Shangqing Hu – SEC01 (NUID 001374342)

Big Data System Engineering with Scala
Spring 2023
Spark Assignment No.2

-List of Tasks Implemented

Url: https://www.kaggle.com/competitions/titanic/dataLinks to an external site.

For this assignment you will use the training and testing datasets.(train.csv & test.csv)

You are to load the dataset using Spark and perform the operations below:

Exploratory Data Analysis- Follow up on the previous spark assignment 1 and explain a few statistics. (20 pts)

Feature Engineering - Create new attributes that may be derived from the existing attributes. This may include removing certain columns in the dataset. (30 pts)

Prediction - Use the train.csv to train a Machine Learning model of your choice & test it on the test.csv. You are required to predict if the records in test.csv survived or not. Note(1 = Survived, 0 = Dead) (50 pts)

Please note: Do not include the test.csv while training the model. Also do not use the 'Survived' column during training. Doing these would defeat the purpose of the entire model.

The classifier must have an accuracy of 70 % for 100 pts.

-Code

```
//Importing required libraries

import org.apache.spark.sql.types.{StructType, StructField, StringType, IntegerType, FloatType};
import org.apache.spark.sql.functions._
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.classification.{RandomForestClassificationModel, RandomForestClassifier}
import org.apache.spark.ml.feature.{IndexToString, StringIndexer, VectorIndexer}
import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator
import org.apache.spark.mllib.stat.{MultivariateStatisticalSummary, Statistics}
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator
import org.apache.spark.ml.tuning.{CrossValidator, ParamGridBuilder}
import org.apache.spark.ml.classification.{GBTClassificationModel, GBTClassifier}
import org.apache.spark.ml.classification.DecisionTreeClassificationModel
import org.apache.spark.ml.classification.DecisionTreeClassifier

import org.apache.spark.sql.types.{StructType, StructField, StringType, IntegerType, FloatType}
import org.apache.spark.sql.functions._
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.classification.{RandomForestClassificationModel, RandomForestClassifier}
import org.apache.spark.ml.feature.{IndexToString, StringIndexer, VectorIndexer}
import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator
import org.apache.spark.mllib.stat.{MultivariateStatisticalSummary, Statistics}
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator
import org.apache.spark.ml.tuning.{CrossValidator, ParamGridBuilder}
import org.apache.spark.ml.classification.{GBTClassificationModel, GBTClassifier}
import org.apache.spark.ml.classification.DecisionTreeClassificationModel
import org.apache.spark.ml.classification.DecisionTreeClassifier
```

```
//Schema for importing training and testing datasets
val dataScheme_train = (new StructType)
.add("PassengerId", IntegerType)
.add("Survived", IntegerType)
.add("Pclass", IntegerType)
.add("Name", StringType)
.add("Sex", StringType)
.add("Age", FloatType)
.add("SibSp", IntegerType)
.add("Parch", IntegerType)
.add("Ticket", StringType)
.add("Fare", FloatType)
.add("Cabin", StringType)
.add("Embarked", StringType)

val dataScheme_test = (new StructType)
.add("PassengerId", IntegerType)
.add("Pclass", IntegerType)
.add("Name", StringType)
.add("Sex", StringType)
.add("Age", FloatType)
.add("SibSp", IntegerType)
.add("Parch", IntegerType)
.add("Ticket", StringType)
.add("Fare", FloatType)
.add("Cabin", StringType)
.add("Embarked", StringType)

val trainSchema = StructType(dataScheme_train)
val testSchema = StructType(dataScheme_test)
val csvFormat = "com.databricks.spark.csv"
val df_train = sqlContext.read.format(csvFormat).option("header", "true").schema(trainSchema).load("/FileStore/tables/train.csv")
val df_test = sqlContext.read.format(csvFormat).option("header", "true").schema(testSchema).load("/FileStore/tables/test.csv")

//Creating table views for training and testing
df_train.createOrReplaceTempView("df_train")
df_test.createOrReplaceTempView("df_test")
```

```
// Compute numcolumn summary statistics.
df_train.describe("Age","SibSp","Parch","Fare").show()
//df_train.show()
```

```
+-------+------------------+------------------+--------------------+-----------------+
|summary|               Age|             SibSp|               Parch|             Fare|
+-------+------------------+------------------+--------------------+-----------------+
|  count|               714|               891|                 891|              891|
|   mean| 29.69911764704046|0.5230078563411896|0.38159371492704824|32.20420804114722|
| stddev|14.526497332370992|1.1027434322934315| 0.8060572211299488|49.69342916316158|
|    min|              0.42|                 0|                   0|              0.0|
|    max|              80.0|                 8|                   6|         512.3292|
+-------+------------------+------------------+--------------------+-----------------+
```

```
//Summary stats for categorical columns
sqlContext.sql("select Survived,count(*) from df_train group by Survived").show()
sqlContext.sql("select Sex, Survived, count(*) from df_train group by Sex,Survived").show()
sqlContext.sql("select Pclass, Survived, count(*) from df_train group by Pclass,Survived").show()
```

```
+--------+--------+
|Survived|count(1)|
+--------+--------+
|       1|     342|
|       0|     549|
+--------+--------+

+------+--------+--------+
|   Sex|Survived|count(1)|
+------+--------+--------+
|  male|       0|     468|
|female|       1|     233|
|female|       0|      81|
|  male|       1|     109|
+------+--------+--------+

+------+--------+--------+
|Pclass|Survived|count(1)|
+------+--------+--------+
|     1|       0|      80|
|     3|       1|     119|
```

```scala
//Calculate avg Age and Fare to fill null values for training
val AvgAge = df_train.select("Age")
  .agg(avg("Age"))
  .collect() match {
  case Array(Row(avg: Double)) => avg
  case _ => 0
}

//Calculate average fare for filling gaps in dataset train
val AvgFare = df_train.select("Fare")
  .agg(avg("Fare"))
  .collect() match {
  case Array(Row(avg: Double)) => avg
  case _ => 0
}

//Calculate avg Age and Fare to fill null values for test data
val AvgAge_test = df_test.select("Age")
  .agg(avg("Age"))
  .collect() match {
  case Array(Row(avg: Double)) => avg
  case _ => 0
}

//Calculate average fare for filling gaps in dataset test
val AvgFare_test = df_test.select("Fare")
  .agg(avg("Fare"))
  .collect() match {
  case Array(Row(avg: Double)) => avg
  case _ => 0
}
```

```
AvgAge: Double = 29.69911764704046
AvgFare: Double = 32.20420804114722
AvgAge_test: Double = 30.272590361490668
AvgFare_test: Double = 35.62718864996656
```

```
//for training
val embarked: (String => String) = {
  case "" => "S"
  case null =>"S"
  case a  => a
}
val embarkedUDF = udf(embarked)

//for test
val embarked_test: (String => String) = {
  case "" => "S"
  case null =>"S"
  case a  => a
}
val embarkedUDF_test = udf(embarked_test)
```

```
embarked: String => String = <function1>
embarkedUDF: org.apache.spark.sql.expressions.UserDefinedFunction = UserDefinedFunction(<function1>,StringType,Some(List(StringType)))
embarked_test: String => String = <function1>
embarkedUDF_test: org.apache.spark.sql.expressions.UserDefinedFunction = UserDefinedFunction(<function1>,StringType,Some(List(StringType)))
```

```
//Filling null values with avg values for training dataset
val imputeddf = df_train.na.fill(Map("Fare" -> AvgFare, "Age" -> AvgAge))
val imputeddf2 = imputeddf.withColumn("Embarked", embarkedUDF(imputeddf.col("Embarked")))
//Splitting training data into training and validation
val Array(trainingData, validationData) = imputeddf2.randomSplit(Array(0.7, 0.3))

//Filling null values with avg values for test dataset
val imputeddf_test = df_test.na.fill(Map("Fare" -> AvgFare_test, "Age" -> AvgAge_test))
val imputeddf2_test = imputeddf_test.withColumn("Embarked", embarkedUDF_test(imputeddf_test.col("Embarked")))
```

```
imputeddf: org.apache.spark.sql.DataFrame = [PassengerId: int, Survived: int ... 10 more fields]
imputeddf2: org.apache.spark.sql.DataFrame = [PassengerId: int, Survived: int ... 10 more fields]
trainingData: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [PassengerId: int, Survived: int ... 10 more fields]
validationData: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [PassengerId: int, Survived: int ... 10 more fields]
imputeddf_test: org.apache.spark.sql.DataFrame = [PassengerId: int, Pclass: int ... 9 more fields]
imputeddf2_test: org.apache.spark.sql.DataFrame = [PassengerId: int, Pclass: int ... 9 more fields]
```

```
//Dropping Cabin feature as it has so many null values
val df1_train=trainingData.drop("Cabin")

val df1_test=imputeddf2_test.drop("Cabin")
```

```
df1_train: org.apache.spark.sql.DataFrame = [PassengerId: int, Survived: int ... 9 more fields]
df1_test: org.apache.spark.sql.DataFrame = [PassengerId: int, Pclass: int ... 8 more fields]
```

```
//Print schema for train and test
df1_train.printSchema()
df1_test.printSchema()
```

```
root
 |-- PassengerId: integer (nullable = true)
 |-- Survived: integer (nullable = true)
 |-- Pclass: integer (nullable = true)
 |-- Name: string (nullable = true)
 |-- Sex: string (nullable = true)
 |-- Age: float (nullable = false)
 |-- SibSp: integer (nullable = true)
 |-- Parch: integer (nullable = true)
 |-- Ticket: string (nullable = true)
 |-- Fare: float (nullable = false)
 |-- Embarked: string (nullable = true)

root
 |-- PassengerId: integer (nullable = true)
 |-- Pclass: integer (nullable = true)
 |-- Name: string (nullable = true)
 |-- Sex: string (nullable = true)
 |-- Age: float (nullable = false)
 |-- SibSp: integer (nullable = true)
 |-- Parch: integer (nullable = true)
```

```scala
//Indexing categorical features
val catFeatColNames = Seq("Pclass", "Sex", "Embarked")
val stringIndexers = catFeatColNames.map { colName =>
  new StringIndexer()
    .setInputCol(colName)
    .setOutputCol(colName + "Indexed")
    .fit(trainingData)
}

//Indexing target feature
val labelIndexer = new StringIndexer()
.setInputCol("Survived")
.setOutputCol("SurvivedIndexed")
.fit(trainingData)

//Assembling features into one vector
val numFeatColNames = Seq("Age", "SibSp", "Parch", "Fare")
val idxdCatFeatColName = catFeatColNames.map(_ + "Indexed")
val allIdxdFeatColNames = numFeatColNames ++ idxdCatFeatColName
val assembler = new VectorAssembler()
  .setInputCols(Array(allIdxdFeatColNames: _*))
  .setOutputCol("Features")


//Randomforest classifier
val randomforest = new RandomForestClassifier()
  .setLabelCol("SurvivedIndexed")
  .setFeaturesCol("Features")

//Retrieving original labels
val labelConverter = new IndexToString()
  .setInputCol("prediction")
  .setOutputCol("predictedLabel")
  .setLabels(labelIndexer.labels)

//Creating pipeline
val pipeline = new Pipeline().setStages(
  (stringIndexers :+ labelIndexer :+ assembler :+ randomforest :+ labelConverter).toArray)
```

```
//Selecting best model
val paramGrid = new ParamGridBuilder()
  .addGrid(randomforest.maxBins, Array(25, 28, 31))
  .addGrid(randomforest.maxDepth, Array(4, 6, 8))
  .addGrid(randomforest.impurity, Array("entropy", "gini"))
  .build()

val evaluator = new BinaryClassificationEvaluator()
  .setLabelCol("SurvivedIndexed")
  .setMetricName("areaUnderPR")

//Cross validator with 10 fold
val cv = new CrossValidator()
  .setEstimator(pipeline)
  .setEvaluator(evaluator)
  .setEstimatorParamMaps(paramGrid)
  .setNumFolds(10)


//Fitting model using cross validation
val crossValidatorModel = cv.fit(trainingData)

//predictions on validation data
val predictions = crossValidatorModel.transform(validationData)

//Accuracy
val accuracy = evaluator.evaluate(predictions)
println("Test Error DT= " + (1.0 - accuracy))
```

```scala
//predicting on test data
val predictions = crossValidatorModel.transform(df1_test)

predictions
  .withColumn("Survived", col("predictedLabel"))
  .select("PassengerId", "Survived")
  .coalesce(1)
  .write
  .format(csvFormat)
  .option("header", "true")
  .save("/FileStore/tables/abc")

sqlContext.sql("select * from output").collect.foreach(println)
```

```scala
////Implementing Gradient boosted tree
val gbt = new GBTClassifier()
  .setLabelCol("SurvivedIndexed")
  .setFeaturesCol("Features")
  .setMaxIter(10)

//Creating pipeline
val pipeline = new Pipeline().setStages(
  (stringIndexers :+ labelIndexer :+ assembler :+ gbt :+ labelConverter).toArray)

val model = pipeline.fit(trainingData)

val predictions = model.transform(validationData)

val evaluator = new BinaryClassificationEvaluator()
  .setLabelCol("SurvivedIndexed")
  .setMetricName("areaUnderPR")

val accuracy = evaluator.evaluate(predictions)
println("Test Error = " + (1.0 - accuracy))
```

```
Test Error = 0.15795530266298752
gbt: org.apache.spark.ml.classification.GBTClassifier = gbtc_73414bc3597d
pipeline: org.apache.spark.ml.Pipeline = pipeline_b2d258946e59
model: org.apache.spark.ml.PipelineModel = pipeline_b2d258946e59
predictions: org.apache.spark.sql.DataFrame = [PassengerId: int, Survived: int ... 19 more fields]
evaluator: org.apache.spark.ml.evaluation.BinaryClassificationEvaluator = binEval_9c5f86857618
accuracy: Double = 0.8420446973370125
```