Machine Learning Systems

Designs that scale

SAMPLE CHAPTER

Jeff Smith

Foreword by Sean Owen

www.itbook.store/books/9781617293337

Machine Learning Systems

by Jeff Smith

Chapter 4

Copyright 2018 Manning Publications

www.itbook.store/books/9781617293337

brief contents

PART 1 FUNDAMENTALS OF REACTIVE MACHINE LEARNING 1

1 ■ Learning reactive machine learning 3
2 ■ Using reactive tools 23

PART 2 BUILDING A REACTIVE MACHINE LEARNING SYSTEM 41

3 ■ Collecting data 43
4 ■ Generating features 69
5 ■ Learning models 93
6 ■ Evaluating models 117
7 ■ Publishing models 135
8 ■ Responding 149

PART 3 OPERATING A MACHINE LEARNING SYSTEM 165

9 ■ Delivering 167
10 ■ Evolving intelligence 177

www.itbook.store/books/9781617293337

4 Generating features

This chapter covers

- Extracting features from raw data
- Transforming features to make them more useful
- Selecting among the features you've created
- How to organize feature-generation code

This chapter is the next step on our journey through the components, or phases, of a machine learning system, shown in figure 4.1. The chapter focuses on turning raw data into useful representations called *features*. The process of building systems that can generate features from data, sometimes called *feature engineering*, can be deceptively complex. Often, people begin with an intuitive understanding of *what* they want the features used in a system to be, with few plans for *how* those features will be produced. Without a solid plan, the process of feature engineering can easily get off track, as you saw in the Sniffable example from chapter 1.

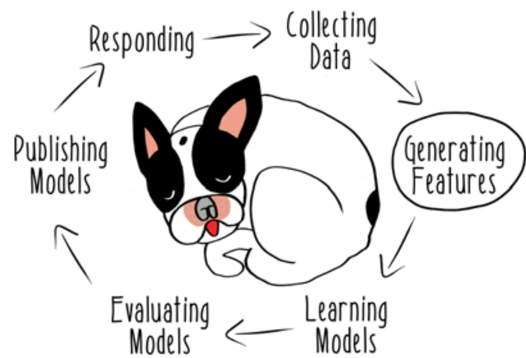


Figure 4.1 Phases of machine learning

www.itbook.store/books/9781617293337

In this chapter, I'll guide you through the three main types of operations in a feature pipeline: extraction, transformation, and selection. Not all systems do all the types of operations shown in this chapter, but all feature engineering techniques can be thought of as falling into one of these three buckets. I'll use type signatures to assign techniques to groups and give our exploration some structure, as shown in table 4.1.

Table 4.1 Phases of feature generation

Phase	Input	Output
Extract	`RawData`	`Feature`
Transform	`Feature`	`Feature`
Select	`Set[Feature]`	`Set[Feature]`

Real-world feature pipelines can have very complex structures. You'll use these groupings to help you understand how you can build a feature-generation pipeline in the best way possible. As we explore these three types of feature-processing operations, I'll introduce common techniques and design patterns that will keep your machine learning system from becoming a tangled, unmaintainable mess. Finally, we'll consider some general properties of data pipelines when discussing the next component of machine learning systems discussed in chapter 5, the model-learning pipeline.

> **Type signatures**
>
> You may not be familiar with the use of *types* to guide how you think about and implement programs. This technique is common in statically typed languages like Scala and Java. In Scala, functions are defined in terms of the inputs they take, the outputs they return, and the types of both. This is called a *type signature*. In this book, I mostly use a fairly simple form of signature notation that looks like this: `Grass => Milk`. You can read this as, "A function from an input of type Grass to an output of type Milk." This would be the type signature of some function that behaves much like a cow.

To cover this enormous scope of functionality, we need to rise above it all to gain some perspective on what features are all about. To that end, we'll join the team of Pidg'n, a microblogging social network for tree-dwelling animals, not too different from Twitter.

We'll look at how we can take the chaos of a short-form, text-based social network and build meaningful representations of that activity. Much like the forest itself, the world of features is diverse and rich, full of hidden complexity. We can, however, begin to peer through the leaves and capture insights about the lives of tree-dwelling animals using the power of reactive machine learning.

www.itbook.store/books/9781617293337

4.1 Spark ML

Before we get started building features, I want to introduce you to more Spark functionality. The `spark.ml` package, sometimes called Spark ML, defines some high-level APIs that can be used to create machine learning pipelines. This functionality can reduce the amount of machine learning–specific code that you need to implement yourself, but using it does involve a change in how you structure your data.

The Spark ML API uses mostly the same nomenclature for feature extraction, transformation, and selection that I use in this chapter, though there are subtle differences. If and when you read the Spark ML documentation, you may see something called a *transformation* operation, which I call an *extraction* operation. These are generally minor, unimportant differences that you can ignore. Different technologies name and structure this functionality differently, and you'll see all sorts of different naming conventions in the machine learning literature. The type signature–based framework for dividing feature-generation functionality that I use in this chapter is just a tool to help you implement and organize your code. Once you've mastered the feature-generation concepts in this chapter, you'll be better equipped to see through differences in nomenclature to the similarities in functionality.

Much of the machine learning functionality in Spark is designed to be used with DataFrames, instead of the RDDs that you've seen up until this point. DataFrames are simply a higher-level API on top of RDDs that give you a richer set of operations. You can think of DataFrames as something like tables in relational databases. They have different columns, which you can define and then query. Much of the recent progress in performance and functionality within Spark has been focused on DataFrames, so to get access to the full power of things like MLlib's machine learning capabilities, you'll need to use DataFrames for some operations. The good news is that they're very similar to the RDDs you've been working with and tabular data structures you may have used in other languages, such as pandas DataFrames in Python or R's data frames.

4.2 Extracting features

Now that I've introduced some of the tools, let's begin to solve the problem. We'll start our exploration of the feature engineering process at the very beginning, with raw data. In this chapter, you'll take on the role of Lemmy, an engineer on the Pidg'n data team.

Your team knows it wants to build all sorts of predictive models about user activity. You're just getting started, though, and all you have are the basics of application data: squawks (text posts of 140 characters or less), user profiles, and the follower relationships. This is a rich dataset, for sure, but you've never put it to much analytical use. To start with, you've decided to focus on the problem of predicting which new users will become *Super Squawkers*, users with more than a million followers.

www.itbook.store/books/9781617293337

To start this project, you'll extract some features to use in the rest of your machine learning system. I define the process of feature extraction as taking in raw data of some sort and returning a feature. Using Scala type signatures, feature extraction can be represented like this: `RawData => Feature`. That type signature can be read as, "A function that takes raw data and returns a feature." If you define a function that satisfies that type signature, it might look something like the stub in the following listing.

Listing 4.1 Extracting features

```
def extract(rawData: RawData): Feature = ???
```

Put differently, any output produced from raw data is a potential feature, regardless of whether it ever gets used to learn a model.

The Pidg'n data team has been collecting data since day one of the app as part of keeping the network running. You have the complete, unaltered record of all the actions ever taken by Pidg'n users, much like the data model discussed in chapter 3. Your team has built a few aggregates of that data for basic analytical purposes. Now you want to take that system to the next level by generating semantically meaningful derived representations of that raw data—features. Once you have features of any kind, you can begin learning models to predict user behavior. In particular, you're interested in seeing if you can understand what makes particular squawks and squawkers more popular than others. If a squawker has the potential to become very popular, you want to provide them with a more streamlined experience, free of advertisements, to encourage them to squawk more.

Let's begin by extracting features from the raw data of the text of squawks. You can start by defining a simple case class and extracting a single feature for a few squawks. Listing 4.2 shows how to extract a feature consisting of the words in the text of a given squawk. This implementation will use Spark's `Tokenizer` to break sentences into words. Tokenization is just one of several common text-processing utilities that come built into Spark that make writing code like this fast and easy. For advanced use cases, you may want to use a more sophisticated text-parsing library, but having common utilities easily available can be very helpful.

Listing 4.2 Extracting word features from squawks

```
case class Squawk(id: Int, text: String)

val squawks = session.createDataFrame(Seq(
  Squawk(123, "Clouds sure make it hard to look
➥ on the bright side of things."),
  Squawk(124, "Who really cares who gets the worm?
➥ I'm fine with sleeping in."),
  Squawk(125, "Why don't french fries grow on trees?")))
➥ .toDF("squawkId", "squawk")
```

Case class to hold a basic data model of a squawk

Creates a DataFrame from a sequence

Instantiates example instances of squawks

Names columns to place values in a DataFrame

www.itbook.store/books/9781617293337

```
val tokenizer = new Tokenizer().setInputCol("squawk")
➥ .setOutputCol("words")

val tokenized = tokenizer.transform(squawks)

tokenized.select("words", "squawkId").show()
```

Sets up a Tokenizer to split the text of squawks into words and put them in an output column

Executes the Tokenizer and populates the words column in a DataFrame

Prints the results for inspection

The operations in listing 4.2 give you a `DataFrame` that contains a column called `words`, which has all the words in the text of the squawk. You could call the values in the `words` column a *feature*. These values could be used to learn a model. But let's make the semantics of the pipeline clearer using the Scala type system.

Using the code in listing 4.3, you can define what a feature is and what specific sort of feature you've produced. Then, you can take the `words` column from that `DataFrame` and use it to instantiate instances of those feature classes. It's the same words that the `Tokenizer` produced for you, but now you have richer representations that you can use to help build up a feature-generation pipeline.

Listing 4.3 Extracting word features from squawks

```
trait FeatureType {
  val name: String
  type V
}

trait Feature extends FeatureType {
  val value: V
}

case class WordSequenceFeature(name: String, value: Seq[String])
➥ extends Feature {
  type V = Seq[String]
}

val wordsFeatures = tokenized.select("words")
  .map(row =>
  WordSequenceFeature("words",
    row.getSeq[String](0)))

wordsFeatures.show()
```

Defines a base trait for all types of features

Requires feature types to have names

Type parameter to hold the type of values generated by feature

Defines a base trait for all features as an extension of feature types

Requires that features have values of the type specified in the feature type

Defines a case class for features consisting of word sequences

Specifies that the type of features being generated is a sequence of strings (words)

Maps over rows and applies a function to each

Selects a words column from the DataFrame

Gets extracted words out of a row

Creates an instance of WordSequenceFeature named words

Prints features for inspection

With this small bit of extra code, you can define your features in a way that's more explicit and less tied to the specifics of the raw data in the original `DataFrame`. The resulting value is an RDD of `WordSequenceFeature`. You'll see later how you can

www.itbook.store/books/9781617293337

continue to use this `Feature` trait with specific case classes defining the different types of features in your pipeline.

Also note that, when operating over the `DataFrame`, you can use a pure, anonymous, higher-order function to create instances of your features. The concepts of purity, anonymous functions, and higher-order functions may have sounded quite abstract when I introduced them in chapter 1. But now that you've seen them put to use in several places, I hope it's clear that they can be very simple to write. Now that you've gotten some Scala and Spark programming under your belt, I hope you're finding it straightforward to think of data transformations like feature extraction in terms of pure functions with no side effects.

You and the rest of the Pidg'n data team could now use these features in the next phase of the machine learning pipeline—model learning—but they probably wouldn't be good enough to learn a model of Super Squawkers. These initial word features are just the beginning. You can encode far more of your understanding of what makes a squawker super into the features themselves.

To be clear, there are sophisticated model-learning algorithms, such as neural networks, that require very little feature engineering on the data that they consume. You *could* use the values you've just produced as features in a model-learning process. But many machine learning systems will require you to do far more with your features before using them in model learning if you want acceptable predictive performance. Different model-learning algorithms have different strengths and weaknesses, as we'll explore in chapter 5, but all of them will benefit from having base features transformed in ways that make the process of model learning simpler. We need to move on to see how to make features out of other features.

4.3 Transforming features

Now that you've extracted some basic features, let's figure out how to make them useful. This process of taking a feature and producing a new feature from it is called *feature transformation*. In this section, I'll introduce you to some common transform functions and discuss how they can be structured. Then I'll show you a very important class of feature transformations: transforming features into concept labels.

What is feature transformation? In the form of a type signature, feature transformation can be expressed as `Feature => Feature`, a function that takes a feature and returns a feature. A stub implementation of a transformation function (sometimes called a *transform*) is shown in the next listing.

Listing 4.4 Transforming features

```
def transform(feature: Feature): Feature = ???
```

In the case of the Pidg'n data team, you've decided to build on your previous feature-engineering work by creating a feature consisting of the frequencies of given words in

www.itbook.store/books/9781617293337

a squawk. This quantity is sometimes called a *term frequency*. Spark has built-in functionality that makes calculating this value easy.

Listing 4.5 Transforming words to term frequencies

```
val hashingTF = new HashingTF()
  .setInputCol("words")
  .setOutputCol("termFrequencies")

val tfs = hashingTF.transform(tokenized)

tfs.select("termFrequencies").show()
```

- `val hashingTF = new HashingTF()`: Instantiates an instance of a class to calculate term frequencies
- `.setInputCol("words")`: Defines an input column to read from when consuming DataFrames
- `.setOutputCol("termFrequencies")`: Defines an output to put term frequencies in
- `val tfs = hashingTF.transform(tokenized)`: Executes the transformation
- `tfs.select("termFrequencies").show()`: Prints term frequencies for inspection

It's worth noting that the `hashingTF` implementation of term frequencies was implemented to consume the `DataFrame` you previously produced, not the features you designed later. Spark ML's concept of a pipeline is focused on connecting operations on `DataFrames`, so it can't consume the features you produced before without more conversion code.

Feature hashing

The use of the term *hashing* in the Spark library refers to the technique of *feature hashing*. Although it's not always used in feature-generation pipelines, feature hashing can be a critically important technique for building large numbers of features. In text-based features like term frequencies, there's no way of knowing a priori what all the possible features could be. Squawkers can write anything they want in a squawk on Pidg'n. Even an English-language dictionary wouldn't contain all the slang terms squawkers might use. Free-text input means that the universe of possible terms is effectively infinite.

One solution is to define a hash range of the size of the total number of distinct features you want to use in your model. Then you can apply a deterministic hashing function to each input to produce a distinct value within the hash range, giving you a unique identifier for each feature. For example, suppose `hash("trees")` returns `65381`. That value will be passed to the model-learning function as the identifier of the feature. This might not seem much more useful than just using `"trees"` as the identifier, but it is. When I discuss prediction services in chapter 7, I'll talk about why you'll want to be able to identify features that the system has possibly never seen before.

Let's take a look at how Spark ML's `DataFrame`-focused API is intended to be used in connecting operations like this. You won't be able to take full advantage of Spark ML until chapter 5, where you'll start learning models, but it's still useful for feature generation. Some of the preceding code can be reimplemented using a `Pipeline` from Spark ML. That will allow you to set the tokenizer and the term frequency operations as stages within a pipeline.

www.itbook.store/books/9781617293337

Listing 4.6 Using Spark ML pipelines

```
val pipeline = new Pipeline()
  .setStages(Array(tokenizer, hashingTF))

val pipelineHashed = pipeline.fit(squawksDF)

println(pipelineHashed.getClass)
```

- Instantiates a new pipeline
- Sets the two stages of this pipeline
- Executes the pipeline
- Prints the type of the result of the pipeline, a PipelineModel

This `Pipeline` doesn't result in a set of features, or even a `DataFrame`. Instead, it returns a `PipelineModel`, which in this case won't be able to do anything useful, because you haven't learned a model yet. We'll revisit this code in chapter 5, where we can go all the way from feature generation through model learning. The main thing to note about this code at this point is that you can encode a pipeline as a clear abstraction within your application. A large fraction of machine learning work involves working with pipeline-like operations. With the Spark ML approach to pipelines, you can be very explicit about how your pipeline is composed by setting the stages of the pipeline in order.

4.3.1 Common feature transforms

Sometimes you don't have library implementations of the feature transform that you need. A given feature transform might have semantics that are specific to your application, so you'll often need to implement feature transforms yourself.

Consider how you could build a feature to indicate that a given Pidg'n user was a Super Squawker (user with more than a million followers). The feature-extraction process will give you the raw data about the number of followers a given squawker has. If you used the number of followers as a feature, that would be called a *numerical* feature. That number would be an accurate snapshot of the data from the follower graph, but it wouldn't necessarily be easy for all model-learning algorithms to use. Because your intention is to express the idea of a Super Squawker, you could use a far simpler representation: a Boolean value representing whether or not the squawker has more than a million followers.

The squirrel, a rather ordinary user, has very few followers. But the sloth is an terrific Super Squawker. To produce meaningful features about the differences between these two squawkers, you'll follow the same process of going from raw data, to numeric features, and then to Boolean features. This series of data transformations is shown for the two users in figure 4.2.

www.itbook.store/books/9781617293337

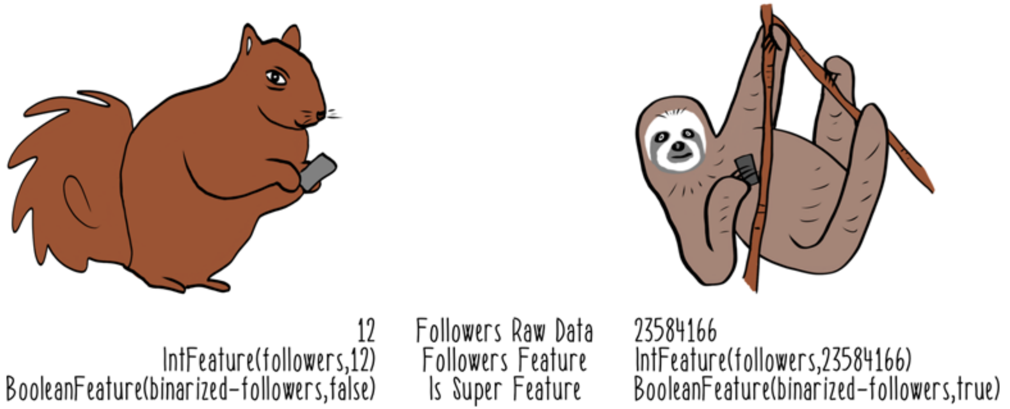


Figure 4.2 Feature transformations

The following listing shows how to implement this approach to binarization to produce a Super Squawker feature.

Listing 4.7 Binarizing a numerical feature

```
case class IntFeature(name: String, value: Int) extends Feature {
  type V = Int
}

case class BooleanFeature(name: String, value: Boolean) extends Feature {
  type V = Boolean
}

def binarize(feature: IntFeature, threshold: Double): BooleanFeature = {
  BooleanFeature("binarized-" + feature.name, feature.value > threshold)
}

val SUPER_THRESHOLD = 1000000

val squirrelFollowers = 12
val slothFollowers = 23584166

val squirrelFollowersFeature = IntFeature("followers", squirrelFollowers)
val slothFollowersFeature = IntFeature("followers", slothFollowers)

val squirrelIsSuper = binarize(squirrelFollowers, SUPER_THRESHOLD)
val slothIsSuper = binarize(slothFollowers, SUPER_THRESHOLD)
```

Annotations:
- Case class representing a numerical feature where the value is an integer
- Specifies that these are integer features
- Case class representing a Boolean feature
- Specifies that these are Boolean features
- Function that takes a numeric integer feature and threshold and returns a Boolean feature
- Adds the name of the transform function to the resulting feature name
- Constant defining the cutoff for a squawker to be super
- Raw numbers of followers for the squirrel and the sloth
- Numeric integer feature representing the number of followers
- Boolean feature indicating the squirrel is not a Super Squawker
- Boolean feature indicating the sloth is a Super Squawker

www.itbook.store/books/9781617293337

The `binarize` function is a good example of a reusable transform function. It also ensures the resulting feature is somewhat self-describing by appending the name of the transform function to the resulting feature. Ensuring that we can identify the operations that were applied to produce a feature is an idea we'll revisit in later chapters. Finally, note that the transformation function `binarize` is a pure function.

Using only pure functions in feature transforms is an important part of establishing a coherent structure for feature-generation code. Separating feature extraction and feature transformation within a code base can be difficult, and the boundaries between the two can be hard to draw. Ideally, any I/O or side-effecting operations should be contained in the feature-extraction phase of the pipeline, with all transformations' functionality being implemented as pure functions. As you'll see later, pure transforms are simple to scale and easy to reuse across features and feature-extraction contexts (model learning and predicting).

There's a huge range of commonly used transformation functions. Similar to binarization, some approaches reduce continuous values to discrete labels. For example, a feature designed to express the time of day when a squawk was posted might not use the full timestamp. Instead, a more useful representation could be to transform all times into a limited set of labels, as shown in table 4.2.

Table 4.2 Transforming times into time labels

Time	Label
7:02	Morning
12:53	Midday
19:12	Night

The implementation of a transform to do this is trivial and is naturally a pure function.

There's another variation on reducing continuous data to labels, called *binning*, in which the source feature is reduced to some arbitrary label defined by the range of values that it falls into. For example, you could take the number of squawks a given user has made and reduce it to one of three labels indicating how active the squawker is, as shown in table 4.3.

Table 4.3 Binning

Squawks	Label	Activity level
7	0_99	Least active squawkers
1,204	1000_99999	Moderately active squawkers
2,344,910	1000000_UP	Most active squawkers

www.itbook.store/books/9781617293337

Again, an implementation of such a transform would be trivial and naturally a pure function. Transforms *should* be easy to write and should correspond closely to their formulation in mathematical notation. When it comes to implementing transforms, you should always abide by the KISS principle: Keep It Simple, Sparrow. Reactive machine learning systems are hard enough to implement without implementing complicated transforms. Usually, an overly long transform implementation is a smell that someone has laid a rotten egg. In a few special cases, you may want to implement something like a transformer with more involved semantics. We'll consider such circumstances later in this chapter and later in the book.

4.3.2 Transforming concepts

Before we leave the topic of transformations, we need to consider one very common and critical class of feature transformations: the ones that produce concepts. As mentioned in chapter 1, *concepts* are the things that a machine learning model is trying to predict. Although some machine learning algorithms can learn models of continuous concepts, such as the number of squawks a given user will write over the course of the next month, many machine learning systems are built to perform classification. In *classification* problems, the learning algorithm is trying to learn a discrete number of class labels, not continuous values. In such systems, the concept has to be produced from the raw data, during feature extraction, and then reduced to a class label via transformation. Concept class labels aren't exactly the same thing as features, but often the difference is just a matter of how we use the piece of data. Typically, and ideally, the same code that might binarize a feature will also binarize a concept.

Building on the code in listing 4.7, in the next listing, take the Boolean feature about Super Squawkers and produce a Boolean concept label that classifies squawkers into super or not.

Listing 4.8 Creating concept labels from features

```
trait Label extends Feature                  <- Defines labels as subtypes of features

case class BooleanLabel(name: String, value: Boolean) extends Label {   <- Creates a case class for Boolean labels
  type V = Boolean
}

def toBooleanLabel(feature: BooleanFeature) = {   <- Defines a simple conversion function from Boolean features to Boolean labels
  BooleanLabel(feature.name, feature.value)
}

val squirrelLabel = toBooleanLabel(squirrelIsSuper)   <- Converts Super Squawker feature values into concept labels
val slothLabel = toBooleanLabel(slothIsSuper)

Seq(squirrelLabel, slothLabel).foreach(println)   <- Prints label values for inspection
```

www.itbook.store/books/9781617293337

In this code, you've defined concept labels as a special subtype of features. That's not how features and labels are generally discussed, but it can be a helpful convention for code reuse in machine learning systems. Whether you intend to do so or not, any given feature value could be used as a concept label if it represents the concept class to be learned. The `Label` trait in listing 4.8 doesn't change the underlying structure of the data in a feature, but it does allow you to annotate when you're using a feature as a concept label. The rest of the code is quite simple, and you arrive at the same conclusion again: people just aren't that interested in what squirrels have to say.

4.4 Selecting features

Again, you find yourself in the same situation: if you've done all the work so far, you might now be finished. You could use the features you've already produced to learn a model. But sometimes it's worthwhile to perform additional processing on features before beginning to learn a model. In the previous two phases of the feature-generation process, you produced all the features you *might* want to use to learn a model, sometimes called a *feature set*. Now that you have that feature set, you could consider throwing some of those features in the trash.

The process of choosing from a feature set which features to use is known as *feature selection*. In type-signature form, it can be expressed `Set[Feature] => Set[Feature]`, a function that takes a set of features and returns another set of features. The next listing shows a stub implementation of a feature selector.

Listing 4.9 A feature selector

```
def select(featureSet: Set[Feature]): Set[Feature] = ???
```

Why would you ever want to discard features? Aren't they useful and valuable? In theory, a robust machine learning algorithm could take as input feature vectors containing arbitrary numbers of features and learn a model of the given concept. In reality, providing a machine learning algorithm with too many features is just going to make it take longer to learn a model and potentially degrade that model's performance. You can find yourself needing to choose among features quite easily. By varying the parameters used in the transformation process, you could create an infinite number of features with a very small amount of code.

Using a modern distributed data-processing framework like Spark makes handling arbitrarily sized datasets easy. It's definitely to your benefit to consider a huge range of features during the feature extraction and transformation phases of your pipeline. And once you've produced all the features in your feature set, you can use some of the facilities in Spark to cut that feature set down to just those features that your model-learning algorithm will use to learn the model. There are implementations of feature-selection functionality in other machine learning libraries; Spark's MLlib is one of many options and certainly not the oldest one. For some cases, the feature-selection functionality provided by MLlib might not be sufficient, but the principles of feature

www.itbook.store/books/9781617293337

selection are the same whether you use a library implementation or something more bespoke. If you end up writing your own version of feature selection, it will still be conceptually similar to MLlib's implementations.

Using the Spark functionality will again require you to leave behind your feature-case classes and the guarantees of static typing to use the machine learning functionality implemented around the high-level DataFrame API. To begin, you'll need to construct a `DataFrame` of training instances. These instances will consist of three parts: an arbitrary identifier, a feature vector, and a concept label. The following listing shows how to build up this collection of instances. Instead of using real features, you'll use some synthetic data, which you can imagine being about various properties of Squawkers.

Listing 4.10 A `DataFrame` of instances

```
val instances = Seq(                                    <- Defines a collection of instances
  (123, Vectors.dense(0.2, 0.3, 16.2, 1.1), 0.0),       <- Hardcodes some synthetic feature
  (456, Vectors.dense(0.1, 1.3, 11.3, 1.2), 1.0),          and concept label data
  (789, Vectors.dense(1.2, 0.8, 14.5, 0.5), 0.0)
)

val featuresName = "features"                           <- Names for features and label columns
val labelName = "isSuper"

val instancesDF = session.createDataFrame(instances)    <- Creates a DataFrame from the instances collection
  .toDF("id", featuresName, labelName)                  <- Sets the name of each column in the DataFrame
```

Once you have a `DataFrame` of instances, you can take advantage of the feature-selection functionality built into MLlib. You can apply a chi-squared statistical test to rank the impact of each feature on the concept label. This is sometimes called *feature importance*. After the features are ranked by this criterion, the less impactful features can be discarded prior to model learning. The next listing shows how you can select the two most important features from your feature vectors.

Listing 4.11 Chi-squared-based feature selection

```
val selector = new ChiSqSelector()                      <- Creates a new feature selector
  .setNumTopFeatures(2)                                 <- Sets the number of features to retain to 2
  .setFeaturesCol(featuresName)                         <- Sets the column where features are
  .setLabelCol(labelName)                               <- Sets the column where concept labels are
  .setOutputCol("selectedFeatures")                     <- Sets the column to place results, the selected features

val selectedFeatures = selector.fit(instancesDF)        <- Fits a chi-squared model to the data
  .transform(instancesDF)                               <- Selects the most important features and returns a new DataFrame

selectedFeatures.show()                                 <- Prints the resulting DataFrame for inspection
```

www.itbook.store/books/9781617293337

As you can see, having standard feature-selection functionality available at a library call makes feature selection pretty convenient. If you had to implement chi-squared-based feature selection yourself, you'd find that the implementation was a lot longer than the code you just wrote.

4.5 Structuring feature code

In this chapter, you've written example implementations of all the most common components of a feature-generation pipeline. As you've seen, some of these components are simple and easy to build, and you could probably see yourself building quite a few of them without any difficulty. If you've Kept It Simple, Sparrow, you shouldn't be intimidated by the prospect of producing lots of feature extraction, transformation, and selection functionality in your system. Or should you?

Within a machine learning system, feature-generation code can often wind up being the largest part of the codebase by some measures. A typical Scala implementation might have a class for each extraction and transformation operation, and that can quickly become unwieldy as the number of classes grows. To prevent feature-generation code from becoming a confusing grab bag of various arbitrary operations, you need to start putting more of your understanding of the semantics of feature generation into the structure of your implementation of feature-generation functionality. The next section introduces one such strategy for structuring your feature-generation code.

4.5.1 Feature generators

At the most basic level, you need to define an implementation of what is a unit of feature-generation functionality. Let's call this a *feature generator*. A feature generator can encompass either extraction or both extraction and transformation operations. The implementation of the extraction and transformation operations may not be very different from what you've seen before, but these operations will all be encapsulated in an independently executable unit of code that produces a feature. Your feature generators will be things that can take raw data and produce features that you want to use to learn a model.

Let's implement your feature generators using a trait. In Scala, *traits* are used to define behaviors in the form of a type. A typical trait will include the signatures and possibly implementations of methods that define the common behavior to the trait. Scala traits are very similar to interfaces in Java, C++, and C# but are much easier and more flexible to use than interfaces in any of those languages.

For the purpose of this section, let's say that your raw data, from the perspective of your feature-generation system, consists of squawks. Feature generation will be the process of going from squawks to features. The corresponding feature-generator trait can be defined.

www.itbook.store/books/9781617293337

Listing 4.12 A feature-generator trait

```
trait Generator {

  def generate(squawk: Squawk): Feature

}
```

The `Generator` trait defines a feature generator to be an object that implements a method, `generate`, that takes a squawk and returns a feature. This is a concrete way of defining the behavior of feature generation. A given implementation of feature generation might need all sorts of other functionality, but this is the part that will be common across all implementations of feature generation. Let's look at one implementation of this trait.

Your team is interested in understanding how squawk length affects squawk popularity. There's an intuition that even 140 characters is too much to read for some squawkers, such as hummingbirds. They just get bored too quickly. Conversely, vultures have been known to stare at the same squawk for hours on end, so long posts are rarely a problem for them. For you to be able to build a recommendation model that will surface relevant content to these disparate audiences, you'll need to encode some of the data around squawk length as a feature. This can easily be implemented using the `Generator` trait.

As discussed before, the idea of length can be captured using the technique of binning to reduce your numeric data to categories. There's not much difference between a 72-character squawk and a 73-character squawk; you're just trying to capture the approximate size of a squawk. You'll divide squawks into three categories based on length: short, moderate, and long. You'll define your thresholds between the categories to be at the thirds of the total possible length. Implemented according to your `Generator` trait, you get something like the following listing.

Listing 4.13 A categorical feature generator

```
object SquawkLengthCategory extends Generator {

  val ModerateSquawkThreshold = 47
  val LongSquawkThreshold = 94

  private def extract(squawk: Squawk): IntFeature = {
    IntFeature("squawkLength", squawk.text.length)
  }

  private def transform(lengthFeature: IntFeature): IntFeature = {
```

Defines a generator as an object that extends the Generator trait

Constant thresholds to compare against

Extracting: uses the length of the squawk to instantiate an IntFeature

Transforming: takes the IntFeature of length, returns the IntFeature of category

www.itbook.store/books/9781617293337

```
  val squawkLengthCategory = lengthFeature match {
    case IntFeature(_, length) if length < ModerateSquawkThreshold => 1
    case IntFeature(_, length) if length < LongSquawkThreshold => 2
    case _ => 3
  }

  IntFeature("squawkLengthCategory", squawkLengthCategory)
}

def generate(squawk: Squawk): IntFeature = {
  transform(extract(squawk))
}
}
```

Uses a pattern-matching structure to determine which category the squawk length falls into

Returns Int for a category (for ease of use in model learning)

Returns a category of 3, a long squawk, in all other cases

Returns a category as a new IntFeature

Generating: extracts a feature from the squawk and then transforms it to a categorical IntFeature

This generator is defined in terms of a singleton object. You don't need to use instances of a class, because all the generation operations are themselves pure functions.

Internal to your implementation of the feature generator, you still used a concept of extraction and transformation, even though you now only expose a `generate` method as the public API to this object. Though that may not always seem necessary, it can be helpful to define all extraction and transformation operations in a consistent manner using feature-based type signatures. This can make it easier to compose and reuse code.

Reuse of code is a huge issue in feature-generation functionality. In a given system, many feature generators will be performing operations very similar to each other.

A given transform might be used dozens of times if it's factored out and reusable. If you don't think about such concerns up front, you may find that your team has reimplemented some transform, like averaging five different times in subtly different ways across your feature-generation codebase. That can lead to tricky bugs and bloated code.

You don't want your feature-generation code to be messier than a tree full of marmosets! Let's take a closer look at the structure of your generator functionality. The `transform` function in listing 4.13 was doing something you might wind up doing a lot in your codebase: categorizing according to some threshold. Let's look at it again.

Listing 4.14 Categorization using pattern matching

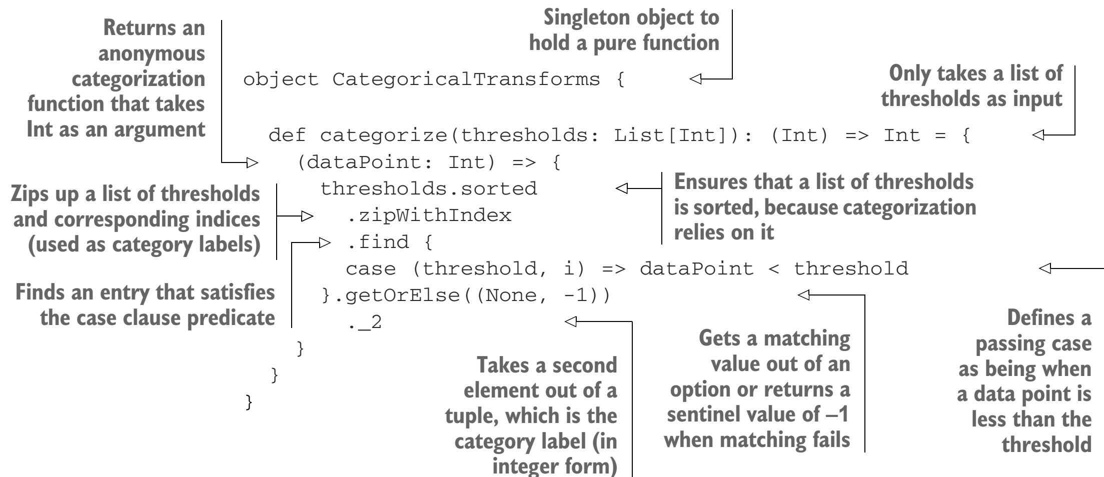
```
private def transform(lengthFeature: IntFeature): IntFeature = {
  val squawkLengthCategory = lengthFeature match {
    case IntFeature(_, length) if length < ModerateSquawkThreshold => 1
    case IntFeature(_, length) if length < LongSquawkThreshold => 2
    case _ => 3
  }
```

www.itbook.store/books/9781617293337

You definitely shouldn't be implementing a comparison against thresholds more than once, so let's find a way to pull that code out and make it reusable. It's also weird that you had to define the class label integers yourself. Ideally, you'd just have to worry about your thresholds and nothing else.

Let's pull out the common parts of this code for reuse and make it more general in the process. The code in the next listing shows one way of doing this. It's a little dense, so we'll walk through it in detail.

Listing 4.15 Generalized categorization




```
object CategoricalTransforms {

  def categorize(thresholds: List[Int]): (Int) => Int = {
    (dataPoint: Int) => {
      thresholds.sorted
        .zipWithIndex
        .find {
        case (threshold, i) => dataPoint < threshold
      }.getOrElse((None, -1))
        ._2
    }
  }
}
```

This solution uses a few techniques that you may not have seen before. For one, this function's return type is `(Int) => Int`, a function that takes an integer and returns an integer. In this case, the function returned will categorize a given integer according to the thresholds previously provided.

The thresholds and categories are also zipped together so they can be operated on as a pair of related values (in the form of a tuple). *Zipping*, or *convolution* as it's sometimes called, is a powerful technique that's commonly used in Scala and other languages in the functional programming tradition. The name *zip* comes from the similarity to the action of a zipper. In this case, you're using a special sort of zip operation that conveniently provides you indices corresponding to the number the elements in the collection being zipped over. This approach to producing indices is far more elegant than C-style iteration using a mutable counter, which you may have seen in other languages, such as Java and C++.

After zipping over the values, you use another new function, `find`, with which you can define the element of a collection you're looking for in terms of a *predicate*. Predicates are Boolean functions that are either true or false, depending on their values. They're commonly used in mathematics, logic, and various forms of programming

www.itbook.store/books/9781617293337

such as logic and functional programming. In this usage, the predicate gives you a clear syntax for defining what constitutes falling into a category bucket.

This code also deals with uncertainty in external usage in ways that you haven't before. Specifically, it sorts the categories, because they might not be provided in a sorted list, but your algorithm relies on operating on them in order. Also, the `find` function returns an `Option` because the `find` operation may or may not find a matching value. In this case, you use the value `-1` to indicate an unusable category, but how a categorization failure should be handled depends a lot on how the functionality will be integrated in the client generator code. When you factor out common feature transforms to shared functions like this, you should take into account the possibilities of future broad usage of the transform. By implementing it with these extra guarantees, you reduce the chances that someone will use your categorization functionality in the future and not get the results they wanted.

The code in listing 4.15 might be a bit harder to understand than the original implementation in listings 4.13 and 4.14. Your refactored version does more work to give you a more general and robust version of categorization. You may not expect every implementer of a feature generator to go through this much work for a simple transform, but because you've factored out this functionality to shared, reusable code, they don't have to. Any feature-generation functionality needing to categorize values according to a list of thresholds can now call this function. The transform from listings 4.13 and 4.14 can now be replaced with the very simple version in listing 4.16. You still have a relatively complex implementation of categorization in listing 4.15, but now, that complex implementation has been factored out to a separate component, which is more general and reusable. As you can see in the next listing, the callers of that functionality, like this transform function, can be quite simple.

Listing 4.16 Refactored categorization transform

```
import CategoricalTransforms.categorize

private def transform(lengthFeature: IntFeature): IntFeature = {
  val squawkLengthCategory = categorize(Thresholds)
  ➥ (lengthFeature.value)
  IntFeature("squawkLengthCategory", squawkLengthCategory)
}
```

Creates the categorization function and applies it to the value for categorization

Once you have dozens of categorical features, this sort of design strategy will make your life a lot easier. Categorization is now simple to plug in and easy to refactor should you decide to change how you want it implemented.

4.5.2 Feature set composition

You've seen how you can choose among the features you produced, but there's actually a zeroth step that's necessary in some machine learning systems. Before you even

www.itbook.store/books/9781617293337

begin the process of feature generation, you may want to choose which feature generators should be executed. Different models need different features provided to them. Moreover, sometimes you need to apply specific overrides to your normal usage of data because of business rules, privacy concerns, or legal reasons.

In the case of Pidg'n, you have some unique challenges due to your global scale. Different regions have different regulatory regimes governing the use of their citizens' data. Recently, a new government has come to power in the rainforests of Panama.

The new minister of commerce, an implacable poison-dart frog, has announced new regulation restricting the use of social-media user data for non-rainforest purposes. After consultation with your lawyers, you decide that the new law means that features using data from rainforest users should only be used in the context of models to be applied on recommendations for residents of the rainforest.

Let's look at what impact this change might have on your codebase. To make things a bit more concise, let's define a simple trait to allow you to make simplified generators quickly. This will be a helper to allow you to skip over generator-implementation details that aren't relevant to feature-set composition. The next listing defines a stub feature generator that returns random integers.

Listing 4.17 A stub feature-generator trait

```
trait StubGenerator extends Generator {
  def generate(squawk: Squawk) = {
    IntFeature("dummyFeature", Random.nextInt())
  }
}
```

- Implementation of the generate method for implementers of trait to use
- Returns random integers

Using this simple helper trait, you can now explore some of the possible impacts that the rainforest data-usage rules might have on your feature-generation code. Let's say the code responsible for assembling your feature generators looks like the following listing.

Listing 4.18 Initial feature set composition

```
object SquawkLanguage extends StubGenerator {}

object HasImage extends StubGenerator {}

object UserData extends StubGenerator {}

val featureGenerators = Set(SquawkLanguage, HasImage, UserData)
```

- Normal feature generator about the language the squawk was written in
- Normal feature generator about whether the squawk contains an image
- User-data feature generator that must be changed
- Set of all the feature generators to execute to produce data

www.itbook.store/books/9781617293337

Figure 4.3 Multiple feature-generator sets

Now you need to restructure this code to have one feature set produced for your normal, global models and one feature set for your rainforest models, as shown in figure 4.3. The following listing shows an approach to defining these two different sets of feature generators.

Listing 4.19 Multiple feature sets

```
object GlobalUserData extends StubGenerator {}

object RainforestUserData extends StubGenerator {}

val globalFeatureGenerators = Set(SquawkLanguage, HasImage,
➥ GlobalUserData)

val rainforestFeatureGenerators = Set(SquawkLanguage, HasImage,
➥ RainforestUserData)
```

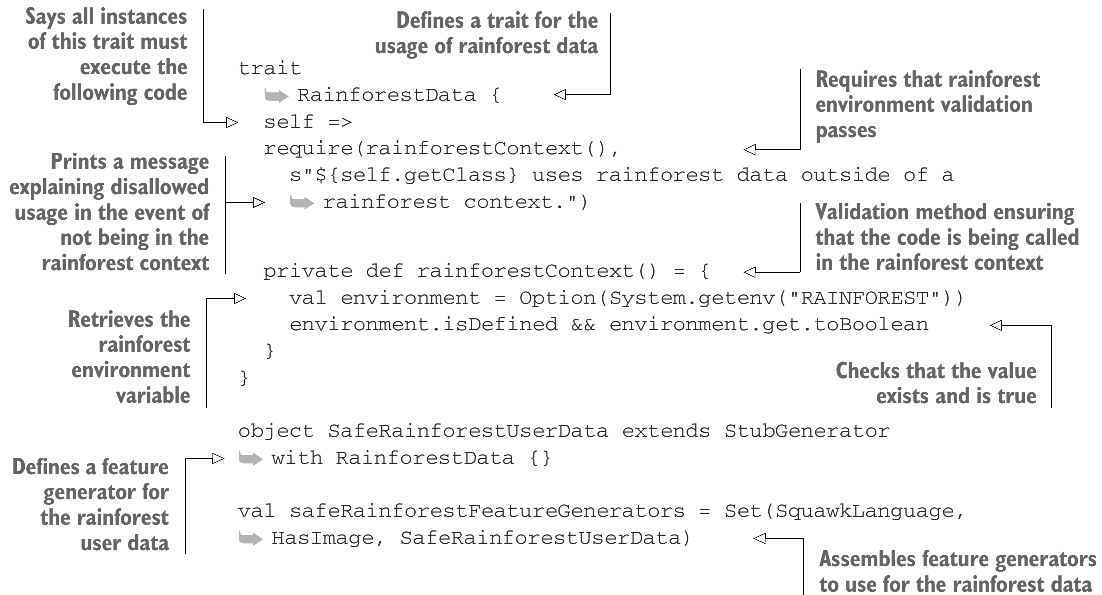
- User-data feature generator that will only access non-rainforest data (GlobalUserData)
- User-data feature generator that will only access rainforest data (RainforestUserData)
- Set of features available to be used on global models (globalFeatureGenerators)
- Set of features available to be used on rainforest models (rainforestFeatureGenerators)

You could stop with this implementation if you chose. As long as the rainforest feature generators are being used for rainforest models, you've done what the frog asked. But there are reasons to keep working on this problem. Machine learning systems are incredibly complicated to implement. Common feature-generation functionality can get reused in all sorts of places. The implementation in listing 4.19 is correct, but with Pidg'n's rapid growth, new engineers unfamiliar with this data-usage issue might refactor this code in such a way as to misuse rainforest feature data.

Let's see if you can make misusing this data even harder by defining a trait that allows you to mark code as having rainforest user data in it.

www.itbook.store/books/9781617293337

Listing 4.20 Ensuring correct usage of rainforest data




```
trait
➥ RainforestData {
  self =>
  require(rainforestContext(),
    s"${self.getClass} uses rainforest data outside of a
    ➥ rainforest context.")

  private def rainforestContext() = {
    val environment = Option(System.getenv("RAINFOREST"))
    environment.isDefined && environment.get.toBoolean
  }
}

object SafeRainforestUserData extends StubGenerator
➥ with RainforestData {}

val safeRainforestFeatureGenerators = Set(SquawkLanguage,
➥ HasImage, SafeRainforestUserData)
```

This code will throw an exception unless you've explicitly defined an environment variable `RAINFOREST` and set it to `TRUE`. If you want to see this switch in action, you can export that variable in a terminal window, if you're using macOS or Linux.

Listing 4.21 Exporting an environment variable

```
export RAINFOREST=TRUE
```

Then you can execute the code from listing 4.20 again, in the same terminal window, without getting exceptions. That's similar to how you can use this in your production feature-generation jobs. Using any of several different mechanisms in your configuration, build, or job-orchestration functionality, you can ensure that this variable is set properly for rainforest feature-generation jobs and not set for global feature-generation jobs. A new engineer creating a new feature-generation job for some other purpose would have no reason to set this variable. If that engineer misused the rainforest feature generator, that misuse would immediately manifest the first time the job was executed in any form.

Configuration

Using environment variables is one of many different methods to configure components of your machine learning system. It has the advantage of being simple to get started with and broadly supported.

www.itbook.store/books/9781617293337

> ***(continued)***
> As your machine learning system grows in complexity, you'll want to ensure that you have a well-thought-out plan for dealing with configuration. After all, properties of your machine learning system set as configurations can determine a lot about whether it remains responsive in the face of errors or changes in load. Part 3 of this book addresses most of these issues, where we consider the challenges of operating a machine learning system. The good news is that you'll find a lot of versatile tools from the Scala and big data ecosystems that will help you tame some of the complexity of dealing with configurations.

4.6 Applications

You're probably not an arboreal animal, and you may not even operate a microblogging service. But if you're doing machine learning, you're probably building features at some point.

In advertising systems, you can build features that capture users' past interactions with various types of products. If a user spends all afternoon looking at different laptops, you probably want to show them an ad for a laptop or maybe a case, but an ad for a sweater wouldn't make a lot of sense. That feature about which types of products the user had been looking at would help the machine-learned model figure that out and make the right recommendation.

At a political polling organization, you could build features pertaining to the demographics of different voters. Things like the average income, education, and home property value could be encoded into features about voting districts. Then those features could be used to learn models about which party a given voting district is likely to vote for.

The applications of features are as endless as the applications of machine learning as a technique. They allow you to encode human intelligence about the problem in a way that a model-learning algorithm can use that intelligence. Machine learning systems are not black-box systems that perform magic tricks. You, the system developer, are the one instructing it how to solve the problem, and features are a big part of how you encode that information.

4.7 Reactivities

This chapter covered a lot, but if you're still interested in learning more about features, there's definitely more to explore. Here are some reactivities to take you even deeper into the world of features:

- *Implement two or more feature extractors of your own.* To do this, you'll probably want to choose some sort of base dataset to work with. If you don't have anything meaningful at hand, you can often use text files and then extract features from the text. Spark has some basic text-processing functionality built in, which you may find helpful. Alternatively, random numbers organized into tabular data can work just

www.itbook.store/books/9781617293337

as well for an activity like this. If you do want to use real data, the UCI Machine Learning Repository at https://archive.ics.uci.edu/ml/index.php is one of the best sources of datasets. Whatever data you use, the point is to decide for yourself what might be some interesting transformations to apply to this dataset.

- *Implement feature-selection functionality*. Using the feature extractors you created in the previous reactivity (or some other extractors), define some basis for including or excluding a given feature within the final output. This could include criteria like the following:
  - Proportion of nonzero values.
  - Number of distinct values.
  - Externally defined business rule/policy. The goal is to ensure that the instances produced by your feature-extraction functionality only include the features that you define as valid.
- *Evaluate the reactivity of an existing feature-extraction pipeline*. If you did the previous two exercises, you can evaluate your own implementation. Alternatively, you can examine examples from open source projects like Spark. As you examine the feature-extraction pipeline, ask yourself questions like the following:
  - Can I find the feature-transform function? Is it implemented as a pure function, or does it have some sort of side effects? Can I easily reuse this transform in other feature extractors?
  - How will bad inputs be handled? Will errors be returned to the user?
  - How will the pipeline behave when it has to handle a thousand records? A million? A billion?
  - What can I discern about the feature extractors from the persisted output? Can I determine when the features were extracted? With which feature extractors?
  - How could I use these feature extractors to make a prediction on a new instance of unseen data?

Summary

- Like chicks cracking through eggs and entering the world of real birds, features are our entry points into the process of building intelligence into a machine learning system. Although they haven't always gotten the attention they deserve, features are a large and crucial part of a machine learning system.
- It's easy to begin writing feature-generation functionality. But that doesn't mean your feature-generation pipeline should be implemented with anything less than the same rigor you'd apply to your real-time predictive application. Feature-generation pipelines can and should be awesome applications that live up to all the reactive traits.
- Feature extraction is the process of producing semantically meaningful, derived representations of raw data.

www.itbook.store/books/9781617293337

- Features can be transformed in various ways to make them easier to learn from.
- You can select among all the features you have to make the model-learning process easier and more successful.
- Feature extractors and transformers should be well structured for composition and reuse.
- Feature-generation pipelines should be assembled into a series of immutable transformations (pure functions) that can easily be serialized and reused.
- Features that rely on external resources should be built with resilience in mind.

We're not remotely done with features. In chapter 5, you'll use features in the learning of models. In chapter 6, you'll generate features when you make predictions about unseen data. Beyond that, in part 3 of the book, we'll get into more-advanced aspects of generating and using features.

www.itbook.store/books/9781617293337

MACHINE LEARNING/SOFTWARE ENGINEERING

Machine Learning Systems

Jeff Smith

If you're building machine learning models to be used on a small scale, you don't need this book. But if you're a developer building a production-grade ML application that needs quick response times, reliability, and good user experience, this is the book for you. It collects principles and practices of machine learning systems that are dramatically easier to run and maintain, and that are reliably better for users.

Machine Learning Systems: Designs that scale teaches you to design and implement production-ready ML systems. You'll learn the principles of reactive design as you build pipelines with Spark, create highly scalable services with Akka, and use powerful machine learning libraries like MLib on massive datasets. The examples use the Scala language, but the same ideas and tools work in Java, as well.

What's Inside

- Working with Spark, MLlib, and Akka
- Reactive design patterns
- Monitoring and maintaining a large-scale system
- Futures, actors, and supervision

Readers need intermediate skills in Java or Scala. No prior machine learning experience is assumed.

Jeff Smith builds large-scale machine learning systems using Scala, Akka, and Spark.

"This book doesn't just cover tools; it covers the whole job of building an entire machine learning system."

—From the Foreword by Sean Owen
Director of Data Science, Cloudera

"A helpful guide for data engineers building resilient machine learning systems."

—Jonathan Woodard, AT&T

"A fantastic entry to the world of robust machine learning systems that will scale with your business."

—Tommy O'Dell
Virtual Gaming Worlds

"You cannot afford to ignore this book!"

—José San Leandro, OSOCO

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/machine-learning-systems

ISBN-13: 978-1-61729-333-7
ISBN-10: 1-61729-333-4

$44.99 / Can $59.99 [INCLUDING eBOOK]

www.itbook.store/books/9781617293337