CS4830 – Big Data Laboratory

Jan – May 2021

Project Report

ME17B016 – Joe Bobby

ME17B158 – Omkar Nath

ME17B170 – Uma T V

Contents

Objective

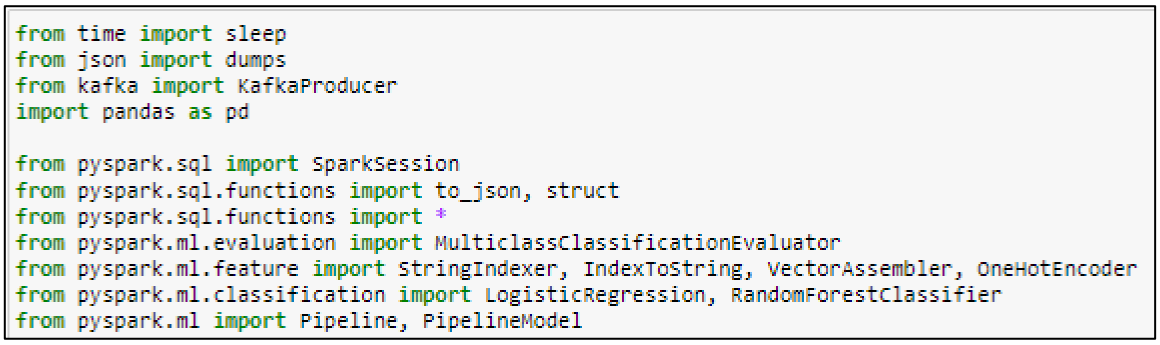
The objective of the project is to use the dataset containing the information about the various tickets issued in New York to predict the part in the city where the ticket was issued.

Pre-processing

The pre-processing and feature engineering is performed in a DataProc Cluster using Spark.

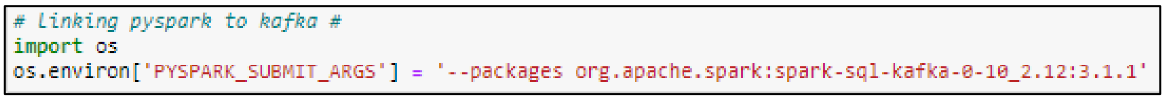
Steps involved in the pre-processing are as follows:

- The required libraries are imported.



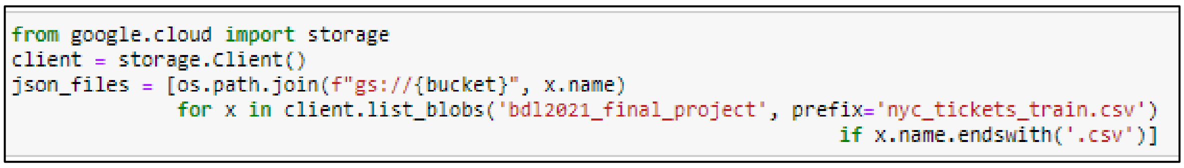
```
from time import sleep
from json import dumps
from kafka import KafkaProducer
import pandas as pd

from pyspark.sql import SparkSession
from pyspark.sql.functions import to_json, struct
from pyspark.sql.functions import *
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.feature import StringIndexer, IndexToString, VectorAssembler, OneHotEncoder
from pyspark.ml.classification import LogisticRegression, RandomForestClassifier
from pyspark.ml import Pipeline, PipelineModel
```

- The PySpark argument is submitted in the OS environment with the spark Kafka version set to be 3.1.1.

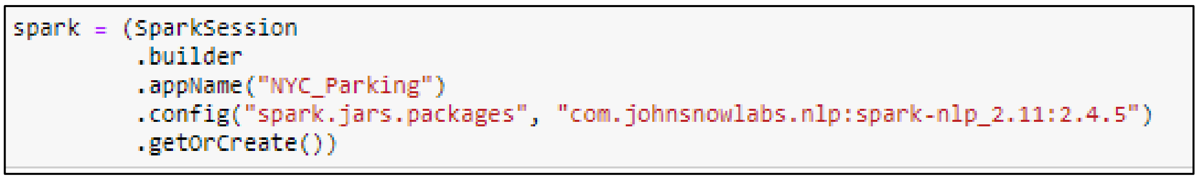


```
# Linking pyspark to kafka #
import os
os.environ['PYSPARK_SUBMIT_ARGS'] = '--packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.1.1'
```

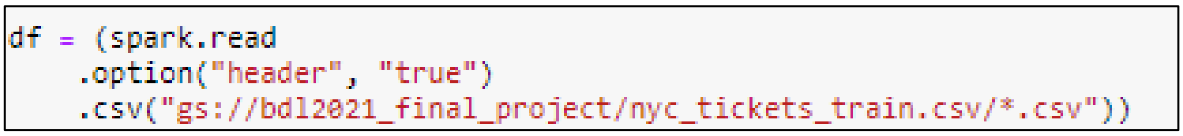
- A spark session is initialized and the file is read.



```
from google.cloud import storage
client = storage.Client()
json_files = [os.path.join(f"gs://{bucket}", x.name)
              for x in client.list_blobs('bdl2021_final_project', prefix='nyc_tickets_train.csv')
                                                      if x.name.endswith('.csv')]
```

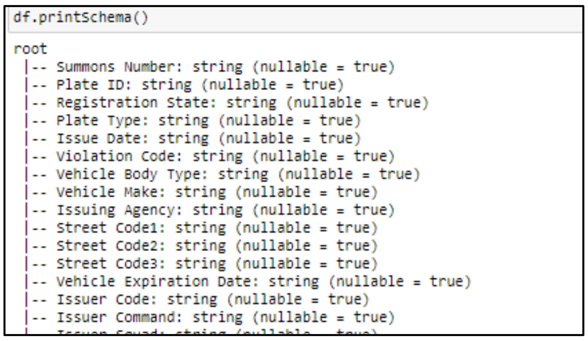


```
spark = (SparkSession
         .builder
         .appName("NYC_Parking")
         .config("spark.jars.packages", "com.johnsnowlabs.nlp:spark-nlp_2.11:2.4.5")
         .getOrCreate())
```

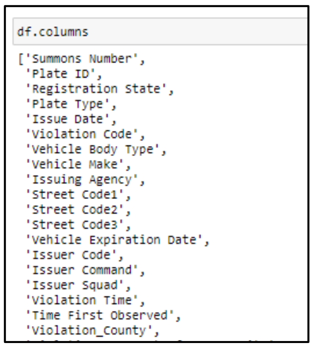


```
df = (spark.read
     .option("header", "true")
     .csv("gs://bdl2021_final_project/nyc_tickets_train.csv/*.csv"))
```

- The features and their types are visualized using schema.



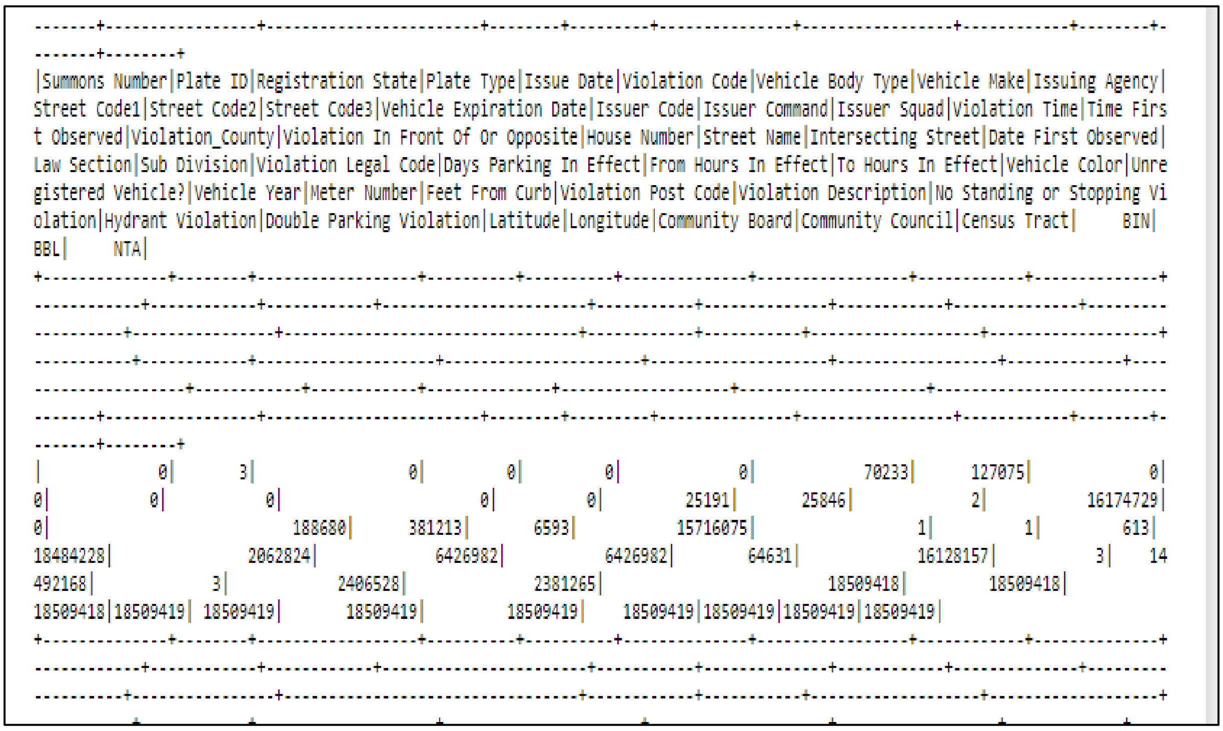
```
df.printSchema()

root
 |-- Summons Number: string (nullable = true)
 |-- Plate ID: string (nullable = true)
 |-- Registration State: string (nullable = true)
 |-- Plate Type: string (nullable = true)
 |-- Issue Date: string (nullable = true)
 |-- Violation Code: string (nullable = true)
 |-- Vehicle Body Type: string (nullable = true)
 |-- Vehicle Make: string (nullable = true)
 |-- Issuing Agency: string (nullable = true)
 |-- Street Code1: string (nullable = true)
 |-- Street Code2: string (nullable = true)
 |-- Street Code3: string (nullable = true)
 |-- Vehicle Expiration Date: string (nullable = true)
 |-- Issuer Code: string (nullable = true)
 |-- Issuer Command: string (nullable = true)
```



```
df.columns

['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',
```

(only a part of the output is displayed as example)

- An aggregated Dataframe is generated that shows the nan values in each column



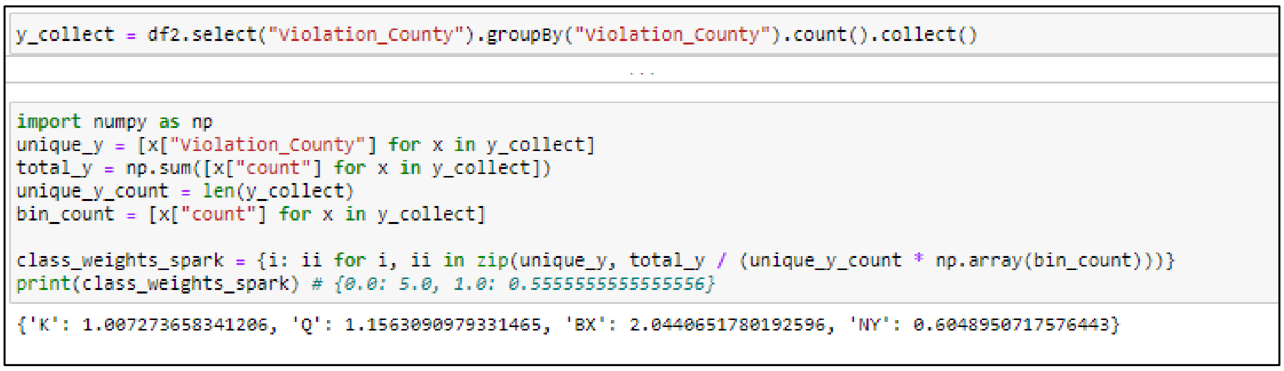
```
-------+----------------+----------------------+--------+---------+--------------+----------------+-----------+--------+-
-------+--------+
|Summons Number|Plate ID|Registration State|Plate Type|Issue Date|Violation Code|Vehicle Body Type|Vehicle Make|Issuing Agency|
Street Code1|Street Code2|Street Code3|Vehicle Expiration Date|Issuer Code|Issuer Command|Issuer Squad|Violation Time|Time Firs
t 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|Unre
gistered Vehicle?|Vehicle Year|Meter Number|Feet From Curb|Violation Post Code|Violation Description|No Standing or Stopping Vi
olation|Hydrant Violation|Double Parking Violation|Latitude|Longitude|Community Board|Community Council|Census Tract|     BIN|
BBL|     NTA|
+-------------+--------+------------------+----------+----------+--------------+-----------------+------------+-------------+
-----------+-----------+-----------+-----------------------+-----------+--------------+------------+--------------+--------
---------+---------------+--------------------------------+-----------+-----------+-------------------+------------------+
----------+-----------+-------------------+--------------------+--------------------+------------------+------------+----
----------------+-----------+-----------+-------------+-------------------+---------------------+------------------------
-------+----------------+----------------------+--------+---------+--------------+----------------+-----------+--------+-
-------+--------+
|             0|       3|                 0|         0|         0|             0|            70233|      127075|             0|
0|           0|           0|                      0|          0|         25191|       25846|             2|        16174729|
0|                        188680|      381213|       6593|           15716075|                 1|            1|          613|
18484228|              2062824|             6426982|             6426982|        64631|             16128157|           3|    14
492168|               3|            2406528|              2381265|                           18509418|           18509418|
18509418|18509419| 18509419|        18509419|           18509419|     18509419|18509419|18509419|18509419|
+-------------+--------+------------------+----------+----------+--------------+-----------------+------------+-------------+
-----------+-----------+-----------+-----------------------+-----------+--------------+------------+--------------+--------
---------+---------------+--------------------------------+-----------+-----------+-------------------+------------------+
```

(because of the large size, it isn't shown properly)

Based on a one-to-one mapping, we can say that the last few are almost completely nan and there are some intermediate large nan values.

- Anything above 40% nan was dropped, for more accuracy, faster computation and to decrease the load on all the tasks.

- Next, the labels of Violation County are counted to check for class imbalances which may affect predictions. (Later on it was found that there was indeed a class imbalance and it did affect the predictions badly)



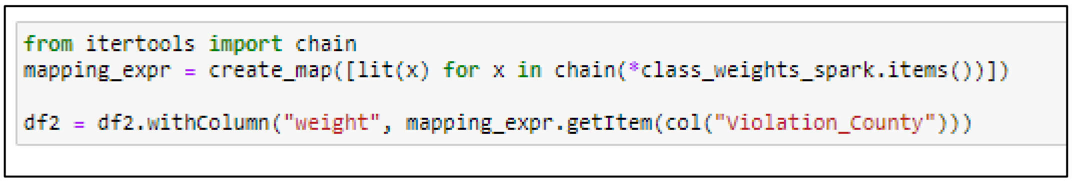
```
y_collect = df2.select("Violation_County").groupBy("Violation_County").count().collect()
```

```
import numpy as np
unique_y = [x["Violation_County"] for x in y_collect]
total_y = np.sum([x["count"] for x in y_collect])
unique_y_count = len(y_collect)
bin_count = [x["count"] for x in y_collect]

class_weights_spark = {i: ii for i, ii in zip(unique_y, total_y / (unique_y_count * np.array(bin_count)))}
print(class_weights_spark) # {0.0: 5.0, 1.0: 0.5555555555555556}
```

```
{'K': 1.007273658341206, 'Q': 1.1563090979331465, 'BX': 2.0440651780192596, 'NY': 0.6048950717576443}
```

"y_collect" is the count aggregation of each of the Violation County. The y count values are used to calculate the class weights to handle class imbalances.

- Then, a column of weights is added by running the mapping expression on the column.

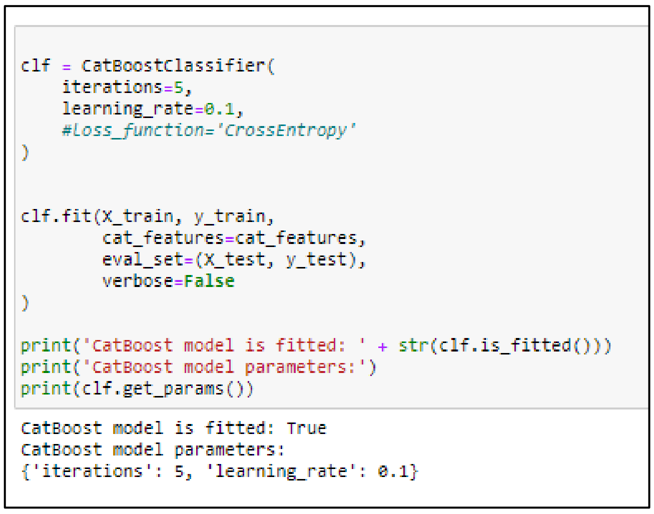


```
from itertools import chain
mapping_expr = create_map([lit(x) for x in chain(*class_weights_spark.items())])

df2 = df2.withColumn("weight", mapping_expr.getItem(col("Violation_County")))
```

Feature Selection

To assess the importance of the features, a section of the data provided for model training was used to fit an assessment model (different from the final model). We use this model to figure out the important features. We then use these selected features to train the Logistic Regression model in the Spark Data Pipeline.

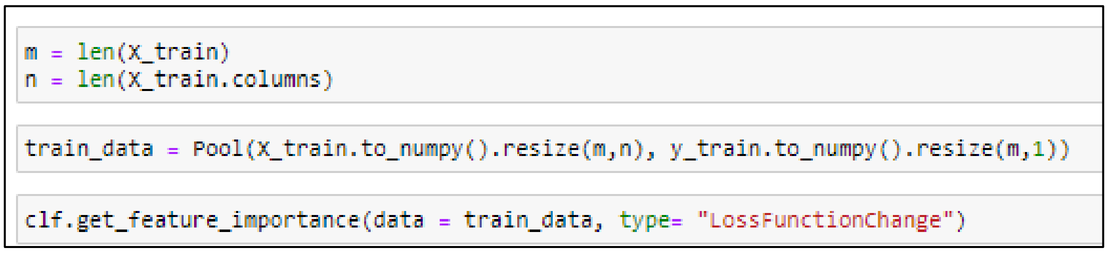
- The columns with positive importance obtained using CatBoost was used to predict values. This process is described as follows:
  - First of all, the CatBoost Classifier is used to fit the training data



```
clf = CatBoostClassifier(
    iterations=5,
    learning_rate=0.1,
    #loss_function='CrossEntropy'
)


clf.fit(X_train, y_train,
        cat_features=cat_features,
        eval_set=(X_test, y_test),
        verbose=False
)

print('CatBoost model is fitted: ' + str(clf.is_fitted()))
print('CatBoost model parameters:')
print(clf.get_params())
```

```
CatBoost model is fitted: True
CatBoost model parameters:
{'iterations': 5, 'learning_rate': 0.1}
```

- The CatBoost library is used to get the feature importance values of the various categorical features

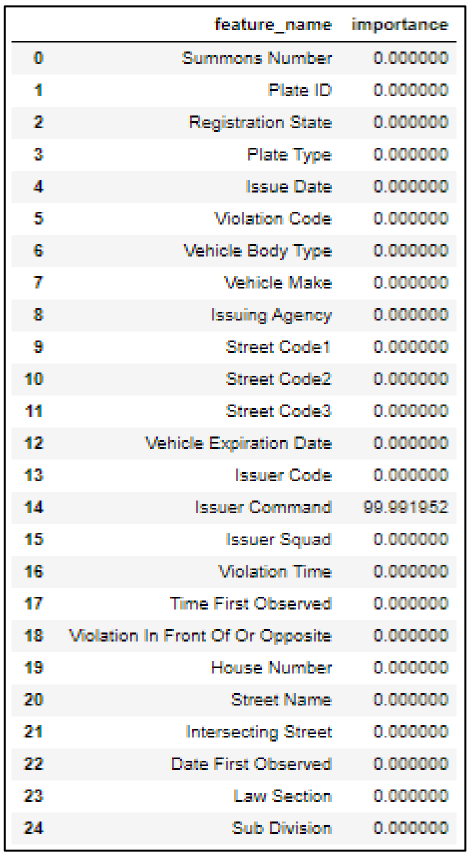


```
m = len(X_train)
n = len(X_train.columns)
```

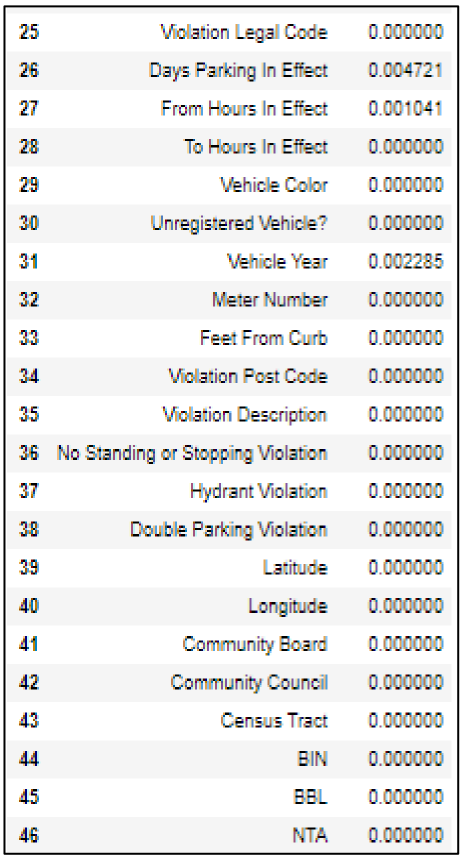
```
train_data = Pool(X_train.to_numpy().resize(m,n), y_train.to_numpy().resize(m,1))
```

```
clf.get_feature_importance(data = train_data, type= "LossFunctionChange")
```

- The features with that gave a positive value of importance were chosen for analysis.



	feature_name	importance
0	Summons Number	0.000000
1	Plate ID	0.000000
2	Registration State	0.000000
3	Plate Type	0.000000
4	Issue Date	0.000000
5	Violation Code	0.000000
6	Vehicle Body Type	0.000000
7	Vehicle Make	0.000000
8	Issuing Agency	0.000000
9	Street Code1	0.000000
10	Street Code2	0.000000
11	Street Code3	0.000000
12	Vehicle Expiration Date	0.000000
13	Issuer Code	0.000000
14	Issuer Command	99.991952
15	Issuer Squad	0.000000
16	Violation Time	0.000000
17	Time First Observed	0.000000
18	Violation In Front Of Or Opposite	0.000000
19	House Number	0.000000
20	Street Name	0.000000
21	Intersecting Street	0.000000
22	Date First Observed	0.000000
23	Law Section	0.000000
24	Sub Division	0.000000



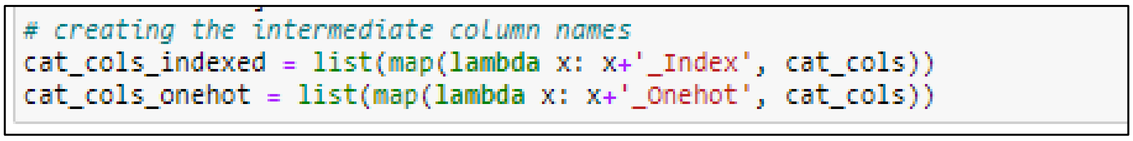
25	Violation Legal Code	0.000000
26	Days Parking In Effect	0.004721
27	From Hours In Effect	0.001041
28	To Hours In Effect	0.000000
29	Vehicle Color	0.000000
30	Unregistered Vehicle?	0.000000
31	Vehicle Year	0.002285
32	Meter Number	0.000000
33	Feet From Curb	0.000000
34	Violation Post Code	0.000000
35	Violation Description	0.000000
36	No Standing or Stopping Violation	0.000000
37	Hydrant Violation	0.000000
38	Double Parking Violation	0.000000
39	Latitude	0.000000
40	Longitude	0.000000
41	Community Board	0.000000
42	Community Council	0.000000
43	Census Tract	0.000000
44	BIN	0.000000
45	BBL	0.000000
46	NTA	0.000000

These dataframes denote the feature importances of the features

- Hence, we have now obtained the important features among all the features given to us and they are:
  [“Issuer Command” , “Days Parking in Effect” , “From Hours in Effect” , “Vehical Year”]

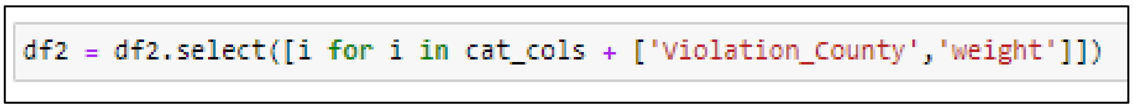
Model Pipeline

- Intermediate column names are then created for indexing and one hot encoding the categorical features



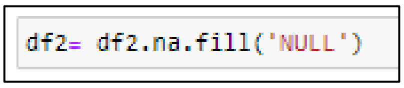
```
# creating the intermediate column names
cat_cols_indexed = list(map(lambda x: x+'_Index', cat_cols))
cat_cols_onehot = list(map(lambda x: x+'_Onehot', cat_cols))
```

- The relevant categorial columns, the Violation County column and the class weights are extracted into the Dataframe.



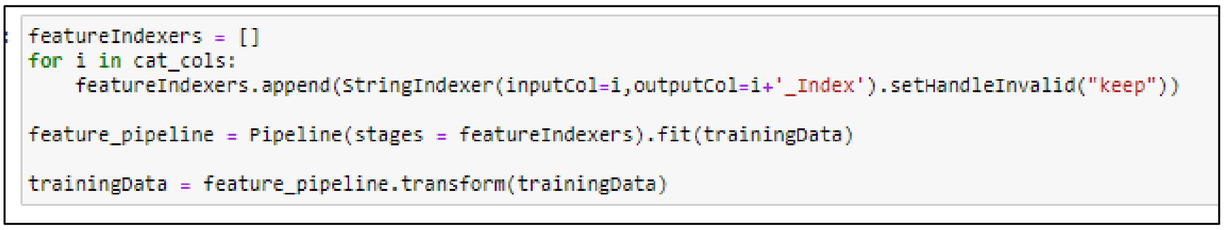
```
df2 = df2.select([i for i in cat_cols + ['Violation_County','weight']])
```

- All the NA values in the Dataframe are replaced by a 'NULL' string for classification. (done in producer as well)



```
df2= df2.na.fill('NULL')
```

- The dataset is then split into train and test sets.
- A pipeline is then created

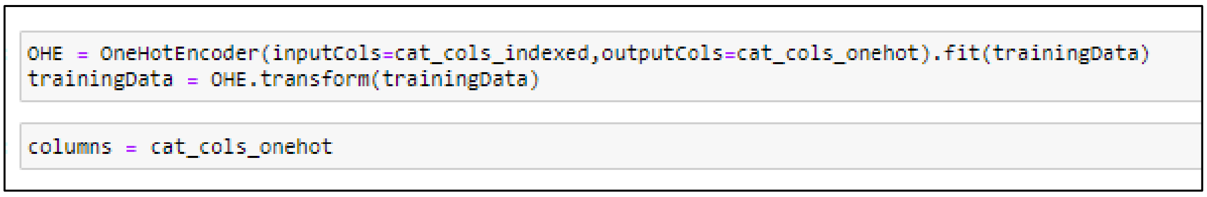


```
featureIndexers = []
for i in cat_cols:
    featureIndexers.append(StringIndexer(inputCol=i,outputCol=i+'_Index').setHandleInvalid("keep"))

feature_pipeline = Pipeline(stages = featureIndexers).fit(trainingData)

trainingData = feature_pipeline.transform(trainingData)
```

Each append is basically the input column as the categorical feature column and the output column as that that column plus the _Index suffix. After applying the transform the training data will have same columns as earlier but with the _Index suffix

- The training data will be transformed multiple times, so we keep an original untransformed copy for use.

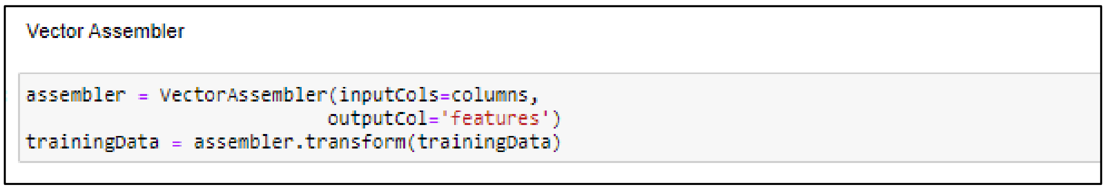
- Then one hot encoding is performed on the training data.



```
OHE = OneHotEncoder(inputCols=cat_cols_indexed,outputCols=cat_cols_onehot).fit(trainingData)
trainingData = OHE.transform(trainingData)
```

```
columns = cat_cols_onehot
```

- Then, the vector assembler is used to convert the one hot encoded training data to one feature vector. Transformations are performed to directly train the model without training the model using a pipeline, because training the model with pipeline takes a lot of time.

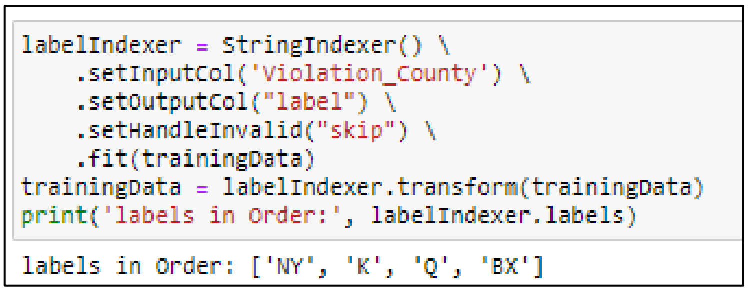


Vector Assembler

```
assembler = VectorAssembler(inputCols=columns,
                            outputCol='features')
trainingData = assembler.transform(trainingData)
```

- There only exist only transforms in the pipeline, and no estimators or fit algorithms.

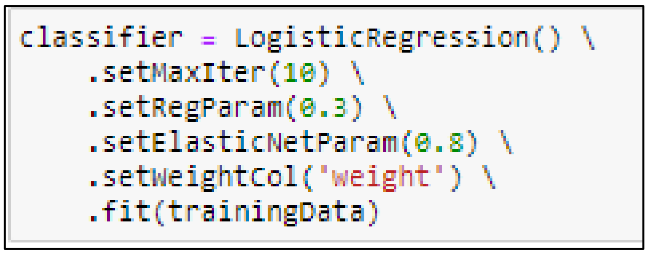
- When we predict something, in order to view the class after prediction instead of the label, the labelIndexer is used. Labels can then be viewed in order.



```
labelIndexer = StringIndexer() \
    .setInputCol('Violation_County') \
    .setOutputCol("label") \
    .setHandleInvalid("skip") \
    .fit(trainingData)
trainingData = labelIndexer.transform(trainingData)
print('labels in Order:', labelIndexer.labels)
```

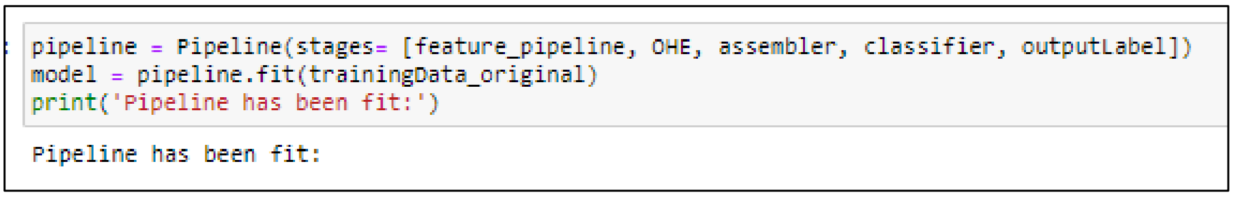
```
labels in Order: ['NY', 'K', 'Q', 'BX']
```

- We then use Logistic Regression algorithm to make predictions.



```
classifier = LogisticRegression() \
    .setMaxIter(10) \
    .setRegParam(0.3) \
    .setElasticNetParam(0.8) \
    .setWeightCol('weight') \
    .fit(trainingData)
```

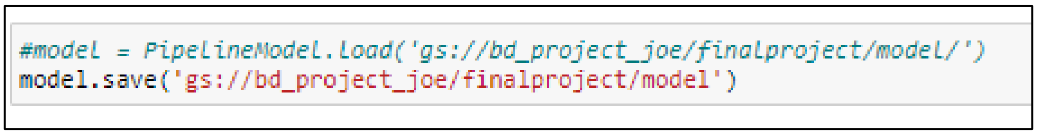
- The four stages of our model are embedded into the pipeline.



```
pipeline = Pipeline(stages= [feature_pipeline, OHE, assembler, classifier, outputLabel])
model = pipeline.fit(trainingData_original)
print('Pipeline has been fit:')
```

```
Pipeline has been fit:
```

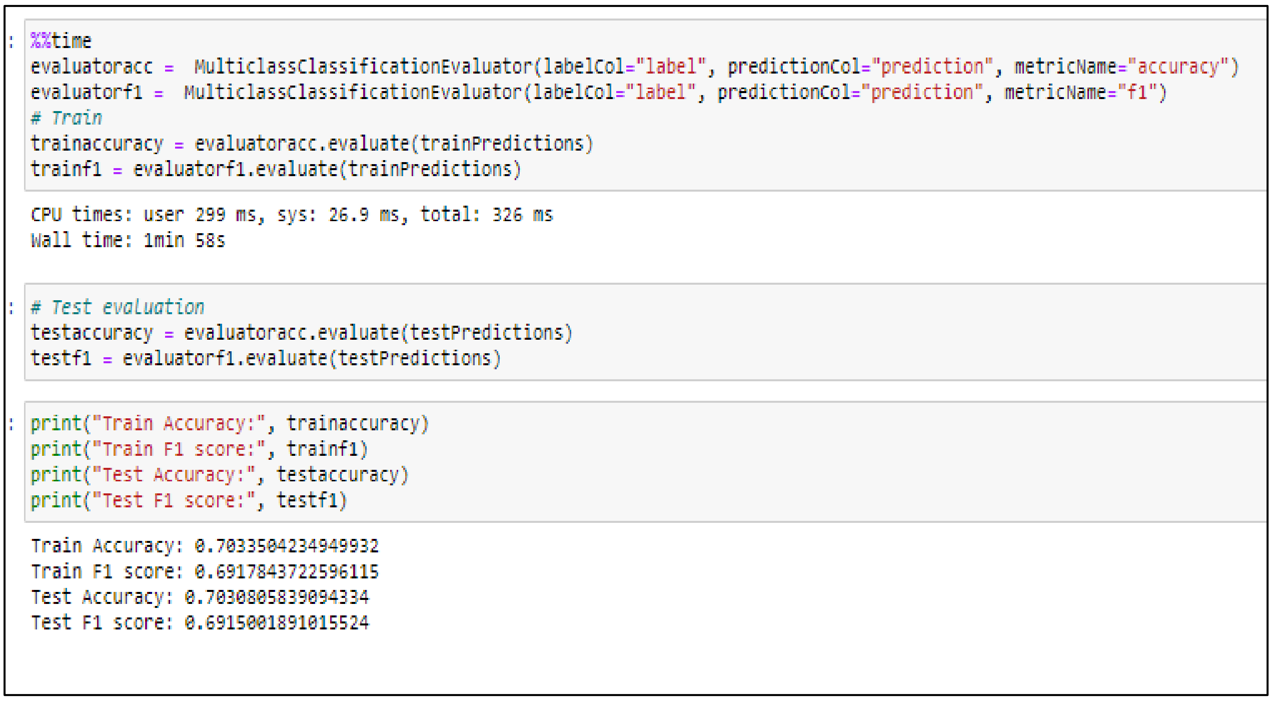
- The existent model has to be removed from the GCP bucket, because if the model is already written the it can't be overwritten. Then, the model is saved in the GCP bucket.



```
#model = PipelineModel.load('gs://bd_project_joe/finalproject/model/')
model.save('gs://bd_project_joe/finalproject/model')
```

Model Evaluation

- Evaluation: We use MulticlassClassificationEvaluator to get the training accuracy and the f1 score for the training data



```
%%time
evaluatoracc =  MulticlassClassificationEvaluator(labelCol="label", predictionCol="prediction", metricName="accuracy")
evaluatorf1 =  MulticlassClassificationEvaluator(labelCol="label", predictionCol="prediction", metricName="f1")
# Train
trainaccuracy = evaluatoracc.evaluate(trainPredictions)
trainf1 = evaluatorf1.evaluate(trainPredictions)
```

```
CPU times: user 299 ms, sys: 26.9 ms, total: 326 ms
Wall time: 1min 58s
```

```
# Test evaluation
testaccuracy = evaluatoracc.evaluate(testPredictions)
testf1 = evaluatorf1.evaluate(testPredictions)
```

```
print("Train Accuracy:", trainaccuracy)
print("Train F1 score:", trainf1)
print("Test Accuracy:", testaccuracy)
print("Test F1 score:", testf1)
```

```
Train Accuracy: 0.7033504234949932
Train F1 score: 0.6917843722596115
Test Accuracy: 0.7030805839094334
Test F1 score: 0.6915001891015524
```

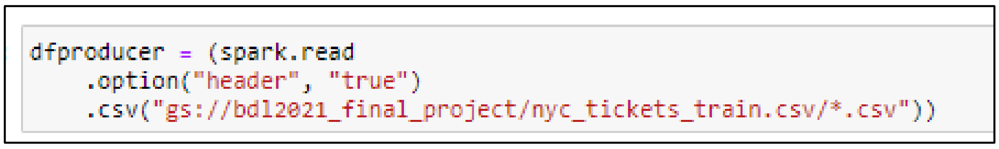
Real-time Computation: Spark Streaming

Kafka Producer – Streaming Source

A DataProc Cluster is used to submit a Spark job for data pre-processing and model training. The model is saved in the GCS Bucket. Code has been written to access this model and make predictions for real-time evaluation. The trained model is used to perform real-time predictions of test data streaming to Kafka. Data is fed into Kafka cluster, and then used in a Spark cluster to generate real time predictions. The accuracies and F1 scores are then printed for every batch.

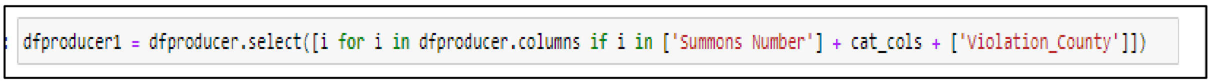
Test Data Predictions: The test data stored on the GCS bucket is streamed into Kafka. Spark Streaming is used to read the data and make real-time predictions using the stored model.

- The test Dataframe is read all over again, to be streamed, processed and predicted.



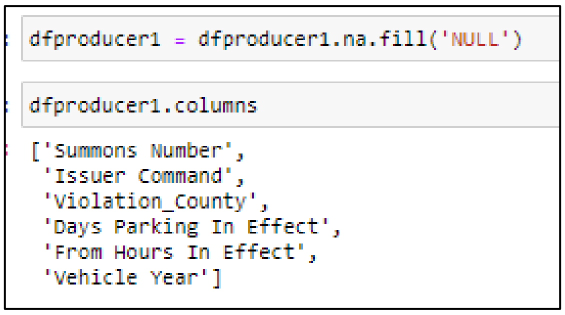
```
dfproducer = (spark.read
    .option("header", "true")
    .csv("gs://bdl2021_final_project/nyc_tickets_train.csv/*.csv"))
```

- The Summons number is included as a primary key for identification of the Violation cases.



```
dfproducer1 = dfproducer.select([i for i in dfproducer.columns if i in ['Summons Number'] + cat_cols + ['Violation_County']])
```

- The NA values in the data are replaced by NULL.



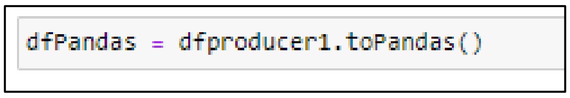
```
dfproducer1 = dfproducer1.na.fill('NULL')
```

```
dfproducer1.columns
```

```
['Summons Number',
 'Issuer Command',
 'Violation_County',
 'Days Parking In Effect',
 'From Hours In Effect',
 'Vehicle Year']
```

Filling it with NULL values is important because if this is not done then the next part which involves converting to json file format will not have any values and that will give an error during Kafka training.

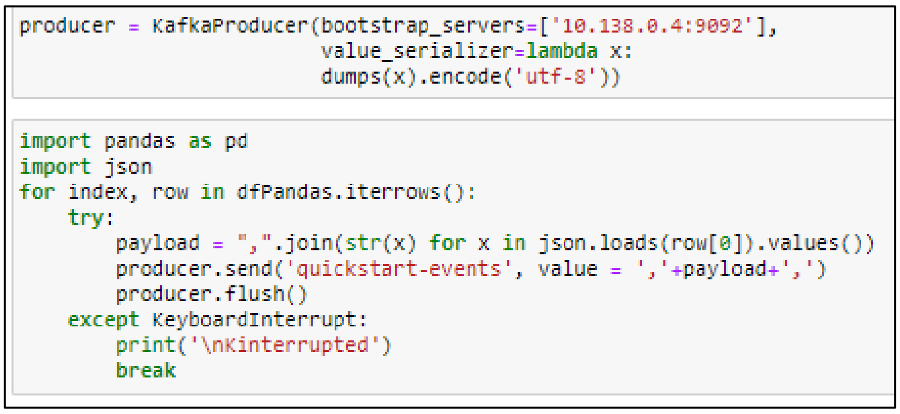
- The data is converted into a Pandas Dataframe in order to iterate through it conveniently.



```
dfPandas = dfproducer1.toPandas()
```

- The Kafka topic ‘quickstart-events’ is created using the command:
  `kafka-topics.sh --create --topic quickstart-events --bootstrap-server localhost:9092`

- Then, the Kafka Producer is initialized and the data is sent in through send and flush commands.



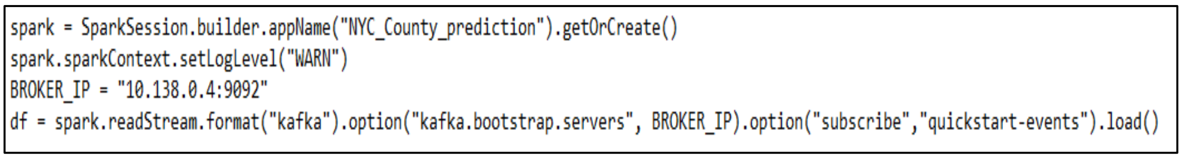
```
producer = KafkaProducer(bootstrap_servers=['10.138.0.4:9092'],
                         value_serializer=lambda x:
                         dumps(x).encode('utf-8'))
```

```
import pandas as pd
import json
for index, row in dfPandas.iterrows():
    try:
        payload = ",".join(str(x) for x in json.loads(row[0]).values())
        producer.send('quickstart-events', value = ','+payload+',')
        producer.flush()
    except KeyboardInterrupt:
        print('\nKinterrupted')
        break
```

Spark Stream Reader – Streaming Sink

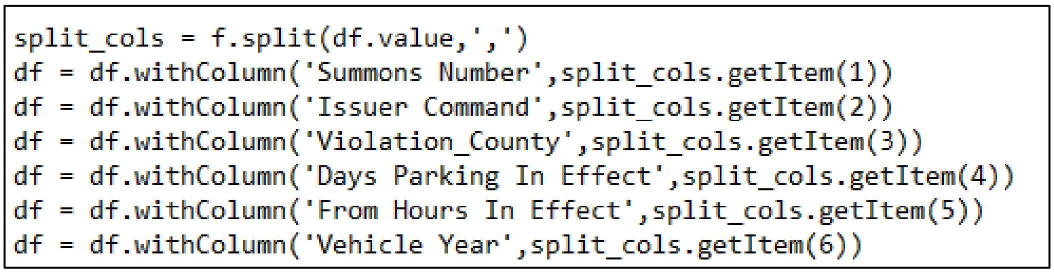
We then have subscriber code to make predictions on the test data using the model saved on the GCP bucket by streaming with Kafka. The code is explained as follows.

- A spark session is started and the stream is read from the Kafka IP address



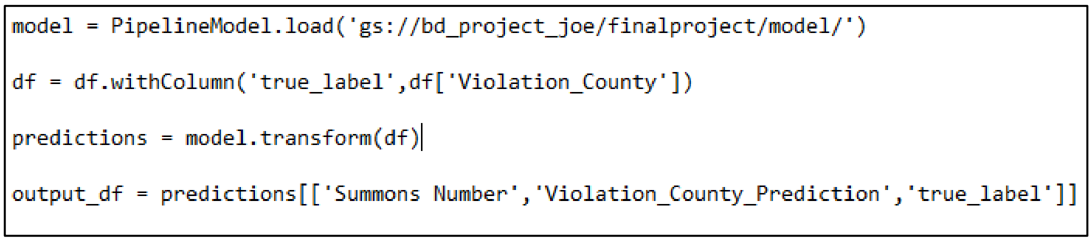
```
spark = SparkSession.builder.appName("NYC_County_prediction").getOrCreate()
spark.sparkContext.setLogLevel("WARN")
BROKER_IP = "10.138.0.4:9092"
df = spark.readStream.format("kafka").option("kafka.bootstrap.servers", BROKER_IP).option("subscribe","quickstart-events").load()
```

Note:
In the producer code, in the producer.send part, comma is added on both sides of the data. This is because in the consumer set up for Kafka where we can see the messages, the double quotes of the string datatype is retained, hence commas are added to remove those double quotes.



```
split_cols = f.split(df.value,',')
df = df.withColumn('Summons Number',split_cols.getItem(1))
df = df.withColumn('Issuer Command',split_cols.getItem(2))
df = df.withColumn('Violation_County',split_cols.getItem(3))
df = df.withColumn('Days Parking In Effect',split_cols.getItem(4))
df = df.withColumn('From Hours In Effect',split_cols.getItem(5))
df = df.withColumn('Vehicle Year',split_cols.getItem(6))
```

- The data is read as string, the string is converted into list of values.
- The row is split into features, we start at 1 instead of 0 because of the extra comma added, for getting the features.
- Then we load the model using pipeline model (which has already been saved) and use the transformations to create the prediction. The final output only has 3 columns: summons number, violation county prediction and true label.



```
model = PipelineModel.load('gs://bd_project_joe/finalproject/model/')

df = df.withColumn('true_label',df['Violation_County'])

predictions = model.transform(df)

output_df = predictions[['Summons Number','Violation_County_Prediction','true_label']]
```

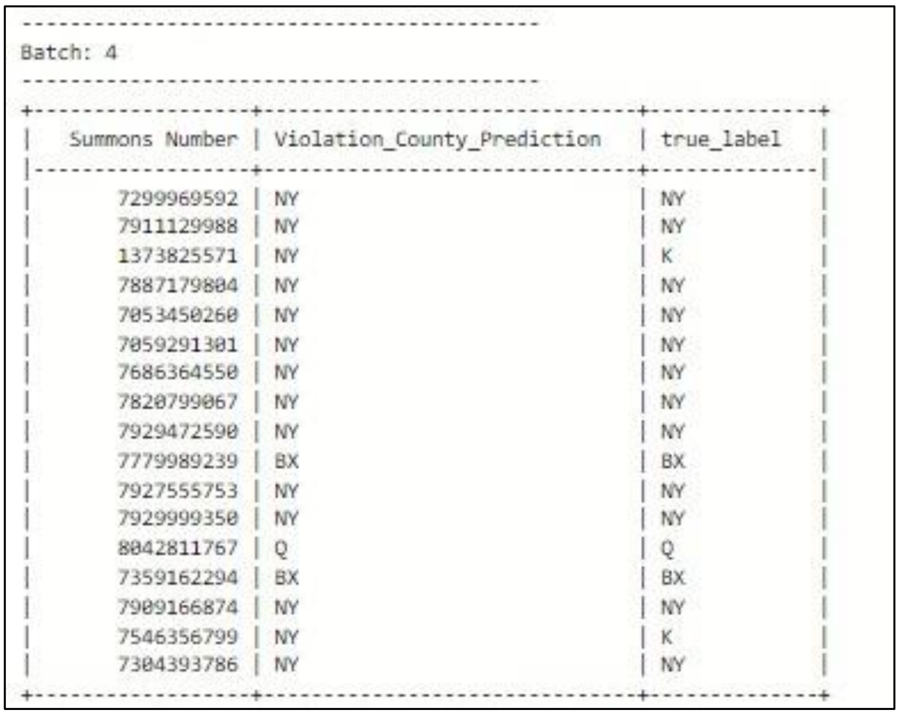
- The layout of the output is created suing the tabulate library. One query is used to show the matrix and one query to show the table completely, but it would be misaligned, because one would run faster than the other and would be asynchronous. Hence it was created such that both outputs would be printed together using one single query using the foreachBatch command.

```
def foreach_batch_function(df, epoch_id):
    dftemp = df.toPandas()
    if dftemp.shape[0] > 0:
        acc = accuracy_score(dftemp['true_label'], dftemp['Violation_County_Prediction'])
        f1 = f1_score(dftemp['true_label'], dftemp['Violation_County_Prediction'], average='weighted')
        output = pd.DataFrame([['Accuracy', acc],['F1-Score', f1]], columns = ['Metric', 'Value'])
        print('---------------------------------------------')
        print("Batch:", epoch_id)
        print('---------------------------------------------')
        print(tabulate(dftemp, headers='keys', tablefmt='psql', showindex=False))
        print('Metics for Batch',epoch_id,'is:')
        print(tabulate(output, headers = 'keys', tablefmt='psql', showindex=False))
    # Transform and write batchDF
    pass
```

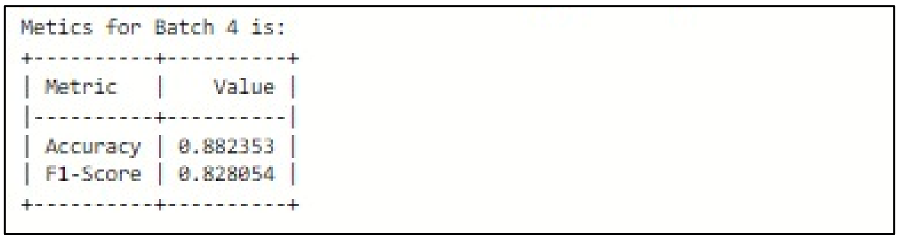
- The table was printed only for non-null Dataframes. (There are a lot of null empty dataframes appearing in streaming)
- Pandas Dataframe was used for easy access and the Accuracy and F1 scores were printed.

Results

The screenshot of the output obtained for one batch in Kafka Streaming is given as follows:

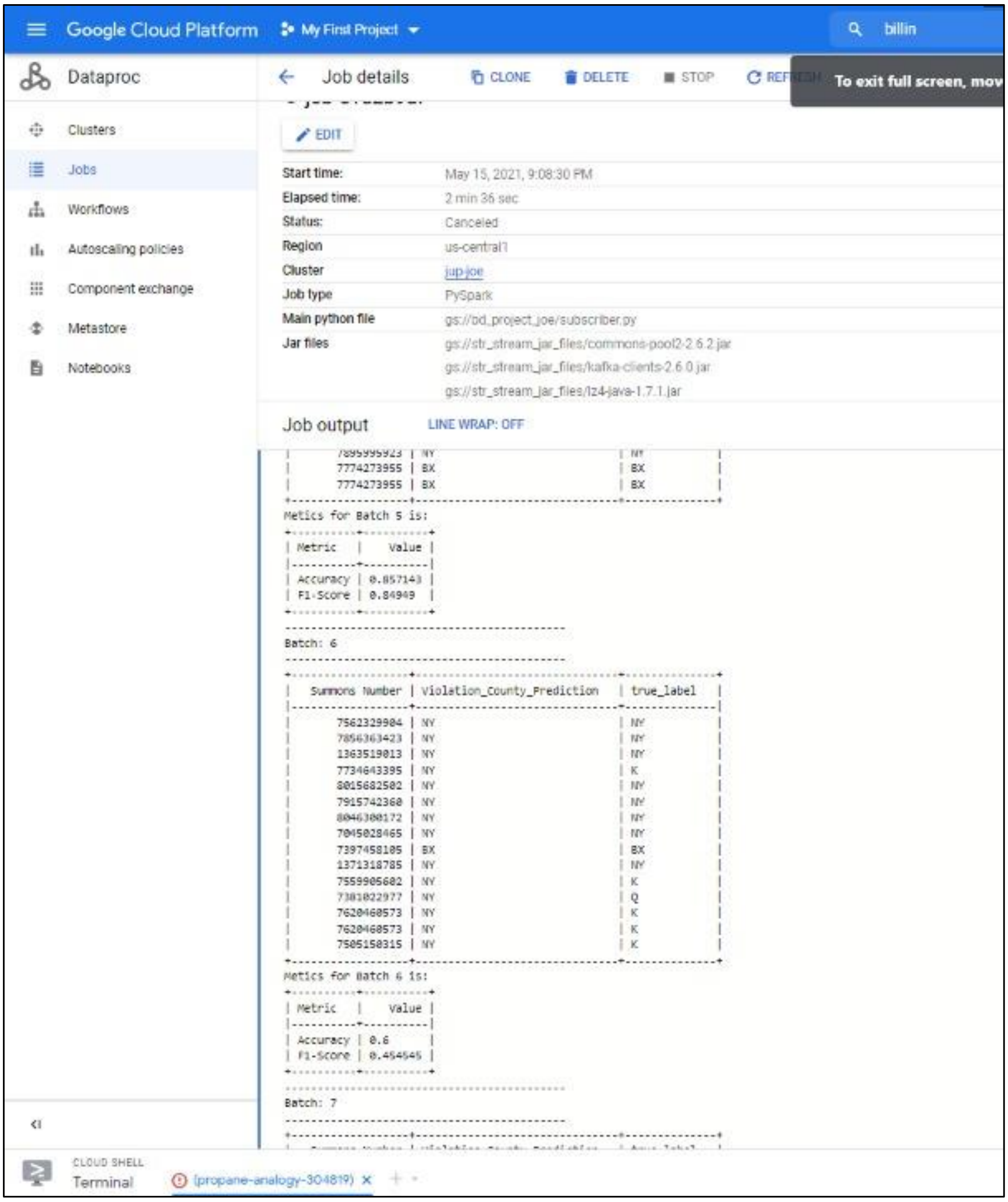


```
-------------------------------------------
Batch: 4
-------------------------------------------
+------------------+---------------------------------+--------------+
|   Summons Number | Violation_County_Prediction     | true_label   |
|------------------+---------------------------------+--------------|
|       7299969592 | NY                              | NY           |
|       7911129988 | NY                              | NY           |
|       1373825571 | NY                              | K            |
|       7887179804 | NY                              | NY           |
|       7053450260 | NY                              | NY           |
|       7059291301 | NY                              | NY           |
|       7686364550 | NY                              | NY           |
|       7820799067 | NY                              | NY           |
|       7929472590 | NY                              | NY           |
|       7779989239 | BX                              | BX           |
|       7927555753 | NY                              | NY           |
|       7929999350 | NY                              | NY           |
|       8042811767 | Q                               | Q            |
|       7359162294 | BX                              | BX           |
|       7909166874 | NY                              | NY           |
|       7546356799 | NY                              | K            |
|       7304393786 | NY                              | NY           |
+------------------+---------------------------------+--------------+
```



```
Metics for Batch 4 is:
+----------+----------+
| Metric   |    Value |
|----------+----------|
| Accuracy | 0.882353 |
| F1-Score | 0.828054 |
+----------+----------+
```

Below is a screenshot of our Google Cloud Platform with the output:

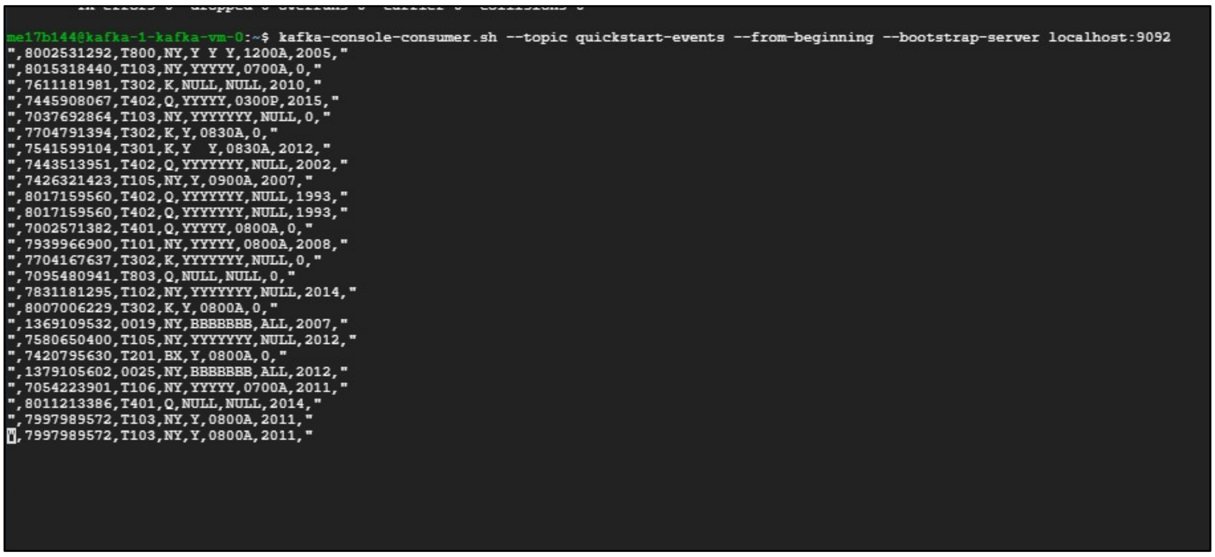


Start time:	May 15, 2021, 9:08:30 PM
Elapsed time:	2 min 36 sec
Status:	Canceled
Region	us-central1
Cluster	jup-joe
Job type	PySpark
Main python file	gs://bd_project_joe/subscriber.py
Jar files	gs://str_stream_jar_files/commons-pool2-2.6.2.jar gs://str_stream_jar_files/kafka-clients-2.6.0.jar gs://str_stream_jar_files/lz4-java-1.7.1.jar

Job output

```
|       7895995923 | NY                                | NY           |
|       7774273955 | BX                                | BX           |
|       7774273955 | BX                                | BX           |
+------------------+-----------------------------------+--------------+
Metics for Batch 5 is:
+----------+----------+
| Metric   |    Value |
|----------+----------|
| Accuracy | 0.857143 |
| F1-Score | 0.84949  |
+----------+----------+
-------------------------------------------
Batch: 6
-------------------------------------------
+------------------+-----------------------------------+--------------+
|   Summons Number | Violation_County_Prediction       | true_label   |
|------------------+-----------------------------------+--------------|
|       7562329904 | NY                                | NY           |
|       7856363423 | NY                                | NY           |
|       1363519013 | NY                                | NY           |
|       7734643395 | NY                                | K            |
|       8015682502 | NY                                | NY           |
|       7915742360 | NY                                | NY           |
|       8046300172 | NY                                | NY           |
|       7045028465 | NY                                | NY           |
|       7397458105 | BX                                | BX           |
|       1371318785 | NY                                | NY           |
|       7559905602 | NY                                | K            |
|       7381022977 | NY                                | Q            |
|       7620460573 | NY                                | K            |
|       7620460573 | NY                                | K            |
|       7505158315 | NY                                | K            |
+------------------+-----------------------------------+--------------+
Metics for Batch 6 is:
+----------+----------+
| Metric   |    Value |
|----------+----------|
| Accuracy | 0.6      |
| F1-Score | 0.454545 |
+----------+----------+
-------------------------------------------
Batch: 7
-------------------------------------------
+------------------+-----------------------------------+--------------+
```

The screenshot of the consumer output is given as follows



```
me17b144@kafka-1-kafka-vm-0:~$ kafka-console-consumer.sh --topic quickstart-events --from-beginning --bootstrap-server localhost:9092
",8002531292,T800,NY,Y Y Y,1200A,2005,"
",8015318440,T103,NY,YYYYY,0700A,0,"
",7611181981,T302,K,NULL,NULL,2010,"
",7445908067,T402,Q,YYYYY,0300P,2015,"
",7037692864,T103,NY,YYYYYYY,NULL,0,"
",7704791394,T302,K,Y,0830A,0,"
",7541599104,T301,K,Y  Y,0830A,2012,"
",7443513951,T402,Q,YYYYYYY,NULL,2002,"
",7426321423,T105,NY,Y,0900A,2007,"
",8017159560,T402,Q,YYYYYYY,NULL,1993,"
",8017159560,T402,Q,YYYYYYY,NULL,1993,"
",7002571382,T401,Q,YYYYY,0800A,0,"
",7939966900,T101,NY,YYYYY,0800A,2008,"
",7704167637,T302,K,YYYYYYY,NULL,0,"
",7095480941,T803,Q,NULL,NULL,0,"
",7831181295,T102,NY,YYYYYYY,NULL,2014,"
",8007006229,T302,K,Y,0800A,0,"
",1369109532,0019,NY,BBBBBBB,ALL,2007,"
",7580650400,T105,NY,YYYYYYY,NULL,2012,"
",7420795630,T201,BX,Y,0800A,0,"
",1379105602,0025,NY,BBBBBBB,ALL,2012,"
",7054223901,T106,NY,YYYYY,0700A,2011,"
",8011213386,T401,Q,NULL,NULL,2014,"
",7997989572,T103,NY,Y,0800A,2011,"
",7997989572,T103,NY,Y,0800A,2011,"
```

Conclusions

- Hence, the given aim of predicting the Violation County of the Parking Defaults from the given dataset was achieved.
- For the individual batches, the accuracies and F1 scores of prediction were printed. The accuracies turn out to be around 90 % and the F1 scores, around 80 %
- Transformations are much faster than fitting algorithms, especially for large datasets.
- The use of CatBoost to select relevant features enabled us to drastically reduce the number of columns we had to work with, thus reducing the run time.