ethical implications before applying it) and then leverage it to predict whether or not a patient has heart disease or not. This is an example of binary classification (healthy heart, unhealthy heart) or multiclass classification (healthly heart, or one of several different diseases).

Classifying images

There are a number of applications from companies like Apple, Google, or Facebook that can predict who is in a given photo by running a classification model that has been trained on historical images of people in your past photos. Another common use case is to classify images or label the objects in images.

Predicting customer churn

A more business-oriented use case might be predicting customer churn—that is, which customers are likely to stop using a service. You can do this by training a binary classifier on past customers that have churned (and not churned) and using it to try and predict whether or not current customers will churn.

Buy or won’t buy

Companies often want to predict whether visitors of their website will purchase a given product. They might use information about users’ browsing pattern or attributes such as location in order to drive this prediction.

There are many more use cases for classification beyond these examples. We will introduce more use cases, as well as Spark’s classification APIs, in [Chapter 26](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch26.html#s6c3---classification).

**REGRESSION**

In classification, our dependent variable is a set of discrete values. In regression, we instead try to predict a continuous variable (a real number). In simplest terms, rather than predicting a category, we want to predict a value on a number line. The rest of the process is largely the same, which is why they’re both forms of supervised learning. We will train on historical data to make predictions about data we have never seen. Here are some typical examples:

Predicting sales

A store may want to predict total product sales on given data using historical sales data. There are a number of potential input variables, but a simple example might be using last week’s sales data to predict the next day’s data.

Predicting height

Based on the heights of two individuals, we might want to predict the heights of their potential children.

Predicting the number of viewers of a show

A media company like Netflix might try to predict how many of their subscribers will watch a particular show.

We will introduce more use cases, as well as Spark’s methods for regression, in [Chapter 27](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch27.html#s6c4---regression).

**Recommendation**

Recommendation is one of the most intuitive applications of advanced analytics. By studying people’s explicit preferences (through ratings) or implicit ones (through observed behavior) for various products or items, an algorithm can make recommendations on what a user may like by drawing similarities between the users or items. By looking at these similarities, the algorithm makes recommendations to users based on what similar users liked, or what other products resemble the ones the user already purchased. Recommendation is a common use case for Spark and well suited to big data. Here are some example use cases:

Movie recommendations

[Netflix uses Spark](http://bit.ly/2Fkx4Mm), although not necessarily its built-in libraries, to make large-scale movie recommendations to its users. It does this by studying what movies users watch and do not watch in the Netflix application. In addition, Netflix likely takes into consideration how similar a given user’s ratings are to other users’.

Product recommendations

Amazon uses product recommendations as one of its main tools to increase sales. For instance, based on the items in our shopping cart, Amazon may recommend other items that were added to similar shopping carts in the past. Likewise, on every product page, Amazon shows similar products purchased by other users.

We will introduce more recommendation use cases, as well as Spark’s methods for generating recommendations, in [Chapter 28](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch28.html#s6c5---recommendation).

**Unsupervised Learning**

*Unsupervised learning* is the act of trying to find patterns or discover the underlying structure in a given set of data. This differs from supervised learning because there is no dependent variable (label) to predict.

Some example use cases for unsupervised learning include:

Anomaly detection

Given some standard event type often occuring over time, we might want to report when a nonstandard type of event occurs. For example, a security officer might want to receive notifications when a strange object (think vehicle, skater, or bicyclist) is observed on a pathway.

User segmentation

Given a set of user behaviors, we might want to better understand what attributes certain users share with other users. For instance, a gaming company might cluster users based on properties like the number of hours played in a given game. The algorithm might reveal that casual players have very different behavior than hardcore gamers, for example, and allow the company to offer different recommendations or rewards to each player.

Topic modeling

Given a set of documents, we might analyze the different words contained therein to see if there is some underlying relation between them. For example, given a number of web pages on data analytics, a topic modeling algorithm can cluster them into pages about machine learning, SQL, streaming, and so on based on groups of words that are more common in one topic than in others.

Intuitively, it is easy to see how segmenting customers could help a platform cater better to each set of users. However, it may be hard to discover whether or not this set of user segments is “correct”. For this reason, it can be difficult to determine whether a particular model is good or not. We will discuss unsupervised learning in detail in [Chapter 29](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch29.html#s6c6---unsupervised-learning).

**Graph Analytics**

While less common than classification and regression, graph analytics is a powerful tool. Fundamentally, graph analytics is the study of structures in which we specify *vertices* (which are objects) and *edges* (which represent the relationships between those objects). For example, the vertices might represent people and products, and edges might represent a purchase. By looking at the properties of vertices and edges, we can better understand the connections between them and the overall structure of the graph. Since graphs are all about relationships, anything that specifies a relationship is a great use case for graph analytics. Some examples include:

Fraud prediction

[Capital One uses Spark’s graph analytics capabilities](https://youtu.be/q5HFMVoN_rc) to better understand fraud networks. By using historical fraudulent information (like phone numbers, addresses, or names) they discover fraudulent credit requests or transactions. For instance, any user accounts within two hops of a fraudulent phone number might be considered suspicious.

Anomaly detection

By looking at how networks of individuals connect with one another, outliers and anomalies can be flagged for manual analysis. For instance, if typically in our data each vertex has ten edges associated with it and a given vertex only has one edge, that might be worth investigating as something strange.

Classification

Given some facts about certain vertices in a network, you can classify other vertices according to their connection to the original node. For instance, if a certain individual is labeled as an influencer in a social network, we could classify other individuals with similar network structures as influencers.

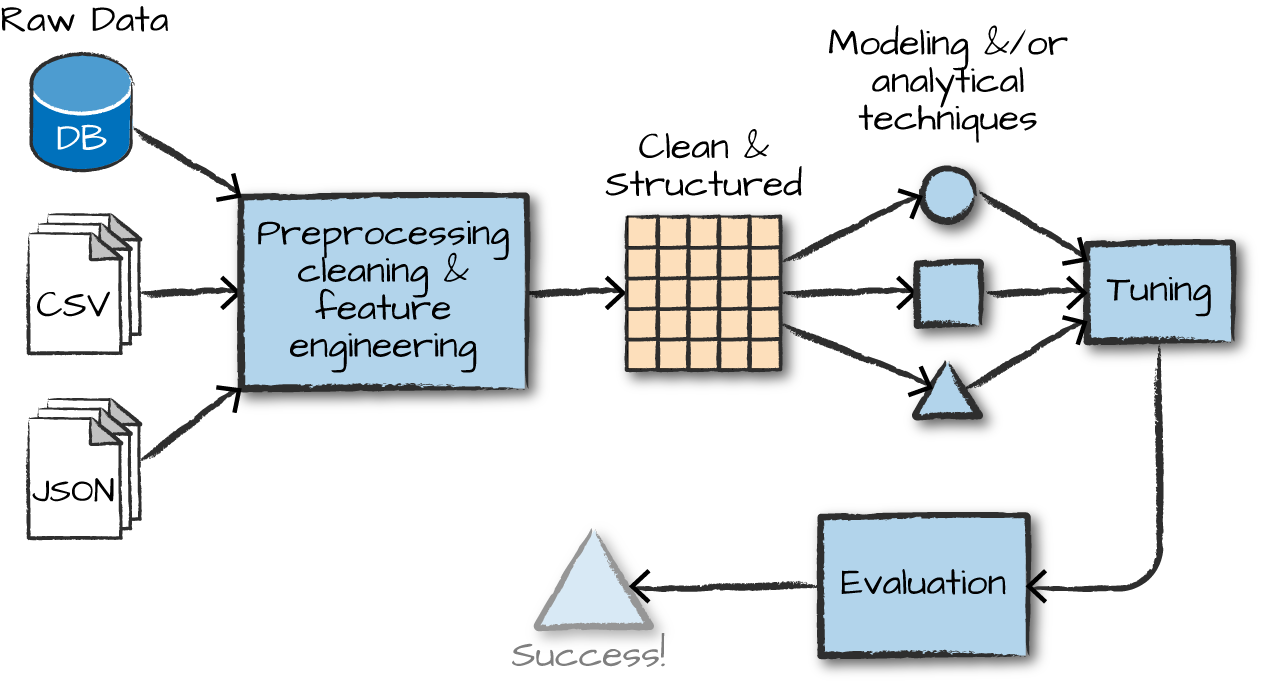
Recommendation

Google’s original web recommendation algorithm, PageRank, is a graph algorithm that analyzes website relationships in order to rank the importance of web pages. For example, a web page that has a lot of links to it is ranked as more important than one with no links to it.

We’ll discuss more examples of graph analytics in [Chapter 30](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch30.html#s6c7---graph-analysis).

**The Advanced Analytics Process**

You should have a firm grasp of some fundamental use cases for machine learning and advanced analytics. However, finding a use case is only a small part of the actual advanced analytics process. There is a lot of work in preparing your data for analysis, testing different ways of modeling it, and evaluating these models. This section will provide structure to the overall anaytics process and the steps we have to take to not just perform one of the tasks just outlined, but actually evaluate success objectively in order to understand whether or not we should apply our model to the real world (Figure 24-1).



*Figure 24-1. The machine learning workflow*

The overall process involves, the following steps (with some variation):

1. Gathering and collecting the relevant data for your task.
2. Cleaning and inspecting the data to better understand it.
3. Performing feature engineering to allow the algorithm to leverage the data in a suitable form (e.g., converting the data to numerical vectors).
4. Using a portion of this data as a training set to train one or more algorithms to generate some candidate models.
5. Evaluating and comparing models against your success criteria by objectively measuring results on a subset of the same data that was not used for training. This allows you to better understand how your model may perform in the wild.
6. Leveraging the insights from the above process and/or using the model to make predictions, detect anomalies, or solve more general business challenges.

These steps won’t be the same for every advanced analytics task. However, this workflow does serve as a general framework for what you’re going to need to be successful with advanced analytics. Just as we did with the various advanced analytics tasks earlier in the chapter, let’s break down the process to better understand the overall objective of each step.

**DATA COLLECTION**

Naturally it’s hard to create a training set without first collecting data. Typically this means at least gathering the datasets you’ll want to leverage to train your algorithm. Spark is an excellent tool for this because of its ability to speak to a variety of data sources and work with data big and small.

**DATA CLEANING**

After you’ve gathered the proper data, you’re going to need to clean and inspect it. This is typically done as part of a process called [exploratory data analysis](https://en.wikipedia.org/wiki/Exploratory_data_analysis), or EDA. EDA generally means using interactive queries and visualization methods in order to better understand distributions, correlations, and other details in your data. During this process you may notice you need to remove some values that may have been misrecorded upstream or that other values may be missing. Whatever the case, it’s always good to know what is in your data to avoid mistakes down the road. The multitude of Spark functions in the structured APIs will provide a simple way to clean and report on your data.

**FEATURE ENGINEERING**

Now that you collected and cleaned your dataset, it’s time to convert it to a form suitable for machine learning algorithms, which generally means numerical features. Proper feature engineering can often make or break a machine learning application, so this is one task you’ll want to do carefully. The process of feature engineering includes a variety of tasks, such as normalizing data, adding variables to represent the interactions of other variables, manipulating categorical variables, and converting them to the proper format to be input into our machine learning model. In MLlib, Spark’s machine learning library, all variables will usually have to be input as vectors of doubles (regardless of what they actually represent). We cover the process of feature engineering in great depth in [Chapter 25](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch25.html#s6c2---preprocessing-and-feature-engineering). As you will see in that chapter, Spark provides the essentials you’ll need to manipulate your data using a variety of machine learning statistical techniques.

**NOTE**

The following few steps (training models, model tuning, and evaluation) are not relevant to all use cases. This is a general workflow that may vary significantly based on the end objective you would like to achieve.

**TRAINING MODELS**

At this point in the process we have a dataset of historical information (e.g., spam or not spam emails) and a task we would like to complete (e.g., classifying spam emails). Next, we will want to train a model to predict the correct output, given some input. During the training process, the parameters inside of the model will change according to how well the model performed on the input data. For instance, to classify spam emails, our algorithm will likely find that certain words are better predictors of spam than others and therefore weight the parameters associated with those words higher. In the end, the trained model will find that certain words should have more influence (because of their consistent association with spam emails) than others. The output of the training process is what we call a model. Models can then be used to gain insights or to make future predictions. To make predictions, you will give the model an input and it will produce an output based on a mathematical manipulation of these inputs. Using the classification example, given the properties of an email, it will predict whether that email is spam or not by comparing to the historical spam and not spam emails that it was trained on.

However, just training a model isn’t the objective—we want to leverage our model to produce insights. Thus, we must answer the question: how do we know our model is any good at what it’s supposed to do? That’s where model tuning and evaluation come in.

**MODEL TUNING AND EVALUATION**

You likely noticed earlier that we mentioned that you should split your data into multiple portions and use only one for training. This is an essential step in the machine learning process because when you build an advanced analytics model you want that model to generalize to data it has not seen before. Splitting our dataset into multiple portions allows us to objectively test the effectiveness of the trained model against a set of data that it has never seen before. The objective is to see if your model understands something fundamental about this data process or whether or not it just noticed the things particular to only the training set (sometimes called *overfitting*). That’s why it is called a *test set*. In the process of training models, we also might take another, separate subset of data and treat that as another type of test set, called a *validation set*, in order to try out different *hyperparameters* (parameters that affect the training process) and compare different variations of the same model without overfitting to the test set.

**WARNING**

Following proper training, validation, and test set best practices is essential to successfully using machine learning. It’s easy to end up overfitting (training a model that does not generalize well to new data) if we do not properly isolate these sets of data. We cannot cover this problem in depth in this book, but almost any machine learning book will cover this topic.

To continue with the classification example we referenced previously, we have three sets of data: a training set for training models, a validation set for testing different variations of the models that we’re training, and lastly, a test set we will use for the final evaluation of our different model variations to see which one performed the best.

**LEVERAGING THE MODEL AND/OR INSIGHTS**

After running the model through the training process and ending up with a well-performing model, you are now ready to use it! Taking your model to production can be a significant challenge in and of itself. We will discuss some tactics later on in this chapter.

**Spark’s Advanced Analytics Toolkit**

The previous overview is just an example workflow and doesn’t encompass all use cases or potential workflows. In addition, you probably noticed that we did not discuss Spark almost at all. This section will discuss Spark’s advanced analytics capabilities. Spark includes several core packages and many external packages for performing advanced analytics. The primary package is MLlib, which provides an interface for building machine learning pipelines.

**What Is MLlib?**

MLlib is a package, built on and included in Spark, that provides interfaces for gathering and cleaning data, feature engineering and feature selection, training and tuning large-scale supervised and unsupervised machine learning models, and using those models in production.

**WARNING**

MLlib actually consists of two packages that leverage different core data structures. The package org.apache.spark.ml includes an interface for use with DataFrames. This package also offers a high-level interface for building machine learning pipelines that help standardize the way in which you perform the preceding steps. The lower-level package, org.apache.spark.mllib, includes interfaces for Spark’s low-level RDD APIs. This book will focus exclusively on the DataFrame API. The RDD API is the lower-level interface, which is in maintenance mode (meaning it will only receive bug fixes, not new features) at this time. It has also been covered fairly extensively in older books on Spark and is therefore omitted here.

**WHEN AND WHY SHOULD YOU USE MLLIB (VERSUS SCIKIT-LEARN, TENSORFLOW, OR FOO PACKAGE)**

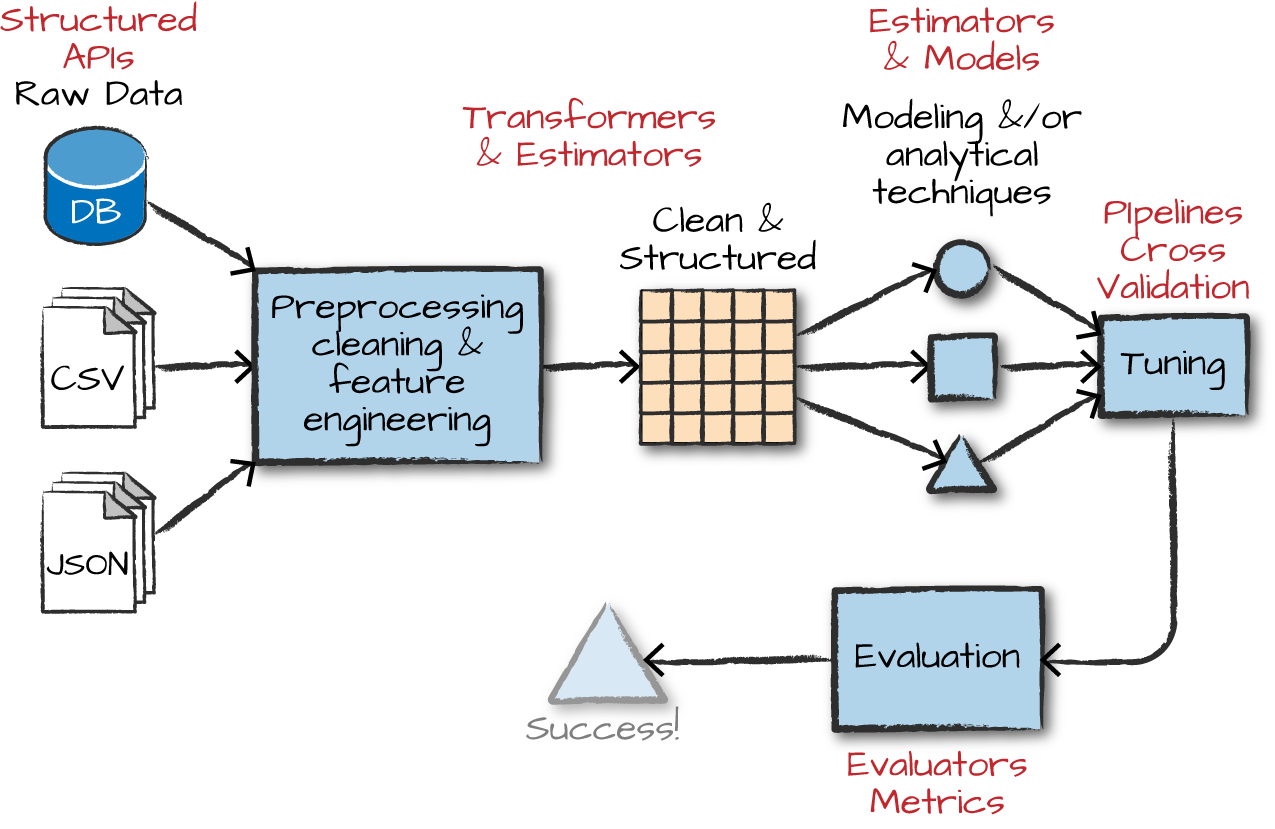
At a high level, MLlib might sound like a lot of other machine learning packages you’ve probably heard of, such as scikit-learn for Python or the variety of R packages for performing similar tasks. So why should you bother with MLlib at all? There are numerous tools for performing machine learning on a single machine, and while there are several great options to choose from, these single machine tools do have their limits either in terms of the size of data you can train on or the processing time. This means single-machine tools are usually *complementary* to MLlib. When you hit those scalability issues, take advantage of Spark’s abilities.

There are two key use cases where you want to leverage Spark’s ability to scale. First, you want to leverage Spark for preprocessing and feature generation to reduce the amount of time it might take to produce training and test sets from a large amount of data. Then you might leverage single-machine learning libraries to train on those given data sets. Second, when your input data or model size become too difficult or inconvenient to put on one machine, use Spark to do the heavy lifting. Spark makes distributed machine learning very simple.

An important caveat to all of this is that while training and data preparation are made simple, there are still some complexities you will need to keep in mind, especially when it comes to deploying a trained model. For example, Spark does not provide a built-in way to serve low-latency predictions from a model, so you may want to export the model to another serving system or a custom application to do that. MLlib is generally designed to allow inspecting and exporting models to other tools where possible.

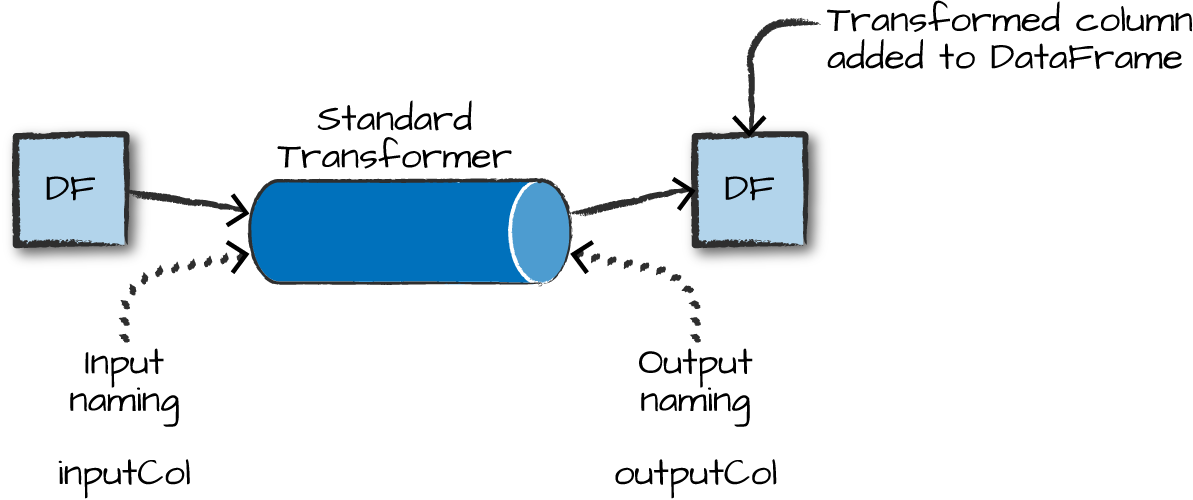
**High-Level MLlib Concepts**

In MLlib there are several fundamental “structural” types: transformers, estimators, evaluators, and pipelines. By structural, we mean you will think in terms of these types when you define an end-to-end machine learning pipeline. They’ll provide the common language for defining what belongs in what part of the pipeline. Figure 24-2 illustrates the overall workflow that you will follow when developing machine learning models in Spark.



*Figure 24-2. The machine learning workflow, in Spark*

*Transformers* are functions that convert raw data in some way. This might be to create a new interaction variable (from two other variables), normalize a column, or simply change an Integer into a Double type to be input into a model. An example of a transformer is one that converts string categorical variables into numerical values that can be used in MLlib. Transformers are primarily used in preprocessing and feature engineering. Transformers take a DataFrame as input and produce a new DataFrame as output, as illustrated in Figure 24-3.



*Figure 24-3. A standard transformer*

*Estimators* are one of two kinds of things. First, estimators can be a kind of transformer that is initialized with data. For instance, to normalize numerical data we’ll need to initialize our transformation with some information about the current values in the column we would like to normalize. This requires two passes over our data—the initial pass generates the initialization values and the second actually applies the generated function over the data. In the Spark’s nomenclature, algorithms that allow users to train a model from data are also referred to as estimators.

An *evaluator* allows us to see how a given model performs according to criteria we specify like a receiver operating characteristic (ROC) curve. After we use an evaluator to select the best model from the ones we tested, we can then use that model to make predictions.

From a high level we can specify each of the transformations, estimations, and evaluations one by one, but it is often easier to specify our steps as *stages* in a *pipeline*. This pipeline is similar to scikit-learn’s pipeline concept.

**LOW-LEVEL DATA TYPES**

In addition to the structural types for building pipelines, there are also several lower-level data types you may need to work with in MLlib (Vector being the most common). Whenever we pass a set of features into a machine learning model, we must do it as a vector that consists of Doubles. This vector can be either sparse (where most of the elements are zero) or dense (where there are many unique values). Vectors are created in different ways. To create a dense vector, we can specify an array of all the values. To create a sparse vector, we can specify the total size and the indices and values of the non-zero elements. Sparse is the best format, as you might have guessed, when the majority of values are zero as this is a more compressed representation. Here is an example of how to manually create a Vector:

*// in Scala*

**import** **org.apache.spark.ml.linalg.Vectors**

**val** denseVec **=** **Vectors**.dense(1.0, 2.0, 3.0)

**val** size **=** 3

**val** idx **=** **Array**(1,2) *// locations of non-zero elements in vector*

**val** values **=** **Array**(2.0,3.0)

**val** sparseVec **=** **Vectors**.sparse(size, idx, values)

sparseVec.toDense

denseVec.toSparse

*# in Python*

**from** **pyspark.ml.linalg** **import** Vectors

denseVec = Vectors.dense(1.0, 2.0, 3.0)

size = 3

idx = [1, 2] *# locations of non-zero elements in vector*

values = [2.0, 3.0]

sparseVec = Vectors.sparse(size, idx, values)

**WARNING**

Confusingly, there are similar datatypes that refer to ones that can be used in DataFrames and others that can only be used in RDDs. The RDD implementations fall under the mllib package while the DataFrame implementations fall under ml.

**MLlib in Action**

Now that we have described some of the core pieces you can expect to come across, let’s create a simple pipeline to demonstrate each of the components. We’ll use a small synthetic dataset that will help illustrate our point. Let’s read the data in and see a sample before talking about it further:

*// in Scala*

**var** df **=** spark.read.json("/data/simple-ml")

df.orderBy("value2").show()

*# in Python*

df = spark.read.json("/data/simple-ml")

df.orderBy("value2").show()

Here’s a sample of the data:

+-----+----+------+------------------+

|color| lab|value1| value2|

+-----+----+------+------------------+

|green|good| 1|14.386294994851129|

...

| red| bad| 16|14.386294994851129|

|green|good| 12|14.386294994851129|

+-----+----+------+------------------+

This dataset consists of a categorical label with two values (good or bad), a categorical variable (color), and two numerical variables. While the data is synthetic, let’s imagine that this dataset represents a company’s customer health. The “color” column represents some categorical health rating made by a customer service representative. The “lab” column represents the true customer health. The other two values are some numerical measures of activity within an application (e.g., minutes spent on site and purchases). Suppose that we want to train a classification model where we hope to predict a binary variable—the label—from the other values.

**TIP**

Apart from JSON, there are some specific data formats commonly used for supervised learning, including LIBSVM. These formats have real valued labels and sparse input data. Spark can read and write for these formats using its data source API. Here’s an example of how to read in data from a libsvm file using that Data Source API.

spark.read.format("libsvm").load(

"/data/sample\_libsvm\_data.txt")

For more information on LIBSVM, see [the documentation](http://www.csie.ntu.edu.tw/~cjlin/libsvm/).

**Feature Engineering with Transformers**

As already mentioned, transformers help us manipulate our current columns in one way or another. Manipulating these columns is often in pursuit of building features (that we will input into our model). Transformers exist to either cut down the number of features, add more features, manipulate current ones, or simply to help us format our data correctly. Transformers add new columns to DataFrames.

When we use MLlib, all inputs to machine learning algorithms (with several exceptions discussed in later chapters) in Spark must consist of type Double (for labels) and Vector[Double] (for features). The current dataset does *not* meet that requirement and therefore we need to transform it to the proper format.

To achieve this in our example, we are going to specify an RFormula. This is a declarative language for specifying machine learning transformations and is simple to use once you understand the syntax. RFormula supports a limited subset of the R operators that in practice work quite well for simple models and manipulations (we demonstrate the manual approach to this problem in [Chapter 25](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch25.html#s6c2---preprocessing-and-feature-engineering)). The basic RFormula operators are:

~

Separate target and terms

+

Concat terms; “+ 0” means removing the intercept (this means that the *y*-intercept of the line that we will fit will be 0)

-

Remove a term; “- 1” means removing the intercept (this means that the *y*-intercept of the line that we will fit will be 0—yes, this does the same thing as “+ 0”

:

Interaction (multiplication for numeric values, or binarized categorical values)

.

All columns except the target/dependent variable

In order to specify transformations with this syntax, we need to import the relevant class. Then we go through the process of defining our formula. In this case we want to use all available variables (the .) and also add in the interactions between value1 and color and value2 and color, treating those as new features:

*// in Scala*

**import** **org.apache.spark.ml.feature.RFormula**

**val** supervised **=** **new** **RFormula**()

.setFormula("lab ~ . + color:value1 + color:value2")

*# in Python*

**from** **pyspark.ml.feature** **import** RFormula

supervised = RFormula(formula="lab ~ . + color:value1 + color:value2")

At this point, we have declaratively specified how we would like to change our data into what we will train our model on. The next step is to *fit* the RFormula transformer to the data to let it discover the possible values of each column. Not all transformers have this requirement but because RFormula will automatically handle categorical variables for us, it needs to determine which columns are categorical and which are not, as well as what the distinct values of the categorical columns are. For this reason, we have to call the fit method. Once we call fit, it returns a “trained” version of our transformer we can then use to actually transform our data.

**NOTE**

We’re using the RFormula transformer because it makes performing several transformations extremely easy to do. In [Chapter 25](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch25.html#s6c2---preprocessing-and-feature-engineering), we’ll show other ways to specify a similar set of transformations and outline the component parts of the RFormula when we cover the specific transformers in MLlib.

Now that we covered those details, let’s continue on and prepare our DataFrame:

*// in Scala*

**val** fittedRF **=** supervised.fit(df)

**val** preparedDF **=** fittedRF.transform(df)

preparedDF.show()

*# in Python*

fittedRF = supervised.fit(df)

preparedDF = fittedRF.transform(df)

preparedDF.show()

Here’s the output from the training and transformation process:

+-----+----+------+------------------+--------------------+-----+

|color| lab|value1| value2| features|label|

+-----+----+------+------------------+--------------------+-----+

|green|good| 1|14.386294994851129|(10,[1,2,3,5,8],[...| 1.0|

...

| red| bad| 2|14.386294994851129|(10,[0,2,3,4,7],[...| 0.0|

+-----+----+------+------------------+--------------------+-----+

In the output we can see the result of our transformation—a column called features that has our previously raw data. What’s happening behind the scenes is actually pretty simple. RFormula inspects our data during the fit call and outputs an object that will transform our data according to the specified formula, which is called an RFormulaModel. This “trained” transformer always has the word Model in the type signature. When we use this transformer, Spark automatically converts our categorical variable to Doubles so that we can input it into a (yet to be specified) machine learning model. In particular, it assigns a numerical value to each possible color category, creates additional features for the interaction variables between colors and value1/value2, and puts them all into a single vector. We then call transform on that object in order to transform our input data into the expected output data.

Thus far you (pre)processed the data and added some features along the way. Now it is time to actually train a model (or a set of models) on this dataset. In order to do this, you first need to prepare a test set for evaluation.

**TIP**

Having a good test set is probably the most important thing you can do to ensure you train a model you can actually use in the real world (in a dependable way). Not creating a representative test set or using your test set for hyperparameter tuning are surefire ways to create a model that does not perform well in real-world scenarios. Don’t skip creating a test set—it’s a requirement to know how well your model actually does!

Let’s create a simple test set based off a random split of the data now (we’ll be using this test set throughout the remainder of the chapter):

*// in Scala*

**val** **Array**(train, test) **=** preparedDF.randomSplit(**Array**(0.7, 0.3))

*# in Python*

train, test = preparedDF.randomSplit([0.7, 0.3])

**Estimators**

Now that we have transformed our data into the correct format and created some valuable features, it’s time to actually fit our model. In this case we will use a classification algorithm called logistic regression. To create our classifier we instantiate an instance of LogisticRegression, using the default configuration or hyperparameters. We then set the label columns and the feature columns; the column names we are setting—label and features—are actually the default labels for all estimators in Spark MLlib, and in later chapters we omit them:

*// in Scala*

**import** **org.apache.spark.ml.classification.LogisticRegression**

**val** lr **=** **new** **LogisticRegression**().setLabelCol("label").setFeaturesCol("features")

*# in Python*

**from** **pyspark.ml.classification** **import** LogisticRegression

lr = LogisticRegression(labelCol="label",featuresCol="features")

Before we actually go about training this model, let’s inspect the parameters. This is also a great way to remind yourself of the options available for each particular model:

*// in Scala*

println(lr.explainParams())

*# in Python*

**print** lr.explainParams()

While the output is too large to reproduce here, it shows an explanation of all of the parameters for Spark’s implementation of logistic regression. The explainParams method exists on all algorithms available in MLlib.

Upon instantiating an untrained algorithm, it becomes time to fit it to data. In this case, this returns a LogisticRegressionModel:

*// in Scala*

**val** fittedLR **=** lr.fit(train)

*# in Python*

fittedLR = lr.fit(train)

This code will kick off a Spark job to train the model. As opposed to the transformations that you saw throughout the book, the fitting of a machine learning model is eager and performed immediately.

Once complete, you can use the model to make predictions. Logically this means tranforming features into labels. We make predictions with the transform method. For example, we can transform our training dataset to see what labels our model assigned to the training data and how those compare to the true outputs. This, again, is just another DataFrame we can manipulate. Let’s perform that prediction with the following code snippet:

fittedLR.transform(train).select("label", "prediction").show()

This results in:

+-----+----------+

|label|prediction|

+-----+----------+

| 0.0| 0.0|

...

| 0.0| 0.0|

+-----+----------+

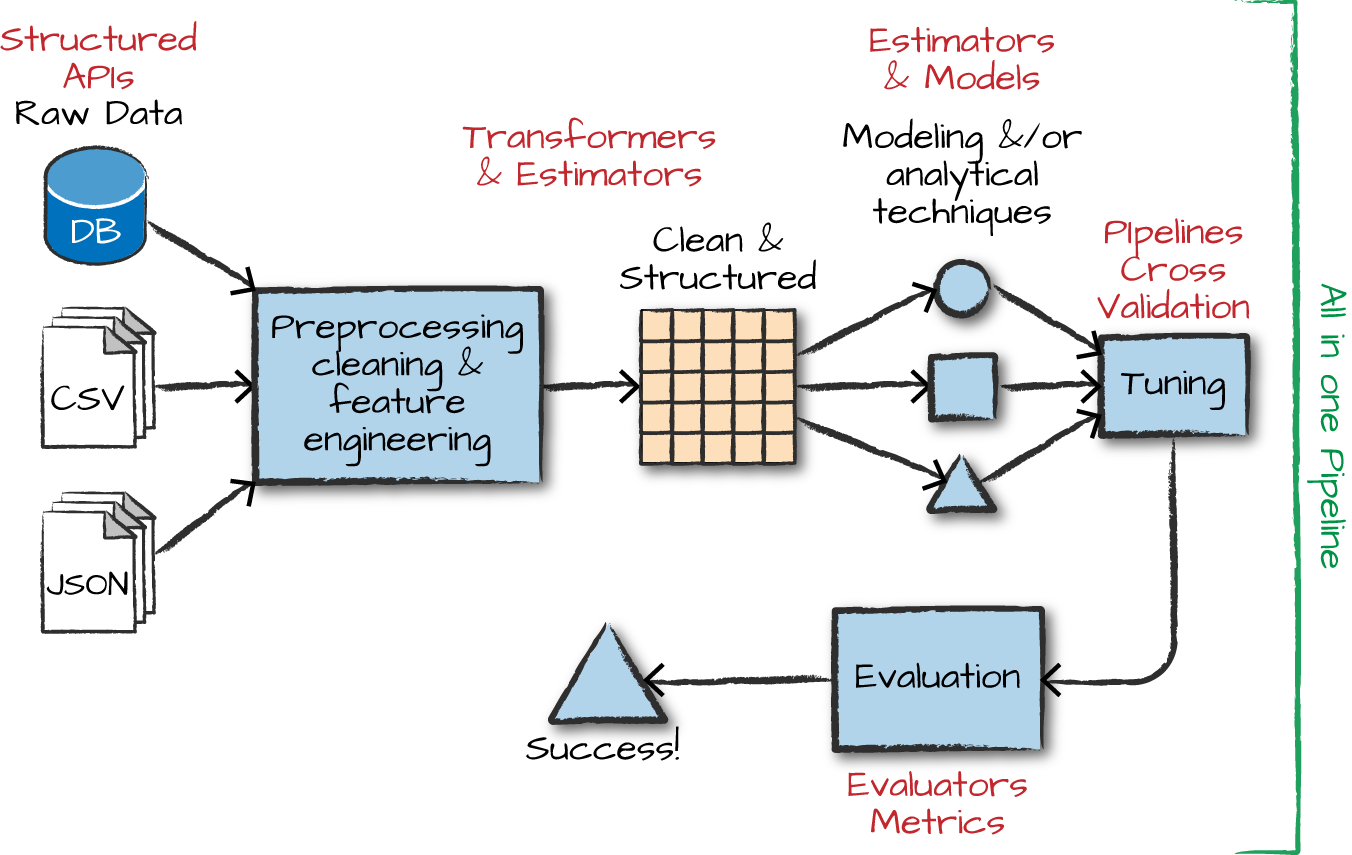
Our next step would be to manually evaluate this model and calculate performance metrics like the true positive rate, false negative rate, and so on. We might then turn around and try a different set of parameters to see if those perform better. However, while this is a useful process, it can also be quite tedious. Spark helps you avoid manually trying different models and evaluation criteria by allowing you to specify your workload as a declarative pipeline of work that includes all your transformations as well as tuning your hyperparameters.

**A REVIEW OF HYPERPARAMETERS**

Although we mentioned them previously, let’s more formally define hyperparameters. Hyperparameters are configuration parameters that affect the training process, such as model architecture and regularization. They are set prior to starting training. For instance, logistic regression has a hyperparameter that determines how much regularization should be performed on our data through the training phase (regularization is a technique that pushes models against overfitting data). You’ll see in the next couple of pages that we can set up our pipeline to try different hyperparameter values (e.g., different regularization values) in order to compare different variations of the same model against one another.

**Pipelining Our Workflow**

As you probably noticed, if you are performing a lot of transformations, writing all the steps and keeping track of DataFrames ends up being quite tedious. That’s why Spark includes the Pipeline concept. A pipeline allows you to set up a dataflow of the relevant transformations that ends with an estimator that is automatically tuned according to your specifications, resulting in a tuned model ready for use. Figure 24-4 illustrates this process.



*Figure 24-4. Pipelining the ML workflow*

Note that it is essential that instances of transformers or models are *not* reused across different pipelines. Always create a new instance of a model before creating another pipeline.

In order to make sure we don’t overfit, we are going to create a holdout test set and tune our hyperparameters based on a validation set (note that we create this validation set based on the original dataset, not the preparedDF used in the previous pages):

*// in Scala*

**val** **Array**(train, test) **=** df.randomSplit(**Array**(0.7, 0.3))

*# in Python*

train, test = df.randomSplit([0.7, 0.3])

Now that you have a holdout set, let’s create the base stages in our pipeline. A stage simply represents a transformer or an estimator. In our case, we will have two estimators. The RFomula will first analyze our data to understand the types of input features and then transform them to create new features. Subsequently, the LogisticRegression object is the algorithm that we will train to produce a model:

*// in Scala*

**val** rForm **=** **new** **RFormula**()

**val** lr **=** **new** **LogisticRegression**().setLabelCol("label").setFeaturesCol("features")

*# in Python*

rForm = RFormula()

lr = LogisticRegression().setLabelCol("label").setFeaturesCol("features")

We will set the potential values for the RFormula in the next section. Now instead of manually using our transformations and then tuning our model we just make them stages in the overall pipeline, as in the following code snippet:

*// in Scala*

**import** **org.apache.spark.ml.Pipeline**

**val** stages **=** **Array**(rForm, lr)

**val** pipeline **=** **new** **Pipeline**().setStages(stages)

*# in Python*

**from** **pyspark.ml** **import** Pipeline

stages = [rForm, lr]

pipeline = Pipeline().setStages(stages)

**Training and Evaluation**

Now that you arranged the logical pipeline, the next step is training. In our case, we won’t train just one model (like we did previously); we will train several variations of the model by specifying different combinations of hyperparameters that we would like Spark to test. We will then select the best model using an Evaluator that compares their predictions on our validation data. We can test different hyperparameters in the entire pipeline, even in the RFormula that we use to manipulate the raw data. This code shows how we go about doing that:

*// in Scala*

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

**val** params **=** **new** **ParamGridBuilder**()

.addGrid(rForm.formula, **Array**(

"lab ~ . + color:value1",

"lab ~ . + color:value1 + color:value2"))

.addGrid(lr.elasticNetParam, **Array**(0.0, 0.5, 1.0))

.addGrid(lr.regParam, **Array**(0.1, 2.0))

.build()

*# in Python*

**from** **pyspark.ml.tuning** **import** ParamGridBuilder

params = ParamGridBuilder()\

.addGrid(rForm.formula, [

"lab ~ . + color:value1",

"lab ~ . + color:value1 + color:value2"])\

.addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0])\

.addGrid(lr.regParam, [0.1, 2.0])\

.build()

In our current paramter grid, there are three hyperparameters that will diverge from the defaults:

* Two different versions of the RFormula
* Three different options for the ElasticNet parameter
* Two different options for the regularization parameter

This gives us a total of 12 different combinations of these parameters, which means we will be training 12 different versions of logistic regression. We explain the ElasticNet parameter as well as the regularization options in [Chapter 26](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch26.html#s6c3---classification).

Now that the grid is built, it’s time to specify our evaluation process. The *evaluator* allows us to automatically and objectively compare multiple models to the same evaluation metric. There are evaluators for classification and regression, covered in later chapters, but in this case we will use the BinaryClassificationEvaluator, which has a number of potential evaluation metrics, as we’ll discuss in [Chapter 26](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch26.html#s6c3---classification). In this case we will use areaUnderROC, which is the total area under the receiver operating characteristic, a common measure of classification performance:

*// in Scala*

**import** **org.apache.spark.ml.evaluation.BinaryClassificationEvaluator**

**val** evaluator **=** **new** **BinaryClassificationEvaluator**()

.setMetricName("areaUnderROC")

.setRawPredictionCol("prediction")

.setLabelCol("label")

*# in Python*

**from** **pyspark.ml.evaluation** **import** BinaryClassificationEvaluator

evaluator = BinaryClassificationEvaluator()\

.setMetricName("areaUnderROC")\

.setRawPredictionCol("prediction")\

.setLabelCol("label")

Now that we have a pipeline that specifies how our data should be transformed, we will perform model selection to try out different hyperparameters in our logistic regression model and measure success by comparing their performance using the areaUnderROC metric.

As we discussed, it is a best practice in machine learning to fit hyperparameters on a validation set (instead of your test set) to prevent overfitting. For this reason, we cannot use our holdout test set (that we created before) to tune these parameters. Luckily, Spark provides two options for performing hyperparameter tuning automatically. We can use TrainValidationSplit, which will simply perform an arbitrary random split of our data into two different groups, or CrossValidator, which performs K-fold cross-validation by splitting the dataset into *k* non-overlapping, randomly partitioned folds:

*// in Scala*

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

**val** tvs **=** **new** **TrainValidationSplit**()

.setTrainRatio(0.75) *// also the default.*

.setEstimatorParamMaps(params)

.setEstimator(pipeline)

.setEvaluator(evaluator)

*# in Python*

**from** **pyspark.ml.tuning** **import** TrainValidationSplit

tvs = TrainValidationSplit()\

.setTrainRatio(0.75)\

.setEstimatorParamMaps(params)\

.setEstimator(pipeline)\

.setEvaluator(evaluator)

Let’s run the entire pipeline we constructed. To review, running this pipeline will test out every version of the model against the validation set. Note the type of tvsFitted is TrainValidationSplitModel. Any time we fit a given model, it outputs a “model” type:

*// in Scala*

**val** tvsFitted **=** tvs.fit(train)

*# in Python*

tvsFitted = tvs.fit(train)

And of course evaluate how it performs on the test set!

evaluator.evaluate(tvsFitted.transform(test)) *// 0.9166666666666667*

We can also see a training summary for some models. To do this we extract it from the pipeline, cast it to the proper type, and print our results. The metrics available on each model are discussed throughout the next several chapters. Here’s how we can see the results:

*// in Scala*

**import** **org.apache.spark.ml.PipelineModel**

**import** **org.apache.spark.ml.classification.LogisticRegressionModel**

**val** trainedPipeline **=** tvsFitted.bestModel.asInstanceOf[**PipelineModel**]

**val** **TrainedLR** **=** trainedPipeline.stages(1).asInstanceOf[**LogisticRegressionModel**]

**val** summaryLR **=** **TrainedLR**.summary

summaryLR.objectiveHistory *// 0.6751425885789243, 0.5543659647777687, 0.473776...*

The objective history shown here provides details related to how our algorithm performed over each training iteration. This can be helpful because we can note the progress our algorithm is making toward the best model. Large jumps are typically expected at the beginning, but over time the values should become smaller and smaller, with only small amounts of variation between the values.

**Persisting and Applying Models**

Now that we trained this model, we can persist it to disk to use it for prediction purposes later on:

tvsFitted.write.overwrite().save("/tmp/modelLocation")

After writing out the model, we can load it into another Spark program to make predictions. To do this, we need to use a “model” version of our particular algorithm to load our persisted model from disk. If we were to use CrossValidator, we’d have to read in the persisted version as the CrossValidatorModel, and if we were to use LogisticRegression manually we would have to use LogisticRegressionModel. In this case, we use TrainValidationSplit, which outputs TrainValidationSplitModel:

*// in Scala*

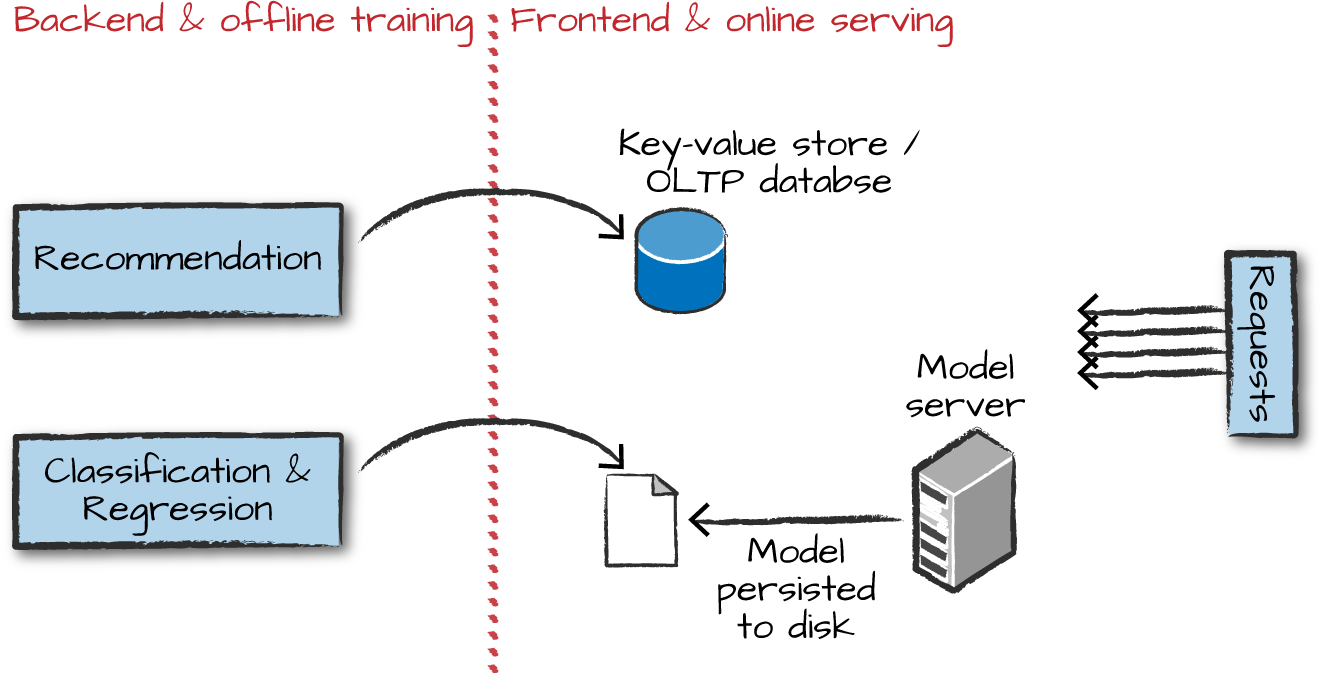
**import** **org.apache.spark.ml.tuning.TrainValidationSplitModel**

**val** model **=** **TrainValidationSplitModel**.load("/tmp/modelLocation")

model.transform(test)

**Deployment Patterns**

In Spark there are several different deployment patterns for putting machine learning models into production. Figure 24-5 illustrates common workflows.



*Figure 24-5. The productionization process*

Here are the various options for how you might go about deploying a Spark model. These are the general options you should be able to link to the process illustrated in Figure 24-5.

* Train your machine learning (ML) model offline and then supply it with offline data. In this context, we mean offline data to be data that is stored for analysis, and not data that you need to get an answer from quickly. Spark is well suited to this sort of deployment.
* Train your model offline and then put the results into a database (usually a key-value store). This works well for something like recommendation but poorly for something like classification or regression where you cannot just look up a value for a given user but must calculate one based on the input.
* Train your ML algorithm offline, persist the model to disk, and then use that for serving. This is not a low-latency solution if you use Spark for the serving part, as the overhead of starting up a Spark job can be high, even if you’re not running on a cluster. Additionally this does not parallelize well, so you’ll likely have to put a load balancer in front of multiple model replicas and build out some REST API integration yourself. There are some interesting potential solutions to this problem, but no standards currently exist for this sort of model serving.
* Manually (or via some other software) convert your distributed model to one that can run much more quickly on a single machine. This works well when there is not too much manipulation of the raw data in Spark but can be hard to maintain over time. Again, there are several solutions in progress. For example, MLlib can export some models to PMML, a common model interchange format.
* Train your ML algorithm online and use it online. This is possible when used in conjunction with Structured Streaming, but can be complex for some models.

While these are some of the options, there are many other ways of performing model deployment and management. This is an area under heavy development and many potential innovations are currently being worked on.

**Conclusion**

In this chapter we covered the core concepts behind advanced analytics and MLlib. We also showed you how to use them. The next chapter will discuss preprocessing in depth, including Spark’s tools for feature engineering and data cleaning. Then we’ll move into detailed descriptions of each algorithm available in MLlib along with some tools for graph analytics and deep learning.