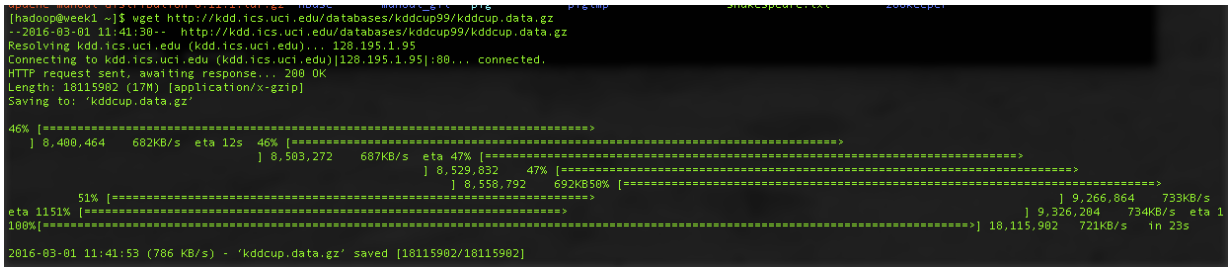
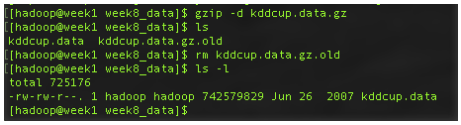
MSDS610 Week 8 Spark Continued Assignment - Nathan Worsham

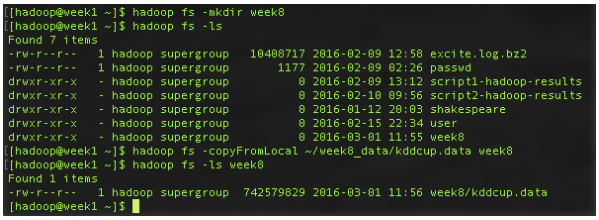
I was very interested in this weeks assignment as it pertains to what I already work in - IT Security. I started by getting the download link and getting the data.



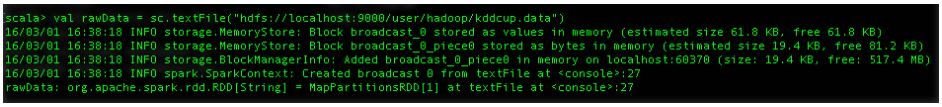
When I tried to unpack it I received errors that `tar: This does not look like a tar archive`, I had even tried providing the `z` option but received the same problem and tried redownloading the file incase it was corrupted. After looking at a superuser.com (2014) question, I realized that the file likely is just a Gzip file and not a TAR. So I then used gzip to unpack it.



I went ahead and created a new folder in the HDFS and then copied the kdd data to it.



The spark shell is opened and the csv file is loaded as an RDD.



I stumbled a bit on the location for the CSV file as I didn't realize the `hadoop fs` command starts off in the user's home directory similar to Linux as the command kept telling me it could not find the file. Once I realized the full path was `/user/hadoop/` I moved the file in the HDFS and then was able to run the command to see what labels where in the data.

```
16/03/01 16:39:32 INFO scheduler.DAGScheduler: Job 0 finished: countByValue at <console>:30, took 24.317409 s
(smurf.,2807886)
(neptune.,1072017)
(normal.,972781)
(satan.,15892)
(ipsweep.,12481)
(portsweep.,10413)
(nmap.,2316)
(back.,2203)
(warezclient.,1020)
(teardrop.,979)
(pod.,264)
(guess_passwd.,53)
(buffer_overflow.,30)
(land.,21)
(warezmaster.,20)
(imap.,12)
(rootkit.,10)
(loadmodule.,9)
(ftp_write.,8)
(multihop.,7)
(phf.,4)
(perl.,3)
(spy.,2)
```

Next the categorical values are removed as K-means clustering requires numerical data.

```
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.Vector)] = MapPartitionsRDD[6] at map at <console>:32

scala> val data = labelsAndData.values.cache()
data: org.apache.spark.rdd.RDD[org.apache.spark.mllib.linalg.Vector] = MapPartitionsRDD[7] at values at <console>:34
```

The Kmeans implementation is imported and then run on the "data" which took some time to run. Then the centroids are printed.

```
16/03/01 20:51:54 INFO clustering.KMeans: Run 0 finished in 2 iterations
16/03/01 20:51:54 INFO clustering.KMeans: Iterations took 124.725 seconds.
16/03/01 20:51:54 INFO clustering.KMeans: KMeans converged in 2 iterations.
16/03/01 20:51:54 INFO clustering.KMeans: The cost for the best run is 4.6634585670213356E18.
16/03/01 20:51:54 INFO rdd.MapPartitionsRDD: Removing RDD 4 from persistence list
16/03/01 20:51:54 INFO storage.BlockManager: Removing RDD 4
model: org.apache.spark.mllib.clustering.KMeansModel = org.apache.spark.mllib.clustering.KMeansModel@27d74c62

scala> model.clusterCenters.foreach(println)
[48.34019491959669,1834.6215497618625,826.2031900016945,5.716117204900345 6E-6,6.487793027561892E-4,7.961734678254053E-6,0.012437658596734055,3.205108575604837E
-5,0.14352904910348827,0.008888308584493399,6.018511237273984E-5,3.6746467745787934E-5,0.012934960793560386,0.0011887482315762398,7.430952366378449E-5,0.0010211
435092468404,0.0,4.0029408686043184E-7,8.351655530445469E-4,334.97350845066668,295.26714620807076,0.17797031701994342,0.17803698940427253,0.05766489875327374,0.05
7729909379127 39,0.7898841322630883,0.02117961060998736,0.028268010896292 84,232.98107822302248,189.21428335201279,0.7537133898086421,0.030710978823798966,0.605
0519309248854,0.006464107887636004,0.17809111843182601,0.17788589813474293,0.05792761150001131,0.05765922142480886]
[10999.0,0.0,1.389937401E9,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,1.0,1.0,0.0,0.0,0.0,0.0,1.0,1.0,1.0,0.0,0.0,0.0,0.65,1.0,0.0,0.0,0.0,0
.0,1.0,1.0]
```

The previous model only did a fitting of 2 clusters on the data. So seeing how the existing labels lands in each cluster, all land in one cluster except for 1 "portsweep" which is a poor model then.

```
clusterLabelCount: scala.collection.Map[(Int, String),Long] = Map((0,p
ortsweep.) -> 10412, (0,rootkit.) -> 10, (0,buffer_overflow.) -> 30, (
0,phf.) -> 4, (0,pod.) -> 264, (0,perl.) -> 3, (0,spy.) -> 2, (0,ftp_w
rite.) -> 8, (0,nmap.) -> 2316, (0,ipsweep.) -> 12481, (0,imap.) -> 12
, (0,warezmaster.) -> 20, (0,satan.) -> 15892, (0,teardrop.) -> 979, (
0,smurf.) -> 2807886, (0,neptune.) -> 1072017, (0,loadmodule.) -> 9, (
0,guess_passwd.) -> 53, (0,normal.) -> 972781, (0,land.) -> 21, (0,mul
tihop.) -> 7, (1,portsweep.) -> 1, (0,warezclient.) -> 1020, (0,back.)
 -> 2203)
```

```
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.   12481
0             land.      21
0       loadmodule.       9
0         multihop.       7
0          neptune. 1072017
0             nmap.    2316
0           normal.  972781
0             perl.       3
0              phf.       4
0              pod.     264
0        portsweep.   10412
0          rootkit.      10
0            satan.   15892
0            smurf. 2807886
0              spy.       2
0         teardrop.     979
0      warezclient.    1020
0      warezmaster.      20
1        portsweep.       1
```

Create a function to define a Euclidean distance and another to return the distance for the data point to its cluster's centroid. This will help determine the number of clusters to use.

```
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.linalg.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
```

Now another function is created to measure the average distance from the data points to the centroids.

```
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.Vector], k: Int)Double
```

Now use this function to evaluate values of k from 5 to 40, counting by 5s. Like the example, the value actually increases for 35, though they received a better output for 30 than I did.

```
16/03/02 01:43:08 INFO executor.Executor: Finished task 5.0 in stage 340.0 (TID 2040). 2709 bytes result sent to driver
16/03/02 01:43:08 INFO scheduler.TaskSetManager: Finished task 5.0 in stage 340.0 (TID 2040) in 7434 ms on localhost (6/6)
16/03/02 01:43:08 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 340.0, whose tasks have all completed, from pool
16/03/02 01:43:08 INFO scheduler.DAGScheduler: ResultStage 340 (mean at <console>:54) finished in 72.451 s
16/03/02 01:43:08 INFO scheduler.DAGScheduler: Job 228 finished: mean at <console>:54, took 72.467009 s
(5,1938.858341059188)
(10,1592.2768468128006)
(15,1407.9929596925513)
(20,1300.2743912738707)
(25,1041.6914354420257)
(30,866.4511133122714)
(35,883.4160185558521)
(40,521.4234074648682)

scala> 16/03/02 01:59:41 INFO storage.BlockManagerInfo: Removed broadcast_497_piece0 on localhost:37837 in memory (size: 12.5 KB, free
: 74.1 MB)
16/03/02 01:59:41 INFO spark.ContextCleaner: Cleaned accumulator 443
```

Running the iteration longer and decrease the threshold for the minimum amount considered significant and then finally run in parallel with `par.map`. Took longer to run than the previous but in my results the output starts to increase at 100 with 90 being the best.

```
scala> kmeans.setRuns(10)
warning: there were 1 deprecation warning(s); re-run with -deprecation for details
res5: kmeans.type = org.apache.spark.mllib.clustering.KMeans@7aeb3664

scala> kmeans.setEpsilon(1.0e-6)
res6: kmeans.type = org.apache.spark.mllib.clustering.KMeans@7aeb3664
```

```
(30,936.6630608255432)
(40,404.5011926950556)
(50,325.4491402759626)
(60,294.09423933023567)
(70,308.4794767407293)
(80,314.0387545764995)
(90,185.08670187535128)
(100,227.57595059595566)
```

Changing each feature into a Z-score or standard score can help normalize the output. This is done by subtracting the mean and dividing by the standard deviation. Various outputs leading up to the final output.

n

```
16/03/02 07:39:59 INFO scheduler.DAGS
n: Long = 4898431
```

sums

```
sums: Array[Double] = Array(2.3680206E8, 8.986765238E9, 5.357035893E9, 28.0, 3178.0, 39.0, 60925.0, 157.0, 703067.0, 39620.0, 334.0, 18
0.0, 63361.0, 5823.0, 364.0, 5002.0, 0.0, 2.0, 4091.0, 1.640844284E9, 1.446345448E9, 871775.1400000014, 872101.7299999918, 282468.46999
999863, 282786.91999999894, 3869193.1300014793, 103746.83999989525, 138433.5999999837, 1.141241758E9, 9.26852923E8, 3692012.2800011584,
150436.22999986156, 2963805.5300003863, 31663.98000003283, 872367.2000000948, 871361.6200001688, 283755.3500000003, 282440.6600000078)
```

sum-of-squares

```
sumSquares: Array[Double] = Array(2.463849195674467E24, 1.5029566894234366E37, 4.1480721617281856E36, 348.0, 2.0716658E7, 41157.0, 4.854
09351029E11, 40315.0, 2.67804240743E11, 3.876282399326076E15, 57514.0, 32330.0, 3.973113289930345E15, 1.918147959E9, 40730.0, 1.3449708E
7, 8.0, 2.0, 6009507.0, 1.3240355904052093E23, 1.2303279161186096E23, 2.2930630690647998E11, 2.2997396165149054E11, 4.76244909831684E10
, 4.764995437982619SE10, 2.6532680322640913E12, 3.8928039520852698E3, 3.945616784466233SE9, 1.4456076820536306E22, 1.0060527982319082E22
, 2.4566943315567 2E12, 1.1858752194650 42E9, 1.9054814751375215E12, 2.4872169439699925E7, 2.2942432803497485E11, 2.2986608716738 8E11, 4
.725207007332573E10, 4.683466205162662E10)
```

standard deviation and means

```
scala> val stdevs = sumSquares.zip(sums).map {
     |   case(sumSq,sum) => math.sqrt(n*sumSq - sum*sum)/n
     | }
stdevs: Array[Double] = Array(7.092160637560864E8, 1.751639521459706E15, 9.202263271748564E14, 0.0084287083113581, 2.056512384574052, 0
.0916628500071092 8, 287.68577159744495, 0.09072036889476061, 233.81918676097567, 28130.323433474063, 0.1083573070299824, 0.0012408222272
68468, 28479.8027413705, 19.747176420017 51, 0.09118608360562963, 1.6578203724101165, 0.0, 6.38978744566 91E-4, 1.1076200324017111, 1.644
073889668342E8, 1.5848272926710385E8, 216.36118707751112, 216.67593997485977, 98.6021992658583, 98.62855554368699, 735.9728858210965, 8
.914593473131658, 28.381068274848623, 5.432462364556549E7, 4.531916393331 04E7, 708.18515442807 05, 15.559303546893325, 623.6970123393882
, 2.253339083152612, 216.41685907071218, 216.59680498512827, 98.2159104102263, 97.78114589612957)

scala> val means = sums.map(_ / n)
means: Array[Double] = Array(48.34243046395876, 1834.6211752293746, 1093.6228137132073, 5.716116037972159E-6, 6.487791703098401E-4, 7.9
617330528897 93E-6, 0.012437656057623349, 3.205107921291532E-5, 0.14352901980246327, 0.000808830419373 0605, 6.818509845295 36E-5, 3.674646
024410674E-5, 0.012934958152926926, 0.001188747988896 8528, 7.430950849363806E-5, 0.0010211433007834 55, 0.0, 4.0029400271229 71E-7, 8.351
653825480036E-4, 334.97344027097654, 295.26708613431526, 0.17797028068783685, 0.1780369530570078, 0.0576650911201 5892, 0.0577301017407 4
085, 0.7898841751576126, 0.021179606286154 74, 0.028260804326933196, 232.98108271811932, 189.21424492863122, 0.7537132359323135, 0.03071
110524979561, 0.6050520115523493, 0.006464106568007762, 0.17809114796147885, 0.17788586181987023, 0.0579278038212644 66, 0....
```

normalize function

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

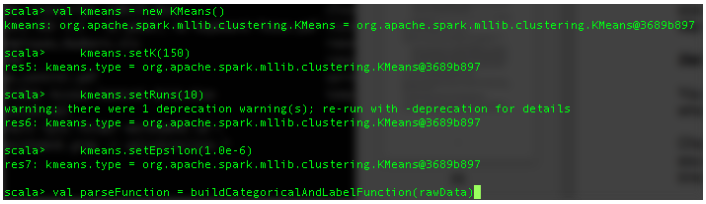
Finally the output of the k values using standard score, this time my output was similar to the example as 100 seems to be the good spot from this data.

```
(60,0.0018468709095810 43)
(70,0.0015864742376780617)
(80,0.0017250487490298987)
(90,0.0013255950070843933)
(100,0.001167844190513282)
(110,0.0011823034792567886)
(120,0.0010299600744796914)
```

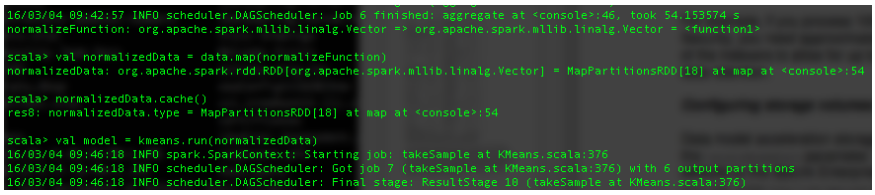
Using "one-hot" encoding, the categorical values are brought back in as features for consideration in the k-means clustering. I used the code they offered from git, but I have to say the code versus the book is not exactly one-to-one and can be confusing finding where they meet up. My output was a bit different from theirs as mine is much bigger values and 160 has the lowest value, which might be explained by using the code from git. Unfortunately it took much too long to get those results that I was not going to try running it again.

```
(80,1.1072907426522338)
(90,0.7689266072822099)
(100,0.6951908257927574)
(110,0.6199581232656192)
(120,0.5505527491675478)
(130,0.4939910868584523)
(140,0.4899918451619 61)
(150,0.4428285002738 9323)
(160,0.4223527403783 2266)
```

Worrying about how long the last clustering portion took, I wanted to just use their recommendation of k=150 and move onto the end, in order to make sure I could complete the assignment on time.



Again I was trying my best at following along with their code from git, but this time I was resolved to run it in pieces if I could. I also shutdown my VM and gave it another cpu and another GB of RAM to try to help the number crunching and time to complete.

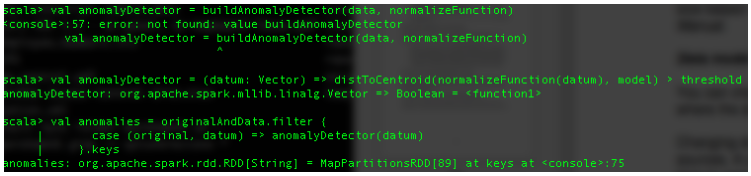


But as it turns out the only portion that took time to compute was the `val model = kmeans.run(normalizedData)`. As expected there are many more centroids now.

After that I went ahead and tried printing the clusters with labels as before like the book showed using their val `clusterLabelCount` but I received several errors and a driver stacktrace that I wasn't sure about so I moved onto the anomaly detection. Here in the git code the line

`val anomalyDetector = buildAnomalyDetector(data, normalizeFunction)`

calls the entire previous function, so I ran the rest of that function and then assumed that the value `anomolyDetector` needed to equal the output of the function `buildAnomalyDetector`.



But then I realized the book's code spells this out differently than the git code and takes this into account.

```
scala> val anomalies = originalAndData.filter { case (original, datum) =>
     |       val normalized = normalizeFunction(datum)
     |       distToCentroid(normalized, model) > threshold
     |     }.keys
anomalies: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[91] at keys at <console>:74
```

Either way they both end up with the same output on anomalies.

```
16/03/04 12:47:33 INFO scheduler.DAGScheduler: Job 42 finished: take at <console>:74, took 8.832429 s
0,tcp,telnet,S1,167,19827,0,0,0,0,0,1,83,1,2,91,0,0,0,0,0,0,1,1,1.00,1.00,0.00,0.00,1.00,0.00,0.00,0.00,21,0,0.29,0.14,0.05,0.25,0.05,0.12,0.00,0.00,normal.
0,tcp,telnet,S1,145,13236,0,0,0,0,0,1,31,1,2,38,0,0,0,0,0,0,1,1,1.00,1.00,0.00,0.00,1.00,0.00,0.00,0.00,29,10,0.28,0.10,0.03,0.20,0.07,0.20,0.00,0.00,normal.
9,tcp,telnet,SF,307,2374,0,0,1,0,0,1,0,1,0,1,3,1,0,0,0,0,1,1,0.00,0.00,0.00,0.00,1.00,0.00,0.00,0.00,69,4,0.03,0.04,0.01,0.75,0.00,0.00,0.00,0.00,normal.
11565,tcp,telnet,SF,565,111864,0,0,0,0,0,1,217,1,2,247,0,0,4,0,0,0,1,1,0.00,0.00,0.00,0.00,1.00,0.00,0.00,0.00,8,3,0.38,0.25,0.12,0.00,0.00,0.00,0.12,0.33,normal.
0,tcp,telnet,S1,142,26316,0,0,0,0,0,1,31,1,2,36,0,0,0,0,0,0,1,1,1.00,1.00,0.00,0.00,1.00,0.00,0.00,0.00,94,22,0.22,0.04,0.01,0.09,0.19,0.82,0.00,0.00,normal.
0,tcp,http,S1,299,26280,0,0,0,1,0,1,0,1,0,0,0,0,0,0,0,0,15,16,0.07,0.06,0.00,0.00,0.00,1.00,0.00,0.00,0.12,231,255,1.00,0.00,0.00,0.00,0.01,0.01,0.01,0.00,0.00,normal.
0,tcp,telnet,S1,2895,14208,0,0,0,0,0,1,0,0,0,0,0,13,0,0,0,0,0,1,1,1.00,1.00,0.00,0.00,1.00,0.00,0.00,0.00,21,2,0.10,0.10,0.05,0.00,0.05,0.50,0.00,0.00,normal.
23,tcp,telnet,SF,104.276,0,0,0,0,0,5,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0.00,0.00,0.00,0.00,1.00,0.00,0.00,0.00,1,2,1.00,0.00,1.00,1.00,0.00,0.00,0.00,0.00,guess_passwd.
13,tcp,telnet,SF,246,11938,0,0,0,0,0,4,1,0,0,0,0,0,2,0,0,0,0,0,1,1,0.00,0.00,0.00,0.00,0.00,1.00,0.00,0.00,0.00,89,2,0.02,0.04,0.01,0.00,0.00,0.00,0.00,0.00,normal.
12249,tcp,telnet,SF,3043,44466,0,0,0,1,0,1,13,1,0,0,12,0,0,0,0,0,0,1,1,0.00,0.00,0.00,0.00,0.00,1.00,0.00,0.00,0.00,61,8,0.13,0.05,0.02,0.00,0.00,0.00,0.00,0.00,normal.
```

The most anomalous event the book had was contained in my top 10 output, but is the 6th listing. My first listing is a telnet connection. I was curious about the connections–more specifically my top listing, so I went back to the KDD cup data for what the columns represented. Also using this site–ita.ee.lbl.gov(Paxson, nd) to understand the "flags". It would seem that an "S1" state is for when 2 sides have established a connection (TCP handshake) but nothing further has been "seen". Though according to the listing, the destination has received 19827 bytes. Telnet connections nowadays themselves are anomalous if they are seen over the internet because they are not a secure connection–anyone listening could see exactly what transpired, in 1999 that probably would have been common to see but today it is replaced by SSH and other more secure communication methods. My guess would be that the connection is anomalous because there are multiple 100% error rates (3x) happening on this connection.

When I had shutdown my VM to give it more resources I also exported a copy of it. While I was running the computations for the anomaly detector I started up the cloned VM on another computer and ran the clustering take-4 functions from the git code–the labels with entropy. This time I ended up receiving a similar output to the book that pointed to k=150 was a good choice.

```
scala>  def clusteringTake4(rawData: RDD[String]): Unit = {
     |     val parseFunction = buildCategoricalAndLabelFunction(rawData)
     |     val labelsAndData = rawData.map(parseFunction)
     |     val normalizedLabelsAndData =
     |       labelsAndData.mapValues(buildNormalizationFunction(labelsAndData.values)).cache()
     |
     |     (80 to 160 by 10).map(k =>
     |       (k, clusteringScore3(normalizedLabelsAndData, k))).toList.foreach(println)
     |
     |     normalizedLabelsAndData.unpersist()
     |   }
clusteringTake4: (rawData: org.apache.spark.rdd.RDD[String])Unit

scala>  clusteringTake4(rawData)
```

```
(80,0.024399277361340916)
(90,0.02313454365166319 6)
(100,0.021064759237645326)
(110,0.023719211980023924)
(120,0.016525320616698 6)
(130,0.021116857949033466)
(140,0.019211408342509645)
(150,0.014650666763673204)
(160,0.017451347862165664)
```

References

superuser.com, 2014. Retrieved from http://superuser.com/questions/841865/extracting-a-tar-gz-file-returns-this-does-not-look-like-a-tar-archive

UCI KDD Archive, 1999. Retrieved from http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

Paxson, Vern. n.d. Retrieved from http://ita.ee.lbl.gov/html/contrib/tcp-reduce-doc.html