Xi'an Jiaotong-Liverpool University
西交利物浦大學

DTS303TC Big Data Security and Analytics

Laboratory Session 3 – Classification and Evaluation in Spark – Decision Tree

(Duration: 2 hours)

Overview:

The Classification algorithm is a Supervised Learning technique that is used to identify the category of new observations on the basis of training data. In Classification, a program learns from the given dataset or observations and then classifies new observation into a number of classes or groups.

Evaluation metrics are tied to machine learning tasks. Using different metrics for performance evaluation, we should be able to improve our model's overall predictive power before we roll it out for production on unseen data.

In this activity, you will be able to perform the following in Spark:

- Generate a categorical variable from a numeric variable
- Aggregate the features into one single column
- Randomly split the data into training and test sets
- Create a decision tree classifier to predict days with low humidity
- Determine the accuracy of a classifier model
- Display the confusion matrix for a classifier model

PART 1: Decision Tree Classification in Spark

Task Instructions

In this part, 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.

Step1: Reconfigure number of CPUs in Cloudera VM. For this hands on exercise, we need at least 2 virtual CPUs for the Cloudera VM.

Step2: Open Jupyter Python Notebook.

a) Open a web browser and navigate to *localhost:8889/tree/Downloads/big-data-4.*
b) Open the handling classification notebook by clicking on *classification.ipynb.*

Step 3: Load classes and data.

a) 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
```

b) 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']
```

c) 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 4: Drop unused and missing data.

a) We do not need the *number* column in our data, so let's remove it from the DataFrame.

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

b) Next, let's remove all rows with missing data.

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

c) We can print the number of rows and columns in our DataFrame.

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

Out[6]: (1064, 10)
```

Step 5: Create categorical variable.

a) 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. Binarizer uses three parameters *threshold, inputCol* and *outputCol*.
The *threshold* argument specifies the threshold value(24.99999) for the variable, *inputCol* is the input column(relative_humidity_3pm) to read, and *outputCol* is the name of the new categorical column(label).

```
In [7]: binarizer=Binarizer(threshold=24.99999,inputCol='relative_humidity_3pm',outputCol='label')
```

b) Applies the Binarizer and creates a new DataFrame with the categorical column by *transform()*.

```
In [8]: binarizedDF=binarizer.transform(df)
```

c) Show the first four values of columns *relative_humidity_3pm* and *label* in the new DataFrame .

```
In [9]: 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
```

Step 6: Aggregate features.

a) Let's aggregate the features we will use to make predictions into a single column. The inputCols argument specifies our list of column (featureColumns) names we defined earlier, and outputCol is the name of the new column(features).

```
In [10]: assembler=VectorAssembler(inputCols=featureColumns,outputCol='features')
```

b) Creates a new DataFrame with the aggregated features in a column.

```
In [11]: assembled=assembler.transform(binarizedDF)
```

Step 7: Split training and test data.

a) We can split the data by calling *randomSplit()*.

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 (13234) so that everyone will get the same decision tree.

```
In [12]: (trainingData, testData)=assembled.randomSplit([0.8,0.2],seed=13234)
```

b) Print the number of rows in each DataFrame to check the sizes (1095 * 80% = 851.2).
NOTE: You need to reconfigure the VM to use 2 CPUs as described in the reading Instructions for Changing the Number of Cloudera VM CPUs.

```
In [13]: trainingData.count(),testData.count()

Out[13]: (839, 225)
```

Step 8: Create and train decision tree.

a) Let's create the decision tree.
The labelCol argument is the column(label) we are trying to predict, featuresCol specifies the aggregated features column (features), maxDepth (5) is stopping criterion for tree induction based on maximum depth of tree, minInstancesPerNode (20) is stopping criterion for tree induction based on minimum number of samples in a node, and impurity is the impurity measure(gini) used to split nodes.

```
In [14]: dt=DecisionTreeClassifier(labelCol="label",featuresCol="features",maxDepth=5,
                                  minInstancesPerNode=20,impurity="gini")
```

b) Create a model by training the decision tree. This is done by executing it in a *Pipeline*:

```
In [15]: pipeline=Pipeline(stages=[dt])
         model=pipeline.fit(trainingData)
```

c) Let's make predictions using our test data set.

```
In [16]: predictions=model.transform(testData)
```

d) Looking at the first ten rows in the prediction, we can see the prediction matches the input.

```
In [17]: predictions.select('prediction','label').show(10)
```

```
+----------+-----+
|prediction|label|
+----------+-----+
|       1.0|  1.0|
|       1.0|  0.0|
|       1.0|  0.0|
|       1.0|  1.0|
|       0.0|  0.0|
|       0.0|  1.0|
|       0.0|  0.0|
|       0.0|  1.0|
|       1.0|  1.0|
|       0.0|  0.0|
+----------+-----+
only showing top 10 rows
```

Step 9: Save predictions to CSV.

a) 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 [19]: predictions.select("prediction","label").write.save(path="file:///home/cloudera/Downloads/big-data-4/predictions.csv",
                                                              format="com.databricks.spark.csv", header='true')
```

PART 2: Decision Tree Evaluation in Spark

Task Instructions

In this part, 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.

a) Open a web browser and navigate to *localhost:8889/tree/Downloads/big-data-4*.

b) Open the handling classification notebook by clicking on *model-evaluation.ipynb*.

Step 2: Load predictions.

a) Execute the first cell to load the classes used in this part.

```
In [1]: from pyspark.sql import SQLContext
        from pyspark.ml.evaluation import MulticlassClassificationEvaluator
        from pyspark.mllib.evaluation import MulticlassMetrics
```

b) Execute the next cell to load the predictions CSV file that we created at the end of the part 1 into a DataFrame.

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

Step 3: Compute accuracy.

a) Let's create an instance of *MulticlassClassificationEvaluator* to determine the accuracy of the predictions.
The first two arguments *labelCol* and *predictionCol* specify the names of the label and prediction columns, and the third argument specifies that we want the overall precision.

```
In [3]: evaluator =MulticlassClassificationEvaluator(
                labelCol='label',predictionCol='prediction',metricName='precision')
```

b) We can compute the accuracy by calling *evaluate()*.

```
In [4]: accuracy=evaluator.evaluate(predictions)
        print("Accuracy=%g " %(accuracy))

        Accuracy=0.786667
```

Step 4: Display confusion matrix. The *MulticlassMetrics* class can be used to generate a confusion matrix of our classifier model. However, unlike *MulticlassClassificationEvaluator*, *MulticlassMetrics* works with RDDs of numbers and not DataFrames, so we need to convert our predictions DataFrame into an RDD.

a) If we use the *rdd* attribute of predictions, we see this is an RDD of Rows.

```
In [5]: predictions.rdd.take(2)

Out[5]: [Row(prediction=1.0, label=1.0), Row(prediction=1.0, label=0.0)]
```

b) Instead, we can map the RDD to tuple to get an RDD of numbers.

```
In [6]: predictions.rdd.map(tuple).take(2)

Out[6]: [(1.0, 1.0), (1.0, 0.0)]
```

c) Let's create an instance of MulticlassMetrics with this RDD.

```
In [7]: metrics=MulticlassMetrics(predictions.rdd.map(tuple))
```

NOTE: the above command can take longer to execute than most Spark commands when first run in the notebook.

d) The *confusionMatrix()* function returns a Spark Matrix, which we can convert to a Python Numpy array, and transpose to view.

```
In [8]: metrics.confusionMatrix().toArray().transpose()

Out[8]: array([[ 93.,  22.],
               [ 26.,  84.]])
```