Lab - 5

Objectives:

1. Revisiting concepts in Apache Spark.
2. Building a custom Spark ML Transformer.
3. Visualizing and feature engineering using Spark ML.
4. Understanding how ***spark-submit*** works.

Instructions:

Building custom Spark ML Transformer:

We are going to build a custom Transformer to convert Celsius to Fahrenheit, using PySpark:

1. Import required modules, like so:

```
from pyspark import keyword_only
from pyspark.ml import Transformer
from pyspark.ml.param.shared import HasInputCol, HasOutputCol
from pyspark.sql.functions import udf
from pyspark.sql.types import *
```

2. Create a Spark DataFrame, like so:

```
df = spark.createDataFrame([float(i) for i in range(0, 200, 3)],
FloatType()).toDF("celsius")
```

3. Check the schema, like so:

```
df.printSchema()
```

4. Create the custom transformer class, like so:

```
class CelsisusToFahrenheitTransformer(Transformer, HasInputCol,
HasOutputCol):

  @keyword_only
  def __init__(self, inputCol=None, outputCol=None):
      super(CelsisusToFahrenheitTransformer, self).__init__()
      kwargs = self._input_kwargs
      self.setParams(**kwargs)

  @keyword_only
  def setParams(self, inputCol=None, outputCol=None):
      kwargs = self._input_kwargs
      return self._set(**kwargs)
  def _transform(self, dataset):

      def f(s):
          # C/5 = (F-32)/9
          return ((9 * s)/5) + 32

      t = FloatType()
      out_col = self.getOutputCol()
      in_col = dataset[self.getInputCol()]

      return dataset.withColumn(out_col, udf(f, t)(in_col))
```

5. Try it out:

```
c_to_f_converter = CelsisusToFahrenheitTransformer(inputCol="celsius",
outputCol="fahrenheit")
converted_df = c_to_f_converter.transform(df)
display(converted_df)
```

Visualizing and feature engineering using Spark ML:

```
file_location = "/FileStore/tables/train_taxi.csv"
training =
spark.read.format("csv").option("header",True).load(file_location)
display(training)

training = training.drop('key')
display(training)

from pyspark.ml.feature import SQLTransformer
SQLTrans = SQLTransformer(statement = 'Select * from __THIS__ where
pickup_longitude > -80 and pickup_longitude < -70 and pickup_latitude >
35 and pickup_latitude < 45 and dropoff_longitude > -80 and
dropoff_longitude < -70 and dropoff_latitude > 35 and dropoff_latitude
< 45')
training.count()

training = SQLTrans.transform(training)
training.count()

colnames = ['fare_amount', 'pickup_latitude', 'pickup_longitude',
'dropoff_latitude', 'dropoff_longitude', 'pickup_datetime',
'passenger_count']
for col1 in colnames:
  df = training.where(training[col1].isNull())
  if df.count() > 0:
    training = training.where(training[col1].isNotNull())
```

```
jfk_lat = 40.6413
jfk_long = 73.7781

SQLTrans = SQLTransformer(statement = 'Select
*,SQRT(POW(40.6413-pickup_latitude,2) +
POW(73.7781-pickup_longitude,2)) as pickup_dist_jfk from __THIS__')

training = SQLTrans.transform(training)
class DateTimeManipulator(Transformer, HasInputCol, HasOutputCol):

    @keyword_only
    def __init__(self, inputCol=None, outputCol=None,
date_format=None):
      super(DateTimeManipulator, self).__init__()
      self.date_format = Param(self, "date_format", "")
      self._setDefault(date_format="yyyy-mm-dd hh:mm:ss")
      kwargs = self._input_kwargs
      self.setParams(**kwargs)

    @keyword_only
    def setParams(self, inputCol=None, outputCol=None, stopwords=None):
      kwargs = self._input_kwargs
      return self._set(**kwargs)

    def setdate_format(self, value):
      self._paramMap[self.stopwords] = value
      return self

    def getdate_format(self):
      return self.getOrDefault(self.date_format)

    def _transform(self, dataset):
      date_format = self.date_format
      def f(s):
        splits = s.split('-')
        month = splits[1]
        splits2 = splits[2].split(" ")
        day = splits2[0]
        hour = splits2[1].split(":")[0]
        return [month, day, hour]

      t = ArrayType(StringType())
      out_col = self.getOutputCol()
      in_col = dataset[self.getInputCol()]
      return dataset.withColumn(out_col, udf(f, t)(in_col))
```

```
Custom_Trans = DateTimeManipulator(inputCol = 'pickup_datetime',
outputCol = 'temp_features')
training = Custom_Trans.transform(training)

from pyspark.sql.types import IntegerType
def f(s):
  return int(s[0])

t = IntegerType()
temp_f = udf(f,t)
training = training.withColumn("Month",
temp_f(training['temp_features']))

def f(s):
  return int(s[1])

t = IntegerType()
temp_f = udf(f,t)
training = training.withColumn("Day",
temp_f(training['temp_features']))
def f(s):
  return int(s[2])

t = IntegerType()
temp_f = udf(f,t)
training = training.withColumn("Hour",
temp_f(training['temp_features']))

training = training.withColumn("fare_amount",
training['fare_amount'].cast('int'))
temp_training = training.sample(False, 0.1)
PD = temp_training.toPandas()


import matplotlib.pyplot as plt
import numpy as np
PD1 = PD[PD['Hour']==10]
maximum = PD1.max()['fare_amount']
minimum = PD1.min()['fare_amount']
width = maximum-minimum/100
arr = np.linspace(minimum, maximum, 100)
counts = []
for i in range(len(arr)-1):
  counts.append(sum(PD1['fare_amount'].between(arr[i],arr[i+1])))

fig, ax = plt.subplots()
ax.bar(arr[0:len(arr)-1], counts)
```

```
display(fig)

import matplotlib.pyplot as plt
import numpy as np
PD1 = PD[PD['Month']==7]
maximum = PD1.max()['fare_amount']
minimum = PD1.min()['fare_amount']
width = maximum-minimum/100
arr = np.linspace(minimum, maximum, 100)
counts = []
for i in range(len(arr)-1):
  counts.append(sum(PD1['fare_amount'].between(arr[i],arr[i+1])))

fig, ax = plt.subplots()
ax.bar(arr[0:len(arr)-1], counts)
display(fig)
```

Using spark-submit:

You are now familiar with performing interactive data analyses and performing ML tasks using notebooks. In production or for running an application, notebooks are not helpful and thus we need a way to submit Spark programs to the cluster. This is performed by using the ***spark-submit*** script that comes with every Spark installation.

Writing a Spark program (WordCount) using PySpark:

1. Import the required modules, like so:

```
import argparse, sys

from pyspark import SparkContext, SparkConf
```

2. Include the main logic, like so:

```
# Build the CLI
parser = argparse.ArgumentParser(description="Word Count")
parser.add_argument("input_file", type=str,
                    help="path to input file")
parser.add_argument("output_path", type=str,
                    help="path to output")

args = parser.parse_args()

# Build a new SparkConf and set the App name
conf = SparkConf().setAppName("Word Count")

# Build a new SparkContext
sc = SparkContext(conf=conf)
```

```
# Change the log level of the SparkContext
sc.setLogLevel("ERROR")

if args.input_file and args.output_path:
    word_count = sc.textFile(args.input_file).flatMap(
        lambda line: line.split(" ")).map(
        lambda word: (word.strip(",").strip("."), 1)).reduceByKey(
        lambda a, b: a + b)

    word_count.saveAsTextFile(args.output_path)

else:
    args.print_usage()
    sys.exit(1)
```

3. Add the Python specific requirements, like so:

```
... SparkContext, SparkConf

if __name__ == "__main__":
    conf = ...
```

Submitting a Spark program to be run on cluster (or local):

You can submit the program by using the following command in the terminal:

```
$ spark-submit word_count.py "/path/to/input/file" "/path/to/output"
```

Report:

1. Build a custom Estimator for normalizing a column(think why it should be an estimator) and fit it into a Pipeline for training a model and then generate predictions from the obtained model. Attach code snippet for the above.

2. Build a PySpark program for the above, while saving the output to a file and then use ***spark-submit*** for submitting to a Spark cluster. Attach code snippet for the above. Also attach the screenshot for the output obtained. **Note:** You can use your laptop as a local Spark cluster.