Addiction patterns Analysis using Decision Tree Classifier, Random Forest and Logistic Regression

Python is great for data science modelling, thanks to its numerous modules and packages that help achieve data science goals. But what if the data you are dealing with cannot be fit into a single machine? Maybe you can implement careful sampling to do your analysis on a single machine, but with distributed computing framework like Pyspark, you can efficiently implement the task for large data sets.

Spark API is available in multiple programming languages (Scala, Java, Python and R). There are debates about how Spark performance varies depending on which language you run it on, but since we are comfortable with python, we went ahead with Pyspark

In [0]:
```
!pip install pyspark
```

In [0]:
```
from pyspark.sql import SparkSession, DataFrameReader, SQLContext
from pyspark.context import SparkContext
sc = SparkContext()
spark = SparkSession(sc)
sqlContext = SQLContext(sc)
```

Introduction

First step in any Apache programming is to create a SparkContext. SparkContext is needed when we want to execute operations in a cluster. SparkContext tells Spark how and where to access a cluster. It is first step to connect with Apache Cluster.

We have just created an Apache spark context

In [0]:
```
df = sqlContext.read.format('csv').options(header='true', inferSchema='true').lo
ad('daily_smoking_times.csv')
```

We import the libraries required here in the below snippet. We shall explore what every function does later below

In [0]:
```
from pyspark.ml import Pipeline
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.feature import VectorAssembler, StringIndexer, VectorIndexer, Mi
nMaxScaler
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder, TrainValidationS
plit
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.classification import LogisticRegression
```

This is a quick visual of how the data schema looks like.

In [0]:
```
display(df)
df.head()
```

```
DataFrame[Gender: string, Age: int, Time1: int, Time2: int, Time3: int, Time4:
int, Time5: int, Time6: int, Time7: int]
```

Out[0]:
```
Row(Gender='F', Age=13, Time1=4, Time2=8, Time3=13, Time4=14, Time5=19, Time
6=22, Time7=0)
```

In [0]:
```
df.printSchema()
```

```
root
 |-- Gender: string (nullable = true)
 |-- Age: integer (nullable = true)
 |-- Time1: integer (nullable = true)
 |-- Time2: integer (nullable = true)
 |-- Time3: integer (nullable = true)
 |-- Time4: integer (nullable = true)
 |-- Time5: integer (nullable = true)
 |-- Time6: integer (nullable = true)
 |-- Time7: integer (nullable = true)
```

We label the column Age as a label using the alias function

In [0]:
```
from pyspark.sql.functions import col
```

In [0]:
```
df2 = df.select("Gender","Time1","Time2","Time3","Time4","Time5","Time6","Time
7", col("Age").alias("label"))
```

In [0]:
```
df2
```

Out[0]:
```
DataFrame[Gender: string, Time1: int, Time2: int, Time3: int, Time4: int, Time
5: int, Time6: int, Time7: int, label: int]
```

Here we split the data into 70% and 30% where training is done using the former and testing using the latter.

In [0]:
```
splits = df2.randomSplit([0.7, 0.3])
```

In [0]:
```
train = splits[0]
```

In [0]:
```
test = splits[1].withColumnRenamed("label", "trueLabel")
train_rows = train.count()
test_rows = test.count()
```

In [0]:
```
print("Training Rows:", train_rows, " Testing Rows:", test_rows)
train.show(5)
test.show(5)
```

```
Training Rows: 4221  Testing Rows: 1779
+------+-----+-----+-----+-----+-----+-----+-----+-----+
|Gender|Time1|Time2|Time3|Time4|Time5|Time6|Time7|label|
+------+-----+-----+-----+-----+-----+-----+-----+-----+
|     F|    4|    8|   12|   14|   16|   20|    0|   19|
|     F|    4|    8|   12|   14|   16|   20|    3|   10|
|     F|    4|    8|   12|   14|   16|   21|    0|   41|
|     F|    4|    8|   12|   14|   16|   21|    2|   41|
|     F|    4|    8|   12|   14|   16|   22|    2|   36|
+------+-----+-----+-----+-----+-----+-----+-----+-----+
only showing top 5 rows

+------+-----+-----+-----+-----+-----+-----+-----+---------+
|Gender|Time1|Time2|Time3|Time4|Time5|Time6|Time7|trueLabel|
+------+-----+-----+-----+-----+-----+-----+-----+---------+
|     F|    4|    8|   12|   14|   16|   20|    3|       37|
|     F|    4|    8|   12|   14|   16|   21|    0|       12|
|     F|    4|    8|   12|   14|   16|   22|    1|       23|
|     F|    4|    8|   12|   14|   16|   23|    2|       40|
|     F|    4|    8|   12|   14|   16|   23|    2|       41|
+------+-----+-----+-----+-----+-----+-----+-----+---------+
only showing top 5 rows
```

In [0]:
```
test.printSchema()
```

```
root
 |-- Gender: string (nullable = true)
 |-- Time1: integer (nullable = true)
 |-- Time2: integer (nullable = true)
 |-- Time3: integer (nullable = true)
 |-- Time4: integer (nullable = true)
 |-- Time5: integer (nullable = true)
 |-- Time6: integer (nullable = true)
 |-- Time7: integer (nullable = true)
 |-- trueLabel: integer (nullable = true)
```

Here we use StringIndexer to encode a string of labels to a column of label indices.The unseen labels will be put at a particular index which is specified if user chooses to keep them. If the input column is numeric, we cast it to string and index the string values.

When downstream pipeline components such as Estimator or Transformer make use of this string-indexed label, you must set the input column of the component to this string-indexed column name. In many cases, you can set the input column with setInputCol.

In [0]:
```
strIdx = StringIndexer(inputCol = "label", outputCol = "typeAge")
labelIdx = StringIndexer(inputCol = "label", outputCol = "idxLabel")
```

Another utility we use here is the VectorAssembler.

VectorAssembler is a transformer that combines a given list of columns into a single vector column. It is useful for combining raw features and features generated by different feature transformers into a single feature vector, in order to train ML models like logistic regression and decision trees. VectorAssembler accepts the following input column types: all numeric types, boolean type, and vector type. In each row, the values of the input columns will be concatenated into a vector in the specified order.

In [0]:
```
catVect = VectorAssembler(inputCols = ["typeAge"], outputCol="ageFeatures")
catIdx = VectorIndexer(inputCol = catVect.getOutputCol(), outputCol = "idxAgeFea
tures")
numVect = VectorAssembler(inputCols = ["Gender","Time1","Time2","Time3","Time
4","Time5","Time6","Time7"], outputCol="numFeatures")
```

We also use another utility called **MinMaxScaler** here.

MinMaxScaler

`MinMaxScaler` transforms a dataset of `Vector` rows, rescaling each feature to a specific range (often [0, 1]). It takes parameters:

- `min`: 0.0 by default. Lower bound after transformation, shared by all features.
- `max`: 1.0 by default. Upper bound after transformation, shared by all features.

`MinMaxScaler` computes summary statistics on a data set and produces a `MinMaxScalerModel`. The model can then transform each feature individually such that it is in the given range.

The rescaled value for a feature E is calculated as,

$$Rescaled(e_i) = \frac{e_i - E_{min}}{E_{max} - E_{min}} * (max - min) + min \quad (1)$$

For the case $E_{max} == E_{min}$, $Rescaled(e_i) = 0.5 * (max + min)$

Note that since zero values will probably be transformed to non-zero values, output of the transformer will be `DenseVector` even for sparse input.

In [0]:
```
minMax = MinMaxScaler(inputCol = numVect.getOutputCol(), outputCol="normFeature
s")
featVect = VectorAssembler(inputCols=["idxAgeFeatures", "normFeatures"], outputC
ol="features")
```

In [0]:
```
cl = []
pipeline = []
```

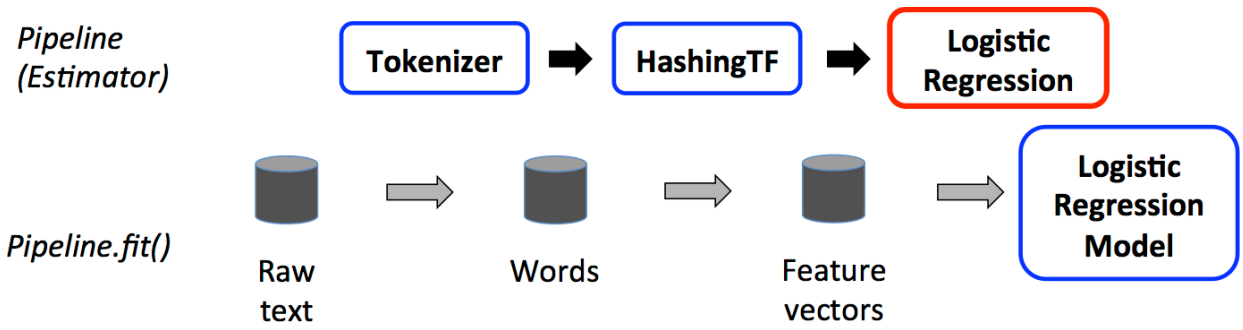
We build 3 classifiers and then insert them into a pipeline.

What is a Pipeline?

In machine learning, it is common to run a sequence of algorithms to process and learn from data. E.g., a simple text document processing workflow might include several stages:

```
Split each document's text into words.
Convert each document's words into a numerical feature vector.
Learn a prediction model using the feature vectors and labels.
```

MLlib represents such a workflow as a Pipeline, which consists of a sequence of PipelineStages (Transformers and Estimators) to be run in a specific order. We will use this simple workflow as a running example in this section.



In [0]:
```
cl.insert(0, DecisionTreeClassifier(labelCol="idxLabel", featuresCol="features"))
cl.insert(1, RandomForestClassifier(labelCol="idxLabel", featuresCol="features"))
cl.insert(2, LogisticRegression(labelCol="idxLabel", featuresCol="features"))
```

Decision trees are a popular family of classification and regression methods. More information about the spark.ml implementation can be found further in the section on decision trees.

Random Forests

Random forests are ensembles of decision trees. Random forests combine many decision trees in order to reduce the risk of overfitting. The spark.ml implementation supports random forests for binary and multiclass classification and for regression, using both continuous and categorical features.

In [0]:
```
for i in range(3):
    pipeline.insert(i, Pipeline(stages=[strIdx, labelIdx, catVect, catIdx, numVe
ct, minMax, featVect, cl[i]]))
    #piplineModel = Pipeline.fit(train)
print("Pipeline complete!")
```

```
Pipeline complete!
```

In [0]:
```
model = []
paramGrid = (ParamGridBuilder().addGrid(cl[0].impurity, ("gini", "entropy")).add
Grid(cl[0].maxDepth, [5, 10, 20]).addGrid(cl[0].maxBins, [5, 10, 20]).build())
cv = CrossValidator(estimator=pipeline[0], evaluator=BinaryClassificationEvaluato
r(), estimatorParamMaps=paramGrid, numFolds=5)

model.insert(0, cv.fit(train))
```

In [0]:
```
paramGrid2 = (ParamGridBuilder().addGrid(cl[1].impurity, ("gini", "entropy")).ad
dGrid(cl[1].maxDepth, [5, 10, 20]).addGrid(cl[1].maxBins, [5, 10, 20]).build())
cv2 = CrossValidator(estimator=pipeline[1], evaluator=BinaryClassificationEvaluat
or(), estimatorParamMaps=paramGrid2, numFolds=5)

model.insert(1, cv2.fit(train))
```

In [0]:
```
paramGrid = (ParamGridBuilder().addGrid(cl[2].impurity, ("gini", "entropy")).add
Grid(cl[2].maxDepth, [5, 10, 20]).addGrid(cl[2].maxBins, [5, 10, 20]).build())
cv = CrossValidator(estimator=pipeline[2], evaluator=BinaryClassificationEvaluato
r(), estimatorParamMaps=paramGrid, numFolds=5)

model.insert(2, cv3.fit(train))
```

In [0]:
```
prediction = []
predicted = []
for i in range(3):
  prediction.insert(i, model[i].transform(test))
  predicted.insert(i, prediction[i].select("features", "prediction", "probabilit
y", "trueLabel"))
  predicted[i].show(30)
```

In [0]:
```
#from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

evaluator = BinaryClassificationEvaluator(
    labelCol="trueLabel", rawPredictionCol="prediction")
for i in range(3):
    #evaluator = MulticlassClassificationEvaluator(
    #labelCol="trueLabel", predictionCol="prediction", metricName="weightedRecal
l")
    areUPR = evaluator.evaluate(predicted[i], {evaluator.metricName: "areaUnderP
R"})
    areUROC = evaluator.evaluate(predicted[i], {evaluator.metricName: "areaUnder
ROC"})
    print("AreaUnderPR = %g " % (areUPR))

    print("AreaUnderROC = %g " % (areUROC))

    tp = float(predicted[i].filter("prediction == 1.0 AND truelabel == 1").count
())
    fp = float(predicted[i].filter("prediction == 1.0 AND truelabel == 0").count
())
    tn = float(predicted[i].filter("prediction == 0.0 AND truelabel == 0").count
())
    fn = float(predicted[i].filter("prediction == 0.0 AND truelabel == 1").count
())

    precision = tp / (tp + fp)
    recall = tp / (tp + fn)
    print("Precision = %g " % (precision))
    print("Recall = %g " % (recall))

    metrics = sqlContext.createDataFrame([
    ("TP", tp),
    ("FP", fp),
    ("TN", tn),
    ("FN", fn),
    ("Precision", tp / (tp + fp)),
    ("Recall", tp / (tp + fn))],["metric", "value"])
    metrics.show()
```