INDIAN INSTITUTE OF TECHNOLOGY MADRAS

Big Data Laboratory - CS4830
Final Project- Team Event Horizon

G Prashant (BS17B011)
Jeswant Krishna K (AE17B030)
Avinash G Bagali (AE17B110)

May 16, 2021

Contents

1 Project Overview and Objective

The task of this project is to perform big data processing and analysis using tools such as Apache Spark and Kafka. The dataset assigned to our team is **"NYC Parking Tickets"** dataset, which contains a large number of records pertaining to parking violations in the City of New York. Each record consists of several details including Registration State, Vehicle Make, Issuing Agency, Violation Location, etc (https://www.kaggle.com/new-york-city/nyc-parking-tickets). The first task is to perform preprocessing and model training using batch computation using PySpark run on a Dataproc Cluster. Precisely, the goal is to train a supervised classification model for predicting the label "Violation County" using the given data, i.e., for predicting the County where a violation has occurred. The next task is to use the saved model stored in Google Cloud Storage (GCS) bucket to perform real-time analysis of streaming data using Apache Kafka. This report provides all the necessary implementational details for each task of the project, right from preprocessing, model training, hyperparameter tuning, model selection to real-time prediction with appropriate screenshots.

2 Task 1: Batch Computation

2.1 Data Exploration and Preprocessing

The dataset contains 51 features in total, most of them being categorical features. These features provide all the information about the parking ticket issued in the City of New York. Figure 1 provides a glimpse of the first batch of the dataset, displayed as a Pandas Dataframe.

Summons Number	Plate ID	Registration State	Plate Type	Issue Date	Violation Code	Vehicle Body Type	Vehicle Make	Issuing Agency	Street Code1	Street Code2	Street Code3	Vehicle Expiration Date	Issuer Code	Issuer Command	Issuer Squad
7922553869	33684MD	NY	COM	06/17/2016	42	VAN	FORD	T	34550	10410	10510	20170430	345557	T501	B
7922553870	60996MC	NY	COM	06/17/2016	69	VAN	CHEVR	T	34550	10410	10510	88888888	345557	T501	B
7922553882	87642ME	NY	COM	06/17/2016	69	VAN	CHEVR	T	34550	10410	10510	20170930	345557	T501	B
7922553894	71985KA	NY	COM	06/17/2016	69	VAN	DODGE	T	34530	0	0	20160701	345557	T501	B
7922553900	66460AN	NY	COM	06/17/2016	69	VAN	GMC	T	34550	0	0	20180531	345557	T501	B
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
8359500927	FFM1969	NY	PAS	06/08/2016	38	SUBN	TOYOT	T	25590	56890	78820	20161128	362197	T201	P
8359500940	2482082	IN	PAS	06/08/2016	51	VAN	FRUEH	T	73980	40404	40404	88880088	362197	T201	P
8359500952	65919JW	NY	COM	06/08/2016	52	VAN	FRUEH	T	0	0	0	88888888	362197	T201	P
8359500964	68718MG	NY	COM	06/08/2016	82	PICK	DODGE	T	0	0	0	20170531	362197	T201	P

Figure 1: Screenshot showing a few records and features of the first batch of the NYC dataset

The target label column is *Violation_County*, which specifies the County in which the violation occurred. It takes one of the following four values.

- BX
- K
- NY
- Q

Figure 2 shows a bar plot of the number of occurrences of each County in the first batch of the NYC Dataset.

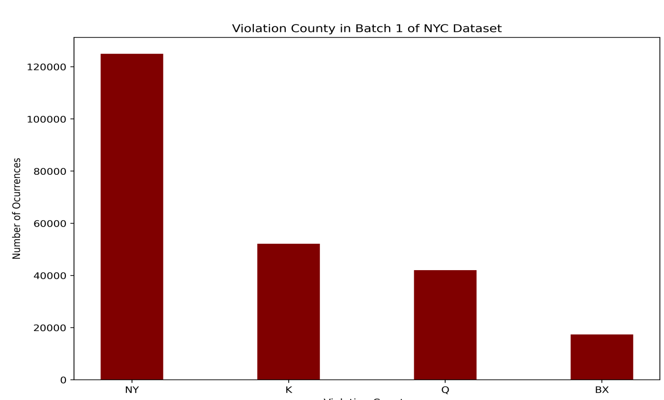

Figure 2: Violation County in Batch 1 of NYC Dataset

From the above figure, it can noticed that most of the violations have occurred in the NY County.

Based on a visual inspection of the first batch of the NYC dataset, it was evident that multiple features had missing values across all the samples of the batch. Further, there were some features which were irrelevant for the purpose of the prediction. Hence, preprocessing the dataset to extract useful features was undoubtedly necessary for this task. The following sections provide all details on the preprocessing steps performed.

2.1.1 Feature Selection

The following features had missing values across all of the data samples in a batch, and were hence removed.

- No Standing or Stopping Violation
- Hydrant Violation
- Double Parking Violation
- Latitude
- Longitude
- Community Board
- Community Council
- Census Tract
- BIN
- BBL
- NTA
- Plate ID
- Unregistered Vehicle?
- Violation Legal Code

Secondly, it was important to remove unnecessary features that might not be of potential importance for the purpose of predicting the Violation County. Given below are features that were removed either because they carried redundant information or because they were not required.

- Summons Number
- Vehicle Expiration Date
- Time First Observed
- Date First Observed
- From Hours In Effect
- To Hours In Effect
- Days Parking In Effect
- House Number
- Issuer Code

After removing all of the above features, the dataset contained 24 features, which are mentioned below.

- Registration State
- Plate Type
- Issue Date
- Violation Code
- Vehicle Body Type
- Vehicle Make
- Issuing Agency
- Street Codes 1, 2 and 3
- Issuer Command
- Issuer Squad
- Violation Time
- Violation In Front Of Or Opposite
- Street Name
- Intersecting Street
- Law Section
- Sub Division
- Vehicle Color
- Vehicle Year
- Meter Number
- Feet From Curb
- Violation Post Code
- Violation Description

2.1.2 Handling Categorical Variables

Now that the important features are selected, the next step is to handle categorical features and other non-numeric features. We considered all the features except *Violation Code, Street Codes (1, 2 and 3), Vehicle Year and Feet From Curb* as categorical and non-numeric features. The features that involve representing Date and Time - *Issue Date* and *Violation Time* were transformed in a special manner for numerical representation. The details of the same given below.

<u>Issue Date:</u>

This feature specifies the date on which the parking ticket is issued, in `MM/DD/YYYY` format. We developed a User Defined Function (UDF) `dateParse` that will transform the date in this format to `YYYYMMDD`, of integer type. This way, it becomes a numerical feature, with the order of the dates being preserved, i.e., higher value indicates a later date. Figure 3 shows the numerical transformation of the feature *Issue Date.*

Issue Date
06/17/2016
06/17/2016
06/17/2016
06/17/2016
06/17/2016
06/17/2016
06/17/2016
06/27/2016
06/27/2016
06/27/2016
06/27/2016
06/27/2016
06/15/2016
06/15/2016
06/15/2016
06/15/2016
06/15/2016
06/15/2016
06/15/2016
06/15/2016

Issue Date
20160617
20160617
20160617
20160617
20160617
20160617
20160617
20160627
20160627
20160627
20160627
20160627
20160615
20160615
20160615
20160615
20160615
20160615
20160615
20160615

Figure 3: Numerical Transformation of Issue Date Feature

<u>Violation Time</u>

This feature specifies the time at which the parking ticket is issued, in `HHMMA` or `HHMMP` 12-hour format, where `A` and `P` represents AM and PM respectively. To preserve the order of this feature, a UDF was defined that can transform the time in the given format to `HHMM` 24-hour format, which is of integer type. This way, a higher value denotes a later time. Figure 4 shows the numerical transformation of the feature *Violation Time*.

Violation Time
1023A
1037A
1046A
1115A
1218P
1225P
1237P
0636A
0639A
0646A
0654A
0705A
0805A
0822A
0827A
0831A
0907A
0918A
0945A
0950A

Violation Time
1023
1037
1046
1115
1218
1225
1237
636
639
646
654
705
805
822
827
831
907
918
945
950

Figure 4: Numerical Transformation of Violation Time Feature

For the other remaining categorical features, it was clearly noticable that "One-Hot Encoding (OHE)" would be the best way to represent most of them, because they did not have an inherent order (for example, *Registration State - NY, PA*). Nevertheless, it is important to note that many of these features had a large number of unique values, which makes OHE an infeasible option. Therefore, Label Encoding was performed for these features using `StringIndexer` in PySpark. Moreover, even the target label *Violation County* was numerically encoded using `StringIndexer`.

2.1.3 Imputation

Imputation was performed to fill the missing values in the dataset with `Imputer` in PySpark. The missing values were imputed with the mode of the corresponding feature column, i.e., with the frequently occurring value. Imputing values with mode is more suitable for features that are non-numeric in nature.

2.1.4 Standard Scaling

The data was scaled by subtracting the feature-wise average and dividing by their standard deviation, as given for any feature x below,

$$\hat{x} = \frac{x - \bar{x}}{s_x}$$

where $\bar{x}$ is the average and s_x is the standard deviation for feature x. This transformation was performed using `StandardScaler` in PySpark. This transformation was performed to scale the data to lie between 0 and 1 for better stability during model training.

2.1.5 Assembly

`VectorAssembler` was used to assemble all the 24 selected and transformed features, prior to model training. They were collectively named as `features` and the target column was named as `label`.

NOTE: Analysis of features and feature selection were performed with the first batch of the NYC Dataset as reference. Our assumption here is that our feature engineering steps will generalize well with other batches of the dataset.

2.2 Model Training

Three popular supervised machine learning models were used to train on the dataset - **Random Forest Classifier, Logistic Regression, Gaussian Naive Bayes**. Hyperparameter tuning was performed and the validation accuracies were considered for choosing the best model. Training multiple models and hyperparameter tuning were made easier using `TrainValidationSplit` in PySpark, where the training size was specified to be 90% of the total dataset. The remaining 10% is the validation set.

2.2.1 Hyperparameter Tuning and Model Selection

Model	Hyperparameters		Validation Accuracy
RandomForestClassifier	maxDepth	numTrees	
	20	**50**	**99.979 %**
	20	100	99.958 %
	30	50	99.979 %
	30	100	99.958 %
LogisticRegression	RegularizationParameter	ElasticNetParameter	
	0	0	69.32335 %
	0	0.5	69.32335 %
	0.5	0	55.00664 %
	0.5	0.5	47.21852 %
	1	0	50.22741 %
	1	0.5	47.21852 %
NaiveBayes	Smoothing		
	0.1		44.37758 %
	0.5		44.37758 %
	1.0		44.37758 %

Table 1: Hyperparameter Tuning of Models

As seen from the Table 1 the best model was Random Forest Classifier with hyperparameters of **maxDepth=20** and **numTrees=50** and has **ValidationAccuracy=99.979 %**. Even though one more model has same accuracy this was chosen as the best because of the lower complexity, in accordance with Occam's Razor principle which states that simple models are better. The next part shows the screenshots of the logs obtained after training of each of the classification models. From this table, it can also be noticed that Logistic Regression model performs better when compared to with regularization (although much worse than Random Forests). Gaussian Naive Bayes model is insensitive to tuning the `smoothing` hyperparameter, with the performances being the worst compared to Random Forest Classifier and Logistic Regression.

2.2.2 Screenshots of Google Cloud Console Dataproc Job Log

`RandomForestClassifier`:

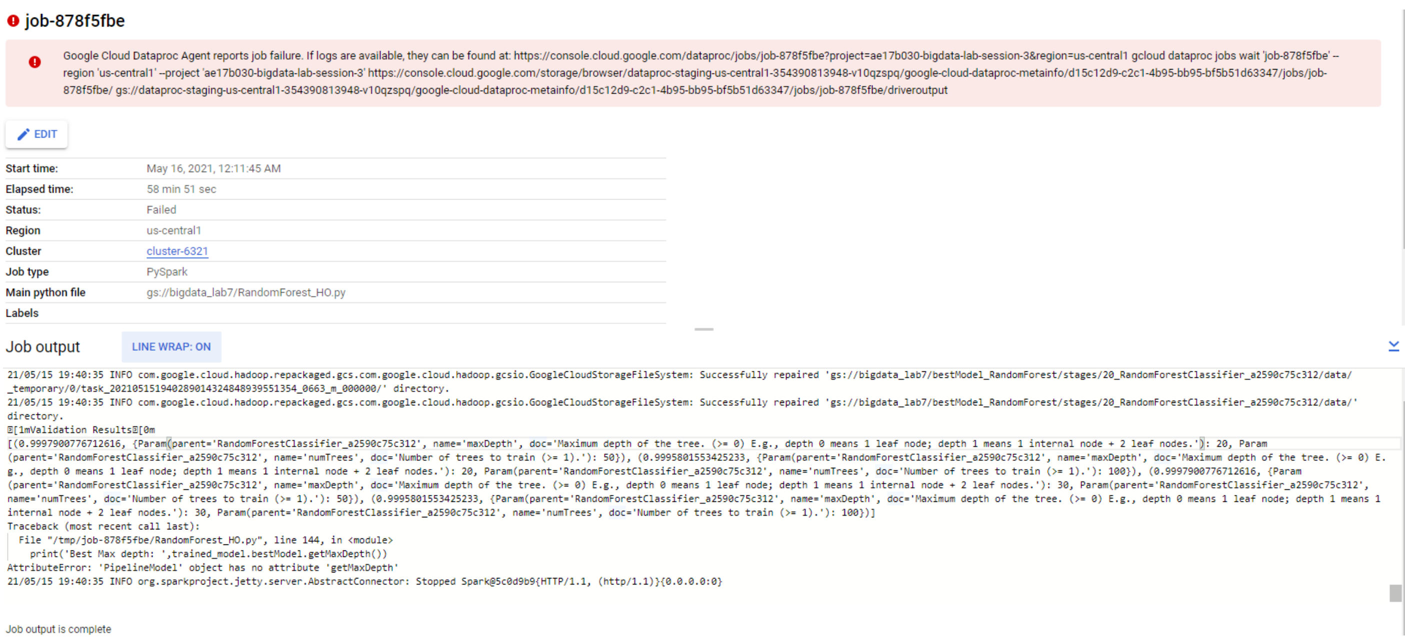

Figure 5: Screenshot showing completed training of RandomForestClassifier along with Validation Accuracy printed

Important Note :- The screenshot is showing job failure but actually this was due to an error in the print statement after the model training was completed.

`LogisticRegression`:

Figure 6: Screenshot showing completed training of LogisticRegression along with Validation Accuracy printed

`NaiveBayes` (Gaussian):

Figure 7: Screenshot showing completed training of NaiveBayes along with Validation Accuracy printed

3 Task 2: Real-time Computation

We read the data from the given Google Cloud Storage bucket and published it to the Kafka topic using publisher.py. Subscriber.py reads the data published to the Kafka topic and stores the data as a streaming Spark dataframe. We loaded the saved model stored in the GCS bucket and performed real-time predictions on it and measured the accuracy and F1-score batch-wise using the `foreachBatch` function.

3.1 Latency

In a period of 9.5 seconds, 100 rows were published to the Kafka topic 'PROJECT', out of which the results of 46 were printed to the console.

3.2 Real-time Prediction Results

```
+----------------+-----+----------+
|Violation_County|label|prediction|
+----------------+-----+----------+
|              NY|  0.0|       0.0|
|               Q|  2.0|       2.0|
|               K|  1.0|       1.0|
|              NY|  0.0|       0.0|
|               Q|  2.0|       2.0|
|               K|  1.0|       1.0|
|               K|  1.0|       1.0|
|              NY|  0.0|       0.0|
|               K|  1.0|       1.0|
|              BX|  3.0|       3.0|
|               K|  1.0|       1.0|
|              NY|  0.0|       0.0|
|               Q|  2.0|       2.0|
|               Q|  2.0|       2.0|
|               Q|  2.0|       2.0|
|               Q|  2.0|       2.0|
|               Q|  2.0|       2.0|
|              NY|  0.0|       0.0|
|               Q|  2.0|       2.0|
|               K|  1.0|       1.0|
+----------------+-----+----------+
```

Figure 8: Real-time predictions for Batch 1

```
|Number of rows in batch 1 = 23|
|  Accuracy on batch 1 is 1.0 |
|  F1 score on batch 1 is 1.0 |
+-----------------------------+
```

Figure 9: Accuracy and F1-score for Batch 1

```
+----------------+-----+----------+
|Violation_County|label|prediction|
+----------------+-----+----------+
|              BX|  3.0|       3.0|
|               K|  1.0|       1.0|
|              NY|  0.0|       0.0|
|               K|  1.0|       1.0|
|              NY|  0.0|       0.0|
|              NY|  0.0|       0.0|
|               Q|  2.0|       2.0|
|              NY|  0.0|       0.0|
|               Q|  2.0|       2.0|
|               K|  1.0|       1.0|
|              NY|  0.0|       0.0|
|              NY|  0.0|       0.0|
|              NY|  0.0|       0.0|
|              NY|  0.0|       0.0|
|               Q|  2.0|       2.0|
|              BX|  3.0|       3.0|
|              NY|  0.0|       0.0|
|              BX|  3.0|       3.0|
|              NY|  0.0|       0.0|
|              BX|  3.0|       3.0|
```

Figure 10: Real-time predictions for Batch 2

```
|Number of rows in batch 3 = 21|
|  Accuracy on batch 3 is 1.0 |
|  F1 score on batch 3 is 1.0 |
+-----------------------------+
```

Figure 11: Accuracy and F1-score for Batch 2

4 Inferences and Conclusion

- In this project, we have processed and analysed the massive-sized NYC Parking Ticket dataset using Apache PySpark by implementing supervised classification algorithms to predict the Violation County.
- Preprocessing the dataset is an extremely essential step prior to model training, due to the noisy and incomplete nature of the dataset.
- By implementing multiple classification models such as Random Forests, Logistic Regression and Gaussian Naive Bayes, and tuning the hyperparameters we observed that the best performing model was Random Forest Classifier with maximum depth 20 and number of trees 50 (with all other hyperparameters taking default values). The best model validation accuracy was 99.979%.
- Logistic Regression and Naive Bayes models yielded low validation accuracies.

- Given that the NYC dataset contains predominantly categorical and non-numeric features, it is evident that tree-based models like Decision Trees and Random Forests are well-suited for the task of predicting the violation County. On the other hand, Logistic Regression and probabilistic models like Naive Bayes cannot perform well unless the complexity is increased - for example, polynomial feature transformation.
- Training each class of algorithms (with hyperparamter tuning) took approximately 1 hour for the massively large training size of close to 40 million samples. This highlights the power of training models in the Cloud using Big Data pipelines.
- After saving the best model (Random Forest Classifier) in GCS bucket, we performed real-time prediction using Apache Kafka. Kafka producer is responsible for sending/publishing the rows of the dataset in batches as messages to a topic. Kafka consumer is responsible for retrieving the messages from the topic, thereby enabling real-time analysis.
- There is a high latency in the number of rows published to Kafka and the number of rows being processed. In our case, the latency seems to be caused primarily due to the Dataproc cluster and we can reduce this latency by scaling this cluster.

5 PySpark Codes

5.1 Pyspark Code for RandomForestClassifier Model

```python
#Importing Requirements
from __future__ import print_function
from pyspark.context import SparkContext
from pyspark.ml.linalg import Vectors
from pyspark.ml.classification import RandomForestClassifier
from pyspark.sql.session import SparkSession
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.sql.types import DoubleType
from pyspark.ml.feature import StringIndexer, Imputer
from pyspark.ml import Pipeline
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.feature import StandardScaler
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder, TrainValidationSplit
from pyspark.sql import functions as F

from pyspark.sql.types import IntegerType
from functools import reduce

sc = SparkContext()
spark = SparkSession(sc)

data_dir = "gs://bdl2021_final_project/nyc_tickets_train.csv"

df = spark.read.format("csv").option("recursiveFileLookup", "true").option("pathGlobFilter", "*.csv").load(data_dir,inferSchema=True, header = True)

drop_cols = ['No Standing or Stopping Violation',
             'Hydrant Violation', 'Double Parking Violation', 'Latitude',
             'Longitude', 'Community Board', 'Community Council', 'Census Tract',
             'BIN', 'BBL', 'NTA','Plate ID','Summons Number','Vehicle Expiration Date','Time First Observed','Date First Observed','From Hours In Effect','To Hours In Effect','Unregistered Vehicle?','Days Parking In Effect','House Number','Violation Legal Code','Issuer Code']

req_cols = ["Registration State", "Plate Type", "Issue Date",  "Violation Code", "Vehicle Body Type",  "Vehicle Make",  "Issuing Agency", "Street Code1",  "Street Code2", "Street Code3",
            "Issuer Command",  "Issuer Squad",  "Violation Time",  "Violation In Front Of Or Opposite", "Street Name",  "Intersecting Street",  "Law Section", "Sub Division", "Vehicle Color",
            "Vehicle Year", "Meter Number", "Feet From Curb", "Violation Post Code", "Violation Description"]

drop_cols = list(map(lambda x: "_".join(x.split()), drop_cols))
req_cols = list(map(lambda x: "_".join(x.split()), req_cols))

oldColumns = df.schema.names
df = reduce(lambda df, idx: df.withColumnRenamed(oldColumns[idx], "_".join(oldColumns[idx].split())), range(len(oldColumns)), df)

df.drop(*drop_cols)

string_cols = req_cols.copy()
string_cols.remove("Violation_Code")
string_cols.remove("Street_Code1")
string_cols.remove("Street_Code2")
string_cols.remove("Street_Code3")
string_cols.remove("Vehicle_Year")
string_cols.remove("Feet_From_Curb")

#Remove Extra two columns
string_cols.remove("Issue_Date")
string_cols.remove("Violation_Time")

#Defining two udfs
@F.udf(returnType=IntegerType())
def dateParse(dstr):
  BrStr = dstr.split("/")
  NewBrStr = [BrStr[2], BrStr[0], BrStr[1]]
  DateAsNo = int("".join(NewBrStr))
  return DateAsNo

@F.udf(returnType=IntegerType())
def timeParse(tstr):
  try:
    NoPart = tstr[:-1]
    dayLight = tstr[-1]
    if(dayLight == 'A'):
      return(int(NoPart))
    else:
      MM = NoPart[-2:]
      HH = str(int(NoPart[:-2])%12+12)
      TimeAsNo = int("".join([HH,MM]))
      return TimeAsNo
  except Exception:
    return None
```

```python

#Pass into both udf's
df = df.withColumn("Issue_Date", dateParse("Issue_Date"))
df = df.withColumn("Violation_Time", timeParse("Violation_Time"))

#Assign the Class Label column as label
indexer = StringIndexer(inputCol="Violation_County", outputCol="Violation_County_ind")
df = indexer.fit(df).transform(df)

df = df.withColumn("label", df["Violation_County_ind"])
df.select(F.col("Violation_Code").cast('int').alias("Violation_Code"))
df.select(F.col("Street_Code1").cast('int').alias("Street_Code1"))
df.select(F.col("Street_Code2").cast('int').alias("Street_Code2"))
df.select(F.col("Street_Code3").cast('int').alias("Street_Code3"))
df.select(F.col("Vehicle_Year").cast('int').alias("Vehicle_Year"))
df.select(F.col("Feet_From_Curb").cast('float').alias("Feet_From_Curb"))

# Label Encoding of Classes and removing the original column
indexers = [StringIndexer(inputCol=column, outputCol=column+"_ind").setHandleInvalid(value="
    skip") for column in string_cols]
req_cols = [col+"_ind" for col in string_cols] + [col for col in req_cols if col not in
    string_cols]

imputer = Imputer()
imputer.setStrategy("mode")
imputer.setInputCols(req_cols)
imputer.setOutputCols([col+"_imp" for col in req_cols])
req_cols = [col+"_imp" for col in req_cols]

assembler1 = VectorAssembler(inputCols = req_cols, outputCol = "features_old")
scaler = StandardScaler(inputCol="features_old", outputCol="scaledFeatures",withStd=True,
    withMean=True)
assembler2 = VectorAssembler(inputCols = ["scaledFeatures"], outputCol = "features")

# Construct a new Logistic Regression object and fit the training data.
rf = RandomForestClassifier()
#pipeline
pipe = Pipeline(stages = indexers + [imputer,assembler1,scaler,assembler2, rf])

# Create a grid of multiple values of the hyper-parameter regParam
paramGrid = ParamGridBuilder().addGrid(rf.maxDepth,[20,30]).addGrid(rf.numTrees,[50,100]).
    build()

#Create a CrossValidator Object
obj = TrainValidationSplit(estimator=pipe,
                           estimatorParamMaps=paramGrid,
                           evaluator=MulticlassClassificationEvaluator(metricName = '
    accuracy'),
                           trainRatio=0.9,
                           seed = 2021)

#Train the model with the CrossValidator Object
trained_model = obj.fit(df)

#SAVING...
trained_model.bestModel.save("gs://bigdata_lab7/bestModel_RandomForest")

# Acquire and print the best model details from the CrossValidator object
print("\033[1mValidation Results\033[0m")
val = list(zip(trained_model.validationMetrics, trained_model.getEstimatorParamMaps()))
print(val)
```

Listing 1: RandomForestClassifier using Pyspark

5.2 Pyspark Code for LogisticRegression Model

```
#Importing Requirements
from __future__ import print_function
from pyspark.context import SparkContext
from pyspark.ml.linalg import Vectors
from pyspark.ml.classification import LogisticRegression
from pyspark.sql.session import SparkSession
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.sql.types import DoubleType
from pyspark.ml.feature import StringIndexer, Imputer
from pyspark.ml import Pipeline
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.feature import StandardScaler
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder, TrainValidationSplit
from pyspark.sql import functions as F

from pyspark.sql.types import IntegerType
from functools import reduce

sc = SparkContext()
spark = SparkSession(sc)

data_dir = "gs://bdl2021_final_project/nyc_tickets_train.csv"

df = spark.read.format("csv").option("recursiveFileLookup", "true").option("pathGlobFilter",
     "*.csv").load(data_dir,inferSchema=True, header = True)

drop_cols = ['No Standing or Stopping Violation',
             'Hydrant Violation', 'Double Parking Violation', 'Latitude',
             'Longitude', 'Community Board', 'Community Council', 'Census Tract',
             'BIN', 'BBL', 'NTA','Plate ID','Summons Number','Vehicle Expiration Date','Time
     First Observed','Date First Observed','From Hours In Effect','To Hours In Effect','
    Unregistered Vehicle?','Days Parking In Effect','House Number','Violation Legal Code','
    Issuer Code']

req_cols = ["Registration State", "Plate Type", "Issue Date",  "Violation Code", "Vehicle
    Body Type",  "Vehicle Make",  "Issuing Agency", "Street Code1",  "Street Code2", "Street
     Code3",
            "Issuer Command",  "Issuer Squad",  "Violation Time",  "Violation In Front Of Or
     Opposite", "Street Name",  "Intersecting Street",  "Law Section", "Sub Division", "
    Vehicle Color",
            "Vehicle Year", "Meter Number", "Feet From Curb", "Violation Post Code", "
    Violation Description"]

drop_cols = list(map(lambda x: "_".join(x.split()), drop_cols))
req_cols = list(map(lambda x: "_".join(x.split()), req_cols))

oldColumns = df.schema.names
df = reduce(lambda df, idx: df.withColumnRenamed(oldColumns[idx], "_".join(oldColumns[idx].
    split())), range(len(oldColumns)), df)

df.drop(*drop_cols)

string_cols = req_cols.copy()
string_cols.remove("Violation_Code")
string_cols.remove("Street_Code1")
string_cols.remove("Street_Code2")
string_cols.remove("Street_Code3")
string_cols.remove("Vehicle_Year")
string_cols.remove("Feet_From_Curb")

#Remove Extra two columns
string_cols.remove("Issue_Date")
string_cols.remove("Violation_Time")

#Defining two udfs
@F.udf(returnType=IntegerType())
def dateParse(dstr):
  BrStr = dstr.split("/")
  NewBrStr = [BrStr[2], BrStr[0], BrStr[1]]
  DateAsNo = int("".join(NewBrStr))
  return DateAsNo

@F.udf(returnType=IntegerType())
def timeParse(tstr):
  try:
    NoPart = tstr[:-1]
    dayLight = tstr[-1]
    if(dayLight == 'A'):
      return(int(NoPart))
    else:
      MM = NoPart[-2:]
      HH = str(int(NoPart[:-2])%12+12)
      TimeAsNo = int("".join([HH,MM]))
      return TimeAsNo
  except Exception:
    return None

#Pass into both udf's
df = df.withColumn("Issue_Date", dateParse("Issue_Date"))
```

```python
df = df.withColumn("Violation_Time", timeParse("Violation_Time"))

#Assign the Class Label column as label
indexer = StringIndexer(inputCol="Violation_County", outputCol="Violation_County_ind")
df = indexer.fit(df).transform(df)

df = df.withColumn("label", df["Violation_County_ind"])
df.select(F.col("Violation_Code").cast('int').alias("Violation_Code"))
df.select(F.col("Street_Code1").cast('int').alias("Street_Code1"))
df.select(F.col("Street_Code2").cast('int').alias("Street_Code2"))
df.select(F.col("Street_Code3").cast('int').alias("Street_Code3"))
df.select(F.col("Vehicle_Year").cast('int').alias("Vehicle_Year"))
df.select(F.col("Feet_From_Curb").cast('float').alias("Feet_From_Curb"))

# Label Encoding of Classes and removing the original column
indexers = [StringIndexer(inputCol=column, outputCol=column+"_ind").setHandleInvalid(value="
    skip") for column in string_cols]
req_cols = [col+"_ind" for col in string_cols] + [col for col in req_cols if col not in
    string_cols]

imputer = Imputer()
imputer.setStrategy("mode")
imputer.setInputCols(req_cols)
imputer.setOutputCols([col+"_imp" for col in req_cols])
req_cols = [col+"_imp" for col in req_cols]

assembler1 = VectorAssembler(inputCols = req_cols, outputCol = "features_old")
scaler = StandardScaler(inputCol="features_old", outputCol="scaledFeatures",withStd=True,
    withMean=True)
assembler2 = VectorAssembler(inputCols = ["scaledFeatures"], outputCol = "features")

# Construct a new Logistic Regression object and fit the training data.
lr = LogisticRegression(maxIter=25)
#pipeline
pipe = Pipeline(stages = indexers + [imputer,assembler1,scaler,assembler2, lr])

# Create a grid of multiple values of the hyper-parameter regParam
paramGrid = ParamGridBuilder().addGrid(lr.regParam,[0,0.5,1]).addGrid(lr.elasticNetParam
    ,[0,0.5]).build()

#Create a CrossValidator Object
obj = TrainValidationSplit(estimator=pipe,
                           estimatorParamMaps=paramGrid,
                           evaluator=MulticlassClassificationEvaluator(metricName = '
    accuracy'),
                           trainRatio=0.9,
                           seed = 2021)

#Train the model with the CrossValidator Object
trained_model = obj.fit(df)

#SAVING...
trained_model.bestModel.save("gs://bigdata_lab7/bestModel_LogisticRegression_FinTRY")

# Acquire and print the best model details from the CrossValidator object
print("\033[1mValidation Results\033[0m")
val = list(zip(trained_model.validationMetrics, trained_model.getEstimatorParamMaps()))
print(val)
```

Listing 2: LogisticRegression using Pyspark

5.3 Pyspark Code for NaiveBayes Model

```python
#Importing Requirements
from __future__ import print_function
from pyspark.context import SparkContext
from pyspark.ml.linalg import Vectors
from pyspark.ml.classification import NaiveBayes
from pyspark.sql.session import SparkSession
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.sql.types import DoubleType
from pyspark.ml.feature import StringIndexer, Imputer
from pyspark.ml import Pipeline
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.feature import StandardScaler
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder, TrainValidationSplit
from pyspark.sql import functions as F

from pyspark.sql.types import IntegerType
from functools import reduce

sc = SparkContext()
spark = SparkSession(sc)

data_dir = "gs://bdl2021_final_project/nyc_tickets_train.csv"

df = spark.read.format("csv").option("recursiveFileLookup", "true").option("pathGlobFilter", "*.csv").load(data_dir,inferSchema=True, header = True)

drop_cols = ['No Standing or Stopping Violation',
             'Hydrant Violation', 'Double Parking Violation', 'Latitude',
             'Longitude', 'Community Board', 'Community Council', 'Census Tract',
             'BIN', 'BBL', 'NTA','Plate ID','Summons Number','Vehicle Expiration Date','Time First Observed','Date First Observed','From Hours In Effect','To Hours In Effect','Unregistered Vehicle?','Days Parking In Effect','House Number','Violation Legal Code','Issuer Code']

req_cols = ["Registration State", "Plate Type", "Issue Date",  "Violation Code", "Vehicle Body Type",  "Vehicle Make",  "Issuing Agency", "Street Code1",  "Street Code2", "Street Code3",
            "Issuer Command",  "Issuer Squad",  "Violation Time",  "Violation In Front Of Or Opposite", "Street Name",  "Intersecting Street",  "Law Section", "Sub Division", "Vehicle Color",
            "Vehicle Year", "Meter Number", "Feet From Curb", "Violation Post Code", "Violation Description"]

drop_cols = list(map(lambda x: "_".join(x.split()), drop_cols))
req_cols = list(map(lambda x: "_".join(x.split()), req_cols))

oldColumns = df.schema.names
df = reduce(lambda df, idx: df.withColumnRenamed(oldColumns[idx], "_".join(oldColumns[idx].split())), range(len(oldColumns)), df)

df.drop(*drop_cols)

string_cols = req_cols.copy()
string_cols.remove("Violation_Code")
string_cols.remove("Street_Code1")
string_cols.remove("Street_Code2")
string_cols.remove("Street_Code3")
string_cols.remove("Vehicle_Year")
string_cols.remove("Feet_From_Curb")

#Remove Extra two columns
string_cols.remove("Issue_Date")
string_cols.remove("Violation_Time")

#Defining two udfs
@F.udf(returnType=IntegerType())
def dateParse(dstr):
  BrStr = dstr.split("/")
  NewBrStr = [BrStr[2], BrStr[0], BrStr[1]]
  DateAsNo = int("".join(NewBrStr))
  return DateAsNo

@F.udf(returnType=IntegerType())
def timeParse(tstr):
  try:
    NoPart = tstr[:-1]
    dayLight = tstr[-1]
    if(dayLight == 'A'):
      return(int(NoPart))
    else:
      MM = NoPart[-2:]
      HH = str(int(NoPart[:-2])%12+12)
      TimeAsNo = int("".join([HH,MM]))
      return TimeAsNo
  except Exception:
    return None

#Pass into both udf's
df = df.withColumn("Issue_Date", dateParse("Issue_Date"))
```

```python
df = df.withColumn("Violation_Time", timeParse("Violation_Time"))

#Assign the Class Label column as label
indexer = StringIndexer(inputCol="Violation_County", outputCol="Violation_County_ind")
df = indexer.fit(df).transform(df)

df = df.withColumn("label", df["Violation_County_ind"])
df.select(F.col("Violation_Code").cast('int').alias("Violation Code"))
df.select(F.col("Street_Code1").cast('int').alias("Street_Code1"))
df.select(F.col("Street_Code2").cast('int').alias("Street_Code2"))
df.select(F.col("Street_Code3").cast('int').alias("Street_Code3"))
df.select(F.col("Vehicle_Year").cast('int').alias("Vehicle_Year"))
df.select(F.col("Feet_From_Curb").cast('float').alias("Feet_From_Curb"))

# Label Encoding of Classes and removing the original column
indexers = [StringIndexer(inputCol=column, outputCol=column+"_ind").setHandleInvalid(value="
    skip") for column in string_cols]
req_cols = [col+"_ind" for col in string_cols] + [col for col in req_cols if col not in
    string_cols]

imputer = Imputer()
imputer.setStrategy("mode")
imputer.setInputCols(req_cols)
imputer.setOutputCols([col+"_imp" for col in req_cols])
req_cols = [col+"_imp" for col in req_cols]

assembler1 = VectorAssembler(inputCols = req_cols, outputCol = "features_old")
scaler = StandardScaler(inputCol="features_old", outputCol="scaledFeatures",withStd=True,
    withMean=True)
assembler2 = VectorAssembler(inputCols = ["scaledFeatures"], outputCol = "features")

# Construct a new NaiveBayes object and fit the training data.
nb = NaiveBayes(modelType="gaussian")
#pipeline
pipe = Pipeline(stages = indexers + [imputer,assembler1,scaler,assembler2, nb])

# Create a grid of multiple values of the hyper-parameter regParam
paramGrid = ParamGridBuilder().addGrid(nb.smoothing,[0.1,0.5,1.0]).build()

#Create a CrossValidator Object
obj = TrainValidationSplit(estimator=pipe,
                           estimatorParamMaps=paramGrid,
                           evaluator=MulticlassClassificationEvaluator(metricName = '
    accuracy'),
                           trainRatio=0.9,
                           seed = 2021)

#Train the model with the CrossValidator Object
trained_model = obj.fit(df)

#SAVING...
trained_model.bestModel.save("gs://bigdata_lab7/bestModel_NaiveBayes")

# Acquire and print the best model details from the CrossValidator object
print("\033[1mValidation Results\033[0m")
val = list(zip(trained_model.validationMetrics, trained_model.getEstimatorParamMaps()))
print(val)
```

Listing 3: NaiveBayes using Pyspark

5.4 Code for Kafka Producer

```python
from kafka import KafkaProducer
from google.cloud import storage

#kafka details
IP_ = "10.128.0.44:9092"
topic_name = 'PROJECT'
producer = KafkaProducer(bootstrap_servers = [IP_])

#download data for gcs
client = storage.Client()
bucket = client.get_bucket("bdl2021_final_project")
blobs_all = list(bucket.list_blobs(prefix="nyc_tickets_train.csv/"))


i = 0
for blob in blobs_all[2:]:      #ignore the first 2 irrelevant files

    #download data from GCS
    content = blob.download_as_string()
    content = content.decode('utf-8')

    #split and iterate over each line
    lines = content.split('\n')
    for line in lines[1:]:                  #ignore title line
        if line != '':
            line = bytes(line,'utf-8')
            producer.send(topic_name,line)
            producer.flush()

    print("{} lines from part no: {} written to Kafka".format(len(lines),i))
    i += 1
```

Listing 4: Kafka Producer

5.5 Code for Kafka Subscriber

```python
#Importing Requirements
from __future__ import print_function
from pyspark.context import SparkContext
from pyspark.ml.linalg import Vectors
from pyspark.ml.classification import RandomForestClassifier
from pyspark.sql.session import SparkSession
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.sql.types import DoubleType
from pyspark.ml.feature import StringIndexer, StringIndexerModel, Imputer
from pyspark.ml import Pipeline, PipelineModel
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.feature import StandardScaler
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.sql import functions as F

from pyspark.sql.types import IntegerType
from functools import reduce


spark = SparkSession.builder.appName("Poject_NYC_tickets").getOrCreate()
spark.sparkContext.setLogLevel('WARN')

#address of Kafka server
IP_ = "10.128.0.48:9092"

#read from kafka
df = spark.readStream \
     .format("kafka") \
     .option("kafka.bootstrap.servers",IP_) \
     .option("subscribe","PROJECT") \
     .load()


#All features
features = ['Summons Number','Plate ID','Registration State','Plate Type','Issue Date','
    Violation Code','Vehicle Body Type','Vehicle Make',
            'Issuing Agency','Street Code1','Street Code2','Street Code3','Vehicle
    Expiration Date','Issuer Code','Issuer Command','Issuer Squad',
            'Violation Time','Time First Observed','Violation_County','Violation In Front Of
     Or Opposite','House Number','Street Name','Intersecting Street',
            'Date First Observed','Law Section','Sub Division','Violation Legal Code','Days
    Parking In Effect','From Hours In Effect','To Hours In Effect',
            'Vehicle Color','Unregistered Vehicle?','Vehicle Year','Meter Number','Feet From
     Curb','Violation Post Code','Violation Description',
            'No Standing or Stopping Violation','Hydrant Violation','Double Parking
    Violation','Latitude','Longitude','Community Board','Community Council',
            'Census Tract','BIN','BBL','NTA']

features = list(map(lambda x: "_".join(x.split()),features))

#add feature columns to DataFrame
columns = F.split(df.value,',')
for i in range(len(features)):
    df = df.withColumn(features[i],columns[i].cast('string'))


#Defining two udfs
@F.udf(returnType=IntegerType())
def dateParse(dstr):
  BrStr = dstr.split("/")
  NewBrStr = [BrStr[2], BrStr[0], BrStr[1]]
  DateAsNo = int("".join(NewBrStr))
  return DateAsNo

@F.udf(returnType=IntegerType())
def timeParse(tstr):
  try:
    NoPart = tstr[:-1]
    dayLight = tstr[-1]
    if(dayLight == 'A'):
      return(int(NoPart))
    else:
      MM = NoPart[-2:]
      HH = str(int(NoPart[:-2])%12+12)
      TimeAsNo = int("".join([HH,MM]))
      return TimeAsNo
  except Exception:
    return None


#Pass into both udf's
df = df.withColumn("Issue_Date", dateParse("Issue_Date"))
df = df.withColumn("Violation_Time", timeParse("Violation_Time"))

#cast columns to appropriate dtype
df =  df.withColumn("Violation_Code", F.col("Violation_Code").cast('int'))
df =  df.withColumn("Street_Code1", F.col("Street_Code1").cast('int'))
df =  df.withColumn("Street_Code2", F.col("Street_Code2").cast('int'))
df =  df.withColumn("Street_Code3", F.col("Street_Code3").cast('int'))
```

```python
df =  df.withColumn("Vehicle_Year", F.col("Vehicle_Year").cast('int'))
df =  df.withColumn("Feet_From_Curb", F.col("Feet_From_Curb").cast('float'))


#load label indexer and transform df
indexer = StringIndexerModel.load('gs://avinashbagali/Label_Indexer/')
df = indexer.transform(df)
df = df.withColumn("label", df["Violation_County_ind"])


#load best model and transform df
best_model = PipelineModel.load("gs://avinashbagali/bestModel_RandomForest/")
df = best_model.transform(df)

#result df -> ["Violation_County","label","prediction"]
result_df = df[["Violation_County","label","prediction"]]


# function to use with foreachBatch to compute accuracy and F1-score
def Metrics(df, epoch_id):
    evaluator_acc = MulticlassClassificationEvaluator(predictionCol = "prediction",
                                                       labelCol="label",
                                                       metricName="accuracy")

    evaluator_f1 = MulticlassClassificationEvaluator(predictionCol = "prediction",
                                                      labelCol="label",
                                                      metricName="f1")

    acc = evaluator_acc.evaluate(df)
    f1  = evaluator_f1.evaluate(df)


    print("+-------------------------------+")
    print("|Number of rows in batch {} = {}|".format(epoch_id,df.count()))
    print("|  Accuracy on batch {} is {}   |".format(epoch_id,round(acc,3)))
    print("|  F1 score on batch {} is {}   |".format(epoch_id,round( f1,3)))
    print("+-------------------------------+")


#print result_df -> ["Violation_County","label","prediction"]
query1 = result_df \
       .writeStream \
       .outputMode("append") \
       .format("console") \
       .start()


#print Accuracy and F1-score
query2 = result_df \
       .writeStream \
       .format("console") \
       .foreachBatch(Metrics) \
       .start()


query1.awaitTermination()
query2.awaitTermination()
```
Listing 5: Kafka Subscriber