CIS5560 Term Project Tutorial

Authors: Ayushi Porwal, Kajal Bhandare, Zalak Patel

Instructor: Jongwook Woo

Date: 05/06/2024

Lab Tutorial

Ayushi Porwal (aporwal@calstatela.edu)
Kajal Bhandare (kbhanda3@calstatela.edu)
Zalak Patel (zpatel6@calstatela.edu)

05/06/2024

US College Net Price Predictive Analysis using Machine Learning Regression Models in Spark ML

Objectives

The objective of the lab is to build a model that predicts the net price of colleges considering the features of public and private colleges using the following machine learning algorithms:

Net Price Prediction

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

Platform Specifications

- Hadoop Version - 3.3.3
- Pyspark Version - 3.2.1
- CPU Speed: 1995.309 MHz
- # of CPU cores: 8
- # of nodes: 5 (2 master nodes, 3 worker nodes)
- Total Memory Size: 806.4 GB

Dataset Specifications

Dataset Name: US Department of Education, College Scorecard

Dataset Size: 2.17 GB

Dataset URL: https://catalog.data.gov/dataset/most-recent-cohorts-scorecard-elements
https://collegescorecard.ed.gov/data/

Dataset Format: CSV

Step 1: Get data manually from the Data source.

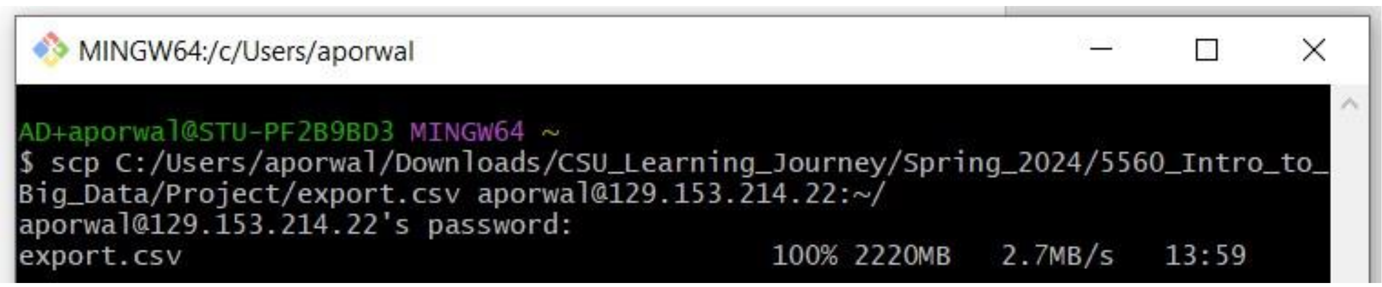
We first need to download the dataset.

1. Download the Dataset from the US Department of Education
   https://collegescorecard.ed.gov/data/

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.

```
scp
C:/Users/aporwal/Downloads/CSU_Learning_Journey/Spring_2024/5560_Intro_to_Big_Da
ta/Project/export.csv aporwal@129.153.214.22:~/
```



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.

```
$ ssh aporwal@129.153.214.22
```

Now enter the password same as username and connect to hadoop cluster

```
AD+aporwal@STU-PF2B9BD3 MINGW64 ~
$ ssh aporwal@129.153.214.22
aporwal@129.153.214.22's password:
Last login: Mon Apr 29 02:45:00 2024 from 35.150.145.118
-bash-4.2$ |
```

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

Step 4: Create a directory "Project" to put the file to HDFS

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

```
hdfs dfs –mkdir Project
```

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

```
hdfs dfs –put export.csv Project/
```

c. Run the following 2 HDFS commands to make sure if export.csv file is uploaded to

Project directory:

```
hdfs dfs -ls
```

```
-bash-4.2$ hdfs dfs -ls
Found 8 items
drwx------   - aporwal hdfs          0 2024-04-13 06:00 .Trash
drwxr-xr-x   - aporwal hdfs          0 2024-04-29 02:45 .sparkStaging
drwxr-xrwx   - aporwal hdfs          0 2024-04-13 09:34 CIS5560
drwxr-xr-x   - aporwal hdfs          0 2024-04-22 00:14 Project
drwxr-xr-x   - aporwal hdfs          0 2024-04-17 22:59 customer
-rw-r--r--   3 aporwal hdfs   72088113 2024-03-25 21:39 flights.csv
drwxr-xr-x   - aporwal hdfs          0 2024-04-17 23:02 movie
-rw-r--r--   3 aporwal hdfs      96368 2024-03-25 21:39 tweets.csv
```

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

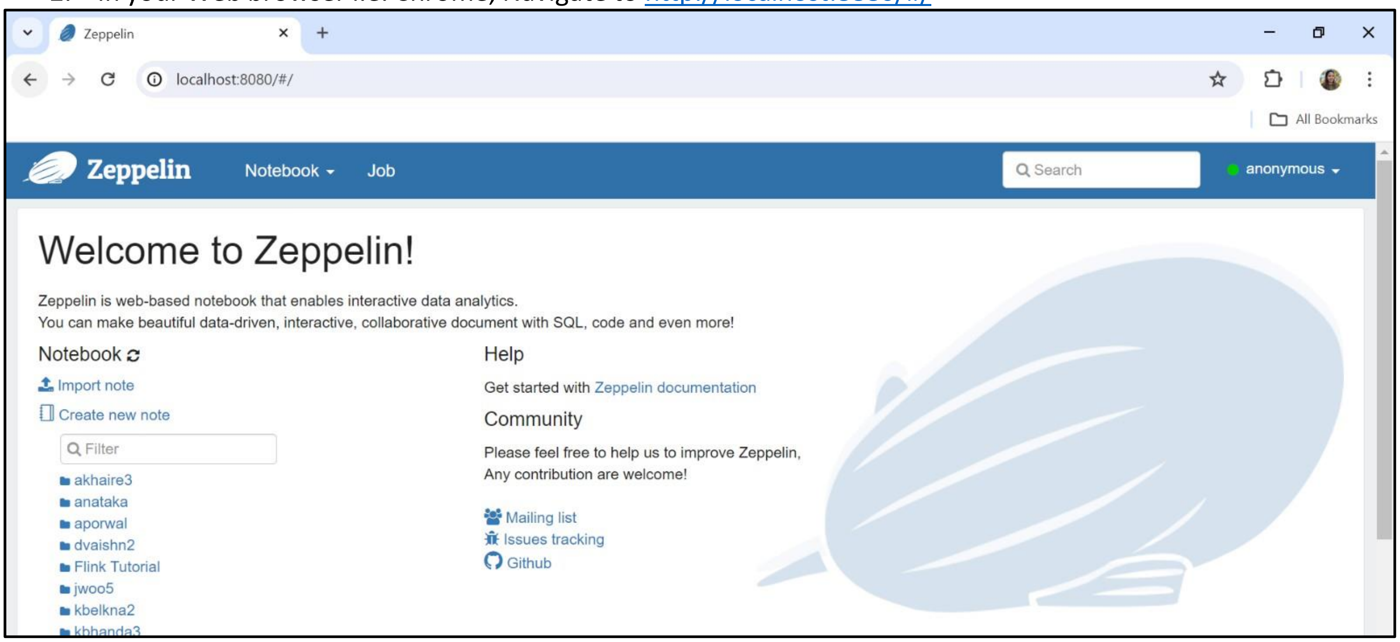
```
-bash-4.2$ hdfs dfs -ls Project/
Found 1 items
-rw-r--r--   3 aporwal hdfs 2327871145 2024-04-22 00:14 Project/export.csv
-bash-4.2$ |
```

Step 5: Login to Zeppelin

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:

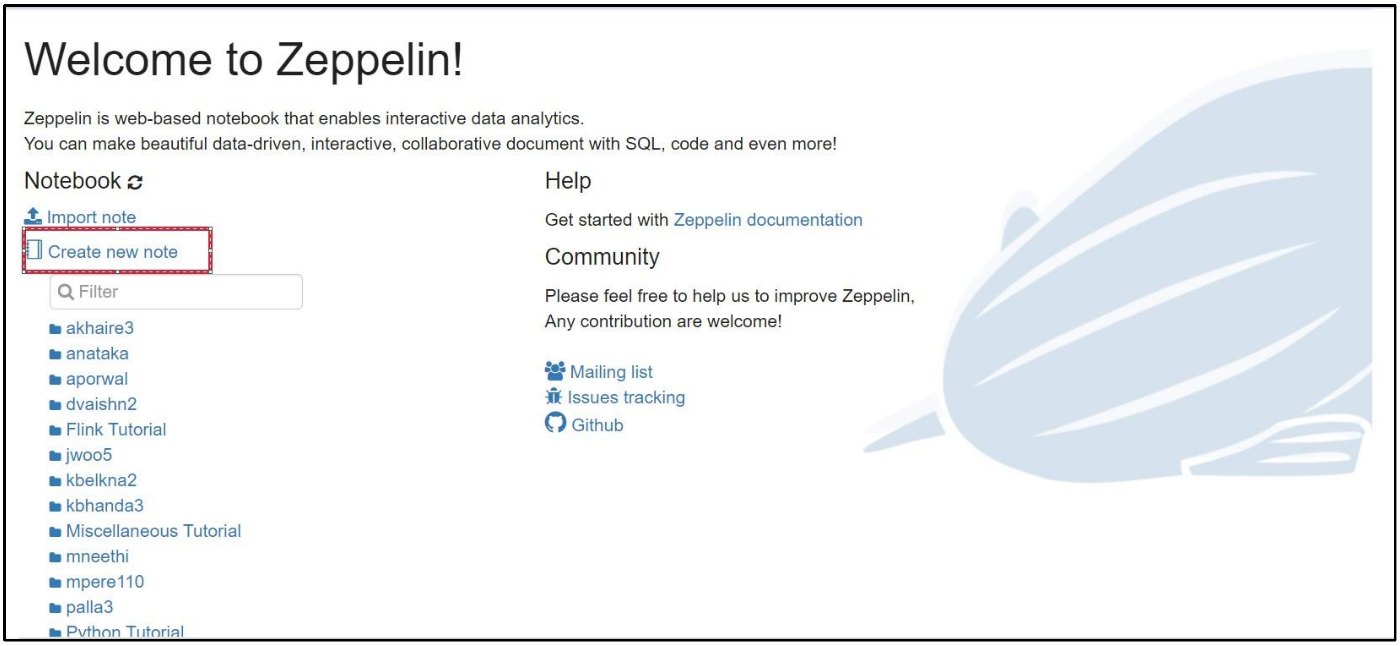
```
ssh -N -L 8080:localhost:8080 kbhanda3@129.153.214.22
```

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



Step 6: Create a new Note in Zeppelin

1. Click on ‘Create new note’ on Zeppelin



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

```
%pyspark
# from pyspark.ml.feature import VectorAssembler

from pyspark.sql.functions import mean, col, when

from pyspark.ml.regression import RandomForestRegressor,GBTRegressor, DecisionTreeRegressor,
LinearRegression
from pyspark.sql.functions import *
from pyspark.ml.feature import *
from pyspark.ml 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

import time
```

```
%pyspark
# from pyspark.ml.feature import VectorAssembler

from pyspark.sql.functions import mean, col, when

from pyspark.ml.regression import RandomForestRegressor,GBTRegressor, DecisionTreeRegressor, LinearRegression
from pyspark.sql.functions import *
from pyspark.ml.feature import *
from pyspark.ml 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

import time
```

Took 0 sec. Last updated by anonymous at May 02 2024, 4:46:20 PM.

Step 7: Load the Files into Dataframe

1. Load files into data frame

```
%pyspark

# File location and type
file_location = "/user/aporwal/Project/export.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.show(10)
```

```
%pyspark

# File location and type
file_location = "/user/aporwal/Project/export.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.show(10)
```

2. Run the following command to sample the Dataset

```
%pyspark
#Create Sample data from raw data
df_college_sample = df.sample(False, 0.1, 60)
df_college_sample.show()
```


```
%pyspark
#Create Sample data from raw data
df_college_sample = df.sample(False, 0.1, 60)
df_college_sample.show()
```


3. Overwrite the sample data to separate CSV

```
%pyspark
#Create CSV file
import pandas as pd
df_college_sample.write.csv("/user/zpatel6/Sample/USscorecard_sample.csv",  header=True,  mode='overwrite')
```

```
%pyspark
#Create CSV file
import pandas as pd

df_college_sample.write.csv("/user/zpatel6/Sample/USscorecard_sample.csv", header=True, mode='overwrite')
```

Took 28 sec. Last updated by anonymous at May 02 2024, 3:02:15 PM.

Step 8: Selecting the features and Label

1. Select features and label columns (i.e. NPT4_PUB, NPT4_PRIV)
2. Create new column (Net_Price) containing the average net prices for the public and the private colleges.
3. Remove the outlines for “Net_Price” label column

```
from pyspark.sql.functions import col, when

columns = ['UNITID','INSTNM','STABBR' , 'LOCALE', 'CONTROL', 'HBCU', 'PBI',
        'ANNHI', 'TRIBAL', 'AANAPII', 'HSI', 'NANTI','SATVRMID', 'SATMTMID' ,'ACTCMMID', 'ACTENMID',
        'ACTMTMID'  , 'SAT_AVG','SAT_AVG_ALL'  , 'MD_EARN_WNE_P10' , 'GT_25K_P6' ,
'GRAD_DEBT_MDN_SUPP'
         , 'RPY_3YR_RT_SUPP', 'NPT4_PUB',
'NPT4_PRIV','COSTT4_A','UGDS','TUITIONFEE_IN','TUITIONFEE_OUT','PCTPELL','LPSTAFFORD_CNT','LPST
AFFORD_AMT','LPPPLUS_CNT','LPPPLUS_AMT','BOOKSUPPLY','ROOMBOARD_ON','OTHEREXPENSE_ON',
'ROOMBOARD_OFF','OTHEREXPENSE_OFF','OTHEREXPENSE_FAM','ENDOWBEGIN','ENDOWEND','ADM_
RATE','ENRL_ORIG_YR2_RT','AGE_ENTRY' ,'UGDS_MEN','UGDS_WOMEN']

# Create a new DataFrame dfp with selected columns
dfp = df.select(*columns)

#Casting values to float
dfp = dfp.withColumn('NPT4_PUB', col('NPT4_PUB').cast("float"))
dfp = dfp.withColumn('NPT4_PRIV', col('NPT4_PRIV').cast("float"))

# Fill missing values with 0 for NPT4_PUB and NPT4_PRIV columns
dfp = dfp.withColumn('NPT4_PUB', when(col('NPT4_PUB').isNull(), 0).otherwise(col('NPT4_PUB')))
dfp = dfp.withColumn('NPT4_PRIV', when(col('NPT4_PRIV').isNull(), 0).otherwise(col('NPT4_PRIV')))

# Calculate Net_Price by summing NPT4_PUB and NPT4_PRIV
dfp = dfp.withColumn('Net_Price', col('NPT4_PUB') + col('NPT4_PRIV'))

# Remove outliers and replace with NaN
dfp = dfp.withColumn('Net_Price', when((col('Net_Price') < 1) | (col('Net_Price') > 55000),
None).otherwise(col('Net_Price')))
```

```
%pyspark

from pyspark.sql.functions import col, when

columns = ['UNITID','INSTNM','STABBR' , 'LOCALE', 'CONTROL', 'HBCU', 'PBI',
           'ANNHI', 'TRIBAL', 'AANAPII', 'HSI', 'NANTI','SATVRMID', 'SATMTMID' ,'ACTCMMID', 'ACTENMID',
           'ACTMTMID'  , 'SAT_AVG','SAT_AVG_ALL'  , 'MD_EARN_WNE_P10' , 'GT_25K_P6' , 'GRAD_DEBT_MDN_SUPP'
            , 'RPY_3YR_RT_SUPP', 'NPT4_PUB', 'NPT4_PRIV','COSTT4_A','UGDS','TUITIONFEE_IN','TUITIONFEE_OUT','PCTPELL','LPSTAFFORD_CNT','LPSTAFFORD_AMT','LPPPLUS_CNT','LPPPLUS_AMT'
                ,'BOOKSUPPLY','ROOMBOARD_ON','OTHEREXPENSE_ON','ROOMBOARD_OFF','OTHEREXPENSE_OFF','OTHEREXPENSE_FAM','ENDOWBEGIN','ENDOWEND','ADM_RATE','ENRL_ORIG_YR2_RT'
                ,'AGE_ENTRY' ,'UGDS_MEN','UGDS_WOMEN']

# Create a new DataFrame dfp with selected columns
dfp = df.select(*columns)

#Casting values to float
dfp = dfp.withColumn('NPT4_PUB', col('NPT4_PUB').cast("float"))
dfp = dfp.withColumn('NPT4_PRIV', col('NPT4_PRIV').cast("float"))

# Fill missing values with 0 for NPT4_PUB and NPT4_PRIV columns
dfp = dfp.withColumn('NPT4_PUB', when(col('NPT4_PUB').isNull(), 0).otherwise(col('NPT4_PUB')))
dfp = dfp.withColumn('NPT4_PRIV', when(col('NPT4_PRIV').isNull(), 0).otherwise(col('NPT4_PRIV')))

# Calculate Net_Price by summing NPT4_PUB and NPT4_PRIV
dfp = dfp.withColumn('Net_Price', col('NPT4_PUB') + col('NPT4_PRIV'))

# Remove outliers and replace with NaN
dfp = dfp.withColumn('Net_Price', when((col('Net_Price') < 1) | (col('Net_Price') > 55000), None).otherwise(col('Net_Price')))
```

FINISHED

Took 2 sec. Last updated by anonymous at May 02 2024, 6:24:34 PM.

Step 9: Data Manipulation pipeline

1. Drop rows containing null values
2. Drop unwanted ‘NPT4_PUB ‘ and ‘NPT4_PRIV’ columns since we have calculated Net_price
3. Remove the 'PrivacySuppressed' and replace it with nan values in the (Earning, Aid and repayment) Columns

```
%pyspark

#Data cleaning

# Drop rows containing null values
dfp = dfp.na.drop()

#Drop unwanted columns since we have calculated Net_price
dfp = dfp.drop('NPT4_PUB', 'NPT4_PRIV')

#remove "PrivacySupressed" values and replace it with null

clean_columns = ['MD_EARN_WNE_P10', 'GT_25K_P6', 
'GRAD_DEBT_MDN_SUPP','RPY_3YR_RT_SUPP','LPSTAFFORD_CNT','LPSTAFFORD_AMT','LPPPLUS_CNT','
LPPPLUS_AMT','ADM_RATE','ENRL_ORIG_YR2_RT','AGE_ENTRY','UGDS_MEN','UGDS_WOMEN']

for column in clean_columns:
   dfp = dfp.withColumn(column, when(dfp[column].isin(['PrivacySuppressed']), 
None).otherwise(dfp[column].cast("float")))

# Show the updated DataFrame
dfp.show(2)
```

```
%pyspark

#Data cleaning

# Drop rows containing null values
dfp = dfp.na.drop()

#Drop unwanted columns since we have calculated Net_price
dfp = dfp.drop('NPT4_PUB', 'NPT4_PRIV')

#remove "PrivacySupressed" values and replace it with null

clean_columns = ['MD_EARN_WNE_P10', 'GT_25K_P6', 'GRAD_DEBT_MDN_SUPP','RPY_3YR_RT_SUPP','LPSTAFFORD_CNT','LPSTAFFORD_AMT','LPPPLUS_CNT','LPPPLUS_AMT','ADM_RATE','ENRL_ORIG_YR2_RT'
    ,'AGE_ENTRY','UGDS_MEN','UGDS_WOMEN']

for column in clean_columns:
    dfp = dfp.withColumn(column, when(dfp[column].isin(['PrivacySuppressed']), None).otherwise(dfp[column].cast("float")))

# Show the updated DataFrame
dfp.show(2)
```

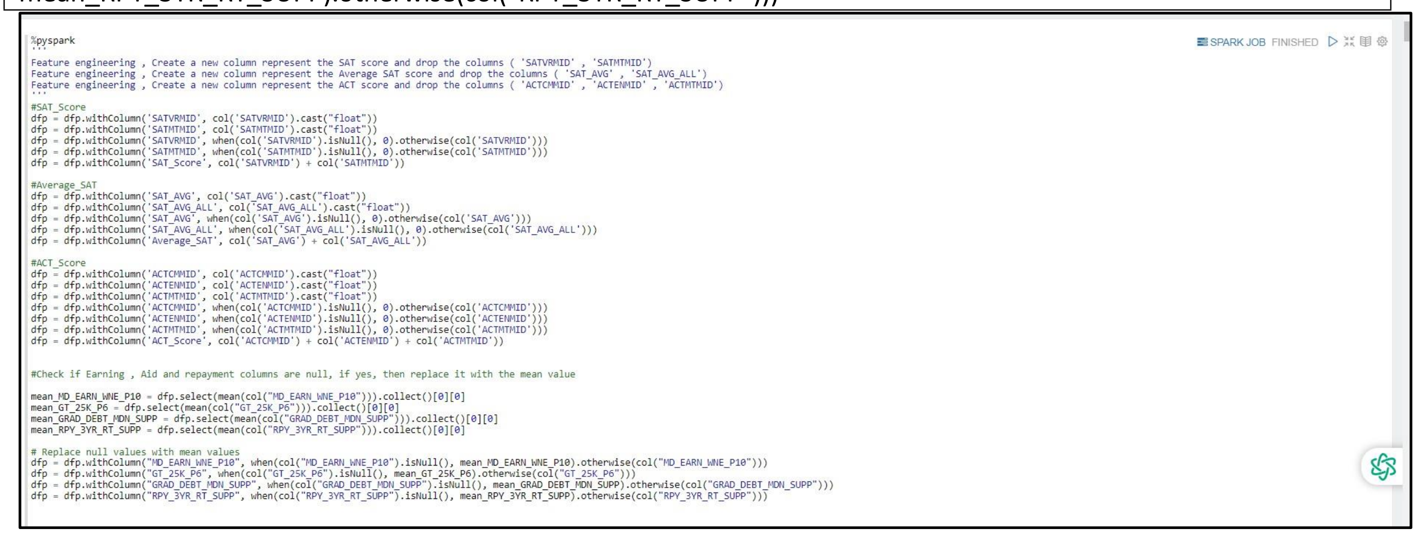
```
|UNITID|              INSTNM|STABBR|LOCALE|CONTROL|HBCU| PBI|ANNHI|TRIBAL|AANAPII| HSI|NANTI|SATVRMID|SATMTMID|ACTCMMID|ACTENMID|ACTMTMID|SAT_AVG|SAT_AVG_ALL|MD_EARN_WNE_P10|GT_25
K_P6|GRAD_DEBT_MDN_SUPP|RPY_3YR_RT_SUPP|COSTT4_A| UGDS|TUITIONFEE_IN|TUITIONFEE_OUT|PCTPELL|LPSTAFFORD_CNT|LPSTAFFORD_AMT|LPPPLUS_CNT|LPPPLUS_AMT|BOOKSUPPLY|ROOMBOARD_ON|OTHEREXPE
NSE_ON|ROOMBOARD_OFF|OTHEREXPENSE_OFF|OTHEREXPENSE_FAM|ENDOWBEGIN| ENDOWEND|ADM_RATE|ENRL_ORIG_YR2_RT|AGE_ENTRY|UGDS_MEN|UGDS_WOMEN|Net_Price|

|100654|Alabama A & M Uni...|    AL|  NULL|      1|NULL|NULL| NULL|  NULL|   NULL|NULL| NULL|     425|     420|      18|      18|      17|    849|        849|           null|
null|          24449.5|     0.35731187|   13762| 4930|         5800|         10672| 0.6317|          null|          null|       null|       null|      1800|        5530|
```

Step 10: Feature Engineering

1. Create a new column represent the SAT score and drop the columns ('SATVRMID' , 'SATMTMID')
2. Create a new column represent the Average SAT score and drop the columns ('SAT_AVG' , 'SAT_AVG_ALL')
3. Create a new column represent the ACT score and drop the columns ('ACTCMMID' , 'ACTENMID' , 'ACTMTMID')
4. Check if Earning, Aid and repayment columns are null, if yes, then replace it with the mean value

```
%pyspark
'''
Feature engineering, Create a new column represent the SAT score and drop the columns ( 'SATVRMID' ,
'SATMTMID')
Feature engineering, Create a new column represent the Average SAT score and drop the columns ( 'SAT_AVG' ,
'SAT_AVG_ALL')
Feature engineering, Create a new column represent the ACT score and drop the columns ( 'ACTCMMID' ,
'ACTENMID' , 'ACTMTMID')
'''
#SAT_Score
dfp = dfp.withColumn('SATVRMID', col('SATVRMID').cast("float"))
dfp = dfp.withColumn('SATMTMID', col('SATMTMID').cast("float"))
dfp = dfp.withColumn('SATVRMID', when(col('SATVRMID').isNull(), 0).otherwise(col('SATVRMID')))
dfp = dfp.withColumn('SATMTMID', when(col('SATMTMID').isNull(), 0).otherwise(col('SATMTMID')))
dfp = dfp.withColumn('SAT_Score', col('SATVRMID') + col('SATMTMID'))

#Average_SAT
dfp = dfp.withColumn('SAT_AVG', col('SAT_AVG').cast("float"))
dfp = dfp.withColumn('SAT_AVG_ALL', col('SAT_AVG_ALL').cast("float"))
dfp = dfp.withColumn('SAT_AVG', when(col('SAT_AVG').isNull(), 0).otherwise(col('SAT_AVG')))
dfp = dfp.withColumn('SAT_AVG_ALL', when(col('SAT_AVG_ALL').isNull(), 0).otherwise(col('SAT_AVG_ALL')))
```

```
dfp = dfp.withColumn('Average_SAT', col('SAT_AVG') + col('SAT_AVG_ALL'))

#ACT_Score
dfp = dfp.withColumn('ACTCMMID', col('ACTCMMID').cast("float"))
dfp = dfp.withColumn('ACTENMID', col('ACTENMID').cast("float"))
dfp = dfp.withColumn('ACTMTMID', col('ACTMTMID').cast("float"))
dfp = dfp.withColumn('ACTCMMID', when(col('ACTCMMID').isNull(), 0).otherwise(col('ACTCMMID')))
dfp = dfp.withColumn('ACTENMID', when(col('ACTENMID').isNull(), 0).otherwise(col('ACTENMID')))
dfp = dfp.withColumn('ACTMTMID', when(col('ACTMTMID').isNull(), 0).otherwise(col('ACTMTMID')))
dfp = dfp.withColumn('ACT_Score', col('ACTCMMID') + col('ACTENMID') + col('ACTMTMID'))


#Check if Earning , Aid and repayment columns are null, if yes, then replace it with the mean value

mean_MD_EARN_WNE_P10 = dfp.select(mean(col("MD_EARN_WNE_P10"))).collect()[0][0]
mean_GT_25K_P6 = dfp.select(mean(col("GT_25K_P6"))).collect()[0][0]
mean_GRAD_DEBT_MDN_SUPP = dfp.select(mean(col("GRAD_DEBT_MDN_SUPP"))).collect()[0][0]
mean_RPY_3YR_RT_SUPP = dfp.select(mean(col("RPY_3YR_RT_SUPP"))).collect()[0][0]

# Replace null values with mean values
dfp = dfp.withColumn("MD_EARN_WNE_P10", when(col("MD_EARN_WNE_P10").isNull(),
mean_MD_EARN_WNE_P10).otherwise(col("MD_EARN_WNE_P10")))
dfp = dfp.withColumn("GT_25K_P6", when(col("GT_25K_P6").isNull(),
mean_GT_25K_P6).otherwise(col("GT_25K_P6")))
dfp = dfp.withColumn("GRAD_DEBT_MDN_SUPP", when(col("GRAD_DEBT_MDN_SUPP").isNull(),
mean_GRAD_DEBT_MDN_SUPP).otherwise(col("GRAD_DEBT_MDN_SUPP")))
dfp = dfp.withColumn("RPY_3YR_RT_SUPP", when(col("RPY_3YR_RT_SUPP").isNull(),
mean_RPY_3YR_RT_SUPP).otherwise(col("RPY_3YR_RT_SUPP")))
```



```
%pyspark
'''
Feature engineering , Create a new column represent the SAT score and drop the columns ( 'SATVRMID' , 'SATMTMID')
Feature engineering , Create a new column represent the Average SAT score and drop the columns ( 'SAT_AVG' , 'SAT_AVG_ALL')
Feature engineering , Create a new column represent the ACT score and drop the columns ( 'ACTCMMID' , 'ACTENMID' , 'ACTMTMID')
'''
#SAT_Score
dfp = dfp.withColumn('SATVRMID', col('SATVRMID').cast("float"))
dfp = dfp.withColumn('SATMTMID', col('SATMTMID').cast("float"))
dfp = dfp.withColumn('SATVRMID', when(col('SATVRMID').isNull(), 0).otherwise(col('SATVRMID')))
dfp = dfp.withColumn('SATMTMID', when(col('SATMTMID').isNull(), 0).otherwise(col('SATMTMID')))
dfp = dfp.withColumn('SAT_Score', col('SATVRMID') + col('SATMTMID'))

#Average_SAT
dfp = dfp.withColumn('SAT_AVG', col('SAT_AVG').cast("float"))
dfp = dfp.withColumn('SAT_AVG_ALL', col('SAT_AVG_ALL').cast("float"))
dfp = dfp.withColumn('SAT_AVG', when(col('SAT_AVG').isNull(), 0).otherwise(col('SAT_AVG')))
dfp = dfp.withColumn('SAT_AVG_ALL', when(col('SAT_AVG_ALL').isNull(), 0).otherwise(col('SAT_AVG_ALL')))
dfp = dfp.withColumn('Average_SAT', col('SAT_AVG') + col('SAT_AVG_ALL'))

#ACT_Score
dfp = dfp.withColumn('ACTCMMID', col('ACTCMMID').cast("float"))
dfp = dfp.withColumn('ACTENMID', col('ACTENMID').cast("float"))
dfp = dfp.withColumn('ACTMTMID', col('ACTMTMID').cast("float"))
dfp = dfp.withColumn('ACTCMMID', when(col('ACTCMMID').isNull(), 0).otherwise(col('ACTCMMID')))
dfp = dfp.withColumn('ACTENMID', when(col('ACTENMID').isNull(), 0).otherwise(col('ACTENMID')))
dfp = dfp.withColumn('ACTMTMID', when(col('ACTMTMID').isNull(), 0).otherwise(col('ACTMTMID')))
dfp = dfp.withColumn('ACT_Score', col('ACTCMMID') + col('ACTENMID') + col('ACTMTMID'))


#Check if Earning , Aid and repayment columns are null, if yes, then replace it with the mean value

mean_MD_EARN_WNE_P10 = dfp.select(mean(col("MD_EARN_WNE_P10"))).collect()[0][0]
mean_GT_25K_P6 = dfp.select(mean(col("GT_25K_P6"))).collect()[0][0]
mean_GRAD_DEBT_MDN_SUPP = dfp.select(mean(col("GRAD_DEBT_MDN_SUPP"))).collect()[0][0]
mean_RPY_3YR_RT_SUPP = dfp.select(mean(col("RPY_3YR_RT_SUPP"))).collect()[0][0]

# Replace null values with mean values
dfp = dfp.withColumn("MD_EARN_WNE_P10", when(col("MD_EARN_WNE_P10").isNull(), mean_MD_EARN_WNE_P10).otherwise(col("MD_EARN_WNE_P10")))
dfp = dfp.withColumn("GT_25K_P6", when(col("GT_25K_P6").isNull(), mean_GT_25K_P6).otherwise(col("GT_25K_P6")))
dfp = dfp.withColumn("GRAD_DEBT_MDN_SUPP", when(col("GRAD_DEBT_MDN_SUPP").isNull(), mean_GRAD_DEBT_MDN_SUPP).otherwise(col("GRAD_DEBT_MDN_SUPP")))
dfp = dfp.withColumn("RPY_3YR_RT_SUPP", when(col("RPY_3YR_RT_SUPP").isNull(), mean_RPY_3YR_RT_SUPP).otherwise(col("RPY_3YR_RT_SUPP")))
```

5. Check if COSTT4_A, UGDS, loan, tuition and expenses, student demographics, addmission_rate, age, gender columns are null, if yes, then replace it with the mean value.

```
%pyspark
#COSTT4_A
dfp = dfp.withColumn('COSTT4_A', col('COSTT4_A').cast("float"))
```

```
#UGDS
dfp = dfp.withColumn('UGDS', col('UGDS').cast("float"))

#loan, tuition and expenses
dfp = dfp.withColumn('TUITIONFEE_IN', col('TUITIONFEE_IN').cast("float"))
dfp = dfp.withColumn('TUITIONFEE_OUT', col('TUITIONFEE_OUT').cast("float"))
dfp = dfp.withColumn('PCTPELL', col('PCTPELL').cast("float"))
dfp = dfp.withColumn('LPSTAFFORD_CNT', col('LPSTAFFORD_CNT').cast("float"))
dfp = dfp.withColumn('LPSTAFFORD_AMT', col('LPSTAFFORD_AMT').cast("float"))
dfp = dfp.withColumn('LPPPLUS_CNT', col('LPPPLUS_CNT').cast("float"))
dfp = dfp.withColumn('LPPPLUS_AMT', col('LPPPLUS_AMT').cast("float"))
dfp = dfp.withColumn('BOOKSUPPLY', col('BOOKSUPPLY').cast("float"))
dfp = dfp.withColumn('ROOMBOARD_ON', col('ROOMBOARD_ON').cast("float"))
dfp = dfp.withColumn('OTHEREXPENSE_ON', col('OTHEREXPENSE_ON').cast("float"))
dfp = dfp.withColumn('ROOMBOARD_OFF', col('ROOMBOARD_OFF').cast("float"))
dfp = dfp.withColumn('OTHEREXPENSE_OFF', col('OTHEREXPENSE_OFF').cast("float"))
dfp = dfp.withColumn('OTHEREXPENSE_FAM', col('OTHEREXPENSE_FAM').cast("float"))
dfp = dfp.withColumn('ENDOWBEGIN', col('ENDOWBEGIN').cast("float"))
dfp = dfp.withColumn('ENDOWEND', col('ENDOWEND').cast("float"))

#student demographics, addmission_rate, age, gender

dfp = dfp.withColumn('ADM_RATE', col('ADM_RATE').cast("float"))
dfp = dfp.withColumn('ENRL_ORIG_YR2_RT', col('ENRL_ORIG_YR2_RT').cast("float"))
dfp = dfp.withColumn('AGE_ENTRY', col('AGE_ENTRY').cast("float"))
dfp = dfp.withColumn('UGDS_MEN', col('UGDS_MEN').cast("float"))
dfp = dfp.withColumn('UGDS_WOMEN', col('UGDS_WOMEN').cast("float"))
```

```
%pyspark
#replace null and 0's with the mean value

mean_COSTT4_A = dfp.select(mean(col("COSTT4_A"))).collect()[0][0]
mean_TUITIONFEE_IN = dfp.select(mean(col("TUITIONFEE_IN"))).collect()[0][0]
mean_TUITIONFEE_OUT = dfp.select(mean(col("TUITIONFEE_OUT"))).collect()[0][0]
mean_PCTPELL = dfp.select(mean(col("PCTPELL"))).collect()[0][0]
mean_LPSTAFFORD_CNT = dfp.select(mean(col("LPSTAFFORD_CNT"))).collect()[0][0]
mean_LPSTAFFORD_AMT = dfp.select(mean(col("LPSTAFFORD_AMT"))).collect()[0][0]
mean_LPPPLUS_CNT = dfp.select(mean(col("LPPPLUS_CNT"))).collect()[0][0]
mean_LPPPLUS_AMT = dfp.select(mean(col("LPPPLUS_AMT"))).collect()[0][0]
mean_BOOKSUPPLY = dfp.select(mean(col("BOOKSUPPLY"))).collect()[0][0]
mean_ROOMBOARD_ON = dfp.select(mean(col("ROOMBOARD_ON"))).collect()[0][0]
mean_OTHEREXPENSE_ON = dfp.select(mean(col("OTHEREXPENSE_ON"))).collect()[0][0]
mean_ROOMBOARD_OFF = dfp.select(mean(col("ROOMBOARD_OFF"))).collect()[0][0]
mean_OTHEREXPENSE_OFF = dfp.select(mean(col("OTHEREXPENSE_OFF"))).collect()[0][0]
mean_OTHEREXPENSE_FAM = dfp.select(mean(col("OTHEREXPENSE_FAM"))).collect()[0][0]
mean_ENDOWBEGIN = dfp.select(mean(col("ENDOWBEGIN"))).collect()[0][0]
mean_ENDOWEND = dfp.select(mean(col("ENDOWEND"))).collect()[0][0]
mean_UGDS = dfp.select(mean(col("UGDS"))).collect()[0][0]
mean_ADM_RATE = dfp.select(mean(col("ADM_RATE"))).collect()[0][0]
```

```
mean_ENRL_ORIG_YR2_RT = dfp.select(mean(col("ENRL_ORIG_YR2_RT"))).collect()[0][0]
mean_AGE_ENTRY = dfp.select(mean(col("AGE_ENTRY"))).collect()[0][0]
mean_UGDS_MEN = dfp.select(mean(col("UGDS_MEN"))).collect()[0][0]
mean_UGDS_WOMEN= dfp.select(mean(col("UGDS_WOMEN"))).collect()[0][0]



# Replace null values with mean values
dfp = dfp.withColumn("COSTT4_A", when((col("COSTT4_A").isNull()) | (col("COSTT4_A") == 0.0),
mean_COSTT4_A).otherwise(col("COSTT4_A")))
dfp = dfp.withColumn("UGDS", when((col("UGDS").isNull()) | (col("UGDS") == 0.0),
mean_UGDS).otherwise(col("UGDS")))
dfp = dfp.withColumn("TUITIONFEE_IN", when((col("TUITIONFEE_IN").isNull()) | (col("TUITIONFEE_IN") == 0.0),
mean_TUITIONFEE_IN).otherwise(col("TUITIONFEE_IN")))
dfp = dfp.withColumn("TUITIONFEE_OUT", when((col("TUITIONFEE_OUT").isNull()) | (col("TUITIONFEE_OUT") ==
0.0), mean_TUITIONFEE_OUT).otherwise(col("TUITIONFEE_OUT")))
dfp = dfp.withColumn("PCTPELL", when((col("PCTPELL").isNull()) | (col("PCTPELL") == 0.0),
mean_PCTPELL).otherwise(col("PCTPELL")))
dfp = dfp.withColumn("LPSTAFFORD_CNT", when((col("LPSTAFFORD_CNT").isNull()) | (col("LPSTAFFORD_CNT")
== 0.0), mean_LPSTAFFORD_CNT).otherwise(col("LPSTAFFORD_CNT")))
dfp = dfp.withColumn("LPSTAFFORD_AMT", when((col("LPSTAFFORD_AMT").isNull()) | (col("LPSTAFFORD_AMT")
== 0.0), mean_LPSTAFFORD_AMT).otherwise(col("LPSTAFFORD_AMT")))
dfp = dfp.withColumn("LPPPLUS_CNT", when((col("LPPPLUS_CNT").isNull()) | (col("LPPPLUS_CNT") == 0.0),
mean_LPPPLUS_CNT).otherwise(col("LPPPLUS_CNT")))
dfp = dfp.withColumn("LPPPLUS_AMT", when((col("LPPPLUS_AMT").isNull()) | (col("LPPPLUS_AMT") == 0.0),
mean_LPPPLUS_AMT).otherwise(col("LPPPLUS_AMT")))
dfp = dfp.withColumn("BOOKSUPPLY", when((col("BOOKSUPPLY").isNull()) | (col("BOOKSUPPLY") == 0.0),
mean_BOOKSUPPLY).otherwise(col("BOOKSUPPLY")))
dfp = dfp.withColumn("ROOMBOARD_ON", when((col("ROOMBOARD_ON").isNull()) | (col("ROOMBOARD_ON")
== 0.0), mean_ROOMBOARD_ON).otherwise(col("ROOMBOARD_ON")))
dfp = dfp.withColumn("OTHEREXPENSE_ON", when((col("OTHEREXPENSE_ON").isNull()) |
(col("OTHEREXPENSE_ON") == 0.0), mean_OTHEREXPENSE_ON).otherwise(col("OTHEREXPENSE_ON")))
dfp = dfp.withColumn("ROOMBOARD_OFF", when((col("ROOMBOARD_OFF").isNull()) |
(col("ROOMBOARD_OFF") == 0.0), mean_ROOMBOARD_OFF).otherwise(col("ROOMBOARD_OFF")))
dfp = dfp.withColumn("OTHEREXPENSE_OFF", when((col("OTHEREXPENSE_OFF").isNull()) |
(col("OTHEREXPENSE_OFF") == 0.0), mean_OTHEREXPENSE_OFF).otherwise(col("OTHEREXPENSE_OFF")))
dfp = dfp.withColumn("OTHEREXPENSE_FAM", when((col("OTHEREXPENSE_FAM").isNull()) |
(col("OTHEREXPENSE_FAM") == 0.0), mean_OTHEREXPENSE_FAM).otherwise(col("OTHEREXPENSE_FAM")))
dfp = dfp.withColumn("ENDOWBEGIN", when((col("ENDOWBEGIN").isNull()) | (col("ENDOWBEGIN") == 0.0),
mean_ENDOWBEGIN).otherwise(col("ENDOWBEGIN")))
dfp = dfp.withColumn("ENDOWEND", when((col("ENDOWEND").isNull()) | (col("ENDOWEND") == 0.0),
mean_ENDOWEND).otherwise(col("ENDOWEND")))
dfp = dfp.withColumn("ADM_RATE", when((col("ADM_RATE").isNull()) | (col("ADM_RATE") == 0.0),
mean_ADM_RATE).otherwise(col("ADM_RATE")))
dfp = dfp.withColumn("ENRL_ORIG_YR2_RT", when((col("ENRL_ORIG_YR2_RT").isNull()) |
(col("ENRL_ORIG_YR2_RT") == 0.0), mean_ENRL_ORIG_YR2_RT).otherwise(col("ENRL_ORIG_YR2_RT")))
dfp = dfp.withColumn("AGE_ENTRY", when((col("AGE_ENTRY").isNull()) | (col("AGE_ENTRY") == 0.0),
mean_AGE_ENTRY).otherwise(col("AGE_ENTRY")))
dfp = dfp.withColumn("UGDS_MEN", when((col("UGDS_MEN").isNull()) | (col("UGDS_MEN") == 0.0),
```

```
mean_UGDS_MEN).otherwise(col("UGDS_MEN")))
dfp = dfp.withColumn("UGDS_WOMEN", when((col("UGDS_WOMEN").isNull()) | (col("UGDS_WOMEN") == 0.0),
mean_UGDS_WOMEN).otherwise(col("UGDS_WOMEN")))
```

FINISHED

```
%pyspark
#COSTT4_A
dfp = dfp.withColumn('COSTT4_A', col('COSTT4_A').cast("float"))

#UGDS
dfp = dfp.withColumn('UGDS', col('UGDS').cast("float"))

#loan, tution and expenses
dfp = dfp.withColumn('TUITIONFEE_IN', col('TUITIONFEE_IN').cast("float"))
dfp = dfp.withColumn('TUITIONFEE_OUT', col('TUITIONFEE_OUT').cast("float"))
dfp = dfp.withColumn('PCTPELL', col('PCTPELL').cast("float"))
dfp = dfp.withColumn('LPSTAFFORD_CNT', col('LPSTAFFORD_CNT').cast("float"))
dfp = dfp.withColumn('LPSTAFFORD_AMT', col('LPSTAFFORD_AMT').cast("float"))
dfp = dfp.withColumn('LPPPLUS_CNT', col('LPPPLUS_CNT').cast("float"))
dfp = dfp.withColumn('LPPPLUS_AMT', col('LPPPLUS_AMT').cast("float"))
dfp = dfp.withColumn('BOOKSUPPLY', col('BOOKSUPPLY').cast("float"))
dfp = dfp.withColumn('ROOMBOARD_ON', col('ROOMBOARD_ON').cast("float"))
dfp = dfp.withColumn('OTHEREXPENSE_ON', col('OTHEREXPENSE_ON').cast("float"))
dfp = dfp.withColumn('ROOMBOARD_OFF', col('ROOMBOARD_OFF').cast("float"))
dfp = dfp.withColumn('OTHEREXPENSE_OFF', col('OTHEREXPENSE_OFF').cast("float"))
dfp = dfp.withColumn('OTHEREXPENSE_FAM', col('OTHEREXPENSE_FAM').cast("float"))
dfp = dfp.withColumn('ENDOWBEGIN', col('ENDOWBEGIN').cast("float"))
dfp = dfp.withColumn('ENDOWEND', col('ENDOWEND').cast("float"))

#student demographics, addmission_rate, age, gender

dfp = dfp.withColumn('ADM_RATE', col('ADM_RATE').cast("float"))
dfp = dfp.withColumn('ENRL_ORIG_YR2_RT', col('ENRL_ORIG_YR2_RT').cast("float"))
dfp = dfp.withColumn('AGE_ENTRY', col('AGE_ENTRY').cast("float"))
dfp = dfp.withColumn('UGDS_MEN', col('UGDS_MEN').cast("float"))
dfp = dfp.withColumn('UGDS_WOMEN', col('UGDS_WOMEN').cast("float"))
```

Took 1 sec. Last updated by anonymous at May 02 2024, 10:01:56 PM.

SPARK JOB ERROR

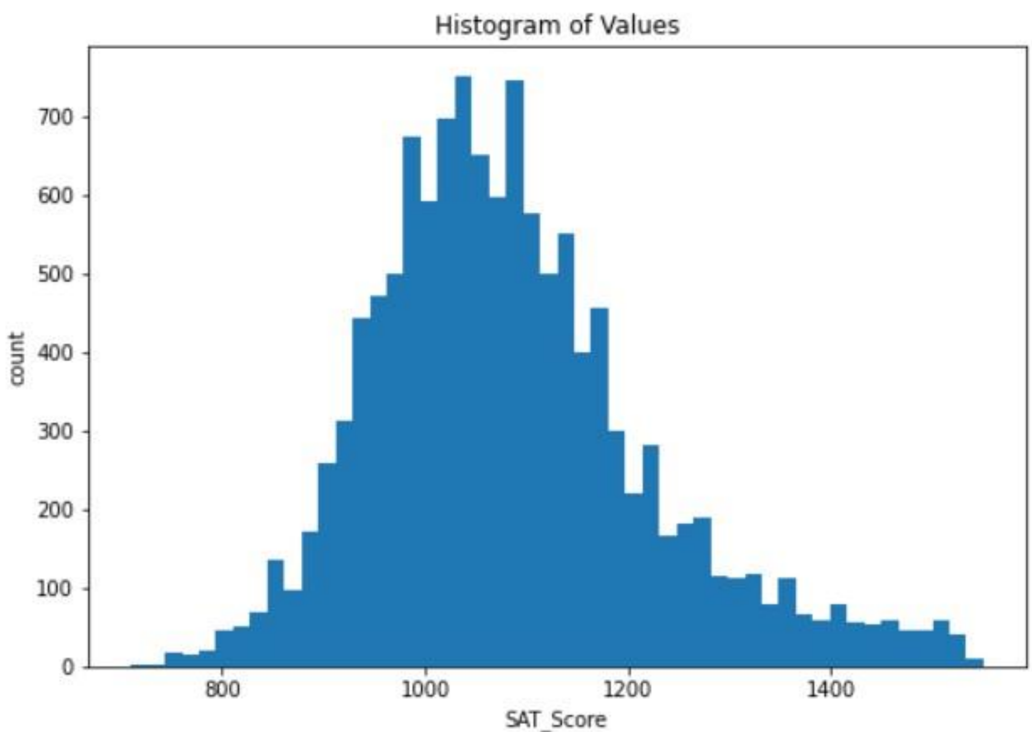
```
%pyspark
#replace null and 0's with the mean value

mean_COSTT4_A = dfp.select(mean(col("COSTT4_A"))).collect()[0][0]
mean_TUITIONFEE_IN = dfp.select(mean(col("TUITIONFEE_IN"))).collect()[0][0]
mean_TUITIONFEE_OUT = dfp.select(mean(col("TUITIONFEE_OUT"))).collect()[0][0]
mean_PCTPELL = dfp.select(mean(col("PCTPELL"))).collect()[0][0]
mean_LPSTAFFORD_CNT = dfp.select(mean(col("LPSTAFFORD_CNT"))).collect()[0][0]
mean_LPSTAFFORD_AMT = dfp.select(mean(col("LPSTAFFORD_AMT"))).collect()[0][0]
mean_LPPPLUS_CNT = dfp.select(mean(col("LPPPLUS_CNT"))).collect()[0][0]
mean_LPPPLUS_AMT = dfp.select(mean(col("LPPPLUS_AMT"))).collect()[0][0]
mean_BOOKSUPPLY = dfp.select(mean(col("BOOKSUPPLY"))).collect()[0][0]
mean_ROOMBOARD_ON = dfp.select(mean(col("ROOMBOARD_ON"))).collect()[0][0]
mean_OTHEREXPENSE_ON = dfp.select(mean(col("OTHEREXPENSE_ON"))).collect()[0][0]
mean_ROOMBOARD_OFF = dfp.select(mean(col("ROOMBOARD_OFF"))).collect()[0][0]
mean_OTHEREXPENSE_OFF = dfp.select(mean(col("OTHEREXPENSE_OFF"))).collect()[0][0]
mean_OTHEREXPENSE_FAM = dfp.select(mean(col("OTHEREXPENSE_FAM"))).collect()[0][0]
mean_ENDOWBEGIN = dfp.select(mean(col("ENDOWBEGIN"))).collect()[0][0]
mean_ENDOWEND = dfp.select(mean(col("ENDOWEND"))).collect()[0][0]
mean_UGDS = dfp.select(mean(col("UGDS"))).collect()[0][0]
mean_ADM_RATE = dfp.select(mean(col("ADM_RATE"))).collect()[0][0]
mean_ENRL_ORIG_YR2_RT = dfp.select(mean(col("ENRL_ORIG_YR2_RT"))).collect()[0][0]
mean_AGE_ENTRY = dfp.select(mean(col("AGE_ENTRY"))).collect()[0][0]
mean_UGDS_MEN = dfp.select(mean(col("UGDS_MEN"))).collect()[0][0]
mean_UGDS_WOMEN= dfp.select(mean(col("UGDS_WOMEN"))).collect()[0][0]


# Replace null values with mean values
dfp = dfp.withColumn("COSTT4_A", when((col("COSTT4_A").isNull()) | (col("COSTT4_A") == 0.0), mean_COSTT4_A).otherwise(col("COSTT4_A")))
dfp = dfp.withColumn("UGDS", when((col("UGDS").isNull()) | (col("UGDS") == 0.0), mean_UGDS).otherwise(col("UGDS")))
dfp = dfp.withColumn("TUITIONFEE_IN", when((col("TUITIONFEE_IN").isNull()) | (col("TUITIONFEE_IN") == 0.0), mean_TUITIONFEE_IN).otherwise(col("TUITIONFEE_IN")))
dfp = dfp.withColumn("TUITIONFEE_OUT", when((col("TUITIONFEE_OUT").isNull()) | (col("TUITIONFEE_OUT") == 0.0), mean_TUITIONFEE_OUT).otherwise(col("TUITIONFEE_OUT")))
dfp = dfp.withColumn("PCTPELL", when((col("PCTPELL").isNull()) | (col("PCTPELL") == 0.0), mean_PCTPELL).otherwise(col("PCTPELL")))
dfp = dfp.withColumn("LPSTAFFORD_CNT", when((col("LPSTAFFORD_CNT").isNull()) | (col("LPSTAFFORD_CNT") == 0.0), mean_LPSTAFFORD_CNT).otherwise(col("LPSTAFFORD_CNT")))
dfp = dfp.withColumn("LPSTAFFORD_AMT", when((col("LPSTAFFORD_AMT").isNull()) | (col("LPSTAFFORD_AMT") == 0.0), mean_LPSTAFFORD_AMT).otherwise(col("LPSTAFFORD_AMT")))
dfp = dfp.withColumn("LPPPLUS_CNT", when((col("LPPPLUS_CNT").isNull()) | (col("LPPPLUS_CNT") == 0.0), mean_LPPPLUS_CNT).otherwise(col("LPPPLUS_CNT")))
dfp = dfp.withColumn("LPPPLUS_AMT", when((col("LPPPLUS_AMT").isNull()) | (col("LPPPLUS_AMT") == 0.0), mean_LPPPLUS_AMT).otherwise(col("LPPPLUS_AMT")))
dfp = dfp.withColumn("BOOKSUPPLY", when((col("BOOKSUPPLY").isNull()) | (col("BOOKSUPPLY") == 0.0), mean_BOOKSUPPLY).otherwise(col("BOOKSUPPLY")))
dfp = dfp.withColumn("ROOMBOARD_ON", when((col("ROOMBOARD_ON").isNull()) | (col("ROOMBOARD_ON") == 0.0), mean_ROOMBOARD_ON).otherwise(col("ROOMBOARD_ON")))
dfp = dfp.withColumn("OTHEREXPENSE_ON", when((col("OTHEREXPENSE_ON").isNull()) | (col("OTHEREXPENSE_ON") == 0.0), mean_OTHEREXPENSE_ON).otherwise(col("OTHEREXPENSE_ON")))
dfp = dfp.withColumn("ROOMBOARD_OFF", when((col("ROOMBOARD_OFF").isNull()) | (col("ROOMBOARD_OFF") == 0.0), mean_ROOMBOARD_OFF).otherwise(col("ROOMBOARD_OFF")))
dfp = dfp.withColumn("OTHEREXPENSE_OFF", when((col("OTHEREXPENSE_OFF").isNull()) | (col("OTHEREXPENSE_OFF") == 0.0), mean_OTHEREXPENSE_OFF).otherwise(col("OTHEREXPENSE_OFF")))
dfp = dfp.withColumn("OTHEREXPENSE_FAM", when((col("OTHEREXPENSE_FAM").isNull()) | (col("OTHEREXPENSE_FAM") == 0.0), mean_OTHEREXPENSE_FAM).otherwise(col("OTHEREXPENSE_FAM")))
dfp = dfp.withColumn("ENDOWBEGIN", when((col("ENDOWBEGIN").isNull()) | (col("ENDOWBEGIN") == 0.0), mean_ENDOWBEGIN).otherwise(col("ENDOWBEGIN")))
dfp = dfp.withColumn("ENDOWEND", when((col("ENDOWEND").isNull()) | (col("ENDOWEND") == 0.0), mean_ENDOWEND).otherwise(col("ENDOWEND")))
dfp = dfp.withColumn("ADM_RATE", when((col("ADM_RATE").isNull()) | (col("ADM_RATE") == 0.0), mean_ADM_RATE).otherwise(col("ADM_RATE")))
dfp = dfp.withColumn("ENRL_ORIG_YR2_RT", when((col("ENRL_ORIG_YR2_RT").isNull()) | (col("ENRL_ORIG_YR2_RT") == 0.0), mean_ENRL_ORIG_YR2_RT).otherwise(col("ENRL_ORIG_YR2_RT")))
```

6. Check the outliers for ‘SAT_Score’, ‘Average_SAT’, ‘ACT_Score’, ‘COSTT4_A’ columns by ploting the histogram.

```
%pyspark
# Finding outliers for SAT_Score
import matplotlib.pyplot as plt

data = dfp.select('SAT_Score').toPandas()

# Plotting a histogram using Matplotlib
plt.hist(data['SAT_Score'], bins=50)
plt.title('Histogram of Values')
plt.xlabel('SAT_Score')
plt.ylabel('count')
```

```
plt.show()
```

```
%pyspark
# Finding outliers for SAT_Score
import matplotlib.pyplot as plt

data = dfp.select('SAT_Score').toPandas()

# Plotting a histogram using Matplotlib
plt.hist(data['SAT_Score'], bins=50)
plt.title('Histogram of Values')
plt.xlabel('SAT_Score')
plt.ylabel('count')
plt.show()
```



7. Remove the outliers for ‘SAT_Score’, ‘Average_SAT’, ‘ACT_Score’, ‘COSTT4_A’ columns.

```
%pyspark
# Remove outliers and replace with NaN

dfp = dfp.withColumn('SAT_Score', when((col('SAT_Score') < 700) | (col('SAT_Score') > 1550),
0.0).otherwise(col('SAT_Score')))
dfp = dfp.withColumn('Average_SAT', when((col('Average_SAT') < 1500) | (col('Average_SAT') > 3100),
0.0).otherwise(col('Average_SAT')))
dfp = dfp.withColumn('ACT_Score', when((col('ACT_Score') < 40) | (col('ACT_Score') > 110),
0.0).otherwise(col('ACT_Score')))
```

```
%pyspark
# Remove outliers and replace with NaN

dfp = dfp.withColumn('SAT_Score', when((col('SAT_Score') < 700) | (col('SAT_Score') > 1550), 0.0).otherwise(col('SAT_Score')))
dfp = dfp.withColumn('Average_SAT', when((col('Average_SAT') < 1500) | (col('Average_SAT') > 3100), 0.0).otherwise(col('Average_SAT')))
dfp = dfp.withColumn('ACT_Score', when((col('ACT_Score') < 40) | (col('ACT_Score') > 110), 0.0).otherwise(col('ACT_Score')))
```

Took 1 sec. Last updated by anonymous at May 13 2024, 10:57:16 PM. (outdated)

8. 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 the column State Code to its respective Index Values.

SPARK JOB FINISHED

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

indexer = StringIndexer(inputCol="STABBR", outputCol="STABBRIndex")
dfp = indexer.fit(dfp).transform(dfp)
```

Took 6 sec. Last updated by anonymous at May 02 2024, 10:03:01 PM.

9. 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 = "US_Scorecard"
dfp.createOrReplaceTempView(temp_table_name)
```

```
%pyspark
if PYSPARK_CLI:
    csv = spark.read.csv(‘"/user/aporwal/Project/export.csv’,
inferSchema=True, header=True)
else:
    csv = spark.sql("SELECT * FROM US_Scorecard")

csv.show(5)
```

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

```
%pyspark

dfp_1= dfp.select('STABBRIndex' , 'CONTROL', 'MD_EARN_WNE_P10',
'GRAD_DEBT_MDN_SUPP', 'SAT_Score', 'Average_SAT', 'ACT_Score',
'COSTT4_A','UGDS','TUITIONFEE_IN','TUITIONFEE_OUT','PCTPELL','LPSTAFFORD_CNT','LPS
TAFFORD_AMT','LPPPLUS_CNT','LPPPLUS_AMT','BOOKSUPPLY','ROOMBOARD_ON','OTHEREXPENSE
_ON','ROOMBOARD_OFF','OTHEREXPENSE_OFF','OTHEREXPENSE_FAM','ENDOWBEGIN','ENDOWEND'
,'ADM_RATE','ENRL_ORIG_YR2_RT','UGDS_MEN','UGDS_WOMEN','AGE_ENTRY',
'RPY_3YR_RT_SUPP', col("Net_Price").alias("label"))

dfp_1.show(2)
```

```
%pyspark
'''
removed fields that were null : NANTI, HBCU,PBI,ANNHI,TRIBAL,AANAPII,HSI,GT_25K_P6
'''

dfp_1= dfp.select('STABBRIndex' , 'CONTROL', 'MD_EARN_WNE_P10', 'GRAD_DEBT_MDN_SUPP', 'SAT_Score', 'Average_SAT', 'ACT_Score', 'COSTT4_A','UGDS','TUITIONFEE_IN','TUITIONFEE_OUT','PCTPELL'
    ,'LPSTAFFORD_CNT','LPSTAFFORD_AMT','LPPPLUS_CNT','LPPPLUS_AMT','BOOKSUPPLY','ROOMBOARD_ON','OTHEREXPENSE_ON','ROOMBOARD_OFF','OTHEREXPENSE_OFF','OTHEREXPENSE_FAM','ENDOWBEGIN','ENDOWEND'
    ,'ADM_RATE','ENRL_ORIG_YR2_RT','UGDS_MEN','UGDS_WOMEN','AGE_ENTRY', 'RPY_3YR_RT_SUPP', col("Net_Price").alias("label"))

dfp_1.show(2)
```

```
|STABBRIndex|CONTROL|   MD_EARN_WNE_P10|GRAD_DEBT_MDN_SUPP|SAT_Score|Average_SAT|ACT_Score|COSTT4_A|   UGDS|TUITIONFEE_IN|TUITIONFEE_OUT|PCTPELL|LPSTAFFORD_CNT|LPSTAFFORD_AMT|LPPPLUS_CNT|LPPPLUS_AMT|BOOK
SUPPLY|ROOMBOARD_ON|OTHEREXPENSE_ON|ROOMBOARD_OFF|OTHEREXPENSE_OFF|OTHEREXPENSE_FAM|  ENDOWBEGIN|    ENDOWEND|ADM_RATE|ENRL_ORIG_YR2_RT|UGDS_MEN|UGDS_WOMEN|AGE_ENTRY|    RPY_3YR_RT_SUPP|  label|
|       18.0|      1|34361.135553258835|           24449.5|    845.0|     1698.0|     53.0| 13762.0| 4930.0|       5800.0|       10672.0| 0.6317|          null|          null|       null|       null|
1800.0|      5530.0|         2080.0|       5530.0|          2080.0|          1300.0|        null|        null|  0.5129|      0.47171316|  0.4771|    0.5229|20.986105| 0.357311874628067| 8190.0|
|       18.0|      1|34361.135553258835|           15250.0|   1185.0|     2216.0|     72.0| 18003.0|10661.0|       5806.0|       13198.0| 0.2882|          null|          null|       null|       null|
1000.0|      9470.0|         4050.0|       9470.0|          4050.0|          4050.0|3.2625632E8|3.3379888E8|  0.8243|       0.4610266|   0.417|     0.583|23.632872|0.5985639691352844|12838.0|
only showing top 2 rows
```

Step 11: 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
splits = dfp_1.randomSplit([0.7,0.3])
train = splits[0]
test = splits[1].withColumnRenamed("label", "trueLabel")

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


```
%pyspark
splits = dfp_1.randomSplit([0.7,0.3])
train = splits[0]
test = splits[1].withColumnRenamed("label", "trueLabel")

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


Step 12: 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 and MinMax scaler to scale the features to the range 0 to 1. We define a pipeline that creates a Feature Vector, MinMax and trains a regression model.

```
%pyspark

assembler = VectorAssembler(inputCols = ['STABBRIndex' ,
'CONTROL','MD_EARN_WNE_P10', 'GRAD_DEBT_MDN_SUPP','RPY_3YR_RT_SUPP',
'SAT_Score', 'Average_SAT',
'ACT_Score','COSTT4_A','UGDS','TUITIONFEE_IN','TUITIONFEE_OUT','PCTPELL','LPSTA
FFORD_CNT','LPSTAFFORD_AMT','LPPPLUS_CNT','LPPPLUS_AMT','BOOKSUPPLY','ROOMBOARD
_ON','OTHEREXPENSE_ON','ROOMBOARD_OFF','OTHEREXPENSE_OFF','OTHEREXPENSE_FAM','E
NDOWBEGIN','ENDOWEND','ADM_RATE','ENRL_ORIG_YR2_RT','UGDS_MEN','UGDS_WOMEN',
'AGE_ENTRY'], outputCol="features")

minMax = MinMaxScaler(inputCol = assembler.getOutputCol(),
outputCol="normFeatures")

rf = RandomForestRegressor(labelCol="label", featuresCol="normFeatures")
```

Feature Importance:

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 the model.

We have performed the feature importance using Random Forest Regression Model.

```
%pyspark
rf = RandomForestRegressor(labelCol="label", featuresCol="normFeatures" ) #numTrees=10

pipeline0_rf = Pipeline(stages=[assembler, minMax, rf])

model = pipeline0_rf.fit(train)

rfModel = model.stages[-1]
#print(rfModel.toDebugString)

import pandas as pd

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

Feature Importance

```
%pyspark
rf = RandomForestRegressor(labelCol="label", featuresCol="normFeatures" ) #numTrees=10

pipeline0_rf = Pipeline(stages=[assembler, minMax, rf])

model = pipeline0_rf.fit(train)

rfModel = model.stages[-1]
#print(rfModel.toDebugString)

import pandas as pd

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

	normFeatures	importance
8	COSTT4_A	0.457702
1	CONTROL	0.125497
10	TUITIONFEE_IN	0.111924
11	TUITIONFEE_OUT	0.105824
18	ROOMBOARD_ON	0.054712
20	ROOMBOARD_OFF	0.027890
3	GRAD_DEBT_MDN_SUPP	0.025712
12	PCTPELL	0.022876
5	SAT_Score	0.012162
9	UGDS	0.010732

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 Train Validation Split 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. Use time pyspark library to evaluate the execution of fit function. We created an array to store the stages of our machine learning pipeline.

```
%pyspark

model = []
pipeline = []

# Train validator parameters

paramGrid = ParamGridBuilder() \
.addGrid(rf.maxDepth, [8, 10]) \
.addGrid(rf.numTrees, [12, 15]) \
.addGrid(rf.minInfoGain, [0.0]) \
.addGrid(rf.maxBins, [58, 60]) \
.build()

# Start recording time
start_time = time.time()
```

```
%pyspark
pipeline.insert(0, Pipeline(stages=[assembler, minMax,rf]))

tv = TrainValidationSplit(estimator=pipeline[0],
evaluator=RegressionEvaluator(), estimatorParamMaps=paramGrid,
trainRatio=0.8)

# the first model
model.insert(0, tv.fit(train))

%pyspark
# End recording time
end_time = time.time()


# Calculate the elapsed time
execution_time = end_time - start_time
print("Random Forest Model execution time with TVS: {:.2f}
seconds".format(execution_time))
```

```
%pyspark

pipeline.insert(0, Pipeline(stages=[assembler, minMax,rf]))

tv = TrainValidationSplit(estimator=pipeline[0], evaluator=RegressionEvaluator(), estimatorParamMaps=paramGrid, trainRatio=0.8)

# the first model
model.insert(0, tv.fit(train))
```

Took 4 min 26 sec. Last updated by anonymous at May 06 2024, 10:55:03 AM.

```
%pyspark
# End recording time
end_time = time.time()


# Calculate the elapsed time
execution_time = end_time - start_time
print("Random Forest Model execution time with TVS: {:.2f} seconds".format(execution_time))
```

```
Random Forest Model execution time with TVS: 266.16 seconds
```

Took 0 sec. Last updated by anonymous at May 06 2024, 10:55:03 AM.

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.

```
%pyspark
# Cross Validator parameters
paramGridCV = ParamGridBuilder() \
    .addGrid(rf.maxDepth, [8, 10]) \
    .addGrid(rf.numTrees, [12, 15]) \
       .addGrid(rf.minInfoGain, [0.0]) \
       .addGrid(rf.maxBins, [58, 60]) \
       .build()
# Start recording time
start_time = time.time()
```

We define a pipeline that creates a Feature Vector and trains a regression model. The pipeline array appends another pipeline at index 1 which will be trained using cross validator.

```
%pyspark
pipeline.insert(1,Pipeline(stages=[assembler,  minMax,rf]))

# K=3, 5
K = 3
cv = CrossValidator(estimator=pipeline[1],
evaluator=RegressionEvaluator(), estimatorParamMaps=paramGridCV,
numFolds=K)

# the second model model.insert(1,cv.fit(train))

# End recording time
end_time = time.time()


# Calculate the elapsed time
execution_time = end_time - start_time
print("Random Forest Model execution time with CV: {:.2f}
seconds".format(execution_time))
```

```
%pyspark
# Start recording time
start_time = time.time()
```

Took 0 sec. Last updated by anonymous at May 06 2024, 10:55:03 AM.

```
%pyspark
pipeline.insert(1, Pipeline(stages=[assembler, minMax, rf]))

# K=3, 5
K = 3
cv = CrossValidator(estimator=pipeline[1], evaluator=RegressionEvaluator(), estimatorParamMaps=paramGridCV, numFolds=K)

# the second model
model.insert(1, cv.fit(train))
```

Took 9 min 16 sec. Last updated by anonymous at May 06 2024, 11:04:19 AM.

```
%pyspark
# End recording time
end_time = time.time()


# Calculate the elapsed time
execution_time = end_time - start_time
print("Random Forest Model execution time with CV: {:.2f} seconds".format(execution_time))
```

```
Random Forest Model execution time with CV: 555.42 seconds
```

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 Net 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 Net Price to the actual Net 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
# Test the model
# list prediction
prediction = []
predicted = []
i = 0
for i in range(2):
  prediction.insert(i, model[i].transform(test))
```

```
%pyspark
# Examine the Predicted and Actual Values
i=0
for i in range(2):
  predicted.insert(i, prediction[i].select("normFeatures",
"prediction", "trueLabel"))
  predicted[i].show(20)
```


```
%pyspark
# Test the model
# list prediction
prediction = []
predicted = []
i = 0
for i in range(2):
  prediction.insert(i, model[i].transform(test))
```

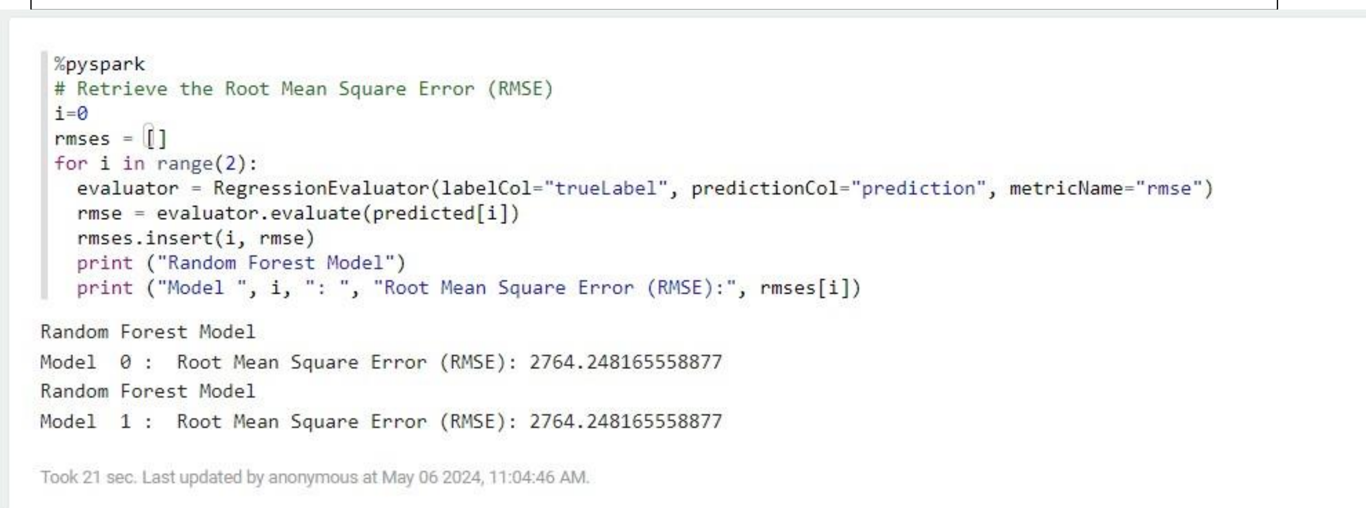

Took 0 sec. Last updated by anonymous at May 06 2024, 11:04:19 AM.


```
%pyspark
# Examine the Predicted and Actual Values
i=0
for i in range(2):
  predicted.insert(i, prediction[i].select("normFeatures", "prediction", "trueLabel"))
  predicted[i].show(20)
```


CALCULATE TRAIN VALIDATION SPLIT & CROSS VALIDATION RMSE AND R2

We will now calculate RMSE and R2 for Random Forest Regression using Train Split Validator and Cross Validator. There are several 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 difference between the predicted and the actual Net Price Values. You can use the RegressionEvaluator class to retrieve the RMSE. Model 0 indicates the Train Validation Split Model and Model 1 indicates the Cross Validation Model.

```
%pyspark
# Retrieve the Root Mean Square Error (RMSE)
i=0
rmses = []
for i in range(2):
  evaluator =
RegressionEvaluator(labelCol="trueLabel",
predictionCol="prediction", metricName="rmse")
  rmse = evaluator.evaluate(predicted[i])
  rmses.insert(i, rmse)
  print ("Random Forest Model")
  print ("Model ", i, ": ", "Root Mean Square Error
(RMSE):", rmses[i])
```



R2 is referred to as coefficient of determination and statistical measure that indicates how well the independent variable(s) explain the variability of the dependent variable. R2 values range from 0 to 1, where 0 indicates that the independent variable(s) do not explain any of the variability of the dependent variable, and 1 indicates that they explain all of it. You can use the RegressionEvaluator class to retrieve the R2. Model 0 indicates the Train Validation Split Model and Model 1 indicates the Cross Validation Model.

```
%pyspark
# Retrieve the R2

i=0
r2s = []
for i in range(2):
  evaluator = RegressionEvaluator(labelCol="trueLabel",
predictionCol="prediction", metricName="r2")
  r2 = evaluator.evaluate(predicted[i])
  r2s.insert(i, r2)
  print ("Model ", i, ": ", "Coefficient of Determination (R2):",
r2s[i])
```

```
%pyspark
# Retrieve the R2

i=0
r2s = []
for i in range(2):
  evaluator = RegressionEvaluator(labelCol="trueLabel", predictionCol="prediction", metricName="r2")
  r2 = evaluator.evaluate(predicted[i])
  r2s.insert(i, r2)
  print ("Model ", i, ": ", "Coefficient of Determination (R2):", r2s[i])

Model  0 :  Coefficient of Determination (R2): 0.8445107852105086
Model  1 :  Coefficient of Determination (R2): 0.8445107852105086
```

Took 22 sec. Last updated by anonymous at May 06 2024, 11:05:08 AM.

CHECKING OVERFITTING FOR MODEL

Model overfitting occurs when a machine learning algorithm learns the training data too well, capturing noise and random fluctuations that are specific to the training set but don't generalize well to new, unseen data. This phenomenon typically results in a model that performs exceptionally well on the training data but poorly on validation or test data.

```
%pyspark
# Finding if the model is overfitted by checking the results with the
train data
prediction = []
predicted = []
i = 0
for i in range(1):
  prediction.insert(i, model[i].transform(train))
i=0
for i in range(1):
  predicted.insert(i, prediction[i].select("normFeatures",
"prediction", "label"))

i=0
rmses = []
for i in range(1):
  evaluator = RegressionEvaluator(labelCol="label",
predictionCol="prediction", metricName="rmse")
  rmse = evaluator.evaluate(predicted[i])
  rmses.insert(i, rmse)
  print ("RF Model(Train data) ", i, ": ", "Root Mean Square Error
(RMSE):", rmses[i])

# Retrieve the R2 for train data

i=0
r2s = []
for i in range(1):
  evaluator = RegressionEvaluator(labelCol="label",
predictionCol="prediction", metricName="r2")
  r2 = evaluator.evaluate(predicted[i])
  r2s.insert(i, r2)
  print ("RF Model(Train data) ", i, ": ", "Coefficient of
Determination (R2):", r2s[i])
```

```
%pyspark
# Finding if the model is overfitted by checking the results with the train data
prediction = []
predicted = []
i = 0
for i in range(1):
  prediction.insert(i, model[i].transform(train))
i=0
for i in range(1):
  predicted.insert(i, prediction[i].select("normFeatures", "prediction", "label"))

i=0
rmses = []
for i in range(1):
  evaluator = RegressionEvaluator(labelCol="label", predictionCol="prediction", metricName="rmse")
  rmse = evaluator.evaluate(predicted[i])
  rmses.insert(i, rmse)
  print ("RF Model(Train data) ", i, ": ", "Root Mean Square Error (RMSE):", rmses[i])

# Retrieve the R2 for train data

i=0
r2s = []
for i in range(1):
  evaluator = RegressionEvaluator(labelCol="label", predictionCol="prediction", metricName="r2")
  r2 = evaluator.evaluate(predicted[i])
  r2s.insert(i, r2)
  print ("RF Model(Train data) ", i, ": ", "Coefficient of Determination (R2):", r2s[i])
```

```
RF Model(Train data)  0 :  Root Mean Square Error (RMSE): 2121.2240897876104
RF Model(Train data)  0 :  Coefficient of Determination (R2): 0.9086053772297566
```

Step 13: 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 and MinMax scaler to scale the features to the range 0 to 1. We define a pipeline that creates a Feature Vector, MinMax and trains a regression model.

```
%pyspark

assembler = VectorAssembler(inputCols = ['STABBRIndex' ,
'CONTROL','MD_EARN_WNE_P10',
'GRAD_DEBT_MDN_SUPP','RPY_3YR_RT_SUPP', 'SAT_Score', 'Average_SAT',
'ACT_Score','COSTT4_A','UGDS','TUITIONFEE_IN','TUITIONFEE_OUT','PCT
PELL','LPSTAFFORD_CNT','LPSTAFFORD_AMT','LPPPLUS_CNT','LPPPLUS_AMT'
,'BOOKSUPPLY','ROOMBOARD_ON','OTHEREXPENSE_ON','ROOMBOARD_OFF','OTH
EREXPENSE_OFF','OTHEREXPENSE_FAM','ENDOWBEGIN','ENDOWEND','ADM_RATE
','ENRL_ORIG_YR2_RT','UGDS_MEN','UGDS_WOMEN', 'AGE_ENTRY'],
outputCol="features")


minMax = MinMaxScaler(inputCol = assembler.getOutputCol(),
outputCol="normFeatures")

gbt = GBTRegressor(labelCol="label", featuresCol="normFeatures")
```

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. Train Validation Split 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 aFeature Vector and trains a regression model. Use time pyspark library to evaluate the execution of fit function. We created an array to store the stages of our machine learning pipeline. Pipeline at index 0 will be tranined using TrainValidationSplit for tuning the hyperparameters of the model.

```
%pyspark

model = []
pipeline = []

# Train Validation Parameters

paramGrid = ParamGridBuilder()\
.addGrid(gbt.maxDepth, [5, 10, 20])\
.addGrid(gbt.maxBins, [52, 55]) \
.addGrid(gbt.maxIter, [10,20,30]) \
.build()

# Start recording time
start_time = time.time()
```

```
%pyspark
pipeline.insert(0, Pipeline(stages=[assembler, minMax, gbt]))

tv = TrainValidationSplit(estimator=pipeline[0],
evaluator=RegressionEvaluator(), estimatorParamMaps=paramGrid,
trainRatio=0.8)

# the first model
model.insert(0, tv.fit(train))

# End recording time
end_time = time.time()


# Calculate the elapsed time
execution_time = end_time - start_time
print("Gradient Boost Trees Model execution time with TVS: {:.2f}
seconds".format(execution_time))
```

```
%pyspark

# Train Validation Parameters

paramGrid = ParamGridBuilder()\
.addGrid(gbt.maxDepth, [5, 10, 20])\
.addGrid(gbt.maxBins, [52, 55]) \
.addGrid(gbt.maxIter, [10,20,30]) \
.build()
```

Took 0 sec. Last updated by anonymous at May 06 2024, 11:05:34 AM.

```
%pyspark
# Start recording time
start_time = time.time()
```

Took 0 sec. Last updated by anonymous at May 06 2024, 11:05:35 AM.

```
%pyspark
pipeline.insert(0, Pipeline(stages=[assembler, minMax, gbt]))

tv = TrainValidationSplit(estimator=pipeline[0], evaluator=RegressionEvaluator(), estimatorParamMaps=paramGrid, trainRatio=0.8)

# the first model
model.insert(0, tv.fit(train))
```

Took 1 hrs 13 min 42 sec. Last updated by anonymous at May 06 2024, 12:19:17 PM.

```
%pyspark
# End recording time
end_time = time.time()


# Calculate the elapsed time
execution_time = end_time - start_time
print("Gradient Boost Trees Model execution time with TVS: {:.2f} seconds".format(execution_time))
```

```
Gradient Boost Trees Model execution time with TVS: 4422.36 seconds
```

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. The pipeline array appends another pipeline at index 1 which will be trained using cross validator for tuning the hyperparameters of the model.

```
%pyspark
# Cross Validation Parameters

paramGridCV = ParamGridBuilder()\
.addGrid(gbt.maxDepth, [5, 10, 20])\
.addGrid(gbt.maxBins, [52, 55]) \
.addGrid(gbt.maxIter, [10,20,30]) \
.build()

# Start recording time
start_time = time.time()
```

```
%pyspark
pipeline.insert(1, Pipeline(stages=[assembler, minMax, gbt]))

# K=3, 5
K = 3
cv = CrossValidator(estimator=pipeline[1],
evaluator=RegressionEvaluator(), estimatorParamMaps=paramGridCV,
numFolds=K)

# the second model
model.insert(1, cv.fit(train))


# End recording time
end_time = time.time()


# Calculate the elapsed time
execution_time = end_time - start_time
print("Gradient Boost Trees Model execution time with CV: {:.2f}
seconds".format(execution_time))
```

```
%pyspark
# Cross Validation Parameters

paramGridCV = ParamGridBuilder()\
.addGrid(gbt.maxDepth, [5, 10, 20])\
.addGrid(gbt.maxBins, [52, 55]) \
.addGrid(gbt.maxIter, [10,20,30]) \
.build()
```

Took 0 sec. Last updated by anonymous at May 06 2024, 12:19:17 PM.

```
%pyspark
# Start recording time
start_time = time.time()
```

Took 0 sec. Last updated by anonymous at May 06 2024, 12:19:17 PM.

```
%pyspark
pipeline.insert(1, Pipeline(stages=[assembler, minMax, gbt]))

# K=3, 5
K = 3
cv = CrossValidator(estimator=pipeline[1], evaluator=RegressionEvaluator(), estimatorParamMaps=paramGridCV, numFolds=K)

# the second model
model.insert(1, cv.fit(train))
```

Took 2 hrs 7 min 48 sec. Last updated by anonymous at May 06 2024, 2:27:05 PM.

```
%pyspark
# End recording time
end_time = time.time()


# Calculate the elapsed time
execution_time = end_time - start_time
print("Gradient Boost Trees Model execution time with CV: {:.2f} seconds".format(execution_time))
```

```
Gradient Boost Trees Model execution time with CV: 7667.39 seconds
```

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 Net 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 Net Price to the actual Net 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
# Test the model
# list prediction
prediction = []
predicted = []
i = 0
for i in range(2):
  prediction.insert(i, model[i].transform(test))

# Examine the Predicted and Actual Values
i=0
for i in range(2):
  predicted.insert(i, prediction[i].select("normFeatures",
"prediction", "trueLabel"))
  predicted[i].show(20)
```


```
%pyspark
# Test the model
# list prediction
prediction = []
predicted = []
i = 0
for i in range(2):
  prediction.insert(i, model[i].transform(test))
```


Took 0 sec. Last updated by anonymous at May 06 2024, 2:27:05 PM.


```
%pyspark
# Examine the Predicted and Actual Values
i=0
for i in range(2):
  predicted.insert(i, prediction[i].select("normFeatures", "prediction", "trueLabel"))
  predicted[i].show(20)
```


We will now calculate RMSE and R2 for Random Forest Regression using Train Split Validator and Cross Validator. There are several 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 difference between the predicted and the actual Net Price Values. You can use the RegressionEvaluator class to retrieve the RMSE. Model 0 indicates the Train Validation Split Model and Model 1 indicates the Cross Validation Model.

```
%pyspark
# Retrieve the Root Mean Square Error (RMSE)
i=0
rmses = []
for i in range(2):
  evaluator = RegressionEvaluator(labelCol="trueLabel",
predictionCol="prediction", metricName="rmse")
  rmse = evaluator.evaluate(predicted[i])
  rmses.insert(i, rmse)
  print ("Gradient Boost Trees Model")
  print ("Model ", i, ": ", "Root Mean Square Error
(RMSE):", rmses[i])
```

CALCULATE TRAIN VALIDATION SPLIT & CROSS VALIDATION RMSE AND R2

We will now calculate RMSE and R2 for Random Forest Regression using Train Split Validator and Cross Validator. There are several 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 difference between the predicted and the actual Net Price Values. You can use the RegressionEvaluator class to retrieve the RMSE. Model 0 indicates the Train Validation Split Model and Model 1 indicates the Cross Validation Model.

```
%pyspark
# Retrieve the Root Mean Square Error (RMSE)
i=0
rmses = []
for i in range(2):
  evaluator =
RegressionEvaluator(labelCol="trueLabel",
predictionCol="prediction", metricName="rmse")
  rmse = evaluator.evaluate(predicted[i])
  rmses.insert(i, rmse)
  print ("Gradient Boost Trees Model")
  print ("Model ", i, ": ", "Root Mean Square Error
(RMSE):", rmses[i])
```

RMSE for Gradient Boost Trees Model

Took 0 sec. Last updated by anonymous at May 06 2024, 2:27:10 PM.

```
%pyspark
# Retrieve the Root Mean Square Error (RMSE)
i=0
rmses = []
for i in range(2):
  evaluator = RegressionEvaluator(labelCol="trueLabel", predictionCol="prediction", metricName="rmse")
  rmse = evaluator.evaluate(predicted[i])
  rmses.insert(i, rmse)
  print ("Gradient Boost Trees Model")
  print ("Model ", i, ": ", "Root Mean Square Error (RMSE):", rmses[i])
```

```
Gradient Boost Trees Model
Model  0 :  Root Mean Square Error (RMSE): 2841.3088635872373
Gradient Boost Trees Model
Model  1 :  Root Mean Square Error (RMSE): 2840.995064044869
```

Took 10 sec. Last updated by anonymous at May 06 2024, 2:27:20 PM.

R2 is referred to as coefficient of determination and statistical measure that indicates how well the independent variable(s) explain the variability of the dependent variable. R2 values range from 0 to 1, where 0 indicates that the independent variable(s) do not explain any of the variability of the dependent variable, and 1 indicates that they explain all of it. You can use the RegressionEvaluator class to retrieve the R2. Model 0 indicates the Train Validation Split Model and Model 1 indicates the Cross Validation Model.

```
%pyspark
# Retrieve the R2
i=0
r2s = []
for i in range(2):
  evaluator = RegressionEvaluator(labelCol="trueLabel",
predictionCol="prediction", metricName="r2")
  r2 = evaluator.evaluate(predicted[i])
  r2s.insert(i, r2)
  print ("Model ", i, ": ", "Coefficient of Determination (R2):",
r2s[i])
```

R2 for Gradient Boost Trees Model

Took 0 sec. Last updated by anonymous at May 06 2024, 2:27:20 PM.

```
%pyspark
# Retrieve the R2
i=0
r2s = []
for i in range(2):
  evaluator = RegressionEvaluator(labelCol="trueLabel", predictionCol="prediction", metricName="r2")
  r2 = evaluator.evaluate(predicted[i])
  r2s.insert(i, r2)
  print ("Model ", i, ": ", "Coefficient of Determination (R2):", r2s[i])
```

```
Model  0 :  Coefficient of Determination (R2): 0.8357206020566925
Model  1 :  Coefficient of Determination (R2): 0.8357568867097773
```

Step 14: 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 and MinMax scaler to scale the features to the range 0 to 1. We define a pipeline that creates a Feature Vector, MinMax and trains a regression model.

```
%pyspark

assembler = VectorAssembler(inputCols = ['STABBRIndex' ,
'CONTROL','MD_EARN_WNE_P10',
'GRAD_DEBT_MDN_SUPP','RPY_3YR_RT_SUPP', 'SAT_Score', 'Average_SAT',
'ACT_Score','COSTT4_A','UGDS','TUITIONFEE_IN','TUITIONFEE_OUT','PCT
PELL','LPSTAFFORD_CNT','LPSTAFFORD_AMT','LPPPLUS_CNT','LPPPLUS_AMT'
,'BOOKSUPPLY','ROOMBOARD_ON','OTHEREXPENSE_ON','ROOMBOARD_OFF','OTH
EREXPENSE_OFF','OTHEREXPENSE_FAM','ENDOWBEGIN','ENDOWEND','ADM_RATE
','ENRL_ORIG_YR2_RT','UGDS_MEN','UGDS_WOMEN', 'AGE_ENTRY'],
outputCol="features")


minMax = MinMaxScaler(inputCol = assembler.getOutputCol(),
outputCol="normFeatures")

dt = DecisionTreeRegressor(labelCol="label",
featuresCol="normFeatures")
```

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. Train Validation Split 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. Use time pyspark library to evaluate the execution of fit function. We created an array to store the stages of our machine learning pipeline. Pipeline at index 0 will be trained using Train Validation Split for tuning the hyperparameters of the model.

```
%pyspark

model = []
pipeline = []

# Train validator parameters

paramGrid = ParamGridBuilder() \
 .addGrid(dt.maxDepth, [15, 20]) \
 .addGrid(dt.minInfoGain, [0.0]) \
 .addGrid(dt.maxBins, [58,60]) \
 .build()

# Start recording time
start_time = time.time()
```

```
%pyspark

pipeline.insert(0, Pipeline(stages=[assembler,minMax, dt]))

tv = TrainValidationSplit(estimator=pipeline[0],
evaluator=RegressionEvaluator(), estimatorParamMaps=paramGrid,
trainRatio=0.8)

# the first model
model.insert(0, tv.fit(train))

# End recording time
end_time = time.time()


# Calculate the elapsed time
execution_time = end_time - start_time
print("Decision Tree Regression Model execution time with TVS: {:.2f}
seconds".format(execution_time))
```

```
%pyspark

# Train validator parameters

paramGrid = ParamGridBuilder() \
 .addGrid(dt.maxDepth, [15, 20]) \
 .addGrid(dt.minInfoGain, [0.0]) \
 .addGrid(dt.maxBins, [58,60]) \
 .build()
```

Took 0 sec. Last updated by anonymous at May 06 2024, 2:27:31 PM.

```
%pyspark
# Start recording time
start_time = time.time()
```

Took 0 sec. Last updated by anonymous at May 06 2024, 2:27:31 PM.

```
%pyspark

pipeline.insert(0, Pipeline(stages=[assembler,minMax, dt]))

tv = TrainValidationSplit(estimator=pipeline[0], evaluator=RegressionEvaluator(), estimatorParamMaps=paramGrid, trainRatio=0.8)

# the first model
model.insert(0, tv.fit(train))
```

Took 1 min 31 sec. Last updated by anonymous at May 06 2024, 2:29:02 PM.

```
%pyspark
# End recording time
end_time = time.time()


# Calculate the elapsed time
execution_time = end_time - start_time
print("Decision Tree Regression Model execution time with TVS: {:.2f} seconds".format(execution_time))
```

```
Decision Tree Regression Model execution time with TVS: 90.87 seconds
```

Took 0 sec. Last updated by anonymous at May 06 2024, 2:29:02 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. The pipeline array appends another pipeline at index 1 which will be trained using cross validator for tuning the hyperparameters of the model.

```
%pyspark

# Cross Validator parameters

paramGridCV = ParamGridBuilder() \
 .addGrid(dt.maxDepth, [15, 20]) \
 .addGrid(dt.minInfoGain, [0.0]) \
 .addGrid(dt.maxBins, [58,60]) \
 .build()

# Start recording time
start_time = time.time()
```

```
%pyspark
pipeline.insert(1, Pipeline(stages=[assembler, minMax, dt]))

# K=3, 5
K = 3
cv = CrossValidator(estimator=pipeline[1],
evaluator=RegressionEvaluator(), estimatorParamMaps=paramGridCV,
numFolds=K)

# the second model
model.insert(1, cv.fit(train))

# End recording time
end_time = time.time()


# Calculate the elapsed time
execution_time = end_time - start_time
print("Decision Tree Regression Model execution time with CV: {:.2f}
seconds".format(execution_time))
```

```
# Cross Validator parameters

paramGridCV = ParamGridBuilder() \
 .addGrid(dt.maxDepth, [15, 20]) \
 .addGrid(dt.minInfoGain, [0.0]) \
 .addGrid(dt.maxBins, [58,60]) \
 .build()
```

Took 0 sec. Last updated by anonymous at May 06 2024, 2:29:02 PM.

```
%pyspark
# Start recording time
start_time = time.time()
```

Took 0 sec. Last updated by anonymous at May 06 2024, 2:29:02 PM.

```
%pyspark
pipeline.insert(1, Pipeline(stages=[assembler, minMax, dt]))

# K=3, 5
K = 3
cv = CrossValidator(estimator=pipeline[1], evaluator=RegressionEvaluator(), estimatorParamMaps=paramGridCV, numFolds=K)

# the second model
model.insert(1, cv.fit(train))
```

Took 3 min 37 sec. Last updated by anonymous at May 06 2024, 2:32:39 PM.

```
%pyspark
# End recording time
end_time = time.time()


# Calculate the elapsed time
execution_time = end_time - start_time
print("Decision Tree Regression Model execution time with CV: {:.2f} seconds".format(execution_time))
```

```
Decision Tree Regression Model execution time with CV: 216.81 seconds
```

Took 0 sec. Last updated by anonymous at May 06 2024, 2:32:39 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 Net 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 Net Price to the actual Net 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
# Test the model
# list prediction
prediction = []
predicted = []
i = 0
for i in range(2):
  prediction.insert(i, model[i].transform(test))
```

```
%pyspark
# Examine the Predicted and Actual Values
i=0
for i in range(2):
  predicted.insert(i, prediction[i].select("normFeatures",
"prediction", "trueLabel"))
  predicted[i].show(20)
```

```
%pyspark
# Test the model
# list prediction
prediction = []
predicted = []
i = 0
for i in range(2):
  prediction.insert(i, model[i].transform(test))
```

Took 1 sec. Last updated by anonymous at May 06 2024, 2:32:40 PM.

```
%pyspark
# Examine the Predicted and Actual Values
i=0
for i in range(2):
  predicted.insert(i, prediction[i].select("normFeatures", "prediction", "trueLabel"))
  predicted[i].show(20)
```

CALCULATE TRAIN VALIDATION SPLIT & CROSS VALIDATION RMSE AND R2

We will now calculate RMSE and R2 for Random Forest Regression using Train Split Validator and Cross Validator. There are several 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 difference between the predicted and the actual Net Price Values. You can use the RegressionEvaluator class to retrieve the RMSE. Model 0 indicates the Train Validation Split Model and Model 1 indicates the Cross Validation Model.

```
%pyspark
# Retrieve the Root Mean Square Error (RMSE)
i=0
rmses = []
for i in range(2):
  evaluator = RegressionEvaluator(labelCol="trueLabel",
predictionCol="prediction", metricName="rmse")
  rmse = evaluator.evaluate(predicted[i])
  rmses.insert(i, rmse)
  print ("Decision Tree Regression Model")
  print ("Model ", i, ": ", "Root Mean Square Error (RMSE):",
rmses[i])
```

RMSE for Decision Tree Regression Model

Took 0 sec. Last updated by anonymous at May 06 2024, 2:32:44 PM.

```
%pyspark
# Retrieve the Root Mean Square Error (RMSE)
i=0
rmses = []
for i in range(2):
  evaluator = RegressionEvaluator(labelCol="trueLabel", predictionCol="prediction", metricName="rmse")
  rmse = evaluator.evaluate(predicted[i])
  rmses.insert(i, rmse)
  print ("Decision Tree Regression Model")
  print ("Model ", i, ": ", "Root Mean Square Error (RMSE):", rmses[i])
```

```
Decision Tree Regression Model
Model  0 :  Root Mean Square Error (RMSE): 3245.624906570822
Decision Tree Regression Model
Model  1 :  Root Mean Square Error (RMSE): 3494.556175565804
```

Took 11 sec. Last updated by anonymous at May 06 2024, 2:32:55 PM.

R2 is referred to as coefficient of determination and statistical measure that indicates how well the independent variable(s) explain the variability of the dependent variable. R2 values range from 0 to 1, where 0 indicates that the independent variable(s) do not explain any of the variability of the dependent variable, and 1 indicates that they explain all of it. You can use the RegressionEvaluator class to retrieve the R2. Model 0 indicates the Train Validation Split Model and Model 1 indicates the Cross Validation Model.

```
%pyspark
# Retrieve the R2

i=0
r2s = []
for i in range(2):
  evaluator = RegressionEvaluator(labelCol="trueLabel",
predictionCol="prediction", metricName="r2")
  r2 = evaluator.evaluate(predicted[i])
  r2s.insert(i, r2)
  print ("Model ", i, ": ", "Coefficient of Determination (R2):",
r2s[i])
```

R2 for Decision Tree Regression Model

Took 0 sec. Last updated by anonymous at May 06 2024, 2:32:55 PM.


```
%pyspark
# Retrieve the R2

i=0
r2s = []
for i in range(2):
  evaluator = RegressionEvaluator(labelCol="trueLabel", predictionCol="prediction", metricName="r2")
  r2 = evaluator.evaluate(predicted[i])
  r2s.insert(i, r2)
  print ("Model ", i, ": ", "Coefficient of Determination (R2):", r2s[i])
```


```
Model  0 :  Coefficient of Determination (R2): 0.7856404359302236
Model  1 :  Coefficient of Determination (R2): 0.7514977857932369
```

Took 9 sec. Last updated by anonymous at May 06 2024, 2:33:04 PM.

Step 15: 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 and MinMax scaler to scale the features to the range 0 to 1. We define a pipeline that creates a Feature Vector, MinMax and trains a regression model.

```
%pyspark

assembler = VectorAssembler(inputCols = ['STABBRIndex' ,
'CONTROL','MD_EARN_WNE_P10',
'GRAD_DEBT_MDN_SUPP','RPY_3YR_RT_SUPP', 'SAT_Score', 'Average_SAT',
'ACT_Score','COSTT4_A','UGDS','TUITIONFEE_IN','TUITIONFEE_OUT','PCT
PELL','LPSTAFFORD_CNT','LPSTAFFORD_AMT','LPPPLUS_CNT','LPPPLUS_AMT'
,'BOOKSUPPLY','ROOMBOARD_ON','OTHEREXPENSE_ON','ROOMBOARD_OFF','OTH
EREXPENSE_OFF','OTHEREXPENSE_FAM','ENDOWBEGIN','ENDOWEND','ADM_RATE
','ENRL_ORIG_YR2_RT','UGDS_MEN','UGDS_WOMEN', 'AGE_ENTRY'],
outputCol="features")


minMax = MinMaxScaler(inputCol = assembler.getOutputCol(),
outputCol="normFeatures")

lr = LinearRegression(labelCol="label", featuresCol="normFeatures")
```

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. Train Validation Split 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. Use time pyspark library to evaluate the execution of fit function. We created an array to store the stages of our machine learning pipeline. Pipeline at index 0 will be trained using Train Validation Split for tuning the hyperparameters of the model.

```
%pyspark

model = []
pipeline = []

# Train validator parameters

paramGrid = ParamGridBuilder() \
.addGrid(lr.maxIter, [20,30,40]) \
.addGrid(lr.regParam, [0.01, 0.1, 1.0]) \
.addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0]) \
.addGrid(lr.standardization, [True, False]) \
.build()

# Start recording time
start_time = time.time()
```

```
%pyspark

pipeline.insert(0, Pipeline(stages=[assembler, minMax, lr]))

tv = TrainValidationSplit(estimator=pipeline[0],
evaluator=RegressionEvaluator(), estimatorParamMaps=paramGrid,
trainRatio=0.8)

# the first model
model.insert(0, tv.fit(train))


# End recording time
end_time = time.time()

# Calculate the elapsed time
execution_time = end_time - start_time
print("Linear Regression Model execution time with TVS: {:.2f}
seconds".format(execution_time))
```

```
%pyspark

# Train validator parameters

paramGrid = ParamGridBuilder() \
.addGrid(lr.maxIter, [20,30,40]) \
.addGrid(lr.regParam, [0.01, 0.1, 1.0]) \
.addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0]) \
.addGrid(lr.standardization, [True, False]) \
.build()
```

Took 0 sec. Last updated by anonymous at May 06 2024, 2:33:05 PM.

```
%pyspark
# Start recording time
start_time = time.time()
```

Took 0 sec. Last updated by anonymous at May 06 2024, 2:33:05 PM.

```
%pyspark

pipeline.insert(0, Pipeline(stages=[assembler, minMax, lr]))

tv = TrainValidationSplit(estimator=pipeline[0], evaluator=RegressionEvaluator(), estimatorParamMaps=paramGrid, trainRatio=0.8)

# the first model
model.insert(0, tv.fit(train))
```

Took 10 min 1 sec. Last updated by anonymous at May 06 2024, 2:43:06 PM.

```
%pyspark
# End recording time
end_time = time.time()


# Calculate the elapsed time
execution_time = end_time - start_time
print("Linear Regression Model execution time with TVS: {:.2f} seconds".format(execution_time))
```

```
Linear Regression Model execution time with TVS: 601.17 seconds
```

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. The pipeline array appends another pipeline at index 1 which will be trained using cross validator for tuning the hyperparameters of the model.

```
%pyspark

# Cross Validator parameters

paramGridCV = ParamGridBuilder() \
.addGrid(lr.maxIter, [20,30,40]) \
.addGrid(lr.regParam, [0.01, 0.1, 1.0]) \
.addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0]) \
.addGrid(lr.standardization, [True, False]) \
.build()

# Start recording time
start_time = time.time()
```

```
%pyspark
pipeline.insert(1, Pipeline(stages=[assembler, minMax, lr]))

# K=3, 5
K = 3
cv = CrossValidator(estimator=pipeline[1],
evaluator=RegressionEvaluator(), estimatorParamMaps=paramGridCV,
numFolds=K)

# the second model
model.insert(1, cv.fit(train))


# End recording time
end_time = time.time()

# Calculate the elapsed time
execution_time = end_time - start_time
print("Linear Regression Model execution time with CV: {:.2f}
seconds".format(execution_time))
```

```
%pyspark

# Cross Validator parameters

paramGridCV = ParamGridBuilder() \
.addGrid(lr.maxIter, [20,30,40]) \
.addGrid(lr.regParam, [0.01, 0.1, 1.0]) \
.addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0]) \
.addGrid(lr.standardization, [True, False]) \
.build()
```

Took 0 sec. Last updated by anonymous at May 06 2024, 2:43:07 PM.

```
%pyspark
# Start recording time
start_time = time.time()
```

Took 0 sec. Last updated by anonymous at May 06 2024, 2:43:07 PM.

```
%pyspark
pipeline.insert(1, Pipeline(stages=[assembler, minMax, lr]))

# K=3, 5
K = 3
cv = CrossValidator(estimator=pipeline[1], evaluator=RegressionEvaluator(), estimatorParamMaps=paramGridCV, numFolds=K)

# the second model
model.insert(1, cv.fit(train))
```

Took 29 min 15 sec. Last updated by anonymous at May 06 2024, 3:12:22 PM.

```
%pyspark
# End recording time
end_time = time.time()


# Calculate the elapsed time
execution_time = end_time - start_time
print("Linear Regression Model execution time with CV: {:.2f} seconds".format(execution_time))
```

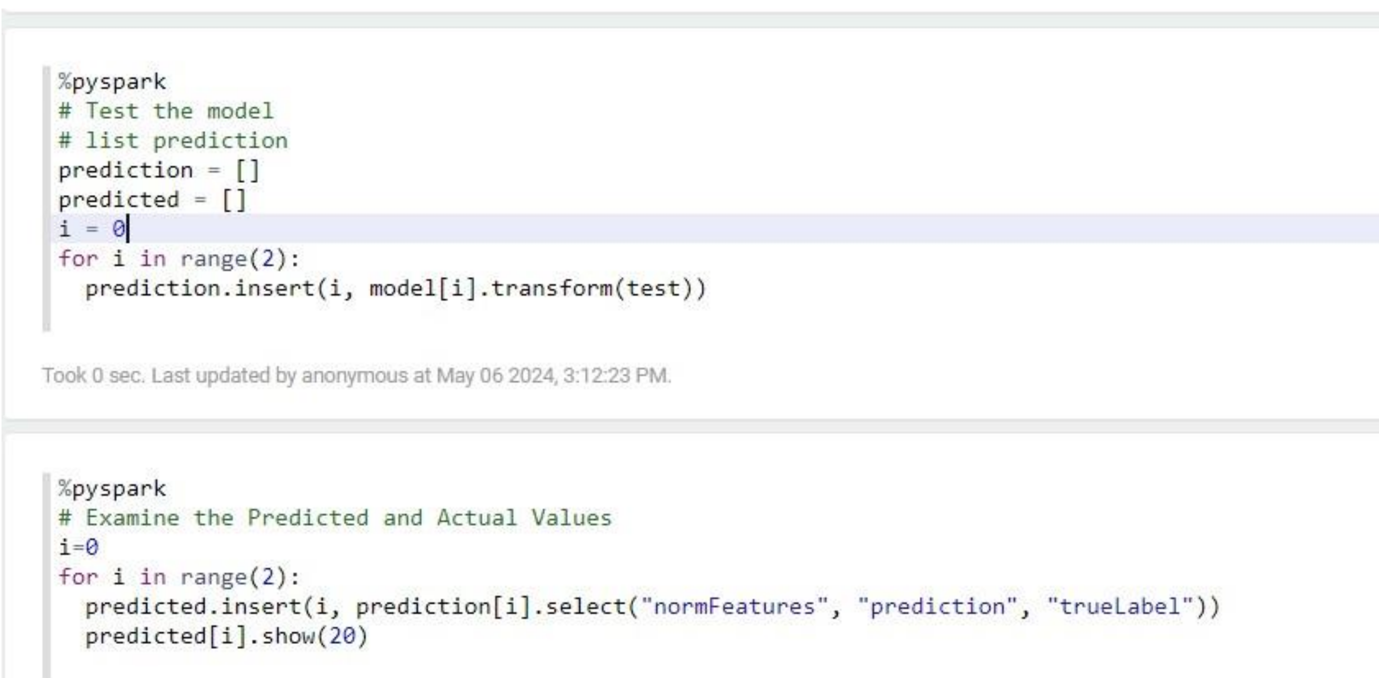
```
Linear Regression Model execution time with CV: 1755.56 seconds
```

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 Net 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 Net Price to the actual Net 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
# Test the model
# list prediction
prediction = []
predicted = []
i = 0
for i in range(2):
  prediction.insert(i, model[i].transform(test))
```

```
%pyspark
# Examine the Predicted and Actual Values
i=0
for i in range(2):
  predicted.insert(i, prediction[i].select("normFeatures",
"prediction", "trueLabel"))
  predicted[i].show(20)
```



CALCULATE TRAIN VALIDATION SPLIT & CROSS VALIDATION RMSE AND R2

We will now calculate RMSE and R2 for Random Forest Regression using Train Split Validator and Cross

Validator. There are several 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 difference between the predicted and the actual Net Price Values. You can use the RegressionEvaluator class to retrieve the RMSE. Model 0 indicates the Train Validation Split Model and Model 1 indicates the Cross Validation Model.

```
%pyspark
# Retrieve the Root Mean Square Error (RMSE)
i=0
rmses = []
for i in range(2):
  evaluator = RegressionEvaluator(labelCol="trueLabel",
predictionCol="prediction", metricName="rmse")
  rmse = evaluator.evaluate(predicted[i])
  rmses.insert(i, rmse)
  print ("Linear Regression Model")
  print ("Model ", i, ": ", "Root Mean Square Error (RMSE):",
rmses[i])
```

RMSE for Linear Regression Model

Took 0 sec. Last updated by anonymous at May 06 2024, 3:12:27 PM.

```
%pyspark
# Retrieve the Root Mean Square Error (RMSE)
i=0
rmses = []
for i in range(2):
  evaluator = RegressionEvaluator(labelCol="trueLabel", predictionCol="prediction", metricName="rmse")
  rmse = evaluator.evaluate(predicted[i])
  rmses.insert(i, rmse)
  print ("Linear Regression Model")
  print ("Model ", i, ": ", "Root Mean Square Error (RMSE):", rmses[i])
```

```
Linear Regression Model
Model  0 :  Root Mean Square Error (RMSE): 3336.8346418246597
Linear Regression Model
Model  1 :  Root Mean Square Error (RMSE): 3337.136963499973
```

Took 9 sec. Last updated by anonymous at May 06 2024, 3:12:36 PM.

R2 is referred to as coefficient of determination and statistical measure that indicates how well the independent variable(s) explain the variability of the dependent variable. R2 values range from 0 to 1, where 0 indicates that the independent variable(s) do not explain any of the variability of the dependent variable, and 1 indicates that they explain all of it. You can use the RegressionEvaluator class to retrieve the R2. Model 0 indicates the Train Validation Split Model and Model 1 indicates the Cross Validation Model.

```
%pyspark
# Retrieve the R2

i=0
r2s = []
for i in range(2):
  evaluator = RegressionEvaluator(labelCol="trueLabel",
predictionCol="prediction", metricName="r2")
  r2 = evaluator.evaluate(predicted[i])
  r2s.insert(i, r2)
  print ("Model ", i, ": ", "Coefficient of Determination (R2):",
r2s[i])
```

R2 for Linear Regression Model

Took 0 sec. Last updated by anonymous at May 06 2024, 3:12:36 PM.

```
%pyspark
# Retrieve the R2

i=0
r2s = []
for i in range(2):
  evaluator = RegressionEvaluator(labelCol="trueLabel", predictionCol="prediction", metricName="r2")
  r2 = evaluator.evaluate(predicted[i])
  r2s.insert(i, r2)
  print ("Model ", i, ": ", "Coefficient of Determination (R2):", r2s[i])
```

```
Model  0 :  Coefficient of Determination (R2): 0.7734231256321283
Model  1 :  Coefficient of Determination (R2): 0.7733820674373695
```

Took 8 sec. Last updated by anonymous at May 06 2024, 3:12:45 PM.

Step 16: 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.

On the basis of time:

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

On the basis of accuracy:

RF > GBT > LR > DT (Random Forest has the best accuracy and DT has the least accuracy)

Algorithm	**Results**	**Time taken to fit the model**
Random Forest-TVS	R2: 0.8471 RMSE: 2724.013	128.63 sec
Random Forest-CV	R2: 0.8447 RMSE: 2744.991	293.09 sec
GBT Regressor-TVS	R2: 0.8480 RMSE: 2773.352	2167.57 sec
GBT Regressor-CV	R2: 0.8475 RMSE: 2778.004	5182.68 sec
Decision Tree-TVS	R2: 0.7657 RMSE: 3414.359	133.31 sec
Decision Tree-CV	R2: 0.7657 RMSE: 3414.359	318.01 sec
Linear Regression-TVS	R2: 0.7700 RMSE: 3314.532	513.58 sec
Linear Regression-CV	R2: 0.7700 RMSE: 3314.532	1458.97 sec

Thus, we can conclude that Random Forest with Train Validation Split is the best fit models with least Root Mean Square Error and highest R2 as the difference between the GBT-TVS R2 is very less, and the RF takes much less time than GBT.

References:

a. URL of Data Source: https://collegescorecard.ed.gov/data/
b. URL of your GitHub: https://github.com/zalak2306/US_College_Net_Price_Prediction_ML_Regression_Models
c. URL of References:

a. The power of regression analysis in price forecasting - FasterCapital. FasterCapital. https://fastercapital.com/content/The-Power-of_Regression-Analysis-in-Price-Forecasting.html.

b. U.S. Department of Education & Posted by U.S. Department of Education. (2023, April 24). Updated college scorecard will help students find high value postsecondary programs - ED.gov blog. ED.gov Blog. https://blog.ed.gov/2023/04/updated-college-scorecard-will-help-students-find-high-value-postsecondary-programs/

c. Matz, S. C., Bukow, C. S., Peters, H., Deacons, C., Dinu, A., & Stachl, C. (2023). Using machine learning to predict student retention from socio-demographic characteristics and app-based engagement metrics. Scientific Reports, 13(1). https://doi.org/10.1038/s41598-023-32484-w

d. lab5RegressionFeatureEngSparkML_(4).pdf