USING PYTHON AND SPARK MACHINE LEARNING TO DO CLASSIFICATION

We've been writing about how to use Spark ML with the Scala programming language. But not many programmers know Scala. Python has moved ahead of Java in terms of number of users, largely based on the strength of machine learning. So, let's turn our attention to using Spark ML with Python.

You could say that Spark is Scala-centric. Scala has both Python and Scala interfaces and command line interpreters. Scala is the default one. The Python one is called **pyspark**. The most examples given by Spark are in Scala and in some cases no examples are given in Python.

(This tutorial is part of our Apache Spark Guide. Use the right-hand menu to navigate.)

Apache Atom

Python is the preferred language to use for data science because of NumPy, Pandas, and matplotlib, which are tools that make working with arrays and drawing charts easier and can work with large arrays of data efficiently. But Spark is designed to work with enormous amount of data, spread across a cluster. It's good practice to use both tools, switching back and forth, perhaps, as the demand warrants it.

But as we will see, because Spark dataframe is not the same as a Pandas dataframe, there is not 100% compatibility among all of these objects. You must convert Spark dataframes to lists and arrays and other structures in order to plot them with matplotlib. Because you can't slice arrays using the familiar , it takes more code to do the same operation.

But the other issue is performance. Apache Atom exists to efficiently convert objects in java

processes to python processes and vice versa. Spark is written in Java and Scala. Scala rides atop Java. Python, of course, runs in a Python process.

Arrow speeds up operations with as the conversion of Spark dataframes to Pandas dataframes and with column wise operations such as .**withcolumn()**.

Spark discusses some of the issues around this and the config change you need to make in Spark to take advantage of this boost in performance in their Apache Arrow documentation.

Heart patient data

Download the data from the University of São Paolo data set, available here. If you are curious, see this discussion.

The columns are:

1. Age
2. Sex
3. Chest pain type (4 values)
4. Resting blood pressure
5. Serum cholesterol in mg/dl
6. Fasting blood sugar > 120 mg/dl
7. Resting electrocardiographic results (values 0,1,2)
8. Maximum heart rate achieved
9. Exercise induced angina
10. Oldpeak = ST depression induced by exercise relative to rest
11. Slope of the peak exercise ST segment
12. Number of major vessels (0-3) colored by fluoroscopy
13. Thal: 3 = normal; 6 = fixed defect; 7 = reversable defect

The code, explained

The goal is to build a predictive binary logistic regression model using Spark ML and Python that predicts whether someone has a heart defect. The code below is available in a Zeppelin notebook here.

First, we read the data in and assign column names. Since the data is small, and because Pandas is easier, we read it into a Pandas dataframe. Then we convert it to a Spark dataframe with spark.createDataFrame().

You might see what I mean about the Spark dataframe lacking some of the features of Pandas. In particular we use Pandas so we can use .**iloc()** to take the first 13 columns and drop the last one, which seems to be noise not intended for the data.

```
%spark.pyspark

import pandas as pd
from pyspark.sql.types import StructType, StructField, NumericType
```

```
cols = ('age',
      'sex',
       'chest pain',
       'resting blood pressure',
       'serum cholesterol',
       'fasting blood sugar',
       'resting electrocardiographic results',
       'maximum heart rate achieved',
       'exercise induced angina',
       'ST depression induced by exercise relative to rest',
      'the slope of the peak exercise ST segment',
      'number of major vessels ',
       'thal',
       'last')

data = pd.read_csv('/home/ubuntu/Downloads/heart.csv', delimiter=' ',
names=cols)

data = data.iloc

data = data.apply(isSick)

df = spark.createDataFrame(data)
```

The field **thal** indicates whether the patient has a heart problem. The numbers are as follows:

- A value of 3 means the patient is healthy (normal).
- A value of 6 means the patient's health problem has been fixed.
- A value of 7 means the patient's health problem can be fixed.

So, write this function isSick() to flag 0 as negative and 1 as positive, because binary logistic regression requires one of two outcomes.

```
def isSick(x):
    if x in (3,7):
        return 0
    else:
        return 1
```

With machine learning and classification or regression problems we have:

- A matrix of features, including the patient's age, blood sugar, etc.
- A vector of **labels**, which indicates whether the patient has a heart problem.

Because we are using a Zeppelin notebook, and PySpark is the Python command shell for Spark, we write **%spark.pyspark** at the top of each Zeppelin cell to indicate the language and interpreter we want to use.

Next, we indicate which columns in the df dataframe we want to use as **features**. Then we use the

VectorAasembler to put all twelve of those fields into a new column called features that contains all of these as an array.

Now we create the Spark dataframe **raw_data** using the **transform()** operation and selecting only the features column.

```
%spark.pyspark
from pyspark.ml.feature import StandardScaler
from pyspark.ml.feature import VectorAssembler

features =   ('age',
      'sex',
       'chest pain',
       'resting blood pressure',
       'serum cholestoral',
       'fasting blood sugar',
       'resting electrocardiographic results',
       'maximum heart rate achieved',
       'exercise induced angina',
       'ST depression induced by exercise relative to rest',
      'the slope of the peak exercise ST segment',
      'number of major vessels ')

assembler = VectorAssembler(inputCols=features,outputCol="features")

raw_data=assembler.transform(df)
raw_data.select("features").show(truncate=False)
```

We use the **Standard Scaler** to put all the numbers on the same scale, which is standard practice for machine learning. This takes the observation and subtracts the mean, and then divides that by the standard deviation.

```
%spark.pyspark
from pyspark.ml.feature import StandardScaler

standardscaler=StandardScaler().setInputCol("features").setOutputCol("Scaled_
features")
raw_data=standardscaler.fit(raw_data).transform(raw_data)
raw_data.select("features","Scaled_features").show(5)
```

Here is what the features data looks like now:

```
+-----------------------------------------------------------+
|features                                                   |
+-----------------------------------------------------------+
||
||
||
||
```

```
||
||
||
||
||
||
||
||
```

As usual, we split the data into **training** and **test** datasets. We don't have much data so we will use a 50/50 split.

```
%spark.pyspark
from pyspark.ml.tuning import ParamGridBuilder, TrainValidationSplit

training, test = raw_data.randomSplit(, seed=12345)
```

Now we create the logistic Regression Model and train it, meaning have the model calculate the coefficients and intercept that most nearly matches the results that we have in the label column **isSick**

```
%spark.pyspark
from pyspark.ml.classification import LogisticRegression

lr = LogisticRegression(labelCol="isSick",
featuresCol="Scaled_features",maxIter=10)
model=lr.fit(training)
predict_train=model.transform(training)
predict_test=model.transform(test)
predict_test.select("isSick","prediction").show(10)
```

Here we show the first few rows in side by side comparison. These are, for the most part, correct.

```
+------+----------+
|isSick|prediction|
+------+----------+
|     0|       0.0|
|     1|       0.0|
|     0|       0.0|
|     0|       0.0|
|     0|       0.0|
|     0|       0.0|
|     0|       0.0|
|     0|       1.0|
|     0|       1.0|
|     0|       0.0|
+------+----------+
```

This shows the coefficients and intercept.

```
%spark.pyspark
print("Multinomial coefficients: " + str(model.coefficientMatrix))
print("Multinomial intercepts: " + str(model.interceptVector))
```

Here they are:

```
Multinomial coefficients: DenseMatrix([])
Multinomial intercepts:
```

Now we use some Spark SQL functions F to create a new column **correct** when **IsSick** is equal to **prediction**, meaning the predicted result equaled the actual results.

```
%spark.pyspark
import pyspark.sql.functions as F
check = predict_test.withColumn('correct', F.when(F.col('isSick') ==
F.col('prediction'), 1).otherwise(0))
check.groupby("correct").count().show()
```

Here are the results:

```
+-------+-----+
|correct|count|
+-------+-----+
|      1|  137|
|      0|   10|
+-------+-----+
```

So, the accuracy is 137 / 137 + 10 = 93%

There are other ways to show the accuracy of the model, like area under the curve. But this is the simplest to understand, unless you are an experienced data scientist and statistician. We will explain more complex ways of checking the accuracy in future articles.