



A STUDY ON NEW YORK CITY TAXI RIDES – Fare and Duration Prediction

Group Members: Yavuz Selim Sefunç, Çağlar Subaşı, Onur Karaman, Suat Buldanlıoğlu
Course: Big Data Technologies and Applications

Term: Summer – 2018

Instructor: Doç. Dr. Altan Çakır

ABSTRACT

In this paper, our aim is to develop a predictive model for ‘NYC Taxi Data’ which contains approximately 200 million yellow taxi trips between 2009 and 2017. The raw data is reachable from the NYC Taxi & Limousine Commission website month by month and it represents 17-21 variables for each taxi ride. The size of the raw data is nearly 200 GB in total.

Possible predictive analyses can be performed on;

- fare amount (average, maximum or minimum)
- tip amount or
- duration

of a trip by treating one of them as a dependent (target) variable. In this study, we decided “fare amount and trip duration” to be estimated, since they would be valuable for taxi vendors optimizing their incomes/operations and for passengers managing their expenditures/times.

Statistical regressive models (linear/nonlinear) and machine learning methods (Decision Tree, Support Vector Machine and Convolutional Neural Network) will be used to produce predictions on the chosen target variable.

Information obtained from this study can be used by numerous authorities and industries for their own purpose. Gained insight about data that can be beneficial in Turkey especially for entities provides taxi services (e.g. BiTaksi).

This paper includes only;

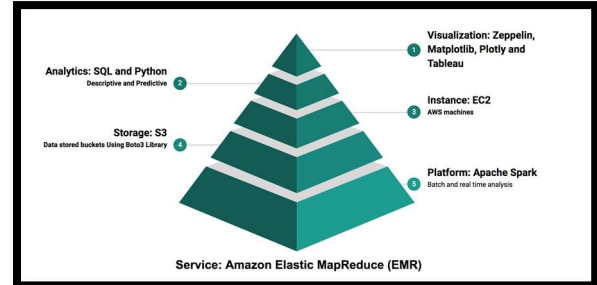
- ✓ Data Processing (Infrastructure),
- ✓ Data Collection,
- ✓ Data Manipulation,
- ✓ EDA¹ and
- ✓ Data Imputation

¹ Exploratory Data Analysis

parts of the study. We will exhibit further parts (Feature Engineering, Feature Selection, Collinearity Detection, Model Selection and Result Interpretation) at the end of the final semester.

A- Data Processing (Infrastructure)

Platforms and tools that are used in this study can be shown as the below graph.



All platforms and tools are provided by Amazon Web Services (AWS). Due to the size of our data, it is considered that using Hadoop/Spark base system would be efficient to store and analyze our data. We have created a S3-Bucket (s3://ludditiesnyc taxi/....) in AWS and downloaded raw data from NYC official website². EC2-Instances and EMR-Clusters are used for analysis via AWS-CLI (terminal), Zeppelin, JupyterHub and Spark interfaces with the help of Pyspark and SparkSQL API's. We also used 'Bootstrap action' to install abovementioned interfaces and related libraries and to configure clusters' instances easily at each time they created.

² http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml

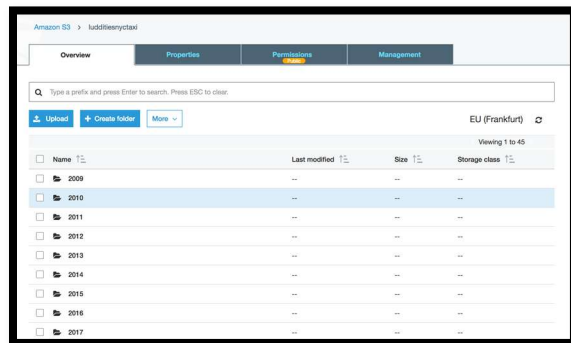
B- Data Collection

We had three options to collect the data, which are;

- downloading data to local machine/external hard disk by using web browser,
- downloading data to HDFS by using ‘-wget’ function on AWS-CLI or
- downloading data directly to S3-Bucket by using ‘Boto3’ dictionary on python library.

We have tried succeed two of them (a and c) and decided to use option (a) for getting 200 GB data due to time and speed constraints.

Above mentioned official link had been used to download data, monthly. Then, we have transferred these monthly files to S3-Buckets by dragging it to AWS interface. The interface appearance of our S3-Bucket is shown below.

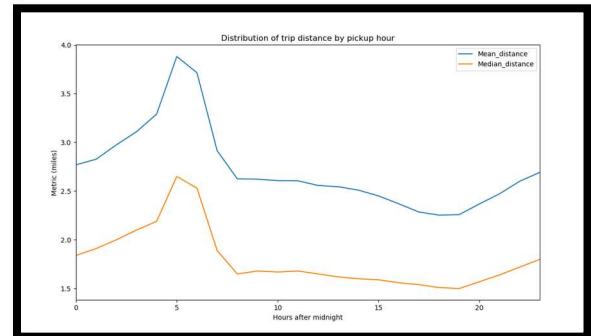


Our S3-Bucket is public, so one can reach it from any IP in the world.

C- Data Understanding

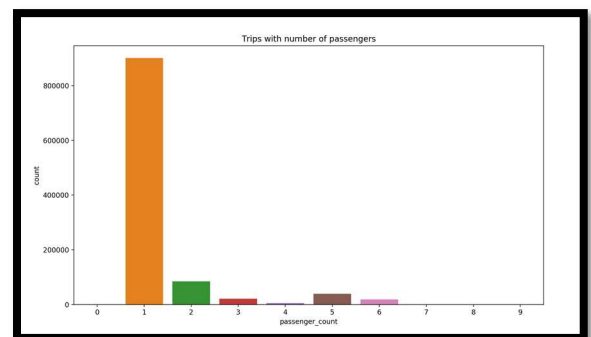
In order to understand what our data represents, drawing graphs/charts is an easy and efficient way. We produced them using platforms and tools mentioned in ‘Data Processing’ part of the report.

1)



Plot is suggesting that the mean trip distances is longer in morning and evening hours. This could be the population that uses cabs to commute for work. But if so the evening commuter are much less than morning commuters. This indicates that the people who takes taxi in the morning to work do not use it when they go back home.

2)



Big part of the total rides is single passenger rides. This seems to be working individuals.

3)



Above map indicates ‘yellow taxi’s pickups and drop-offs locations. A detailed one is located on our homework folder which is “nyc_map.html”.

A dictionary of the data is added to the ‘Appendix’ part of the report. It might be helpful to grasp meaning of each variable of the data.

D- Data Manipulation

In the Data Manipulation process, one should handle;

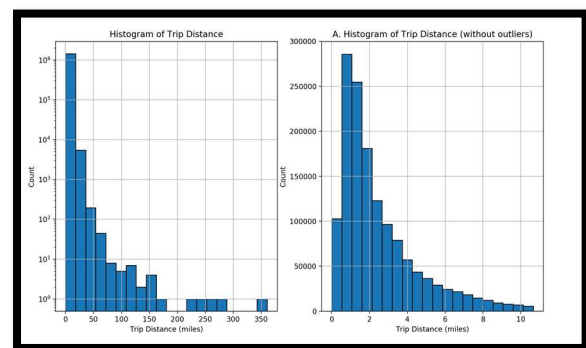
- Outliers
- Missing Values
- Repeated Rows (Uniqueness)

in the dataset.

Outliers have potential to create noise in our data variance. We have checked them for columns such as “fare_amount” by using Pyspark RDD data structure like below.

```
Sort lines (taxi trips) wrt total_amount (charged to customers) in descending order and
print first 15 of them...
>>> lines = sc.textFile("file:/home/ubuntu/taxidata/yellow_tripdata_2017-01.csv")
>>> content = lines.filter(lambda line: line != header)
>>> max_15_total_amount = content.map(lambda line: (line.split(",")[0], float(line.split(",")[4])),
Output: line.split(",")[9], float(line.split(",")[16]))).takeOrdered(15, lambda x: -x[3])
>>> max_15_total_amount
Output: [('1', 0.0, '2', 625901.6), ('1', 0.0, '3', 538580.0), ('2', 0.0, '2', 9001.3), ('1', 15.9, '4',
8043.84), ('1', 6.0, '3', 3009.8), ('1', 1.9, '2', 3009.3), ('1', 0.5, '1', 2000.28), ('1', 4.1, '1', 2000.28),
('1', 0.0, '1', 2000.28), ('1', 0.7, '1', 1000.29), ('1', 17.0, '2', 963.88), ('1', 7.8, '4', 930.34), ('1', 0.8,
'3', 899.34), ('1', 5.0, '3', 824.38), ('1', 17.0, '3', 776.3)]
```

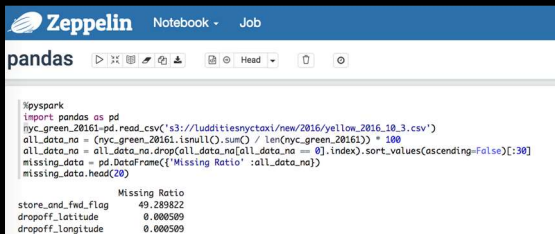
As can be seen from above output some trips have enormously high “fare_amount” then other trips which can be treated as outliers. So, values which behave distinguishably different will be deleted/detached our data manually.



First figure is a “trip_distance” histogram with outliers. But, second one is the same without outliers. We did exclude any data point located

further than 3 standard deviations of the median point.

To handle missing values in our data, firstly, we detect the percentage of null values at each column by using below python code.



```

%pySpark
import pandas as pd
nyc_green_20161=pd.read_csv('s3://ludditiesnyc taxi/new/2016/yellow_2016_10_3.csv')
all_data_no = (nyc_green_20161.isnull().sum() / len(nyc_green_20161)) * 100
all_data_no = all_data_no.drop(all_data_no[all_data_no == 0].index).sort_values(ascending=False)[:30]
missing_data = pd.DataFrame({'Missing Ratio' :all_data_no})
missing_data.head(20)

Missing Ratio
store_and_fwd_flag    49.289822
dropoff_latitude      0.000509
dropoff_longitude     0.000509

```

Above screen-shoot represents one-month implementation. One can adapt this code to whole data by using loop logic via EMR-Cluster platform. We got “Out of Memory/ Java Heap Space/GC Overhead Limit Exceeded” errors while trying mentioned implementation on even 1 year data. Memory allocation problem is the one that we have faced frequently in our study as expected. Although it is not necessary to use all data while establishing a predictive model³, we need it as a whole for manipulation (cleaning and imputing) processes. This problem might be solved by;

- using advance instances (‘p’ or ‘r’ type) as workers/nodes,
- increasing workers’ number,
- using Kyro Serialization⁴,
- tuning driver and executor memory parameters of spark environment,
- using GC1 garbage collector method⁵,
- implementing ‘chunk-size’ functionality of Pandas,
- implementing ‘repartition’ functionality of Pyspark and

- applying ‘broadcasting’ method of Pyspark.

These methods have been tried both individually and simultaneously to get rid of memory allocation restrictions of our big data tools. Up to this report, we have achieved to use one-year data which has approximately 10-20 GB size.

Every data may have duplicates in it, thus, it is important to check its uniqueness. We checked our data in terms of uniqueness with the help of Pandas library. We found that there are no repeating rows in our data. There is an example below.

```

10115997    False
10115998    False
10115999    False
10116000    False
10116001    False
10116002    False
10116003    False
10116004    False
10116005    False
10116006    False
10116007    False
10116008    False
10116009    False
10116010    False
10116011    False
10116012    False
10116013    False
10116014    False
10116015    False
10116016    False
10116017    False
Length: 10116018, dtype: bool

In [3]: df_2016_1.shape
Out[3]: (10116018, 17)

```

³ Randomly chosen sample of 10.000 data from 2009 – 2017 would be enough to create a statistically robust models.

⁴ Spark is using ‘Java serialization’ as default

⁵ Spark is using ‘Java Paralel GC’ as default

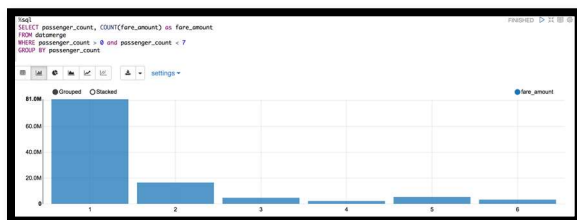
E- Exploratory Data Analysis (EDA)

EDA is an open-ended process where we make plots and calculate statistics in order to explore our data. The purpose of this process is to find anomalies, patterns, trends, or relationships between independent and target variables.

EDA generally starts out with a high-level overview, and then narrows in to specific parts of the dataset once as we find interesting areas to examine. To begin with EDA, we will focus on a single variable, “fare_amount”, since it has been chosen as the target value that our machine learning models will try to predict.

Outputs of EDA will offer us some insights about the relationships in our data and drive us to “Feature Engineering” process where we shape our data to strengthen correlations between independent variables and dependent variable.

As a first step of this part, we have used some categorical variables as a separator to discover distinguishable behaviors of target variable if it exists. Below output is a sample and simple beginning of that kind of exploration.



Graph tells us that taxi rides, which include 1 passenger only, are representing most of the total revenue in our dataset. So, we continue our journey with a deep look through this possible relationship between variables by taking averages.

passenger_count	avg(fare_amount)
1	12.87388027655138
2	13.65698184001725
3	13.58012402389335
4	13.713629135554983
5	13.0199006617575
6	12.97138821238673

As can be seen from the above figure, there is no difference in terms of average ‘fare_amount’ with respect to count of passengers changing 1 to 6. So, ‘passenger_count’ will not be considered and treated as a useful discriminator of ‘fare_amount’.

Another categorical variable, which we can check whether its subcategories indicate differentiation in terms of target variable, is ‘vendor_id’. All taxi rides in our data are served by two vendors ‘Creative Mobile Technologies’ and ‘VeriFone Inc.’ having identity number ‘1’ and ‘2’, respectively.

Below figure comprises a Spark RDD experiment searching abovementioned differentiation.

```
1- Calculate sum of total_amount variable wrt vendors_id...
>>> vendors_total_revenue = content_2.map(lambda line: (line.split(",")[0],))
Output: float(line.split(",")[16]))).reduceByKey(lambda x,y: x+y)
>>> vendors_total_revenue.collect()
Output: [(1, 67916970.82997812), (2, 82849477.31987293)]

2- Calculate average total_amount wrt vendors...

Firstly, we should add a 'counter' to each line:
>>> vendors_average_revenue_1 = content_2.map(lambda line: (line.split(",")[0],))
Output: float(line.split(",")[16]))).mapValues(lambda x: (x, 1))
>>> vendors_average_revenue_1.take(10)
Output: [(1, (15.3, 1)), (1, (7.25, 1)), (1, (7.3, 1)), (1, (8.5, 1)), (2, (52.8, 1)), (1, (5.3, 1)), (2, (27.96, 1)), (1, (8.75, 1)), (1, (8.3, 1)), (2, (8.3, 1))]

Then we should sum each total_amount and counter wrt vendor_id:
>>> vendors_average_revenue_2 = content_2.map(lambda line: (line.split(",")[0],))
Output: float(line.split(",")[16]))).mapValues(lambda x: (x, 1)).reduceByKey(lambda x,y: (x[0] + y[0], x[1] + y[1]))
>>> vendors_average_revenue_2.collect()
Output: [(1, (67916970.82997812, 4397921)), (2, (82849477.31987293, 5312203))]

Finally, we have to give the total total_amount by total count for each vendor:
>>> vendors_average_revenue_3 = vendors_average_revenue_2.mapValues(lambda x: x[0]/x[1])
>>> vendors_average_revenue_3.collect()
Output: [(1, 15.442971992898036), (2, 15.596067642722414)]
```

In this example, we conclude that vendor ‘VeriFone Inc.’ has gained slightly (%1) more than ‘Creative Mobile Technologies’ for each trip. So, the dependent variable ‘vendor_id’ is considered to be useless for creating distinguishable sub-classes of target variable.

Variables representing pickup and drop-off coordinates/zones of rides might be a good discriminator for fare_amount. These type of variables are shown as discrete numeric in our dataset, but they can be easily transformed (Feature Engineering) into categoric type by creating buckets/intervals which symbolizes neighbor zones of the New York City.

Below figures respectively stands for pickup and drop-off operation of taxi trips and they represent average 'fare_amount' for each zone.

```

SELECT ROUND(DOlocationID, 4) AS Pick_Zone,
COUNT(DOlocationID) AS num_pickups,
SUM(fare_amount) AS total_revenue,
SUM(fare_amount)/COUNT(*) AS avr_revenue
FROM yellow_tripdata
WHERE (fare_amount/15) < distance BETWEEN 2 AND 10)
GROUP BY Pick_Zone

```

Pick_Zone	num_pickups	total_revenue	avr_revenue [f]
1.0	905	62255.970000000016	68.79555
219.0	5734	311432.67999999993	54.31334
154.0	252	13073.5	53.06844
215.0	7585	398079.47	52.48246
10.0	23774	1247329.35	52.46611
99.0	10	509.0	50.9
176.0	20	1016.5	50.825
117.0	110	5360.7	48.72664
44.0	400	34136.5	85.34125

```

SELECT ROUND(DOlocationID, 4) AS Drop_Zone,
COUNT(DOlocationID) AS num_dropoffs,
SUM(fare_amount) AS total_revenue,
SUM(fare_amount)/COUNT(*) AS avr_revenue
FROM yellow_tripdata
WHERE (fare_amount/15) < distance BETWEEN 2 AND 10)
GROUP BY Drop_Zone

```

Drop_Zone	num_dropoffs	total_revenue	avr_revenue [f]
44.0	400	34136.5	85.34125
204.0	613	48662.5	79.4198
84.0	741	58697.2	79.2135
5.0	580	44925.25	77.45733
265.0	200985	1.4889927150000002E7	73.98526
1.0	208795	1.4889886999999999E7	71.01645
110.0	2	141.5	70.75
109.0	1042	71476.0	68.59501

We can interpret these outputs that some zones including pickup or drop-off activity seems to have higher fare_amount than others.

As a result, we realize that only the pickup and drop-off zone information may bring some useful variances for our target variable. So, we will focus on these types of variables in “Feature Engineering” part in order to sharp their separation capabilities on ‘fare_amount’.

Current Status & Waiting Challenges

During the analyses, we have used ‘SQL’ and ‘Python’ API’s of Apache Spark to handle with our huge data. It had been successfully finalized while working on only 1-month length data (2-3 GB) through both aforesaid libraries. On the other hand, enlarging the scope of the data (e.g. 1 year – 15 GB) has brought “Out of Memory” error, when using Python-Pandas library which offers advanced visualization capabilities with ‘plotly’ and ‘matplotlib’ dictionaries. SQL library, in contrast, could handle this challenge, but it has limited visualization capacity.

So, working with 1 year or more sized data has the priority to be achieved for our study. We will focus on the possible solutions of memory error mentioned in “Data Manipulation” part of the report.

And surely, we are going to complete remaining parts of our study index, from F to K, listed below.

- A- Data Processing (Infrastructure) ,
- B- Data Collection,
- C- Data Understanding,
- D- Data Manipulation,
- E- EDA,
- F- Data Imputation,
- G- Feature Engineering,
- H- Feature Selection,
- I- Collinearity Detection,
- J- Model Selection and
- K- Interpretation of the Results

To work on the prediction of trip duration, we will try to create it by taking subtraction of ‘pickup_datetime’ and ‘dropoff_datetime’ variables for whole data. Of course, this is a further task to be worked on after handling with memory errors.

Appendix

Data Dictionary

(http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml)

Field Name	Description
VendorID	A code indicating the TPEP provider that provided the record. 1= Creative Mobile Technologies, LLC; 2= VeriFone Inc.
trip_pickup_datetime	The date and time when the meter was engaged.
trip_dropoff_datetime	The date and time when the meter was disengaged.
Passenger_count	The number of passengers in the vehicle. This is a driver-entered value.
Trip_distance	The elapsed trip distance in miles reported by the taximeter.
PULocationID	TLC Taxi Zone in which the taximeter was engaged
DOLocationID	TLC Taxi Zone in which the taximeter was disengaged
RateCodeID	The final rate code in effect at the end of the trip. 1= Standard rate 2=JFK 3=Newark 4=Nassau or Westchester 5=Negotiated fare 6=Group ride
Store_and_fwd_flag	This flag indicates whether the trip record was held in vehicle memory before sending to the vendor, aka "store and forward," because the vehicle did not have a connection to the server. Y= store and forward trip N= not a store and forward trip
Payment_type	A numeric code signifying how the passenger paid for the trip. 1= Credit card 2= Cash 3= No charge 4= Dispute 5= Unknown 6= Voided trip
Fare_amount	The time-and-distance fare calculated by the meter.
Extra	Miscellaneous extras and surcharges. Currently, this only includes the \$0.50 and \$1 rush hour and overnight charges.
MTA_tax	\$0.50 MTA tax that is automatically triggered based on the metered rate in use.
Improvement_surcharge	\$0.30 improvement surcharge assessed trips at the flag drop. The improvement surcharge began being levied in 2015.
Tip_amount	Tip amount – This field is automatically populated for credit card tips. Cash tips are not included.
Tolls_amount	Total amount of all tolls paid in trip.
Total_amount	The total amount charged to passengers. Does not include cash tips.

Table shows the most actual (2017) columns/variables in the data which might shows some differences between early years. For example;

- 'PULocationID' and 'DOLocationID' columns were represented by latitude and longitude coordinates in 2016/06 and before.
- 'trip_type' variable were dropped from dataset at the end of 2016.