Big Data Lab
CS4830
Final Project Report

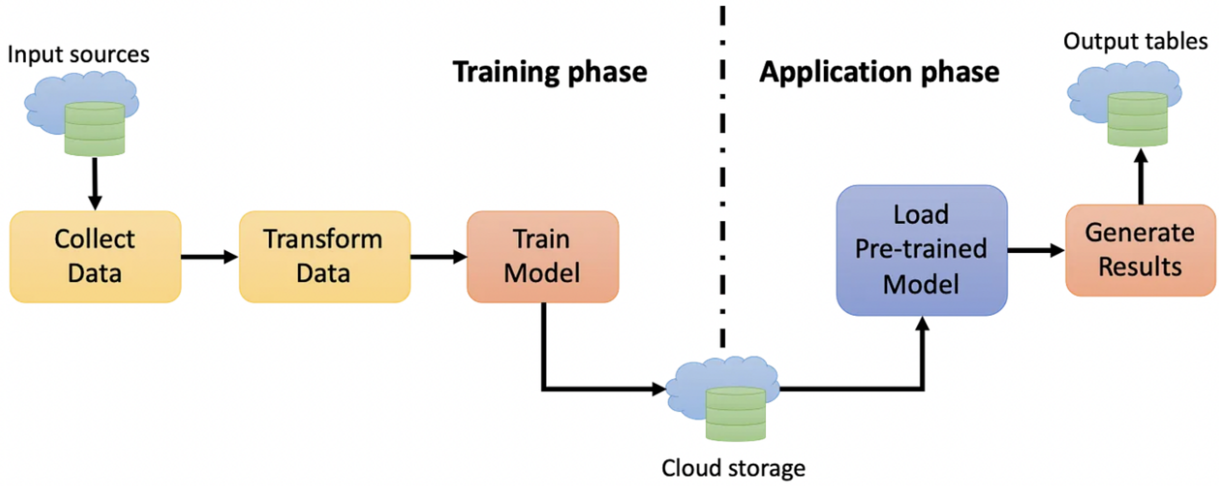
Team Name:
GigaByteCGS

Team Members:
Chetan Reddy N (ME19B093)
Shrung D N (ME19B168)
Gangadhar (ME19B190)

Dataset Used:
NYC Parking Tickets Dataset

Objective:

- The objective of the model is to train a model on a very large dataset and further using the trained model to perform real-time predictions using big-data technologies
- The dataset used is NYC Parking Tickets and the prediction variable is the violation precinct (the part in the city where the ticket was issued). The dataset has about **22 million rows (data points).**
- The following image gives a gist of the project



- **Batch Computation (or Training Phase):**
  a) This is the first part of the project where a model is trained on a large dataset after performing the preprocessing steps. The computation is done using **Dataproc Cluster** and **PySpark**.
  b) The trained model is stored in a GCS bucket

- **Real-Time Computation (or Application Phase):**
  a) The test data is streamed into **Kafka** from a GCS bucket
  b) **Spark Streaming** is then used to read the data and make real-time predictions using the stored model. A producer script is written to read the data from the GCS bucket and publish it to a kafka topic. A consumer code accesses the data from the topic and generates the predictions.

Pre-Processing the Data:

1. **Loading the Data:**
   A Dataproc Cluster is created with a web interface enabled. It should be noted that processing the given dataset on Google Colab or locally on the PC is very difficult. The CSV file is accessed from the given location: **gs://bdl2023-final-proj/trainingdatanyc.csv**

2. **Basic Analysis of the Dataset:**
   Size of the Dataset: **22436132**
   Number of Columns: **43** (including prediction variable)

3. **Violation Location vs Violation Precinct:**
   The target variable mentioned in the question was “Violation Precinct”. We noticed that there is another column named “Violation Location” with exactly the same entries with slightly different forms. **<u>We decided to not use “Violation Location” as a feature variable</u>** as that would beat the purpose of training and using an ML model.

Violation Precinct	Violation Location
71	0071
108	108
109	109
71	0071
0	null
34	0034
115	115
0	null
0	null
0	null
103	103
94	0094
109	109
0	null
14	0014
0	null
88	0088
88	0088
10	0010
52	0052

4. **Checking NaN or Null Values in the Columns:**
   For each column, the number of NaN values was computed as a percentage of the total dataset. It was found that 8 columns had more than 70% NaN values. The columns with high NaN value percentages were eliminated.

Percentage of NULL Values in each Column

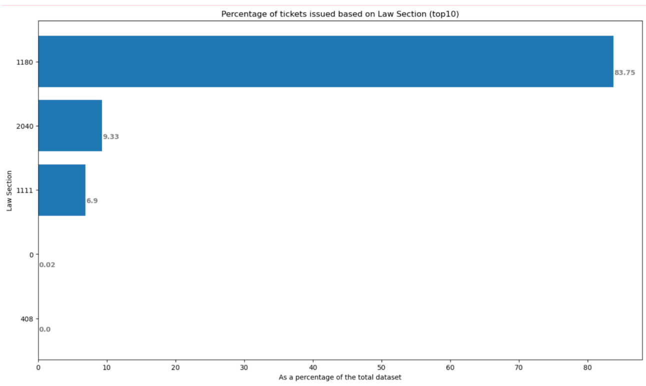
Double Parking Violation	:	100.0 %
Hydrant Violation	:	100.0 %
No Standing or Stopping Violation	:	100.0 %
Time First Observed	:	89.5 %
Unregistered Vehicle?	:	88.78 %
Violation Legal Code	:	83.76 %
Meter Number	:	81.44 %
Intersecting Street	:	71.64 %
From Hours In Effect	:	45.3 %
To Hours In Effect	:	45.3 %
Violation Post Code	:	27.45 %
Days Parking In Effect	:	25.43 %
House Number	:	17.94 %
Violation In Front Of Or Opposite	:	17.04 %
Violation Location	:	16.35 %
Issuer Squad	:	16.23 %
Issuer Command	:	16.23 %
Violation County	:	15.84 %
Violation Description	:	11.23 %
Vehicle Color	:	1.17 %

5. **Checking the Number of Unique Values in the Columns:**
   Most of the features in this dataset are categorical features. The only way for the model to process it is to use One-Hot Encoding or Label Encoding (among other encoding methods). However, if a categorical column has a very high number of unique values, it is not very useful and is eliminated.

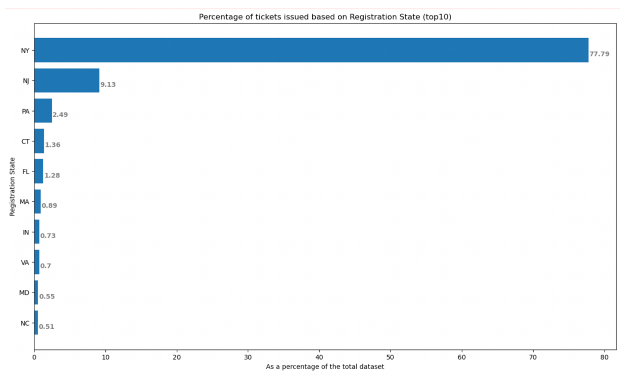
```
Number of Unique Values in each column

Law Section                           :       5
Violation In Front Of Or Opposite     :       6
Feet From Curb                        :      17
Issuing Agency                        :      19
Violation County                      :      20
Issuer Squad                          :      46
Registration State                    :      69
Plate Type                            :      89
Vehicle Year                          :     100
Violation Code                        :     100
Violation Description                 :     107
Sub Division                          :     142
Days Parking In Effect                :     185
Violation Location                    :     590
Violation Precinct                    :     591
From Hours In Effect                  :     658
To Hours In Effect                    :     726
Issuer Precinct                       :     815
Violation Post Code                   :    1071
Date First Observed                   :    1148
Violation Time                        :    2056
Issue Date                            :    2854
Vehicle Body Type                     :    3323
Vehicle Color                         :    3867
Issuer Command                        :    4968
Street Code1                          :    6813
Street Code3                          :    6939
Street Code2                          :    7145
Vehicle Expiration Date               :    8592
Vehicle Make                          :   10387
Issuer Code                           :   51779
House Number                          :   69450
Street Name                           :  161140
Plate ID                              : 4621763
Summons Number                        :21355951
```

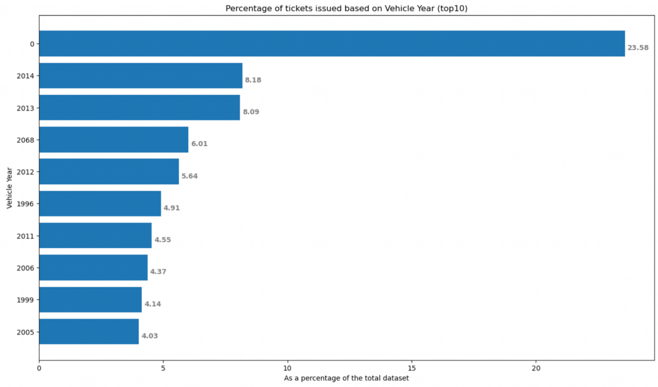
6. **Visualisations:**
The frequency of different values for each column is graphed and insights are obtained. Based on the Steps 3 and 4, the number of columns finally used are reduced to 13.



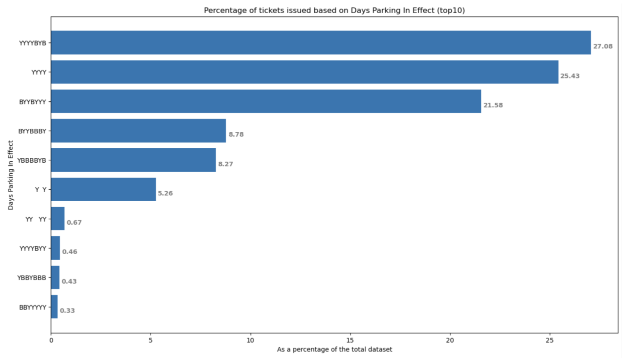
Percentage of Tickets issued Based on Law Section



Percentage of Tickets issued Based on Registration State



Percentage of Tickets issued based on Vehicle Year



Percentage of Tickets issued based on Days Parking in Effect

7. **Categorical Features and Numerical Features:**
   StringIndexer is used to convert categorical features into labels. Null and unseen values are given a special label (used the argument handleInvalid='keep'). The numerical features are converted to float datatype and nan values are handled here by using 'fillna' function.

Model Training and Best Model:

- A simple logistic regression model was used initially with few features to check how much data could be handled.
- The configuration of the Dataproc Cluster is limited to 8 CPU cores due to which it was not possible to process all 2 crore rows at once. Therefore, a subset of 200,000 rows was used for model training.
- The following is the time taken by different processes using two different cluster configurations. C2 Compute Optimised cluster was used for all the training (but it's more expensive).

DataProc Cluster Configuration	df.count()	Null/NaN Values Count	Training Random Forest with 10000 rows	Training Random Forest with 100000 rows
4 vCPU 4GB N2 Gen	~15 second	16 min	2 min 30 seconds	10 min
C2 Compute Optimised 8 vCPU 32GB	~11 seconds	~ 8 min	1 min	3 min 34 sec

- A pipeline was constructed which includes the preprocessing objects i.e. indexers, assembler and the ML model.

```
rf = RandomForestClassifier(numTrees=50, maxBins=1000,maxDepth=10)
list_of_stages = indexers+[target_indexer,assembler,rf]
pipeline = Pipeline(stages=list_of_stages)

model_rf = pipeline.fit(train_df_raw)
print("Model Trained")
```

- The following models were experimented - Logistic Regression, Random Forest and Naive Bayes Algorithm.
- After hyperparameter tuning using grid search, **Random Forest Classifier** generated the best results as shown below:

```
Accuracy on test df: 0.87
F1 Score on test df: 0.85
```

- This is because the dataset has many categorical features. A tree-based algorithm such as Random Trees will be able to grasp these highly non-linear features and can perform much better than its counterparts.

Real-Time Computation:

- The trained model is stored in a GCS bucket.
- A producer script is written to read the csv data from a local copy of the same and streamed it into a Kafka topic.

```
# Streaming data into kafka topic
for i, d in enumerate(df.toJSON().collect()):
    print(f'{i}: Sending Message to {TOPIC_NAME}')
    msg = d.encode('utf-8')
    producer.send(TOPIC_NAME, msg)
    time.sleep(2)
```

- A consumer script subscribes to the topic and accesses the data.

```
    # Read from kafka topic
    records = spark\
        .readStream\
        .format('kafka')\
        .option('kafka.bootstrap.servers', BOOTSTRAP_SERVER)\
        .option('subscribe', TOPIC_NAME)\
        .option("startingOffsets",'latest')\
        .option("includetimestamp",'true')\
        .load()
```

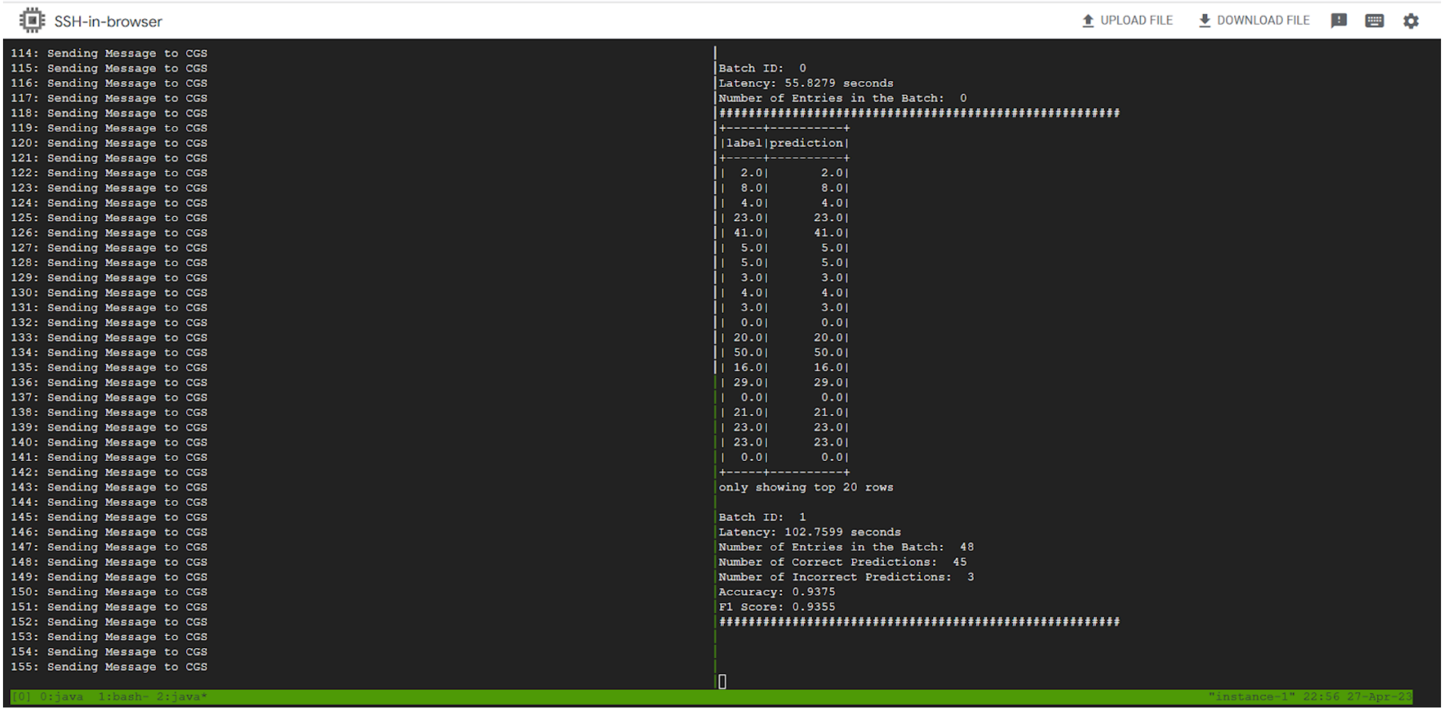
- The trained model is invoked and predictions are made. The F1_score and accuracy are calculated and displayed

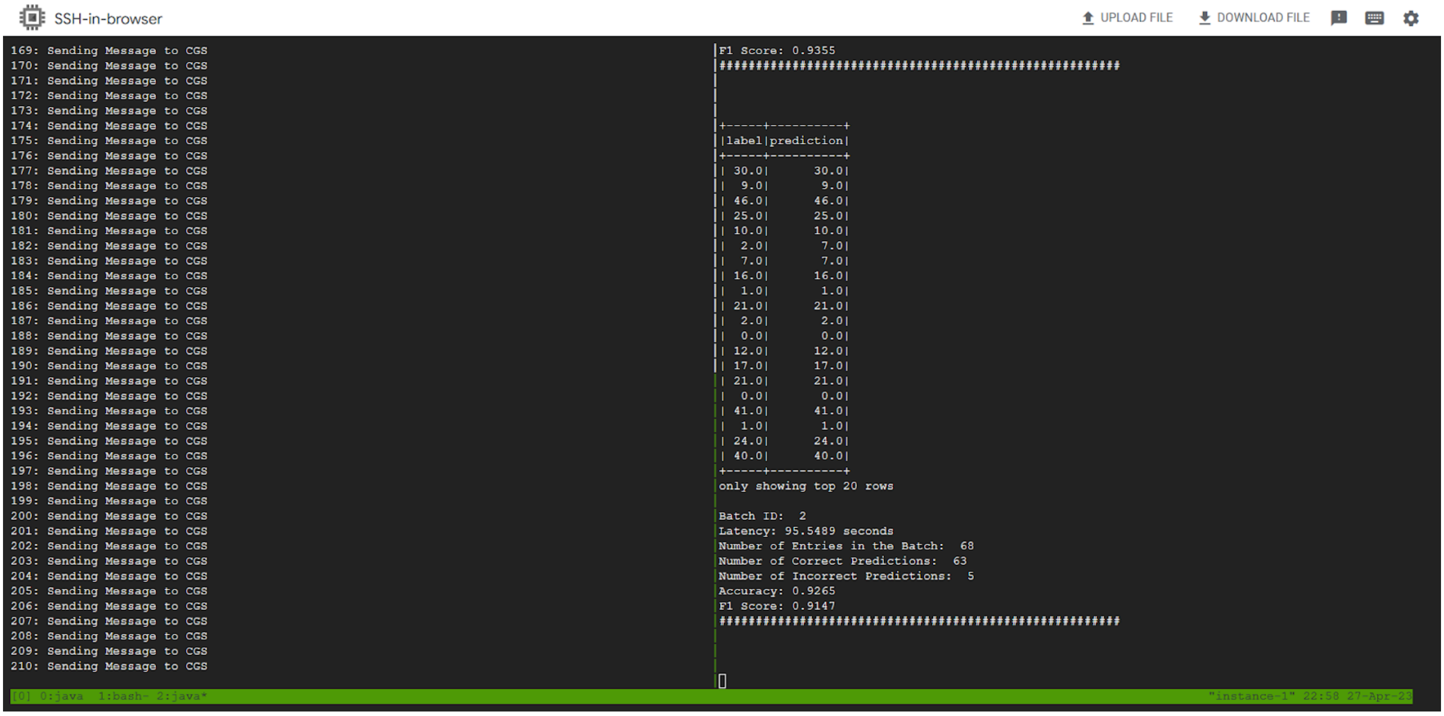
```
    # Function to print logs in real-time
    def process_data(df, ID):
        t1 = time.time()
        df_pp = preprocess_data(df)
        preds = model.transform(df_pp)
        print_preds = preds.select(['label', 'prediction'])
        num_rows = print_preds.count()
        accuracy = accuracy_evaluator.evaluate(preds)
        f1_score = f1_evaluator.evaluate(preds)
        t2 = time.time()
        latency = t2 - t1

        try:
            print_preds.show()
            print('Batch ID: ', ID)
            print('Latency: {:.4f} seconds'.format(latency))
            print('Number of Entries in the Batch: ', num_rows)
            print('Number of Correct Predictions: ', int(num_rows * accuracy))
            print('Number of Incorrect Predictions: ', num_rows - int(num_rows * accuracy))
            print('Accuracy: {:.4f}'.format(accuracy))
            print('F1 Score: {:.4f}'.format(f1_score))
            print('#######################################################')
            print('\n\n')
        except:
            print('#######################################################')
```

Results:

- The results are shown below:





- The left side of the screen shows the output of the producer. Every row (data point) is streamed to the topic with a frequency of 0.5 Hz.
- The right side of the screen shows the output of the consumer. It takes a group of data points based on a time window and the ML model gives the predictions on the data points in this time window. The true values and predictions are displayed along with other metrics

Conclusion:

- Big data processing is very time-consuming and cannot be done using traditional data processing techniques and libraries.
- The provided data has mostly categorical columns (nominal data). Therefore, **tree-based algorithms** (which are essentially composed of if-else statements suitable for nominal data) **are best suited** for the NYC parking dataset.
- The high latency obtained is due to the random forest model, which takes a significant amount of time for prediction. This latency can be reduced by increasing the number of workers.
- We observed that simpler models like logistic regression and naive bayes models can predict much quicker, but with lower accuracy. Hence, the time for computation and the accuracy is a trade-off and a suitable model must be chosen based on the application.