**Big Data Spring 2018**

**Project 5**

Nicolas Eldering

Yifan Chen

**Member 1: Nicolas Eldering**  
My teammate and I agree that I handled 50% of the overall project. My specific tasks included:

* Task 1: worked on half the code and documentation

**Member 2: Yifan Chen**  
My teammate and I agree that I handled 50% of the overall project. My specific tasks included:

* Task 1: worked on half the code and documentation

**Some concepts:**

1. Pipeline: Pipeline is used to tie the stages together and feed each stage to each other in order. It consists of a sequence of stages, each of which is either an Estimator or a Transformer.

2. pipeline.fit(): If a stage is Estimator, its Estimator.fit() method will be called on the input dataset to fit a model. If a stage is a Transformer, its Transformer.transform() method will be called to produce the dataset for next stage.

3. StringIndexer: it’s used to encode labels to label indices.

4. VectorAssembler: we use VectorAssembler to combine all the feature columns into a single vector column. Because in pipeline, it can only handle the vector type features.

5. BinaryClassificationEvaluator: Because the label column only has two elements, 0 and 1. If more than two, we use MultiClassificationEvaluator.

// Databricks notebook source

/\*

\* import creditcard info from csv file into DataFrame

\*/

// COMMAND ----------

import org.apache.spark.sql.types.{StructType,StructField,IntegerType,DoubleType,StringType}

val fileAddress = "/FileStore/tables/creditcard.csv"

// Becasue null are not meaningful for ML algorithms and cannot be represented using scala.Double

val nullable = false

val schemaArray = Array(

StructField("Time", DoubleType, nullable),

StructField("V1", DoubleType, nullable),

StructField("V2", DoubleType, nullable),

StructField("V3", DoubleType, nullable),

StructField("V4", DoubleType, nullable),

StructField("V5", DoubleType, nullable),

StructField("V6", DoubleType, nullable),

StructField("V7", DoubleType, nullable),

StructField("V8", DoubleType, nullable),

StructField("V9", DoubleType, nullable),

StructField("V10", DoubleType, nullable),

StructField("V11", DoubleType, nullable),

StructField("V12", DoubleType, nullable),

StructField("V13", DoubleType, nullable),

StructField("V14", DoubleType, nullable),

StructField("V15", DoubleType, nullable),

StructField("V16", DoubleType, nullable),

StructField("V17", DoubleType, nullable),

StructField("V18", DoubleType, nullable),

StructField("V19", DoubleType, nullable),

StructField("V20", DoubleType, nullable),

StructField("V21", DoubleType, nullable),

StructField("V22", DoubleType, nullable),

StructField("V23", DoubleType, nullable),

StructField("V24", DoubleType, nullable),

StructField("V25", DoubleType, nullable),

StructField("V26", DoubleType, nullable),

StructField("V27", DoubleType, nullable),

StructField("V28", DoubleType, nullable),

StructField("Amount", DoubleType, nullable),

StructField("Class", DoubleType, nullable)

)

val ccardSchema = StructType(schemaArray)

val csvFormat = "com.databricks.spark.csv"

// generate a DataFrame

val rawCCardDF = sqlContext.read

.format(csvFormat)

.option("header", "true")

.schema(ccardSchema)

.load(fileAddress)

rawCCardDF.cache()

rawCCardDF.count()

display(rawCCardDF)

// Databricks notebook source

/\*

\* import creditcard info from csv file into DataFrame

\*/

// COMMAND ----------

import org.apache.spark.sql.types.{StructType,StructField,IntegerType,DoubleType,StringType}

val fileAddress = "/FileStore/tables/creditcard.csv"

// Becasue null are not meaningful for ML algorithms and cannot be represented using scala.Double

val nullable = false

val schemaArray = Array(

StructField("Time", DoubleType, nullable),

StructField("V1", DoubleType, nullable),

StructField("V2", DoubleType, nullable),

StructField("V3", DoubleType, nullable),

StructField("V4", DoubleType, nullable),

StructField("V5", DoubleType, nullable),

StructField("V6", DoubleType, nullable),

StructField("V7", DoubleType, nullable),

StructField("V8", DoubleType, nullable),

StructField("V9", DoubleType, nullable),

StructField("V10", DoubleType, nullable),

StructField("V11", DoubleType, nullable),

StructField("V12", DoubleType, nullable),

StructField("V13", DoubleType, nullable),

StructField("V14", DoubleType, nullable),

StructField("V15", DoubleType, nullable),

StructField("V16", DoubleType, nullable),

StructField("V17", DoubleType, nullable),

StructField("V18", DoubleType, nullable),

StructField("V19", DoubleType, nullable),

StructField("V20", DoubleType, nullable),

StructField("V21", DoubleType, nullable),

StructField("V22", DoubleType, nullable),

StructField("V23", DoubleType, nullable),

StructField("V24", DoubleType, nullable),

StructField("V25", DoubleType, nullable),

StructField("V26", DoubleType, nullable),

StructField("V27", DoubleType, nullable),

StructField("V28", DoubleType, nullable),

StructField("Amount", DoubleType, nullable),

StructField("Class", DoubleType, nullable)

)

val ccardSchema = StructType(schemaArray)

val csvFormat = "com.databricks.spark.csv"

// generate a DataFrame

val rawCCardDF = sqlContext.read

.format(csvFormat)

.option("header", "true")

.schema(ccardSchema)

.load(fileAddress)

rawCCardDF.cache()

// Lazy evaluation

rawCCardDF.count()

display(rawCCardDF)

// COMMAND ----------

/\*

\* split DataFrame to training and testing data. 70% training is used for training, and 30% is used for testing

\*/

val splitDF = rawCCardDF.randomSplit(Array(0.7, 0.3), seed=11L)

val (trainData, testData) = (splitDF(0), splitDF(1))

trainData.cache()

testData.cache()

trainData.count()

testData.count()

// COMMAND ----------

/\*

\* prepare assembler, dtc for pipeline

\* The best is Logistic Regression, then Random Forest, finally Decision Tree

\* logistic regression can be used to predict a binary outcome by using binomial logistic regression, or it can be used to predict a multiclass outcome by using multinomial logistic regression. So it's more suitable in this question. (0.7549) While DT is (0.7310)

\*/

// COMMAND ----------

// Import the ML algorithms we will use.

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

import org.apache.spark.ml.classification.{DecisionTreeClassifier, DecisionTreeClassificationModel, \_}

import org.apache.spark.ml.Pipeline

import org.apache.spark.ml.evaluation.\_

// for assembler use, tell assembler which columns will be treated as features in Decision Tree： V1-V28 + Time + Amount

val categoricalColumns = Array("Time","V1","V2","V3","V4","V5","V6","V7","V8","V9","V10","V11","V12","V13","V14","V15","V16","V17","V18","V19","V20","V21","V22","V23","V24","V25","V26","V27","V28","Amount")

// assemble those 30 features into one Vector, named features

val assembler = new VectorAssembler().setInputCols(categoricalColumns).setOutputCol("features")

// DecisionTreeClassifier: Learn to predict column "Class" using the "features" column

val dtc = new DecisionTreeClassifier().setLabelCol("Class").setFeaturesCol("features").setMaxDepth(5)

// // LogisticRegression: Create initial LogisticRegression model

// val lr = new LogisticRegression().setLabelCol("Class").setFeaturesCol("features")

// // RandomForestClassifier: Create an initial RandomForest model.

// val rf = new RandomForestClassifier().setLabelCol("Class").setFeaturesCol("features")

// stages in our Pipeline. in this question, we donnot need to use StringIndexer, because Label is already DoubleType, no need to reIndex it. Pipeline only accepts DoubleType Label.

// assembler is Transformer. dtc is Estimator. They are in order in stages.

val stages = Array(assembler, dtc)

// val stages = Array(assembler, lr)

// val stages = Array(assembler, rf)

// Since we will have more than 1 stages of feature transformations, we use a Pipeline to tie the stages together. This simplifies our code.

// Chain assembler + dtc together into a single ML Pipeline.

val pipeline = new Pipeline().setStages(stages)

// use trainData prepared before to train Decision Tree

val model = pipeline.fit(trainData)

// val tree = model.stages.last.asInstanceOf[DecisionTreeClassificationModel]

// print("depth= " + tree.depth)

// display(tree)

// COMMAND ----------

/\*

\* Given the class imbalance ratio, we recommend measuring the accuracy using the Area Under the Precision-Recall Curve (AUPRC). Confusion matrix accuracy is not meaningful for unbalanced classification. Default metric is areaUnderROC. The evaluator currently accepts 2 kinds of metrics - areaUnderROC and areaUnderPR. Using areaUnderPR makes accuracy down a lot.

\* Since even if the model predicts all the records as normal transactions, it will still get an accuracy above 99%.

\* Because there are only two results in "label", so we use BinaryClassificationEvaluator()

\* 1. define the evaluator using AUPRC (default option)

\* 2. run the evaluator and test the accuracy

\*/

// COMMAND ----------

val predictions = model.transform(testData)

val evaluator = new BinaryClassificationEvaluator().setLabelCol("Class")

evaluator.setMetricName("areaUnderPR")

evaluator.evaluate(predictions)

// evaluator.getMetricName()

// // print "Class" and "prediction"

// predictions.select("Class","prediction").show()

**Result (By using areaUnderPR):**

