CIS5560 Term Project Tutorial

Authors: Shailja Pandit , Anish Omprakash Pandey, Surya Kiran Golagani

Instructor: Jongwook Woo

Date: 05/22/2022

Lab Tutorial

Shailja Pandit (spandit3@calstatela.edu)

Anish Omprakash Pandey (apandey9@calstatela.edu)

Surya Kiran Golagani (sgolaga@calstatela.edu)

05/22/2022

E-Commerce Multi-Category Store

Predictive Analysis using machine learning models in Spark ML

Objectives

The objective of the lab is to build a model that predicts the optimal price and EventType considering the features of Item and User Details using the following machine learning algorithms:

Item Price Prediction

- Linear Regression
- Decision Tree Regression
- Random Forest Regression
- Gradient Boost Tree Regression

Event-Type Prediction

- Decision Tree Classifier

You will learn how to build above listed regression and classification model.

Platform Specifications

- Version - 3.2.1-amzn-3.1
- CPU Speed: 2.40 GHz
- # of CPU cores: 4
- # Number of nodes: 3
- Total Memory Size: 481GB

Dataset Specifications

Dataset Name: eCommerce behavior data from multi category store

Dataset Size: 5 GB

Dataset URL: https://www.kaggle.com/datasets/mkechinov/ecommerce-behavior-data-from-multi-category-store?select=2019-Oct.csv

Dataset Format: CSV

Step 1: Get data manually from the Data source.

We first need to Download the dataset from Kaggle

1. Login to Kaggle.
2. Download the Dataset for the month of October from https://www.kaggle.com/datasets/mkechinov/ecommerce-behavior-data-from-multi-category-store?select=2019-Oct.csv

Step 2: Uploading data to Hadoop File System (HDFS)

Manually uploading the file to the Hadoop directory , we need to first transfer it to the local directory using scp commands.

NOTE:

1. Please Replace **"[server-path]"** with the location for uploading the file into the server (For eg. **/tmp/[user-name]/)**
2. Please Replace **"[Dataset-filename]"** with the path of the downloaded file from Kaggle

NOTE: Replace your username instead of "**apandey9**"

```
scp 2019-Oct_Master.csv apandey9@35.87.210.247:~/tmp/ap9/
```

Now we have to upload the file to HDFS folder. Run the following HDFS commands to create the directory in HDFS

Step 3: Connect to Hadoop Spark cluster.

For that Now open a another shell terminal and paste the ssh command to connect to the Hadoop Spark cluster.
· You may download and install Git Bash: https://git-scm.com/downloads
· If you use Linux or Mac computers, you can simply open "terminal". You can search for "term" to find it. Then it will show you terminal that you can use.
3. You have to use your Calstate LA email account (apandey9)name to log in to the cluster, In a terminal, you need to type in the following ssh shell command to connect.

NOTE: Replace your username instead of "**apandey9**"

$ ssh **apandey9**@35.87.210.247

```
$ ssh apandey9@35.87.210.247
```

To enter password, type in your username as password and press enter.

Now you will be logged in.

```
-bash-4.2$
-bash-4.2$
```

Now you can run all the queries below to complete the tutorial.

Step 4: Create a directory "cis-5560" to put the file to HDFS

a. Run the following HDFS commands to create the directory in HDFS

```
hdfs dfs -mkdir cis-5560
```

b. Next, you can run the following shell command to put file in respective directory

```
hdfs dfs -put 2019-Oct_Master.csv cis-5560/
```

c. Run the following 2 HDFS commands to make sure if **usedcarsdata*.csv** file is uploaded to **used_cars** directory (table):

```
hdfs dfs -ls
```

```
-bash-4.2$ hdfs dfs -ls
Found 3 items
drwxr-xr-x   - apandey9 hdfs          0 2022-05-21 05:32 .hiveJars
drwxr-xr-x   - apandey9 hdfs          0 2022-05-19 20:41 .sparkStaging
drwxr-xr-x   - apandey9 hdfs          0 2022-05-22 01:40 cis-5560
-bash-4.2$
```

This command will display the list of all files in hdfs.

```
hdfs dfs -ls cis-5560/
```

```
-bash-4.2$ hdfs dfs -ls cis-5560
Found 1 items
-rw-r--r--   3 apandey9 hdfs 5668612855 2022-05-09 05:39 cis-5560/2019-Oct_Master.csv
-bash-4.2$
```

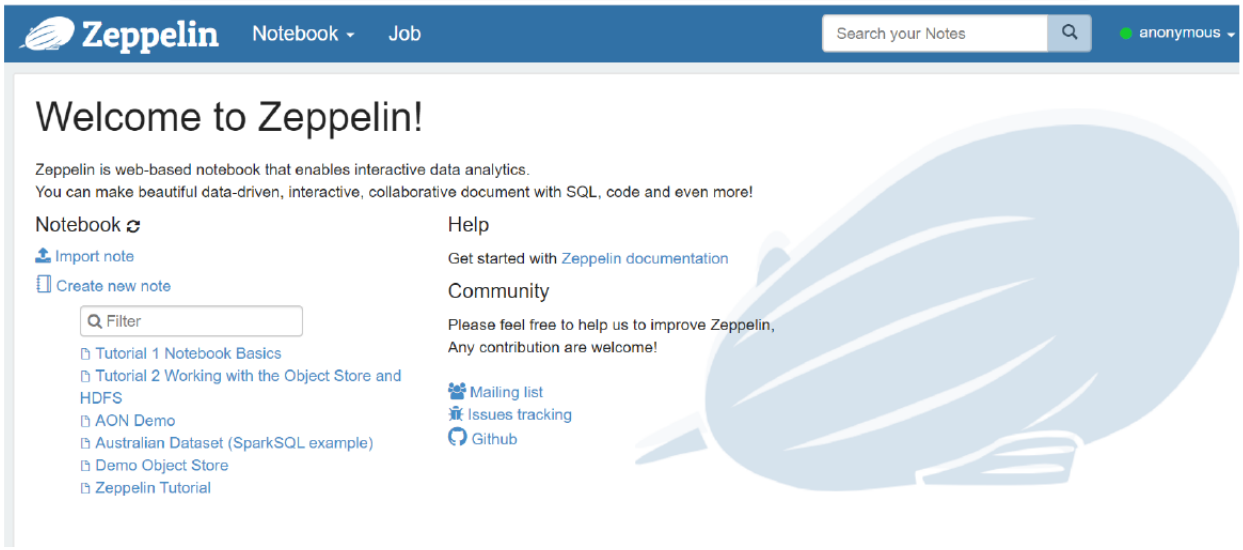
Step 5: Login to Zepplin

1. Ssh to the Oracle server and ssh –L –M for port forwarding to open ipython file to Zeppelin. You have to use your username:

NOTE: Replace your username instead of "**apandey9**"

```
ssh -N -L 8880:localhost:8090 apandey9@35.87.210.247
```

2. In your Web browser i.e. Chrome , Navigate to http://localhost:8880/#/



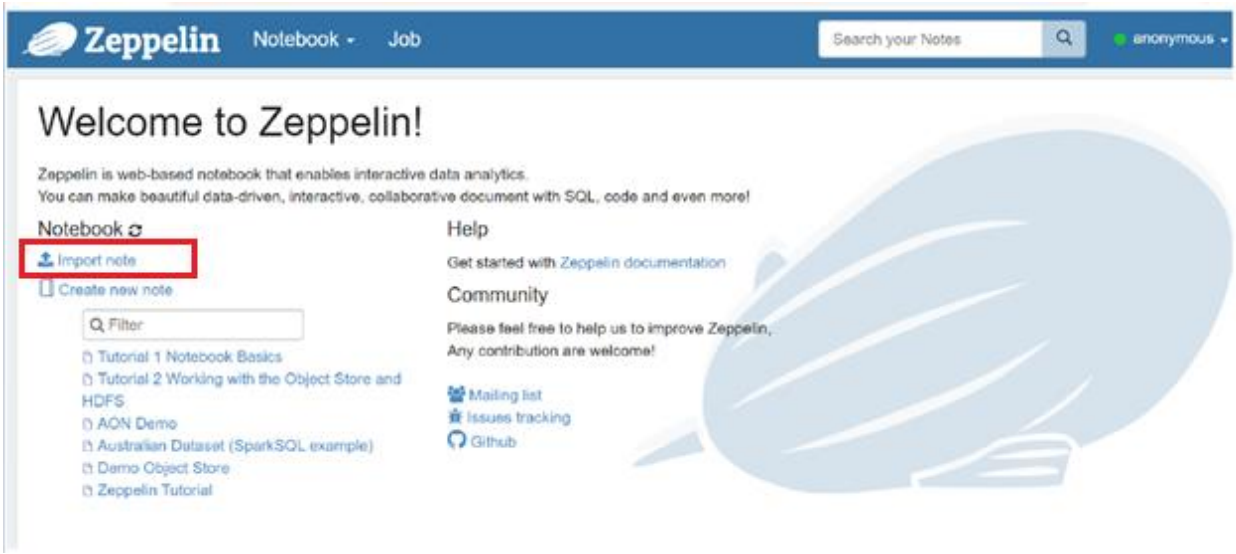
ALTERNATIVE 1 (Step 6-Step 7)

Step 6: Download the python files from github

1. Navigate to Github Link : https://github.com/shailjapandit05/CIS-5560-e-Commerce-Prediction-Project
2. Download the two ipynb files – eCommerceRegression.ipynb and eCommerceClassification.ipynb .

Step 7: Run file in Zeppelin

a. On the Notebook page, Navigate to Import Note > Browse. Upload the Python Files for eCommerceRegression.ipynb from Github Link and skip the previous steps. Then, locate and select the python file of the zeppelin notebook. Change the file name to "**/username/filename**", this will create the folder of your username and file in it.



b. After importing create a folder by your **username/file name** [the same file name in your computer]

c. After creating the folder in zeppelin. Go In that folder and open the file

d. After opening the file replace all **%python** with **%pyspark**

e. After replacing run all the commands. Now you may compare the results of all the Regression algorithms.

f. Follow the same process from **Step "a to e"** for eCommerceClassification.ipynb and You may see the results for Classification algorithm.

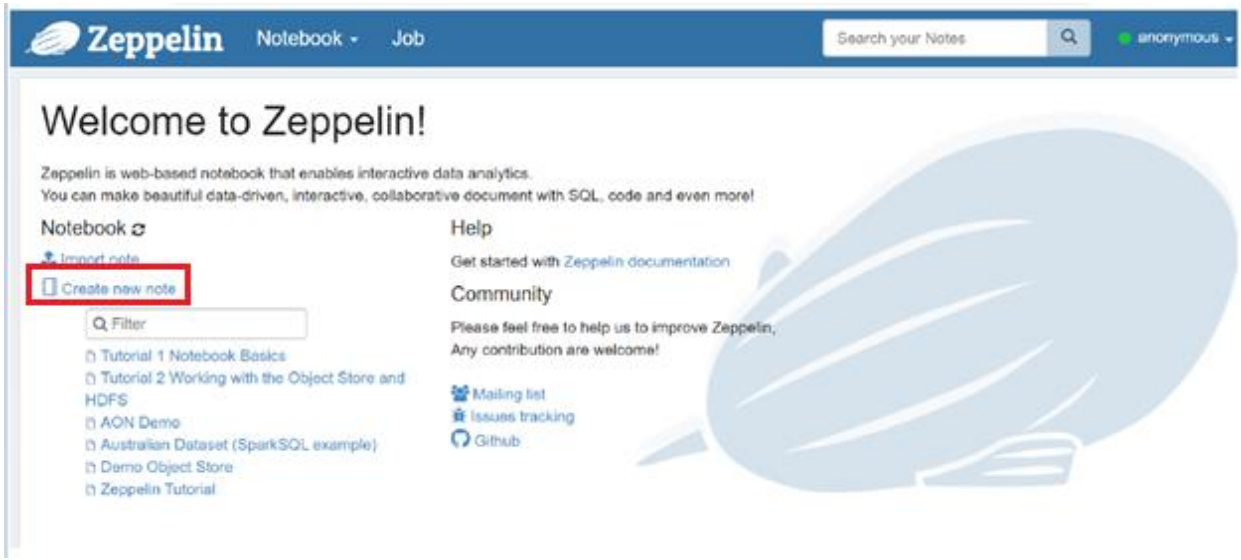
ALTERNATIVE 2 (Step 8- Step 14)

Step 8: Create a new Note

Click on 'Create new note' on Zeppelin

1. First download all the libraries you need from Github.
2. Then go to zeppelin as mentioned in **"Step 5"**

3. Then click on create note



4. Now a new blank note will open.
5. Now you can copy paste these codes below and run all the commands to see and compare all the results.

```
%pyspark
from pyspark.sql.types import *
from pyspark.ml.regression import
LinearRegression ,DecisionTreeRegressor,
RandomForestRegressor,GBTRegressionModel, GBTRegressor,
FMRegressor
from pyspark.sql.functions import *
from pyspark.ml.feature import *
from pyspark.ml import *
from pyspark.ml.classification import *
from pyspark.ml.evaluation import *
from pyspark.mllib.evaluation import *
from pyspark.sql.types import *
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator,
TrainValidationSplit
from pyspark.context import SparkContext
from pyspark.sql.session import SparkSession
```

DataFrame Schema , that should be a Table schema.

```
%pyspark
eCommerceSchema = StructType([
  StructField("event_time", TimestampType(), False),
  StructField("event_type", StringType(), False),
  StructField("product_id", IntegerType(), False),
  StructField("category_id", LongType(), False),
  StructField("StringType", StringType(), False),
  StructField("brand", StringType(), False),
  StructField("price", DoubleType(), False),
  StructField("user_id", IntegerType(), False),
  StructField("user_session", StringType(), False),
])
```

NOTE: you have to use your username in plae of "**apandey9**"

6. Load Source Data- We need to load the data from Hadoop server.

```
%pyspark
# File location and type
file_location = "/user/apandey9/5560/2019-Oct_Master.csv"
file_type = "csv"

# CSV options
infer_schema = "true"
first_row_is_header = "true"
delimiter = ","

# The applied options are for CSV files. For other file types,
these will be ignored.
df = spark.read.format(file_type) \
.option("inferSchema", infer_schema) \
.option("header", first_row_is_header) \
.option("sep", delimiter) \
.load(file_location)
df = df.withColumnRenamed('# File format is event_time',
'event_time')
df = df.filter((col("brand") != "No value") &
(col("category_code") != "No value"))
df.show()
```

```
%pyspark
# File location and type
file_location = "/user/apandey9/5560/2019-Oct_Master.csv"
file_type = "csv"

# CSV options
infer_schema = "true"
first_row_is_header = "true"
delimiter = ","

# The applied options are for CSV files. For other file types, these will be ignored.
df = spark.read.format(file_type) \
  .option("inferSchema", infer_schema) \
  .option("header", first_row_is_header) \
  .option("sep", delimiter) \
  .load(file_location)

df = df.withColumnRenamed('# File format is event_time', 'event_time')
df = df.filter((col("brand") != "No value") & (col("category_code") != "No value"))

df.show()
```

```
|2019-10-01 00:00:...|      view|   2900536|2053013554776244595|appliances.kitche...|elenberg|  51.46|555158050|b5bdd0b3-4ca2-4c5...|
|2019-10-01 00:00:...|      view|   1005011|2053013555631882655|electronics.smart...| samsung| 900.64|530282093|50a293fb-5940-41b...|
```

7. Set the values in Event Type column to Numerical values. For example in the code below, ‘View’ is set to 1. ‘Cart’ is set to 2 and ‘Purchase’ to 3.

```
%pyspark
df = df.withColumn("event_type", when(df.event_type == 'view',
1) \
  .when(df.event_type == 'cart', 2) \
   .when(df.event_type == 'purchase', 3))

df.show()
```

Use String indexer to identify column as categorical variable, i.e., want to convert the textual data to numeric data keeping the categorical context. For our Project we converted columns like Event Type, Category Code and Brand Values to their respective Index Values.

```
%pyspark
from pyspark.ml.feature import StringIndexer

indexer = StringIndexer(inputCol="brand",
outputCol="brandIndex")
indexer1 = StringIndexer(inputCol="category_code",
outputCol="category_codeIndex")
df = indexer.fit(df).transform(df)
df = indexer1.fit(df).transform(df)
df.show()
```

```
%pyspark
from pyspark.ml.feature import StringIndexer

indexer = StringIndexer(inputCol="brand", outputCol="brandIndex")
indexer1 = StringIndexer(inputCol="category_code", outputCol="category_codeIndex")
df = indexer.fit(df).transform(df)
df = indexer1.fit(df).transform(df)
df.show()
```

```
+-------------------+----------+----------+-------------------+--------------------+-------+------+--------+--------------------+----------+-----------------+
|         event_time|event_type|product_id|        category_id|       category_code|  brand| price| user_id|        user_session|brandIndex|category_codeIndex|
+-------------------+----------+----------+-------------------+--------------------+-------+------+--------+--------------------+----------+-----------------+
|2019-10-01 00:00:  |         1|   3900821|2053013552326770905|appliances.enviro...|  aqua| 33.2|55474717|9333dfbd-b87a-470...|     164.0|             28.0|
```

8. Create a Temporary View of a Dataframe and Display the first 5 rows of the table to ensure if all the columns are displayed properly and are ready to be used to build a model.

```
%pyspark
# Create a view or table
temp_table_name = "2019_Oct_Master_csv"
df.createOrReplaceTempView(temp_table_name)
```

```
%pyspark
if PYSPARK_CLI:
    csv = spark.read.csv('2019_Oct_Master_csv',
inferSchema=True, header=True)
else:
    csv = spark.sql("SELECT * FROM 2019_Oct_Master_csv")

csv.show(5)
```

```
%pyspark
# Create a view or table
temp_table_name = "2019_Oct_1_txt"
df.createOrReplaceTempView(temp_table_name)
```

Took 0 sec. Last updated by anonymous at May 11 2022, 9:28:41 PM.

```
%pyspark
if PYSPARK_CLI:
    csv = spark.read.csv('2019_Oct_1_txt', inferSchema=True, header=True)
else:
    csv = spark.sql("SELECT * FROM 2019_Oct_1_txt")


csv.show(5)
```

```
+--------------------+----------+----------+-------------------+--------------------+------+-------+---------+--------------------+----------+------------------+
|          event_time|event_type|product_id|        category_id|       category_code| brand|  price|  user_id|        user_session|brandIndex|category_codeIndex|
+--------------------+----------+----------+-------------------+--------------------+------+-------+---------+--------------------+----------+------------------+
|2019-10-01 00:00:...|         1|   3900821|2053013552326770905|appliances.enviro...|  aqua|   33.2|554748717|9333dfbd-b87a-470...|     164.0|              28.0|
|2019-10-01 00:00:...|         1|   1307067|2053013558920217191|  computers.notebook|lenovo| 251.74|550050854|7c90fc70-0e80-459...|       7.0|               2.0|
|2019-10-01 00:00:...|         1|   1004237|2053013555631882655|electronics.smart...| apple|1081.98|535871217|c6bd7419-2748-4c5...|       1.0|               0.0|
|2019-10-01 00:00:...|         1|   1480613|2053013561092866779|   computers.desktop|pulser| 908.62|512742880|0d0d91c2-c9c2-4e8...|      40.0|               9.0|
|2019-10-01 00:00:...|         1|  28719074|2053013565480109009|  apparel.shoes.keds| baden| 102.71|520571932|ac1cd4e5-a3ce-422...|      29.0|              10.0|
+--------------------+----------+----------+-------------------+--------------------+------+-------+---------+--------------------+----------+------------------+
only showing top 5 rows
```

9. Select Features and Label. The features are descriptive attributes, and the label is what you're attempting to predict or forecast.

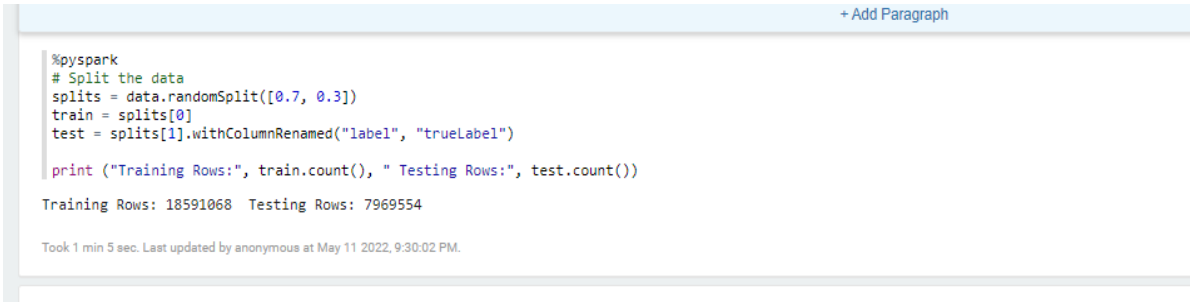
```
%pyspark
data = csv.select("product_id", "brandIndex",
"category_codeIndex", "event_type", "user_id",
col("price").alias("label"))
data.show(5)
```

Step 9: Splitting the Dataset

This step is to split the data into Train and Test data in the ratio of 70:30. Training dataset is used to build a model and Testing dataset is used to Test the model built.

```
%pyspark
# Split the data
splits = data.randomSplit([0.7, 0.3])
train = splits[0]
test = splits[1].withColumnRenamed("label", "trueLabel")

print ("Training Rows:", train.count(), " Testing Rows:",
test.count())
```



Step 10: Random Forest Regression

Run Random Forest Regression algorithm using Train Split Validation and Cross Validation.

PREPARE THE TRAINING DATA

To train the regression model, you need a training data set that includes a vector of numeric features, and a label column. In this exercise, you will use the VectorAssembler class to transform the feature columns into a vector. We define a pipeline that creates a Feature Vector and trains a regression model.

```
%pyspark
assembler_rf = VectorAssembler(inputCols=["product_id", "brandIndex",
"category_codeIndex", "event_type" , "user_id" ],
outputCol="features")

rf = RandomForestRegressor(labelCol="label", featuresCol="features",
maxBins=3000)
```

TRAIN SPLIT VALIDATOR

PARAMETER BUILDING , DEFINE PIPELINE AND TUNE PARAMETERS USING TRAIN SPLIT VALIDATOR

You can tune the parameters to find the best model for your data. To do this use the TrainSplit Validator class to evaluate each combination of parameters defined in a ParameterGrid. Fitting the model takes a long time to run because every parameter combination is tried. We define a pipeline that creates a Feature Vector and trains a regression model.

```
%pyspark
paramGrid_rf = ParamGridBuilder() \
  .addGrid(rf.maxDepth, [2, 3]) \
  .addGrid(rf.minInfoGain, [0.0]) \
  .build()
```

```
%pyspark
pipeline0_rf = Pipeline(stages=[assembler_rf, rf])

tvs_rf = TrainValidationSplit(estimator=pipeline0_rf,
evaluator=RegressionEvaluator(), estimatorParamMaps=paramGrid_rf,
trainRatio=0.8)

model0_rf = tvs_rf.fit(train)
```

TEST THE MODEL AND EXAMINE THE PREDICTED AND ACTUAL VALUES

Now you're ready to use the transform method of the model to generate some predictions. You can use this approach to predict Item Price where the label is unknown; but in this case you are using the test data which includes a known true label value, so you can compare the predicted Item Price to the actual Item Price.
You can plot the predicted values against the actual values to see how accurately the model has predicted. In a perfect model, the resulting scatter plot should form a perfect diagonal line with each predicted value being identical to the actual value - in practice, some variance is to be expected. Run the cells below to create a temporary table from the predicted DataFrame and then retrieve the predicted and actual label values.

```
%pyspark
prediction0_rf = model0_rf.transform(test)
predicted0_rf = prediction0_rf.select("features", "prediction",
"trueLabel")
predicted0_rf.show(20)
```

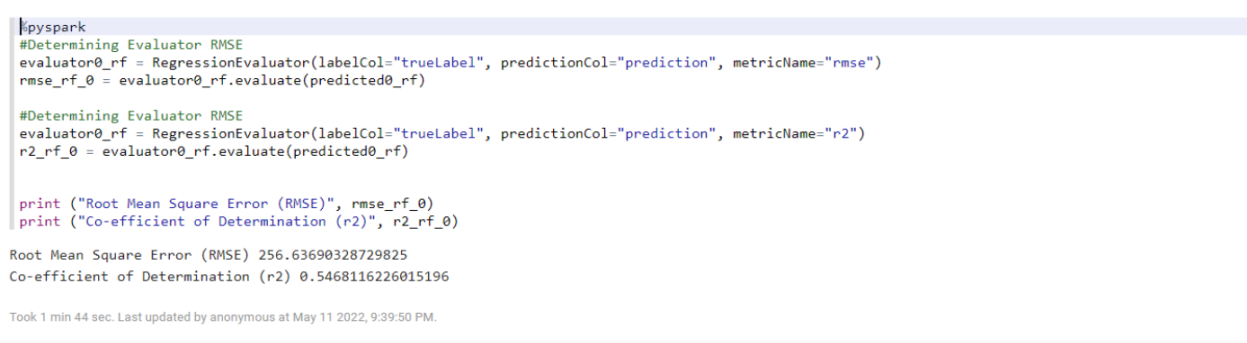
CALCULATE TRAIN VALIDATION SPLIT RMSE AND R2

We will now calculate RMSE and R2 for Random Forest Regression using Train Split Validator. There are a number of metrics used to measure the variance between predicted and actual values. Of these, the root mean square error (RMSE) is a commonly used value that is measured in the same units as the prediced and actual values - so in this case, the RMSE indicates the average number of minutes between predicted and actual Item Price Values. You can use the RegressionEvaluator class to retrieve the RMSE . The ideal Accuracy of R2 is 1. The closer the value to 1 the lesser the error, and the better the model.

```
%pyspark
#Determining Evaluator RMSE
evaluator0_rf = RegressionEvaluator(labelCol="trueLabel",
predictionCol="prediction", metricName="rmse")
rmse_rf_0 = evaluator0_rf.evaluate(predicted0_rf)

#Determining Evaluator RMSE
evaluator0_rf = RegressionEvaluator(labelCol="trueLabel",
predictionCol="prediction", metricName="r2")
r2_rf_0 = evaluator0_rf.evaluate(predicted0_rf)


print ("Root Mean Square Error (RMSE)", rmse_rf_0)
print ("Co-efficient of Determination (r2)", r2_rf_0)
```



CROSS VALIDATOR

PARAMETER BUILDING, DEFINE PIPELINE AND TUNE PARAMETERS USING CROSS VALIDATOR

You can tune the parameters to find the best model for your data. To do this use the Cross Validator class to evaluate each combination of parameters defined in a ParameterGrid against multiple folds of the data split into training and validation datasets, in order to find the best performing parameters. Fitting the model takes a long time to run because every parameter combination is tried multiple times. We define a pipeline that creates a Feature Vector and trains a regression model.

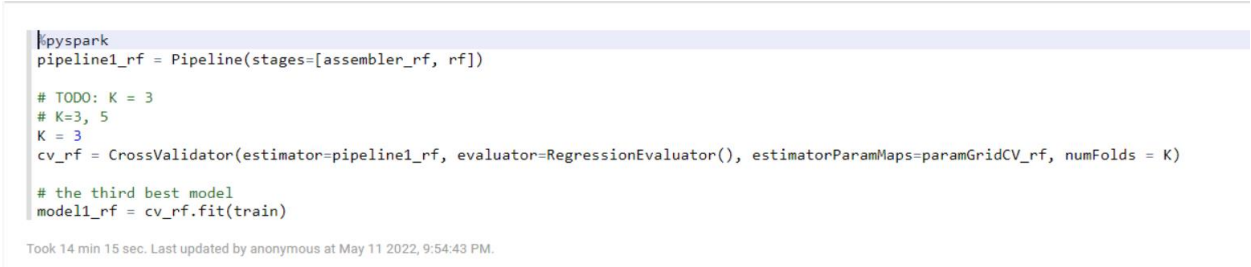
```
%pyspark
# TODO: params refered to the reference above
paramGridCV_rf = ParamGridBuilder() \
  .addGrid(rf.maxDepth, [3, 5]) \
  .addGrid(rf.minInfoGain, [0.0]) \
  .build()
```

```
%pyspark
pipeline1_rf = Pipeline(stages=[assembler_rf, rf])

# TODO: K = 3
# K=3, 5
```

```
K = 3
cv_rf = CrossValidator(estimator=pipeline1_rf,
evaluator=RegressionEvaluator(), estimatorParamMaps=paramGridCV_rf,
numFolds = K)

# the third best model
model1_rf = cv_rf.fit(train)
```



TEST THE MODEL AND EXAMINE THE PREDICTED AND ACTUAL VALUES

Now you're ready to use the transform method of the model to generate some predictions. You can use this approach to predict Item Price where the label is unknown; but in this case you are using the test data which includes a known true label value, so you can compare the predicted Item Price to the actual Item Price.
You can plot the predicted values against the actual values to see how accurately the model has predicted. In a perfect model, the resulting scatter plot should form a perfect diagonal line with each predicted value being identical to the actual value - in practice, some variance is to be expected. Run the cells below to create a temporary table from the predicted DataFrame and then retrieve the predicted and actual label values.

```
%pyspark
prediction1_rf = model1_rf.transform(test)
predicted1_rf = prediction1_rf.select("features", "prediction",
"trueLabel")
predicted1_rf.show(20)
```

CALCULATE CROSS VALIDATOR RMSE AND R2

We will now calculate RMSE and R2 for Random Forest Regression using Cross Validator. There are a number of metrics used to measure the variance between predicted and actual values. Of these, the root mean square error (RMSE) is a commonly used value that is measured in the same units as the prediced and actual values - so in this case, the RMSE indicates the average number of minutes between predicted and actual Item Price Values. You can use the RegressionEvaluator class to retrieve the RMSE . The ideal Accuracy of R2 is 1. The closer the value to 1 the lesser the error, the better the model.

```
%pyspark
#Determining Evaluator RMSE
evaluator1_rf = RegressionEvaluator(labelCol="trueLabel",
predictionCol="prediction", metricName="rmse")
rmse_rf_1 = evaluator1_rf.evaluate(predicted1_rf)
```

```
#Determining Evaluator RMSE
evaluator1_rf = RegressionEvaluator(labelCol="trueLabel",
predictionCol="prediction", metricName="r2")
r2_rf_1 = evaluator1_rf.evaluate(predicted1_rf)


print ("Root Mean Square Error (RMSE)", rmse_rf_1)
print ("Co-efficient of Determination (r2)", r2_rf_1)
```

```
%pyspark
#Determining Evaluator RMSE
evaluator1_rf = RegressionEvaluator(labelCol="trueLabel", predictionCol="prediction", metricName="rmse")
rmse_rf_1 = evaluator1_rf.evaluate(predicted1_rf)

#Determining Evaluator RMSE
evaluator1_rf = RegressionEvaluator(labelCol="trueLabel", predictionCol="prediction", metricName="r2")
r2_rf_1 = evaluator1_rf.evaluate(predicted1_rf)


print ("Root Mean Square Error (RMSE)", rmse_rf_1)
print ("Co-efficient of Determination (r2)", r2_rf_1)
```

```
Root Mean Square Error (RMSE) 220.02916714002336
Co-efficient of Determination (r2) 0.6668797294274402
```

Took 1 min 49 sec. Last updated by anonymous at May 11 2022, 9:58:52 PM.

Step 11: Gradient Boost Tree Regression

Run Gradient Boost Tree Algorithm using Train Split Validation and Cross Validation.

PREPARE THE TRAINING DATA

To train the regression model, you need a training data set that includes a vector of numeric features, and a label column. In this exercise, you will use the VectorAssembler class to transform the feature columns into a vector. We define a pipeline that creates a Feature Vector and trains a regression model.

```
%pyspark
assembler_gbt = VectorAssembler(inputCols = ["product_id",
"brandIndex", "category_codeIndex", "event_type" , "user_id"],
outputCol="features")

#gbt = GBTRegression(labelCol="label", featuresCol="normFeatures")
gbt = GBTRegressor(labelCol="label", featuresCol="features",
maxBins=3000)
```

TRAIN SPLIT VALIDATOR

PARAMETER BUILDING , DEFINE PIPELINE AND TUNE PARAMETERS USING TRAIN SPLIT VALIDATOR

You can tune the parameters to find the best model for your data. To do this use the TraiSplit Validator class to evaluate each combination of parameters defined in a ParameterGrid. Fitting the model takes a

long time to run because every parameter combination is tried. We define a pipeline that creates a Feature Vector and trains a regression model.

```
%pyspark
paramGrid_gbt = ParamGridBuilder() \
  .addGrid(gbt.maxDepth, [2, 3]) \
  .addGrid(gbt.minInfoGain, [0.0]) \
  .build()
```

```
%pyspark
pipeline0_gbt = Pipeline(stages=[assembler_gbt, gbt])
```

```
%pyspark
model = pipeline0_gbt.fit(train)
```

```
%pyspark
rfModel = model.stages[-1]
print(rfModel.toDebugString)
```

FEATURE IMPORTANCE

NOTE: Feature Importance works well in Databricks but does this code does not work in Zepplin.

Feature Importance refers to calculating the score for all the input features for a given model. This score indicates the “importance” of each feature. The higher the score , the larger the impact on model.

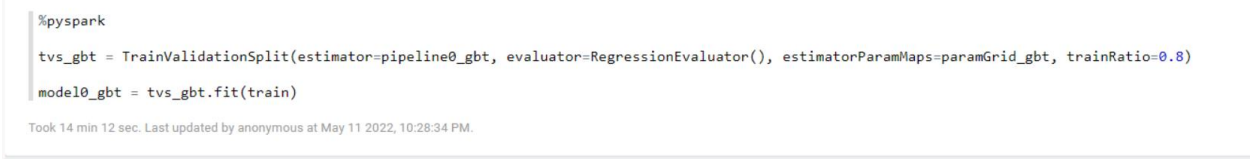
```
%pyspark

featureImp = pd.DataFrame(list(zip(assembler_gbt.getInputCols(),
rfModel.featureImportances)),
columns=["feature", "importance"])
featureImp.sort_values(by="importance", ascending=False)
```

```
%pyspark

tvs_gbt = TrainValidationSplit(estimator=pipeline0_gbt,
evaluator=RegressionEvaluator(), estimatorParamMaps=paramGrid_gbt,
trainRatio=0.8)

model0_gbt = tvs_gbt.fit(train)
```



TEST THE MODEL , EXAMINE THE PREDICTED AND ACTUAL VALUES

Now you're ready to use the transform method of the model to generate some predictions. You can use this approach to predict Item Price where the label is unknown; but in this case you are using the test

data which includes a known true label value, so you can compare the predicted Item Price to the actual Item Price.
You can plot the predicted values against the actual values to see how accurately the model has predicted. In a perfect model, the resulting scatter plot should form a perfect diagonal line with each predicted value being identical to the actual value - in practice, some variance is to be expected. Run the cells below to create a temporary table from the **predicted** DataFrame and then retrieve the predicted and actual label values .

```
%pyspark
prediction0_gbt = model0_gbt.transform(test)
predicted0_gbt = prediction0_gbt.select("features", "prediction",
"trueLabel")
predicted0_gbt.show(20)
```

CALCULATE TRAIN VALIDATION SPLIT RMSE AND R2

We will now calculate RMSE and R2 for Gradient Boost Tree Regression using Train Split Validator. There are a number of metrics used to measure the variance between predicted and actual values. Of these, the root mean square error (RMSE) is a commonly used value that is measured in the same units as the predicted and actual values - so in this case, the RMSE indicates the average number of minutes between predicted and actual Item Price Values. You can use the RegressionEvaluator class to retrieve the RMSE . The ideal Accuracy of R2 is 1. The closer the value to 1 the lesser the error, the better the model

```
%pyspark
#Determining Evaluator RMSE
evaluator0_gbt = RegressionEvaluator(labelCol="trueLabel",
predictionCol="prediction", metricName="rmse")
rmse_gbt_0 = evaluator0_gbt.evaluate(predicted0_gbt)

#Determining Evaluator RMSE
evaluator0_gbt = RegressionEvaluator(labelCol="trueLabel",
predictionCol="prediction", metricName="r2")
r2_gbt_0 = evaluator0_gbt.evaluate(predicted0_gbt)


print ("Root Mean Square Error (RMSE)", rmse_gbt_0)
print ("Co-efficient of Determination (r2)", r2_gbt_0)
```

```
%pyspark
#Determining Evaluator RMSE
evaluator0_gbt = RegressionEvaluator(labelCol="trueLabel", predictionCol="prediction", metricName="rmse")
rmse_gbt_0 = evaluator0_gbt.evaluate(predicted0_gbt)

#Determining Evaluator RMSE
evaluator0_gbt = RegressionEvaluator(labelCol="trueLabel", predictionCol="prediction", metricName="r2")
r2_gbt_0 = evaluator0_gbt.evaluate(predicted0_gbt)


print ("Root Mean Square Error (RMSE)", rmse_gbt_0)
print ("Co-efficient of Determination (r2)", r2_gbt_0)
```

```
Root Mean Square Error (RMSE) 204.2626003189075
Co-efficient of Determination (r2) 0.7129098690079052
```

Took 1 min 39 sec. Last updated by anonymous at May 11 2022, 10:31:37 PM.

CROSS VALIDATOR

PARAMETER BUILDING, DEFINE PIPELINE AND TUNE PARAMETERS USING CROSS VALIDATOR

You can tune the parameters to find the best model for your data. To do this use the Cross Validator class to evaluate each combination of parameters defined in a ParameterGrid against multiple folds of the data split into training and validation datasets, in order to find the best performing parameters. Fitting the model takes a long time to run because every parameter combination is tried multiple times. We define a pipeline that creates a Feature Vector and trains a regression model.

```
%pyspark
# TODO: params refered to the reference above
paramGridCV_gbt = ParamGridBuilder() \
  .addGrid(gbt.maxDepth, [3, 5]) \
  .addGrid(gbt.minInfoGain, [0.0]) \
  .build()
```

```
%pyspark
pipeline1_gbt = Pipeline(stages=[assembler_gbt, gbt])
K = 3
cv_gbt = CrossValidator(estimator=pipeline1_gbt,
evaluator=RegressionEvaluator(), estimatorParamMaps=paramGridCV_gbt,
numFolds = K)

model1_gbt = cv_gbt.fit(train)
```

```
%pyspark
pipeline1_gbt = Pipeline(stages=[assembler_gbt, gbt])
K = 3
cv_gbt = CrossValidator(estimator=pipeline1_gbt, evaluator=RegressionEvaluator(), estimatorParamMaps=paramGridCV_gbt, numFolds = K)

model1_gbt = cv_gbt.fit(train)
```

Took 35 min 58 sec. Last updated by anonymous at May 11 2022, 11:07:46 PM.

TEST THE MODEL AND EXAMINE THE PREDICTED AND ACTUAL VALUES

Now you're ready to use the transform method of the model to generate some predictions. You can use this approach to predict Item Price where the label is unknown; but in this case you are using the test data which includes a known true label value, so you can compare the predicted Item Price to the actual Item Price.
You can plot the predicted values against the actual values to see how accurately the model has predicted. In a perfect model, the resulting scatter plot should form a perfect diagonal line with each predicted value being identical to the actual value - in practice, some variance is to be expected. Run the cells below to create a temporary table from the **predicted** DataFrame and then retrieve the predicted and actual label values.

```
%pyspark
prediction1_gbt = model1_gbt.transform(test)
predicted1_gbt = prediction1_gbt.select("features", "prediction",
"trueLabel")
predicted1_gbt.show(20)
```

CALCULATE CROSS VALIDATOR RMSE AND R2

We will now calculate RMSE and R2 for Gradient Boost Tree Regression using Cross Validator. There are a number of metrics used to measure the variance between predicted and actual values. Of these, the root mean square error (RMSE) is a commonly used value that is measured in the same units as the predicted and actual values - so in this case, the RMSE indicates the average number of minutes between predicted and actual Item Price Values. You can use the RegressionEvaluator class to retrieve the RMSE . The ideal Accuracy of R2 is 1. The closer the value to 1 the lesser the error, the better the model.

```
%pyspark
#Determining Evaluator RMSE
evaluator1_gbt = RegressionEvaluator(labelCol="trueLabel",
predictionCol="prediction", metricName="rmse")
rmse_gbt_1 = evaluator1_gbt.evaluate(predicted1_gbt)

#Determining Evaluator RMSE
evaluator1_gbt = RegressionEvaluator(labelCol="trueLabel",
predictionCol="prediction", metricName="r2")
r2_gbt_1 = evaluator1_gbt.evaluate(predicted1_gbt)


print ("Root Mean Square Error (RMSE)", rmse_gbt_1)
print ("Co-efficient of Determination (r2)", r2_gbt_1)
```

```
%pyspark
#Determining Evaluator RMSE
evaluator1_gbt = RegressionEvaluator(labelCol="trueLabel", predictionCol="prediction", metricName="rmse")
rmse_gbt_1 = evaluator1_gbt.evaluate(predicted1_gbt)

#Determining Evaluator RMSE
evaluator1_gbt = RegressionEvaluator(labelCol="trueLabel", predictionCol="prediction", metricName="r2")
r2_gbt_1 = evaluator1_gbt.evaluate(predicted1_gbt)


print ("Root Mean Square Error (RMSE)", rmse_gbt_1)
print ("Co-efficient of Determination (r2)", r2_gbt_1)
```

```
Root Mean Square Error (RMSE) 168.20361802168077
Co-efficient of Determination (r2) 0.8053245366067966
```

Took 1 min 42 sec. Last updated by anonymous at May 11 2022, 11:10:48 PM.

Step 12: Decision Tree Regression

Run Decision Tree Algorithm using Train Split Validation and Cross Validation.

PREPARE THE TRAINING DATA

To train the regression model, you need a training data set that includes a vector of numeric features, and a label column. In this exercise, you will use the VectorAssembler class to transform the feature columns into a vector. We define a pipeline that creates a Feature Vector and trains a regression model.

```
%pyspark
assembler_dt = VectorAssembler(inputCols = ["product_id",
"brandIndex", "category_codeIndex", "event_type" , "user_id"],
outputCol="features")

#gbt = GBTRegression(labelCol="label", featuresCol="normFeatures")
dt = DecisionTreeRegressor(labelCol="label", featuresCol="features",
maxBins=3000)
```

TRAIN SPLIT VALIDATOR

PARAMETER BUILDING , DEFINE PIPELINE AND TUNE PARAMETERS USING TRAIN SPLIT VALIDATOR

You can tune the parameters to find the best model for your data. To do this use the TrainSplit Validator class to evaluate each combination of parameters defined in a ParameterGrid. Fitting the model takes a long time to run because every parameter combination is tried. We define a pipeline that creates a Feature Vector and trains a regression model.

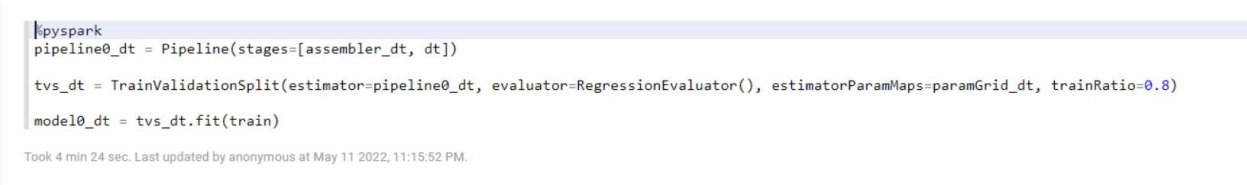
```
%pyspark
paramGrid_dt = ParamGridBuilder() \
  .addGrid(dt.maxDepth, [2, 3]) \
  .addGrid(dt.minInfoGain, [0.0]) \
```

```
   .build()
```

```
%pyspark
pipeline0_dt = Pipeline(stages=[assembler_dt, dt])

tvs_dt = TrainValidationSplit(estimator=pipeline0_dt,
evaluator=RegressionEvaluator(), estimatorParamMaps=paramGrid_dt,
trainRatio=0.8)

model0_dt = tvs_dt.fit(train)
```



TEST THE MODEL AND EXAMINE THE PREDICTED AND ACTUAL VALUES

Now you're ready to use the transform method of the model to generate some predictions. You can use this approach to predict Item Price where the label is unknown; but in this case you are using the test data which includes a known true label value, so you can compare the predicted Item Price to the actual Item Price.
You can plot the predicted values against the actual values to see how accurately the model has predicted. In a perfect model, the resulting scatter plot should form a perfect diagonal line with each predicted value being identical to the actual value - in practice, some variance is to be expected. Run the cells below to create a temporary table from the predicted DataFrame and then retrieve the predicted and actual label values.

```
%pyspark
prediction0_dt = model0_dt.transform(test)
predicted0_dt = prediction0_dt.select("features", "prediction",
"trueLabel")
predicted0_dt.show(20)
```

CALCULATE TRAIN VALIDATION SPLIT RMSE AND R2

We will now calculate RMSE and R2 for Decision Tree Regression using Train Split Validator. There are a number of metrics used to measure the variance between predicted and actual values. Of these, the root mean square error (RMSE) is a commonly used value that is measured in the same units as the predicted and actual values - so in this case, the RMSE indicates the average number of minutes between predicted and actual Item Price Values. You can use the RegressionEvaluator class to retrieve the RMSE . The ideal Accuracy of R2 is 1. The closer the value to 1 the lesser the error, and the better the model.

```
%pyspark
```

```
#Determining Evaluator RMSE
evaluator0_dt = RegressionEvaluator(labelCol="trueLabel",
predictionCol="prediction", metricName="rmse")
rmse_dt_0 = evaluator0_dt.evaluate(predicted0_dt)

#Determining Evaluator RMSE
evaluator0_dt = RegressionEvaluator(labelCol="trueLabel",
predictionCol="prediction", metricName="r2")
r2_dt_0 = evaluator0_dt.evaluate(predicted0_dt)


print ("Root Mean Square Error (RMSE)", rmse_dt_0)
print ("Co-efficient of Determination (r2)", r2_dt_0)
```

CROSS VALIDATOR

PARAMETER BUILDING, DEFINE PIPELINE AND TUNE PARAMETERS USING CROSS VALIDATOR

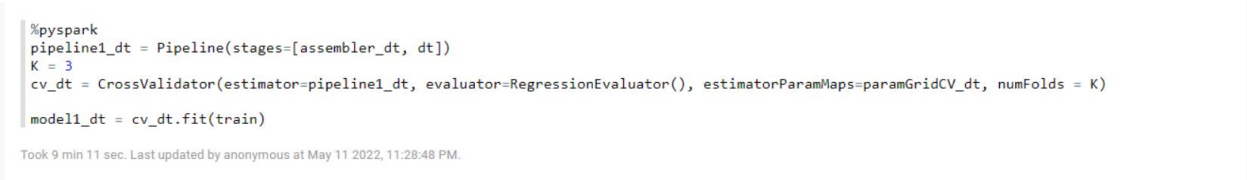
You can tune the parameters to find the best model for your data. To do this use the Cross Validator class to evaluate each combination of parameters defined in a ParameterGrid against multiple folds of the data split into training and validation datasets, in order to find the best performing parameters. Fitting the model takes a long time to run because every parameter combination is tried multiple times. We define a pipeline that creates a Feature Vector and trains a regression model.

```
%pyspark
# TODO: params refered to the reference above
paramGridCV_dt = ParamGridBuilder() \
  .addGrid(dt.maxDepth, [3, 5]) \
  .addGrid(dt.minInfoGain, [0.0]) \
  .build()
```

```
%pyspark
pipeline1_dt = Pipeline(stages=[assembler_dt, dt])
K = 3
```

```
cv_dt = CrossValidator(estimator=pipeline1_dt,
evaluator=RegressionEvaluator(), estimatorParamMaps=paramGridCV_dt,
numFolds = K)

model1_dt = cv_dt.fit(train)
```



TEST THE MODEL AND EXAMINE THE PREDICTED AND ACTUAL VALUES

Now you're ready to use the transform method of the model to generate some predictions. You can use this approach to predict Item Price where the label is unknown; but in this case you are using the test data which includes a known true label value, so you can compare the predicted Item Price to the actual Item Price.
You can plot the predicted values against the actual values to see how accurately the model has predicted. In a perfect model, the resulting scatter plot should form a perfect diagonal line with each predicted value being identical to the actual value - in practice, some variance is to be expected. Run the cells below to create a temporary table from the **predicted** DataFrame and then retrieve the predicted and actual label values.

```
%pyspark
prediction1_dt = model1_dt.transform(test)
predicted1_dt = prediction1_dt.select("features", "prediction",
"trueLabel")
predicted1_dt.show(20)
```

CALCULATE CROSS VALIDATOR RMSE AND R2

We will now calculate RMSE and R2 for Decision Tree Regressor using Cross Validator. There are a number of metrics used to measure the variance between predicted and actual values. Of these, the root mean square error (RMSE) is a commonly used value that is measured in the same units as the predicted and actual values - so in this case, the RMSE indicates the average number of minutes between predicted and actual Item Price Values. You can use the RegressionEvaluator class to retrieve the RMSE . The ideal Accuracy of R2 is 1. The closer the value to 1 the lesser the error, the better the model.

```
%pyspark
#Determining Evaluator RMSE
evaluator1_dt = RegressionEvaluator(labelCol="trueLabel",
predictionCol="prediction", metricName="rmse")
rmse_dt_1 = evaluator1_dt.evaluate(predicted1_dt)

#Determining Evaluator RMSE
evaluator1_dt = RegressionEvaluator(labelCol="trueLabel",
predictionCol="prediction", metricName="r2")
r2_dt_1 = evaluator1_dt.evaluate(predicted1_dt)
```

```
print ("Root Mean Square Error (RMSE)", rmse_dt_1)
print ("Co-efficient of Determination (r2)", r2_dt_1)
```

```
%pyspark
#Determining Evaluator RMSE
evaluator1_dt = RegressionEvaluator(labelCol="trueLabel", predictionCol="prediction", metricName="rmse")
rmse_dt_1 = evaluator1_dt.evaluate(predicted1_dt)

#Determining Evaluator RMSE
evaluator1_dt = RegressionEvaluator(labelCol="trueLabel", predictionCol="prediction", metricName="r2")
r2_dt_1 = evaluator1_dt.evaluate(predicted1_dt)


print ("Root Mean Square Error (RMSE)", rmse_dt_1)
print ("Co-efficient of Determination (r2)", r2_dt_1)
```

```
Root Mean Square Error (RMSE) 224.1105329240129
Co-efficient of Determination (r2) 0.6544068804162189
```

Took 1 min 52 sec. Last updated by anonymous at May 11 2022, 11:35:45 PM.

Step 13: Linear Regression

Run Linear Regression using Train Split Validation and Cross Validation

PREPARE THE TRAINING DATA

To train the regression model, you need a training data set that includes a vector of numeric features, and a label column. In this exercise, you will use the VectorAssembler class to transform the feature columns into a vector. We define a pipeline that creates a Feature Vector and trains a regression model.

```
%pyspark
assembler_lr = VectorAssembler(inputCols = ["product_id",
"brandIndex", "category_codeIndex", "event_type" , "user_id"],
outputCol="features")

#gbt = GBTRegression(labelCol="label", featuresCol="normFeatures")
lr = LinearRegression(labelCol="label", featuresCol="features")
```

TRAIN SPLIT VALIDATOR

PARAMETER BUILDING , DEFINE PIPELINE AND TUNE PARAMETERS USING TRAIN SPLIT VALIDATOR

You can tune the parameters to find the best model for your data. To do this use the TrainSplit Validator class to evaluate each combination of parameters defined in a ParameterGrid. Fitting the model takes a long time to run because every parameter combination is tried. We define a pipeline that creates a Feature Vector and trains a regression model.

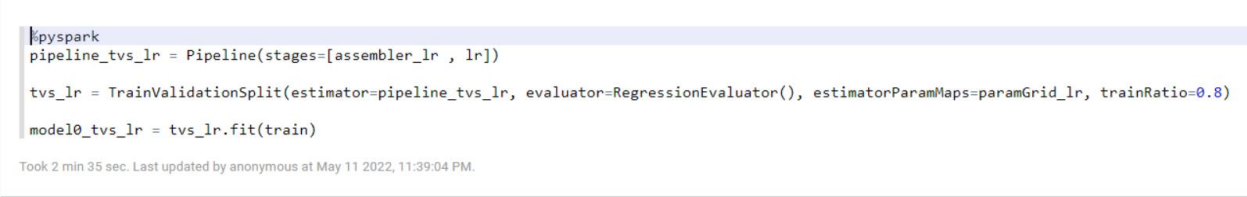
```
%pyspark
paramGrid_lr = ParamGridBuilder() \
  .addGrid(lr.maxIter, [10]) \
```

```
  .addGrid(lr.regParam, [0.3]) \
  .build()
```

```
%pyspark
pipeline_tvs_lr = Pipeline(stages=[assembler_lr , lr])

tvs_lr = TrainValidationSplit(estimator=pipeline_tvs_lr,
evaluator=RegressionEvaluator(), estimatorParamMaps=paramGrid_lr,
trainRatio=0.8)

model0_tvs_lr = tvs_lr.fit(train)
```



TEST THE MODEL AND EXAMINE THE PREDICTED AND ACTUAL VALUES

Now you're ready to use the transform method of the model to generate some predictions. You can use this approach to predict Item Price where the label is unknown; but in this case you are using the test data which includes a known true label value, so you can compare the predicted Item Price to the actual Item Price.
You can plot the predicted values against the actual values to see how accurately the model has predicted. In a perfect model, the resulting scatter plot should form a perfect diagonal line with each predicted value being identical to the actual value - in practice, some variance is to be expected. Run the cells below to create a temporary table from the predicted DataFrame and then retrieve the predicted and actual label values.

```
%pyspark
prediction0_lr = model0_tvs_lr.transform(test)
predicted0_lr = prediction0_lr.select("features", "prediction",
"trueLabel")
predicted0_lr.show(20)
```

CALCULATE TRAIN VALIDATION SPLIT RMSE AND R2

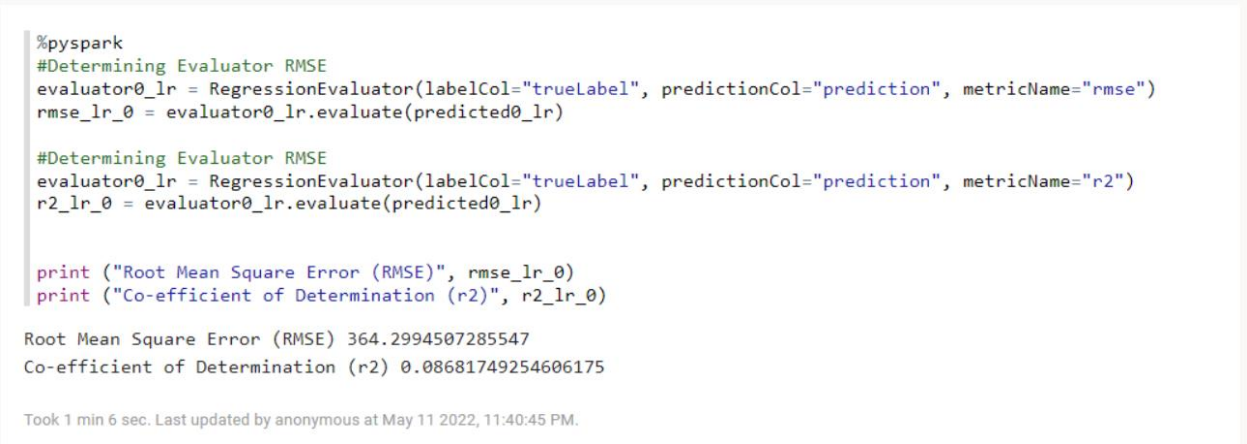
We will now calculate RMSE and R2 for Linear Regression using Train Split Validator. There are a number of metrics used to measure the variance between predicted and actual values. Of these, the root mean square error (RMSE) is a commonly used value that is measured in the same units as the predicted and actual values - so in this case, the RMSE indicates the average number of minutes between the predicted and actual Item Price Values. You can use the RegressionEvaluator class to retrieve the RMSE . The ideal Accuracy of R2 is 1. The closer the value to 1 the lesser the error, and the better the model.

```
%pyspark
#Determining Evaluator RMSE
```

```
evaluator0_lr = RegressionEvaluator(labelCol="trueLabel",
predictionCol="prediction", metricName="rmse")
rmse_lr_0 = evaluator0_lr.evaluate(predicted0_lr)

#Determining Evaluator RMSE
evaluator0_lr = RegressionEvaluator(labelCol="trueLabel",
predictionCol="prediction", metricName="r2")
r2_lr_0 = evaluator0_lr.evaluate(predicted0_lr)


print ("Root Mean Square Error (RMSE)", rmse_lr_0)
print ("Co-efficient of Determination (r2)", r2_lr_0)
```



CROSS VALIDATOR

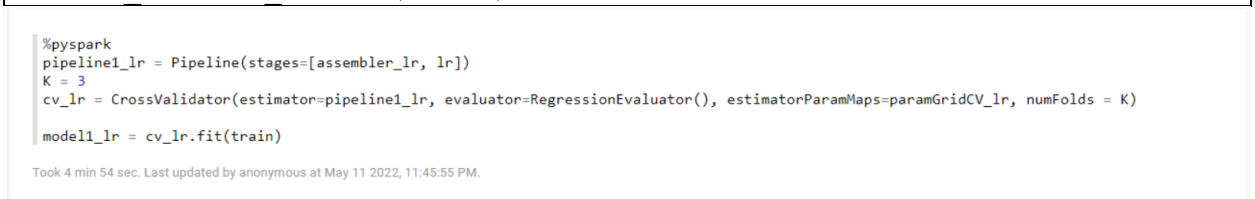
PARAMETER BUIDING AND TRAIN USING CROSS VALIDATOR

You can tune the parameters to find the best model for your data. To do this use the Cross Validator class to evaluate each combination of parameters defined in a ParameterGrid against multiple folds of the data split into training and validation datasets, in order to find the best performing parameters. Fitting the model takes a long time to run because every parameter combination is tried multiple times. We define a pipeline that creates a Feature Vector and trains a regression model.

```
%pyspark
# TODO: params refered to the reference above
paramGridCV_lr = ParamGridBuilder() \
  .addGrid(lr.maxIter, [20]) \
  .addGrid(lr.regParam, [0.5]) \
  .build()
```

```
%pyspark
pipeline1_lr = Pipeline(stages=[assembler_lr, lr])
K = 3
cv_lr = CrossValidator(estimator=pipeline1_lr,
evaluator=RegressionEvaluator(), estimatorParamMaps=paramGridCV_lr,
numFolds = K)
```

```
model1_lr = cv_lr.fit(train)
```



TEST THE MODEL AND EXAMINE THE PREDICTED AND ACTUAL VALUES

Now you're ready to use the transform method of the model to generate some predictions. You can use this approach to predict Item Price where the label is unknown; but in this case you are using the test data which includes a known true label value, so you can compare the predicted Item Price to the actual Item Price.
You can plot the predicted values against the actual values to see how accurately the model has predicted. In a perfect model, the resulting scatter plot should form a perfect diagonal line with each predicted value being identical to the actual value - in practice, some variance is to be expected. Run the cells below to create a temporary table from the **predicted** DataFrame and then retrieve the predicted and actual label values.

```
%pyspark
prediction1_lr = model1_lr.transform(test)
predicted1_lr = prediction1_lr.select("features", "prediction",
"trueLabel")
predicted1_lr.show(20)
```

CALCULATE CROSS VALIDATOR RMSE AND R2

We will now calculate RMSE and R2 for Linear Regression using Cross Validator. There are a number of metrics used to measure the variance between predicted and actual values. Of these, the root mean square error (RMSE) is a commonly used value that is measured in the same units as the prediced and actual values - so in this case, the RMSE indicates the average number of minutes between predicted and actual Item Price Values. You can use the RegressionEvaluator class to retrieve the RMSE . The ideal Accuracy of R2 is 1. The closer the value to 1 the lesser the error, the better the model.

```
%pyspark
#Determining Evaluator RMSE
evaluator1_lr = RegressionEvaluator(labelCol="trueLabel",
predictionCol="prediction", metricName="rmse")
rmse_lr_1 = evaluator1_lr.evaluate(predicted1_lr)

#Determining Evaluator RMSE
evaluator1_lr = RegressionEvaluator(labelCol="trueLabel",
predictionCol="prediction", metricName="r2")
r2_lr_1 = evaluator1_lr.evaluate(predicted1_lr)

print ("Root Mean Square Error (RMSE)", rmse_lr_1)
print ("Co-efficient of Determination (r2)", r2_lr_1)
```

```
%pyspark
#Determining Evaluator RMSE
evaluator1_lr = RegressionEvaluator(labelCol="trueLabel", predictionCol="prediction", metricName="rmse")
rmse_lr_1 = evaluator1_lr.evaluate(predicted1_lr)

#Determining Evaluator RMSE
evaluator1_lr = RegressionEvaluator(labelCol="trueLabel", predictionCol="prediction", metricName="r2")
r2_lr_1 = evaluator1_lr.evaluate(predicted1_lr)


print ("Root Mean Square Error (RMSE)", rmse_lr_1)
print ("Co-efficient of Determination (r2)", r2_lr_1)
```

```
Root Mean Square Error (RMSE) 364.29949657239035
Co-efficient of Determination (r2) 0.0868172627143049
```

Took 1 min 3 sec. Last updated by anonymous at May 11 2022, 11:47:18 PM.

Step 14: Decision Tree Classifier

Run Decision Tree Classifier using Cross-Validation

PREPARE THE TRAINING DATA

To train the regression model, you need a training data set that includes a vector of numeric features, and a label column. In this exercise, you will use the VectorAssembler class to transform the feature columns into a vector. We define a pipeline that creates a Feature Vector and trains a regression model.

```
%pyspark
assembler_dt = VectorAssembler(inputCols=["product_id", "brandIndex",
"category_codeIndex", "price", "user_id" ], outputCol="features")

dt = DecisionTreeClassifier(labelCol="label", featuresCol=
"features", maxBins=3000)
```

CROSS VALIDATOR

PARAMETER BUILDING, DEFINE PIPELINE AND TUNE PARAMETERS USING CROSS VALIDATOR

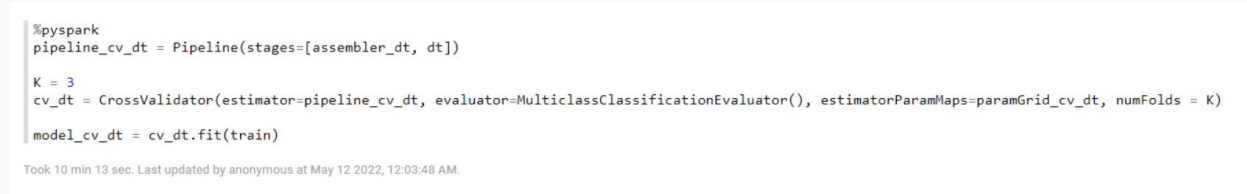
You can tune the parameters to find the best model for your data. To do this use the Cross Validator class to evaluate each combination of parameters defined in a ParameterGrid against multiple folds of the data split into training and validation datasets, in order to find the best performing parameters. Fitting the model takes a long time to run because every parameter combination is tried multiple times. We define a pipeline that creates a Feature Vector and trains a regression model.

```
%pyspark
# TODO: params refered to the reference above
paramGrid_cv_dt = ParamGridBuilder() \
  .addGrid(dt.maxDepth, [3, 5]) \
  .addGrid(dt.minInfoGain, [0.0]) \
  .build()
```

```
%pyspark
pipeline_cv_dt = Pipeline(stages=[assembler_dt, dt])

K = 3
cv_dt = CrossValidator(estimator=pipeline_cv_dt,
evaluator=MulticlassClassificationEvaluator(),
estimatorParamMaps=paramGrid_cv_dt, numFolds = K)

model_cv_dt = cv_dt.fit(train)
```



TEST THE MODEL AND EXAMINE THE PREDICTED AND ACTUAL VALUES

Now you're ready to use the transform method of the model to generate some predictions. You can use this approach to predict Item Price where the label is unknown; but in this case you are using the test data which includes a known true label value, so you can compare the predicted Item Price to the actual Item Price.
You can plot the predicted values against the actual values to see how accurately the model has predicted. In a perfect model, the resulting scatter plot should form a perfect diagonal line with each predicted value being identical to the actual value - in practice, some variance is to be expected. Run the cells below to create a temporary table from the predicted DataFrame and then retrieve the predicted and actual label values.

```
%pyspark
prediction_cv_dt = model_cv_dt.transform(test)
prediction_cv_dt.select("features", "prediction", "trueLabel")
```

CALCULATE ACCURACY, TEST ERROR, PRECISION, RECALL

We will now calculate Accuracy, Test Error , Precision and Recall values for Decision Tree Classifier using Cross Validator. Accuracy is a fraction of Properly predicted cases. Test Error Implies error on a test set. Precision quantifies the number of positive class predictions that actually belong to the positive class. Recall quantifies the number of positive class predictions made out of all positive examples in the dataset

```
%pyspark
evaluator_cv_dt =
MulticlassClassificationEvaluator(labelCol="trueLabel",
predictionCol="prediction", metricName="accuracy")
accuracy_cv_dt = evaluator_cv_dt.evaluate(prediction_cv_dt)
print ("Average Accuracy =", accuracy_cv_dt)
print ("Test Error = ", (1 - accuracy_cv_dt))
```

```
evaluator_cv_dt =
MulticlassClassificationEvaluator(labelCol="trueLabel",
predictionCol="prediction", metricName="weightedPrecision")
precision_cv_dt = evaluator_cv_dt.evaluate(prediction_cv_dt)
print ("Precision =", precision_cv_dt)

evaluator_cv_dt =
MulticlassClassificationEvaluator(labelCol="trueLabel",
predictionCol="prediction", metricName="weightedRecall")
Recall_cv_dt = evaluator_cv_dt.evaluate(prediction_cv_dt)
print ("Recall =", Recall_cv_dt)
```

```
%pyspark
evaluator_cv_dt = MulticlassClassificationEvaluator(labelCol="trueLabel", predictionCol="prediction", metricName="accuracy")
accuracy_cv_dt = evaluator_cv_dt.evaluate(prediction_cv_dt)
print ("Average Accuracy =", accuracy_cv_dt)
print ("Test Error = ", (1 - accuracy_cv_dt))

evaluator_cv_dt = MulticlassClassificationEvaluator(labelCol="trueLabel", predictionCol="prediction", metricName="weightedPrecision")
precision_cv_dt = evaluator_cv_dt.evaluate(prediction_cv_dt)
print ("Precision =", precision_cv_dt)

evaluator_cv_dt = MulticlassClassificationEvaluator(labelCol="trueLabel", predictionCol="prediction", metricName="weightedRecall")
Recall_cv_dt = evaluator_cv_dt.evaluate(prediction_cv_dt)
print ("Recall =", Recall_cv_dt)
```

```
Average Accuracy = 0.9488104877249472
Test Error =  0.05118951227505275
Precision = 0.9002413416168523
Recall = 0.9488104877249472
```

Took 2 min 1 sec. Last updated by anonymous at May 12 2022, 12:05:49 AM.

Step 15: Compare the Results

Once you run all the Regression models. You can now compare the results of all the Regression models. The Results should look like something as below. According to the Comparison table below. We can arrange various Regression Algorithms in the below order.

<u>On the basis of time :</u>

GBT> RF >DT> LR (GBT taking the most time and LR taking the least)

<u>On the basis of accuracy :</u>

GBT>RF>DT> LR (GBT having the best accuracy and LR having the least)

Algorithm	Results	Time taken to fit the model
Linear Regression (LR)	R2: 0.086 RMSE: 364.299	4 min 54 sec
Random Forest Regression (RF)	R2: 0.67 RMSE: 220.029	14 min 15 sec
Decision Tree Regression (DT)	R2: 0.65 RMSE: 224.11	9 min 11 sec
Gradient Boost Tree Regression(GBT)	R2: 0.81 RMSE: 168.20	35 min 58 sec
Decision Tree Classifier	Accuracy: 0.94881 Test Error: 0.05118 Precision: 0.90024 Recall: 0.94881	10 min 13 sec

Thus we can Conclude that even though GBT model takes the most time to run . It is the best model, with R2 closer to 1 and least RMSE value.

References:

1. URL of Data Source: https://www.kaggle.com/datasets/mkechinov/ecommerce-behavior-data-from-multi-category-store?select=2019-Oct.csv
2. URL of your Github: https://github.com/shailjapandit05/CIS-5560-e-Commerce-Prediction-Project
3. URL of References:
    i. https://towardsdatascience.com/multi-class-text-classification-with-pyspark-7d78d022ed35
    ii. https://spark.apache.org/docs/latest/ml-classification-regression.html#regression