

University of Technology Sydney
Big Data Engineering (94693)
Autumn 2021

Assignment 1

By

Hnin Pwint Tin (13738339)

Master of Data Science and Innovation

24 April 2021

Background

Our client is New York City Taxi and Limousine Commission (TLC). TLC is the agency responsible for licencing and regulating New York City' taxi cabs since 1971. There are two Taxi Companies under the control of TLC. Our client wants to automate the process of taxi fare prediction based on the data from the two taxi companies were provided to us.

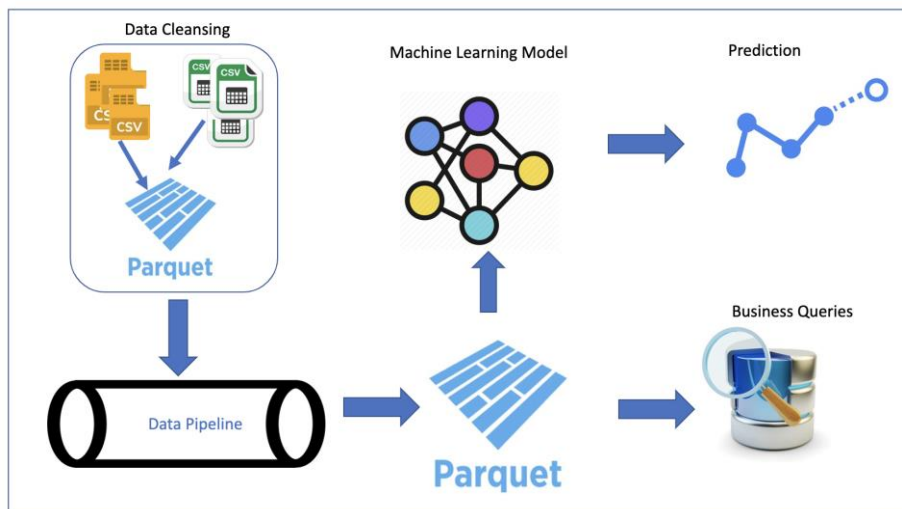
Objective and Audience

The project is to build the machine learning algorithm on large dataset using Spark to help the client with taxi-fare prediction. The dataset contains millions of historic records with an array of attributes captured such as pick-up and drop-off dates/times, locations, trip distances, itemized fares, rate types, payment types, and driver-reported passenger counts.

This document is prepared for the readers who would be maintaining and improving the project of taxi fare prediction model in future.

Overview of the Project (High-level)

The diagram below represents the high-level overview of the project and design.



Technical Requirement

All the required packages will be installed, and environment will be created inside the docker container. The version of spark and Hadoop used are as followed. The details can be referred in Docker file.

- Spark_version = 2.4.5
- Hadoop_version = 3.0.0

System Features Used During the Development

- System Type: 64-bit Window operating system
- RAM: 16 GB (13 GB usable)

- Processor: 2.10 GHz

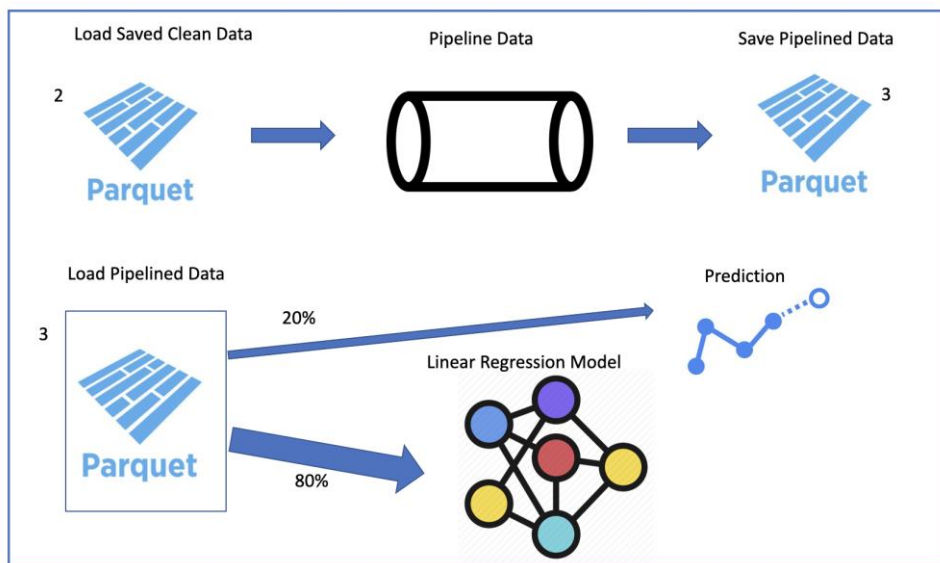
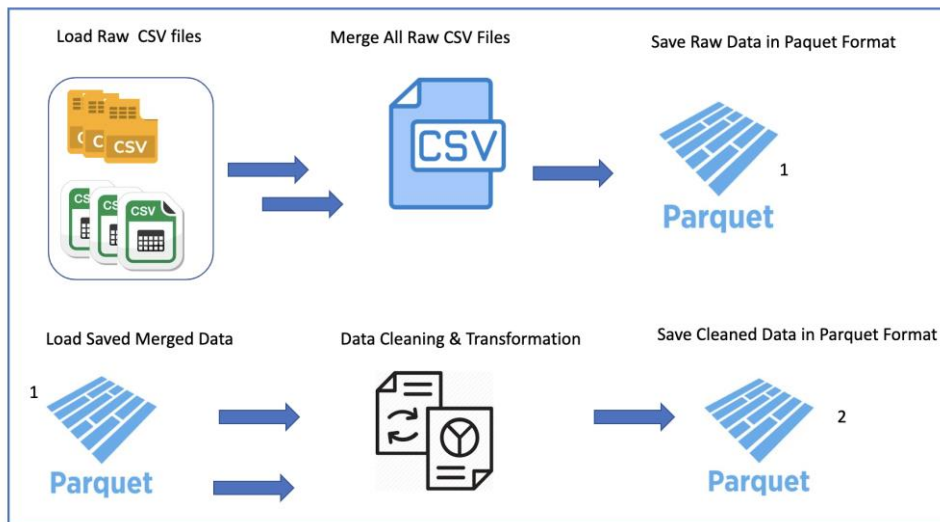
Original Data Size

- 48 CSV files in total -1 CSV file for each month of the year for Green and Yellow Company

| Taxi-Company | Year | CSV File-Size (KB) |
|--------------|------|--------------------------|
| Green | 2017 | 1,026,801 |
| Green | 2018 | 767,990 |
| Yellow | 2017 | 9,772,455 |
| Yellow | 2018 | 8,818,648 |
| Total | | 20,385,894 KB (20.38 GB) |

Overview of Data Processing until Model Prediction (Mid-Level)

The overview of the data processing and modelling workflow is demonstrated in the below two diagrams.



Data Cleaning and Preparation

Step –1: Load the Raw Data and Save in Optimised Compressed Format

Once the raw data csv files were downloaded, the files were merged and saved in the optimised file format i.e parquet as our choice. The reason of doing it is discussed in detail in the session “Problem Encountered”. Before merging the data, the following three steps were performed.

1. **Add a new column:** “taxi_company” is added with the respective values to differentiate green or yellow taxi company. This feature would add value in model training and making business queries.
2. **Uniform column naming:** the wording of column names for taxi pick-up and drop-off times are different in two taxi companies. To achieve the uniformity, the column names are renamed to “pick_up_datetime” and “drop_off_datetime”. So that spark can Union the two data sets of two different companies.
3. **Drop different columns:** The data set on Green Taxi Company has two extra columns - “ehail_fees” and “trip_type” which do not exist in Yellow Taxi Company dataset. Those two columns are dropped from the Green Taxi dataset.

After the first clean-up was done all number of columns and naming are uniform, the datasets were merged and saved in the Paquet format.

Step – 2: Year Filtering and Extracting features

Actual data preparation is going to perform in Step 2. The data saved in the optimised format in the step 1 would be used from now on.

1. **Drop the rows with invalid years in pickup-datetime:** The data downloaded is supposed to be within the year 2017 and 2018. However, the years such as 2042, 2037, 2053, 2003, 2084 were identified during the SQL queries. The observations with the invalid years were filtered out and omitted since the first stage of the project.
2. **Extract weekday and hour from pickup-datetime:** As new features, hour and weekdays as number as well as abbreviation (Mon, Tue) are extracted from the pick-up datetime. Weekday abbreviations might be useful for answering business queries and weekday number might be useful for modelling as integer feature.
3. **Filter the data range:**
 - 1) trip_distance is set between 0 and 100 miles by referring to the average taxi trip reported by Schaller Consulting [here](#) was 5 miles.
 - 2) Passenger_count: set between 0 to 10 which would allow the limousines service also.
 - 3) Fare_amount, total_amount, tip_amount, mta_tax : set 0 to the lower limit as the dataset contains negative values in those columns and 200 as the upper limit.

Step – 3: Adding New Features with User defined functions.

The following features were created and added into the dataset. The values were calculated through UDF.

1. **Trip Duration in Second (trip_duration_sec):** The duration is the difference between drop-off time and pick-up time. This feature is prepared for model training purpose as well as for business queries.

2. **Speed in Km per hour (*speed_km_hr*):** The speed is calculated using the original column – trip_distance and trip_duration_sec created in No 1). This is also meant for business queries and to save the calculation time when executing SQL.
3. **Duration range in minute (*trip_duration_range_mins*):** for the discretisation purpose of the trip duration continuous values in seconds already calculated in No 1). The duration seconds are binned into five categories - < mins, 5-10 mins, 10-20 mins, 20-30 mins and >30 mins.

Step – 4: Save the cleaned data set in Paquet format.

The new features are created with the interest of modelling and business queries in the steps 1 through 3. Step 4 is to save the cleaned dataset with the features we are interested in to serve the different purpose. Following is the list of features in the cleaned dataset including the new features highlighted in colour which was saved in Parquet format.

| New Features | Original Features | | |
|---------------------------------|-------------------|-----------------------|-----------------|
| taxi_company | Drop_off_datetime | Payment_type | MTA_tax |
| speed_km_hr | Pickup_datetime | Total_amount | Fare_amount |
| trip_duration_range_mins | VendorID | Tolls_amount | Passenger_count |
| trip_duration_sec | Trip_distance | Store_and_fwd_flag | Extra |
| Weekday_num | RateCodeID | Tip_amount | PULocationID |
| Weekday_abbre | Trip_type | Improvement_surcharge | DOLocationID |

Choice of File Format

In this project Apache Parquet format is used to save the raw data as well as processed data. Apache Parquet is a file format designed to support fast data processing for complex data. It has several characteristics. Unlike csv format, Parquet is column-based storage. It means that the values of each column are stored next to each other. When querying the columnar storage, the non-relevant columns can be skipped so that query within the subset of the required columns can be done quickly. As a result, aggregation queries are less time consuming compared to row-oriented CSV files. This storage nature of parquet file is the reason to choose for and suit our business query purpose and data processing performance.

The column-oriented storage feature not only works for the query efficiency but also has the advantage over data compression. When data is hosted in cloud platform like Amazon Athena - for instance the charges for scanning 27 GB of CSV data in Athena will cost only for the size of 0.22 GB if the data is saved in compressed columnar format.

Following is the list of file sizes achieved after converting from CSV to Parquet format.

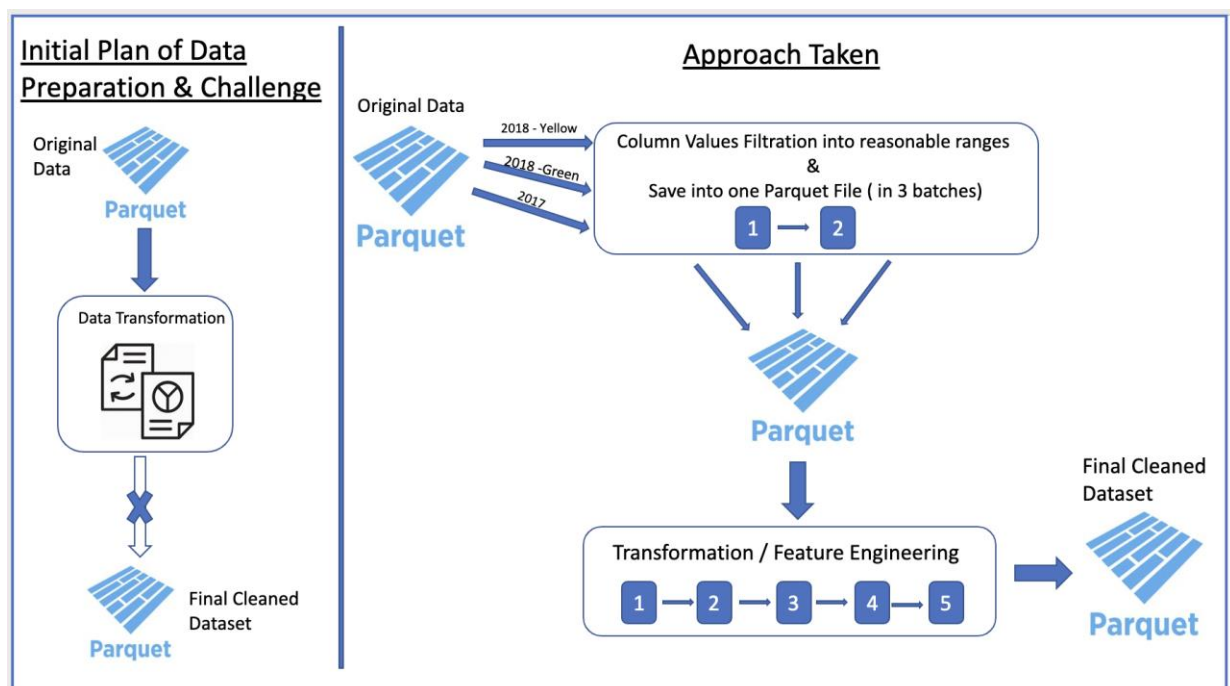
| File Format | File | File Size (GB) |
|-------------|-------------------------------------|----------------|
| CSV | Raw Data (2017-2018 Green + Yellow) | 20.38 |
| Parquet | Raw Data (2017-2018 Green + Yellow) | 4.10 |
| Parquet | Cleaned Data | 3.53 |
| Parquet | Cleaned Data + Engineered Features. | 3.73 |

The Summary of Data Cleaning Steps (Step 1 - 4)

The following diagram presents the summary of data clean-up process. It also explains that how the processes performed was different from the initial plan when the CPU and memory capacity could not handle the entire load of data transformation and saving in Spark.

The approach taken was illustrated on the right panel. As the raw data set contained the records with dates of invalid years (2032, 2003, 2029, 2053, 2021), the year filtration was done at the very first step. The data was split into 3 batches by year and company to perform the column value filtration where the values in many columns are beyond the reasonable ranges for instance fare_amount below zero. When the data filtration was done in three batches, they were saved into Parquet format in three batches with “append” mode. The result of this process is the first version of cleaned dataset. It would be used in next step of feature engineering to create the new features.

In the next step, the whole cleaned dataset in Parquet format was used to carry out the series of new columns creation in one go. Finally, the cleaned and engineered dataset was saved into the disk in Parquet format. This second and final version of cleaned dataset was ready for building pipeline and running business queries.



Step – 5: Selection of features and Pipeline prior Modelling

1. The data used in the modelling are observation in year 2018 in the months of
 - October
 - November
 - December
2. Basically, the features are selected based on which were known at the time the taxi started the pick-up. The features which are unknown before the taxi reached the destination such as tip-amount and tax are not considered in the model training. The selected features in the modelling are described in the table below.

| Categorical Features | Numerical Features | Target Feature |
|---------------------------------|--------------------|----------------|
| trip_duration_range_mins | Tolls_amount | Total_amount |
| Taxi_company | Passenger_count | |
| Weekday_num | Trip_distance | |
| PULocationID | pickup_hour | |
| RateCodeID | | |
| store_and_fwd_flag | | |
| | | |

3. The pipeline is created to index the categorical variables and do one-hot encoding. The data with selected features are applied into the pipeline to do transformation.

Step – 6: Modelling & Accuracy Measures

The target variable “total_amount” is continuous in nature. The problem is considered as regression problem. There are many algorithms offered by Spark to solve the regression problems.

In our approach, there are only 10 features selected to use in the model training. Out of 10 features, 6 are categorical variables and 4 are numeric features. The problem can be considered simple and not complexed. The linear regression algorithm would be good enough to train the model and learn from the data. It is not worth trying out the sophisticated algorithm such as neural network, xgboost and random forest which need high computation power.

To start with, the transformed data first 21 months from the pipeline was separated for into for training and the rest last 3 months of data for testing. The RMSE scores are monitored along the way every after increment to the train data size. Following table lists the result of model prediction accuracy in terms of Root Mean Square Error (RMSE).

| Model | Data Size | RMSE - Score |
|-------------------|---|--------------|
| Linear Regression | Train Data 100% of first 21 months Data | 0.461471 |
| | Prediction on last 3 months Test Data | 0.470649 |

The prediction accuracy of the linear regression was seen good. The RMSE of the prediction was 0.46 when training on 21 months of data and 0.47 on last 3 months of test data. It is consistent and not-overfitting. It can be concluded that Linear Regression model performed extremely well and any further trying on different algorithms or retraining the model were not required.

Linear Regression

```
[116] 1 from pyspark.ml.regression import LinearRegression
      2
      3 lr = LinearRegression(featuresCol='features', labelCol='total_amount', maxIter=10, regParam=0.3, elasticNetParam=0.8)
```

```
[117] 1 lr_model = lr.fit(pipelined_cleaned_21month_df)
```

```
▶ 1 lr_predictions_train = lr_model.transform(pipelined_cleaned_21month_df)
  2 lr_predictions_train.select("prediction", "total_amount", "features").show(20)
```

```
┌-----+-----+-----+
| prediction | total_amount | features |
|-----+-----+-----+
| 9.08329902236744 | 8.76 | (298,[0,6,8,11,18... |
| 21.428128581429874 | 21.8 | (298,[0,6,9,11,47... |
| 27.524592744206984 | 27.88 | (298,[0,6,9,11,17... |
| 17.293518928586572 | 17.76 | (298,[0,6,9,11,18... |
| 15.720854920704756 | 15.96 | (298,[0,6,7,11,22... |
| 15.079499754350337 | 15.36 | (298,[0,6,7,11,30... |
| 18.197390130852703 | 18.5 | (298,[0,6,7,11,37... |
| 9.62444741352145 | 9.3 | (298,[0,6,8,11,60... |
| 37.65650017733958 | 39.06 | (298,[0,6,9,11,24... |
| 9.610622671881897 | 9.36 | (298,[0,6,8,11,16... |
| 15.845376023500343 | 15.8 | (298,[0,6,7,11,14... |
| 13.793631385913939 | 13.8 | (298,[0,6,7,11,27... |
| 21.51537875119506 | 21.96 | (298,[0,6,9,11,38... |
| 15.230764921294728 | 15.36 | (298,[0,6,7,11,30... |
| 14.614366723847542 | 14.65 | (298,[0,6,7,11,50... |
| 13.830865861231329 | 13.8 | (298,[0,6,7,11,19... |
| 10.136956175194163 | 9.8 | (298,[0,6,7,11,61... |
| 9.109262286341595 | 8.75 | (298,[0,6,8,11,18... |
| 8.683516324771663 | 8.3 | (298,[0,6,8,11,12... |
| 9.717533685040546 | 9.3 | (298,[0,6,8,11,37... |
└-----+-----+-----+
only showing top 20 rows
```

```
[119] 1 from pyspark.ml.evaluation import RegressionEvaluator
      2 lr_evaluator = RegressionEvaluator(predictionCol="prediction", \
      3                                 labelCol="total_amount", metricName="rmse")
      4 rmse = lr_evaluator.evaluate(lr_predictions_train)
      5 print("Root Mean Squared Error (RMSE) on Train data = %g" % rmse)
```

```
Root Mean Squared Error (RMSE) on Train data = 0.461471
```



```
1 lr_predictions_test = lr_model.transform(pipelined_cleaned_3month_df)
2 lr_predictions_test.select("prediction", "total_amount", "features").show(20)
```

```
3 +-----+-----+-----+
| prediction | total_amount | features |
+-----+-----+-----+
| 12.997035248889294 | 12.8 | (288,[0,7,11,77,2... |
| 8.278218108661902 | 7.8 | (288,[0,8,11,54,2... |
| 15.831465154675325 | 15.8 | (288,[0,7,11,113,... |
| 19.933035724450573 | 19.8 | (288,[0,7,11,92,2... |
| 44.23234117573518 | 44.3 | (288,[0,9,11,113,... |
| 15.062966643818646 | 14.8 | (288,[0,7,11,92,2... |
| 35.146524721841736 | 35.3 | (288,[0,10,11,113... |
| 12.997035248889294 | 12.8 | (288,[0,7,11,77,2... |
| 8.278218108661902 | 7.8 | (288,[0,8,11,54,2... |
| 15.831465154675325 | 15.8 | (288,[0,7,11,113,... |
| 19.933035724450573 | 19.8 | (288,[0,7,11,92,2... |
| 44.23234117573518 | 44.3 | (288,[0,9,11,113,... |
| 15.062966643818646 | 14.8 | (288,[0,7,11,92,2... |
| 35.146524721841736 | 35.3 | (288,[0,10,11,113... |
| 12.997035248889294 | 12.8 | (288,[0,7,11,77,2... |
| 8.278218108661902 | 7.8 | (288,[0,8,11,54,2... |
| 15.831465154675325 | 15.8 | (288,[0,7,11,113,... |
| 19.933035724450573 | 19.8 | (288,[0,7,11,92,2... |
| 44.23234117573518 | 44.3 | (288,[0,9,11,113,... |
| 15.062966643818646 | 14.8 | (288,[0,7,11,92,2... |
+-----+-----+-----+
only showing top 20 rows
```

```
2] 1 from pyspark.ml.evaluation import RegressionEvaluator
2 lr_evaluator = RegressionEvaluator(predictionCol="prediction", \
3 labelCol="total_amount", metricName="rmse")
4 rmse = lr_evaluator.evaluate(lr_predictions_test)
5 print("Root Mean Squared Error (RMSE) on Test data = %g" % rmse)
```

Root Mean Squared Error (RMSE) on Test data = 0.470649

Problems Encountered

The very first problem was when building the pipeline – data transformation was not completed even after 45 mins. It was unusually long without showing any error. This was where the development was stuck at the beginning.

```
Pipeline_model = Pipeline.fit(df_cleaned)
```

List of things tried out:

1. **Configure the spark session in script:** Manually set the spark session for memory allocation according to the core in my PC as the screen shot followed. As the execution speed of data transformation was not seen improved. This is because the above setting is in client mode. It is explained in the spark configuration document [here](#). The config must not be set through the SparkConf directly in your application, because the driver JVM has already started at that point. Instead, it should be set through the --driver-

java-options command line option or in the default properties file. In my case, in the Docker file.

```
spark = SparkSession.builder \
    .appName('new_york_taxi_fare_prediction') \
    .master('local[3]') \
    .config("spark.executor.memory", "8g") \
    .config("spark.driver.memory", "8g") \
    .config("spark.executor.cores", "3") \
    .config("spark.cores.max", "3") \
    .getOrCreate()
```

2. **Increase the memory:** The Spark Configuration in Dockerfile was updated to increase the memory to 10 GB. `ENV SPARK_OPT="--driver-java-options=-Xmx12288M"` as my local system is 16GB RAM. After setting this, the processing of feature engineering tasks is two times faster than the config in the number 1. However, execution of data transformation through Pipeline was still taking peculiarly long. I imagined that data size I was trying to feed into the Pipeline could be the reason of taking execution time long when doing the transformation.
3. **Reduce the data size:** I tried with just 5% of train data which is 80 % of the entire dataset. Although the pipeline was running on 5% of 80% of the data, it was still unacceptably taking longer. The reason could be neither of memory nor data size. The culprit could be something else.
4. **Save the cleaned data in Apache Parquet format:** The data I was trying to process in the above three attempts were in spark data frame csv format. I had not saved the cleaned data in the optimised yet in this stage. The next thing I tried was to save the cleaned data in the parquet format with the intention of supplying the optimised format data into the pipeline. What I faced next was, although the execution of writing data was running for about 1 hour, the local directory was seen as empty, and nothing got written. The Spark is lazy in nature. The series of transformation in RDD were adding new columns, filtering and binning the column values. Spark analyse all the operations and create a DAG (Directed Acyclic Graph) before execution. When the action (fitting into the pipeline) was executed, DAG is triggered, and I assumed that it would be too much of workload for Spark to handle.
5. **Save the raw data in Apache Parquet format:** Another strategy I took in this step was to do saving the unprocessed version of raw merged data, instead of saving the cleaned version of data. At the end, the optimised format of raw merged datasets of two taxi companies was saved successfully within 20 minutes. When data preparation and cleansing were performed on Parquet format data, the processes ran faster and completed within one minute. It had reached the point where I can continue with data pipeline process.

By loading the data of the saved and optimised format, the data preparation jobs were performed efficiently and categorical encoding, string Indexing and Vector Assembler were able to run through the Spark ML Pipeline with ease.

Ideally, the cleaned data with engineered features should be saved in the optimised format. Due to the hardware limitation, I had to choose to save the data of raw data to continue to the building pipeline and modelling.

Lesson Learned from Experience:

1. **Processing Speed:** To work on the Big Data efficiently, data on optimised format is best way and helped to faster the processes of data cleaning, feature creation, data transformation. Later it should pay off in the querying time as well.
2. **Size:** The original raw data was the size of 19 GB. After converting to the Paquet format, the raw data had been compressed to the size of 4 GB.

Memory Issue when fitting the model.

The second problem I experienced was “Out of Memory Error” when fitting the model to the training data when the system does not have enough memory for the data it needs to process.


java.lang.OutOfMemoryError: GC Overhead limit exceeded.

```
at java.lang.Thread.run(Thread.java:746)
Caused by: java.lang.OutOfMemoryError: GC overhead limit exceeded
at org.apache.spark.ml.tree.impl.TreePoint$.org$apache$spark$ml$tree$impl$TreePoint$$labeledPointToTreePoint
(TreePoint.scala:89)
at org.apache.spark.ml.tree.impl.TreePoint$$anonfun$convertToTreeRDD$2.apply(TreePoint.scala:73)
at org.apache.spark.ml.tree.impl.TreePoint$$anonfun$convertToTreeRDD$2.apply(TreePoint.scala:73)
```

java.lang.OutOfMemoryError: Java heap space

```
at org.apache.spark.ml.util.Instrumentation$.anonfun$11.apply(Instrumentation.scala:185)
at scala.util.Try$.apply(Try.scala:192)
at org.apache.spark.ml.util.Instrumentation$.instrumented(Instrumentation.scala:185)
at org.apache.spark.ml.regression.DecisionTreeRegressor.train(DecisionTreeRegressor.scala:104)
at org.apache.spark.ml.regression.DecisionTreeRegressor.train(DecisionTreeRegressor.scala:47)
at org.apache.spark.ml.Predictor.fit(Predictor.scala:118)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
at java.lang.reflect.Method.invoke(Method.java:498)
at py4j.reflection.MethodInvoker.invoke(MethodInvoker.java:244)
at py4j.reflection.ReflectionEngine.invoke(ReflectionEngine.java:357)
at py4j.Gateway.invoke(Gateway.java:282)
at py4j.commands.AbstractCommand.invokeMethod(AbstractCommand.java:132)
at py4j.commands.CallCommand.execute(CallCommand.java:79)
at py4j.GatewayConnection.run(GatewayConnection.java:238)
at java.lang.Thread.run(Thread.java:748)
Caused by: java.lang.OutOfMemoryError: Java heap space
```

Java and Network Connection Error was experienced.



```
1 lr_model = lr.fit(train_data) #4:54

Streaming output truncated to the last 5000 lines.
Traceback (most recent call last):
  File "/content/gdrive/MyDrive/2021 - MDSI/BDE/spark-2.4.5-bin-hadoop2.7/python/lib/py4j-0.10.7-src.zip/py4j/java_
response = connection.send_command(command)
  File "/content/gdrive/MyDrive/2021 - MDSI/BDE/spark-2.4.5-bin-hadoop2.7/python/lib/py4j-0.10.7-src.zip/py4j/java_
    "Error while receiving", e, proto.ERROR_ON_RECEIVE)
py4j.protocol.Py4JNetworkError: Error while receiving

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/content/gdrive/MyDrive/2021 - MDSI/BDE/spark-2.4.5-bin-hadoop2.7/python/lib/py4j-0.10.7-src.zip/py4j/java_
    connection = self.deque.pop()
IndexError: pop from an empty deque

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
```

```

/usr/local/spark/python/lib/py4j-0.10.7-src.zip/py4j/java_gateway.py in start(self)
1077         "server ({0}:{1})".format(self.address, self.port)
1078         logger.exception(msg)
-> 1079         raise Py4JNetworkError(msg, e)
1080
1081     def _authenticate_connection(self):
Py4JNetworkError: An error occurred while trying to connect to the Java server (127.0.0.1:37961)

```

The strategy tried out were:

1. **Reduce the data size:** to 10%: By reducing the size of data and with the selection of a minimal features, the Linear Regression model was run smoothly.
2. **Increase the local device RAM.**
3. **Run on google Colab Cloud environment:** The extra work here is to set up and mimic the same environment used in local system to Google Colab. The following screen shot is how the environment is set up in Google Colab. In this attempt, the challenge here is to set up the environment multiple times whenever time-out occurs.
4. **Restart the spark session by restating Docker container.**

```

[6] 1 !apt-get install openjdk-8-jdk-headless -qq > /dev/null

[7] 1 !wget -q https://archive.apache.org/dist/spark/spark-2.4.5/spark-2.4.5-bin-hadoop2.
2
[8] 1 !tar xf spark-2.4.5-bin-hadoop2.7.tgz

[9] 1 !pip install -q findspark

[10] 1 ## Set the environment variable
2 os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
3 os.environ["SPARK_HOME"] = "/content/gdrive/MyDrive/2021 - MDSI/BDE/spark-2.4.5-bin-
1 import findspark
2 findspark.init()

```

Lesson Learned from Experience

Check the hardware ability of the existing development environment and learn to know the storage requirement for the raw data and intermediary processed-data output throughout the project. 80% of the effort in this project was invested in trying out many solutions to resolve the mostly memory issues.

Data Quality Issues Identified During SQL queries.

During the SQL queries for average amount paid per trip, some unreasonable amounts were noticed.

```

1 # Question 4.a.v What was the average amount paid per trip?
2 spark.sql( '''
3 SELECT
4     YEAR(pickup_datetime) year,
5     MONTH(pickup_datetime) month,
6     ROUND((SUM(total_amount)/COUNT(*)), 2) avg_amount_per_trip
7 FROM nyc_view
8 GROUP BY YEAR(pickup_datetime), MONTH(pickup_datetime)
9 ORDER BY 1,2
10 ''' ).show(24)

```

```

+---+---+---+
|year|month|avg_amount_per_trip|
+---+---+---+
|2017| 1 |          1.62 |
|2017| 2 |          1.66 |
|2017| 3 |          1.67 |
|2017| 4 |          1.64 |
|2017| 5 |          1.63 |
|2017| 6 |          1.6 |
|2017| 7 |          1.66 |

```

The data quality issues were discovered late.

- The values in *passenger_count* column in the raw data were as three-digit values.
- *rateCodeid* values are in float, which was supposed to be 1-6 categorical value according to data dictionary.
- Fare_amount was from 1-2

```
1 df_original_data = spark.read.parquet("/content/gdrive/MyDrive/2021 - MDSI/BDE/data/original_df.parquet")
```

```
1 df_original_data.show()
```

```

+---+---+---+---+---+---+---+---+---+---+
|fwd_flag|ratecodeid|pulocationid|dolocationid|passenger_count|trip_distance|fare_amount|extra_mta_tax|tip_amount|tol|
+---+---+---+---+---+---+---+---+---+---+
|2|1.26|1|N|43|237|2|6.5|0.5|0.5|
|3|21.71|1|N|236|262|1|62.5|0.5|0.5|
|2|1.40|1|N|161|164|2|10.5|0.5|0.5|
|2|2.30|1|N|113|230|1|9|0.5|0.5|
|1|3.40|1|N|230|193|1|13|0.5|0.5|
|1|19.50|2|N|132|166|1|52|0|0.5|
|1|17.36|1|N|132|182|2|47.5|0.5|0.5|
|2|1.00|1|N|100|170|1|8|0.5|0.5|
|2|2.04|1|N|162|141|1|8.5|0.5|0.5|
|2|2.52|1|N|141|75|2|9|0.5|0.5|
|2|1.69|1|N|170|246|2|8.5|0.5|0.5|
|2|1.17|1|N|230|142|1|7|0.5|0.5|
|2|.43|1|N|234|164|2|4|0.5|0.5|
|2|1.58|1|N|170|113|1|8|0.5|0.5|
|2|.42|1|N|114|113|2|4|0.5|0.5|
|4|1.10|1|N|164|161|2|6|0.5|0.5|
|3|.40|1|N|161|48|1|4.5|0.5|0.5|
|3|.30|1|N|230|230|2|3.5|0.5|0.5|
|4|4.00|1|N|230|226|2|14|0.5|0.5|
|1|2.10|1|N|237|48|2|10|0.5|0.5|

```

This took me back to check the original raw data and how the data merging was performed at the very beginning.

The reason: the two data frames were merged with **UNION** function and all the columns in the two datasets were disorderly mixed up and joined. Due to hardware limitation and tediously slow

execution, the EDA such as `describe()` function was not able to perform at the earlier stage. This is the main reason.

The solution: `unionByName`. This function finds the same name of columns and merge accordingly.

```
# Drop the uncommon column
df_green = df_green.drop("ehail_fee", "trip_type")

# Rename the columns tpep_pickup_datetime to pickup_datetime / tpep_dropoff_datetime to dropoff_datetime
df_yellow = df_yellow.withColumnRenamed("tpep_pickup_datetime", "pickup_datetime")
df_yellow = df_yellow.withColumnRenamed("tpep_dropoff_datetime", "dropoff_datetime")
# Make the column names lowercase
for col in df_yellow.columns:
    df_yellow = df_yellow.withColumnRenamed(col, col.lower())

# Rename the columns lpep_pickup_datetime to pickup_datetime / lpep_dropoff_datetime to dropoff_datetime
df_green = df_green.withColumnRenamed("lpep_pickup_datetime", "pickup_datetime")
df_green = df_green.withColumnRenamed("lpep_dropoff_datetime", "dropoff_datetime")
for col in df_green.columns:
    df_green = df_green.withColumnRenamed(col, col.lower())

df = df_green.unionByName(df_yellow)
```

SQL Queries and Recommendation to Business

According to the queries performed, a couple of findings are listed below.

- Every month at during 18:00 –21:00 hours of the day got maximum number of passenger pickups.
- About 61% of the total trips, the drivers earned tip amount.
- Out of the overall trips, 0.67% of the trips got tip amount more than 10 dollars.
- If the trip durations in minutes are categorised into below 5mins, 5-10 min, 10-20min, 20-30 min and above 30min, the trips of less than 5 min duration earned the highest total fare compared to the rest of the duration bins. Therefore, ***taxi drivers are suggested to target the passenger pick up for short distance which would take about 5 minutes or within 5 to 10 minutes to maximise their income.***

Appendix

SQL 4.a.i



```
1 # Question 4.a.i - What is the total number of trip for each month
2 spark.sql('''
3 SELECT YEAR(pickup_datetime) year,
4         MONTH(pickup_datetime) month,
5         COUNT (1) number_of_trips
6 FROM nyc_view
7 GROUP BY YEAR(pickup_datetime), MONTH(pickup_datetime)
8 ORDER BY 1,2
9 ''').show(24)
```



| year | month | number_of_trips |
|------|-------|-----------------|
| 2017 | 1 | 7454214 |
| 2017 | 2 | 7095996 |
| 2017 | 3 | 7692202 |
| 2017 | 4 | 7659237 |
| 2017 | 5 | 7545181 |
| 2017 | 6 | 7239610 |
| 2017 | 7 | 6591228 |
| 2017 | 8 | 6453155 |
| 2017 | 9 | 6676683 |
| 2017 | 10 | 7258643 |
| 2017 | 11 | 6790598 |
| 2017 | 12 | 6916721 |
| 2018 | 1 | 8107463 |
| 2018 | 2 | 7912285 |
| 2018 | 3 | 8607307 |
| 2018 | 4 | 8447906 |
| 2018 | 5 | 8338469 |
| 2018 | 6 | 7907786 |
| 2018 | 7 | 8272445 |
| 2018 | 8 | 7180931 |
| 2018 | 9 | 8179537 |
| 2018 | 10 | 7731997 |
| 2018 | 11 | 7035319 |
| 2018 | 12 | 7164661 |

SQL 4.a.ii

```

1 # Question 4.a.ii - Which weekday had the most trip
2 spark.sql(''
3 WITH trip_rank AS(
4 SELECT *,
5     RANK() OVER(PARTITION BY year,month ORDER BY number_of_trips DESC) rank
6 FROM (
7     SELECT
8         YEAR(pickup_datetime) year,
9         MONTH(pickup_datetime) month,
10        week_day_abb weekday,
11        Count(*) number_of_trips
12
13     FROM nyc_view
14     GROUP BY YEAR(pickup_datetime), MONTH(pickup_datetime), week_day_abb
15     ORDER BY 1,2,3
16 )
17 )
18 SELECT * FROM trip_rank
19 WHERE rank = 7
20 ORDER BY year,month
21 '').show(24)

```

| year | month | weekday | number_of_trips | rank |
|------|-------|---------|-----------------|------|
| 2017 | 1 | Wed | 948978 | 7 |
| 2017 | 2 | Mon | 915248 | 7 |
| 2017 | 3 | Tue | 792731 | 7 |
| 2017 | 4 | Mon | 906003 | 7 |
| 2017 | 5 | Sun | 947080 | 7 |
| 2017 | 6 | Sun | 879103 | 7 |
| 2017 | 7 | Tue | 825031 | 7 |
| 2017 | 8 | Mon | 753575 | 7 |
| 2017 | 9 | Mon | 782838 | 7 |
| 2017 | 10 | Wed | 936837 | 7 |
| 2017 | 11 | Mon | 850211 | 7 |
| 2017 | 12 | Mon | 787822 | 7 |
| 2018 | 1 | Thu | 955768 | 7 |
| 2018 | 2 | Sun | 1032196 | 7 |
| 2018 | 3 | Wed | 932273 | 7 |
| 2018 | 4 | Tue | 1091916 | 7 |
| 2018 | 5 | Mon | 970723 | 7 |
| 2018 | 6 | Sun | 980493 | 7 |
| 2018 | 7 | Fri | 1003048 | 7 |
| 2018 | 8 | Mon | 858048 | 7 |
| 2018 | 9 | Mon | 842760 | 7 |
| 2018 | 10 | Sun | 971632 | 7 |
| 2018 | 11 | Mon | 891876 | 7 |
| 2018 | 12 | Tue | 861179 | 7 |

SQL 4.a.iii

```

1 # Question 4.a.iii Which hour of the day had the most trip 3:45 - 3:54
2 spark.sql('''
3 WITH trip_rank AS(
4 SELECT *,
5     RANK() OVER(PARTITION BY year,month ORDER BY total_number_of_trips DESC) rank
6 FROM (
7     SELECT
8         YEAR(pickup_datetime) year,
9         MONTH(pickup_datetime) month,
10        Hour(pickup_datetime) hour,
11        Count(*) total_number_of_trips
12
13     FROM nyc_view
14     GROUP BY YEAR(pickup_datetime), MONTH(pickup_datetime), hour(pickup_datetime)
15     ORDER BY 1,2,3
16 )
17 )
18 SELECT * FROM trip_rank
19 WHERE rank = 1
20 ORDER BY year,month
21 ''').show(24)

```

| year | month | hour | total_number_of_trips | rank |
|------|-------|------|-----------------------|------|
| 2017 | 1 | 18 | 462433 | 1 |
| 2017 | 2 | 18 | 453654 | 1 |
| 2017 | 3 | 19 | 494218 | 1 |
| 2017 | 4 | 19 | 474410 | 1 |
| 2017 | 5 | 21 | 475430 | 1 |
| 2017 | 6 | 21 | 453632 | 1 |
| 2017 | 7 | 19 | 405442 | 1 |
| 2017 | 8 | 19 | 404956 | 1 |
| 2017 | 9 | 19 | 426336 | 1 |
| 2017 | 10 | 19 | 462706 | 1 |
| 2017 | 11 | 21 | 425210 | 1 |
| 2017 | 12 | 21 | 418193 | 1 |
| 2018 | 1 | 18 | 535981 | 1 |
| 2018 | 2 | 18 | 521754 | 1 |
| 2018 | 3 | 18 | 556677 | 1 |
| 2018 | 4 | 18 | 553714 | 1 |
| 2018 | 5 | 18 | 523926 | 1 |
| 2018 | 6 | 19 | 494108 | 1 |
| 2018 | 7 | 18 | 509901 | 1 |
| 2018 | 8 | 18 | 464695 | 1 |
| 2018 | 9 | 18 | 512175 | 1 |
| 2018 | 10 | 19 | 494093 | 1 |
| 2018 | 11 | 18 | 431556 | 1 |
| 2018 | 12 | 18 | 434255 | 1 |

SQL 4.a.iv

```

1 # Question 4.a.iv What was the average number of passenger?
2 ### Avg passenger = total Passenger / total number of Trip
3 spark.sql('''
4 SELECT
5     YEAR(pickup_datetime) year,
6     MONTH(pickup_datetime) month,
7     SUM(passenger_count) total_passenger_count,
8     COUNT(*) total_trip_count,
9     FLOOR((SUM(passenger_count)/COUNT(*))) avg_passenger_count
10 FROM nyc_view
11 GROUP BY YEAR(pickup_datetime), MONTH(pickup_datetime)
12 ORDER BY 1,2
13 ''').show(24)

```

| year | month | total_passenger_count | total_trip_count | avg_passenger_count |
|------|-------|-----------------------|------------------|---------------------|
| 2017 | 1 | 11977257 | 7454214 | 1 |
| 2017 | 2 | 11373058 | 7095996 | 1 |
| 2017 | 3 | 12262729 | 7692202 | 1 |
| 2017 | 4 | 12287367 | 7659237 | 1 |
| 2017 | 5 | 12054726 | 7545181 | 1 |
| 2017 | 6 | 11595562 | 7239610 | 1 |
| 2017 | 7 | 10668300 | 6591228 | 1 |
| 2017 | 8 | 10402774 | 6453155 | 1 |
| 2017 | 9 | 10743991 | 6676683 | 1 |
| 2017 | 10 | 11644272 | 7258643 | 1 |
| 2017 | 11 | 10875274 | 6790598 | 1 |
| 2017 | 12 | 11217601 | 6916721 | 1 |
| 2018 | 1 | 12582854 | 8107463 | 1 |
| 2018 | 2 | 12208190 | 7912285 | 1 |
| 2018 | 3 | 13334384 | 8607307 | 1 |
| 2018 | 4 | 13105392 | 8447906 | 1 |
| 2018 | 5 | 12910119 | 8338469 | 1 |
| 2018 | 6 | 12249109 | 7907786 | 1 |
| 2018 | 7 | 12924577 | 8272445 | 1 |
| 2018 | 8 | 11135215 | 7180931 | 1 |
| 2018 | 9 | 12676099 | 8179537 | 1 |
| 2018 | 10 | 11829012 | 7731997 | 1 |
| 2018 | 11 | 10820820 | 7035319 | 1 |
| 2018 | 12 | 11089031 | 7164661 | 1 |

SQL 4.a.v

```
1 # Question 4.a.v What was the average amount paid per trip?
2 spark.sql(''
3 SELECT
4     YEAR(pickup_datetime) year,
5     MONTH(pickup_datetime) month,
6     ROUND(AVG(total_amount),2) avg_amount_per_trip
7 FROM nyc_view
8 GROUP BY YEAR(pickup_datetime), MONTH(pickup_datetime)
9 ORDER BY 1,2
10 '').show(24)
```

```
> +---+---+-----+
|year|month|avg_amount_per_trip|
+---+---+-----+
|2017| 1|15.71|
|2017| 2|15.74|
|2017| 3|15.87|
|2017| 4|15.83|
|2017| 5|15.82|
|2017| 6|15.73|
|2017| 7|15.97|
|2017| 8|15.95|
|2017| 9|15.95|
|2017|10|15.96|
|2017|11|15.91|
|2017|12|15.86|
|2018| 1|15.6|
|2018| 2|15.6|
|2018| 3|15.59|
|2018| 4|15.79|
|2018| 5|15.88|
|2018| 6|15.82|
|2018| 7|15.92|
|2018| 8|15.94|
|2018| 9|16.11|
|2018|10|16.09|
|2018|11|16.04|
|2018|12|15.96|
+---+---+-----+
```

SQL 4.b.i

```
1 # Question 4.b.i For Each Taxi Color
2 ## What was the average, median, mini and max trip_duration in seconds
3 ### http://spark.apache.org/docs/latest/sql-ref-functions-builtin.html
4 spark.sql(''
5 SELECT
6     taxi_company,
7     ROUND(AVG(trip_duration_sec),2) avg_trip_duration_sec,
8     MIN(trip_duration_sec) min_trip_duration_sec,
9     MAX(trip_duration_sec) max_trip_duration_sec,
10    APPROX_PERCENTILE(trip_duration_sec, 0.5) median_trip_duration_sec
11 FROM nyc_view
12 GROUP BY taxi_company
13 '').show()
```

```
+---+-----+-----+-----+-----+
|taxi_company|avg_trip_duration_sec|min_trip_duration_sec|max_trip_duration_sec|median_trip_duration_sec|
+---+-----+-----+-----+-----+
|Green|872.1|58|1999|787|
|Yellow|844.31|58|1999|747|
+---+-----+-----+-----+-----+
```

SQL 4.b.ii

```
1 ##### Q-4.b.ii For Each Taxi Color What was the average, median, mini and max trip distance in Km
2 spark.sql('''
3 SELECT
4     taxi_company,
5     ROUND((AVG(trip_distance)/0.621371),2) avg_trip_dist_km,
6     ROUND((MIN(trip_distance)/0.621371),2) min_trip_dist_km,
7     ROUND((MAX(trip_distance)/0.621371),2) max_trip_dist_km,
8     ROUND((APPROX_PERCENTILE(trip_distance, 0.5)),2) median_trip_dist_km
9 FROM nyc_view
10 GROUP BY taxi_company
11 ''').show()
```

| taxi_company | avg_trip_dist_km | min_trip_dist_km | max_trip_dist_km | median_trip_dist_km |
|--------------|------------------|------------------|------------------|---------------------|
| Green | 4.65 | 1.61 | 55.31 | 2.0 |
| Yellow | 4.89 | 1.61 | 52.95 | 2.28 |

SQL 4.b.iii

```
1 ##### Q-4.b.iii For Each Taxi Color What was the average, median, mini and max speed in Km per hour
2 # speed_km_hr
3 spark.sql('''
4 SELECT
5     taxi_company,
6     ROUND(AVG(speed_km_hr),2) avg_speed_km_hr,
7     MIN(speed_km_hr) min_speed_km_hr,
8     MAX(speed_km_hr) max_speed_km_hr,
9     APPROX_PERCENTILE(speed_km_hr, 0.5) median_speed_km_hr
10 FROM nyc_view
11 GROUP BY taxi_company
12 ''').show()
```

| taxi_company | avg_speed_km_hr | min_speed_km_hr | max_speed_km_hr | median_speed_km_hr |
|--------------|-----------------|-----------------|-----------------|--------------------|
| Green | 18.99 | 2.9 | 99.99 | 16.7 |
| Yellow | 20.6 | 2.91 | 99.9 | 18.51 |

SQL 4.C

```
1 ##### Q-4.c What was the percentage of trips where the driver received tips?
2 spark.sql('''
3 SET tipped_trip_count = (SELECT COUNT(*) tipped_trip_count
4 FROM nyc_view
5 WHERE tip_amount > 0)
6 ''')
7
8 spark.sql('''
9 SELECT ROUND((${tipped_trip_count}/COUNT(*)) * 100, 2) trip_percent_with_tips
10 FROM nyc_view
11 ''').show()
```

| trip_percent_with_tips |
|------------------------|
| 61.83 |

SQL 4.d

```

1 ##### Q-4.d What was the percentage of trips where the driver received tips >= 10$ ?
2 # https://www.javaer101.com/en/article/41891788.html ref for variable
3 spark.sql(''
4 SET high_tip_trip_count = (SELECT COUNT(*) high_tip_trip_count
5 FROM nyc_view
6 WHERE tip_amount >= 10)
7 '')
8
9 spark.sql(''
10 SELECT ROUND((${high_tip_trip_count}/COUNT(*)) * 100, 2) trip_percent_with_morethan_10dollars_tips
11 FROM nyc_view
12 '').show()
13
+-----+
|trip_percent_with_morethan_10dollars_tips|
+-----+
|                                0.67|
+-----+

```

SQL 4.e.i

```

1 ##### Q.4.e.i For each bin of trip duration - what is the average speed (Km per hour) - speed_km_hr
2 spark.sql(''
3 SELECT
4     trip_duration_range_mins,
5     ROUND(SUM(speed_km_hr)/ COUNT(*),2) average_speed_km_per_hr
6 FROM nyc_view
7 GROUP BY trip_duration_range_mins
8 '').show()
9
+-----+-----+
|trip_duration_range_mins|average_speed_km_per_hr|
+-----+-----+
|          10-20 mins|          17.91|
|           5-10 mins|          19.27|
|          20-30 mins|          20.72|
|           <5 mins|          28.16|
|           >30 mins|          23.51|
+-----+-----+

```

SQL 4.e.ii

```

1 ##### Q.4.e.ii For each bin of trip duration - what is the average distance (dist per dollar)
2 spark.sql(''
3 SELECT trip_duration_range_mins,
4     ROUND((SUM(trip_distance)/0.621371),2) total_distance,
5     ROUND(SUM(total_amount),2) sum_fare_amount,
6     ROUND((SUM(trip_distance)/0.621371) / SUM(total_amount)), 2) avg_dist_km_per_dollar
7
8 FROM nyc_view
9 GROUP BY trip_duration_range_mins
10 '').show()

```

| trip_duration_range_mins | total_distance | sum_fare_amount | avg_dist_km_per_dollar |
|--------------------------|----------------|-----------------|------------------------|
| 10-20 mins | 3.6325974517E8 | 1.27872238223E9 | 0.28 |
| 5-10 mins | 1.2740173054E8 | 5.1183941346E8 | 0.25 |
| 20-30 mins | 2.7642736134E8 | 8.351790931E8 | 0.33 |
| <5 mins | 1.01388973E7 | 4.116772026E7 | 0.25 |
| >30 mins | 6.819661047E7 | 1.9123135426E8 | 0.36 |

Deliverables

1. BDE_AT1_Data_Processing_HninPwintTin_13738339.ipnyb
2. BDE_AT1_Pipeline_Modelling_HninPwintTin_13738339.ipnyb [link to colab here](#)
3. BDE_AT1_NYC_Biz_Queries_HninPwintTin_13738339.sql

4. BDE_AT1_Technical_Handover_Report_HninPwintTin_13738339.pdf