Data processing on a big dataset with Databricks Spark

Student Name: *Yasaman Mohammadi*
Student ID: 24612626
course Name*: Big Data Engineering*

1. Introduction

The New York City Taxi and Limousine Commission (TLC) is the agency responsible for licensing and regulating New York City's taxi cabs since 1971. TLC has publicly published millions of trip records from both yellow and green taxi cabs.
Each record includes fields capturing pick-up and drop-off dates/times, locations, trip distances, itemised fares, rate types, payment types, and driver-reported passenger counts.

Yellow taxi cabs are the iconic taxi vehicles from New York city that have the right to pick up street-hailing passengers anywhere in the city. There are around 13,600 authorised taxis in New York City and each taxi must have a yellow medallion affixed to it.

Green taxis were introduced in August 2013 to improve taxi service and availability in the boroughs. Green taxis may respond to street hails, but only in certain designated areas.

1.1. Role

The goal of this assignment is to analyse a large dataset using Spark. You will have to load the available data, perform data transformation and analysis on it and finally train a Machine Learning algorithm for predicting a continuous outcome.

2. Setup

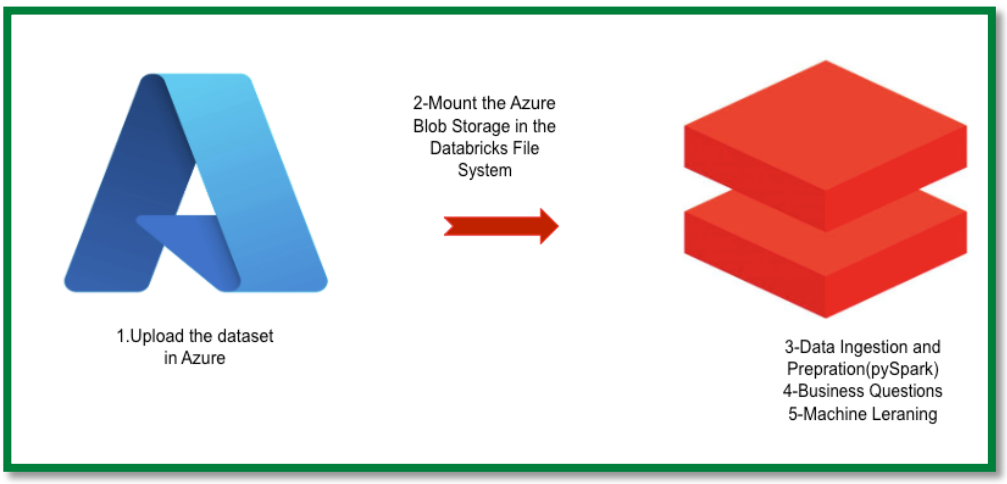

Figure 1-workflow

2.1 Upload data into Azure Storage

First, the dataset parquet files for yellow and green taxi cabs from **2015 to 2022** were downloaded.Next, an Azure account was created, and the data was uploaded to a container called "bde2" on a storage account called "bdemohammadiyasaman".

2.2 Mount the Azure Blob Storage into the Databricks File System:

In the Databricks File System, the Azure Blob Storage was mounted.

Mount the Azure Blob Storage into the Databricks File System

```
#Create 3 variables
storage_account_name = "bdemohammadiyasaman"
storage_account_access_key = "uZjq1op2dZPzHodcuA70WYFcu34ovcsYfcUYZYq+WjlCPVZQugTrzH2z4zj3FZKFqMTwGU7Sn+Sa+AStBxN8dw=="
blob_container_name = "bde2"
mount_name ='mount_name'

```

```
dbutils.fs.mount(
  source = f'wasbs://{blob_container_name}@{storage_account_name}.blob.core.windows.net',
  mount_point = f'/mnt/{blob_container_name}/',
  extra_configs = {'fs.azure.account.key.' + storage_account_name + '.blob.core.windows.net': storage_account_access_key}
)
```

We read parquet files of each taxi colors inside "/mnt/bde2" and merge each color by the file name's pattern together and then save it to variables called "green_taxi_df" and "yellow_taxi_df".

```
from pyspark.sql.functions import lit

#define the path for green taxi
green_taxi_path= "/mnt/bde2/green_*.parquet"
yellow_taxi_path= "/mnt/bde2/yellow_*.parquet"

# Read green taxi files from Azure Storage
green_taxi_df = spark.read.parquet(green_taxi_path)

# Read yellow taxi files from Azure Storage
yellow_taxi_df = spark.read.parquet(yellow_taxi_path)

# Define the DBFS paths for copying
dbfs_green_taxi_path = "/dbfs/mnt/bde2/green_taxi"
dbfs_yellow_taxi_path = "/dbfs/mnt/bde2/yellow_taxi"

# Write green taxi data to DBFS
green_taxi_df.write.parquet(dbfs_green_taxi_path, mode="overwrite")

# Write yellow taxi data to DBFS
yellow_taxi_df.write.parquet(dbfs_yellow_taxi_path, mode="overwrite")
```

Total number of each taxi color by reading the files stored on DBFS:

- Green taxi: 66,200,401
- Yellow taxi: 663,055,251

the location referential csv was loaded.

Table 1-Zone Table

```
#loading zone csv file
zone_csv ="/mnt/bde2/taxi_zone_lookup.csv"

#make a dataframe out of it
zone_df = spark.read.option("header", True).csv(zone_csv)

display(zone_df)
```

▸ (2) Spark Jobs

▸ zone_df: pyspark.sql.dataframe.DataFrame = [LocationID: string, Borough: string ... 2 more fields]

Table +

	LocationID	Borough	Zone	service_zone
1	1	EWR	Newark Airport	EWR
2	2	Queens	Jamaica Bay	Boro Zone
3	3	Bronx	Allerton/Pelham Gardens	Boro Zone
4	4	Manhattan	Alphabet City	Yellow Zone
5	5	Staten Island	Arden Heights	Boro Zone
6	6	Staten Island	Arrochar/Fort Wadsworth	Boro Zone
7	7	Queens	Astoria	Boro Zone

265 rows | 6.49 seconds runtime

```
# Convert the "Green" 2015 parquet into a csv file
parquet_file_path = "/mnt/bde2/green_taxi_2015.parquet"
csv_file_path = "/mnt/bde2/green_2015.csv"

#send it to your Azure Blob Storage
azure_blob_storage_loc ="/mnt/bde2/green_2015.csv"


#Read the Parquet file into a DataFrame
df_green_2015 = spark.read.parquet(parquet_file_path)

# Save the DataFrame as a CSV file
df_green_2015.write.mode("overwrite").csv(csv_file_path, header=True, sep=",")

#Copy the CSV file to Azure Blob Storage
dbutils.fs.cp(csv_file_path, azure_blob_storage_loc, recurse=True)

print("CSV file saved to Azure blob storage")
```

After converting the "Green" 2015 Parquet file into a CSV format and sending it to our Azure Blob Storage, we observed that the Parquet file is significantly smaller than its CSV counterpart. Since our dataset comprises a vast 729,255,652 records, processing this extensive dataset is time-consuming. Therefore, leveraging a data storage and processing method that offers faster read and write capabilities is advisable.

2. Analysing Data on Databricks

4.1 Data Ingestion and preparation

We load the data by reading it from DBFS:

```
#load the data back into a DataFrame (you can run from here)
loaded_green_taxi_df = spark.read.format("parquet").load("/mnt/bde2/green_*.parquet")
loaded_yellow_taxi_df = spark.read.format("parquet").load("/mnt/bde2/yellow_*.parquet")
```

4.2Data Cleaning

We initiated the data cleaning process using PySpark to manipulate data frames.

a. *Removing trips finishing before the start time*

Upon inspecting the schemas of both data frames, it became evident that there were discrepancies in the column names containing pickup and drop-off dates. Consequently, we harmonized these column names to combine them in the subsequent steps.

```
#rename unmatched columns
loaded_green_taxi_df = loaded_green_taxi_df.withColumnRenamed("lpep_pickup_datetime", "tpep_pickup_datetime")
loaded_green_taxi_df = loaded_green_taxi_df.withColumnRenamed("lpep_dropoff_datetime", "tpep_dropoff_datetime")
```

```
# Filter out unrealistic trips
green_clean = loaded_green_taxi_df[loaded_green_taxi_df["tpep_dropoff_datetime"] > loaded_green_taxi_df["tpep_pickup_datetime"]]
```

Total number of unrealistic trips of each taxi:

- Green taxi: 56443
- Yellow taxi: 714917

b. *Removing trips where the pickup/dropoff datetime is outside of the range.*

We extract the year, weekday, month, day of the year, and trip duration from the “tpep_pickup_datetime” column to filter out unrealistic trip information.

```
def datetime_info(df):
    df = df.withColumn("year", year(col("tpep_pickup_datetime")))
    df = df.withColumn("weekday", dayofweek(col("tpep_pickup_datetime")))
    df = df.withColumn("hour", hour(col("tpep_pickup_datetime")))
    df = df.withColumn("month", month(col("tpep_pickup_datetime")))
    df = df.withColumn("day_of_year", dayofyear(col("tpep_pickup_datetime")))

    pickup_timestamp = unix_timestamp("tpep_pickup_datetime")
    dropoff_timestamp = unix_timestamp("tpep_dropoff_datetime")
    df = df.withColumn("trip_duration_sec", (dropoff_timestamp - pickup_timestamp).cast(IntegerType()))

    return df
```

As the dataset exclusively comprises data from 2015 to 2020, we have confined our dataset to this specific timeframe.

```
# Filter out rows with unrealistic years
yellow_clean = yellow_clean.filter((col("year") >= start_year) & (col("year") <= end_year))
```

c. Trips with negative speed

```
from pyspark.sql import functions as F

# Assuming you have a DataFrame named green_clean
green_clean = green_clean.withColumn(
    "speed",
    ((green_clean["trip_distance"] *1.60934) / green_clean["trip_duration_sec"]) * 3600.0 #miles/h
)
```

Total number of negative speeds of each taxi:

- Green taxi: 19526
- Yellow taxi: 11449

As the dataset lacks a "speed" column, we calculated speed using the "trip_distance" and "trip_duration_sec" columns. We converted this speed measurement from miles per second to kilometers per hour by multiplying it by (1.60934 * 3600).

d. Trips with very high speed (look for NYC and outside of NYC speed limit)

we have 5 service zones in the datasets:

Table 2-service_zone

```
+------------+
|service_zone|
+------------+
|         EWR|
|         N/A|
| Yellow Zone|
|    Airports|
|   Boro Zone|
+------------+
```

Inside NYC:

Yellow Zone: The Yellow Zone is the standard taxi service area within New York City.

Outside NYC:

EWR: This likely refers to Newark Liberty International Airport, which is located in Newark, New Jersey, and is outside of New York City.

Table 3-Servise zone / Brough / Zone

LocationID	Borough	Zone	service_zone
1	EWR	Newark Airport	EWR

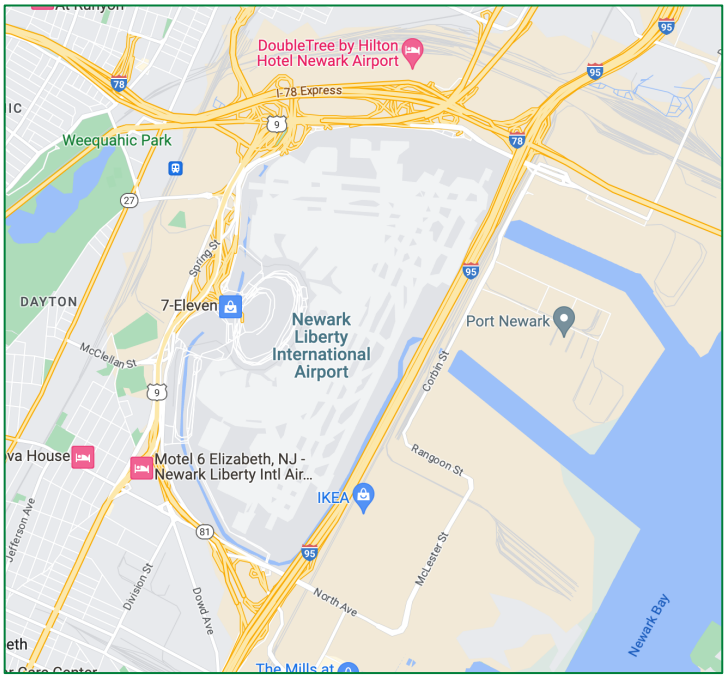

Figure 2-Newark Airport

N/A: "N/A" usually means "Not Applicable" or "Not Available," so it does not specify a location.

Airports: This could encompass various airports, but it is important to note that some airports, like JFK and LaGuardia, are located within NYC, while others, like Newark Liberty International Airport, are outside NYC.

Table 4-Airport's zones

LocationID	Borough	Zone	service_zone
132	Queens	JFK Airport	Airports
138	Queens	LaGuardia Airport	Airports

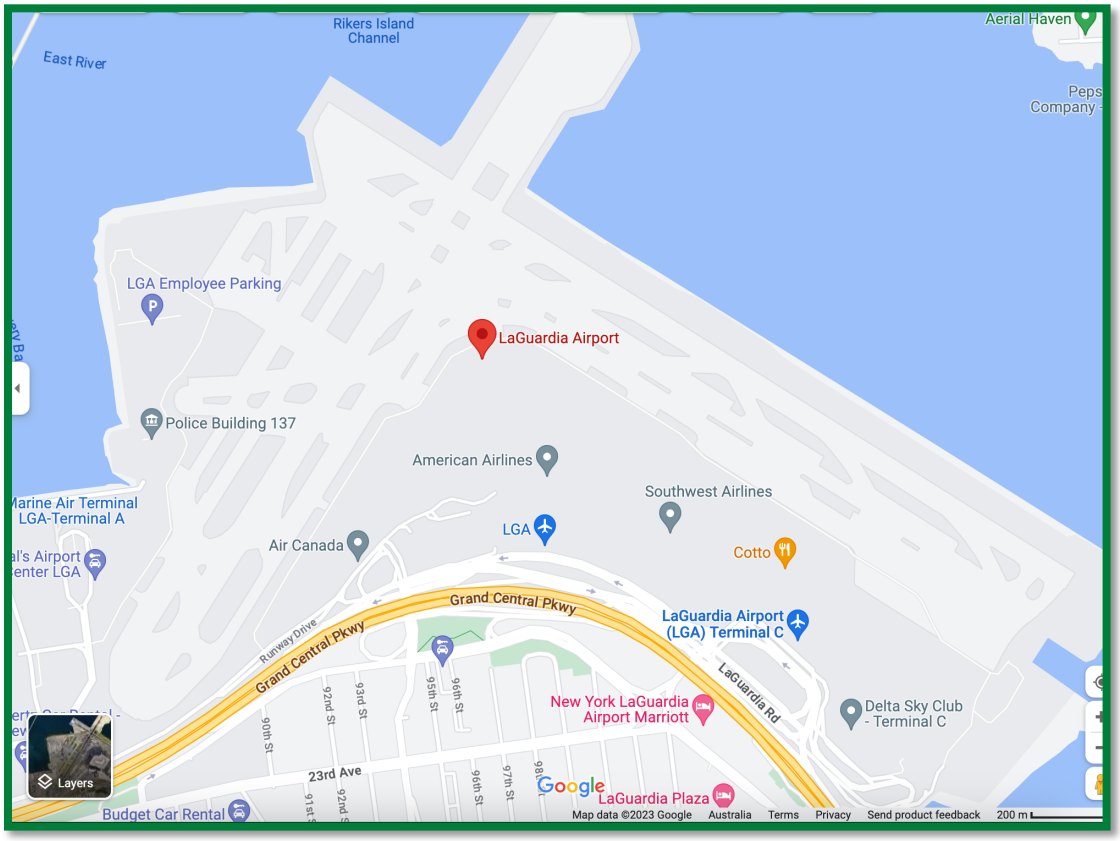

Figure 3-LaGuardia Airport

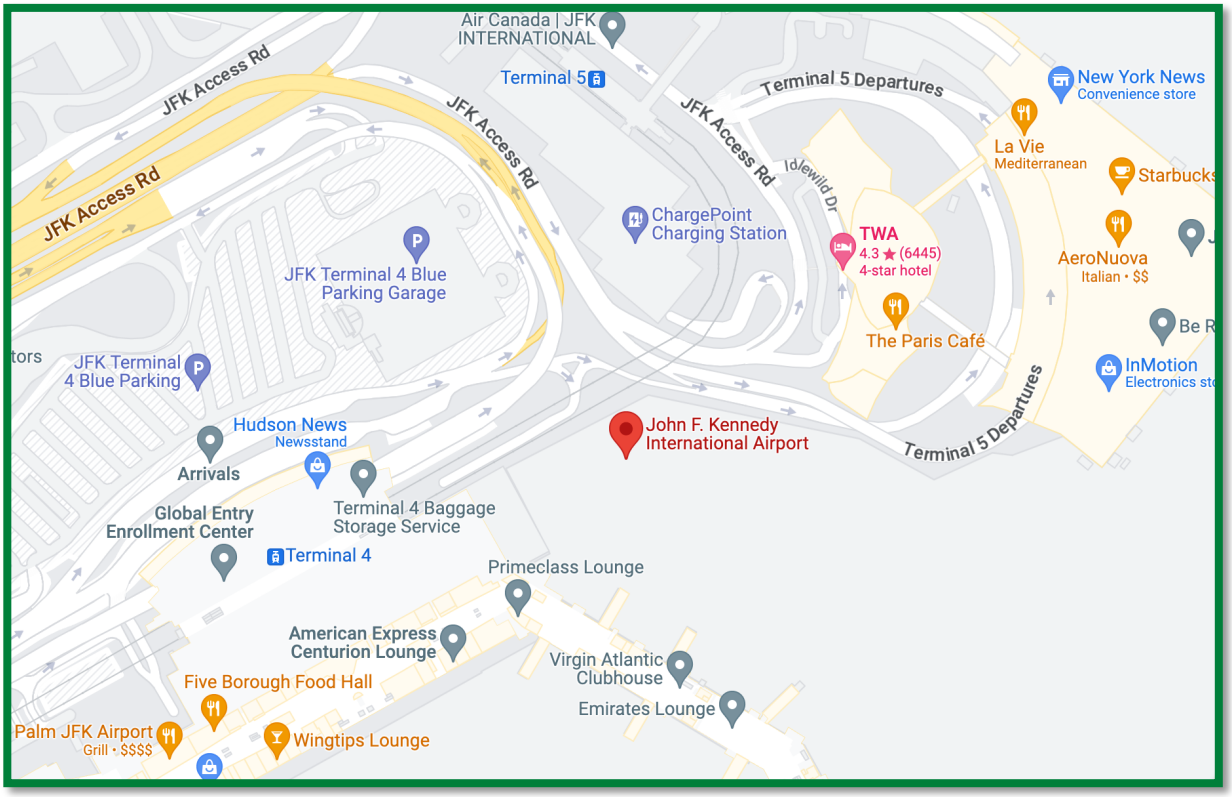

Figure 4-John F.kennedy Airport

Boro Zone:
This term needs to be more specific to determine whether it is inside or outside NYC without additional context. "Boro" could refer to one of the boroughs of NYC (such as Brooklyn, Queens, etc.), but it could also refer to areas outside the city.
To determine whether "Boro Zone" is inside or outside NYC, you must provide more information or context about its location.

Table 5-Boro Zone

LocationID	Borough	Zone	service_zone
2	Queens	Jamaica Bay	Boro Zone
3	Bronx	Allerton/Pelham G...	Boro Zone
5	Staten Island	Arden Heights	Boro Zone
6	Staten Island	Arrochar/Fort Wad...	Boro Zone
7	Queens	Astoria	Boro Zone
8	Queens	Astoria Park	Boro Zone
9	Queens	Auburndale	Boro Zone
10	Queens	Baisley Park	Boro Zone
11	Brooklyn	Bath Beach	Boro Zone
14	Brooklyn	Bay Ridge	Boro Zone
15	Queens	Bay Terrace/Fort ...	Boro Zone
16	Queens	Bayside	Boro Zone
17	Brooklyn	Bedford	Boro Zone
18	Bronx	Bedford Park	Boro Zone
19	Queens	Bellerose	Boro Zone
20	Bronx	Belmont	Boro Zone
21	Brooklyn	Bensonhurst East	Boro Zone
22	Brooklyn	Bensonhurst West	Boro Zone
23	Staten Island	Bloomfield/Emerso...	Boro Zone

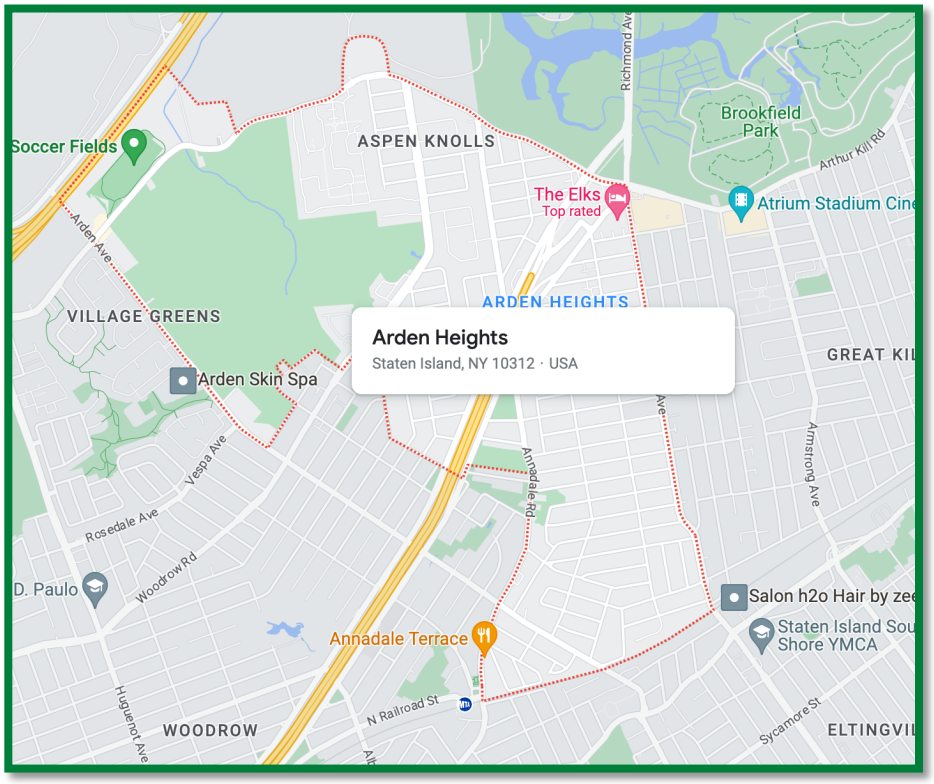

Figure 5-Arden Heights

Upon inspecting the data, we've discerned that the service zone at EWR and Airport encompasses the highways outside of New York City, while other zones comprise the highways within New York City itself.

Speed limit:
Speed limit within the city = 72 km per hour
Speed limit of highways outside cities = 89 km per hour

```
# First merge for PULocationID
merged_green_clean = green_clean.join(zone_df, green_clean["PULocationID"] == zone_df["LocationID"], "left").\
    withColumnRenamed("Borough", "Borough_pu").\
    withColumnRenamed("service_zone", "service_zone_pu").\
    withColumnRenamed("Zone", "Zone_pu")

# Drop the additional LocationID
merged_green_clean = merged_green_clean.drop("LocationID")

# Second merge for DOLocationID
merged_green_clean = merged_green_clean.join(zone_df, merged_green_clean["DOLocationID"] == zone_df["LocationID"], "left").\
    withColumnRenamed("Borough", "Borough_do").\
    withColumnRenamed("service_zone", "service_zone_do").\
    withColumnRenamed("Zone", "Zone_do")

# Drop the additional LocationID
merged_green_clean = merged_green_clean.drop("LocationID")


# Define the conditions to set the "airport" column
airport_condition = (
    (col("service_zone_pu") == "EWR") |
    (col("service_zone_pu") == "Airports") |
    (col("service_zone_do") == "EWR") |
    (col("service_zone_do") == "Airports")
)

# Create the "airport" column based on the conditions
merged_green_clean = merged_green_clean.withColumn("airport", when(airport_condition, 1).otherwise(0))
```

We combine our yellow and green dataframes with zone dataframes to create a new column named "airport," which will have a value of 1 when the service zone is either EWR or Airport. This allows us to filter the database based on permissible speed limits.

```
# Subset rows based on the conditions
merged_green_clean = merged_green_clean.filter(
    (col("airport") == 1) & (col("speed") < Highways_outside_cities) |
    (col("airport") == 0) & (col("speed") < Within_city_limits)
)
```

Table 6-Last seven columns of "merged_green_clean":

Borough_pu	Zone_pu	service_zone_pu	Borough_do	Zone_do	service_zone_do	airport
Brooklyn	Williamsburg (North Side)	Boro Zone	Queens	JFK Airport	Airports	1
Queens	Forest Hills	Boro Zone	Queens	JFK Airport	Airports	1
Queens	Forest Hills	Boro Zone	Queens	JFK Airport	Airports	1
Queens	Old Astoria	Boro Zone	Queens	LaGuardia Airport	Airports	1
Queens	Jackson Heights	Boro Zone	Queens	LaGuardia Airport	Airports	1
Queens	Woodside	Boro Zone	Queens	LaGuardia Airport	Airports	1
Queens	Elmhurst	Boro Zone	Queens	JFK Airport	Airports	1

e. Trips that are travelling too short or too long (duration wise)

Table 7-Trip duration summary

summary	trip_duration_sec
count	657030191
mean	1008.7557058941908
stddev	11794.146474767647
min	1
max	45466304

Upon analysing the summary of "trip_duration_sec" it became evident that the maximum trip duration is 45,466,304 seconds, which equates to over 12 hours. This duration is deemed unrealistic, given that taxi drivers typically cannot work more than ten consecutive hours and must allow an 8-hour rest period. As a result, we have refined our data filtering criteria to encompass trips lasting between 2 minutes and 10 hours.

f. Trips that are travelling too short or too long (distance wise)

Table 8-Trip distance summary

summary	trip_distance
count	64674304
mean	3.040716150574546
stddev	3.2683284831357056
min	0.01
max	391.09

After reviewing the summary of "trip_distance," it became clear that the maximum recorded trip distance is 391 miles, which appears to be implausible. To gain more informative insights, we created plots depicting the frequency of trips based on trip distance.

⚠ ***issues:*** After completing the first part of our work, we encountered an issue when exporting the notebook. It became enormous and impossible to download due to the abundance of commands, tables, and plots. To address this problem, we removed unnecessary commands, including some of the plots from the notebook to reduce its size, making it downloadable.

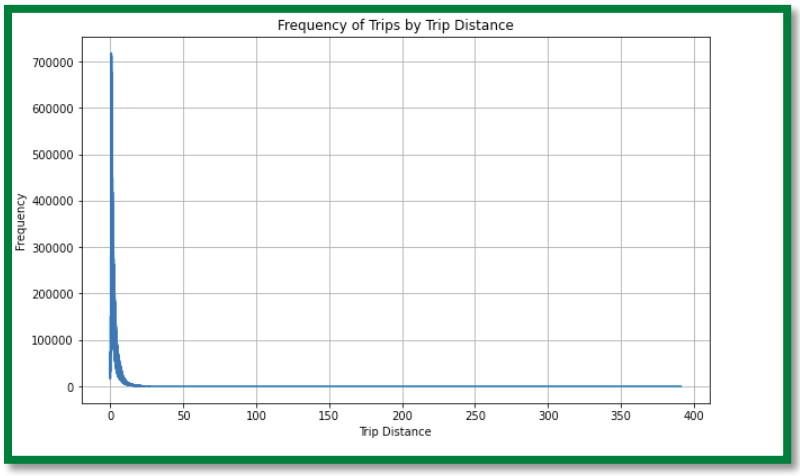

Figure 6-Frequency of Trips by Trip Distance plot

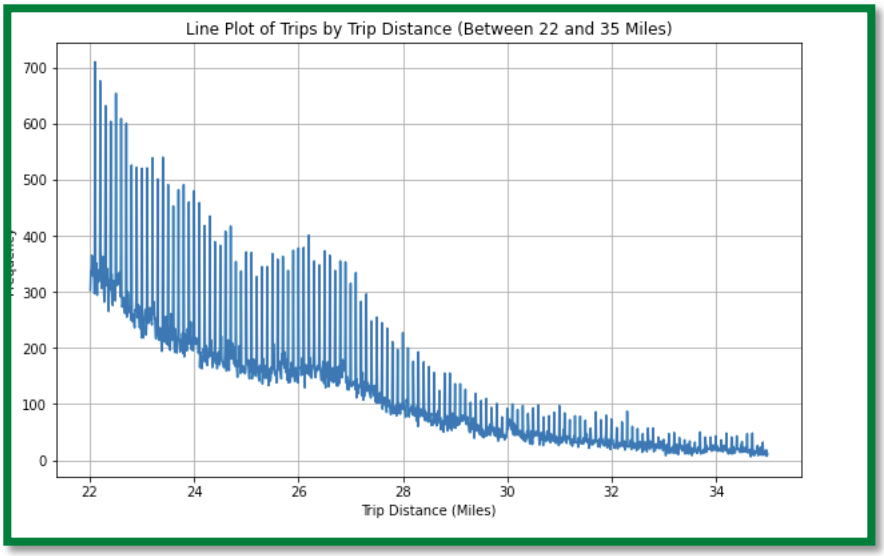

Figure 7-Frequency of Trips by Trip Distance plot (Between 22 and 35 miles)

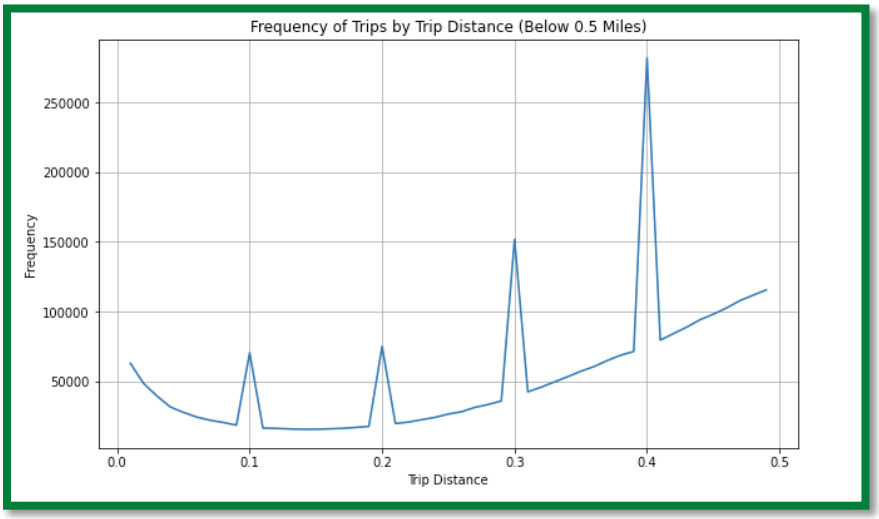

Figure 8-Frequency of Trips by Trip Distance plot

At the end we filter out our datafrems by trip distance between 0.5 and 35 miles.

we apply a filtering process to our data frames, focusing on trips ranging from 0.5 to 35 miles.

```
merged_green_clean = merged_green_clean.filter((col("trip_distance") > 0.5) & (col("trip_distance") <= 35))
```

g. Removing unrealistic passenger count

```
merged_yellow_clean = merged_yellow_clean.filter((col("passenger_count") <= 4))

merged_green_clean = merged_green_clean.filter((col("passenger_count") <= 4))
```

Given that the legal maximum passenger capacity for a standard taxicab is four, we have refined our data frames to include only records conforming to this passenger limit.

h. removing unrealistic fare

Table 9-Fare amount

```
+-------+-----------------+
|summary|      fare_amount|
+-------+-----------------+
|  count|         56595100|
|   mean|12.81510297923337|
| stddev|9.685457171022586|
|    min|           -499.0|
|    max|           8011.5|
+-------+-----------------+
```

Upon examining the "fare_amount" summary, it became evident that the data contains negative values for this column. Consequently, we refined the dataset by exclusively retaining positive fare values.

4.1.2 Final table

```
# Get the column names of merged_green_clean and merged_yellow_clean
green_columns = set(merged_green_clean.columns)
yellow_columns = set(merged_yellow_clean.columns)

# Find columns that are in merged_green_clean but not in merged_yellow_clean
green_not_in_yellow = green_columns - yellow_columns

# Find columns that are in merged_yellow_clean but not in merged_green_clean
yellow_not_in_green = yellow_columns - green_columns

# Print the results
print("Columns in merged_green_clean but not in merged_yellow_clean:", green_not_in_yellow)
print("Columns in merged_yellow_clean but not in merged_green_clean:", yellow_not_in_green)

Columns in merged_green_clean but not in merged_yellow_clean: {'trip_type', 'ehail_fee'}
Columns in merged_yellow_clean but not in merged_green_clean: {'airport_fee'}
```

We scrutinized the attributes within 'merged_yellow_clean' and 'merged_green_clean' to detect disparities. Notably, we discerned that the attributes "trip_type," "ehail_fee," and "airport_fee" were exclusive to each respective dataset.
To establish uniformity, we introduced the absent columns into each dataset, initializing them with a value of 0. Following this harmonization, the schema of both datasets became identical, enabling a seamless merger. Subsequently, we introduced a 'color' column to each dataset, denoting '1' for yellow and '0' for green, facilitating their subsequent merging.

Table 10-Final merged dataset

	VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	store_and_fwd_flag	RatecodeID	PULocationID	DOLocationID	passenger_count
1	2	2015-05-01T00:07:45.000+0000	2015-05-01T00:12:50.000+0000	N	1	255	112	1
2	2	2015-05-01T00:26:23.000+0000	2015-05-01T00:55:12.000+0000	N	1	255	13	1
3	2	2015-05-01T00:20:16.000+0000	2015-05-01T00:30:37.000+0000	N	1	80	225	1
4	2	2015-05-01T00:37:21.000+0000	2015-05-01T00:50:00.000+0000	N	1	37	112	1
5	2	2015-05-01T00:06:15.000+0000	2015-05-01T00:31:20.000+0000	N	1	255	164	1

	trip_distance	fare_amount	extra	mta_tax	tip_amount	tolls_amount	ehail_fee	improvement_surcharge	total_amount	payment_type	trip_type
1	1.09	5.5	0.5	0.5	1	0	null	0.3	7.8	1	1
2	5.73	22	0.5	0.5	4.66	0	null	0.3	27.96	1	1
3	2.35	9.5	0.5	0.5	0	0	null	0.3	10.8	2	1
4	2.68	11	0.5	0.5	3.08	0	null	0.3	15.38	1	1
5	4.74	19.5	0.5	0.5	0	0	null	0.3	20.8	2	1

congestion_surcharge	year	weekday	hour	month	day_of_year	trip_duration_sec	trip_distance_km	trip_duration_min	speed
null	2015	6	0	5	121	305	1.7541806000000002	5.083333333333333	20.70508249180328
null	2015	6	0	5	121	1729	9.2215182	28.816666666666666	19.200384916136496
null	2015	6	0	5	121	621	3.781949	10.35	21.92434202898551
null	2015	6	0	5	121	759	4.3130312	12.65	20.457064980237153
null	2015	6	0	5	121	1505	7.628271600000001	25.083333333333332	18.247028411960137

Borough_pu	Zone_pu	service_zone_pu	Borough_do	Zone_do	service_zone_do	airport	airport_fee	colour
Brooklyn	Williamsburg (North Side)	Boro Zone	Brooklyn	Greenpoint	Boro Zone	0	0	0
Brooklyn	Williamsburg (North Side)	Boro Zone	Manhattan	Battery Park City	Yellow Zone	0	0	0
Brooklyn	East Williamsburg	Boro Zone	Brooklyn	Stuyvesant Heights	Boro Zone	0	0	0
Brooklyn	Bushwick South	Boro Zone	Brooklyn	Greenpoint	Boro Zone	0	0	0
Brooklyn	Williamsburg (North Side)	Boro Zone	Manhattan	Midtown South	Yellow Zone	0	0	0

We stored the merged file as "combined_df" in the Parquet format within the DBFS to efficiently utilise it in other notebooks.

Export the combined data into a parquet file in DBFS

```
1
2  # Assuming you have already imported necessary libraries and created the 'combined_df' DataFrame
3
4  parquet_path = "/dbfs/mnt/bde2/combined_df"  # Replace with your desired path
5
6  try:
7      combined_df.write.parquet(parquet_path)
8      print("Parquet file saved successfully.")
9  except Exception as e:
10     print(f"An error occurred while saving the Parquet file: {str(e)}")
11
12
```

▸ (2) Spark Jobs

```
Parquet file saved successfully.
```

4.2 Business Questions (Only use SparkSQL)

1. *For each year and month*
   a. *What was the total number of trips?*
   b. *Which day of week (e.g. monday, tuesday, etc..) had the most trips?*
   c. *Which hour of the day had the most trips?*
   d. *What was the average number of passengers?*
   e. *What was the average amount paid per trip (using total_amount)?*
   f. *What was the average amount paid per passenger (using total_amount)?*

	year_month	day_of_week	hour_of_day	total_trips	avg_passenger_count	avg_total_amount	avg_amount_per_passenger	total_trips_year_month
1	2015-01	Friday	19	148571	1.3233740097327205	15.067663339423193	13.121540763193343	12126713
2	2015-02	Friday	19	123224	1.2999091086152048	16.060129844840827	14.102754592795597	11922556
3	2015-03	Sunday	0	142680	1.3880081300813008	15.815984861229863	13.392446590186927	12852284
4	2015-04	Thursday	19	141749	1.289927971273166	16.216791017929477	14.283314440862487	12604849
5	2015-05	Friday	19	141199	1.3297473778142905	16.142677497723877	13.997675542998412	12809867
6	2015-06	Tuesday	19	123466	1.281705084800674	15.863494322324714	14.023058554873375	11935923
7	2015-07	Wednesday	19	124139	1.291809987191777	15.84360942169782	13.944513502881327	11201027

⚠ ***issues:*** In our efforts to establish a lasting reference for our final data frame, we aimed to create a permanent table. This would have enabled us to employ it in other notebooks for running SQL queries. However, a challenge emerged as tables in Databricks are tied to a cluster. Since we are utilizing Databricks

Community Edition and setting up a new cluster each time, our table would become inaccessible or "lost." We decided to save it as a Parquet file in the Databricks File System (DBFS) and load it as a temporary view whenever a new cluster is initiated.

2. *For each taxi colour (yellow and green):*
   a. *What was the average, median, minimum and maximum trip duration in minutes (with 2 decimals, eg. 90 seconds = 1.50 min)?*
   b. *What was the average, median, minimum and maximum trip distance in **km**?*
   c. *What was the average, median, minimum and maximum speed in **km per hour**?*

color	average_duration	median_duration	max_duration	minimum_duration	average_distance	median_distance	maximum_distance	minimum_distance	average_speed	median_speed	max_speed	minimum_speed
green	14.38	10.97	600	2	3.058507112052920	2	35	0.51	20.309331534459300	18.448118526315800	88.9783685915493	0.0825740781935556
yellow	15.01	11.82	600	2	3.1517041585320800	1.8	35	0.51	18.922224825853100	16.62822598187310	88.9994506460945	0.08351228738588510

3. *For each taxi colour (yellow and green), each pair of pick up and drop off locations (use boroughs not the id), each month, each day of week and each hours:*
   a. *What was the total number of trips?*
   b. *What was the average distance?*
   c. *What was the average amount paid per trip (using total_amount)?*
   d. *What was the total amount paid (using total_amount)?*

	taxi_color	pickup_location	dropoff_location	month_name	day_of_week	hour	total_trips	avg_distance	avg_amount_paid	total_amount_paid
1	green	Bronx	Bronx	September	Friday	0	356	3.95	11.58	4124.26
2	green	Bronx	Brooklyn	September	Friday	0	4	25.97	58.73	234.92
3	green	Bronx	Manhattan	September	Friday	0	130	7.67	18.92	2459.92
4	green	Bronx	Queens	September	Friday	0	17	16.51	38.44	653.4
5	green	Bronx	Unknown	September	Friday	0	5	9.78	21.94	109.7
6	green	Brooklyn	Bronx	September	Friday	0	13	25.13	53.13	690.73
7	green	Brooklyn	Brooklyn	September	Friday	0	5806	4.11	13.19	76584.43

4. the percentage of trips where drivers received tips:

	total_trips	tipped_trips	tip_percentage
1	624243303	402744119	64.51717095954172

5. *The percentage of trips where the driver received tips of at least $5:*

	tipped_trips	tipped_trips_at_least_5	percentage_tipped_at_least_5
1	402744119	51008181	12.665158494840739

6. *Classify each trip into bins of durations:*
   a. *Under 5 Mins*
   b. *From 5 mins to 10 mins*
   c. *From 10 mins to 20 mins*
   d. *From 20 mins to 30 mins*
   e. *From 30 mins to 60 mins*
   f. *At least 60 mins*

Calculating Average speed km per hour and Average distance per dollar (km per $) for each bins:

	duration_bin	avg_speed_km_per_hour	avg_distance_per_dollar
1	Under 5 Mins	21.15	0.18
2	5 to 10 Mins	17.29	0.21
3	10 to 20 Mins	17.71	0.26
4	20 to 30 Mins	21.13	0.3
5	30 to 60 Mins	25.55	0.35
6	At least 60 Mins	21.94	0.4

7. A taxi driver can maximize their income with trips lasting less than 5 minutes, given that shorter durations result in a higher average distance covered per dollar earned, leading to greater earnings.

5. Machine Learning

a. Build a baseline model by using the answer of Part 2 Q3c (average paid) and calculate its RMSE.

```
# Average total amount from question 3c
avg_total_amount =17.41

#create a column with avg_total amount

baseline_pred = combined_df.withColumn ("pred_total_amount",lit(avg_total_amount))

from pyspark.ml.evaluation import RegressionEvaluator
# Evaluate the RMSE of the baseline model
evaluator = RegressionEvaluator (labelCol="total_amount", predictionCol="pred_total_amount", metricName="rmse")

baseline_rmse = evaluator.evaluate (baseline_pred)
print("Baseline RMSE:", baseline_rmse)
```

```
▸ (1) Spark Jobs
▸ baseline_pred: pyspark.sql.dataframe.DataFrame = [VendorID: long, tpep_pickup_datetime: timestamp ... 37 more fields]
Baseline RMSE: 194.4292058133119
```

The average payment was determined through SparkSQL, and a baseline model was constructed by using pyspark. The resulting RMSE (Root Mean Square Error) value for this model was 194.42.

*b.Use all data except **October/NovemberDecember 2022** to train and validate models*

- **Data Preparation**

  We retrieve data from the Databricks File System (DBFS) and choose our machine learning features.

```
# wanted columns
cols_list=[
  'weekday',
 'hour',
 'year',
 'month',
 'trip_distance_km',
 'trip_duration_min',
 'total_amount',
 'speed',
 'Zone_pu']
```

- **Pipeline**

This pipeline is used for preprocessing categorical and numerical features in a machine-learning workflow. Here's a short explanation of what each step does:

1. An empty list called "stages" is created to store the stages of the pipeline.
2. For each categorical column (cat_col) in the "cat_cols" list:
    - A StringIndexer is instantiated, which assigns a unique numerical index to each category in the column. The output is stored as "{cat_col}_ind".
    - A OneHotEncoder is instantiated to convert the indexed categorical values into binary vectors. The output is stored as "{cat_col}_ohe".
    - Both StringIndexer and OneHotEncoder are added to the "stages" list.
3. A new list called "cat_cols_ohe" is created by adding the "_ohe" suffix to each element in the "cat_cols" list. This new list contains the names of the one-hot-encoded categorical columns.
4. A VectorAssembler is instantiated. It takes the one-hot-encoded categorical columns ("cat_cols_ohe") and numerical columns ("num_cols") as input and assembles them into a single feature vector. The output feature vector is stored as "features".
5. The VectorAssembler is added to the "stages" list.
6. Finally, a Pipeline is created with the "stages" list, which defines the sequence of data transformations. This pipeline can preprocess the data before training a machine learning model.

In summary, this pipeline prepares the data by indexing and one-hot encoding categorical features and assembling them with numerical features into a single feature vector for machine learning model input.

- **Model**

Initially, we evaluate the models using a subset of data with limited rows. Following that, we train the models using data spanning from January 2020 to November 2022 and subsequently test their performance on data from October, November, and December 2022.

1.Linear Regression

The RMSE for this model was 6.059, demonstrating a significant improvement over the baseline model's performance.

2.Decision Tree

While this model performed exceptionally well with a small dataset, our attempts to scale it up were consistently plagued by clustering determination and notebook detachment issues. Even after reducing the feature set, these problems persisted. Given our constraint of utilizing Databricks Community, which has its limitations, employing more complex modules beyond linear regression on this extensive dataset may not be prudent.

- **Best model**

Of all the models we explored, linear regression emerged as the standout performer. Due to limitations with Databricks Community and our extensive dataset, it delivered superior speed and significantly boosted accuracy compared to the baseline model.

Appendix:

Challenges encountered while executing the Random Forest Model: