CIS5560 Term Project Tutorial

Authors: Sapan Shah; Snehil Sarkar; Katherine Belknap;
Instructor: Jongwook Woo
Date: 05/14/2024

Lab Tutorial

Sapan Shah(sshah82@calstatela.edu)

Snehil Sarkar(ssarkar4@calstatela.edu)

Katherine Belknap(kbelkna2@calstatela.edu)

05/14/2024

Analysis of Anti-Money Laundering (AML) with Spark ML

Note: As per the feedback, dataset is rebalanced by Under-Sampling and for all algorithms, equal numbers of hyper parameters are used for better comparison.

Objectives:

This dataset is segregated into two categories based on the prevalence of money laundering: high and low ratios. It encompasses information on individuals, companies, and banks engaged in both lawful and illicit financial transactions.

- The aim is to develop a model capable of identifying money laundering patterns.
- Utilizing this dataset with its sample, a fraud detection model can be trained using Databricks, which offers a scalable computing environment.
- PySpark's SQL and DataFrame APIs enable various operations such as aggregation, filtering, and joining.
- Subsequently, Performance Metrics can be assessed through techniques like Cross-Validation or Train-Validation Splits.

The findings of this study hold the potential to significantly impact the financial sector, corporations, and governments. By predicting future occurrences of money laundering activities and visualizing potential trends, it could lead to safer and more secure financial transactions.

Platform Specifications:

- **Hadoop Version:** 3.3.3
- **CPU Speed:** 1995.312 MHz
- **No of CPU cores:** 8
- **No of nodes:** 5
- **Spark Version:** 3.2.1
- **Total Memory Size:** 860.4 GB

Dataset Specifications: -

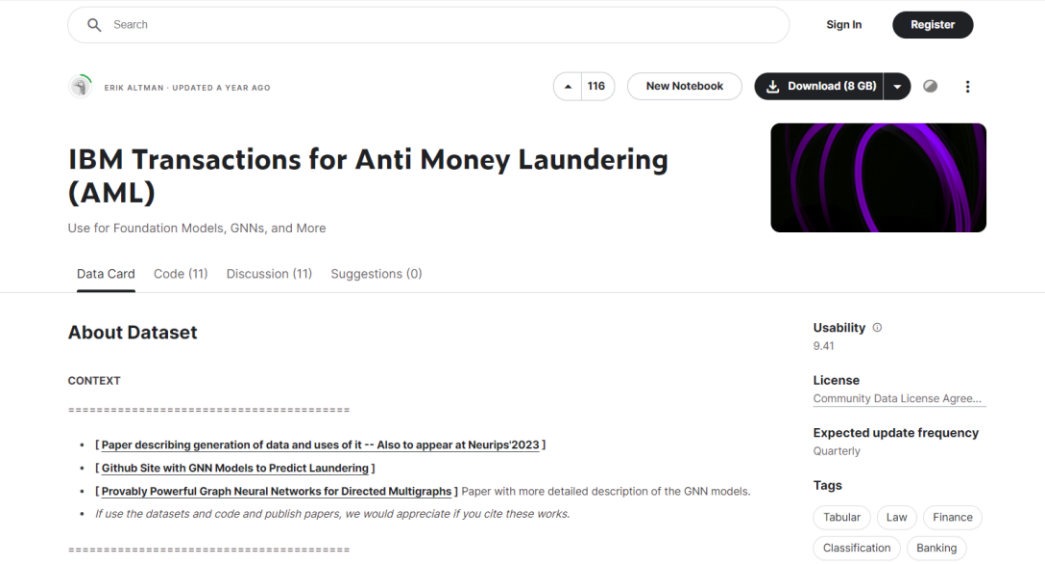
Dataset Name: IBM Transactions for Anti Money Laundering (AML)

Dataset Size: 2.82 GB

Dataset URL: https://www.kaggle.com/datasets/ealtman2019/ibm-transactions-for-anti-money-laundering-aml

Dataset Format: CSV

Dataset: HI_Medium_Trans.csv



Step 1: Get data from the Data source.

Download the dataset from Kaggle.

1. Log in to Kaggle.
2. It shows 6 files consisting of 3 with HI – Higher Illicit and other 3 with LI- Lower Illicit values files.

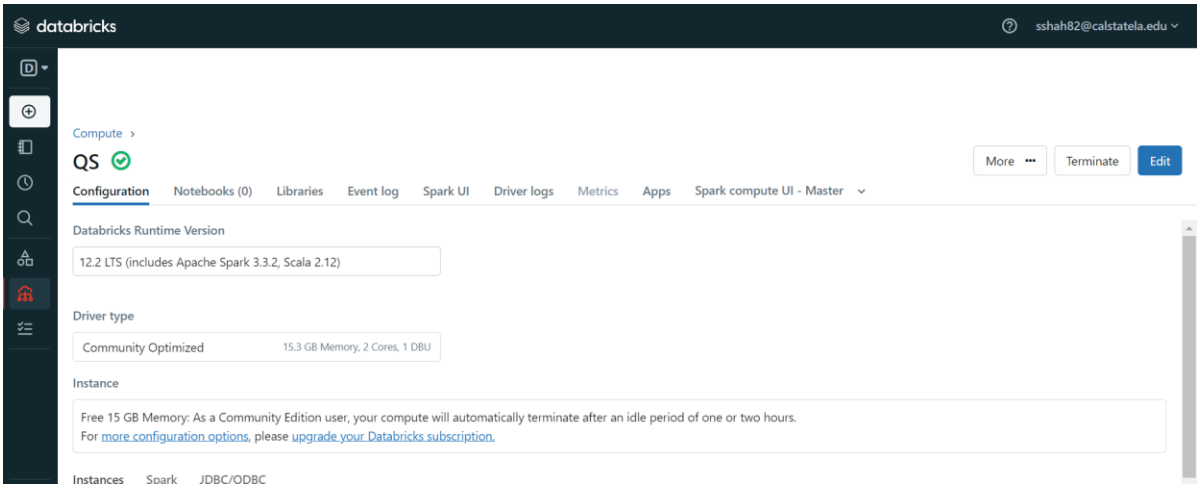
3. Out of these files, 1 small HI- Higher Illicit of 0.4 GB file is used for model training in Databricks.

4. And 1 medium HI – Higher Illicit of 2.82 GB file is used to run models for all algorithms on spark-submit.

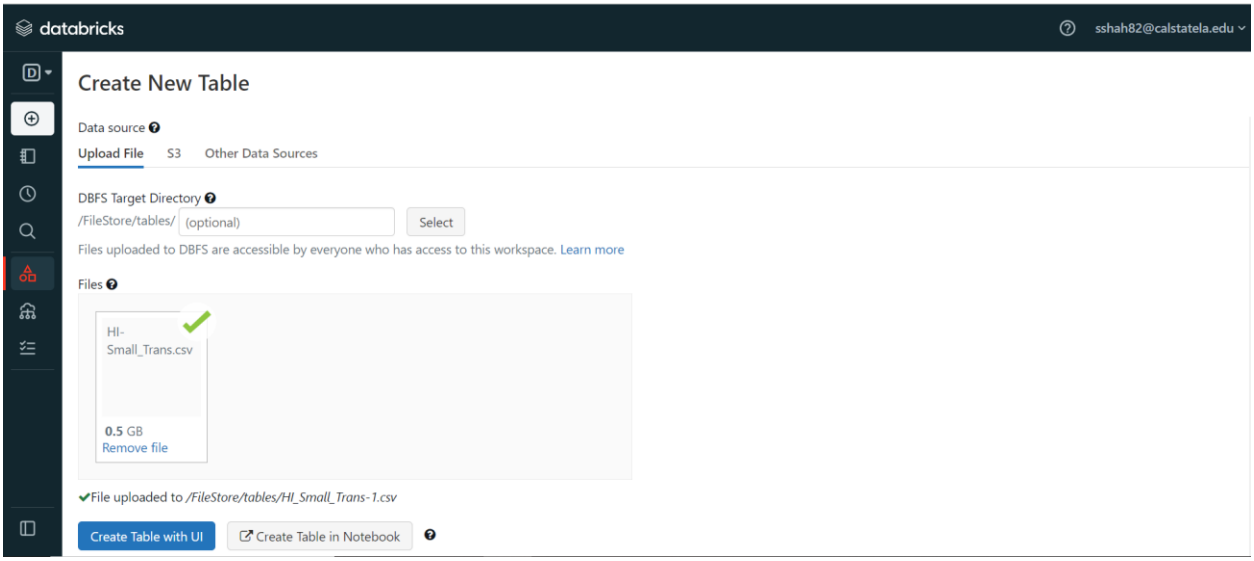
Step 2: Sign in to Databricks.

Login to databricks using the below URL. https://community.cloud.databricks.com/login.html

Step 3: Create a cluster in the databricks.



Step 4: Upload HI_Small_Trans.csv by creating new table in notebook.



Step 5: Import all necessary libraries.

```
# Databricks notebook source
import logging
from pyspark.sql.functions import *
from pyspark.ml import Pipeline
from pyspark.ml.feature import VectorAssembler, StringIndexer, VectorIndexer
from pyspark.ml.tuning import ParamGridBuilder, TrainValidationSplit, CrossValidator
from pyspark.mllib.evaluation import MulticlassMetrics
from pyspark.ml.classification import LogisticRegression, GBTClassifier,
RandomForestClassifier, LinearSVC
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from time import time
from pyspark.context import SparkContext
from pyspark.sql.session import SparkSession
from pyspark.storagelevel import StorageLevel
from pyspark.sql.types import StructType, StructField, IntegerType, StringType, FloatType

# Configure logging
logging.basicConfig(level=logging.INFO, format='%(message)s')
```

```
# Databricks notebook source
import logging
from pyspark.sql.functions import *
from pyspark.ml import Pipeline
from pyspark.ml.feature import VectorAssembler, StringIndexer, VectorIndexer
from pyspark.ml.tuning import ParamGridBuilder, TrainValidationSplit, CrossValidator
from pyspark.mllib.evaluation import MulticlassMetrics
from pyspark.ml.classification import LogisticRegression, GBTClassifier, RandomForestClassifier, LinearSVC
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from time import time
from pyspark.context import SparkContext
from pyspark.sql.session import SparkSession
from pyspark.storagelevel import StorageLevel
from pyspark.sql.types import StructType, StructField, IntegerType, StringType, FloatType

# Configure logging
logging.basicConfig(level=logging.INFO, format='%(message)s')
```

- Use following code in Spark-Submit with 'True'.

Code:

```
IS_SPARK_SUBMIT_CLI = True
if IS_SPARK_SUBMIT_CLI:
    sc = SparkContext.getOrCreate()
    spark = SparkSession(sc)
```

```
IS_SPARK_SUBMIT_CLI = True
if IS_SPARK_SUBMIT_CLI:
    sc = SparkContext.getOrCreate()
    spark = SparkSession(sc)
```

-Following Code is automatically generated after successfully uploaded file in new notebook.

Code:

```
# File location and type
file_location = "/FileStore/tables/HI_Small_Trans.csv"

file_type = "csv"

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

launderingSchema = StructType([
StructField("TimeStamp", StringType(), False),
StructField("From_Bank", IntegerType(), False),
StructField("From_acc", StringType(), False),
```

```
StructField("To_Bank", IntegerType(), False),
StructField("To_acc", StringType(), False),
StructField("Amount_Received", FloatType(), False),
StructField("Receiving_Currency", StringType(), False),
StructField("Amount_Paid", FloatType(), False),
StructField("Payment_Currency", StringType(), False),
StructField("Payment_Format", StringType(), False),
StructField("Is_Laundering", IntegerType(), False)
])

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

display(df)
#print(df)
```

```
# File location and type
file_location = "/FileStore/tables/HI_Small_Trans.csv"
file_type = "csv"

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

launderingSchema = StructType([
StructField("TimeStamp", StringType(), False),
StructField("From_Bank", IntegerType(), False),
StructField("From_acc", StringType(), False),
StructField("To_Bank", IntegerType(), False),
StructField("To_acc", StringType(), False),
StructField("Amount_Received", FloatType(), False),
StructField("Receiving_Currency", StringType(), False),
StructField("Amount_Paid", FloatType(), False),
StructField("Payment_Currency", StringType(), False),
StructField("Payment_Format", StringType(), False),
StructField("Is_Laundering", IntegerType(), False)
])

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

display(df)
```

Output:

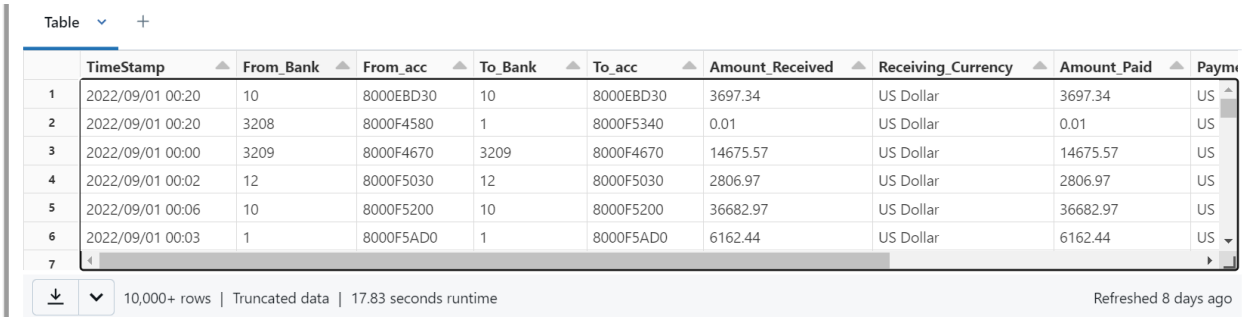


Table +

	TimeStamp	From_Bank	From_acc	To_Bank	To_acc	Amount_Received	Receiving_Currency	Amount_Paid	Paym
1	2022/09/01 00:20	10	8000EBD30	10	8000EBD30	3697.34	US Dollar	3697.34	US
2	2022/09/01 00:20	3208	8000F4580	1	8000F5340	0.01	US Dollar	0.01	US
3	2022/09/01 00:00	3209	8000F4670	3209	8000F4670	14675.57	US Dollar	14675.57	US
4	2022/09/01 00:02	12	8000F5030	12	8000F5030	2806.97	US Dollar	2806.97	US
5	2022/09/01 00:06	10	8000F5200	10	8000F5200	36682.97	US Dollar	36682.97	US
6	2022/09/01 00:03	1	8000F5AD0	1	8000F5AD0	6162.44	US Dollar	6162.44	US
7									

10,000+ rows | Truncated data | 17.83 seconds runtime Refreshed 8 days ago

Step 6: Define Schema & get count of laundering data.

```
df = spark.read.format(file_type) \
.schema(launderingSchema) \
.option("header", first_row_is_header) \
.option("sep", delimiter) \
.load(file_location)

#print(df)
display(df)

#Display the schema
df.printSchema()
```

```
df = spark.read.format(file_type) \
.schema(launderingSchema) \
.option("header", first_row_is_header) \
.option("sep", delimiter) \
.load(file_location)

print(df)

# COMMAND ----------

#Display the schema
df.printSchema()
```

```
print(df.count())
Laundering_df = df.filter(col("Is_Laundering") == 1)
print(Laundering_df.count())
test_df = df.filter(col("Is_Laundering") == 0)
print(test_df.count())
```

```
print(df.count())
Laundering_df = df.filter(col("Is_Laundering") == 1)
print(Laundering_df.count())
test_df = df.filter(col("Is_Laundering") == 0)
print(test_df.count())
```

Step 7: Use of Indexer to transform string data into numeric with new column

```
indexer1 = StringIndexer(inputCol="Receiving_Currency",
outputCol="Receiving_CurrencyIndex")
indexer2 = StringIndexer(inputCol="Payment_Currency",
outputCol="Payment_CurrencyIndex")
indexer3 = StringIndexer(inputCol="Payment_Format", outputCol="Payment_FormatIndex")


df = indexer1.fit(df).transform(df)
df = indexer2.fit(df).transform(df)
df = indexer3.fit(df).transform(df)
```

```
1  indexer1 = StringIndexer(inputCol="Receiving_Currency", outputCol="Receiving_CurrencyIndex")
2  indexer2 = StringIndexer(inputCol="Payment_Currency", outputCol="Payment_CurrencyIndex")
3  indexer3 = StringIndexer(inputCol="Payment_Format", outputCol="Payment_FormatIndex")
4
5
6  df = indexer1.fit(df).transform(df)
7  df = indexer2.fit(df).transform(df)
8  df = indexer3.fit(df).transform(df)
```

Create new dataframe with defining ‘Label’

```
df3 = df.select("From_Bank", "To_Bank", "Amount_Received", "Receiving_CurrencyIndex",
"Amount_Paid", "Payment_CurrencyIndex", "Payment_FormatIndex",
col("Is_Laundering").alias("label"))
df3.show(2)
```

```
1  df3 = df.select("From_Bank", "To_Bank", "Amount_Received", "Receiving_CurrencyIndex", "Amount_Paid", "Payment_CurrencyIndex",
   "Payment_FormatIndex", col("Is_Laundering").alias("label"))
2  df3.show(2)
```

▸ (1) Spark Jobs

▸ df3: pyspark.sql.dataframe.DataFrame = [From_Bank: integer, To_Bank: integer ... 6 more fields]

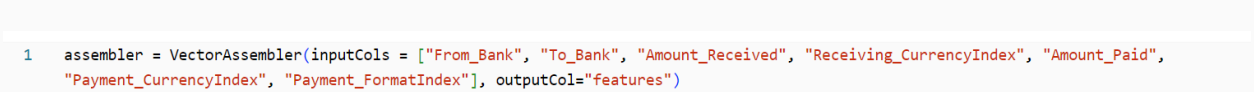
```
+---------+-------+---------------+----------------------+-----------+---------------------+--------------------+-----+
|From_Bank|To_Bank|Amount_Received|Receiving_CurrencyIndex|Amount_Paid|Payment_CurrencyIndex|Payment_FormatIndex|label|
+---------+-------+---------------+----------------------+-----------+---------------------+--------------------+-----+
|       10|     10|        3697.34|                   0.0|    3697.34|                  0.0|                4.0|    0|
|     3208|      1|           0.01|                   0.0|       0.01|                  0.0|                0.0|    0|
+---------+-------+---------------+----------------------+-----------+---------------------+--------------------+-----+
only showing top 2 rows
```

Step 8: Data balancing with UnderSampling

```
#Balancing by undersampling
Laundering_df2 = df3.filter(col("label") == 1)
print(Laundering_df2.count())
test_df2 = df3.filter(col("label") == 0)
print(test_df2.count())
balanced_ratio = Laundering_df2.count() / test_df2.count()
print(balanced_ratio)
undersampled_df2 = test_df2.sample(withReplacement=False, fraction=balanced_ratio)
print(undersampled_df2.count())
df3 = Laundering_df2.union(undersampled_df2)
print(df3.count())
```

Vector Assembler to form arrays of data for features & Label

```
assembler = VectorAssembler(inputCols = ["From_Bank", "To_Bank", "Amount_Received",
"Receiving_CurrencyIndex", "Amount_Paid", "Payment_CurrencyIndex",
"Payment_FormatIndex"], outputCol="features")
```



Step 9: Split the data into Train(70%) & Test Dataset(30%).

```
splits = df3.randomSplit([0.7,0.3])
train = splits[0]
test = splits[1].withColumnRenamed("label", "trueLabel")
print ("Training Rows:", train.count(), " Testing Rows:", test.count())
```

```
1  splits = df3.randomSplit([0.7,0.3])
2  train = splits[0]
3  test = splits[1].withColumnRenamed("label", "trueLabel")
4  print ("Training Rows:", train.count(), " Testing Rows:", test.count())
```

```
▸ (4) Spark Jobs
▸ train:  pyspark.sql.dataframe.DataFrame = [From_Bank: integer, To_Bank: integer ... 6 more fields]
▸ test:  pyspark.sql.dataframe.DataFrame = [From_Bank: integer, To_Bank: integer ... 6 more fields]
Training Rows: 7205  Testing Rows: 3173
```

Step 10: Defining ALL Algorithms with pipeline & Hyper parameter

1) Logistic Regression: -

```
# Define Logistic Regression
lr = LogisticRegression(labelCol="label",featuresCol="features",maxIter=10,regParam=0.3)


# Combine stages into pipeline
pipeline = Pipeline(stages=[assembler, lr])

#Define paramGrid
paramGrid = ParamGridBuilder() \
 .addGrid(lr.regParam, [0.01, 0.1, 1.0]) \
 .addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0]) \
 .build()
```

```
# Define Logistic Regression
lr = LogisticRegression(labelCol="label",featuresCol="features",maxIter=10,regParam=0.3)
```

```
# Combine stages into pipeline
pipeline = Pipeline(stages=[assembler, lr])
```

```
#Define paramGrid
paramGrid = ParamGridBuilder() \
 .addGrid(lr.regParam, [0.01, 0.1, 1.0]) \
 .addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0]) \
 .build()
```

2) LinearSVC: -

```
# Define LinerSVC
lsvc = LinearSVC(labelCol="label",featuresCol="features")

# Combine stages into pipeline
pipeline_lsvc = Pipeline(stages=[assembler, lsvc])

# Define paramGrid
paramGrid_lsvc = (ParamGridBuilder()
.addGrid(lsvc.regParam, [0.01, 0.1, 1.0]) \
.addGrid(lsvc.maxIter, [10, 20, 30]) \
.build())
```

```
# Define LinerSVC
lsvc = LinearSVC(labelCol="label",featuresCol="features")

# Combine stages into pipeline
pipeline_lsvc = Pipeline(stages=[assembler, lsvc])

# Define paramGrid
paramGrid_lsvc = (ParamGridBuilder()
.addGrid(lsvc.regParam, [0.01, 0.1, 1.0]) \
.addGrid(lsvc.maxIter, [10, 20, 30]) \
.build())
```

3) Random Forest: -

```
# Define Random Forest
rf = RandomForestClassifier(labelCol="label",featuresCol="features",numTrees=10, maxDepth=5)

# Combine stages into pipeline
pipeline = Pipeline(stages=[assembler, rf])

# Define paramGrid
paramGrid = ParamGridBuilder() \
    .addGrid(rf.numTrees, [10, 20, 30]) \
    .addGrid(rf.maxDepth, [3, 5, 8]) \
    .build()
```

```
# Define Random Forest
rf = RandomForestClassifier(labelCol="label",featuresCol="features",numTrees=10, maxDepth=5)

# Combine stages into pipeline
pipeline = Pipeline(stages=[assembler, rf])

# Define paramGrid
paramGrid = ParamGridBuilder() \
    .addGrid(rf.numTrees, [10, 20, 30]) \
    .addGrid(rf.maxDepth, [3, 5, 8]) \
    .build()
```

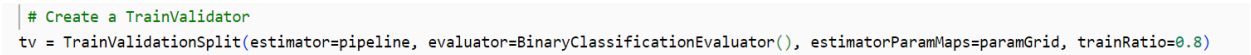
4) Gradient Boosted Tree (GBT): -

```
# Define Gradient Boosted Tree
gbt = GBTClassifier(featuresCol="features", labelCol="label", maxIter=10)

# Combine stages into pipeline
pipeline = Pipeline(stages=[assembler, gbt])

# Define paramGrid
paramGrid = ParamGridBuilder() \
   .addGrid(gbt.maxDepth, [3, 5, 8]) \
   .addGrid(gbt.maxIter, [10, 20, 30]) \
   .build()
```

Step 11: Model train, calculating time and prediction (Same Code apply to all algorithms)

i) With Train-validation Split

```
 # Create a TrainValidator
tv = TrainValidationSplit(estimator=pipeline, evaluator=BinaryClassificationEvaluator(),
estimatorParamMaps=paramGrid, trainRatio=0.8)
```



```
#Training the model and Calculating its time
import time
start_time = time.time() # Start time

tvModel = tv.fit(train)

end_time = time.time()# End time
```

```
print("Model trained!")
# Calculate training time
training_time = end_time - start_time
# Calculate minutes and seconds
minutes = int(training_time // 60)
seconds = int(training_time % 60)

logging.info("Training time: %02d:%02d" % (minutes, seconds))
```

#Prediction:

```
prediction = tvModel.transform(test)
predicted = prediction.select("features", "prediction", "probability", "trueLabel")

predicted.show(100, truncate=False)
```

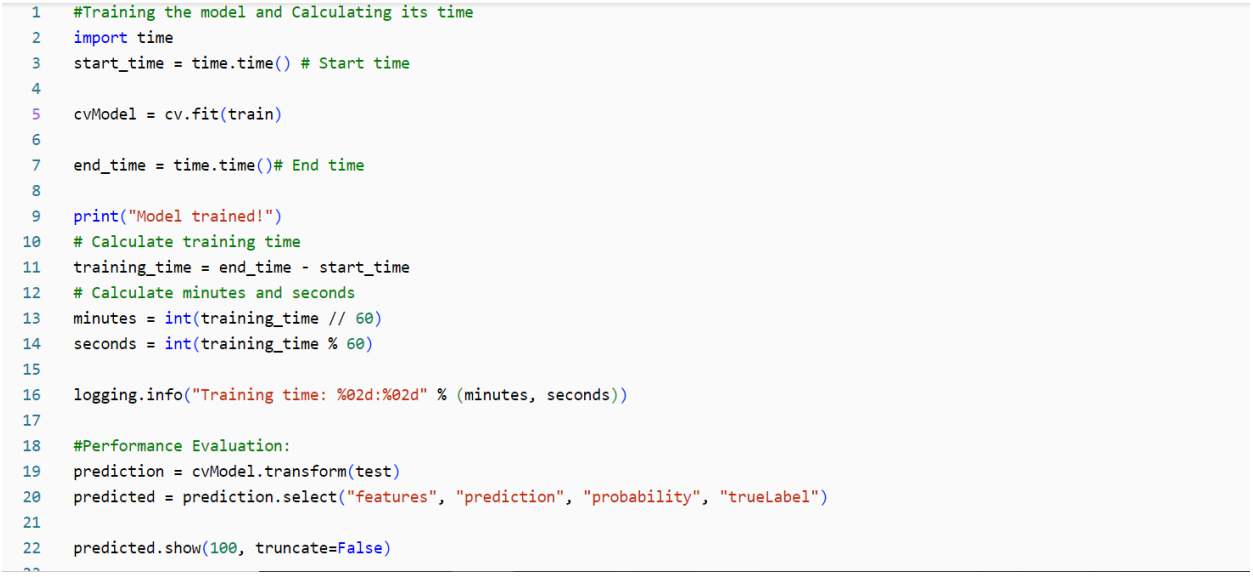
```
#Training the model and Calculating its time
import time
start_time = time.time() # Start time

tvModel = tv.fit(train)

end_time = time.time()# End time

print("Model trained!")
# Calculate training time
training_time = end_time - start_time
# Calculate minutes and seconds
minutes = int(training_time // 60)
seconds = int(training_time % 60)

logging.info("Training time: %02d:%02d" % (minutes, seconds))

#Performance Evaluation:
prediction = tvModel.transform(test)
predicted = prediction.select("features", "prediction", "probability", "trueLabel")

predicted.show(100, truncate=False)
```

ii) With Cross-validation Split

```
# Create a CrossValidator
cv = CrossValidator(estimator=pipeline, evaluator=BinaryClassificationEvaluator(),
estimatorParamMaps=paramGrid, numFolds=3)
```

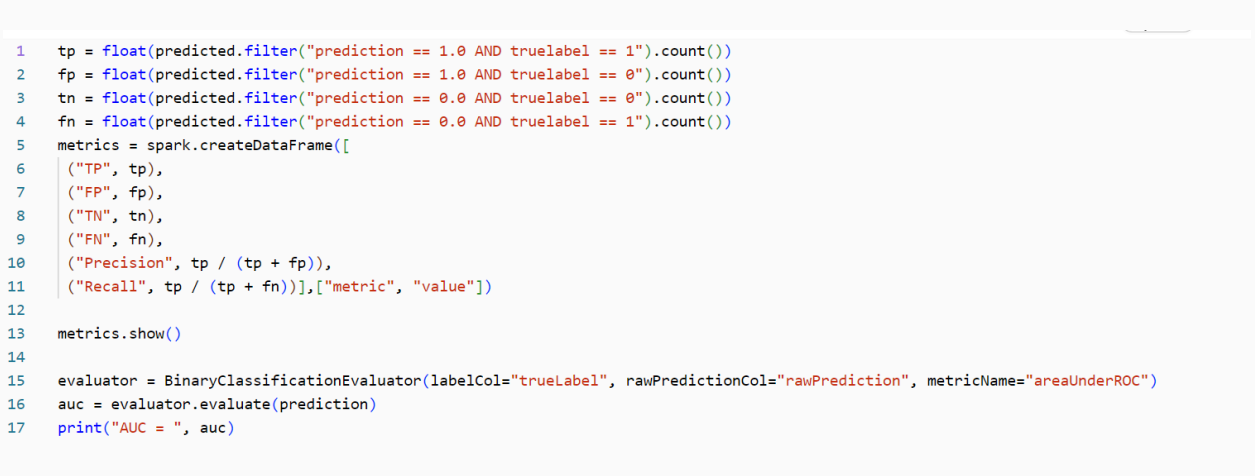
```
# Create a CrossValidator
cv = CrossValidator(estimator=pipeline, evaluator=BinaryClassificationEvaluator(), estimatorParamMaps=paramGrid, numFolds=3)
```

```
#Training the model and Calculating its time
import time
start_time = time.time() # Start time

cvModel = cv.fit(train)

end_time = time.time()# End time

print("Model trained!")
# Calculate training time
training_time = end_time - start_time
# Calculate minutes and seconds
minutes = int(training_time // 60)
seconds = int(training_time % 60)

logging.info("Training time: %02d:%02d" % (minutes, seconds))

#Prediction:

prediction = cvModel.transform(test)
predicted = prediction.select("features", "prediction", "probability", "trueLabel")

predicted.show(100, truncate=False)
```



Step 12: Evaluator & Metrics Code (Recall, Precision, AUC) for TVS/CV

```
tp = float(predicted.filter("prediction == 1.0 AND truelabel == 1").count())
fp = float(predicted.filter("prediction == 1.0 AND truelabel == 0").count())
tn = float(predicted.filter("prediction == 0.0 AND truelabel == 0").count())
fn = float(predicted.filter("prediction == 0.0 AND truelabel == 1").count())
metrics = spark.createDataFrame([
 ("TP", tp),
 ("FP", fp),
 ("TN", tn),
 ("FN", fn),
 ("Precision", tp / (tp + fp)),
 ("Recall", tp / (tp + fn))],["metric", "value"])

metrics.show()

evaluator = BinaryClassificationEvaluator(labelCol="trueLabel", rawPredictionCol="rawPrediction",
metricName="areaUnderROC")
auc = evaluator.evaluate(prediction)
print("AUC = ", auc)
```



```
tp = float(predicted.filter("prediction == 1.0 AND truelabel == 1").count())
fp = float(predicted.filter("prediction == 1.0 AND truelabel == 0").count())
tn = float(predicted.filter("prediction == 0.0 AND truelabel == 0").count())
fn = float(predicted.filter("prediction == 0.0 AND truelabel == 1").count())
metrics = spark.createDataFrame([
 ("TP", tp),
 ("FP", fp),
 ("TN", tn),
 ("FN", fn),
 ("Precision", tp / (tp + fp)),
 ("Recall", tp / (tp + fn))],["metric", "value"])

metrics.show()

evaluator = BinaryClassificationEvaluator(labelCol="trueLabel", rawPredictionCol="rawPrediction", metricName="areaUnderROC")
auc = evaluator.evaluate(prediction)
print("AUC = ", auc)
```

Step 13: Feature Importance: (for example GBT-TVS in our case)

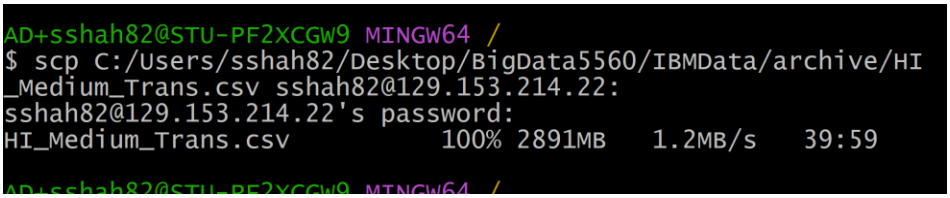
```
# Get the best model from TrainValidationSplit
best_model = tvModel.bestModel
gbtModel = best_model.stages[-1]

# Feature importance
import pandas as pd
featureImp = pd.DataFrame(list(zip(assembler.getInputCols(), gbtModel.featureImportances)), columns=["feature",
"importance"])
featureImp = featureImp.sort_values(by="importance", ascending=False)
```

```
# Get the best model from TrainValidationSplit
best_model = tvModel.bestModel
gbtModel = best_model.stages[-1]

# Feature importance
import pandas as pd
featureImp = pd.DataFrame(list(zip(assembler.getInputCols(), gbtModel.featureImportances)), columns=["feature", "importance"])
featureImp = featureImp.sort_values(by="importance", ascending=False)
```

Steps to run code at Spark ML:

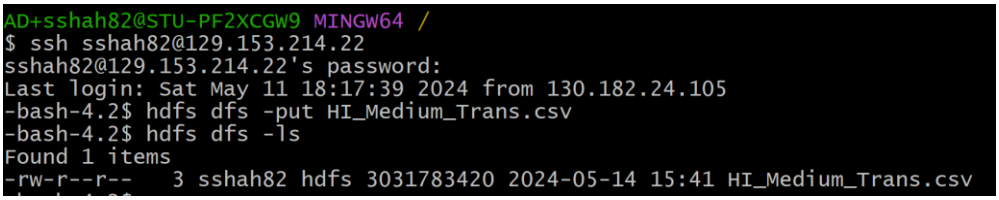
1) Download Big Datafile- HI_Medium_Trans.csv on GitBash through Linux command.

```
$ scp C:/Users/sshah82/Desktop/BigData5560/IBMData/archive/HI_Medium_Trans.csv sshah82@129.153.214.22:
```



2) Move the .csv file to Hadoop file system.

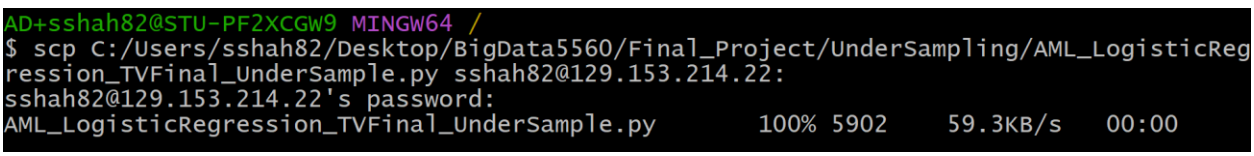
```
bash-4.2$ hdfs dfs -put HI_Medium_Trans.csv
```



3) Store all databricks source files in .py format on local system.

For Ex.

```
scp C:/Users/sshah82/Desktop/BigData5560/Final_Project/UnderSampling/AML_LogisticRegression_TVFinal_UnderSample.py sshah82@129.153.214.22:
```



4) Get all .py files on GitBash through linux commands.

5) All .py files has given path to fetch .csv file from Hadoop file system.

6) Run all .py files on spark-submit.

```
-bash-4.2$ spark-submit AML_LogisticRegression_TVFinal_UnderSample.py
```

-bash-4.2$ spark-submit AML_LogisticRegression_TVFinal_UnderSample.py

Results in SPARK-SUBMIT:

Logistic Regression

Train Validation Split

Model Train time:

```
MINGW64:/c/Users/sshah82
mory (size: 33.7 KiB, free: 366.0 MiB)
24/05/09 12:52:19 INFO BlockManagerInfo: Removed broadcast_436_piece0 on bigdaiwn0.sub03291929060.trainingvcn.oraclevcn.com:19781 in me
mory (size: 33.7 KiB, free: 366.0 MiB)
24/05/09 12:52:19 INFO MemoryStore: Block broadcast_445_piece0 stored as bytes in memory (estimated size 56.9 KiB, free 359.6 MiB)
24/05/09 12:52:19 INFO BlockManagerInfo: Added broadcast_445_piece0 in memory on bigdaimn0.sub03291929060.trainingvcn.oraclevcn.com:132
85 (size: 56.9 KiB, free: 365.7 MiB)
24/05/09 12:52:19 INFO SparkContext: Created broadcast 445 from rdd at ClassificationSummary.scala:191
24/05/09 12:52:19 INFO FileSourceScanExec: Planning scan with bin packing, max size: 134217728 bytes, open cost is considered as scanni
ng 4194304 bytes.
24/05/09 12:52:19 INFO MemoryStore: Block broadcast_446 stored as values in memory (estimated size 568.0 KiB, free 359.0 MiB)
24/05/09 12:52:19 INFO MemoryStore: Block broadcast_446_piece0 stored as bytes in memory (estimated size 56.9 KiB, free 359.0 MiB)
24/05/09 12:52:19 INFO BlockManagerInfo: Added broadcast_446_piece0 in memory on bigdaimn0.sub03291929060.trainingvcn.oraclevcn.com:132
85 (size: 56.9 KiB, free: 365.6 MiB)
24/05/09 12:52:19 INFO SparkContext: Created broadcast 446 from rdd at ClassificationSummary.scala:191
24/05/09 12:52:19 INFO FileSourceScanExec: Planning scan with bin packing, max size: 134217728 bytes, open cost is considered as scanni
ng 4194304 bytes.
Model trained!
Training time: 02:26
24/05/09 12:52:19 INFO FileSourceStrategy: Pushed Filters:
24/05/09 12:52:19 INFO FileSourceStrategy: Post-Scan Filters:
24/05/09 12:52:19 INFO FileSourceStrategy: Output Data Schema: struct<From_Bank: int, To_Bank: int, Amount_Received: float, Receiving_C
```

Metrics:

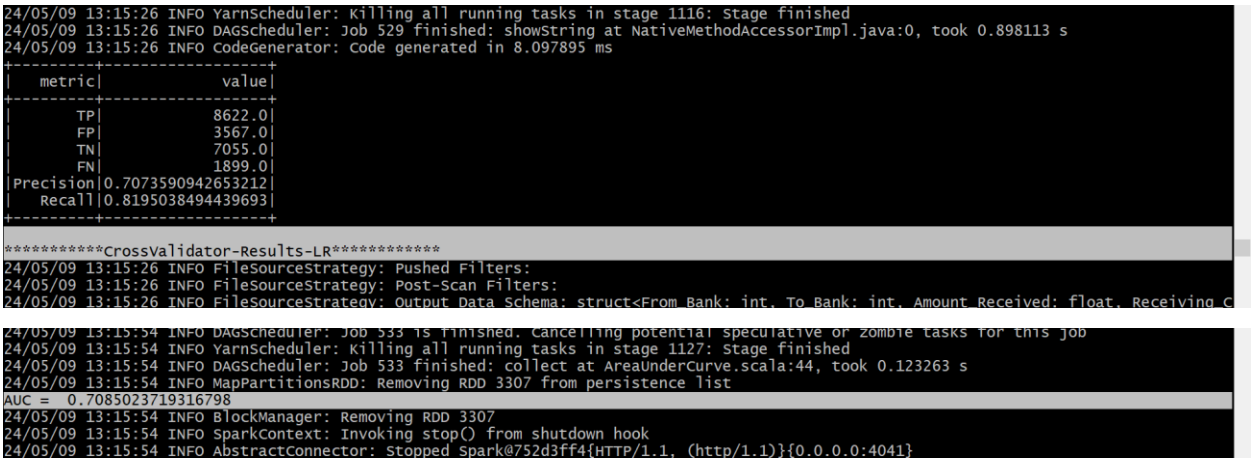
```
24/05/09 12:54:18 INFO CodeGenerator: Code generated in 8.472779 ms
+---------+------------------+
|   metric|             value|
+---------+------------------+
|       TP|            8752.0|
|       FP|            2976.0|
|       TN|            7359.0|
|       FN|            1704.0|
|Precision|0.7462482946793997|
|   Recall|0.8370313695485846|
+---------+------------------+

**********TrainValidator-Results-LR**********
24/05/09 12:54:18 INFO FileSourceStrategy: Pushed Filters:
24/05/09 12:54:18 INFO FileSourceStrategy: Post-Scan Filters:
```

```
24/05/09 12:54:45 INFO DAGScheduler: Job 189 finished: collect at AreaUnderCurve.scala:44, took 0.144032 s
24/05/09 12:54:45 INFO MapPartitionsRDD: Removing RDD 1195 from persistence list
AUC =  0.708159693496631
24/05/09 12:54:45 INFO BlockManager: Removing RDD 1195
24/05/09 12:54:45 INFO SparkContext: Invoking stop() from shutdown hook
```

CrossValidation Split –

Model Train time:

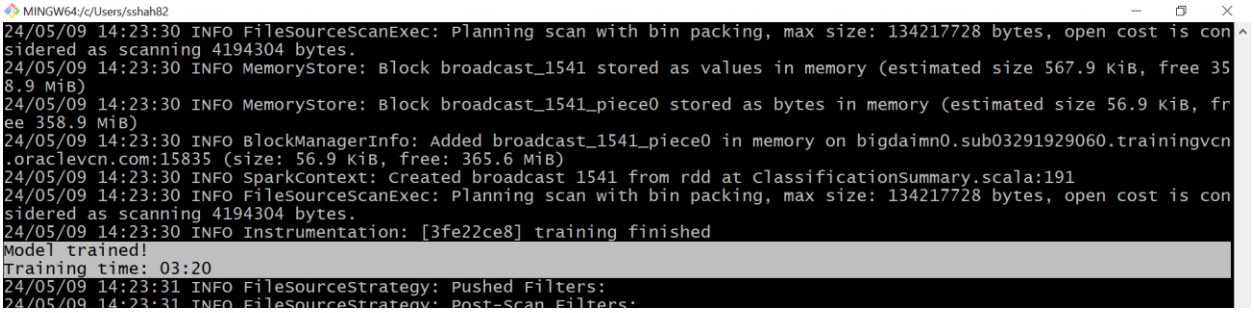
Metrics:



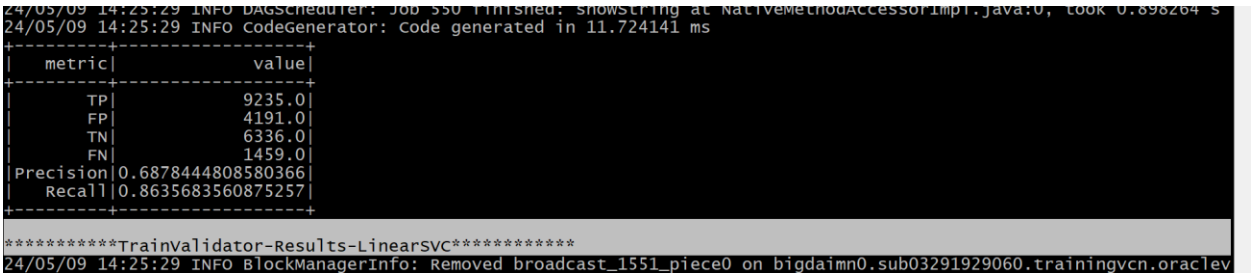
Linear SVC:

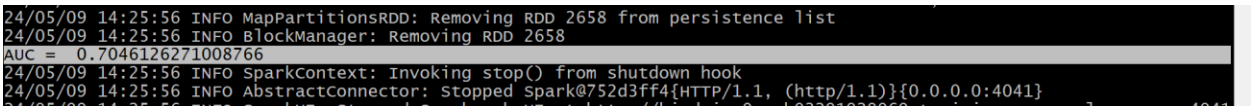
TrainValidation Split –

Model Train time:



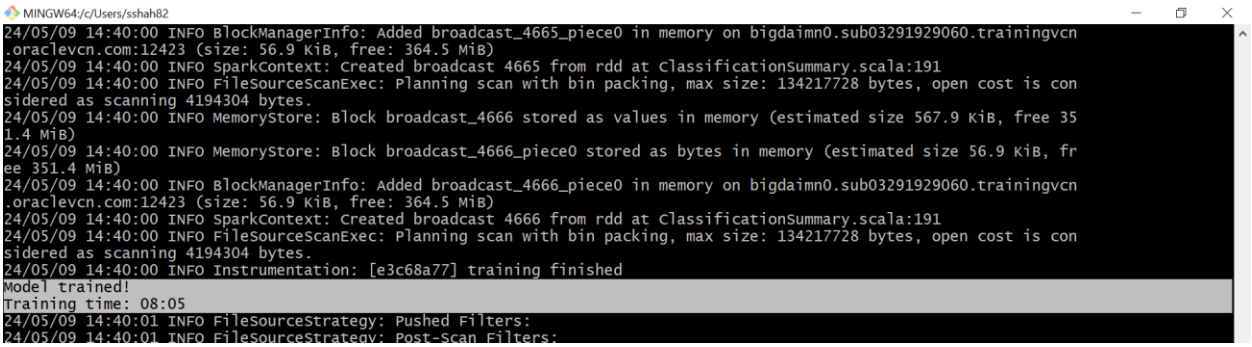
Metrics:



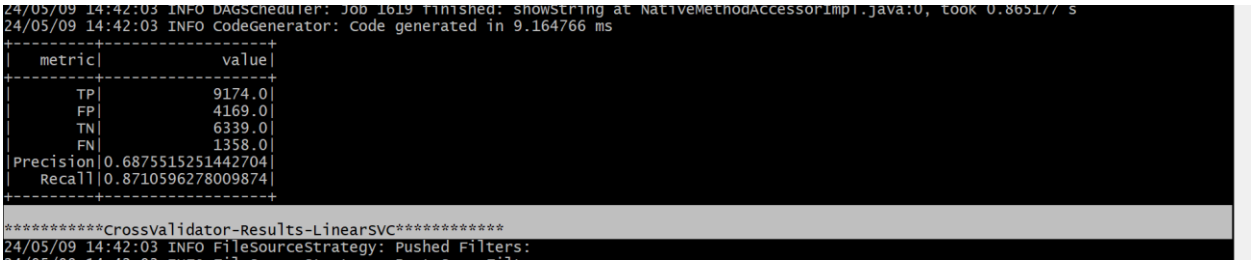


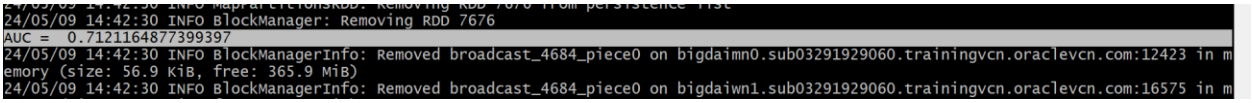
CrossValidation Split –

Model Train time:



Metrics:

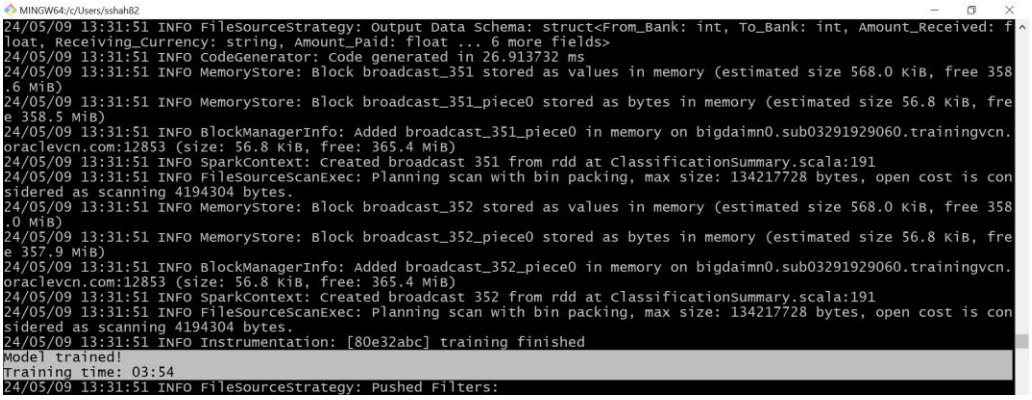




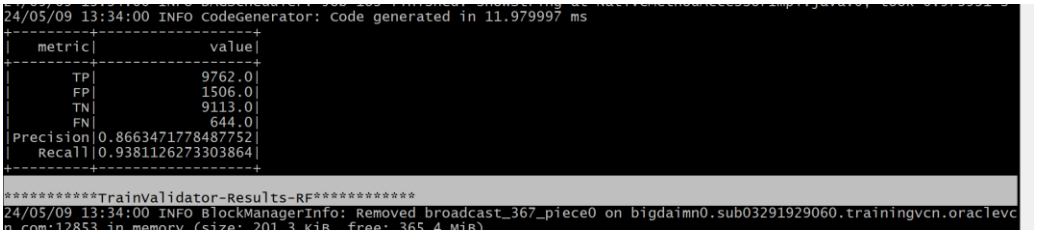
Random Forest

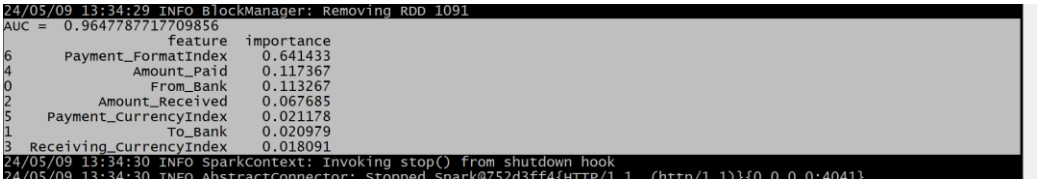
TrainValidation Split –

Model Train time:



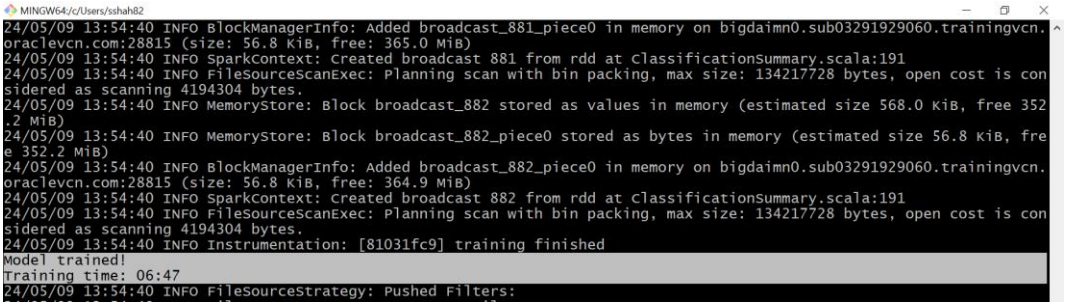
Metrics & Feature Importance:



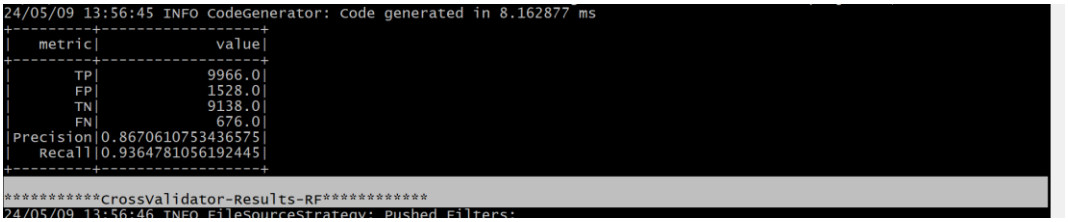


CrossValidation Split –

Model Train time:



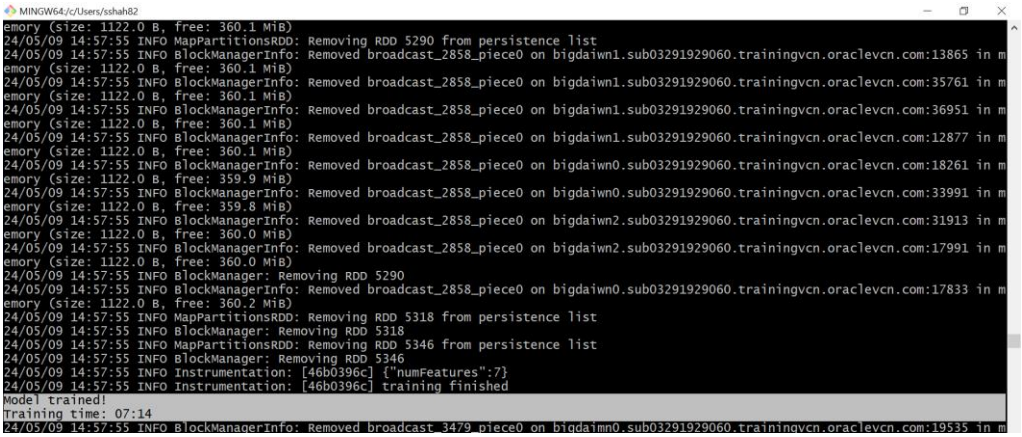
Metrics & Feature Importance:



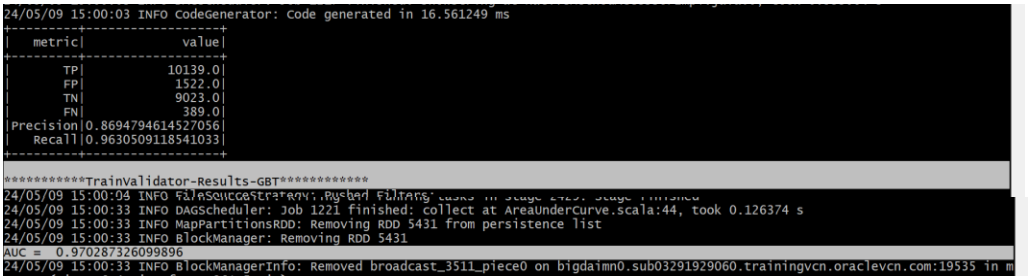
Gradient Boosted Tree

TrainValidation Split –

Model Train time:

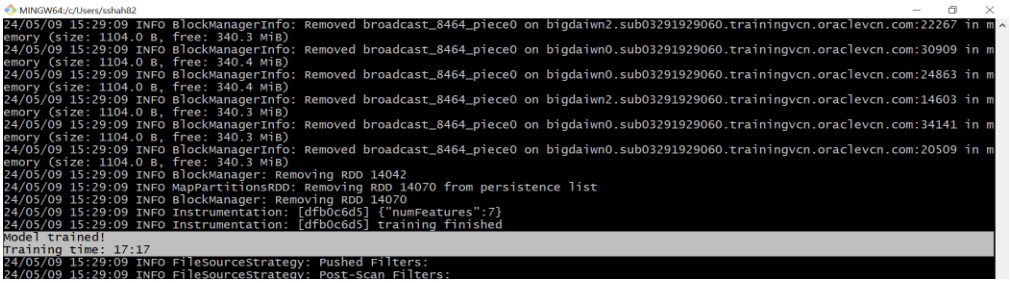


Metrics & Feature Importance:

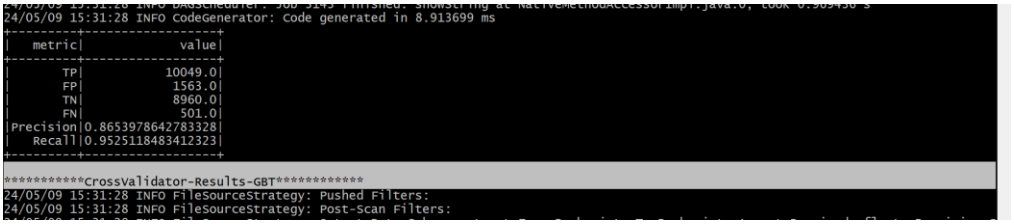


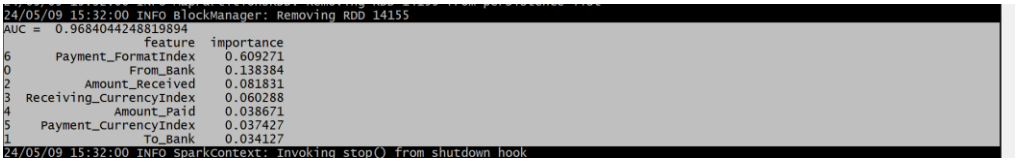
CrossValidation Split –

Model Train time:



Metrics & Feature Importance:





Summary:

AML- Algorithms with UnderSampling								
Algorithm	**TP**	**FP**	**TN**	**FN**	**Precision**	**Recall**	**AUC**	**Time**
	Train Validation Split							
Logistic Regression	8752	2976	7359	1704	0.7462	0.8370	0.7082	2.26m
LinearSVC	9235	4191	6336	1459	0.6878	0.8635	0.7046	3.20m
Random Forest	9762	1506	9113	644	0.8663	0.9381	0.9648	3.54m
Gradient Boosted Tree	10139	1522	9023	389	0.8695	0.9631	0.9703	7.14m
	Cross Validation Split							
Logistic Regression	8622	3567	7055	1899	0.7074	0.8195	0.7085	5.18m
LinearSVC	9174	4169	6339	1358	0.6876	0.8711	0.7121	8.05m
Random Forest	9966	1528	9138	676	0.8671	0.9365	0.9655	6.47m
Gradient Boosted Tree	10049	1563	8960	501	0.8654	0.9525	0.9684	17.17m

Feature Importance for GBT-Train Validation Split

feature	importance
Payment_FormatIndex	0.618812
From_Bank	0.137167
Amount_Received	0.062771
Receiving_CurrencyIndex	0.060838
Amount_Paid	0.048353
Payment_CurrencyIndex	0.036077
To_Bank	0.035981

Conclusion:

- Best Model is **Gradient Boosted Tree (GBT) with Train Validation Split (TVS)** due to highest Recall and AUC values with less time.

- 'Payment Format', specifically ACH type, is the most important feature responsible for money laundering instances.

	GBT - TV
Recall	0.9631
AUC	0.9703
Time	7.14m

References:

1. URL of Data Source:

https://www.kaggle.com/datasets/ealtman2019/ibm-transactions-for-anti-money-laundering-aml

2. URL of GitHub:

https://github.com/ssarkar4/AntiMoneyLaundering_BigData

3. URL of References:

https://spark.apache.org/docs/2.2.0/ml-pipeline.html

https://spark.apache.org/docs/latest/ml-tuning.html