## HU Extension Assignment 11 E63 Big Data Analytics

### Handed out: 04/16/2016 Due by 11:30PM EST on Friday, 04/22/2016

Please, describe every step of your work and present all intermediate and final results in a Word document. Please, copy past text version of all essential command and snippets of results into the Word document with explanations of the purpose of those commands. We cannot retype text that is in JPG images. Please, always submit a separate copy of the original, working scripts and/or class files you used. Sometimes we need to run your code and retyping is too costly. Please include in your MS Word document only relevant portions of the console output or output files. Sometime either console output or the result file is too long and including it into the MS Word document makes that document too hard to read. PLEASE DO NOT EMBED files into your MS Word document. For issues and comments visit the class Discussion Board. You are not obliged to use Java or Eclipse. You are welcome to use any language and any IDE of your choice.

**Problem 1.** Remove the header of the attached Samll\_Car\_Data.csv file and then import it into Spark. Randomly select 10% of you data for testing and use remaining data for training. Look initially at horsepower and displacement. Treat displacement as a feature and horsepower as the target variable. Use MLlib linear regression to identify the model for the relationship. Use test data to illustrate accuracy of your ability to predict the relationship. Create a diagram using D3 which presents the model (straight line), original test data and predictions of your analysis. Please label your axes and use different colors for original data and predicted data.

This is the Scala program I have for this Problem:

// create spark conf and spark context

**val** conf = **new** SparkConf().setMaster("local").setAppName("Assignment11")

**val** sc = **new** SparkContext(conf)

// take input and output file names from program parameters

**val** inputFileName = args(0)

**val** outputFileName = args(1)

// construct RDD from input file

**val** rawCarDataRDDWithHeaders = sc.textFile(inputFileName)

// get the headers row of input file

**val** headersRow = rawCarDataRDDWithHeaders.first()

// construct a headers map

**val** headersMap: *Map*[*String*, Int] = Map("Record\_num" -> 0, "Acceleration" -> 1,

"Cylinders" -> 2, "Displacement" -> 3, "Horsepower" -> 4, "Manufacturer" -> 5, "Model" -> 6,

"Model\_Year" -> 7, "MPG" -> 8, "Origin" -> 9, "Weight" -> 10)

// create an RDD of data without the headers

**val** uncleanCarDataRDDOfUnsplitLines = rawCarDataRDDWithHeaders.filter(rowAsAString => (rowAsAString != headersRow))

// split the data by comma

**val** uncleanCarDataRDD = uncleanCarDataRDDOfUnsplitLines.map(rowAsAString => rowAsAString.split(","))

// trim the data as it contains spaces

**val** cleanCarDataRDD = uncleanCarDataRDD.map(rowAsArrayOfValues => rowAsArrayOfValues.map(value => value.trim()))

// construct an entire data RDD, also filter the NaN values

**val** entireDataRDD = cleanCarDataRDD.map { rowAsArrayOfValues =>

**var** label = 0

**if** (rowAsArrayOfValues(headersMap("Horsepower")) != "NaN") {

label = rowAsArrayOfValues(headersMap("Horsepower")).toInt

}

**var** feature = 0.0

**if** (rowAsArrayOfValues(headersMap("Displacement")) != "NaN") {

feature = rowAsArrayOfValues(headersMap("Displacement")).toDouble

}

**LabeledPoint**(label, Vectors.dense(Array(feature)))

}

// cache the RDD

entireDataRDD.cache

// create indexed RDD in order to split it into test and training data

**val** entireDataRDDWithIdx = entireDataRDD.zipWithIndex().map(mapEntry => (mapEntry.\_2, mapEntry.\_1))

// create test and training data RDDs (these will be with ids)

**val** testDataRDDWithIdx = entireDataRDDWithIdx.sample(**false**, 0.1, 40)

**val** trainDataRDDWithIdx = entireDataRDDWithIdx.subtract(testDataRDDWithIdx)

// create test and training RDDs

**val** testDataRDD = testDataRDDWithIdx.map(mapEntry => mapEntry.\_2)

**val** trainingDataRDD = trainDataRDDWithIdx.map(mapEntry => mapEntry.\_2)

println("Training data size:" + trainingDataRDD.collect().size)

println("Test data size:" + testDataRDD.collect().size)

println()

// cache the training data RDD

trainingDataRDD.cache

// these are my settings for linear regression.

// I tried a lot with different settings but these were the ones that worked for me

**val** numIterationsForLR = 100000

**val** stepSizeForLR = 0.001

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Linear Regression with SGD model start \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**val** linearWithSGDTrainedModel = LinearRegressionWithSGD.train(trainingDataRDD, numIterationsForLR, stepSizeForLR)

**val** linearWithSGDPredictedVsActual = testDataRDD.map { testDataRow =>

(Math.round(linearWithSGDTrainedModel.predict(testDataRow.features)).toDouble, testDataRow.label)

}

println("Predicted vs Actual value of Horse Power for Linear Regression with SGD:")

linearWithSGDPredictedVsActual.collect().foreach(println)

**var** metrics = **new** RegressionMetrics(linearWithSGDPredictedVsActual)

println("Performance metrics for Linear Regression:")

printMetrics(metrics)

println()

**val** linearWithSGDOutputRDD = testDataRDD.map(testDataRow =>

(testDataRow.features(0), Math.round(linearWithSGDTrainedModel.predict(testDataRow.features)).toDouble, testDataRow.label)).saveAsTextFile(outputFileName)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Linear Regression with SGD model end \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

I then made a build using sbt tool:

rpulekar-m1:big-data-analytics-harvard rpulekar$ cd scala\_workspace\_for\_course/Assignment11 && sbt clean package && cd ../..

[info] Loading global plugins from /Users/rpulekar/.sbt/0.13/plugins

[info] Set current project to Assignment11 (in build file:/Users/rpulekar/work/big-data-analytics-harvard/scala\_workspace\_for\_course/Assignment11/)

[success] Total time: 0 s, completed Apr 22, 2016 7:42:16 PM

[info] Updating {file:/Users/rpulekar/work/big-data-analytics-harvard/scala\_workspace\_for\_course/Assignment11/}assignment11...

[info] Resolving org.fusesource.jansi#jansi;1.4 ...

[info] Done updating.

[info] Compiling 6 Scala sources to /Users/rpulekar/work/big-data-analytics-harvard/scala\_workspace\_for\_course/Assignment11/target/scala-2.10/classes...

[warn] Multiple main classes detected. Run 'show discoveredMainClasses' to see the list

[info] Packaging /Users/rpulekar/work/big-data-analytics-harvard/scala\_workspace\_for\_course/Assignment11/target/scala-2.10/assignment11\_2.10-0.0.1.jar ...

[info] Done packaging.

[success] Total time: 14 s, completed Apr 22, 2016 7:42:30 PM

rpulekar-m1:big-data-analytics-harvard rpulekar$

Then I scp ed over the file to my VM:

rpulekar-m1:big-data-analytics-harvard rpulekar$ scp -i private\_keys/cloudera\_id\_dsa scala\_workspace\_for\_course/Assignment11/target/scala-2.10/assignment11\_2.10-0.0.1.jar cloudera@192.168.71.158:~/shared/

assignment11\_2.10-0.0.1.jar 100% 119KB 119.4KB/s 00:00

rpulekar-m1:big-data-analytics-harvard rpulekar$

Then I ran the program in spark:

[cloudera@localhost ~]$ spark-submit --class e63.mllib.assignment11.Problem1 shared/assignment11\_2.10-0.0.1.jar file:///home/cloudera/assignment11/input/Small\_Car\_Data.csv file:///home/cloudera/assignment11/output/Problem1linearWithSGDOutput

SLF4J: Class path contains multiple SLF4J bindings.

SLF4J: Found binding in [jar:file:/usr/lib/zookeeper/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]

SLF4J: Found binding in [jar:file:/usr/lib/flume-ng/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]

SLF4J: See http://www.slf4j.org/codes.html#multiple\_bindings for an explanation.

SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]

16/04/22 16:48:46 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable

16/04/22 16:48:47 WARN Utils: Your hostname, localhost.localdomain resolves to a loopback address: 127.0.0.1; using 192.168.71.158 instead (on interface eth0)

16/04/22 16:48:47 WARN Utils: Set SPARK\_LOCAL\_IP if you need to bind to another address

16/04/22 16:48:48 WARN MetricsSystem: Using default name DAGScheduler for source because spark.app.id is not set.

16/04/22 16:48:51 WARN DomainSocketFactory: The short-circuit local reads feature cannot be used because libhadoop cannot be loaded.

Training data size:89

Test data size:11

16/04/22 16:48:52 WARN BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeSystemBLAS

16/04/22 16:48:52 WARN BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeRefBLAS

Predicted vs Actual value of Horse Power for Linear Regression with SGD:

(220.0,215.0)

(227.0,225.0)

(56.0,95.0)

(125.0,105.0)

(116.0,90.0)

(49.0,60.0)

(78.0,108.0)

(84.0,120.0)

(175.0,145.0)

(60.0,79.0)

(59.0,82.0)

Performance metrics for Linear Regression:

MSE = 612.0909090909091

RMSE = 24.740471076576313

R-squared = 0.7701462984687385

MAE = 21.90909090909091

Explained variance = 4007.826446280992

Performance metrics for Random Forest:

MSE = 392.27272727272725

RMSE = 19.8058760794045

R-squared = 0.8526928973552067

MAE = 17.363636363636363

Explained variance = 1969.1487603305782

Performance metrics for Gradient Boosted:

MSE = 237.54545454545453

RMSE = 15.412509677059559

R-squared = 0.910796417332365

MAE = 14.272727272727273

Explained variance = 2453.909090909091

Performance metrics for Naive Bayes Model:

MSE = 3710.3636363636365

RMSE = 60.91275429960163

R-squared = -0.39332377458739143

MAE = 40.18181818181818

Explained variance = 1047.4049586776857

I also tried Random Forest, Gradient Boost and Naïve Bayes models.

The program gave following output file for Linear Regression Model:

[cloudera@localhost Problem1linearWithSGDOutput]$ pwd

/home/cloudera/assignment11/output/Problem1linearWithSGDOutput

[cloudera@localhost Problem1linearWithSGDOutput]$ ls

part-00000 \_SUCCESS

[cloudera@localhost Problem1linearWithSGDOutput]$

I renamed the output file:

[cloudera@localhost Problem1linearWithSGDOutput]$ mv part-00000 problem1\_original\_vs\_predicted.csv

[cloudera@localhost Problem1linearWithSGDOutput]$ vi problem1\_original\_vs\_predicted.csv

I added headers to that output file and removed brackets.

The output file which has Displacement, HorsePowerPredicted, HorsePowerActual (prediction done with Linear Regression) was then like this:

Displacement,HorsePowerPredicted,HorsePowerActual

440.0,220.0,215.0

455.0,227.0,225.0

113.0,56.0,95.0

250.0,125.0,105.0

232.0,116.0,90.0

98.0,49.0,60.0

156.0,78.0,108.0

168.0,84.0,120.0

350.0,175.0,145.0

120.0,60.0,79.0

119.0,59.0,82.0

I wrote an html file that uses d3 library:

<script>

**var** margin = {

top : 20,

right : 80,

bottom : 30,

left : 50

}, width = 960 - margin.left - margin.right, height = 500 - margin.top

- margin.bottom;

**var** x = d3.scale.linear().range([ 0, width ]);

**var** y = d3.scale.linear().range([ height, 0 ]);

**var** color = d3.scale.category10();

**var** xAxis = d3.svg.axis().scale(x).orient("bottom");

**var** yAxis = d3.svg.axis().scale(y).orient("left");

**var** line = d3.svg.line().interpolate("basis").x(**function**(d) {

**return** x(d.Displacement);

}).y(**function**(d) {

**return** y(d.HorsePower);

});

**var** svg = d3.select("body").append("svg").attr("width",

width + margin.left + margin.right).attr("height",

height + margin.top + margin.bottom).append("g").attr(

"transform",

"translate(" + margin.left + "," + margin.top + ")");

d3.csv("../input\_files/problem1\_original\_vs\_predicted.csv", **function**(error, data) {

**if** (error)

**throw** error;

color.domain(d3.keys(data[0]).filter(**function**(key) {

**return** key !== "Displacement";

}));

data.forEach(**function**(d) {

d.Displacement = +d.Displacement;

});

**var** horsePowerValueTypes = color.domain().map(**function**(name) {

**return** {

name : name,

values : data.map(**function**(d) {

**return** {

Displacement : d.Displacement,

HorsePower : +d[name]

};

})

};

});

x.domain(d3.extent(data, **function**(d) {

**return** d.Displacement;

}));

y.domain([ d3.min(horsePowerValueTypes, **function**(c) {

**return** d3.min(c.values, **function**(v) {

**return** v.HorsePower;

});

}), d3.max(horsePowerValueTypes, **function**(c) {

**return** d3.max(c.values, **function**(v) {

**return** v.HorsePower;

});

}) ]);

svg.append("g").attr("class", "x axis").attr("transform",

"translate(0," + height + ")").call(xAxis)

.append("text").attr("class", "label").attr("x", width)

.attr("y", -6).style("text-anchor", "end").text(

"Displacement").style("font-size","15px");

svg.append("g").attr("class", "y axis").call(yAxis).append("text")

.attr("transform", "rotate(-90)").attr("y", 6).attr("dy",

".71em").style("text-anchor", "end").text(

"Horse Power").style("font-size","15px");

**var** horsePowerValueType = svg.selectAll(".horsePowerValueType").data(horsePowerValueTypes).enter().append("g")

.attr("class", "horsePowerValueType");

horsePowerValueType.append("path").attr("class", "line").attr("d", **function**(d) {

**return** line(d.values);

}).style("stroke", **function**(d) {

**return** color(d.name);

});

horsePowerValueType.append("text").datum(**function**(d) {

**return** {

name : d.name,

value : d.values[d.values.length - 1]

};

}).style("fill", **function**(d) {

**return** color(d.name);

}).attr(

"transform",

**function**(d) {

**return** "translate(" + x(d.value.Displacement) + ","

+ y(d.value.HorsePower) + ")";

}).attr("x", 3).attr("y", ".35em").text(**function**(d) {

**return** d.name;

});

});

</script>

I have python server running:

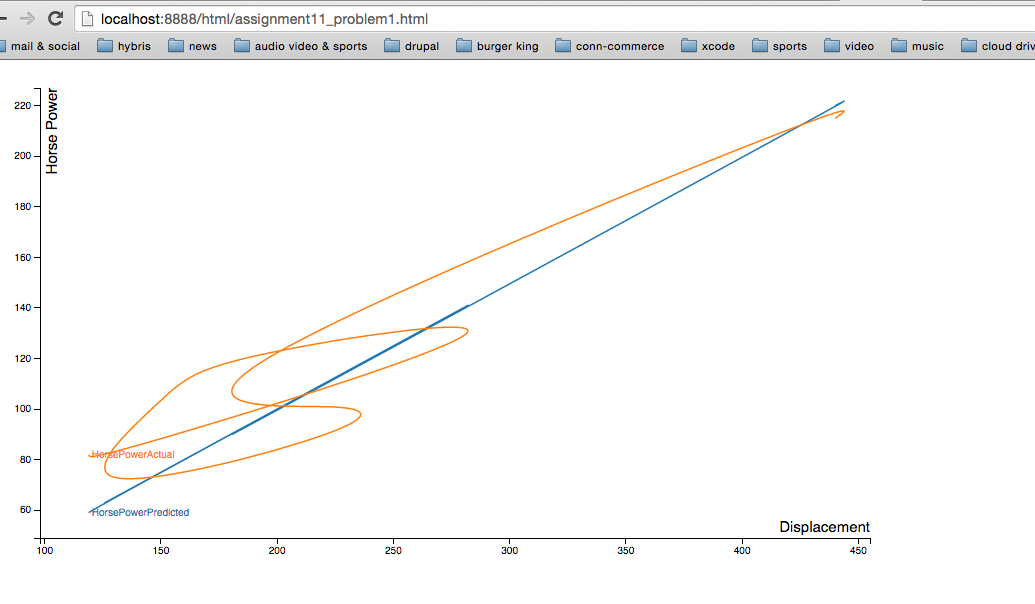
rpulekar-m1:WebContent rpulekar$ python -m SimpleHTTPServer 8888 &

[2] 39832

Then I accessed the output of html file in the web browser:

<http://localhost:8888/html/assignment11_problem1.html>

and I got this:



Deliverables:

* Problem1.scala (the scala program for this problem)
* problem1\_original\_vs\_predicted.csv (the csv file containing displacement, predicted horsepower and actual horsepower)
* RohanPulekar\_Assignment11\_Porblem1.zip (a zip file containing html/ assignment11\_problem1.html and input\_files/ problem1\_original\_vs\_predicted.csv)

**Problem 2**. Treat: cylinders, displacement, manufacturer, model\_year, origin and weight as features and use linear regression to predict two target variable: horsepower and acceleration. Please note that some of those are categorical variables. Use test data to assess quality of prediction for both target variables. Which of two target variables is easier to predict, in the sense that predicted values differ less from the original values.

For this, I treated manufacturer and origin as categories.

I wrote two programs:

* one for calculating horsepower
* one for calculating acceleration

This is the program for predicting horsepower:

**def** main(args: Array[*String*]) {

// create spark conf and spark context

**val** conf = **new** SparkConf().setAppName("Assignment11-Problem2ForHorsePower")

**val** sc = **new** SparkContext(conf)

// take input file name from program parameter

**val** inputFileName = args(0)

// construct RDD from input file

**val** rawCarDataRDDWithHeaders = sc.textFile(inputFileName)

// get the headers row of input file

**val** headersRow = rawCarDataRDDWithHeaders.first()

// construct a headers map

**val** headersMap: Map[*String*, Int] = Map("Record\_num" -> 0, "Acceleration" -> 1,

"Cylinders" -> 2, "Displacement" -> 3, "Horsepower" -> 4, "Manufacturer" -> 5, "Model" -> 6,

"Model\_Year" -> 7, "MPG" -> 8, "Origin" -> 9, "Weight" -> 10)

// create an RDD of data without the headers

**val** uncleanCarDataRDDOfUnsplitLines = rawCarDataRDDWithHeaders.filter(rowAsAString => (rowAsAString != headersRow))

// split the data by comma

**val** uncleanCarDataRDD = uncleanCarDataRDDOfUnsplitLines.map(rowAsAString => rowAsAString.split(","))

// trim the data as it contains spaces

**val** cleanCarDataRDD = uncleanCarDataRDD.map(rowAsArrayOfValues => rowAsArrayOfValues.map(value => value.trim()))

// create category features

**val** manufacturerCategoriesRDD = cleanCarDataRDD.map(rowAsArrayOfValues => rowAsArrayOfValues(headersMap("Manufacturer"))).distinct().collect

**val** originCategoriesRDD = cleanCarDataRDD.map(rowAsArrayOfValues => rowAsArrayOfValues(headersMap("Origin"))).distinct().collect

// create categories map

**val** categoriesMap = manufacturerCategoriesRDD.union(originCategoriesRDD).zipWithIndex.toMap

**val** numberOfCategories = categoriesMap.size

println("Number of categories:" + numberOfCategories)

// construct the entire data RDD, also filter the NaN values

**val** dataRDDForHorsePower = cleanCarDataRDD.map { rowAsArrayOfValues =>

**var** label = 0.0

**if** (rowAsArrayOfValues(headersMap("Horsepower")) != "NaN") {

label = rowAsArrayOfValues(headersMap("Horsepower")).toInt

}

**val** categoryFeatures = Array.ofDim[Double](numberOfCategories)

**val** manufacturerCategoryIdx = categoriesMap(rowAsArrayOfValues(headersMap("Manufacturer")))

categoryFeatures(manufacturerCategoryIdx) = 1.0

**val** originCategoryIdx = categoriesMap(rowAsArrayOfValues(headersMap("Origin")))

categoryFeatures(originCategoryIdx) = 1.0

**val** nonCategoryFeatures = rowAsArrayOfValues.slice(2, 3).union(rowAsArrayOfValues.slice(3, 4)).union(rowAsArrayOfValues.slice(7, 8)).map(feature => feature.toDouble)

**val** features = categoryFeatures ++ nonCategoryFeatures

**LabeledPoint**(label, Vectors.dense(features))

}

// cache the RDD

dataRDDForHorsePower.cache

// create indexed RDD in order to split it into test and training data

**val** dataRDDForHorsePowerWithIdx = dataRDDForHorsePower.zipWithIndex().map(mapEntry => (mapEntry.\_2, mapEntry.\_1))

// create test and training data RDDs (these will be with ids)

**val** testDataRDDForHorsePowerWithIdx = dataRDDForHorsePowerWithIdx.sample(**false**, 0.1, 40)

**val** trainDataRDDForHorsePowerWithIdx = dataRDDForHorsePowerWithIdx.subtract(testDataRDDForHorsePowerWithIdx)

// create test and training RDDs

**val** testDataRDDForHorsePower = testDataRDDForHorsePowerWithIdx.map(mapEntry => mapEntry.\_2)

**val** trainingDataRDDForHorsePower = trainDataRDDForHorsePowerWithIdx.map(mapEntry => mapEntry.\_2)

println("Training data size:" + trainingDataRDDForHorsePower.collect().size)

println("Test data size:" + testDataRDDForHorsePower.collect().size)

println()

// cache the training data RDD

trainingDataRDDForHorsePower.cache

// these are my settings for linear regression.

// I tried a lot with different settings but these were the ones that worked for me

**val** numIterationsForLR = 10000

**val** stepSizeForLR = 0.0001

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Linear Regression with SGD model for Horse Power start \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**val** linearSGDTrainedModelForHorsePower = LinearRegressionWithSGD.train(trainingDataRDDForHorsePower, numIterationsForLR, stepSizeForLR)

**val** linearSGDPredictedVsActualForHorsePower = testDataRDDForHorsePower.map { testDataRow =>

(Math.round(linearSGDTrainedModelForHorsePower.predict(testDataRow.features)).toDouble, testDataRow.label)

}

println("Predicted vs Actual value of Horse Power for Linear Regression with SGD")

linearSGDPredictedVsActualForHorsePower.collect().foreach(println)

**var** metrics = **new** RegressionMetrics(linearSGDPredictedVsActualForHorsePower)

println("Performance metrics for Linear Regression with SGD Model for Horse Power:")

printMetrics(metrics)

println()

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Linear Regression with SGD model for Horse Power end \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// I was unable to scale the features with StandardScaler

// If do that the linear regression model gives bad results. So I decided not to

// I was unable to use SVMWithSGD and LogisticRegressionWithSGD since label is not a boolean (0 or 1)

}

For Acceleration, program is similar except for reading from acceleration column.

I ran the spark job for horsepower:

[cloudera@localhost ~]$ spark-submit --class e63.mllib.assignment11.Problem2ForHorsePower shared/assignment11\_2.10-0.0.1.jar file:///home/cloudera/assignment11/input/Small\_Car\_Data.csv

SLF4J: Class path contains multiple SLF4J bindings.

SLF4J: Found binding in [jar:file:/usr/lib/zookeeper/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]

SLF4J: Found binding in [jar:file:/usr/lib/flume-ng/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]

SLF4J: See http://www.slf4j.org/codes.html#multiple\_bindings for an explanation.

SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]

16/04/22 17:06:59 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable

16/04/22 17:07:00 WARN Utils: Your hostname, localhost.localdomain resolves to a loopback address: 127.0.0.1; using 192.168.71.158 instead (on interface eth0)

16/04/22 17:07:00 WARN Utils: Set SPARK\_LOCAL\_IP if you need to bind to another address

16/04/22 17:07:01 WARN MetricsSystem: Using default name DAGScheduler for source because spark.app.id is not set.

16/04/22 17:07:04 WARN DomainSocketFactory: The short-circuit local reads feature cannot be used because libhadoop cannot be loaded.

Number of categories:34

Training data size:89

Test data size:11

16/04/22 17:07:06 WARN BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeSystemBLAS

16/04/22 17:07:06 WARN BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeRefBLAS

Predicted vs Actual value of Horse Power for Linear Regression with SGD

(201.0,215.0)

(207.0,225.0)

(70.0,95.0)

(127.0,105.0)

(120.0,90.0)

(66.0,60.0)

(89.0,108.0)

(94.0,120.0)

(167.0,145.0)

(76.0,79.0)

(76.0,82.0)

Performance metrics for Linear Regression with SGD Model for Horse Power:

MSE = 375.5454545454546

RMSE = 19.378995189262383

R-squared = 0.8589743589743589

MAE = 17.363636363636363

Explained variance = 2484.9173553719006

Performance metrics for Random Forest Model for Horse Power:

MSE = 397.81818181818176

RMSE = 19.945379961740056

R-squared = 0.8506104562749444

MAE = 15.090909090909092

Explained variance = 1623.603305785124

Performance metrics for Gradient Boosted Model for Horse Power:

MSE = 195.0

RMSE = 13.96424004376894

R-squared = 0.9267731784071653

MAE = 12.636363636363637

Explained variance = 3349.1487603305777

[cloudera@localhost ~]$

I also used Random Forest and Gradient Boost models

Then I ran the job for Acceleration:

[cloudera@localhost ~]$ spark-submit --class e63.mllib.assignment11.Problem2ForAcceleration shared/assignment11\_2.10-0.0.1.jar file:///home/cloudera/assignment11/input/Small\_Car\_Data.csv

SLF4J: Class path contains multiple SLF4J bindings.

SLF4J: Found binding in [jar:file:/usr/lib/zookeeper/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]

SLF4J: Found binding in [jar:file:/usr/lib/flume-ng/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]

SLF4J: See http://www.slf4j.org/codes.html#multiple\_bindings for an explanation.

SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]

16/04/22 17:09:58 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable

16/04/22 17:09:59 WARN Utils: Your hostname, localhost.localdomain resolves to a loopback address: 127.0.0.1; using 192.168.71.158 instead (on interface eth0)

16/04/22 17:09:59 WARN Utils: Set SPARK\_LOCAL\_IP if you need to bind to another address

16/04/22 17:10:01 WARN MetricsSystem: Using default name DAGScheduler for source because spark.app.id is not set.

16/04/22 17:10:03 WARN DomainSocketFactory: The short-circuit local reads feature cannot be used because libhadoop cannot be loaded.

Number of categories:34

Training data size:89

Test data size:11

16/04/22 17:10:05 WARN BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeSystemBLAS

16/04/22 17:10:05 WARN BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeRefBLAS

Predicted vs Actual value of Acceleration for Linear Regression with SGD:

(13.0,8.5)

(13.0,10.0)

(13.0,15.0)

(14.0,14.5)

(14.0,17.6)

(14.0,22.1)

(14.0,15.5)

(14.0,16.7)

(14.0,12.0)

(15.0,18.6)

(15.0,19.4)

Performance metrics for Linear Regression with SGD Model for Acceleration:

MSE = 14.357272727272733

RMSE = 3.7890991973386936

R-squared = 0.05401269862014113

MAE = 3.263636363636364

Explained variance = 2.8066942148760323

Performance metrics for Random Forest Model for Acceleration:

MSE = 3.9390909090909108

RMSE = 1.9847143142253272

R-squared = 0.7404569760730115

MAE = 1.5909090909090913

Explained variance = 6.6116528925619855

Performance metrics for Gradient Boosted Model for Acceleration:

MSE = 3.6300000000000012

RMSE = 1.9052558883257653

R-squared = 0.7608226876204789

MAE = 1.5363636363636366

Explained variance = 9.269504132231408

[cloudera@localhost ~]$

I also used Random Forest and Gradient Boost models

I think Acceleration is easier to predict than HorsePower.

Because mean absolute error during prediction of Acceleration is lower than that during prediction of HorsePower

Deliverables:

- Problem2ForHorsePower.scala (the scala program for predicting horse power using linear regression)

- Problem2ForAcceleration.scala (the scala program for predicting acceleration using linear regression)

**Problem 3**. Repeat above analysis with decision tree method. Compare predicting ability/quality of this technique with that of the linear regression.

For this I used DecisionTree class of Scala mllib.

Again I used two Scala files:

* one to prodict HorsePower
* one to predict Acceleration

This is the code for predicting horsepower:

**def** main(args: Array[*String*]) {

// create spark conf and spark context

**val** conf = **new** SparkConf().setAppName("Assignment11-Problem2ForHorsePower")

**val** sc = **new** SparkContext(conf)

// take input file name from program parameter

**val** inputFileName = args(0)

// construct RDD from input file

**val** rawCarDataRDDWithHeaders = sc.textFile(inputFileName)

// get the headers row of input file

**val** headersRow = rawCarDataRDDWithHeaders.first()

// construct a headers map

**val** headersMap: Map[*String*, Int] = Map("Record\_num" -> 0, "Acceleration" -> 1,

"Cylinders" -> 2, "Displacement" -> 3, "Horsepower" -> 4, "Manufacturer" -> 5, "Model" -> 6,

"Model\_Year" -> 7, "MPG" -> 8, "Origin" -> 9, "Weight" -> 10)

// create an RDD of data without the headers

**val** uncleanCarDataRDDOfUnsplitLines = rawCarDataRDDWithHeaders.filter(rowAsAString => (rowAsAString != headersRow))

// split the data by comma

**val** uncleanCarDataRDD = uncleanCarDataRDDOfUnsplitLines.map(rowAsAString => rowAsAString.split(","))

// trim the data as it contains spaces

**val** cleanCarDataRDD = uncleanCarDataRDD.map(rowAsArrayOfValues => rowAsArrayOfValues.map(value => value.trim()))

// create category features

**val** manufacturerCategoriesRDD = cleanCarDataRDD.map(rowAsArrayOfValues => rowAsArrayOfValues(headersMap("Manufacturer"))).distinct().collect

**val** originCategoriesRDD = cleanCarDataRDD.map(rowAsArrayOfValues => rowAsArrayOfValues(headersMap("Origin"))).distinct().collect

// create categories map

**val** categoriesMap = manufacturerCategoriesRDD.union(originCategoriesRDD).zipWithIndex.toMap

**val** numberOfCategories = categoriesMap.size

println("Number of categories:" + numberOfCategories)

// construct the entire data RDD, also filter the NaN values

**val** dataRDDForHorsePower = cleanCarDataRDD.map { rowAsArrayOfValues =>

**var** label = 0.0

**if** (rowAsArrayOfValues(headersMap("Horsepower")) != "NaN") {

label = rowAsArrayOfValues(headersMap("Horsepower")).toInt

}

**val** categoryFeatures = Array.ofDim[Double](numberOfCategories)

**val** manufacturerCategoryIdx = categoriesMap(rowAsArrayOfValues(headersMap("Manufacturer")))

categoryFeatures(manufacturerCategoryIdx) = 1.0

**val** originCategoryIdx = categoriesMap(rowAsArrayOfValues(headersMap("Origin")))

categoryFeatures(originCategoryIdx) = 1.0

**val** nonCategoryFeatures = rowAsArrayOfValues.slice(2, 3).union(rowAsArrayOfValues.slice(3, 4)).union(rowAsArrayOfValues.slice(7, 8)).union(rowAsArrayOfValues.slice(10, 11)).map(feature => **if** (feature == "NaN") 0.0 **else** feature.toDouble)

**val** features = categoryFeatures ++ nonCategoryFeatures

**LabeledPoint**(label, Vectors.dense(features))

}

// cache the RDD

dataRDDForHorsePower.cache

**val** features = dataRDDForHorsePower.map(labeledPoint => labeledPoint.features)

**val** featuresMatrix = **new** RowMatrix(features)

**val** featuresMatrixSummary = featuresMatrix.computeColumnSummaryStatistics()

**val** scaler = **new** StandardScaler(withMean = **true**, withStd = **true**).fit(features)

**val** scaledDataRDDForHorsePower = dataRDDForHorsePower.map(lp => **LabeledPoint**(lp.label, scaler.transform(lp.features)))

// create indexed RDD in order to split it into test and training data

**val** dataRDDForHorsePowerWithIdx = scaledDataRDDForHorsePower.zipWithIndex().map(mapEntry => (mapEntry.\_2, mapEntry.\_1))

// create test and training data RDDs (these will be with ids)

**val** testDataRDDForHorsePowerWithIdx = dataRDDForHorsePowerWithIdx.sample(**false**, 0.1, 40)

**val** trainDataRDDForHorsePowerWithIdx = dataRDDForHorsePowerWithIdx.subtract(testDataRDDForHorsePowerWithIdx)

// create test and training RDDs

**val** testDataRDDForHorsePower = testDataRDDForHorsePowerWithIdx.map(mapEntry => mapEntry.\_2)

**val** trainingDataRDDForHorsePower = trainDataRDDForHorsePowerWithIdx.map(mapEntry => mapEntry.\_2)

println("Training data size:" + trainingDataRDDForHorsePower.collect().size)

println("Test data size:" + testDataRDDForHorsePower.collect().size)

println()

// cache the training data RDD

trainingDataRDDForHorsePower.cache

// these are my settings for decision tree.

// I tried a lot with different settings but these were the ones that worked for me

**val** maxDepth = 5

**val** maxBins = 9

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Decision Tree model for Horse Power start \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**val** dTreeTrainedModelForHorsePower = DecisionTree.trainRegressor(trainingDataRDDForHorsePower, Map[Int, Int](), "variance", maxDepth, maxBins)

**val** dTreePredictedVsActualForHorsePower = testDataRDDForHorsePower.map { testDataRow =>

(Math.round(dTreeTrainedModelForHorsePower.predict(testDataRow.features)).toDouble, testDataRow.label)

}

println("Predicted vs Actual value of Horse Power for Decision Tree model")

dTreePredictedVsActualForHorsePower.collect().foreach(println)

println()

**var** metrics = **new** RegressionMetrics(dTreePredictedVsActualForHorsePower)

println("Performance metrics for Decision Tree Model for Horse Power:")

printMetrics(metrics)

println()

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Decision Tree model for Horse Power end \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

}

For acceleration, program is the same except for reading from Acceleration column

This is the how I ran the program for HorsePower:

[cloudera@localhost ~]$ spark-submit --class e63.mllib.assignment11.Problem3ForHorsePower shared/assignment11\_2.10-0.0.1.jar file:///home/cloudera/assignment11/input/Small\_Car\_Data.csv

SLF4J: Class path contains multiple SLF4J bindings.

SLF4J: Found binding in [jar:file:/usr/lib/zookeeper/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]

SLF4J: Found binding in [jar:file:/usr/lib/flume-ng/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]

SLF4J: See http://www.slf4j.org/codes.html#multiple\_bindings for an explanation.

SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]

16/04/22 17:18:57 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable

16/04/22 17:18:57 WARN Utils: Your hostname, localhost.localdomain resolves to a loopback address: 127.0.0.1; using 192.168.71.158 instead (on interface eth0)

16/04/22 17:18:57 WARN Utils: Set SPARK\_LOCAL\_IP if you need to bind to another address

16/04/22 17:18:59 WARN MetricsSystem: Using default name DAGScheduler for source because spark.app.id is not set.

16/04/22 17:19:01 WARN DomainSocketFactory: The short-circuit local reads feature cannot be used because libhadoop cannot be loaded.

Number of categories:40

Training data size:89

Test data size:11

Predicted vs Actual value of Horse Power for Decision Tree model

(207.0,215.0)

(207.0,225.0)

(86.0,95.0)

(94.0,105.0)

(96.0,90.0)

(65.0,60.0)

(94.0,108.0)

(94.0,120.0)

(150.0,145.0)

(86.0,79.0)

(94.0,82.0)

Performance metrics for Decision Tree Model for Horse Power:

MSE = 158.27272727272728

RMSE = 12.58064892097094

R-squared = 0.9405650832666083

MAE = 11.0

Explained variance = 2244.603305785124

[cloudera@localhost ~]$

This is the how I ran the program for Acceleration:

[cloudera@localhost ~]$ spark-submit --class e63.mllib.assignment11.Problem3ForAcceleration shared/assignment11\_2.10-0.0.1.jar file:///home/cloudera/assignment11/input/Small\_Car\_Data.csv

SLF4J: Class path contains multiple SLF4J bindings.

SLF4J: Found binding in [jar:file:/usr/lib/zookeeper/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]

SLF4J: Found binding in [jar:file:/usr/lib/flume-ng/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]

SLF4J: See http://www.slf4j.org/codes.html#multiple\_bindings for an explanation.

SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]

16/04/22 17:21:52 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable

16/04/22 17:21:52 WARN Utils: Your hostname, localhost.localdomain resolves to a loopback address: 127.0.0.1; using 192.168.71.158 instead (on interface eth0)

16/04/22 17:21:52 WARN Utils: Set SPARK\_LOCAL\_IP if you need to bind to another address

16/04/22 17:21:54 WARN MetricsSystem: Using default name DAGScheduler for source because spark.app.id is not set.

16/04/22 17:21:56 WARN DomainSocketFactory: The short-circuit local reads feature cannot be used because libhadoop cannot be loaded.

Number of categories:40

Training data size:89

Test data size:11

Predicted vs Actual value of Acceleration for Decision Tree model

(11.0,8.5)

(11.0,10.0)

(17.0,15.0)

(22.0,14.5)

(17.0,17.6)

(22.0,22.1)

(15.0,15.5)

(17.0,16.7)

(14.0,12.0)

(15.0,18.6)

(17.0,19.4)

Performance metrics for Decision Tree Model for Acceleration:

MSE = 8.266363636363637

RMSE = 2.875128455628311

R-squared = 0.45533701440846863

MAE = 2.045454545454546

Explained variance = 12.509173553719007

[cloudera@localhost ~]$

Comparison between Linear regression and decision tree methods:

Linear regression is good when predicting continuous data. Decision tree is good when predicting classified data (e.g. when value target/label can be one of N values. N can be 2,3,4,…)

Decision trees try to create a classification based model of the data. i.e. they create buckets/subbuckets. The number/depth of these buckets/subbuckets depends on the standard deviation for each bucket. If standard deviation is big, more subbuckets are created until standard deviation is small enough. Smaller the standard deviation, more accurate the prediction.

Linear regression works by forming a straight line between the data points. That straight line should represent the data as closely as possible.

Deliverables:

* Problem3ForHorsePower.scala (scala file for predicting horse power using decision tree)
* Problem3ForAcceleration.scala (scala file for predicting acceleration using decision tree)