Classification in Spark

By the end of this activity, you will be able to perform the following in Spark:

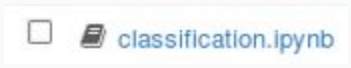
1. Generate a categorical variable from a numeric variable
2. Aggregate the features into one single column
3. Randomly split the data into training and test sets
4. Create a decision tree classifier to predict days with low humidity.

In this activity, you will be programming in a Jupyter Python Notebook. If you have not already started the Jupyter Notebook server, see the instructions in the Reading *Instructions for Starting Jupyter*.

Step 1. **Open Jupyter Python Notebook.** Open a web browser by clicking on the web browser icon at the top of the toolbar:

Navigate to *localhost:8889/tree/Downloads/big-data-4*:

Open the handling missing values notebook by clicking on *classification.ipynb:*



Step 2. **Load classes and data.** Execute the first cell in the notebook to load the classes used for this exercise.

```
In [1]: from pyspark.sql import SQLContext
        from pyspark.sql import DataFrameNaFunctions
        from pyspark.ml import Pipeline
        from pyspark.ml.classification import DecisionTreeClassifier
        from pyspark.ml.feature import Binarizer
        from pyspark.ml.feature import VectorAssembler, StringIndexer, VectorIndexer
```

Next, execute the second cell which loads the weather data into a DataFrame and prints the columns.

```
In [2]: sqlContext = SQLContext(sc)
        df = sqlContext.read.load('file:///home/cloudera/Downloads/big-data-4/daily_weather.csv',
                                  format='com.databricks.spark.csv',
                                  header='true',inferSchema='true')
        df.columns

Out[2]: ['number',
         'air_pressure_9am',
         'air_temp_9am',
         'avg_wind_direction_9am',
         'avg_wind_speed_9am',
         'max_wind_direction_9am',
         'max_wind_speed_9am',
         'rain_accumulation_9am',
         'rain_duration_9am',
         'relative_humidity_9am',
         'relative_humidity_3pm']
```

Execute the third cell, which defines the columns in the weather data we will use for the decision tree classifier.

```
In [3]: featureColumns = ['air_pressure_9am','air_temp_9am','avg_wind_direction_9am','avg_wind_speed_9am'
                'max_wind_direction_9am','max_wind_speed_9am','rain_accumulation_9am',
                'rain_duration_9am']
```

Step 3. **Drop unused and missing data.** We do not need the *number* column in our data, so let's remove it from the DataFrame:

```
In [4]: df = df.drop('number')
```

Next, let's remove all rows with missing data:

```
In [5]: df = df.na.drop()
```

We can print the number of rows and columns in our DataFrame:

```
In [6]: df.count(), len(df.columns)
Out[6]: (1064, 10)
```

Step 4. **Create categorical variable.** Let's create a categorical variable to denote if the humidity is not low. If the value is less than 25%, then we want the categorical value to be 0, otherwise the

categorical value should be 1. We can create this categorical variable as a column in a DataFrame using *Binarizer:*

```
In [7]: binarizer = Binarizer(threshold=24.99999, inputCol="relative_humidity_3pm", outputCol="label")
        binarizedDF = binarizer.transform(df)
```

The *threshold* argument specifies the threshold value for the variable, *inputCol* is the input column to read, and *outputCol* is the name of the new categorical column. The second line applies the *Binarizer* and creates a new DataFrame with the categorical column. We can look at the first four values in the new DataFrame:

```
In [8]: binarizedDF.select("relative_humidity_3pm","label").show(4)
+---------------------+-----+
|relative_humidity_3pm|label|
+---------------------+-----+
|   36.160000000000494|  1.0|
|     19.4265967985621|  0.0|
|   14.460000000000045|  0.0|
|   12.742547353761848|  0.0|
+---------------------+-----+
only showing top 4 rows
```

The first row's humidity value is greater than 25% and the label is 1. The other humidity values are less than 25% and have labels equal to 0.

Step 5. **Aggregate features.** Let's aggregate the features we will use to make predictions into a single column:

```
In [9]: assembler = VectorAssembler(inputCols=featureColumns, outputCol="features")
        assembled = assembler.transform(binarizedDF)
```

The *inputCols* argument specifies our list of column names we defined earlier, and *outputCol* is the name of the new column. The second line creates a new DataFrame with the aggregated features in a column.

Step 6. **Split training and test data.** We can split the data by calling *randomSplit():*

```
In [10]: (trainingData, testData) = assembled.randomSplit([0.8,0.2], seed = 13234 )
```

The first argument is how many parts to split the data into and the *approximate* size of each. This specifies two sets of 80% and 20%. Normally, the seed should not be specified, but we use a specific value here so that everyone will get the same decision tree.

We can print the number of rows in each DataFrame to check the sizes (1095 * 80% = 851.2):

```
In [11]: trainingData.count(), testData.count()
Out[11]: (854, 210)
```

Step 7. **Create and train decision tree.** Let's create the decision tree:

```
In [12]: dt = DecisionTreeClassifier(labelCol="label", featuresCol="features", maxDepth=5,
                                     minInstancesPerNode=20, impurity="gini")
```

The *labelCol* argument is the column we are trying to predict, *featuresCol* specifies the aggregated features column, *maxDepth* is stopping criterion for tree induction based on maximum depth of tree, *minInstancesPerNode*is stopping criterion for tree induction based on minimum number of samples in a node, and *impurity* is the impurity measure used to split nodes.

We can create a model by training the decision tree. This is done by executing it in a *Pipeline:*

```
In [13]: pipeline = Pipeline(stages=[dt])
         model = pipeline.fit(trainingData)
```

Let's make predictions using our test data set:

```
In [14]: predictions = model.transform(testData)
```

Looking at the first ten rows in the prediction, we can see the prediction matches the input:

```
In [15]: predictions.select("prediction", "label").show(10)
+----------+-----+
|prediction|label|
+----------+-----+
|       1.0|  1.0|
|       1.0|  1.0|
|       1.0|  1.0|
|       1.0|  1.0|
|       1.0|  1.0|
|       1.0|  1.0|
|       0.0|  0.0|
|       1.0|  1.0|
|       1.0|  1.0|
|       1.0|  1.0|
+----------+-----+
only showing top 10 rows
```

Step 8. **Save predictions to CSV.** Finally, let's save the predictions to a CSV file. In the next Spark hands-on activity, we will evaluate the accuracy.

Let's save only the *prediction* and *label* columns to a CSV file:

```
In [16]: predictions.select("prediction", "label").write.save(path="file:///home/cloudera/Downloads/big-data-4/predictions.csv",
                                                              format="com.databricks.spark.csv",
                                                              header='true')
```