CHAPTER 5

Anomaly Detection in Network Traffic with K-means Clustering

Sean Owen

> *There are known knowns; there are things that we know that we know. We also know there are known unknowns; that is to say, we know there are some things we do not know. But there are also unknown unknowns, the ones we don't know we don't know.*
>
> —Donald Rumsfeld

Classification and regression are powerful, well-studied techniques in machine learning. Chapter 4 demonstrated using a classifier as a predictor of unknown values. But there was a catch: in order to predict unknown values for new data, we had to know the target values for many previously seen examples. Classifiers can only help if we, the data scientists, know what we are looking for and can provide plenty of examples where input produced a known output. These were collectively known as *supervised learning* techniques, because their learning process receives the correct output value for each example in the input.

However, sometimes the correct output is unknown for some or all examples. Consider the problem of dividing up an ecommerce site's customers by their shopping habits and tastes. The input features are their purchases, clicks, demographic information, and more. The output should be groupings of customers: perhaps one group will represent fashion-conscious buyers, another will turn out to correspond to price-sensitive bargain hunters, and so on.

If you were asked to determine this target label for each new customer, you would quickly run into a problem in applying a supervised learning technique like a classifier: you don't know *a priori* who should be considered fashion-conscious, for example. In fact, you're not even sure if "fashion-conscious" is a meaningful grouping of the site's customers to begin with!

Fortunately, *unsupervised learning* techniques can help. These techniques do not learn to predict a target value, because none is available. They can, however, learn structure in data and find groupings of similar inputs, or learn what types of input are likely to occur and what types are not. This chapter will introduce unsupervised learning using clustering implementations in MLlib.

Anomaly Detection

The inherent problem of anomaly detection is, as its name implies, that of finding unusual things. If we already knew what "anomalous" meant for a data set, we could easily detect anomalies in the data with supervised learning. An algorithm would receive inputs labeled "normal" and "anomaly", and learn to distinguish the two. However, the nature of anomalies is that they are unknown unknowns. Put another way, an anomaly that has been observed and understood is no longer an anomaly.

Anomaly detection is often used to find fraud, detect network attacks, or discover problems in servers or other sensor-equipped machinery. In these cases, it's important to be able to find new types of anomalies that have never been seen before—new forms of fraud, intrusions, and failure modes for servers.

Unsupervised learning techniques are useful in these cases because they can learn what input data normally looks like, and therefore detect when new data is unlike past data. Such new data is not necessarily attacks or fraud; it is simply unusual, and therefore, worth further investigation.

K-means Clustering

Clustering is the best-known type of unsupervised learning. Clustering algorithms try to find natural groupings in data. Data points that are like one another but unlike others are likely to represent a meaningful grouping, so clustering algorithms try to put such data into the same cluster.

K-means clustering may be the most widely used clustering algorithm. It attempts to detect k clusters in a data set, where k is given by the data scientist. k is a hyperparameter of the model, and the right value will depend on the data set. In fact, choosing a good value for k will be a central plot point in this chapter.

What does "like" mean when the data set contains information like customer activity? Or transactions? K-means requires a notion of distance between data points. It is common to use simple Euclidean distance to measure distance between data points with K-means, and as it happens, this is the only distance function supported by Spark MLlib as of this writing. The Euclidean distance is defined for data points whose features are all numeric. "Like" points are those whose intervening distance is small.

To K-means, a cluster is simply a point: the center of all the points that make up the cluster. These are, in fact, just feature vectors containing all numeric features, and can be called vectors. However, it may be more intuitive to think of them as points here, because they are treated as points in a Euclidean space.

This center is called the cluster *centroid*, and is the arithmetic mean of the points—hence the name K-*means*. To start, the algorithm picks some data points as the initial cluster centroids. Then each data point is assigned to the nearest centroid. Then for each cluster, a new cluster centroid is computed as the mean of the data points just assigned to that cluster. This process is repeated.

Enough about K-means for now. Some more interesting details will emerge in the use case to follow.

Network Intrusion

So-called cyberattacks are increasingly visible in the news. Some attacks attempt to flood a computer with network traffic to crowd out legitimate traffic. But in other cases, attacks attempt to exploit flaws in networking software to gain unauthorized access to a computer. While it's quite obvious when a computer is being bombarded with traffic, detecting an exploit can be like searching for a needle in an incredibly large haystack of network requests.

Some exploit behaviors follow known patterns. For example, accessing every port on a machine in rapid succession is not something any normal software program should ever need to do. However, it is a typical first step for an attacker looking for services running on the computer that may be exploitable.

If you were to count the number of distinct ports accessed by a remote host in a short time, you would have a feature that probably predicts a port-scanning attack quite well. A handful is probably normal; hundreds indicates an attack. The same goes for detecting other types of attacks from other features of network connections—number of bytes sent and received, TCP errors, and so forth.

But what about those unknown unknowns? The biggest threat may be the one that has never yet been detected and classified. Part of detecting potential network intrusions is detecting anomalies. These are connections that aren't known to be attacks but do not resemble connections that have been observed in the past.

Here, unsupervised learning techniques like K-means can be used to detect anomalous network connections. K-means can cluster connections based on statistics about each of them. The resulting clusters themselves aren't interesting per se, but they collectively define types of connections that are like past connections. Anything not close to a cluster could be anomalous. Clusters are interesting insofar as they define

regions of normal connections; everything else outside is unusual and potentially anomalous.

KDD Cup 1999 Data Set

The KDD Cup was an annual data mining competition organized by a special interest group of the Assocation for Computing Machinery (ACM). Each year, a machine learning problem was posed, along with a data set, and researchers were invited to submit a paper detailing their best solution to the problem. It was like Kaggle before there was Kaggle. In 1999, the topic was network intrusion, and the data set is still available. The remainder of this chapter will walk through building a system to detect anomalous network traffic using Spark, by learning from this data.

Don't use this data set to build a real network intrusion system! The data did not necessarily reflect real network traffic at the time—even if it did, it reflects traffic patterns from 17 years ago.

Fortunately, the organizers had already processed raw network packet data into summary information about individual network connections. The data set is about 708 MB in size and contains about 4.9 million connections. This is large, if not massive, and is certainly sufficient for our purposes here. For each connection, the data set contains information like the number of bytes sent, login attempts, TCP errors, and so on. Each connection is one line of CSV-formatted data set, containing 38 features, like this:

```
0,tcp,http,SF,215,45076,
0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,1,
0.00,0.00,0.00,0.00,1.00,0.00,0.00,0,0,0.00,
0.00,0.00,0.00,0.00,0.00,0.00,0.00,normal.
```

This connection, for example, was a TCP connection to an HTTP service—215 bytes were sent and 45,706 bytes were received. The user was logged in, and so on. Many features are counts, like `num_file_creations` in the 17th column.

Many features take on the value 0 or 1, indicating the presence or absence of a behavior, like `su_attempted` in the 15th column. They look like the one-hot encoded categorical features from Chapter 4, but are not grouped and related in the same way. Each is like a yes/no feature, and is therefore arguably a categorical feature. It is not always valid to translate categorical features as numbers and treat them as if they had an ordering. However, in the special case of a binary categorical feature, in most machine learning algorithms, mapping these to a numeric feature taking on values 0 and 1 will work well.

The rest are ratios like `dst_host_srv_rerror_rate` in the next-to-last column, and take on values from 0.0 to 1.0, inclusive.

Interestingly, a label is given in the last field. Most connections are labeled `normal.`, but some have been identified as examples of various types of network attacks. These would be useful in learning to distinguish a known attack from a normal connection, but the problem here is anomaly detection and finding potentially new and unknown attacks. This label will be mostly set aside for our purposes.

A First Take on Clustering

Unzip the *kddcup.data.gz* data file and copy it into HDFS. This example, like others, will assume the file is available at */user/ds/kddcup.data*. Open the `spark-shell`, and load the CSV data as a data frame. It's a CSV file again, but without header information. It's necessary to supply column names as given in the accompanying *kddcup.names* file.

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

val data = dataWithoutHeader.toDF(
  "duration", "protocol_type", "service", "flag",
  "src_bytes", "dst_bytes", "land", "wrong_fragment", "urgent",
  "hot", "num_failed_logins", "logged_in", "num_compromised",
  "root_shell", "su_attempted", "num_root", "num_file_creations",
  "num_shells", "num_access_files", "num_outbound_cmds",
  "is_host_login", "is_guest_login", "count", "srv_count",
  "serror_rate", "srv_serror_rate", "rerror_rate", "srv_rerror_rate",
  "same_srv_rate", "diff_srv_rate", "srv_diff_host_rate",
  "dst_host_count", "dst_host_srv_count",
  "dst_host_same_srv_rate", "dst_host_diff_srv_rate",
  "dst_host_same_src_port_rate", "dst_host_srv_diff_host_rate",
  "dst_host_serror_rate", "dst_host_srv_serror_rate",
  "dst_host_rerror_rate", "dst_host_srv_rerror_rate",
  "label")
```

Begin by exploring the data set. What labels are present in the data, and how many are there of each? The following code simply counts by label and prints the results in descending order by count.

```
data.select("label").groupBy("label").count().orderBy($"count".desc).show(25)
```

```
...
+----------------+-------+
|           label|  count|
+----------------+-------+
|          smurf.|2807886|
|        neptune.|1072017|
```

```
|          normal.| 972781|
|           satan.|  15892|
...
|             phf.|      4|
|            perl.|      3|
|             spy.|      2|
+----------------+-------+
```

There are 23 distinct labels, and the most frequent are `smurf.` and `neptune.` attacks.

Note that the data contains nonnumeric features. For example, the second column may be `tcp`, `udp`, or `icmp`, but K-means clustering requires numeric features. The final label column is also nonnumeric. To begin, these will simply be ignored.

Aside from this, creating a K-means clustering of the data follows the same pattern as was seen in Chapter 4. A `VectorAssembler` creates a feature vector, a `KMeans` implementation creates a model from the feature vectors, and a `Pipeline` stitches it all together. From the resulting model, it's possible to extract and examine the cluster centers.

```
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.clustering.{KMeans, KMeansModel}
import org.apache.spark.ml.feature.VectorAssembler

val numericOnly = data.drop("protocol_type", "service", "flag").cache()

val assembler = new VectorAssembler().
  setInputCols(numericOnly.columns.filter(_ != "label")).
  setOutputCol("featureVector")

val kmeans = new KMeans().
  setPredictionCol("cluster").
  setFeaturesCol("featureVector")

val pipeline = new Pipeline().setStages(Array(assembler, kmeans))
val pipelineModel = pipeline.fit(numericOnly)
val kmeansModel = pipelineModel.stages.last.asInstanceOf[KMeansModel]

kmeansModel.clusterCenters.foreach(println)

...
[48.34019491959669,1834.6215497618625,826.2031900016945,793027561892E-4,...
[10999.0,0.0,1.309937401E9,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,...
```

It's not easy to interpret the numbers intuitively, but each of these represents the center (also known as centroid) of one of the clusters that the model produced. The values are the coordinates of the centroid in terms of each of the numeric input features.

Two vectors are printed, meaning K-means was fitting k=2 clusters to the data. For a complex data set that is known to exhibit at least 23 distinct types of connections, this

is almost certainly not enough to accurately model the distinct groupings within the data.

This is a good opportunity to use the given labels to get an intuitive sense of what went into these two clusters by counting the labels within each cluster.

```
val withCluster = pipelineModel.transform(numericOnly)

withCluster.select("cluster", "label").
  groupBy("cluster", "label").count().
  orderBy($"cluster", $"count".desc).
  show(25)
```

```
...
+-------+----------------+-------+
|cluster|           label|  count|
+-------+----------------+-------+
|      0|          smurf.|2807886|
|      0|        neptune.|1072017|
|      0|         normal.| 972781|
|      0|          satan.|  15892|
|      0|        ipsweep.|  12481|
...
|      0|            phf.|      4|
|      0|           perl.|      3|
|      0|            spy.|      2|
|      1|      portsweep.|      1|
+-------+----------------+-------+
```

The result shows that the clustering was not at all helpful. Only one data point ended up in cluster 1!

Choosing k

Two clusters are plainly insufficient. How many clusters are appropriate for this data set? It's clear that there are 23 distinct patterns in the data, so it seems that *k* could be at least 23, or likely even more. Typically, many values of *k* are tried to find the best one. But what is "best"?

A clustering could be considered good if each data point were near its closest centroid, where "near" is defined by the Euclidean distance. This is a simple, common way to evaluate the quality of a clustering, by the mean of these distances over all points, or sometimes, the mean of the distances squared. In fact, `KMeansModel` offers a `computeCost` method that computes the sum of squared distances and can easily be used to compute the mean squared distance.

Unfortunately, there is no simple `Evaluator` implementation to compute this measure, not like those available to compute multiclass classification metrics. It's simple

enough to manually evaluate the clustering cost for several values of *k*. Note that this code could take 10 minutes or more to run.

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

def clusteringScore0(data: DataFrame, k: Int): Double = {
  val assembler = new VectorAssembler().
    setInputCols(data.columns.filter(_ != "label")).
    setOutputCol("featureVector")
  val kmeans = new KMeans().
    setSeed(Random.nextLong()).
    setK(k).
    setPredictionCol("cluster").
    setFeaturesCol("featureVector")
  val pipeline = new Pipeline().setStages(Array(assembler, kmeans))
  val kmeansModel = pipeline.fit(data).stages.last.asInstanceOf[KMeansModel]
  kmeansModel.computeCost(assembler.transform(data)) / data.count() ❶
}

(20 to 100 by 20).map(k => (k, clusteringScore0(numericOnly, k))).
  foreach(println)

...
(20,6.649218115128446E7)
(40,2.5031424366033625E7)
(60,1.027261913057096E7)
(80,1.2514131711109027E7)
(100,7235531.565096531)
```

❶ Compute mean from total squared distance ("cost")

The `(x to y by z)` syntax is a Scala idiom for creating a collection of numbers between a start and end (inclusive), with a given difference between successive elements. This is a compact way to create the values "20, 40, …, 100" for *k*, and then do something with each.

The printed result shows that the score decreases as *k* increases. Note that scores are shown in scientific notation; the first value is over 10^7, not just a bit over 6.

Again, your values will be somewhat different. The clustering depends on a randomly chosen initial set of centroids.

However, this much is obvious. As more clusters are added, it should always be possible to put data points closer to the nearest centroid. In fact, if *k* is chosen to equal the number of data points, the average distance will be 0 because every point will be its own cluster of one!

Worse, in the preceding results, the distance for k=80 is higher than for k=60. This shouldn't happen because higher k always permits at least as good a clustering as a lower k. The problem is that K-means is not necessarily able to find the optimal clustering for a given k. Its iterative process can converge from a random starting point to a local minimum, which may be good but is not optimal.

This is still true even when more intelligent methods are used to choose initial centroids. K-means++ and K-means|| are variants of selection algorithms that are more likely to choose diverse, separated centroids and lead more reliably to a good clustering. Spark MLlib, in fact, implements K-means||. However, all still have an element of randomness in selection and can't guarantee an optimal clustering.

The random starting set of clusters chosen for k=80 perhaps led to a particularly suboptimal clustering, or it may have stopped early before it reached its local optimum.

We can improve it by running the iteration longer. The algorithm has a threshold via `setTol()` that controls the minimum amount of cluster centroid movement considered significant; lower values mean the K-means algorithm will let the centroids continue to move longer. Increasing the maximum number of iterations with `setMaxIter()` also prevents it from potentially stopping too early at the cost of possibly more computation.

```
def clusteringScore1(data: DataFrame, k: Int): Double = {
  ...
    setMaxIter(40). ❶
    setTol(1.0e-5) ❷
  ...
}

(20 to 100 by 20).map(k => (k, clusteringScore1(numericOnly, k))).
  foreach(println)
```

❶ Increase from default 20

❷ Decrease from default 1.0e-4

This time, at least the scores decrease consistently:

```
(20,1.8041795813813403E8)
(40,6.33056876207124E7)
(60,9474961.544965891)
(80,9388117.93747141)
(100,8783628.926311461)
```

We want to find a point past which increasing k stops reducing the score much—or an "elbow" in a graph of k versus score, which is generally decreasing but eventually flattens out. Here, it seems to be decreasing notably past 100. The right value of k may be past 100.

Visualization with SparkR

At this point, it could be useful to step back and understand more about the data before clustering again. In particular, looking at a plot of the data points could be helpful.

Spark itself has no tools for visualization, but the popular open source statistical environment R has libraries for both data exploration and data visualization. Furthermore, Spark also provides some basic integration with R via SparkR. This brief section will demonstrate using R and SparkR to cluster the data and explore the clustering.

SparkR is a variant of the `spark-shell` used throughout this book, and is invoked with the command `sparkR`. It runs a local R interpreter, like `spark-shell` runs a variant of the Scala shell as a local process. The machine that runs `sparkR` needs a local installation of R, which is not included with Spark. This can be installed, for example, with `sudo apt-get install r-base` on Linux distributions like Ubuntu, or `brew install R` with Homebrew on macOS.

sparkR is a command-line shell environment, like R. To view visualizations, it's necessary to run these commands within an IDE-like environment that can display images. RStudio is an IDE for R (and works with SparkR); it runs on a desktop operating system so it will only be usable here if you are experimenting with Spark locally rather than on a cluster.

If you are running Spark locally, download the free version of RStudio and install it. If not, then most of the rest of this example can still be run with sparkR on a command line; for example, on a cluster. It won't be possible to display visualizations this way though.

If running via RStudio, launch the IDE and configure `SPARK_HOME` and `JAVA_HOME`, if your local environment does not already set them, to point to the Spark and JDK installation directories, respectively.

```
Sys.setenv(SPARK_HOME = "/path/to/spark") ❶
Sys.setenv(JAVA_HOME = "/path/to/java")
library(SparkR, lib.loc = c(file.path(Sys.getenv("SPARK_HOME"), "R", "lib")))
sparkR.session(master = "local[*]",
  sparkConfig = list(spark.driver.memory = "4g"))
```

❶ Replace with actual paths, of course.

Note that these steps aren't needed if you are running `sparkR` on the command line. Instead, it accepts command-line configuration parameters like `--driver-memory`, just like `spark-shell`.

SparkR is an R-language wrapper around the same DataFrame and MLlib APIs that have been demonstrated in this chapter. It's therefore possible to recreate a K-means simple clustering of the data:

```
clusters_data <- read.df("/path/to/kddcup.data", "csv", ❶
                         inferSchema = "true", header = "false")
colnames(clusters_data) <- c( ❷
  "duration", "protocol_type", "service", "flag",
  "src_bytes", "dst_bytes", "land", "wrong_fragment", "urgent",
  "hot", "num_failed_logins", "logged_in", "num_compromised",
  "root_shell", "su_attempted", "num_root", "num_file_creations",
  "num_shells", "num_access_files", "num_outbound_cmds",
  "is_host_login", "is_guest_login", "count", "srv_count",
  "serror_rate", "srv_serror_rate", "rerror_rate", "srv_rerror_rate",
  "same_srv_rate", "diff_srv_rate", "srv_diff_host_rate",
  "dst_host_count", "dst_host_srv_count",
  "dst_host_same_srv_rate", "dst_host_diff_srv_rate",
  "dst_host_same_src_port_rate", "dst_host_srv_diff_host_rate",
  "dst_host_serror_rate", "dst_host_srv_serror_rate",
  "dst_host_rerror_rate", "dst_host_srv_rerror_rate",
  "label")

numeric_only <- cache(drop(clusters_data, ❸
                           c("protocol_type", "service", "flag", "label")))

kmeans_model <- spark.kmeans(numeric_only, ~ ., ❹
                             k = 100, maxIter = 40, initMode = "k-means||")
```

❶ Replace with path to *kddcup.data*.

❷ Name columns.

❸ Drop nonnumeric columns again.

❹ `~ .` means all columns.

From here, it's straightforward to assign a cluster to each data point. The operations above show usage of the SparkR APIs, which naturally correspond to core Spark APIs but are expressed as R libraries in R-like syntax. The actual clustering is executed using the same JVM-based, Scala-language implementation in MLlib. These operations are effectively a *handle* or remote control to distributed operations that are not executing in R.

R has its own rich set of libraries for analysis, and its own similar concept of a data frame. It is sometimes useful, therefore, to pull some data down into the R interpreter in order to be able to use these native R libraries, which are unrelated to Spark.

Of course, R and its libraries are not distributed, and so it's not feasible to pull the whole data set of 4,898,431 data points into R. However, it's easy to pull only a sample:

```
clustering <- predict(kmeans_model, numeric_only)
clustering_sample <- collect(sample(clustering, FALSE, 0.01)) ❶

str(clustering_sample)

...
'data.frame': 48984 obs. of  39 variables:
 $ duration                   : int  0 0 0 0 0 0 0 0 0 0 ...
 $ src_bytes                  : int  181 185 162 254 282 310 212 214 181 ...
 $ dst_bytes                  : int  5450 9020 4528 849 424 1981 2917 3404 ...
 $ land                       : int  0 0 0 0 0 0 0 0 0 0 ...
...
 $ prediction                 : int  33 33 33 0 0 0 0 0 33 33 ...
```

❶ 1% sample without replacement

`clustering_sample` is actually a local R data frame, not a Spark DataFrame, so it can be manipulated like any other data in R. Above, `str()` shows the structure of the data frame.

For example, it's possible to extract the cluster assignment and then show statistics about the distribution of assignments:

```
clusters <- clustering_sample["prediction"] ❶
data <- data.matrix(within(clustering_sample, rm("prediction"))) ❷

table(clusters)

...
clusters
    0    11    14    18    23    25    28    30    31    33    36    ...
47294     3     1     2     2   308   105     1    27  1219    15    ...
```

❶ Only the clustering assignment column

❷ Everything but the clustering assignment

For example, this shows that most points fell into cluster 0. Although much more could be done with this data in R, further coverage of this is beyond the scope of this book.

To visualize the data, a library called `rgl` is required. It will only be functional if running this example in RStudio. First, install (once) and load the library:

```
install.packages("rgl")
library(rgl)
```

Note that R may prompt you to download other packages or compiler tools to complete installation, because installing the package means compiling its source code.

This data set is 38-dimensional. It will have to be projected down into at most three dimensions in order to visualize it with a *random projection*:

```
random_projection <- matrix(data = rnorm(3*ncol(data)), ncol = 3) ❶
random_projection_norm <-
  random_projection / sqrt(rowSums(random_projection*random_projection))

projected_data <- data.frame(data %*% random_projection_norm) ❷
```

❶ Make a random 3D projection and normalize

❷ Project and make a new data frame

This creates a 3D data set out of a 38D data set by choosing three random unit vectors and projecting the data onto them. This is a simplistic, rough-and-ready form of dimension reduction. Of course, there are more sophisticated dimension reduction algorithms, like principal component analysis (PCA) or the singular value decomposition (SVD). These are available in R but take much longer to run. For purposes of visualization in this example, a random projection achieves much the same result, faster.

Finally, the clustered points can be plotted in an interactive 3D visualization:

```
num_clusters <- max(clusters)
palette <- rainbow(num_clusters)
colors = sapply(clusters, function(c) palette[c])
plot3d(projected_data, col = colors, size = 10)
```

Note that this will require running RStudio in an environment that supports the `rgl` library and graphics. For example, on macOS, it requires that X11 from Apple's Developer Tools be installed.

The resulting visualization in Figure 5-1 shows data points in 3D space. Many points fall on top of one another, and the result is sparse and hard to interpret. However, the dominant feature of the visualization is its L shape. The points seem to vary along two distinct dimensions, and little in other dimensions.

This makes sense because the data set has two features that are on a much larger scale than the others. Whereas most features have values between 0 and 1, the bytes-sent and bytes-received features vary from 0 to tens of thousands. The Euclidean distance between points is therefore almost completely determined by these two features. It's almost as if the other features don't exist! So it's important to normalize away these differences in scale to put features on near-equal footing.

Feature Normalization

We can normalize each feature by converting it to a standard score. This means subtracting the mean of the feature's values from each value, and dividing by the standard deviation, as shown in the standard score equation:

$$normalized_i = \frac{feature_i - \mu_i}{\sigma_i}$$

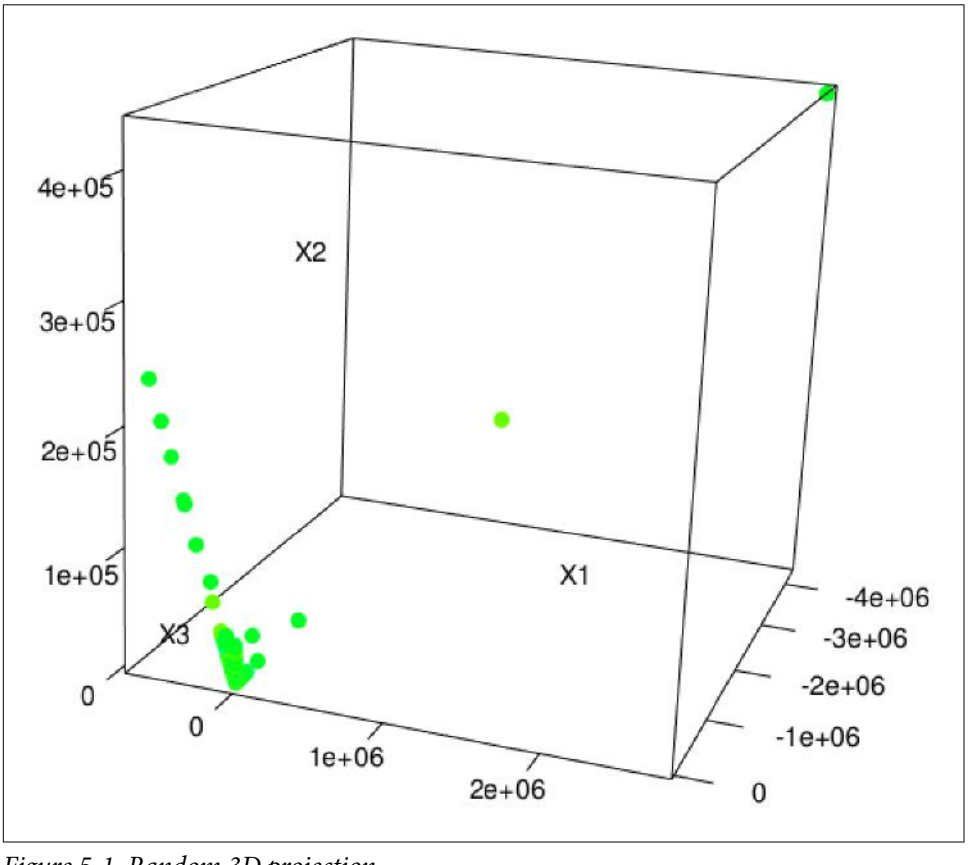

Figure 5-1. Random 3D projection

In fact, subtracting means has no effect on the clustering because the subtraction effectively shifts all the data points by the same amount in the same directions. This does not affect interpoint Euclidean distances.

MLlib provides `StandardScaler`, a component that can perform this kind of standardization and be easily added to the clustering pipeline.

We can run the same test with normalized data on a higher range of *k*:

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

def clusteringScore2(data: DataFrame, k: Int): Double = {
  val assembler = new VectorAssembler().
    setInputCols(data.columns.filter(_ != "label")).
    setOutputCol("featureVector")
  val scaler = new StandardScaler()
    .setInputCol("featureVector")
    .setOutputCol("scaledFeatureVector")
    .setWithStd(true)
    .setWithMean(false)
  val kmeans = new KMeans().
    setSeed(Random.nextLong()).
    setK(k).
    setPredictionCol("cluster").
    setFeaturesCol("scaledFeatureVector").
    setMaxIter(40).
    setTol(1.0e-5)
  val pipeline = new Pipeline().setStages(Array(assembler, scaler, kmeans))
  val pipelineModel = pipeline.fit(data)
  val kmeansModel = pipelineModel.stages.last.asInstanceOf[KMeansModel]
  kmeansModel.computeCost(pipelineModel.transform(data)) / data.count()
}

(60 to 270 by 30).map(k => (k, clusteringScore2(numericOnly, k))).
  foreach(println)
```

This has helped put dimensions on more equal footing, and the absolute distances between points (and thus the cost) is much smaller in absolute terms. However, there isn't yet an obvious value of *k* beyond which increasing it does little to improve the cost:

```
(60,1.2454250178069293)
(90,0.7767730051608682)
(120,0.5070473497003614)
(150,0.4077081720067704)
(180,0.3344486714980788)
(210,0.276237617334138)
(240,0.24571877339169032)
(270,0.21818167354866858)
```

Another 3D visualization of the normalized data points reveals a richer structure, as expected. Some points are spaced in regular, discrete intervals in a direction; these are likely projections of discrete dimensions in the data, like counts. With 100 clusters, it's hard to make out which points come from which clusters. One large cluster seems to dominate, and many clusters correspond to small compact subregions (some of which are omitted from this zoomed detail of the entire 3D visualization). The result, shown in Figure 5-2, does not necessarily advance the analysis but is an interesting sanity check.

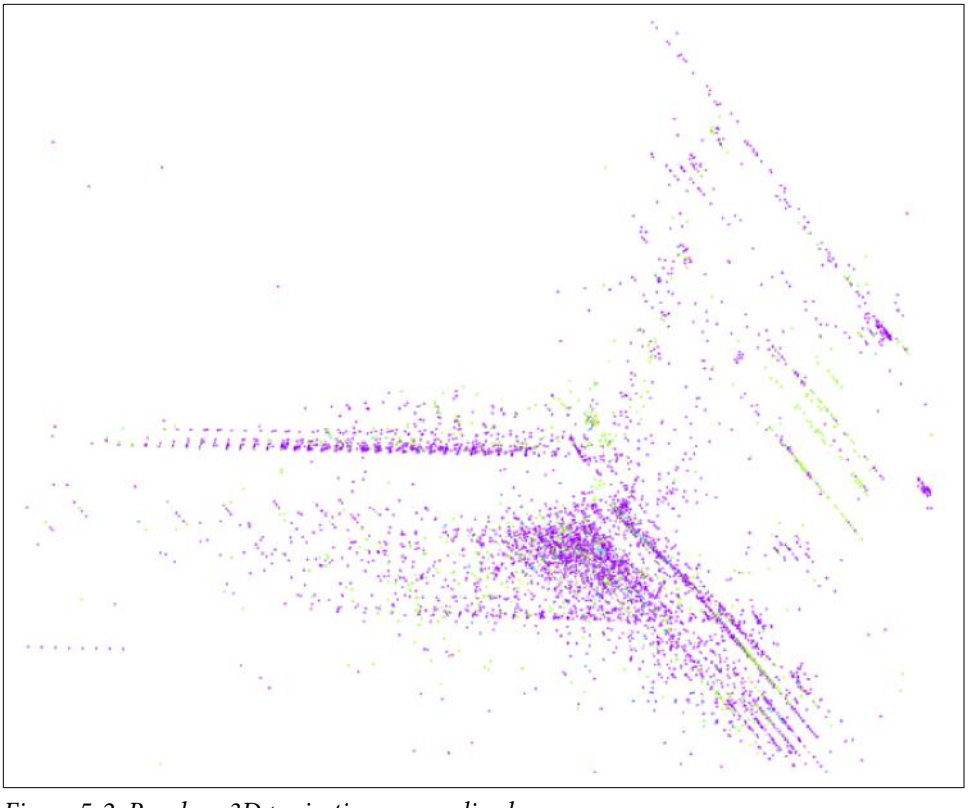

Figure 5-2. Random 3D projection, normalized

Categorical Variables

Normalization was a valuable step forward, but more can be done to improve the clustering. In particular, several features have been left out entirely because they aren't numeric. This is throwing away valuable information. Adding them back in some form should produce a better-informed clustering.

Earlier, three categorical features were excluded because nonnumeric features can't be used with the Euclidean distance function that K-means uses in MLlib. This is the reverse of the issue noted in Chapter 4, where numeric features were used to represent categorical values but a categorical feature was desired.

The categorical features can translate into several binary indicator features using one-hot encoding, which can be viewed as numeric dimensions. For example, the second column contains the protocol type: `tcp`, `udp`, or `icmp`. This feature could be thought of as *three* features, as if features "is TCP," "is UDP," and "is ICMP" were in the data set. The single feature value `tcp` might become `1,0,0`; `udp` might be `0,1,0`; and so on.

Here again, MLlib provides components that implement this transformation. In fact, one-hot-encoding string-valued features like `protocol_type` is actually a two-step process. First, the string values are converted to integer indices like 0, 1, 2, and so on using `StringIndexer`. Then these integer indices are encoded into a vector with `OneHotEncoder`. These two steps can be thought of as a small `Pipeline` in themselves.

```
import org.apache.spark.ml.feature.{OneHotEncoder, StringIndexer}

def oneHotPipeline(inputCol: String): (Pipeline, String) = {
  val indexer = new StringIndexer().
    setInputCol(inputCol).
    setOutputCol(inputCol + "_indexed")
  val encoder = new OneHotEncoder().
    setInputCol(inputCol + "_indexed").
    setOutputCol(inputCol + "_vec")
  val pipeline = new Pipeline().setStages(Array(indexer, encoder))
  (pipeline, inputCol + "_vec") ❶
}
```

❶ Return Pipeline and name of output vector column

This method produces a `Pipeline` that can be added as a component in the overall clustering pipeline; pipelines can be composed. All that is left is to make sure to add the new vector output columns into `VectorAssembler`'s output and proceed as before with scaling, clustering, and evaluation. The source code is omitted for brevity here, but can be found in the repository accompanying this chapter.

```
(60,39.739250062068685)
(90,15.814341529964691)
(120,3.5008631362395413)
(150,2.2151974068685547)
(180,1.587330730808905)
(210,1.3626704802348888)
(240,1.1202477806210747)
(270,0.9263659836264369)
```

These sample results suggest, possibly, k=180 as a value where the score flattens out a bit. At least the clustering is now using all input features.

Using Labels with Entropy

Earlier, we used the given label for each data point to create a quick sanity check of the quality of the clustering. This notion can be formalized further and used as an alternative means of evaluating clustering quality, and therefore, of choosing k.

The labels tell us something about the true nature of each data point. A good clustering, it seems, should agree with these human-applied labels. It should put together points that share a label frequently and not lump together points of many different labels. It should produce clusters with relatively homogeneous labels.

You may recall from Chapter 4 that we have metrics for homogeneity: Gini impurity and entropy. These are functions of the proportions of labels in each cluster, and produce a number that is low when the proportions are skewed toward few, or one, label. Entropy will be used here for illustration.

```
def entropy(counts: Iterable[Int]): Double = {
  val values = counts.filter(_ > 0)
  val n = values.map(_.toDouble).sum
  values.map { v =>
    val p = v / n
    -p * math.log(p)
  }.sum
}
```

A good clustering would have clusters whose collections of labels are homogeneous and so have low entropy. A weighted average of entropy can therefore be used as a cluster score:

```
val clusterLabel = pipelineModel.transform(data).
  select("cluster", "label").as[(Int, String)] ❶

val weightedClusterEntropy = clusterLabel.
  groupByKey { case (cluster, _) => cluster }. ❷
  mapGroups { case (_, clusterLabels) =>
    val labels = clusterLabels.map { case (_, label) => label }.toSeq
    val labelCounts = labels.groupBy(identity).values.map(_.size) ❸
    labels.size * entropy(labelCounts)
  }.collect()

weightedClusterEntropy.sum / data.count() ❹
```

❶ Predict cluster for each datum

❷ Extract collections of labels, per cluster

❸ Count labels in collections

❹ Average entropy weighted by cluster size

As before, this analysis can be used to obtain some idea of a suitable value of k. Entropy will not necessarily decrease as k increases, so it is possible to look for a local minimum value. Here again, results suggest k=180 is a reasonable choice because its score is actually lower than 150 *and* 210:

```
(60,0.03475331900669869)
(90,0.051512668026335535)
(120,0.02020028911919293)
(150,0.019962563512905682)
(180,0.01110240886325257)
(210,0.01259738444250231)
```

```
(240,0.01357435960663116)
(270,0.010119881917660544)
```

Clustering in Action

Finally, with confidence, we can cluster the full normalized data set with k=180. Again, we can print the labels for each cluster to get some sense of the resulting clustering. Clusters do seem to be dominated by one type of attack each, and contain only a few types.

```
val pipelineModel = fitPipeline4(data, 180) ❶
val countByClusterLabel = pipelineModel.transform(data).
  select("cluster", "label").
  groupBy("cluster", "label").count().
  orderBy("cluster", "label")
countByClusterLabel.show()

...
+-------+----------+------+
|cluster|     label| count|
+-------+----------+------+
|      0|     back.|   324|
|      0|   normal.| 42921|
|      1|  neptune.|  1039|
|      1|portsweep.|     9|
|      1|    satan.|     2|
|      2|  neptune.|365375|
|      2|portsweep.|   141|
|      3|portsweep.|     2|
|      3|    satan.| 10627|
|      4|  neptune.|  1033|
|      4|portsweep.|     6|
|      4|    satan.|     1|
...
```

❶ See accompanying source code for `fitPipeline4()` definition

Now we can make an actual anomaly detector. Anomaly detection amounts to measuring a new data point's distance to its nearest centroid. If this distance exceeds some threshold, it is anomalous. This threshold might be chosen to be the distance of, say, the 100th-farthest data point from among known data:

```
import org.apache.spark.ml.linalg.{Vector, Vectors}

val kMeansModel = pipelineModel.stages.last.asInstanceOf[KMeansModel]
val centroids = kMeansModel.clusterCenters

val clustered = pipelineModel.transform(data)
val threshold = clustered.
  select("cluster", "scaledFeatureVector").as[(Int, Vector)].
```

```
  map { case (cluster, vec) => Vectors.sqdist(centroids(cluster), vec) }.
  orderBy($"value".desc).take(100).last ❶
```

❶ Single output implicitly named "value"

The final step is to apply this threshold to all new data points as they arrive. For example, Spark Streaming can be used to apply this function to small batches of input data arriving from sources like Flume, Kafka, or files on HDFS. Data points exceeding the threshold might trigger an alert that sends an email or updates a database.

As an example, we will apply it to the original data set, to see some of the data points that are, we might believe, most anomalous within the input.

```
val originalCols = data.columns
val anomalies = clustered.filter { row =>
  val cluster = row.getAs[Int]("cluster")
  val vec = row.getAs[Vector]("scaledFeatureVector")
  Vectors.sqdist(centroids(cluster), vec) >= threshold
}.select(originalCols.head, originalCols.tail:_*) ❶

anomalies.first() ❷

...
[9,tcp,telnet,SF,307,2374,0,0,1,0,0,1,0,1,0,1,3,1,0,0,0,0,1,1,
 0.0,0.0,0.0,0.0,1.0,0.0,0.0,69,4,0.03,0.04,0.01,0.75,0.0,0.0,
 0.0,0.0,normal.]
```

❶ Note odd `(String, String*)` signature for selecting columns

❷ `show()` works; hard to read

This example shows a slightly different way of operating on data frames. Pure SQL can't express the computation of squared distance. A UDF could be used, as before, to define a function of two columns that returns a squared distance. However, it's also possible to interact with rows of data programmatically as a `Row` object, much like in JDBC.

A network security expert would be more able to interpret why this is or is not actually a strange connection. It appears unusual at least because it is labeled `normal`, but involves connections to 69 different hosts.

Where to Go from Here

The `KMeansModel` is, by itself, the essence of an anomaly detection system. The preceding code demonstrated how to apply it to data to detect anomalies. This same code could be used within Spark Streaming to score new data as it arrives in near real time, and perhaps trigger an alert or review.

MLlib also includes a variation called `StreamingKMeans`, which can update a clustering incrementally as new data arrives in a `StreamingKMeansModel`. We could use this to continue to learn, approximately, how new data affects the clustering, and not just to assess new data against existing clusters. It can be integrated with Spark Streaming as well. However, it has not been updated for the new DataFrame-based APIs.

This model is only a simplistic one. For example, Euclidean distance is used in this example because it is the only distance function supported by Spark MLlib at this time. In the future, it may be possible to use distance functions that can better account for the distributions of and correlations between features, such as the Mahalanobis distance.

There are also more sophisticated cluster-quality evaluation metrics that could be applied (even without labels) to pick *k*, such as the Silhouette coefficient. These tend to evaluate not just closeness of points within one cluster, but closeness of points to other clusters. Finally, different models could be applied instead of simple K-means clustering; for example, a Gaussian mixture model or DBSCAN could capture more subtle relationships between data points and the cluster centers. Spark MLlib already implements Gaussian mixture models; implementations of others may become available in Spark MLlib or other Spark-based libraries in the future.

Of course, clustering isn't just for anomaly detection. In fact, it's more often associated with use cases where the actual clusters matter! For example, clustering can also be used to group customers according to their behaviors, preferences, and attributes. Each cluster, by itself, might represent a usefully distinguishable type of customer. This is a more data-driven way to segment customers rather than leaning on arbitrary, generic divisions like "age 20–34" and "female."