# Homework Writeup

## Using Apache Spark

### //--Anomaly Detection--

/\*

Anomaly detection is used to find fraud and other type of attacks. Since normal users actions would be classified, but these attacks, would be an unusual interaction, which would classify it outside of most clusters. This is known as unsupervised learning, since we don't know if an instance is an anomaly or not.

\*/

### //--K-means Clustering--

/\*

Clustering is one of the most common unsupervised learning alg. We have to define how many clusters, k, we want (IMPORTANT HYPERPARAMETER)

We find the centroid of each cluster, aka the means of each instance, and if the distance from the centroid is little, then it is a 'like' instance (this is within Euclidean space).

\*/

### 

### //--Network Intrusion--

/\*

While it may be easy to determine most common attacks, like accessing every port when it's only normal for consumers to access one or two, but still find a needle in a haystack of thousands of network requests.

Most malicious attacks have some known patterns, but what do we do when itis a new type of attack that has never been seen before?

"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."

All items within the cluster are normal connections, anything outside the clusters are unusual and could be an anomaly.

\*/

### 

### //--KDD Cup 1999 Data Set--

/\*

Check out kaggle, seems supppper cool!!! Data set is 708mb, with 5 million connections. Each connection is one line of CSV with 38 attributes.

Example:

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.

Most of these categorical features are 0 or 1, so we can convert them to binary. In the example, we see that it is a normal connection, and some other instances will even mention what attack this is, but not the point of the project. We will be setting aside that label so we can focus on working on distinguishing anomaly

\*/

### //--A First Take on Clustering--

val dataWithoutHeader = spark.read. //reading the data in, but lacks header

option("inferSchema", true).

option("header", false).

csv("/proj/cse398-498/course/AAS\_CH5/kddcup.data")

dataWithoutHeader.head()

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") //adding our column labels to our dataWithoutHeader

data.select("label").groupBy("label").count().orderBy($"count".desc).show(25) //select count of labels and order by descending, showing top 25

/\*

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

| label| count|

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

| 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|

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

\*/

//23 total labels

//As of right now, the second column may be nonnumeric (tcp, udp, icmp) and same label column. K-mean clustering need numerical, so ignore those

//"A VectorAssembler creates a feature vector, a KMeans implementation creates a model off feature vectors, and a Pipeline stitches it all together"

import org.apache.spark.ml.Pipeline

import org.apache.spark.ml.clustering.{KMeans, KMeansModel}

import org.apache.spark.ml.feature.VectorAssembler

val numericData = data.drop("protocol\_type", "service", "flag").cache()

val assembler = new VectorAssembler().

setInputCols(numericData.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(numericData)

val kmeansModel = pipelineModel.stages.last.asInstanceOf[KMeansModel]

kmeansModel.clusterCenters.foreach(println)

/\*

[48.34019491959669,1834.6215497618625,826.2031900016945,5.7161172049003456E-6,6.487793027561892E-4,7.961734678254053E-6,0.012437658596734055,3.205108575604837E-5,0.14352904910348827,0.00808830584493399,6.818511237273984E-5,3.6746467745787934E-5,0.012934960793560386,0.0011887482315762398,7.430952366370449E-5,0.0010211435092468404,0.0,4.082940860643104E-7,8.351655530445469E-4,334.9735084506668,295.26714620807076,0.17797031701994256,0.1780369894027269,0.05766489875327379,0.05772990937912744,0.7898841322630906,0.02117961060991097,0.028260810096297884,232.98107822302248,189.21428335201279,0.7537133898007772,0.03071097882384052,0.605051930924901,0.006464107887636894,0.1780911843182284,0.1778858981346887,0.05792761150001272,0.05765922142401037]

[10999.0,0.0,1.309937401E9,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,1.0,0.0,0.0,1.0,1.0,1.0,0.0,0.0,255.0,1.0,0.0,0.65,1.0,0.0,0.0,0.0,1.0,1.0]

\*/

/\*

The values are the coordinates of the centroid with respect to input features.

It printed of 2 vectors, meaning that it found k-2 clusters to the data. This nowhere near the 23 distinct types of connections

\*/

c

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| portsweep.| 10412|

| 0| nmap.| 2316|

| 0| back.| 2203|

| 0| warezclient.| 1020|

| 0| teardrop.| 979|

| 0| pod.| 264|

| 0| guess\_passwd.| 53|

| 0|buffer\_overflow.| 30|

| 0| land.| 21|

| 0| warezmaster.| 20|

| 0| imap.| 12|

| 0| rootkit.| 10|

| 0| loadmodule.| 9|

| 0| ftp\_write.| 8|

| 0| multihop.| 7|

| 0| phf.| 4|

| 0| perl.| 3|

| 0| spy.| 2|

| 1| portsweep.| 1| //only one data point in cluster 1

+-------+----------------+-------+ //no other data points PERIOD

\*/

### //--Choosing K--

/\*

It is clear there are 23 distint types in our data, but k may be greater than 23. How do we find K? A clustering is considered good if its

data in the cluster is near its centroid, with closer the mean euclidian distance the better. Since there is no evaluator for K mean, we can use

kmeansModel computeCost to "computes the sum of squared distances and can easily be used to compute the mean squared distance."

\*/

import org.apache.spark.sql.DataFrame

import scala.util.Random

def clusteringScore0(data: DataFrame, k: Int): Double = { //input is DF and int for k, returns a double

val assembler = new VectorAssembler().

setInputCols(data.columns.filter(\_ != "label")).

setOutputCol("featureVector") //create a vector including every column but label

val kmeans = new KMeans().

setSeed(Random.nextLong()).

setK(k). //expected number of clusters

setPredictionCol("cluster").

setFeaturesCol("featureVector")

val pipeline = new Pipeline().setStages(Array(assembler, kmeans))

val kmeansModel = pipeline.fit(data).stages.last.asInstanceOf[KMeansModel] //grab our model

kmeansModel.computeCost(assembler.transform(data)) / data.count() //"Compute mean from total squared distance (“cost”)"

}

(20 to 100 by 20).map(k => (k, clusteringScore0(numericData, k))).

//first we have a loop that goes from 20 to 100 by 20, then the value of k is set to the loop val, and then printed out as a tuple (k, score)

foreach(println)

/\*

(20,1.077001256461918E8)

(40,1.749847326308563E7)

(60,1.1806382683844185E7)

(80,1.5515384120060476E7)

(100,4691964.790611011)

\*/

/\*

as more clusters increase, we can have the data break down more, meaning each cluster increase will decrease the distance.

if we want to give as many clusters as data, we will have a distance of 0!!

Another thing to notice is that, from 60 to 80, distance increased. Kmeans is not necessarily able to find best cluster given k.

Better versions: Kmeans++ and Kmeans||. Cant gurantee optimalness given the randomness.

For k=80, the program may have been just a suboptimal cluster, or stopped early before maximum

We can improve everything by increasing iterations, by decreasing the setTol value, which allows the centroid to move longer

\*/

def clusteringScore1(data: DataFrame, k: Int): Double = { //input is DF and int for k, returns a double

val assembler = new VectorAssembler().

setInputCols(data.columns.filter(\_ != "label")).

setOutputCol("featureVector") //create a vector including every column but label

val kmeans = new KMeans().

setSeed(Random.nextLong()).

setK(k). //expected number of clusters

setMaxIter(40). //increase from default 20 to 40

setTol(1.0e-5). //decrease from default 1.0e-4

setPredictionCol("cluster").

setFeaturesCol("featureVector")

val pipeline = new Pipeline().setStages(Array(assembler, kmeans))

val kmeansModel = pipeline.fit(data).stages.last.asInstanceOf[KMeansModel] //grab our model

kmeansModel.computeCost(assembler.transform(data)) / data.count() //"Compute mean from total squared distance (“cost”)"

}

(20 to 100 by 20).map(k => (k, clusteringScore1(numericData, k))).

//first we have a loop that goes from 20 to 100 by 20, then the value of k is set to the loop val, and then printed out as a tuple (k, score)

foreach(println)

/\*

(20,5.454092597347078E7)

(40,4.566135854151461E7)

(60,3.787964426066939E7)

(80,9618261.718023658)

(100,9925572.773724418)

\*/

/\*

we can find the best k value for cluster when we see the decrease is no longer as exponential and more flat, which seems to show that the right

value of k is most likely past 100

\*/

### //--Visualization with SparkR--

clusters\_data <- read.df("/path/to/kddcup.data", "csv", inferSchema = "true", header = "false")

colnames(clusters\_data) <- c(

"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, c("protocol\_type", "service", "flag", "label"))) //drop nonnumeric values

kmeans\_model <- spark.kmeans(numeric\_only, ~ ., k = 100, maxIter = 40, initMode = "k-means||") //~ 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 executed in R.

\*/

clustering <- predict(kmeans\_model, numeric\_only)

clustering\_sample <- collect(sample(clustering, FALSE, 0.01))

str(clustering\_sample)

/\*

'data.frame': 48990 obs. of 39 variables:

$ duration : int 0 0 0 0 0 0 0 0 0 0 ...

$ src\_bytes : int 212 212 217 234 239 236 239 307 212 230 ...

$ dst\_bytes : int 1402 1444 63479 1515 2164 2112 2112 468 1247 2395 ...

$ land : int 0 0 0 0 0 0 0 0 0 0 ...

$ wrong\_fragment : int 0 0 0 0 0 0 0 0 0 0 ...

$ urgent : int 0 0 0 0 0 0 0 0 0 0 ...

$ hot : int 0 0 0 0 0 0 0 0 0 0 ...

$ num\_failed\_logins : int 0 0 0 0 0 0 0 0 0 0 ...

$ logged\_in : int 1 1 1 1 1 1 1 1 1 1 ...

$ num\_compromised : int 0 0 0 0 0 0 0 0 0 0 ...

$ root\_shell : int 0 0 0 0 0 0 0 0 0 0 ...

$ su\_attempted : int 0 0 0 0 0 0 0 0 0 0 ...

$ num\_root : int 0 0 0 0 0 0 0 0 0 0 ...

$ num\_file\_creations : int 0 0 0 0 0 0 0 0 0 0 ...

$ num\_shells : int 0 0 0 0 0 0 0 0 0 0 ...

$ num\_access\_files : int 0 0 0 0 0 0 0 0 0 0 ...

$ num\_outbound\_cmds : int 0 0 0 0 0 0 0 0 0 0 ...

$ is\_host\_login : int 0 0 0 0 0 0 0 0 0 0 ...

$ is\_guest\_login : int 0 0 0 0 0 0 0 0 0 0 ...

$ count : int 4 12 2 8 5 18 30 8 19 14 ...

$ srv\_count : int 4 12 2 8 5 20 31 8 19 18 ...

$ serror\_rate : num 0 0 0 0 0 0 0 0 0 0 ...

$ srv\_serror\_rate : num 0 0 0 0 0 0 0 0 0 0 ...

$ rerror\_rate : num 0 0 0 0 0 0 0 0 0 0 ...

$ srv\_rerror\_rate : num 0 0 0 0 0 0 0 0 0 0 ...

$ same\_srv\_rate : num 1 1 1 1 1 1 1 1 1 1 ...

$ diff\_srv\_rate : num 0 0 0 0 0 0 0 0 0 0 ...

$ srv\_diff\_host\_rate : num 0 0 0 0 0 0.1 0.06 0 0 0.11 ...

$ dst\_host\_count : int 4 12 2 8 40 53 174 242 255 255 ...

$ dst\_host\_srv\_count : int 95 103 255 255 255 255 255 255 255 255 ...

$ dst\_host\_same\_srv\_rate : num 1 1 1 1 1 1 1 1 1 1 ...

$ dst\_host\_diff\_srv\_rate : num 0 0 0 0 0 0 0 0 0 0 ...

$ dst\_host\_same\_src\_port\_rate: num 0.25 0.08 0.5 0.12 0.03 0.02 0.01 0 0 0 ...

$ dst\_host\_srv\_diff\_host\_rate: num 0.05 0.05 0.04 0.05 0.04 0.04 0.02 0.01 0 0 ...

$ dst\_host\_serror\_rate : num 0 0 0 0 0 0 0 0 0 0 ...

$ dst\_host\_srv\_serror\_rate : num 0 0 0 0 0 0 0 0 0 0 ...

$ dst\_host\_rerror\_rate : num 0 0 0 0 0 0 0 0 0 0 ...

$ dst\_host\_srv\_rerror\_rate : num 0 0 0 0 0 0 0 0 0 0 ...

$ prediction : int 0 0 28 0 0 0 0 0 0 0 ...

\*/

//it’s possible to extract the cluster assignment and then show statistics about the distribution of assignments

clusters <- clustering\_sample["prediction"] //only the clustering assignmnet column (prediction)

data <- data.matrix(within(clustering\_sample, rm("prediction"))) //everything but the clustering assignment

table(clusters)

/\*

clusters

0 11 17 18 23 25 28 31 33 36 40 64 79

47430 1 1 1 3 266 87 42 1140 15 2 1 1

\*/

//Didn't realize we shouldn't do part over SparkR D:

### //--Feature Normalization--

/\*

We can normalize our attributes by converting it to standard score by taking the feature value and subtracting the feature means from it and dividing by standard deviation.

normalized\_i = (feature\_i/u\_i)/o\_i

Standardizing the data has no effect on clustering because it shifts all points and normalizes them. We used StandardScaler to do so.

\*/

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(numericData, k))).

foreach(println)

/\*

(60,1.2085102766608882)

(90,0.7556338365580493)

(120,0.5066266588775019)

(150,0.4078229693345159)

(180,0.33735657183433865)

(210,0.2808956418075503)

(240,0.24121210670843768)

(270,0.22372480024764824)

\*/

//--Categorical Variables--

/\*

Normalizing the data is not enough. We can improve this more by putting our nominal values back, which is leaving out valuable information.

"The categorical features can translate into several binary indicator features using one-hot encoding, which can be viewed as numeric dimensions."

For example, instead of having one column to hold tcp, udp, and icmp, we can have 3 columns, where being tcp makes it 0,1,1 and udp 0,1,0

This is a two step process, which we can use the pipeline to take advantage off.

1. convert string values to integer indicies using StringIndexer

2. incode these integer into a vector using OneHotEncoder

\*/

import org.apache.spark.ml.{PipelineModel, Pipeline}

import org.apache.spark.ml.clustering.{KMeans, KMeansModel}

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

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

import org.apache.spark.sql.{DataFrame, SparkSession}

import scala.util.Random

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")

}

def clusteringScore3(data: DataFrame, k: Int): Double = {

val (protoTypeEncoder, protoTypeVecCol) = oneHotPipeline("protocol\_type")

val (serviceEncoder, serviceVecCol) = oneHotPipeline("service")

val (flagEncoder, flagVecCol) = oneHotPipeline("flag")

// Original columns, without label / string columns, but with new vector encoded cols

val assembleCols = Set(data.columns: \_\*) --

Seq("label", "protocol\_type", "service", "flag") ++

Seq(protoTypeVecCol, serviceVecCol, flagVecCol)

val assembler = new VectorAssembler().

setInputCols(assembleCols.toArray).

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(protoTypeEncoder, serviceEncoder, flagEncoder, 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, clusteringScore3(data, k))).

foreach(println)

/\*

(60,40.154767336639836)

(90,16.17255608429269)

(120,3.1997476335409756)

(150,2.161171637755817)

(180,1.5142138090107071)

(210,1.2522240551903454)

(240,1.1111356336856535)

(270,0.9741348064961258)

\*/

### //--Using Labels with Entropy--

/\*

We have been using the given label for each instance to doubel check the quality of clustering. This practice can be nromalized more.

We want all instances to pretty match up with the human based labels given, and not to put different labeled instances near one another.

In Chapter 4 we used Gini Impurity and Entropy.

\*/

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

}

def fitPipeline4(data: DataFrame, k: Int): PipelineModel = {

val (protoTypeEncoder, protoTypeVecCol) = oneHotPipeline("protocol\_type")

val (serviceEncoder, serviceVecCol) = oneHotPipeline("service")

val (flagEncoder, flagVecCol) = oneHotPipeline("flag")

// Original columns, without label / string columns, but with new vector encoded cols

val assembleCols = Set(data.columns: \_\*) --

Seq("label", "protocol\_type", "service", "flag") ++

Seq(protoTypeVecCol, serviceVecCol, flagVecCol)

val assembler = new VectorAssembler().

setInputCols(assembleCols.toArray).

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(protoTypeEncoder, serviceEncoder, flagEncoder, assembler, scaler, kmeans))

pipeline.fit(data)

}

def clusteringScore4(data: DataFrame, k: Int): Double = {

val pipelineModel = fitPipeline4(data, k)

// Predict cluster for each datum

val clusterLabel = pipelineModel.transform(data).

select("cluster", "label").as[(Int, String)]

val weightedClusterEntropy = clusterLabel.

// Extract collections of labels, per cluster

groupByKey { case (cluster, \_) => cluster }.

mapGroups { case (\_, clusterLabels) =>

val labels = clusterLabels.map { case (\_, label) => label }.toSeq

// Count labels in collections

val labelCounts = labels.groupBy(identity).values.map(\_.size)

labels.size \* entropy(labelCounts)

}.collect()

// Average entropy weighted by cluster size

weightedClusterEntropy.sum / data.count()

}

(60 to 270 by 30).map(k => (k, clusteringScore4(data, k))).

foreach(println)

### //--Clustering in Action--

/\*

Since from p6 we saw that 180 was the best choice based of entropy, we are going to rebuild it with 180 = k and show each label goes under what cluster

\*/

val pipelineModel = fitPipeline4(data, 180)

val countByClusterLabel = pipelineModel.transform(data).

select("cluster", "label").

groupBy("cluster", "label").count().

orderBy("cluster", "label")

countByClusterLabel.show()

/\*

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

|cluster| label| count|

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

| 0| ipsweep.| 40|

| 0| nmap.| 6|

| 0| normal.| 3374|

| 0|portsweep.| 2|

| 0| satan.| 7|

| 0| smurf.|2807852|

| 1| neptune.| 3|

| 1| normal.| 22641|

| 2| neptune.| 1034|

| 2| normal.| 4|

| 2|portsweep.| 7|

| 2| satan.| 4|

| 3| ipsweep.| 13|

| 3| neptune.| 1044|

| 3|portsweep.| 14|

| 3| satan.| 3|

| 4| ipsweep.| 13|

| 4| neptune.| 1038|

| 4|portsweep.| 14|

| 4| satan.| 3|

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

\*/

val kMeansModel = pipelineModel.stages.last.asInstanceOf[KMeansModel] //grab the best kMeansmodel

val centroids = kMeansModel.clusterCenters //save to centroids

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 //Single output implicitly named “value"

//the threshold would be the 100th farthest item within the cluster

val originalCols = data.columns

val anomalies = clustered.filter { row =>

val cluster = row.getAs[Int]("cluster") //grab the cluster we are workign with

val vec = row.getAs[Vector]("scaledFeatureVector") //grab scaled feature vector

Vectors.sqdist(centroids(cluster), vec) >= threshold //any that are above the threshold

}.select(originalCols.head, originalCols.tail:\_\*)

anomalies.first()

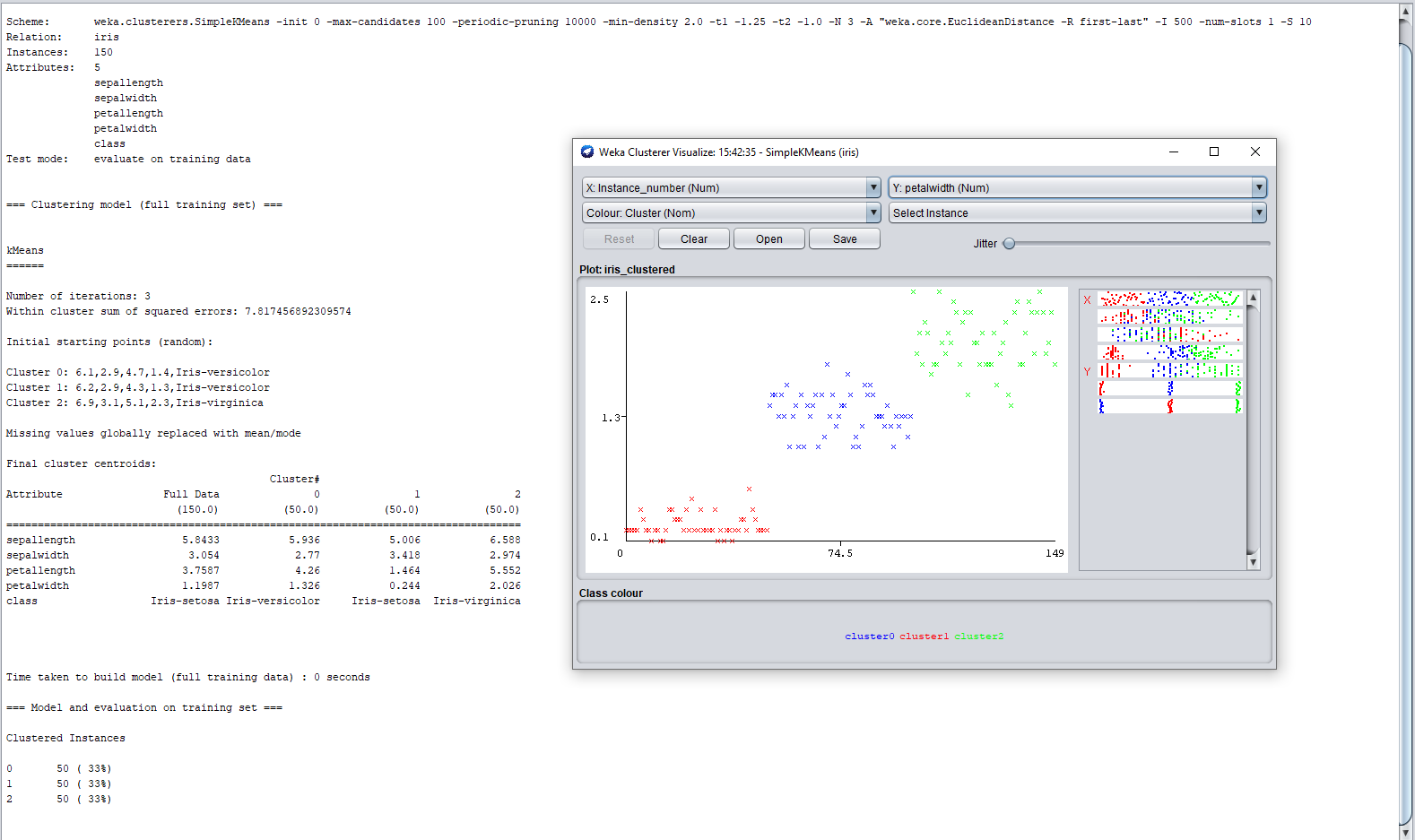
/\*

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

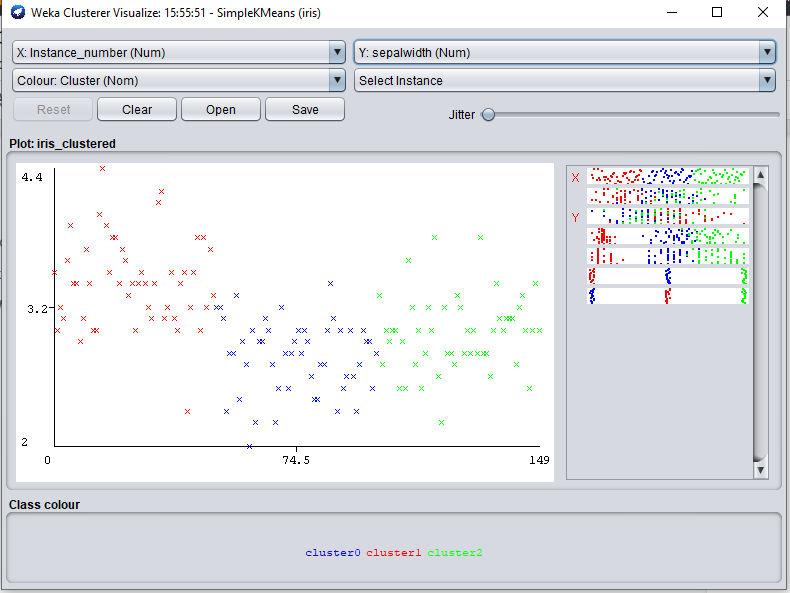
\*/

## Using Weka

Clustering was done in Weka by first finding and loading up the iris.arff dataset. Once this was done, we clicked on Cluster and chose simple K means as one of the dropdown options. Once we have selected the right cluster algorithm, it is necessary to then choose the number of clusters, in which case is 3 and can be modified by selecting the textbook next to the classification option. Once that has been selected, we just need to run it on the training set.



A few things I did notice. While following the Geeks for Geeks tutorial, each image didn’t match up with the instructions given. Each image on the tutorial showed that they used the EM clusterer and due to that, I don’t know if the output I received was the correct one based off the instructions. When speaking about the actual data petal length and width showed the most contrast and had most likely the lowest mean distance from centroid. However, when looking at sepal width, our display is an inverse of the relationship between instance and petal length/width.



# 

# Extension Writeup

## Iris Clustering

import org.apache.spark.ml.{PipelineModel, Pipeline}

import org.apache.spark.ml.clustering.{KMeans, KMeansModel}

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

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

import org.apache.spark.sql.{DataFrame, SparkSession}

import scala.util.Random

val dataWithoutHeader = spark.read. //parse the data directly from csv by following these options

option("header", "false"). //keep header

option("inferSchema", "true"). //this means infer type for column

csv("iris2.csv") //this be where it stored

//If you try to read the csv in with headers, it will always say that variable we are dealing with is string

Another huge waste of time here was trying to find function to load in arff or trying to convert it within spark. It was way smarter to just edit the iris dataset by hand using excel or <https://pulipulichen.github.io/jieba-js/weka/arff2csv/> and saving it as a csv to work with.

val data = dataWithoutHeader.toDF(

"sepal\_length", "sepal\_width", "petal\_length", "petal\_width", "class") //adding our column labels to our dataWithoutHeader

data.show()

/\*

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

|sepal\_length|sepal\_width|petal\_length|petal\_width| class|

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

| 5.1| 3.5| 1.4| 0.2|Iris-setosa|

| 4.9| 3.0| 1.4| 0.2|Iris-setosa|

| 4.7| 3.2| 1.3| 0.2|Iris-setosa|

| 4.6| 3.1| 1.5| 0.2|Iris-setosa|

| 5.0| 3.6| 1.4| 0.2|Iris-setosa|

| 5.4| 3.9| 1.7| 0.4|Iris-setosa|

| 4.6| 3.4| 1.4| 0.3|Iris-setosa|

| 5.0| 3.4| 1.5| 0.2|Iris-setosa|

| 4.4| 2.9| 1.4| 0.2|Iris-setosa|

| 4.9| 3.1| 1.5| 0.1|Iris-setosa|

| 5.4| 3.7| 1.5| 0.2|Iris-setosa|

| 4.8| 3.4| 1.6| 0.2|Iris-setosa|

| 4.8| 3.0| 1.4| 0.1|Iris-setosa|

| 4.3| 3.0| 1.1| 0.1|Iris-setosa|

| 5.8| 4.0| 1.2| 0.2|Iris-setosa|

| 5.7| 4.4| 1.5| 0.4|Iris-setosa|

| 5.4| 3.9| 1.3| 0.4|Iris-setosa|

| 5.1| 3.5| 1.4| 0.3|Iris-setosa|

| 5.7| 3.8| 1.7| 0.3|Iris-setosa|

| 5.1| 3.8| 1.5| 0.3|Iris-setosa|

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

\*/

data.select("class").groupBy("class").count().orderBy($"count".desc).show()

/\*

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

| class|count|

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

| Iris-setosa| 50|

| Iris-virginica| 50|

|Iris-versicolor| 50|

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

\*/

val assembler = new VectorAssembler().

setInputCols(data.columns.filter(\_ != "class")).

setOutputCol("featureVector")

val kmeans = new KMeans().

setPredictionCol("cluster").

setFeaturesCol("featureVector")

val pipeline = new Pipeline().setStages(Array(assembler, kmeans))

val pipelineModel = pipeline.fit(data)

val kmeansModel = pipelineModel.stages.last.asInstanceOf[KMeansModel]

kmeansModel.clusterCenters.foreach(println)

val withCluster = pipelineModel.transform(data)

withCluster.select("cluster", "class").

groupBy("cluster", "class").count().

orderBy($"cluster", $"count".desc).

show(25)

/\*

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

|cluster| class|count|

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

| 0| Iris-setosa| 50|

| 0|Iris-versicolor| 3|

| 1| Iris-virginica| 50|

| 1|Iris-versicolor| 47|

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

\*/

Right off, we can tell that the iris-setosa instances are actually correlated enough that when generating only a double cluster, virginica and versicolor were forces into the same group for the most part.

def clusteringScore0(data: DataFrame, k: Int): Double = { //input is DF and int for k, returns a double

val assembler = new VectorAssembler().

setInputCols(data.columns.filter(\_ != "class")).

setOutputCol("featureVector") //create a vector including every column but class

val kmeans = new KMeans().

setSeed(Random.nextLong()).

setK(k). //expected number of clusters

setPredictionCol("cluster").

setFeaturesCol("featureVector")

val pipeline = new Pipeline().setStages(Array(assembler, kmeans))

val kmeansModel = pipeline.fit(data).stages.last.asInstanceOf[KMeansModel] //grab our model

kmeansModel.computeCost(assembler.transform(data)) / data.count() //"Compute mean from total squared distance (“cost”)"

}

(2 to 10 by 1).map(k => (k, clusteringScore0(data, k))).foreach(println)

/\*

(2,1.0157913765156006)

(3,0.5262722761743098)

(4,0.47560297882910724)

(5,0.33374438771381715)

(6,0.25959159829060496)

(7,0.24888640031641263)

(8,0.20232136176001567) //lowest value before increasing again

(9,0.20872691425647608)

(10,0.20203308618381602)

\*/

At this point, I really wasn’t concerned about the fact the values aren’t changing much, but what struck me as unique was that at iteration 9, the mean distance increased.

def clusteringScore1(data: DataFrame, k: Int): Double = { //input is DF and int for k, returns a double

val assembler = new VectorAssembler().

setInputCols(data.columns.filter(\_ != "class")).

setOutputCol("featureVector") //create a vector including every column but class

val kmeans = new KMeans().

setSeed(Random.nextLong()).

setK(k). //expected number of clusters

setMaxIter(40). //increase from default 20 to 40

setTol(1.0e-5). //decrease from default 1.0e-4

setPredictionCol("cluster").

setFeaturesCol("featureVector")

val pipeline = new Pipeline().setStages(Array(assembler, kmeans))

val kmeansModel = pipeline.fit(data).stages.last.asInstanceOf[KMeansModel] //grab our model

kmeansModel.computeCost(assembler.transform(data)) / data.count() //"Compute mean from total squared distance (“cost”)"

}

(2 to 10 by 1).map(k => (k, clusteringScore1(data, k))).foreach(println)

/\*

(2,1.0157913765156006)

(3,0.5263004388398425)

(4,0.47560297882910724)

(5,0.33245769509255996)

(6,0.26167887261758693)

(7,0.23361839968311168) //lowest value before increasing again

(8,0.23935369068839407)

(9,0.20702411477411337)

(10,0.17662681818181974)

\*/

Just like earlier, at iteration 8, it increased from its previous value. I think at this point, I may have an idea of what is going on. Lets keep comparing asd see what happens.

def clusteringScore2(data: DataFrame, k: Int): Double = {

val assembler = new VectorAssembler().

setInputCols(data.columns.filter(\_ != "class")).

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()

}

(2 to 10 by 1).map(k => (k, clusteringScore2(data, k))).foreach(println)

/\*

(2,1.4816030602123282)

(3,0.9335069634658385)

(4,0.7628156641257224)

(5,0.705178958270994)

(6,0.6464178124155567)

(7,0.486393794433478)

(8,0.4335494544262959)

(9,0.3941283307562037)

(10,0.3155397406655387)

\*/

Interesting enough, the distance increased when normalizing the data. Which is the exact opposite of what typically happens.

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

}

def fitPipeline4(data: DataFrame, k: Int): PipelineModel = {

val assembler = new VectorAssembler().

setInputCols(data.columns.filter(\_ != "class")).

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))

pipeline.fit(data)

}

def clusteringScore4(data: DataFrame, k: Int): Double = {

val pipelineModel = fitPipeline4(data, k)

// Predict cluster for each datum

val clusterLabel = pipelineModel.transform(data).

select("cluster", "class").as[(Int, String)]

val weightedClusterEntropy = clusterLabel.

// Extract collections of labels, per cluster

groupByKey { case (cluster, \_) => cluster }.

mapGroups { case (\_, clusterLabels) =>

val labels = clusterLabels.map { case (\_, label) => label }.toSeq

// Count labels in collections

val labelCounts = labels.groupBy(identity).values.map(\_.size)

labels.size \* entropy(labelCounts)

}.collect()

// Average entropy weighted by cluster size

weightedClusterEntropy.sum / data.count()

}

(2 to 10 by 1).map(k => (k, clusteringScore4(data, k))).foreach(println)

/\*

(2,0.4620981203732969)

(3,0.37303245806738555)

(4,0.34083198433786593)

(5,0.33534496305711625)

(6,0.31018676754053165)

(7,0.3317133492276407)

(8,0.19068803427125436)

(9,0.26344491728695874)

(10,0.1696281418987278)

\*/

At this point its kinda confirmed to me what is going on. The data set is too sparse and the generated values made for versicolor and virginica may be too similar.

val pipelineModel = fitPipeline4(data, 2)

val countByClusterLabel = pipelineModel.transform(data).

select("cluster", "class").

groupBy("cluster", "class").count().

orderBy("cluster", "class")

countByClusterLabel.show()

/\*

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

|cluster| class|count|

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

| 0|Iris-versicolor| 50|

| 0| Iris-virginica| 50|

| 1| Iris-setosa| 50|

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

\*/

val pipelineModel = fitPipeline4(data, 3)

val countByClusterLabel = pipelineModel.transform(data).

select("cluster", "class").

groupBy("cluster", "class").count().

orderBy("cluster", "class")

countByClusterLabel.show()

/\*

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

|cluster| class|count|

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

| 0|Iris-versicolor| 50|

| 0| Iris-virginica| 50|

| 1| Iris-setosa| 37|

| 2| Iris-setosa| 13|

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

\*/

/\*

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

|cluster| class|count|

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

| 0| Iris-setosa| 50|

| 1|Iris-versicolor| 12|

| 1| Iris-virginica| 39|

| 2|Iris-versicolor| 38|

| 2| Iris-virginica| 11|

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

\*/

/\*

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

|cluster| class|count|

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

| 0|Iris-versicolor| 11|

| 0| Iris-virginica| 36|

| 1| Iris-setosa| 50|

| 2|Iris-versicolor| 39|

| 2| Iris-virginica| 14|

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

\*/

val pipelineModel = fitPipeline4(data, 6)

val countByClusterLabel = pipelineModel.transform(data).

select("cluster", "class").

groupBy("cluster", "class").count().

orderBy("cluster", "class")

countByClusterLabel.show()

/\*

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

|cluster| class|count|

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

| 0|Iris-versicolor| 22|

| 0| Iris-virginica| 15|

| 1| Iris-setosa| 35|

| 2| Iris-virginica| 12|

| 3|Iris-versicolor| 9|

| 3| Iris-virginica| 21|

| 4|Iris-versicolor| 19|

| 4| Iris-virginica| 2|

| 5| Iris-setosa| 15|

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

\*/

val pipelineModel = fitPipeline4(data, 9)

val countByClusterLabel = pipelineModel.transform(data).

select("cluster", "class").

groupBy("cluster", "class").count().

orderBy("cluster", "class")

countByClusterLabel.show()

/\*

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

|cluster| class|count|

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

| 0|Iris-versicolor| 19|

| 1| Iris-setosa| 27|

| 2| Iris-virginica| 9|

| 3| Iris-setosa| 23|

| 4|Iris-versicolor| 17|

| 4| Iris-virginica| 7|

| 5|Iris-versicolor| 4|

| 5| Iris-virginica| 14|

| 6|Iris-versicolor| 10|

| 6| Iris-virginica| 1|

| 7| Iris-virginica| 16|

| 8| Iris-virginica| 3|

+-------+-

As I mentioned before, It seems like setosa instances actually correlate well enough. However, it just looks like scaling the data in this case doesn’t do much. So I rewrote the function so it wouldn’t scale, and ended up actually working better!

def clusteringScore5(data: DataFrame, k: Int): Double = {

val pipelineModel = fitPipeline5(data, k)

// Predict cluster for each datum

val clusterLabel = pipelineModel.transform(data).

select("cluster", "class").as[(Int, String)]

val weightedClusterEntropy = clusterLabel.

// Extract collections of labels, per cluster

groupByKey { case (cluster, \_) => cluster }.

mapGroups { case (\_, clusterLabels) =>

val labels = clusterLabels.map { case (\_, label) => label }.toSeq

// Count labels in collections

val labelCounts = labels.groupBy(identity).values.map(\_.size)

labels.size \* entropy(labelCounts)

}.collect()

// Average entropy weighted by cluster size

weightedClusterEntropy.sum / data.count()

}

| Scaled | 3 clusters | Non scaled |
| --- | --- | --- |
| +-------+---------------+-----+  |cluster| class|count|  +-------+---------------+-----+  | 0| Iris-setosa| 50|  | 1|Iris-versicolor| 12|  | 1| Iris-virginica| 39|  | 2|Iris-versicolor| 38|  | 2| Iris-virginica| 11|  +-------+---------------+-----+ | vs | +-------+---------------+-----+  |cluster| class|count|  +-------+---------------+-----+  | 0| Iris-setosa| 50|  | 1|Iris-versicolor| 47|  | 1| Iris-virginica| 14|  | 2|Iris-versicolor| 3|  | 2| Iris-virginica| 36|  +-------+---------------+-----+ |

(2 to 10 by 1).map(k => (k, clusteringScore5(data, k))).foreach(println)

/\*

countByClusterLabel: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [cluster: int, class: string ... 1 more field]

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

|cluster| class|count|

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

| 0|Iris-versicolor| 47|

| 0| Iris-virginica| 50|

| 1| Iris-setosa| 50|

| 1|Iris-versicolor| 3|

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

pipelineModel: org.apache.spark.ml.PipelineModel = pipeline\_6c50ed69fb29

countByClusterLabel: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [cluster: int, class: string ... 1 more field]

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

|cluster| class|count|

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

| 0| Iris-setosa| 50|

| 1|Iris-versicolor| 47|

| 1| Iris-virginica| 14|

| 2|Iris-versicolor| 3|

| 2| Iris-virginica| 36|

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

pipelineModel: org.apache.spark.ml.PipelineModel = pipeline\_5e962b361224

countByClusterLabel: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [cluster: int, class: string ... 1 more field]

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

|cluster| class|count|

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

| 0|Iris-versicolor| 23|

| 0| Iris-virginica| 12|

| 1| Iris-setosa| 50|

| 2| Iris-virginica| 18|

| 3| Iris-virginica| 11|

| 4| Iris-virginica| 8|

| 5|Iris-versicolor| 27|

| 5| Iris-virginica| 1|

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

pipelineModel: org.apache.spark.ml.PipelineModel = pipeline\_c7bd8bef0fc3

countByClusterLabel: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [cluster: int, class: string ... 1 more field]

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

|cluster| class|count|

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

| 0|Iris-versicolor| 24|

| 0| Iris-virginica| 1|

| 1| Iris-setosa| 23|

| 2|Iris-versicolor| 21|

| 3| Iris-virginica| 12|

| 4| Iris-virginica| 3|

| 5| Iris-setosa| 27|

| 6|Iris-versicolor| 4|

| 6| Iris-virginica| 12|

| 7|Iris-versicolor| 1|

| 7| Iris-virginica| 13|

| 8| Iris-virginica| 9|

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

\*/

## Max iteration hyperparameter modification

import org.apache.spark.ml.{PipelineModel, Pipeline}

import org.apache.spark.ml.clustering.{KMeans, KMeansModel}

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

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

import org.apache.spark.sql.{DataFrame, SparkSession}

import scala.util.Random

val dataWithoutHeader = spark.read. //parse the data directly from csv by following these options

option("header", "false"). //keep header

option("inferSchema", "true"). //this means infer type for column

csv("iris2.csv") //this be where it stored

//If you try to read the csv in with headers, it will always say that variable we are dealing with is string

val data = dataWithoutHeader.toDF(

"sepal\_length", "sepal\_width", "petal\_length", "petal\_width", "class") //adding our column labels to our dataWithoutHeader

def fitPipeline5(data: DataFrame, k: Int): PipelineModel = {

val assembler = new VectorAssembler().

setInputCols(data.columns.filter(\_ != "class")).

setOutputCol("featureVector")

val kmeans = new KMeans().

setSeed(Random.nextLong()).

setK(3).

setPredictionCol("cluster").

setFeaturesCol("featureVector").

setMaxIter(k).

setTol(1.0e-6)

val pipeline = new Pipeline().setStages(

Array(assembler, kmeans))

pipeline.fit(data)

}

def clusteringScore5(data: DataFrame, k: Int): Double = {

val pipelineModel = fitPipeline5(data, k)

// Predict cluster for each datum

val clusterLabel = pipelineModel.transform(data).

select("cluster", "class").as[(Int, String)]

val weightedClusterEntropy = clusterLabel.

// Extract collections of labels, per cluster

groupByKey { case (cluster, \_) => cluster }.

mapGroups { case (\_, clusterLabels) =>

val labels = clusterLabels.map { case (\_, label) => label }.toSeq

// Count labels in collections

val labelCounts = labels.groupBy(identity).values.map(\_.size)

labels.size \* entropy(labelCounts)

}.collect()

// Average entropy weighted by cluster size

weightedClusterEntropy.sum / data.count()

}

(20 to 200 by 20).map(k => (k, clusteringScore5(data, k), fitPipeline5(data, 2).transform(data).

select("cluster", "class").

groupBy("cluster", "class").count().

orderBy("cluster", "class").show(false))).foreach(println)

/\*

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

|cluster|class |count|

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

|0 |Iris-versicolor|2 |

|0 |Iris-virginica |36 |

|1 |Iris-setosa |50 |

|2 |Iris-versicolor|48 |

|2 |Iris-virginica |14 |

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

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

|cluster|class |count|

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

|0 |Iris-versicolor|15 |

|0 |Iris-virginica |49 |

|1 |Iris-setosa |50 |

|2 |Iris-versicolor|35 |

|2 |Iris-virginica |1 |

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

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

|cluster|class |count|

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

|0 |Iris-setosa |50 |

|1 |Iris-virginica |34 |

|2 |Iris-versicolor|50 |

|2 |Iris-virginica |16 |

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

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

|cluster|class |count|

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

|0 |Iris-versicolor|48 |

|0 |Iris-virginica |14 |

|1 |Iris-setosa |50 |

|2 |Iris-versicolor|2 |

|2 |Iris-virginica |36 |

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

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

|cluster|class |count|

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

|0 |Iris-setosa |50 |

|1 |Iris-versicolor|48 |

|1 |Iris-virginica |14 |

|2 |Iris-versicolor|2 |

|2 |Iris-virginica |36 |

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

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

|cluster|class |count|

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

|0 |Iris-versicolor|45 |

|0 |Iris-virginica |9 |

|1 |Iris-setosa |50 |

|2 |Iris-versicolor|5 |

|2 |Iris-virginica |41 |

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

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

|cluster|class |count|

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

|0 |Iris-versicolor|48 |

|0 |Iris-virginica |14 |

|1 |Iris-setosa |50 |

|2 |Iris-versicolor|2 |

|2 |Iris-virginica |36 |

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

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

|cluster|class |count|

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

|0 |Iris-setosa |30 |

|1 |Iris-versicolor|46 |

|1 |Iris-virginica |50 |

|2 |Iris-setosa |20 |

|2 |Iris-versicolor|4 |

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

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

|cluster|class |count|

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

|0 |Iris-versicolor|42 |

|0 |Iris-virginica |5 |

|1 |Iris-setosa |50 |

|2 |Iris-versicolor|8 |

|2 |Iris-virginica |45 |

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

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

|cluster|class |count|

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

|0 |Iris-versicolor|46 |

|0 |Iris-virginica |13 |

|1 |Iris-setosa |50 |

|2 |Iris-versicolor|4 |

|2 |Iris-virginica |37 |

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

(20,0.2895730091214502,())

(40,0.2895730091214502,())

(60,0.273021191057774,())

(80,0.2895730091214502,())

(100,0.273021191057774,())

(120,0.273021191057774,())

(140,0.2895730091214502,())

(160,0.2895730091214502,())

(180,0.273021191057774,())

(200,0.2895730091214502,())

\*/

val pipelineModel = fitPipeline5(data, 80)

val countByClusterLabel = pipelineModel.transform(data).

select("cluster", "class").

groupBy("cluster", "class").count().

orderBy("cluster", "class")

countByClusterLabel.show()

/\*

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

|cluster|class |count|

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

|0 |Iris-versicolor|47 |

|0 |Iris-virginica |14 |

|1 |Iris-versicolor|3 |

|1 |Iris-virginica |36 |

|2 |Iris-setosa |50 |

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

\*/

Basically the simplest idea conveyed here is that the more iterations we have, the higher the chance we get a better clustering.

val pipelineModel = fitPipeline5(data, 1000)

val countByClusterLabel = pipelineModel.transform(data).

select("cluster", "class").

groupBy("cluster", "class").count().

orderBy("cluster", "class")

countByClusterLabel.show()

/\*

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

|cluster| class|count|

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

| 0|Iris-versicolor| 48|

| 0| Iris-virginica| 14|

| 1| Iris-setosa| 50|

| 2|Iris-versicolor| 2|

| 2| Iris-virginica| 36|

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

\*/

Even though this one iterated over 1000 times, there was no visible difference between this iteration and the 80 iterations. And even modifying the threshold to be 1-e-10 we end up with

/\*

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

|cluster| class|count|

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

| 0|Iris-versicolor| 3|

| 0| Iris-virginica| 36|

| 1| Iris-setosa| 50|

| 2|Iris-versicolor| 47|

| 2| Iris-virginica| 14|

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

\*/

Basically the only way we can end up with the best result is just getting lucky with a really good seed since our data is a bit too sparse. Below is the best run, at 160 iterations, with iterations 140, 180, and 200 all scoring the same

