Bike Rental Data Set

December 7, 2018

1 Part 1: Bike Rental Data Set from UCI Machine Learning Repository

1.0.1 Soufiane MOUTEI - Ahmed BEN SAAD

```
In [1]: from pyspark.sql import SparkSession

        spark = SparkSession.builder.appName("ML").getOrCreate()
```

```
In [2]: spark
```

```
Out[2]: <pyspark.sql.session.SparkSession at 0x1062b9898>
```

Let's load the data using Spark

```
In [3]: rowData = spark.read.csv("Bike Rental UCI dataset.csv", inferSchema=True, header=True)
        rowData.show(n=5)
```

```
+------+---+----+---+-------+----------+----------+----+----+---------+---------+----+------+
|season| yr|mnth| hr|holiday|workingday|weathersit|temp| hum|windspeed|dayOfWeek|days|demand|
+------+---+----+---+-------+----------+----------+----+----+---------+---------+----+------+
|     1|  0|   1|  0|      0|         0|         1|0.24|0.81|      0.0|      Sat|   0|    16|
|     1|  0|   1|  1|      0|         0|         1|0.22| 0.8|      0.0|      Sat|   0|    40|
|     1|  0|   1|  2|      0|         0|         1|0.22| 0.8|      0.0|      Sat|   0|    32|
|     1|  0|   1|  3|      0|         0|         1|0.24|0.75|      0.0|      Sat|   0|    13|
|     1|  0|   1|  4|      0|         0|         1|0.24|0.75|      0.0|      Sat|   0|     1|
+------+---+----+---+-------+----------+----------+----+----+---------+---------+----+------+
only showing top 5 rows
```

We'll transform some categorical features using One-Hot Encoding: season, hr, mnth, dayOfWeek. These features don't represent an order between them, for instance, if we give Sunday 0 and Monday 1, for the algorithm, it may mean that Monday > Sunday which is not the case; hence One-Hot Encoding

To do that, we'll use **StringIndexer** if necessary to encode a string column of labels to a column of label indices, then perform One Hot Encoding using **OneHotEncoderEstimator**:

In [4]:
```python
from pyspark.ml.feature import StringIndexer
from pyspark.ml.feature import OneHotEncoderEstimator

indexer = StringIndexer(inputCol='dayOfWeek', outputCol='day_cat')
rowData = indexer.fit(rowData).transform(rowData)

onehot_encoder = OneHotEncoderEstimator(
    inputCols=["day_cat", "season", "hr", "mnth"],
    outputCols=["day_onehot", "season_onehot", "hr_onehot", "mnth_onehot"]
)
rowData = onehot_encoder.fit(rowData).transform(rowData)
rowData.show(n=10)
```

```
+------+---+----+---+-------+----------+----------+----+----+---------+---------+----+------+---
|season| yr|mnth| hr|holiday|workingday|weathersit|temp| hum|windspeed|dayOfWeek|days|demand|day
+------+---+----+---+-------+----------+----------+----+----+---------+---------+----+------+---
|     1|  0|   1|  0|      0|         0|         1|0.24|0.81|      0.0|      Sat|   0|    16|
|     1|  0|   1|  1|      0|         0|         1|0.22| 0.8|      0.0|      Sat|   0|    40|
|     1|  0|   1|  2|      0|         0|         1|0.22| 0.8|      0.0|      Sat|   0|    32|
|     1|  0|   1|  3|      0|         0|         1|0.24|0.75|      0.0|      Sat|   0|    13|
|     1|  0|   1|  4|      0|         0|         1|0.24|0.75|      0.0|      Sat|   0|     1|
|     1|  0|   1|  5|      0|         0|         2|0.24|0.75|   0.0896|      Sat|   0|     1|
|     1|  0|   1|  6|      0|         0|         1|0.22| 0.8|      0.0|      Sat|   0|     2|
|     1|  0|   1|  7|      0|         0|         1| 0.2|0.86|      0.0|      Sat|   0|     3|
|     1|  0|   1|  8|      0|         0|         1|0.24|0.75|      0.0|      Sat|   0|     8|
|     1|  0|   1|  9|      0|         0|         1|0.32|0.76|      0.0|      Sat|   0|    14|
+------+---+----+---+-------+----------+----------+----+----+---------+---------+----+------+---
only showing top 10 rows
```

Now we'll define the columns to use in the algorithm:

In [5]:
```python
cols = (
    rowData
    .drop('dayOfWeek')
    .drop('day_cat')
    .drop('demand')
    .drop('hr')
    .drop('mnth')
    .drop('season')
    .columns
)
print(cols)
```

```
['yr', 'holiday', 'workingday', 'weathersit', 'temp', 'hum', 'windspeed', 'days', 'day_onehot',
```

We'll split now the data into 70% for training and 30% for testing:

```
In [6]: trainData, testData = rowData.randomSplit([0.7, 0.3])
        trainData.cache()
        testData.cache();
```

We're going to use **VectorAssembler** to assemble the columns in *cols* into one column so it can be used in the algorithms of *pyspark.ml*. The algorithm that is going to be used is Decision Trees. We're going to use a pipeline to perform these two classes ('VectorAssembler' and 'DecisionTreeRegressor').

We're going to use a **CrossValidator** to perform a cross-validation. It requires an estimator (our pipeline), a set of parameters so we can find the best parameters to use. We'll use the MAE (Mean Absolute Error) of **RegressionEvaluator** in order to evaluate the grid of parameters.

```
In [7]: from pyspark.ml.regression import DecisionTreeRegressor
        from pyspark.ml.tuning import CrossValidator
        from pyspark.ml import Pipeline
        from pyspark.ml.tuning import ParamGridBuilder
        from pyspark.ml.feature import VectorAssembler
        from pyspark.ml.evaluation import RegressionEvaluator
        from pyspark.ml.feature import VectorIndexer

        vec_assembler = VectorAssembler(inputCols=cols, outputCol = 'features')

        tree = DecisionTreeRegressor(featuresCol='features', labelCol='demand')

        pipeline_tree = Pipeline(stages=[vec_assembler, tree])

        paramGrid_tree = (
            ParamGridBuilder()
            .addGrid(tree.maxDepth, [2, 5, 15, 20])
            .addGrid(tree.maxBins, [40, 300])
            .addGrid(tree.minInfoGain, [0.0, 0.05])
            .build()
        )

        evaluator_tree = RegressionEvaluator(labelCol="demand", metricName="mae")

        crossval_tree = CrossValidator(estimator=pipeline_tree,
                                  estimatorParamMaps=paramGrid_tree,
                                  evaluator=evaluator_tree,
                                  numFolds=5)

        # Run cross-validation, and choose the best set of parameters.
        cvModel_tree = crossval_tree.fit(trainData)

        # Predict for testData
        results_tree = cvModel_tree.transform(testData)

        print("MAE: %f" % evaluator_tree.evaluate(results_tree))
        print("R_2: %f" % evaluator_tree.evaluate(results_tree, {evaluator_tree.metricName: "r2"
```

```
MAE: 44.352057
R_2: 0.844362
```

We got an MAE of 44.35 and a R^2 score of 0.84 which is really an improvement of what we got using LinearRegression ($MAE = 75.37$ and $R^2 = 0.67$) given that Decision Trees is good at capturing the non-linearity in the data.

We can improve more this score using Random Forest algorithm, a bagging method that is known as good at reducing the variance:

```
In [8]: from pyspark.ml.regression import RandomForestRegressor

        tree = RandomForestRegressor(featuresCol='features', labelCol='demand')

        pipeline_tree = Pipeline(stages=[vec_assembler, tree])

        paramGrid_tree = (
            ParamGridBuilder()
            .addGrid(tree.maxDepth, [2, 5, 15, 20])
            .addGrid(tree.maxBins, [40, 300])
            .addGrid(tree.minInfoGain, [0.0, 0.05])
            .build()
        )

        evaluator_tree = RegressionEvaluator(labelCol="demand", metricName="mae")

        crossval_tree = CrossValidator(estimator=pipeline_tree,
                                       estimatorParamMaps=paramGrid_tree,
                                       evaluator=evaluator_tree,
                                       numFolds=5)

        # Run cross-validation, and choose the best set of parameters.
        cvModel_tree = crossval_tree.fit(trainData)

        # Predict for testData
        results_tree = cvModel_tree.transform(testData)

        print("MAE: %f" % evaluator_tree.evaluate(results_tree))
        print("R_2: %f" % evaluator_tree.evaluate(results_tree, {evaluator_tree.metricName: "r2"
```

```
MAE: 39.817655
R_2: 0.899465
```

We got an MAE of 39.82 and an R^2 score (0.90) which is really an improvement of what we got before.