Cluster Analysis for Anomaly Detection

Sean O'Malley

Table of Contents:	

Spark and Scala Overview

- Scala for Data Scientist
    - A vast majority of introductory examples should be written in Scala, because we think that learning how to work with Spark in the same language in which the underlying framework is written has a number of advantages
        - Reduces the performance overhead of data translation
        - Gives access to latest more readily than wrapper languages
        - Helps understand the Spark philosophy and "think in Spark"
    - Retrieving data with Spark and Scala is much different, because you're using the same language for everything.
        - You are writing Scala to retrieve data from the cluster via Spark
        - You are writing Scala to manipulate data locally on your own machine, and then you can send Scala code into the cluster so that you can perform the exact same transformations that you performed locally on data that is still stored in the cluster.
        -
- Spark Programming Model
    - Spark programming starts with a data set or few, usually residing in some form of distributed, persistent storage like HDFS.
    - Writing a Spark program usually consists of a few steps:
        - Defining a set of transformations on input data sets
        - Invoking actions that output the transformed data sets to persistent storage or return results to the drivers local memory
        - Running local computations that operate on the results computed in a distributed fashion. These can help you decide what transformations and actions to undertake next.
    - Understanding Spark means understanding the intersection between two sets of abstractions the framework offers: storage and execution.
    - Allows any intermediate step in a data processing pipeline to be cached in memory for later use.

Spark and Scala Installation

Note: I have decided to use a stand alone mode for this entire process within a passwordless hadoop user account on a basic install CentOs using VMWare. This entire assignment is completed via the command line, spark and scala.

```
[hadoop@localhost ~]$ wget http://downloads.lightbend.com/scala/2.12.1/scala-2.12.1.tgz
--2017-02-15 09:23:30--  http://downloads.lightbend.com/scala/2.12.1/scala-2.12.1.tgz
Resolving downloads.lightbend.com (downloads.lightbend.com)... 52.84.125.169, 52.84.125.10, 52.84.125.117, ...
Connecting to downloads.lightbend.com (downloads.lightbend.com)|52.84.125.169|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 19700349 (19M) [application/octet-stream]
Saving to: 'scala-2.12.1.tgz'

100%[======================================>] 19,700,349  2.54MB/s   in 8.3s

2017-02-15 09:23:39 (2.25 MB/s) - 'scala-2.12.1.tgz' saved [19700349/19700349]
```

I then tar'd it, mv'd it and chown'd it

```
[hadoop@localhost ~]$ tar xvf scala-2.12.1.tgz
scala-2.12.1/
scala-2.12.1/lib/
scala-2.12.1/lib/scala-parser-combinators_2.12-1.0.4.jar
scala-2.12.1/lib/scala-swing_2.12-2.0.0-M2.jar
scala-2.12.1/lib/scala-reflect.jar
scala-2.12.1/lib/jline-2.14.1.jar
scala-2.12.1/lib/scala-compiler.jar
scala-2.12.1/lib/scalap-2.12.1.jar
scala-2.12.1/lib/scala-library.jar
scala-2.12.1/lib/scala-xml_2.12-1.0.6.jar
scala-2.12.1/bin/
```

```
[hadoop@localhost ~]$ mv /home/hadoop/scala-2.12.1 /home/hadoop/scala
[hadoop@localhost ~]$ ls -l
total 725048
drwxrwxr-x.  8 hadoop hadoop       159 Feb 15 00:38 apache-hive-1.2.1-bin
-rw-rw-r--.  1 hadoop hadoop  92834839 Jun 26  2015 apache-hive-1.2.1-bin.tar.gz
-rw-rw-r--.  1 hadoop hadoop 234608362 Jun 13  2016 apache-mahout-distribution-0.12.2.tar.gz
-rw-rw-r--.  1 hadoop hadoop   3845425 Nov 18  2015 apache-maven-3.3.9-src.tar.gz
-rw-rw-r--.  1 hadoop hadoop     21055 Feb 15 01:31 derby.log
drwxr-xr-x. 10 hadoop hadoop       161 Feb 15 00:03 hadoop
-rw-rw-r--.  1 hadoop hadoop 214092195 Aug 25 13:25 hadoop-2.7.3.tar.gz
drwxrwxr-x.  3 hadoop hadoop        18 Feb 15 00:00 hadoopdata
drwxrwxr-x. 10 hadoop hadoop      4096 Feb 15 08:59 mahout
drwxr-xr-x. 16 hadoop hadoop      4096 Nov 10  2015 maven
drwxrwxr-x.  5 hadoop hadoop       133 Feb 15 01:31 metastore_db
drwxr-xr-x. 16 hadoop hadoop      4096 Feb 15 05:46 pig
-rw-rw-r--.  1 hadoop hadoop 177279333 Jun  7  2016 pig-0.16.0.tar.gz
-rw-rw-r--.  1 hadoop hadoop     35634 Feb 15 04:23 sample.txt
drwxrwxr-x.  6 hadoop hadoop        50 Dec  5 04:15 scala
-rw-rw-r--.  1 hadoop hadoop  19700349 Dec  5 04:21 scala-2.12.1.tgz
-rw-rw-r--.  1 hadoop hadoop       381 May 12  2014 student.txt
```

```
[hadoop@localhost ~]$ chown -R hadoop:hadoop /home/hadoop/scala
```

Next, I established the SCALA_HOME and the PATH to the scala program.

```
export SCALA_HOME=/home/hadoop/scala
export PATH=$SCALA_HOME/bin:$PATH
```

I returned to my hadoop user home directory location ran a wget to download the Spark tar.gz

```
[hadoop@localhost ~]$ wget http://d3kbcqa49mib13.cloudfront.net/spark-2.1.0-bin-hadoop2.7.tgz
--2017-02-15 09:40:16--  http://d3kbcqa49mib13.cloudfront.net/spark-2.1.0-bin-hadoop2.7.tgz
Resolving d3kbcqa49mib13.cloudfront.net (d3kbcqa49mib13.cloudfront.net)... 52.84.126.208, 52.84.126.151, 52.84.126.30, ...
Connecting to d3kbcqa49mib13.cloudfront.net (d3kbcqa49mib13.cloudfront.net)|52.84.126.208|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 195636829 (187M) [application/x-tar]
Saving to: 'spark-2.1.0-bin-hadoop2.7.tgz'

100%[=======================================================================================>] 195,636,829 2.46MB/s   in 1m 41s

2017-02-15 09:41:57 (1.86 MB/s) - 'spark-2.1.0-bin-hadoop2.7.tgz' saved [195636829/195636829]
```

I then followed the process of un-taring the spark file, mv'ing it, and chown'ing it

```
[hadoop@localhost ~]$ tar xvf spark-2.1.0-bin-hadoop2.7.tgz
spark-2.1.0-bin-hadoop2.7/
spark-2.1.0-bin-hadoop2.7/NOTICE
spark-2.1.0-bin-hadoop2.7/jars/
spark-2.1.0-bin-hadoop2.7/jars/bonecp-0.8.0.RELEASE.jar
spark-2.1.0-bin-hadoop2.7/jars/commons-net-2.2.jar
spark-2.1.0-bin-hadoop2.7/jars/javax.servlet-api-3.1.0.jar
spark-2.1.0-bin-hadoop2.7/jars/hadoop-annotations-2.7.3.jar
spark-2.1.0-bin-hadoop2.7/jars/hadoop-hdfs-2.7.3.jar
spark-2.1.0-bin-hadoop2.7/jars/oro-2.0.8.jar
```

```
[hadoop@localhost ~]$ mv /home/hadoop/spark-2.1.0-bin-hadoop2.7 /home/hadoop/spark
[hadoop@localhost ~]$ ls -l
total 916100
drwxrwxr-x.  8 hadoop hadoop       159 Feb 15 00:38 apache-hive-1.2.1-bin
-rw-rw-r--.  1 hadoop hadoop  92834839 Jun 26  2015 apache-hive-1.2.1-bin.tar.gz
-rw-rw-r--.  1 hadoop hadoop 234608362 Jun 13  2016 apache-mahout-distribution-0.12.2.tar.gz
-rw-rw-r--.  1 hadoop hadoop   3845425 Nov 18  2015 apache-maven-3.3.9-src.tar.gz
-rw-rw-r--.  1 hadoop hadoop     21055 Feb 15 01:31 derby.log
drwxr-xr-x. 10 hadoop hadoop       161 Feb 15 00:03 hadoop
-rw-rw-r--.  1 hadoop hadoop 214092195 Aug 25 13:25 hadoop-2.7.3.tar.gz
drwxrwxr-x.  3 hadoop hadoop        18 Feb 15 00:00 hadoopdata
drwxrwxr-x. 10 hadoop hadoop      4096 Feb 15 08:59 mahout
drwxr-xr-x. 16 hadoop hadoop      4096 Nov 10  2015 maven
drwxrwxr-x.  5 hadoop hadoop       133 Feb 15 01:31 metastore_db
drwxr-xr-x. 16 hadoop hadoop      4096 Feb 15 05:46 pig
-rw-rw-r--.  1 hadoop hadoop 177279333 Jun  7  2016 pig-0.16.0.tar.gz
-rw-rw-r--.  1 hadoop hadoop     35634 Feb 15 04:23 sample.txt
drwxrwxr-x.  6 hadoop hadoop        50 Dec  5 04:15 scala
-rw-rw-r--.  1 hadoop hadoop  19700349 Dec  5 04:21 scala-2.12.1.tgz
drwxr-xr-x. 12 hadoop hadoop       193 Dec 15 19:18 spark
-rw-rw-r--.  1 hadoop hadoop 195636829 Dec 28 17:49 spark-2.1.0-bin-hadoop2.7.tgz
```

```
[hadoop@localhost ~]$ chown -R hadoop:hadoop /home/hadoop/spark
```

Next, I established the SPARK_HOME and PATH to the Spark program, then source'd .bashrc

```
export SPARK_HOME=/home/hadoop/spark
export PATH=$SPARK_HOME/bin:$PATH
```

I then tested my spark shell to make sure I had everything up and running okay.

```
Spark context Web UI available at http://172.16.245.143:4040
Spark context available as 'sc' (master = local[*], app id = local-1487177715520).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.1.0
      /_/

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_111)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

(Business Problem Continues Below)

Business Problem and Methodology

Anomaly Detection and Unsupervised learning

- When a priori knowledge / data is not available, supervised learning is no longer an option and one has to move onto unsupervised learning techniques. These methods can learn structure in data, find groupings of similar output, or learn what types of input are likely to occur and what types are not.
- In considering an anomaly, we cannot use supervised learning because the notion of a dataset existing that has previously labeled something as “normal” and “anomaly” is an impossible one. The very nature of an anomaly is that they are unknown unknowns.
- 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 the past data...flagging certain data as unusual.

K-Means Clustering

- 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, and so clustering algorithms try to put such data into the same cluster.
- K-means clustering attempts to detect ***k*** clusters in a dataset, where ***k*** is given by data scientist. ***K*** is a hyper-parameter 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.
- 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, in fact, this is the only distance function supported by Spark MLib as of now.
    - Euclidean distance is defined for data points whose features are all numeric, “like” distances 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. The points are treated as points in a Euclidean space.
- The center cluster is called a centroid, and is the arithmetic mean of the points.
- To start, the algorithm picks some data points as the initial cluster centroids, then each data point is assigned to the nearest centroid. Next, 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.

Network Intrusion

- Cyber attacks are a much more common occurrence, and often quite obvious when a computer is being bombarded with traffic, but detecting an exploit can be like searching for a needle in a haystack of network requests.
- Some exploit behaviors follow known patterns, such as accessing every port on a machine in a rapid succession, but this still doesn’t account for cyber attacks that are and unknown unknown.
- The biggest threat may be the one that has never yet been detected and classified. Part of the potential network intrusions is detecting anomalies.
- 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 Data

- We will use already processed raw network packet data into summary information and about individual network connections. For each connections, 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 containing 38 features.
- Many of the columns are binary, or related to connections or speed/frequency/ratios.
- Most connections are labeled as normal. But some have been identified as examples of various types of network attacks. These would be useful in learning to distinguish known attack from a normal connections, but the problem here is anomaly detection and finding potentially new and unknown attacks.

Now let's build it!!!

First we must ingest the kddcup.data.gz

```
[hadoop@localhost ~]$ wget http://kdd.ics.uci.edu/databases/kddcup99/kddcup.data_10_per
cent.gz
--2017-02-15 10:58:00--  http://kdd.ics.uci.edu/databases/kddcup99/kddcup.data_10_perce
nt.gz
Resolving kdd.ics.uci.edu (kdd.ics.uci.edu)... 128.195.1.95
Connecting to kdd.ics.uci.edu (kdd.ics.uci.edu)|128.195.1.95|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2144903 (2.0M) [application/x-gzip]
Saving to: 'kddcup.data_10_percent.gz'

100%[=============================================>] 2,144,903    820KB/s   in 2.6s

2017-02-15 10:58:03 (820 KB/s) - 'kddcup.data_10_percent.gz' saved [2144903/2144903]
```

Then I gunzipped it

```
[hadoop@localhost ~]$ gunzip kddcup.data_10_percent.gz
[hadoop@localhost ~]$ ls -l
total 989236
drwxrwxr-x.  8 hadoop hadoop       159 Feb 15 00:38 apache-hive-1.2.1-bin
-rw-rw-r--.  1 hadoop hadoop  92834839 Jun 26  2015 apache-hive-1.2.1-bin.t
-rw-rw-r--.  1 hadoop hadoop 234608362 Jun 13  2016 apache-mahout-distribut
.12.2.tar.gz
-rw-rw-r--.  1 hadoop hadoop   3845425 Nov 18  2015 apache-maven-3.3.9-src.
z
-rw-rw-r--.  1 hadoop hadoop     21055 Feb 15 01:31 derby.log
drwxr-xr-x. 10 hadoop hadoop       161 Feb 15 00:03 hadoop
-rw-rw-r--.  1 hadoop hadoop 214092195 Aug 25 13:25 hadoop-2.7.3.tar.gz
drwxrwxr-x.  3 hadoop hadoop        18 Feb 15 00:00 hadoopdata
-rw-rw-r--.  1 hadoop hadoop  74889749 Jun 26  2007 kddcup.data_10_percent
```

I then started up my spark-shell

```
Spark context available as 'sc' (master = local[*], app id = local-1487182305829
).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.1.0
      /_/

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_111)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

I then loaded the kdd cup data into my Spark shell environment

```
scala> val rawData = sc.textFile("/home/hadoop/kddcup.data_10_percent")
rawData: org.apache.spark.rdd.RDD[String] = /home/hadoop/kddcup.data_10_percent
MapPartitionsRDD[1] at textFile at <console>:24
```

Next, I ran the below function to explore the dataset. This code counts by label into label-count tuples, sorts them descending by count and prints the results. We can see 23 distinct labels and the most frequent are smurf and neptune attacks.

```
scala> rawData.map(_.split(",").last).countByValue().toSeq.sortBy(_._2).reverse.
foreach(println)
(smurf.,280790)
(neptune.,107201)
(normal.,97278)
(back.,2203)
(satan.,1589)
(ipsweep.,1247)
(portsweep.,1040)
(warezclient.,1020)
(teardrop.,979)
(pod.,264)
(nmap.,231)
(guess_passwd.,53)
(buffer_overflow.,30)
(land.,21)
```

Note that the data contains non-numeric 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 non-numeric.

The below spark code:

- Splits the csv into columns
- Removes the three categorical value columns starting from index 1
- Removes the final column
- Remaining columns are converted into an array of numeric values (double objects) and emitted with the final label column in a tuple:

```
scala> import org.apache.spark.mllib.linalg._
import org.apache.spark.mllib.linalg._

scala> val labelsAndData = rawData.map { line =>
     | val buffer = line.split(',').toBuffer
     | buffer.remove(1,3)
     | val label = buffer.remove(buffer.length-1)
     | val vector = Vectors.dense(buffer.map(_.toDouble).toArray)
     | (label, vector)
     | }
labelsAndData: org.apache.spark.rdd.RDD[(String, org.apache.spark.mllib.linalg.V
ector)] = MapPartitionsRDD[6] at map at <console>:29

scala> val data = labelsAndData.values.cache()
data: org.apache.spark.rdd.RDD[org.apache.spark.mllib.linalg.Vector] = MapPartit
ionsRDD[7] at values at <console>:31
```

K-means will operate on just the feature vectors. So, the RDD data contains just the second element of each tuple, which in an RDD of tuples are accessed with **values**.

Clustering the data with **Spark MLib** is as simple as importing the K-Means implementation and running it. The following code clusters the data to create a **KMeansModel**, and then prints its centroids.

```
scala> import org.apache.spark.mllib.clustering._
import org.apache.spark.mllib.clustering._
```

```
scala> val kmeans = new KMeans()
kmeans: org.apache.spark.mllib.clustering.KMeans = org.apache.spark.mllib.cluste
ring.KMeans@5513745d
```

```
scala> val model = kmeans.run(data)
17/02/15 11:44:31 WARN BLAS: Failed to load implementation from: com.github.fomm
il.netlib.NativeSystemBLAS
17/02/15 11:44:31 WARN BLAS: Failed to load implementation from: com.github.fomm
il.netlib.NativeRefBLAS
model: org.apache.spark.mllib.clustering.KMeansModel = org.apache.spark.mllib.cl
ustering.KMeansModel@78fb8a66
```

```
scala> model.clusterCenters.foreach(println)
1[47.979395571029514,1622.078830816566,868.5341828266062,4.453261001578883E-5,0.0
06432937937735314,1.4169466823205539E-5,0.03451682118132869,1.5181571596291647E-
4,0.14824703453301485,0.01021213716043885,1.1133152503947209E-4,3.64357718310999
54E-5,0.011351767134933808,0.0010829521072021374,1.0930731549329986E-4,0.0010080
563539937655,0.0,0.0,0.0,0.0013865835391279706,332.2862475203433,292.9071434354884,0
.1766854175944295,0.1766078094004292,0.05743309987449898,0.05771839196793656,0.7
915488441762849,0.020981640419416685,0.028996862475203982,232.4707319541719,188.
6660459090725,0.7537812031901855,0.030905611108874582,0.6019355289259479,0.00668
35148374550625,0.17675395732965873,0.17644162179668482,0.05811762681672762,0.057
41111695882669]
2[2.0,6.9337564E8,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0
,0.0,57.0,3.0,0.79,0.67,0.21,0.33,0.05,0.39,0.0,255.0,3.0,0.01,0.09,0.22,0.0,0.1
8,0.67,0.05,0.33]
```

As you can see, two vectors were printed, meaning K-means was fitting k = 2 clusters to the data. With 23 distinct types of connections, this model could definitely be tuned for improvement.

Nevertheless, let's do a little EDA of the clusters. The code below uses the model to assign each data point to a cluster, counts occurrences of cluster and label pairs, and prints them:

```
scala> val clusterLabelCount = labelsAndData.map { case (label,datum) =>
     | val cluster = model.predict(datum)
     | (cluster, label)
     | }.countByValue
clusterLabelCount: scala.collection.Map[(Int, String),Long] = Map((0,portsweep.)
 -> 1039, (0,rootkit.) -> 10, (0,buffer_overflow.) -> 30, (0,phf.) -> 4, (0,pod.
) -> 264, (0,perl.) -> 3, (0,spy.) -> 2, (0,ftp_write.) -> 8, (0,nmap.) -> 231,
(0,ipsweep.) -> 1247, (0,imap.) -> 12, (0,warezmaster.) -> 20, (0,satan.) -> 158
9, (0,teardrop.) -> 979, (0,smurf.) -> 280790, (0,neptune.) -> 107201, (0,loadmo
dule.) -> 9, (0,guess_passwd.) -> 53, (0,normal.) -> 97278, (0,land.) -> 21, (0,
multihop.) -> 7, (1,portsweep.) -> 1, (0,warezclient.) -> 1020, (0,back.) -> 220
3)
```

```
scala> clusterLabelCount.toSeq.sorted.foreach {
     | case ((cluster,label),count) =>
     | println(f"$cluster%1s$label%18s$count%8s")
     | }
0              back.    2203
0  buffer_overflow.      30
0         ftp_write.       8
0      guess_passwd.      53
0              imap.      12
0           ipsweep.    1247
0              land.      21
0        loadmodule.       9
0          multihop.       7
0           neptune.  107201
0              nmap.     231
0            normal.   97278
0              perl.       3
0               phf.       4
```

```
0               pod.     264
0         portsweep.    1039
0           rootkit.      10
0             satan.    1589
0             smurf.  280790
0               spy.       2
0          teardrop.     979
0       warezclient.    1020
0       warezmaster.      20
1         portsweep.       1
```

Looking at the results, we could see that only one data point ended up in cluster 1. This obviously won't work, so now it's time for us to determine the number of k-means clusters by establishing k.

A cluster could be considered good if each data point were near to its closest centroid. So, we define a Euclidean distance function, and a function that returns the distance from a data point to its nearest cluster's centroid:

```
scala> def distance(a: Vector, b: Vector) =
     | math.sqrt(a.toArray.zip(b.toArray).
     | map(p => p._1 - p._2).map(d => d*d).sum)
distance: (a: org.apache.spark.mllib.linalg.Vector, b: org.apache.spark.mllib.li
nalg.Vector)Double

scala> def distToCentroid(datum: Vector, model: KMeansModel) = {
     | val cluster = model.predict(datum)
     | val centroid = model.clusterCenters(cluster)
     | distance(centroid, datum)
     | }
distToCentroid: (datum: org.apache.spark.mllib.linalg.Vector, model: org.apache.
spark.mllib.clustering.KMeansModel)Double
```

We define Euclidean distance as:

- square root of the sum square of differences in corresponding elements of two vectors

$$\mathrm{dist}(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

Using this, we can define a function that measures the average distance to a centroid, for a model built with a given k.

```
scala> import org.apache.spark.rdd._
import org.apache.spark.rdd._

scala> def clusteringScore(data: RDD[Vector], k: Int) = {
     | val kmeans = new KMeans()
     | kmeans.setK(k)
     | val model = kmeans.run(data)
     | data.map(datum => distToCentroid(datum, model)).mean()
     | }
clusteringScore: (data: org.apache.spark.rdd.RDD[org.apache.spark.mllib.linalg.V
ector], k: Int)Double
```

Now, we can use this function we just built to evaluate the values of k from (EG: 5 to 40) using the Scala syntax of (x to y by z):

```
scala> (5 to 40 by 5).map(k=>(k,clusteringScore(data,k))).foreach(println)
(5,1883.6502407468217)
(10,1643.7083310576097)
(15,1635.5829493401843)
(20,1228.8175497321229)
(25,1559.2670404967444)
(30,1592.9346951726368)
(35,1031.224294059217)
(40,1591.9465853820595)
```

As more clusters are added, it should always be possible to make data points closer to a 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.

K means is not always necessarily able to find the optimal clustering given k. Its iterative process can converge from a random starting point to a local minimum, which may be good but not optimal….meaning there are more advanced methods out there. Within the **MLib**, there is **K-means++** and **K-means||**, both of which are definitely worth checking out!

We can use the algorithm **setRuns()** to set the number of times the clustering is run for one k, with a threshold set via **setEpsilon()** function. We can improve the iteration by running it longer, but setting the epsilon controls the minimum amount of cluster centroid movement that is considered significant; lower values means the K-means algorithm will let you set the centroids and continue to move longer.

After we set runs and epsilon, we want to run the same test again, but will instead try larger values. This causes the computation to happen in parallel in the Spark shell. Also, the computation of each k is also a distributed operation on the cluster. It is parallelism inside of parallelism.

```
scala> kmeans.setRuns(10)
warning: there was one deprecation warning; re-run with -deprecation for details
17/02/15 12:39:37 WARN KMeans: Setting number of runs has no effect since Spark
2.0.0.
res4: kmeans.type = org.apache.spark.mllib.clustering.KMeans@5513745d

scala> kmeans.setEpsilon(1.0e-6)
res5: kmeans.type = org.apache.spark.mllib.clustering.KMeans@5513745d

scala> (30 to 100 by 10).par.map(k=>(k,clusteringScore(data, k))).toList.foreach
(println)
(30,1310.4129425048523)
(40,1313.8534746070288)
(50,970.611885363189)
(60,1256.8942284338948)
(70,800.5085351242753)
(80,982.6592683984987)
(90,1033.819679625953)
(100,913.4957548117394)
```

Looking at the results, 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.

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 it by the standard deviation.

The standard scores can be computed from the count, sum, and sum-of-squares of each feature. This can be done jointly, with reduce operations used to add entire arrays at once, and fold to accumulate sums of squares from an array of zeros:

```
scala> val dataAsArray = data.map(_.toArray)
dataAsArray: org.apache.spark.rdd.RDD[Array[Double]] = MapPartitionsRDD[789] at
map at <console>:39

scala> val numCols = dataAsArray.first().length
17/02/15 13:08:02 WARN Executor: 1 block locks were not released by TID = 1848:
[rdd_7_0]
numCols: Int = 38

scala> val n = dataAsArray.count()
n: Long = 494021

scala> val sums = dataAsArray.reduce( (a,b) => a.zip(b).map(t => t._1 + t._2))
sums: Array[Double] = Array(2.3702783E7, 1.494715024E9, 4.29073257E8, 22.0, 3178
.0, 7.0, 17053.0, 75.0, 73237.0, 5045.0, 55.0, 18.0, 5608.0, 535.0, 54.0, 498.0,
 0.0, 0.0, 685.0, 1.64156109E8, 1.4470199E8, 87286.92000000006, 87248.4600000000
4, 28373.30999999999, 28514.370000000017, 391041.0099999683, 10365.74000000023,
14325.030000000272, 1.14845446E8, 9.3204803E7, 372383.0000000154, 15268.08000000
6223, 297368.4099999968, 3301.78999999955, 87320.169999998, 87166.35999999823, 2
8711.31999999978, 28362.56999999956)
```

```
scala> val sumSquares = dataAsArray.fold( new Array[Double](numCols))( (a,b) =>
a.zip(b).map(t => t._1 + t._2 * t._2))
sumSquares: Array[Double] = Array(2.415481508223714E22, 2.3272157961788754E35, 2
.6464655573501947E29, 302.0, 3.384665E7, 225.0, 7.8973446405E10, 13925.0, 3.9805
82619E9, 1.259778184821E12, 1725.0, 482.0, 1.971010179246E12, 1.0460181E7, 2072.
0, 222762.0, 0.0, 0.0, 350509.0, 2.83050870811833E21, 2.5518255143796804E21, 3.7
3102454994023 47E9, 3.7393207330669594E9, 4.903336955040625E8, 4.9152868165556365
E8, 6.372774525101904E10, 5504605.314151012, 6.598916835427707 4E7, 3.45301376175
86514E20, 2.3760698086689666E20, 5.860065406452493E10, 1.5126269174992431E7, 4.0
7063224484678 6E10, 372920.18809994817, 3.728456274573894E9, 3.734936478083449E9,
 4.85534760089647 65E8, 4.810645156018754E8)

scala> val stdevs = sumSquares.zip(sums).map {
     | case(sumSq,sum) => math.sqrt(n*sumSq - sum*sum)/n
     | }
stdevs: Array[Double] = Array(2.2112057412905115E8, 6.863499759639276E14, 7.3442
4268130645E11, 0.024724644838279363, 8.277229727971912, 0.021341181438645187, 39
9.8230600259483, 0.16788995749005403, 89.76355078488069, 1596.887555789264, 0.05
909096381382015, 0.031235647884114825, 1997.4307357840669, 4.6014730470139495, 0
.06476219376906601, 0.6715020759478973, 0.0, 0.0, 0.8423184107164992, 7.56936665
0925729E7, 7.187085021339817E7, 86.90413687078873, 87.0007023641198, 31.50448919
680598, 31.5428550412096, 359.16211190333695, 3.3379652199470935, 11.55745617889
1192, 2.6437869185397055E7, 2.19309221260232E7, 344.41137477478526, 5.5333282316
50567, 287.0498168847462, 0.8688051608464176, 86.87422094314323, 86.949684517521
13, 31.349939860825497, 31.205289897465853)

scala> val means = sums.map(_ / n)
means: Array[Double] = Array(47.97930249928647, 3025.6102959185946, 868.53242473
49809, 4.453251987263699E-5, 0.0064329249161472896, 1.4169438141293589E-5, 0.034
51877551763994, 1.5181540865671702E-4, 0.1482467344505598, 0.010212116488975164,
 1.1133129968159248E-4, 3.6435698077612086E-5, 0.01135174415662492, 0.0010829499
150845814, 1.0930709423283626E-4, 0.001008054313480601, 0.0, 0.0, 0.001386580732
3980154, 332.2856902844211, 292.9065566038691, 0.176686659069149, 0.176608808127
58978, 0.05743340870124952, 0.05771894312185113, 0.791547343129074, 0.0209823873
8839084, 0.028996803779596965, 232.47077755803903, 188.66567008285074, 0.7537796
976242213, 0.03090573072805857, 0.6019347558099691, 0.0066835013086479125, 0.176
75396390031597, 0.17644262086024323, 0.05811761038498319, 0.057411668734729006)

scala> def normalize(datum: Vector) = {
     | val normalizedArray = (datum.toArray, means, stdevs).zipped.map((value, m
ean, stdev) => if (stdev <= 0) (value - mean) else (value - mean) / stdev )
     | Vectors.dense(normalizedArray)
     | }
```

Now that the above code has normalized the data, lets run the same test on a high range of k.

```
scala> val normalizedData = data.map(normalize).cache()
normalizedData: org.apache.spark.rdd.RDD[org.apache.spark.mllib.linalg.Vector] =
 MapPartitionsRDD[790] at map at <console>:55

scala> (60 to 120 by 10).par.map(k => (k, clusteringScore(normalizedData, k))).t
oList.foreach(println)
17/02/15 13:17:35 WARN BlockManager: Asked to remove block broadcast_944, which
does not exist
(60,0.005744138723519239)
(70,0.004309800110582937)
(80,0.003092077904446496)
(90,0.0040586089004989384)
(100,0.002840657435711368)
(110,0.005063136167446247)
(120,0.00264775725974362)
```

With 100 clusters, the resulting data shows us when visualized that one large k structure dominates the view, and has many clusters that correspond to small subregions. This isn't as big of a deal, but we want to now apply our entropy function to the normalized data to test for homogeneity.

Using Labels with Entropy

In our example, good clustering will hopefully give us a few types of little known attacks to look out for. This would have clusters whose collections of labels are homogeneous and have low entropy. A weighted average of entropy can therefore be used as a cluster score:

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

I then created a function to:

- Predict cluster for each datum
- Swap keys/values
- Extract collections of labels per cluster
- Count labels in collections
- Average entropy weighted by cluster size

```
scala> def clusteringScore(normalizedLabelsAndData: RDD[(String,Vector)],k:Int={
     | val model = kmeans.run(normalizedLabelsAndData.values)
     | val labelsAndClusters = normalizedLabelsAndData.mapValues(model.predict)
     | val clustersAndLabels = labelsAndClusters.map(_.swap)
     | val labelsInCluster = clustersAndLabels.groupByKey().values
     | val labelCounts = labelsInCluster.map( _.groupBy(l => l).map(_._2.size))
     | val n = normalizedLabelsAndData.count()
     | labelCounts.map(m => m.sum * entropy(m)).sum / n
     | }
```

Running this function we get 150 as a suggested choice.

```
(80,1.0079370754411006)
(90,0.9637681417493124)
(100,0.9403615199645968)
(110,0.4731764778562114)
(120,0.37056636906883805)
(130,0.36584249542565717)
(140,0.10532529463749402)
(150,0.10380319762303959)
(160,0.14469129892579444)
```

Anomaly Detector

Now, let's proceed with the full normalized data set with k=150 to actually make an anomaly detector. Anomaly detection amounts to measuring a new data points distance to its nearest centroid. If this distance exceeds some threshold, it's anomalous. This threshold can be chosen to be a distance of, say, the 100th farthest data point from among known data:

```
scala> val distances = normalizedData.map( datum => distToCentroid(datum, model) )
distances: org.apache.spark.rdd.RDD[Double] = MapPartitionsRDD[1190] at map at <console>:65

scala> val threshold = distances.top(100).last
threshold: Double = 1916.7367458942508
```

The final step is to apply this threshold to all new data points as they arrive, we can do so using Spark Streaming to apply this function to small batches of input data arriving from various sources.

Data points exceeding the threshold will send alerts to the database.

```
scala> val anomalies = originalAndData.filter {
     | case (original, datum) => val normalized = normalizeFunction(datum)
     | distToCentroid(normalized, model) > threshold
     | }.keys
```

For us the winner is a analogous winner is a .normal connection with over 200 connections!

```
0,tcp,http,S1,299,26280,
0,0,0,1,0,1,0,1,0,0,0,0,0,0,0,0,15,16,
0.07,0.06,0.00,0.00,1.00,0.00,0.12,231,255,1.00,
0.00,0.00,0.01,0.01,0.01,0.00,0.00,normal.
```

What next?

The **KMeansModel** is the essence of an anomaly detection system. This same code within Spark Streaming to score new data as it arrives in near real time could be incredibly valuable and can be found in MLib as **StreamingKMeansModel**.

Resources

1. Learning Spark, Holden Karau
2. Spark for Data Analytics, Sean Owen
3. Hadoop: The Definitive Guide, Tom White