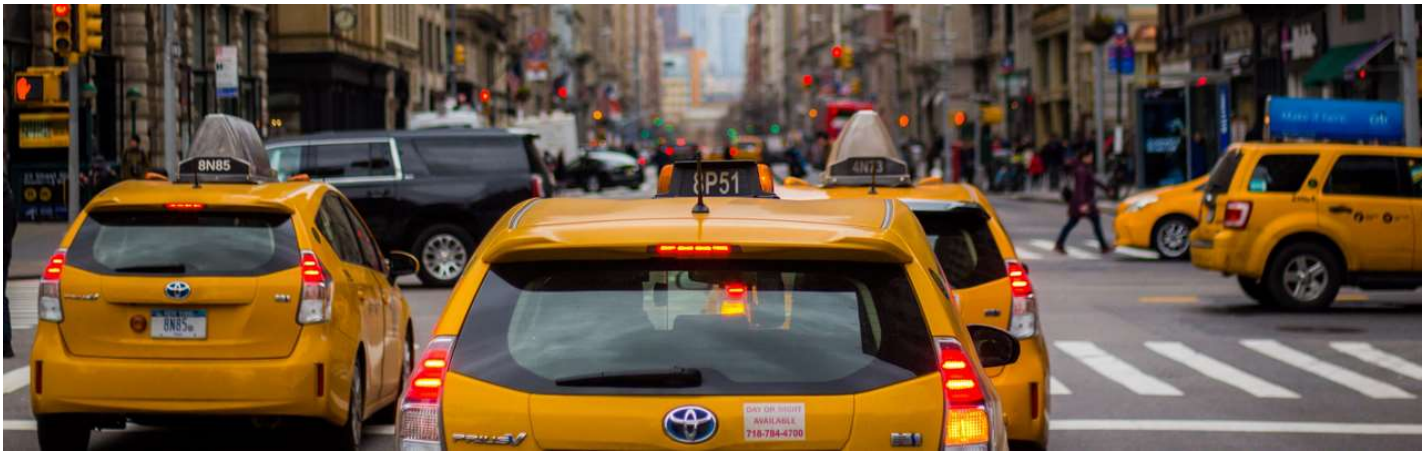
New York City Taxi Fare Prediction



The goal of this project is to predict with reasonable accuracy, the taxi fares of new york city based on the dataset provided by this kaggle competition: https://www.kaggle.com/competitions/new-york-city-taxi-fare-prediction/overview

The dataset consists of following csv files:

- train.csv: Input features and target fare_amount values for the training set (about 55M rows)
- test.csv: Input features for the test set (about 10K rows). Your goal is to predict fare_amount for each row.
- sample_submission.csv: a sample submission file in the correct format (columns key and fare_amount). This file 'predicts' fare_amount to be $11.35 for all rows, which is the mean fare_amount from the training set.

Following are the features of the training and test dataset:

- pickup_datetime - timestamp value indicating when the taxi ride started.
- pickup_longitude - float for longitude coordinate of where the taxi ride started.
- pickup_latitude - float for latitude coordinate of where the taxi ride started.
- dropoff_longitude - float for longitude coordinate of where the taxi ride ended.
- dropoff_latitude - float for latitude coordinate of where the taxi ride ended.
- passenger_count - integer indicating the number of passengers in the taxi ride.

The target column fare_amount is a float dollar amount of the cost of the taxi ride. This value is only in the training set; this is what we are predicting in the test set

Import Libraries

Lets start by importing all the necessary python libraries and packages required for this project:

```
import pandas as pd
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
```

```
from sklearn.cluster import KMeans
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import MinMaxScaler

%matplotlib inline
```

Download the Dataset

The following code helped me download the dataset as a zip file directly to the current working directory of my google colab workspace. This made it convenient to run the file hassle-free everytime the session ended.

```
import os

os.environ['KAGGLE_USERNAME'] = "snehagilada"
os.environ['KAGGLE_KEY'] = "10d84b24c61fca71746a2a25fca79978"
```

```
!kaggle competitions download -c new-york-city-taxi-fare-prediction
```

```
Downloading new-york-city-taxi-fare-prediction.zip to /content
100% 1.55G/1.56G [00:16<00:00, 117MB/s]
100% 1.56G/1.56G [00:16<00:00, 101MB/s]
```

Following code is to unzip the file and read the training and test csv as pandas dataframe. I have only taken 1% of the training data for the purpose of this project. Increasing the sample size might give better results.

```
!unzip new-york-city-taxi-fare-prediction.zip
```

```
Archive:  new-york-city-taxi-fare-prediction.zip
  inflating: GCP-Coupons-Instructions.rtf
  inflating: sample_submission.csv
  inflating: test.csv
  inflating: train.csv
```

```
!ls -lh
```

```
total 6.9G
-rw-r--r-- 1 root root  486 Dec 12  2019 GCP-Coupons-Instructions.rtf
-rw-r--r-- 1 root root 1.6G May 30 12:20 new-york-city-taxi-fare-prediction.zip
drwxr-xr-x 1 root root 4.0K May 25 13:42 sample_data
-rw-r--r-- 1 root root 336K Dec 12  2019 sample_submission.csv
-rw-r--r-- 1 root root 960K Dec 12  2019 test.csv
-rw-r--r-- 1 root root 5.4G Dec 12  2019 train.csv
```

```
import random
frac = 0.01
```

```
%%time

dtypes = {'fare_amount': 'float32',
          'pickup_longitude': 'float32',
          'pickup_latitude': 'float32',
          'dropoff_longitude': 'float32'}
```

```
train_df = pd.read_csv('train.csv',
                       header=0,
                       dtype = dtypes,
                       parse_dates= ['pickup_datetime'],
                       skiprows= lambda i: i>0 and random.random()>frac)
```

```
CPU times: user 2min 12s, sys: 4.35 s, total: 2min 16s
Wall time: 2min 55s
```

```
test_df = pd.read_csv('test.csv',
                      header=0,
                      dtype=dtypes,
                      parse_dates=['pickup_datetime'])
```

About the Dataset: This section shows the number of rows & columns, the column names and their data types. A sample of first 5 rows is shown to better understand the data

```
train_df.shape
```

```
(553933, 8)
```

```
train_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 553933 entries, 0 to 553932
Data columns (total 8 columns):
 #   Column             Non-Null Count   Dtype
---  ------             --------------   -----
 0   key                553933 non-null  object
 1   fare_amount        553933 non-null  float32
 2   pickup_datetime    553933 non-null  datetime64[ns, UTC]
 3   pickup_longitude   553933 non-null  float32
 4   pickup_latitude    553933 non-null  float32
 5   dropoff_longitude  553931 non-null  float32
 6   dropoff_latitude   553931 non-null  float64
 7   passenger_count    553933 non-null  int64
dtypes: datetime64[ns, UTC](1), float32(4), float64(1), int64(1), object(1)
memory usage: 25.4+ MB
```

```
train_df.head()
```

	key	fare_amount	pickup_datetime	pickup_longitude	pickup_latitude	dropoff_longitude
0	2014-05-01 09:12:00.000000198	7.000000	2014-05-01 09:12:00+00:00	-73.966202	40.767502	-73.98091
1	2010-05-17 07:44:00.00000096	17.299999	2010-05-17 07:44:00+00:00	-73.950974	40.785633	-74.01023
2	2009-09-20 11:32:00.0000008	11.300000	2009-09-20 11:32:00+00:00	-73.975868	40.790142	-74.00669
3	2011-11-15 13:07:50.0000006	49.799999	2011-11-15 13:07:50+00:00	-73.983849	40.761852	-73.78312
4	2011-12-03 20:56:00.000000132	10.500000	2011-12-03 20:56:00+00:00	-74.001968	40.734997	-73.98149

Data Cleaning & Pre-processing

In this section, we check for null values. Since there are negligible null values, we simply drop the rows with null values and move on.

```
train_df.isnull().sum()
```

```
key                  0
fare_amount          0
pickup_datetime      0
pickup_longitude     0
pickup_latitude      0
dropoff_longitude    2
dropoff_latitude     2
passenger_count      0
dtype: int64
```

```
train_df.dropna(inplace=True)
```

```
# After deletion of 3 rows with null values
train_df.shape
```

```
(553931, 8)
```

Remove Outliers: The descriptive statistics of all the features below show that the range of pickup and dropoff longitudes/latitudes is unrealistic (starting from -2498 to 3280). We need to filter out the outliers to ensure that the location data is reliable for further analysis.

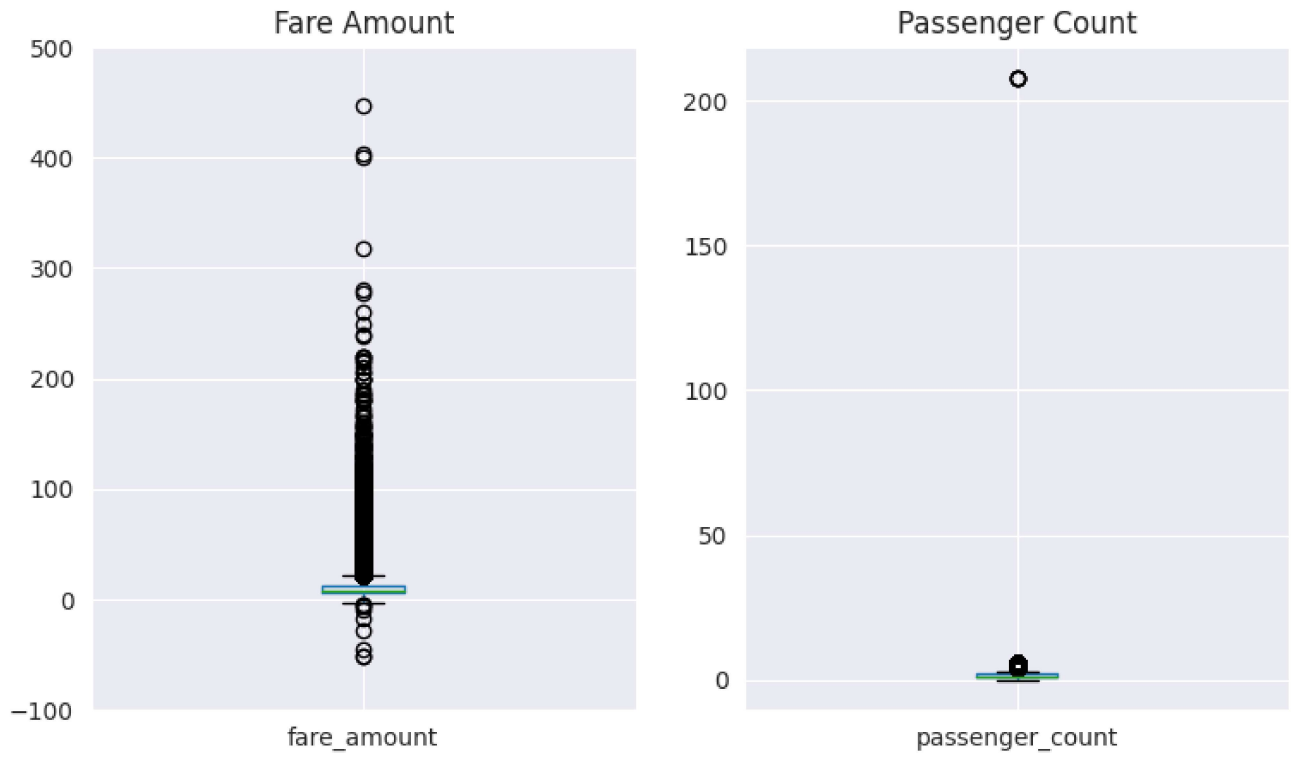
```
train_df.describe()
```

	fare_amount	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_
count	553931.000000	553931.000000	553931.000000	553931.000000	553931.000000	553931.(
mean	11.324110	-72.497375	39.939270	-72.521698	39.933612	1.6
std	9.718474	12.331344	9.843944	11.725052	8.994808	1.4
min	-52.000000	-1131.729980	-2503.788818	-2125.745361	-2498.918857	0.(
25%	6.000000	-73.992081	40.734917	-73.991379	40.733988	1.(
50%	8.500000	-73.981827	40.752670	-73.980186	40.753177	1.(
75%	12.500000	-73.967056	40.767185	-73.963692	40.768107	2.(
max	447.000000	3224.134277	3280.144043	2520.869873	2602.101893	208.(

Lets plot the features using a box plot to get the Q1, Q2 and Q3 values for each of them. Basis that, we can identify a realistic range of values for each feature

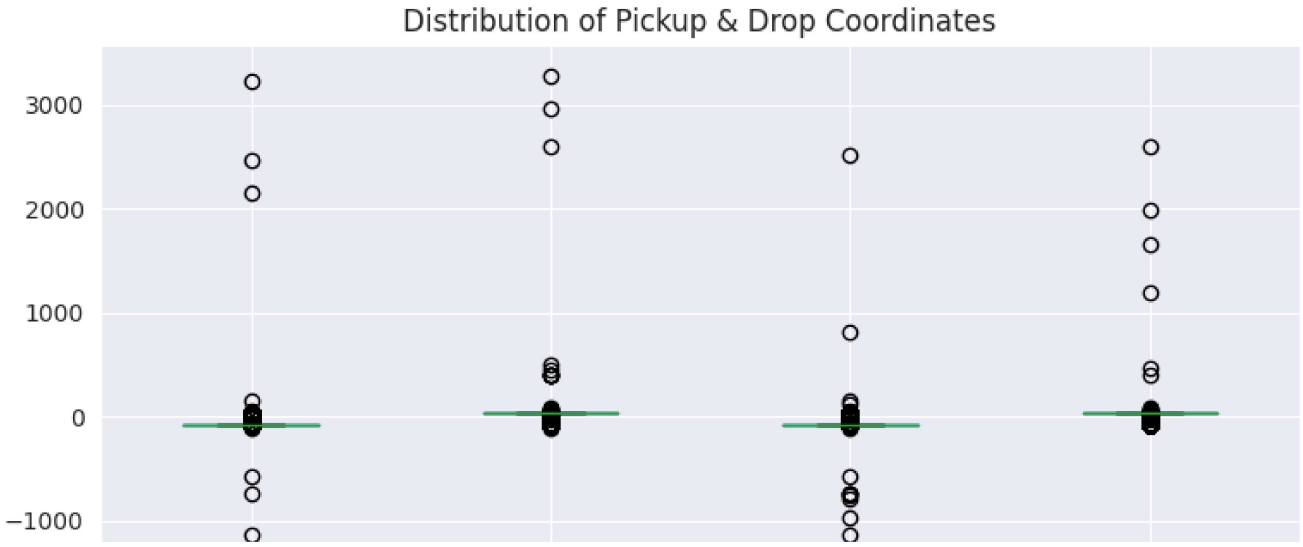
```
sns.set_style('darkgrid')
matplotlib.rcParams['font.size'] = 10
```

```
matplotlib.rcParams['figure.figsize'] = (9,5)
matplotlib.rcParams['figure.facecolor'] = '#00000000'


fig, ax = plt.subplots(1,2)

train_df.boxplot(column='fare_amount', ax=ax[0])
ax[0].set_title('Fare Amount')
ax[0].set_ylim(-100, 500)

train_df.boxplot(column='passenger_count', ax=ax[1])
ax[1].set_title('Passenger Count')

plt.show()
```



- The fare values which are less than 0 need to be removed.
- Similarly, passenger count may range from 1 to 6 based on the number of people a taxi can realistically acommodate

```
matplotlib.rcParams['figure.figsize'] = (9,5)

ax= train_df[['pickup_longitude', 'pickup_latitude', 'dropoff_longitude', 'dropoff_latitude']].boxplot()
ax.set_title('Distribution of Pickup & Drop Coordinates')

plt.show()
```



Here we notice that there are a few outliers in the longitude and latitude data which need to be removed. We can figure out the approximate range for these coordinates by analysing the minimum and maximum values for these features given in the test data:

```
test_df[['pickup_longitude', 'pickup_latitude', 'dropoff_longitude', 'dropoff_latitude']].describe()
```

	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude
count	9914.000000	9914.000000	9914.000000	9914.000000
mean	-73.974716	40.751041	-73.973656	40.751743
std	0.042774	0.033541	0.039072	0.035435
min	-74.252190	40.573143	-74.263245	40.568973
25%	-73.992500	40.736125	-73.991249	40.735254
50%	-73.982327	40.753052	-73.980015	40.754065
75%	-73.968012	40.767113	-73.964062	40.768757
max	-72.986534	41.709557	-72.990967	41.696683

```
print(' Range of longitude:', (min(test_df['pickup_longitude'].min(), test_df['dropoff_longitude'].min()),
                               max(test_df['pickup_longitude'].max(), test_df['dropoff_longitude'].max())))

print(' Range of latitude:', (min(test_df['pickup_latitude'].min(), test_df['dropoff_latitude'].min()),
                              max(test_df['pickup_latitude'].max(), test_df['dropoff_latitude'].max())))
```

```
      Range of longitude: (-74.263245, -72.986534)
      Range of latitude: (40.568973, 41.709557)
```

Based on the test data, we have arrived at values for minimum and maximum coordinates. We will now filter the outliers from the training data using the following code:

```
train_df = train_df[(train_df['pickup_longitude']>=-75) &
                    (train_df['pickup_longitude']<=-72) &
                    (train_df['dropoff_longitude']>=-75) &
                    (train_df['dropoff_longitude']<=-72) &
```

```
                    (train_df['pickup_latitude']>=40) &
                    (train_df['pickup_latitude']<=42) &
                    (train_df['dropoff_latitude']>=40) &
                    (train_df['dropoff_latitude']<=42) &
                    (train_df['fare_amount']>=1) &
                    (train_df['fare_amount']<=500) &
                    (train_df['passenger_count']>=1) &
                    (train_df['passenger_count']<=6)]
```

```
train_df.shape
```

```
(540373, 8)
```

```
train_df.describe()
```

	fare_amount	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_
count	540373.000000	540373.000000	540373.000000	540373.000000	540373.000000	540373.(
mean	11.315798	-73.975151	40.751053	-73.974312	40.751369	1.6
std	9.623745	0.039048	0.030043	0.038211	0.033246	1.3
min	1.110000	-74.927368	40.114994	-74.976364	40.016667	1.(
25%	6.000000	-73.992287	40.736549	-73.991577	40.735505	1.(
50%	8.500000	-73.982109	40.753410	-73.980629	40.753878	1.(
75%	12.500000	-73.968277	40.767605	-73.965347	40.768406	2.(
max	447.000000	-72.685089	42.000000	-72.358086	41.929782	6.(

The data now looks consistent and realistic.

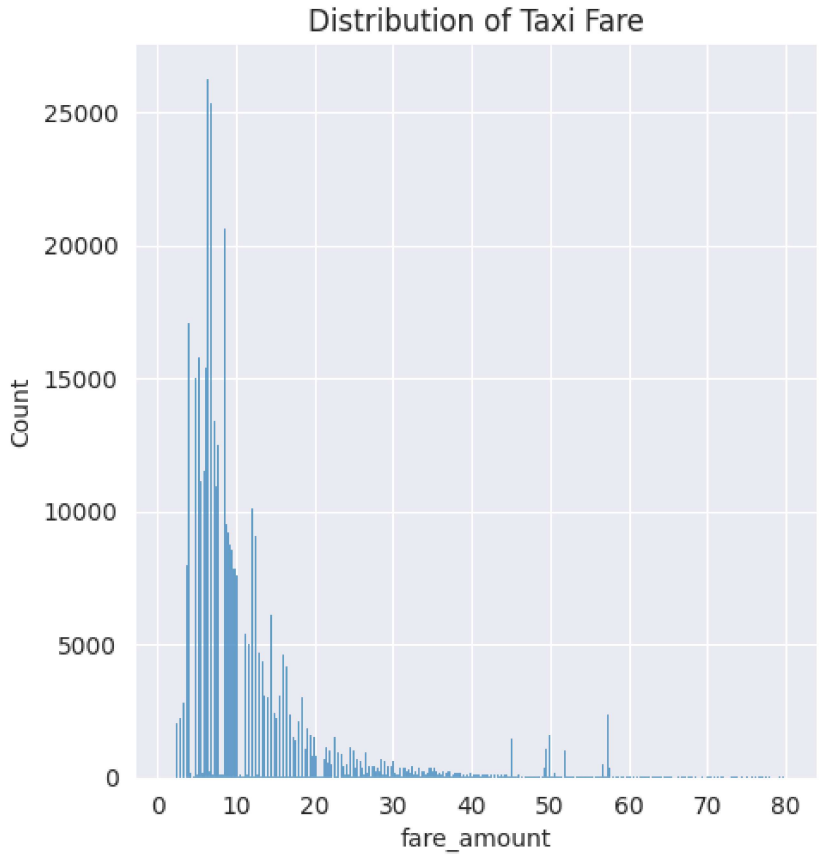
```
train_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 540373 entries, 0 to 553932
Data columns (total 8 columns):
 #   Column             Non-Null Count   Dtype
---  ------             --------------   -----
 0   key                540373 non-null  object
 1   fare_amount        540373 non-null  float32
 2   pickup_datetime    540373 non-null  datetime64[ns, UTC]
 3   pickup_longitude   540373 non-null  float32
 4   pickup_latitude    540373 non-null  float32
 5   dropoff_longitude  540373 non-null  float32
 6   dropoff_latitude   540373 non-null  float64
 7   passenger_count    540373 non-null  int64
dtypes: datetime64[ns, UTC](1), float32(4), float64(1), int64(1), object(1)
memory usage: 28.9+ MB
```

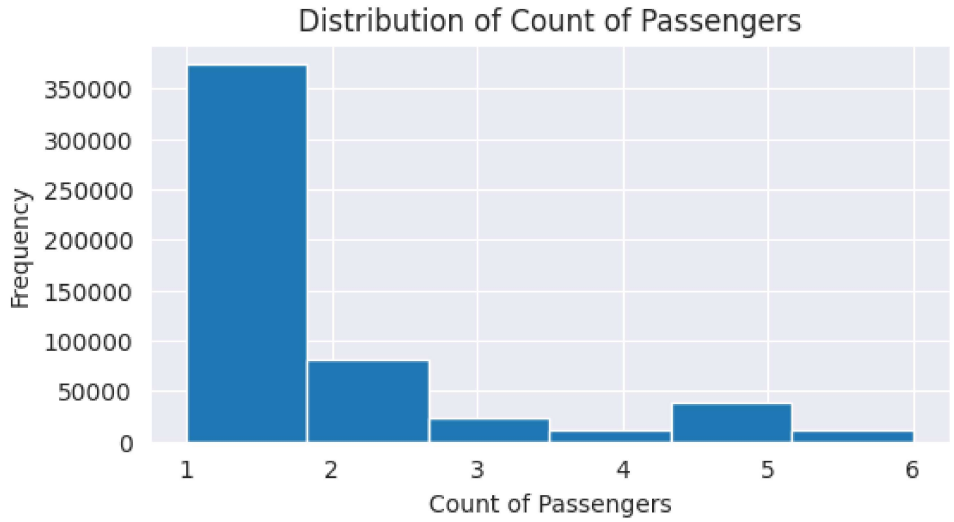
Feauture Engineering & EDA

```
# Distribution of trip fare
sns.displot(train_df[train_df['fare_amount']<80]['fare_amount'])
```

```
plt.title('Distribution of Taxi Fare')
plt.show()
```




```
# Distribution of count of passengers
plt.figure(figsize=(6,3))
plt.hist(train_df['passenger_count'], bins=6)
plt.xlabel('Count of Passengers')
plt.ylabel('Frequency')
plt.title('Distribution of Count of Passengers')

plt.show()
```



from the above plot, we can conclude that taxis are most frequently hired by single passengers

Add Trip Distance

A useful feature to predict the fare would be the distance travelled during the trip. Since the coordinates are on a sphere, a straight line distance between the pickup and drop coordinates is likely to be an inaccurate measure of the actual distance travelled.

We have used the haversine distance which gives the distance between two points on a sphere:

```
# Distance between pickup and drop coordinates

def haversine(lon1, lat1, lon2, lat2):
  """
  Calculate the distance between 2 points on a sphere in km
  """
  r = 6371            #Radius of Earth in km
  lon1, lat1, lon2, lat2 = map(np.radians, [lon1,lat1,lon2,lat2])
  delta_lon = lon2-lon1
  delta_lat = lat2-lat1

  #a = sin²((ϕB - ϕA)/2) + cos ϕA . cos ϕB . sin²((λB - λA)/2)
  a = np.sin(delta_lat/2)**2 + np.cos(lat1)*np.cos(lat2)*np.sin(delta_lon/2)**2

  #c = 2 * atan2( √a, √(1−a) )
  c = 2*np.arctan2(np.sqrt(a), np.sqrt(1-a))

  #d = r*c
  d = r*c
  return d


train_df['distance'] = haversine(train_df['pickup_longitude'], train_df['pickup_latitude'], train_df['dropoff


test_df['distance'] = haversine(test_df['pickup_longitude'], test_df['pickup_latitude'], test_df['dropoff_lor


test_df['distance'].describe()
```

```
count    9914.000000
mean        3.435374
std         3.972377
min         0.000009
25%         1.298096
50%         2.217061
75%         4.045468
max        99.996141
Name: distance, dtype: float64
```

Since the test data has trip distance only upto 100km, we can filter out larger distances from the training data as well.

```
train_df[train_df['distance']>100].shape[0]
```

```
15
```

```
train_df = train_df[train_df['distance']<=100]
```

Fare/km is another useful feature which will help us in exploratory data analysis

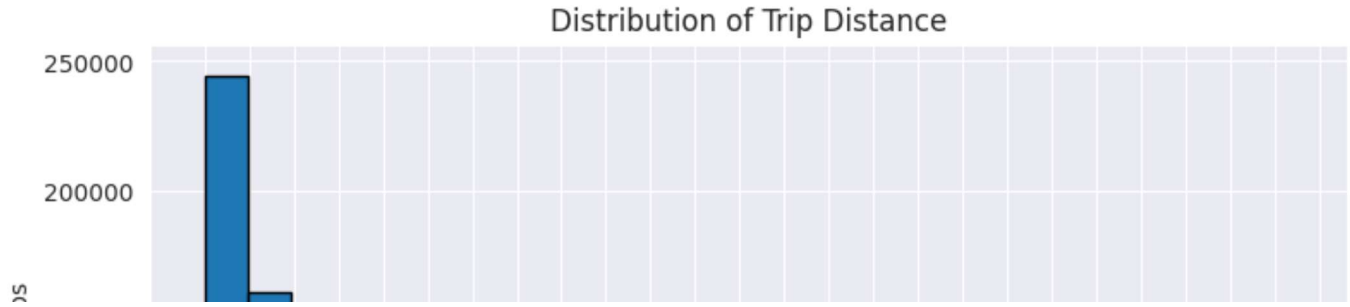
```
train_df['fare_per_km'] = np.where(train_df['distance']<1, train_df['fare_amount'], train_df['fare_amount']/t
```

```
<ipython-input-33-c3e87f8a2836>:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.
  train_df['fare_per_km'] = np.where(train_df['distance']<1, train_df['fare_amount'], train_df['fare_am
```

```
train_df.head()
```

	key	fare_amount	pickup_datetime	pickup_longitude	pickup_latitude	dropoff_longitude
0	2014-05-01 09:12:00.000000198	7.000000	2014-05-01 09:12:00+00:00	-73.966202	40.767502	-73.98091
1	2010-05-17 07:44:00.00000096	17.299999	2010-05-17 07:44:00+00:00	-73.950974	40.785633	-74.01023
2	2009-09-20 11:32:00.0000008	11.300000	2009-09-20 11:32:00+00:00	-73.975868	40.790142	-74.00669
3	2011-11-15 13:07:50.0000006	49.799999	2011-11-15 13:07:50+00:00	-73.983849	40.761852	-73.78312
4	2011-12-03 20:56:00.000000132	10.500000	2011-12-03 20:56:00+00:00	-74.001968	40.734997	-73.98149

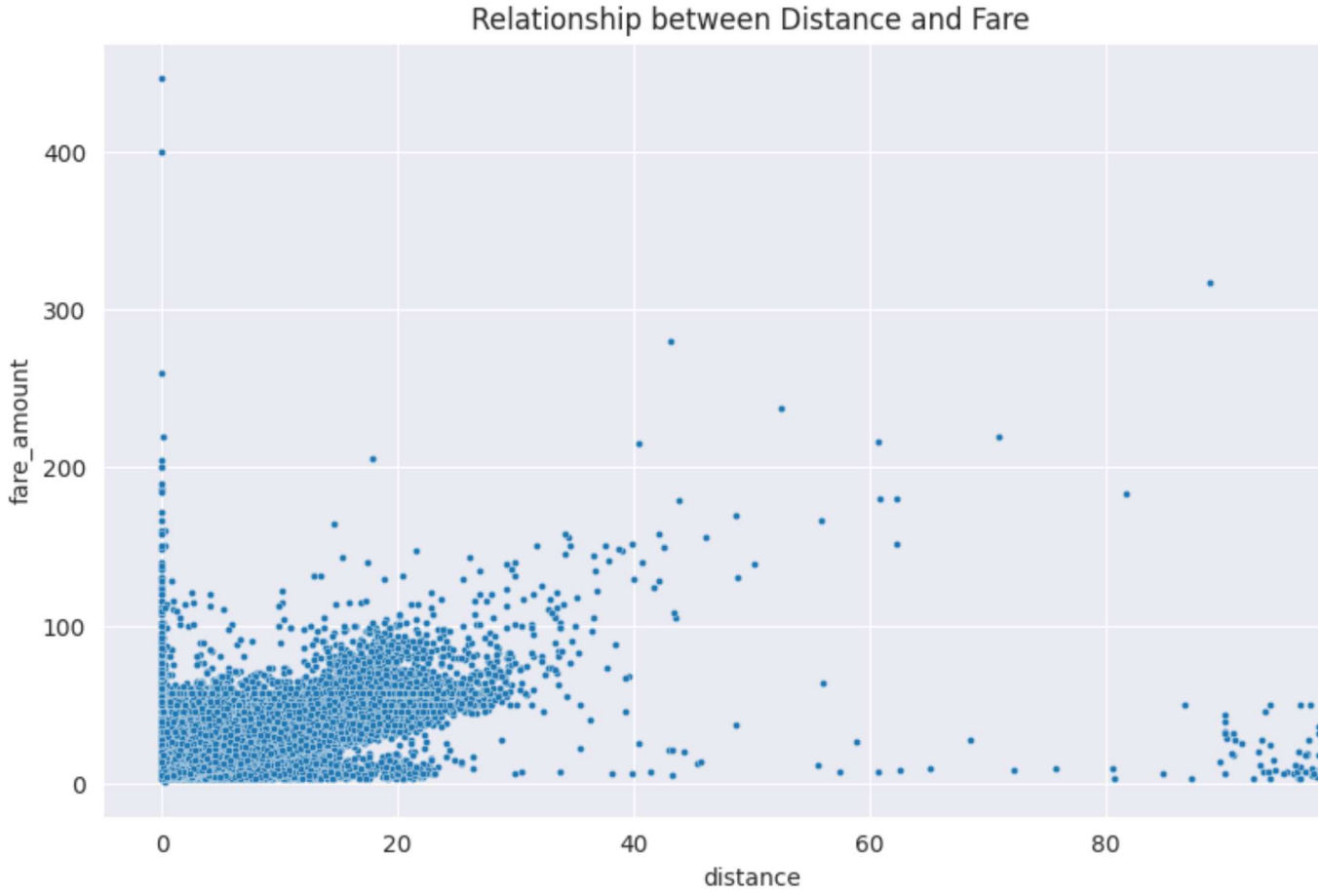
```
# Number of rides by trip_distance
ax = train_df[train_df['distance']<50]['distance'].hist(bins = 25, edgecolor = 'black')
ax.set_title('Distribution of Trip Distance')
ax.set_xlabel('Distance in km')
ax.set_ylabel('Number of Trips')
ax.set_xticks(np.arange(0,51,2))

plt.show()
```



It can be observed that almost 50% of the rides are of 0-2km distance. The number of rides keeps reducing as the distance increases. New Yorkers mostly travel short distances via cabs

```
plt.figure(figsize=(10,6))
sns.scatterplot(x='distance', y='fare_amount', data=train_df, s=10)
plt.title('Relationship between Distance and Fare')
```

```
Text(0.5, 1.0, 'Relationship between Distance and Fare')
```



- The general trend show that the fare amount increases with increase in distance but there are anomalies.
- There are many trips with negligible (almost 0) distance which have very high fares (these could be round trips so we do not have the exact distance travelled)
- There are a few trips with >90km distance which have pretty low fares

Add Datetime Features

Segregating the pickup_datetime feature into respective parts: Year, Month, Date, Day, Hour will help us further analyse the impact of these features on the fare:

```
data = [train_df, test_df]
for x in data:
  x['year'] = x['pickup_datetime'].dt.year
  x['month'] = x['pickup_datetime'].dt.month
  x['day'] = x['pickup_datetime'].dt.day
  x['day_of_week'] = x['pickup_datetime'].dt.dayofweek
  x['hour'] = x['pickup_datetime'].dt.hour
```

```
train_df[['year','month','day','day_of_week','hour']].describe()
```

	year	month	day	day_of_week	hour
count	540358.000000	540358.000000	540358.000000	540358.000000	540358.000000
mean	2011.736216	6.275734	15.734976	3.036822	13.515658
std	1.864855	3.438678	8.679333	1.951137	6.524253
min	2009.000000	1.000000	1.000000	0.000000	0.000000
25%	2010.000000	3.000000	8.000000	1.000000	9.000000
50%	2012.000000	6.000000	16.000000	3.000000	14.000000
75%	2013.000000	9.000000	23.000000	5.000000	19.000000
max	2015.000000	12.000000	31.000000	6.000000	23.000000

Lets replace the numeric values for the day_of_week & month to actual names of weekdays and months:

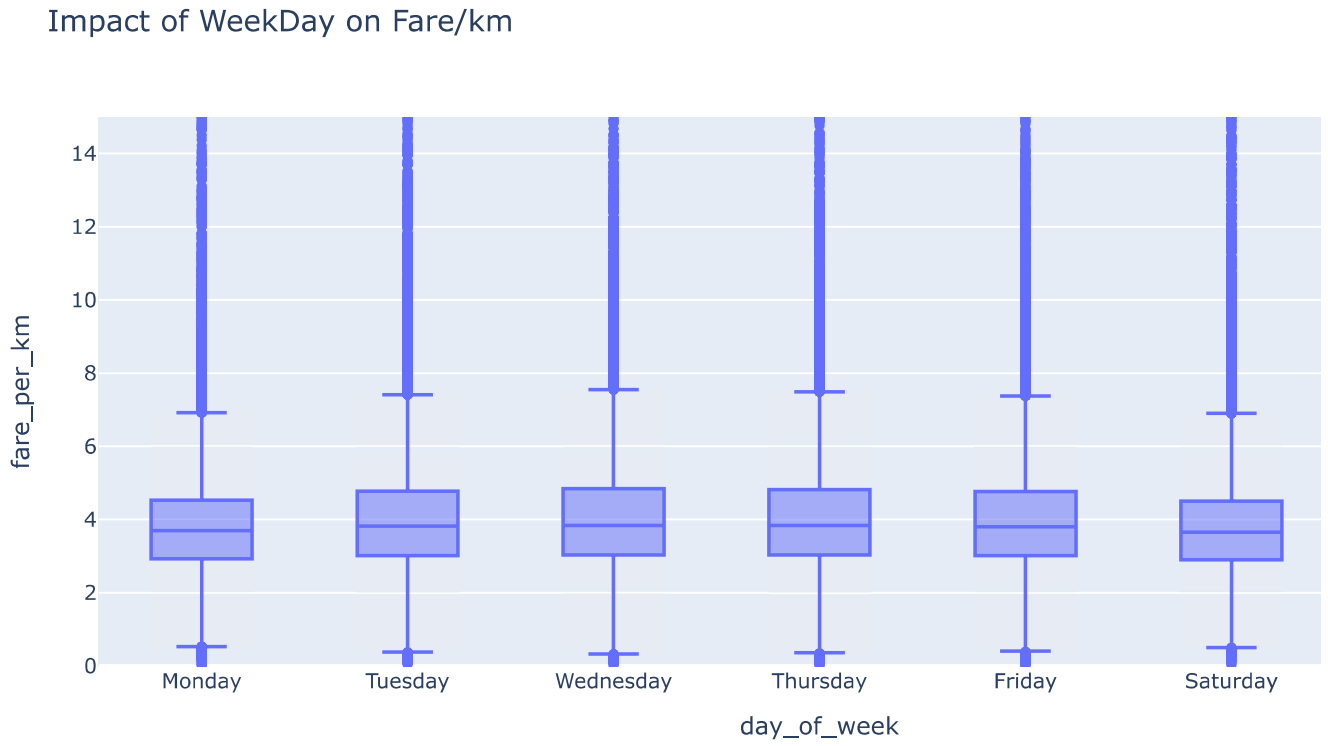
```
dayofweek_mapping = {
    0:'Monday',
    1:'Tuesday',
    2:'Wednesday',
    3:'Thursday',
    4:'Friday',
    5:'Saturday',
    6:'Sunday'
}

train_df['day_of_week'] = train_df['day_of_week'].replace(dayofweek_mapping)
test_df['day_of_week'] = test_df['day_of_week'].replace(dayofweek_mapping)
```

```
month_mapping = {
    1:'Jan',
    2:'Feb',
    3:'Mar',
    4:'Apr',
    5:'May',
    6:'Jun',
    7:'Jul',
    8:'Aug',
    9:'Sep',
    10:'Oct',
```

```
    11:'Nov',
    12:'Dec'
}

train_df['month'] = train_df['month'].replace(month_mapping)
test_df['month'] = test_df['month'].replace(month_mapping)
```

Using boxplots, lets find out if these datetime features cause any significant variation in the fare:

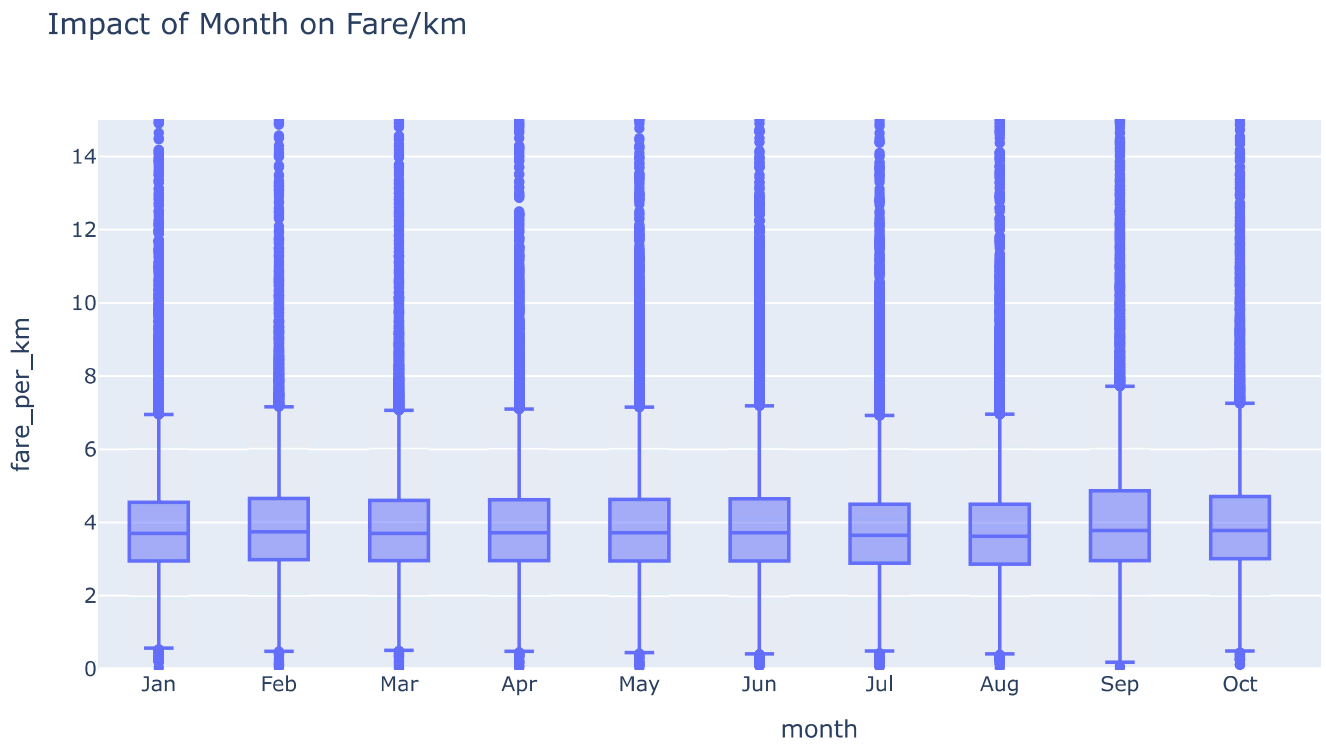
```
#Impact of weekday on fare
category_order = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
fig = px.box(train_df, x='day_of_week', y='fare_per_km',
             category_orders={'day_of_week': category_order},
             title='Impact of WeekDay on Fare/km')
fig.update_yaxes(range=[0,15])
fig.update_layout(height=500, width=1000)
fig.show()
```



- Fares on Saturday, Sunday and Monday seem to be slightly lower that other days - but overall there's not much of an impact.

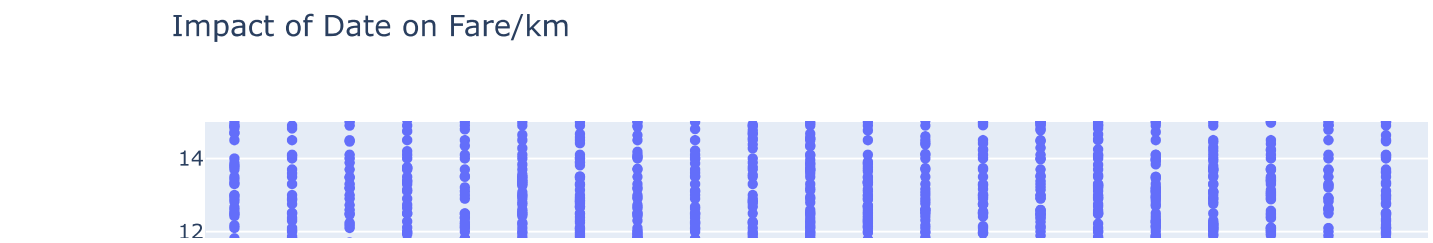
```
#Impact of month on fare
month_order = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
fig = px.box(train_df, x='month', y='fare_per_km',
             category_orders={'month': month_order},
             title='Impact of Month on Fare/km')
fig.update_yaxes(range=[0,15])
```

```
fig.update_layout(height=500, width=1000)
fig.show()
```



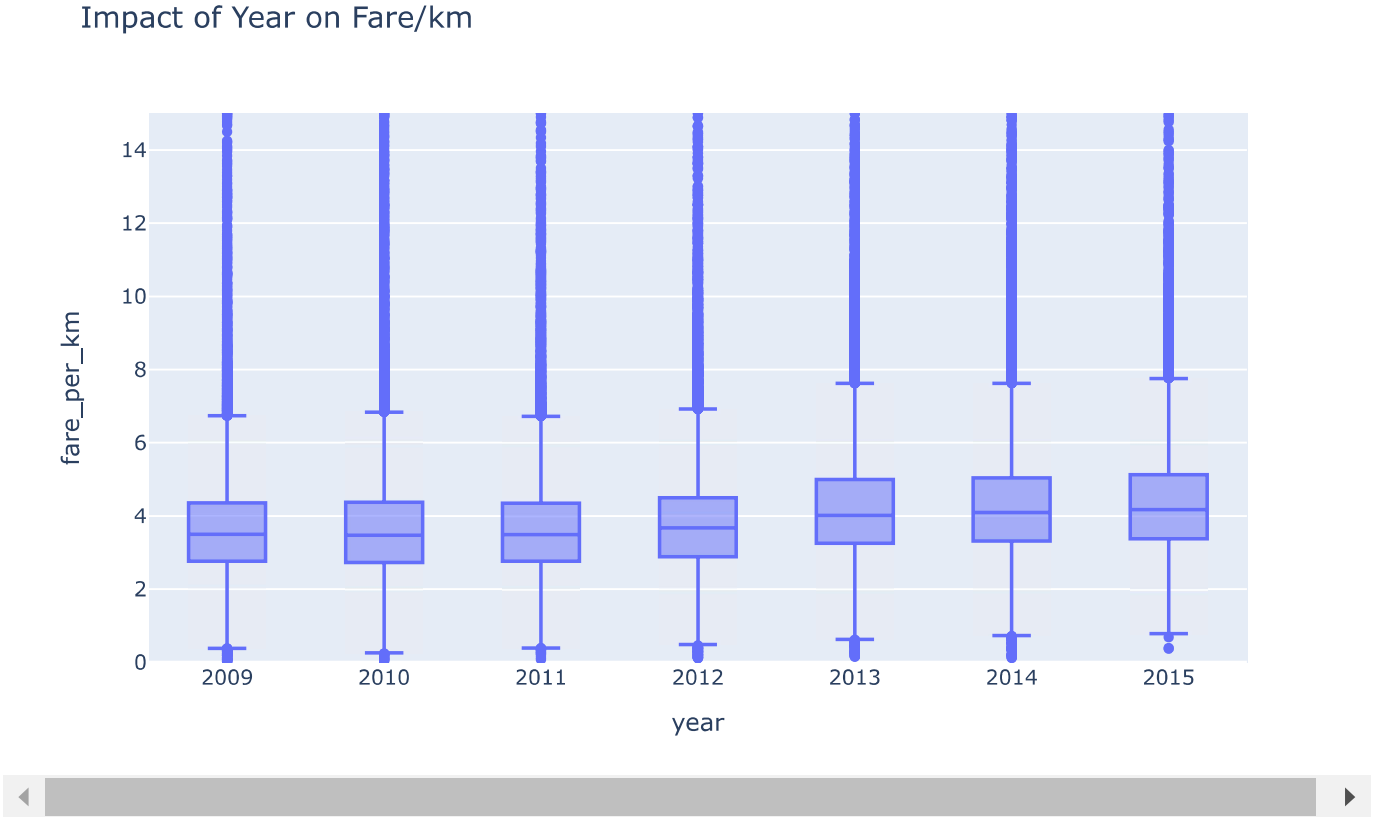
- Months do not seem to have any sizeable impact on the fare

```
#Impact of Date on Fare
fig = px.box(train_df, x='day', y='fare_per_km', title='Impact of Date on Fare/km')
fig.update_yaxes(range=[0,15])
fig.update_layout(height=500, width=1200)
fig.show()
```



The distribution of fare on all dates seems to be similar. So, day of the month doesn't have an impact on fare.

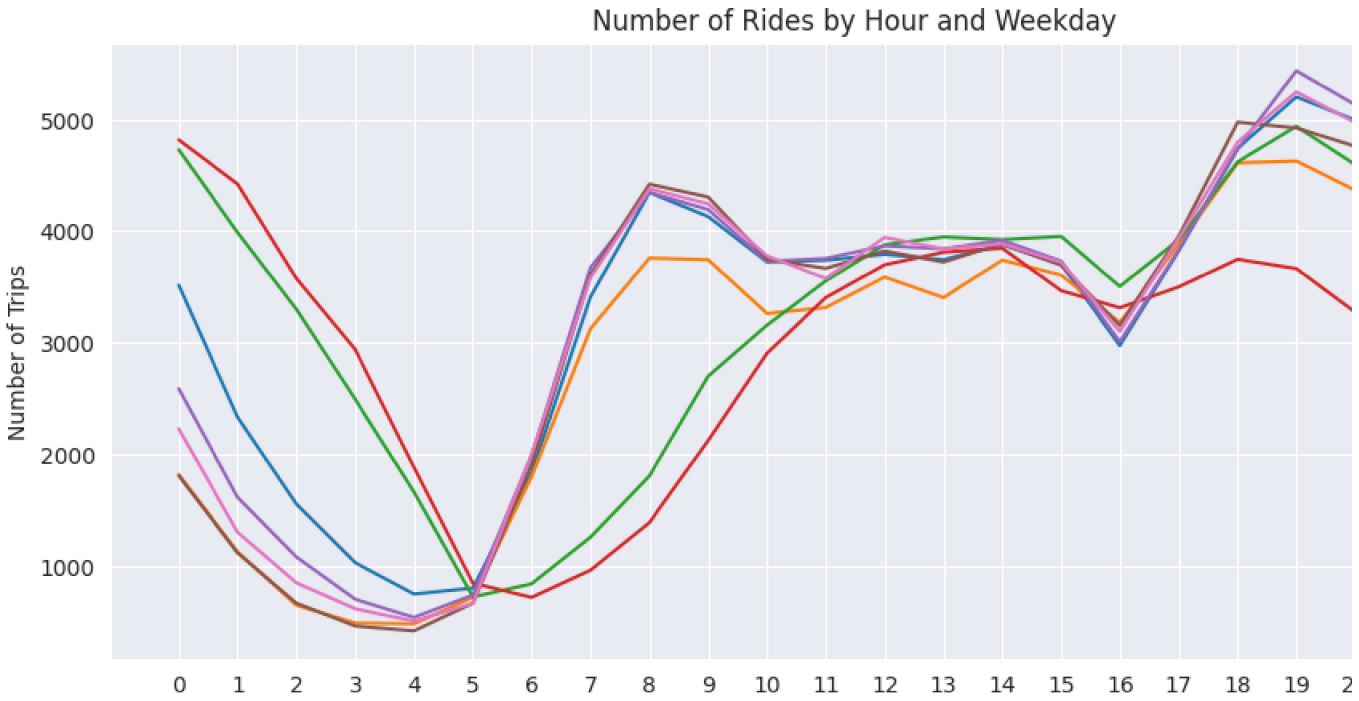
```
#Impact of year on fare
fig = px.box(train_df, x='year', y='fare_per_km', title='Impact of Year on Fare/km')
fig.update_yaxes(range=[0,15])
fig.update_layout(height=500, width=800)
fig.show()
```



- The mean fare seems to be incresing with each year. This could be attributed to inflation and the general increase in prices of goods and services year on year.

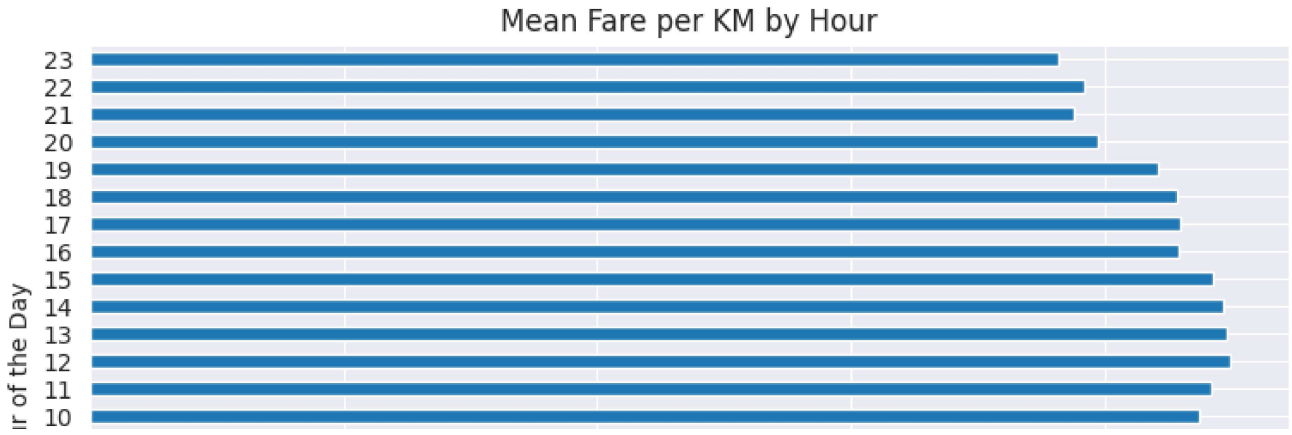
```
# Busiest hours of the day

data = pd.pivot_table(train_df, values = 'key', index = 'hour', columns='day_of_week', aggfunc='count')

matplotlib.rcParams['figure.figsize'] = (12,5)
ax = data.plot(kind='line')
ax.set_title('Number of Rides by Hour and Weekday')
ax.set_xlabel('Hour of the Day')
ax.set_ylabel('Number of Trips')
```

```
ax.set_xticks(np.arange(0,24,1))

order = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
handles, labels = ax.get_legend_handles_labels()
ax.legend(handles=[handles[labels.index(x)] for x in order], labels=order)

plt.show()
```



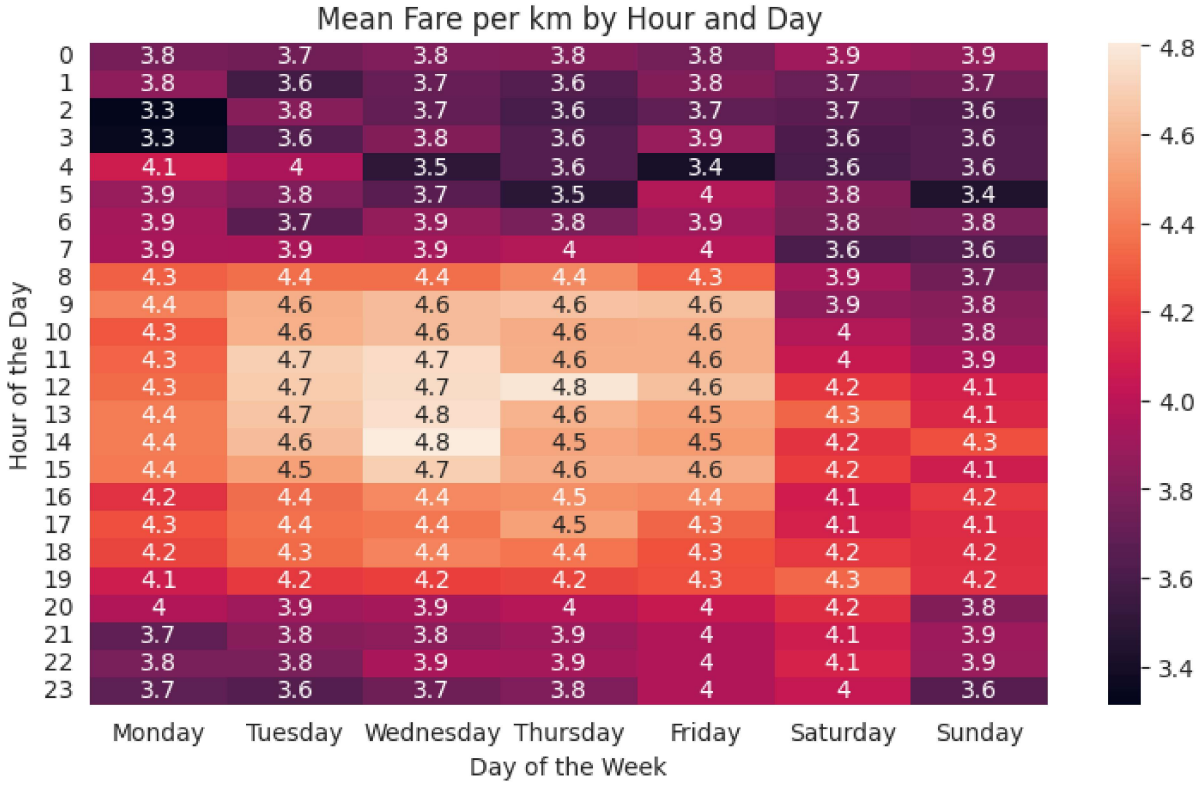
- 7am to 9am in the morning (except on saturday and sunday) and 6pm to 10pm in the evening (except on Sunday) seem to be the busiest hours of the day
- Friday and Saturday nights are busier than other days of the week

```
#  Relationship between hour of the day and fare/km

data = pd.DataFrame(train_df.groupby('hour').mean(numeric_only=True)['fare_per_km'])

matplotlib.rcParams['figure.figsize'] = (9,5)
data.plot(kind='barh')
plt.xlabel('Mean Fare per km')
plt.ylabel('Hour of the Day')
plt.title('Mean Fare per KM by Hour')

plt.show()
```



- fare/km is higher between 8 am to 7 pm than other times of the day. This could be because of working hours.
- So will this be true for all days of the week?


```
# Relationship between fare/km and hour/day

column_order = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']

matplotlib.rcParams['figure.figsize'] = (9,5)
data = pd.pivot_table(train_df, values = 'fare_per_km', index = 'hour', columns='day_of_week', aggfunc='mean'
sns.heatmap(data, annot=True)
plt.title('Mean Fare per km by Hour and Day')
plt.ylabel('Hour of the Day')
plt.xlabel('Day of the Week')
plt.show()
```



- The increase in fare/km during working hours (8 am to 7 pm) holds true only for weekdays.
- Friday and saturday evenings (6 pm to 10 pm) also show a spike in fare/km

After analysis of hour and weekday, we see that the fare per km is significantly higher during working hours on weekdays. We can add a feature signifying weather the timing of the ride falls within these hours:

```
# Adding Office Hours as a Feature
train_df['office_hours'] = np.where((train_df['hour']>=8) &
                                    (train_df['hour']<=18) &
                                    (~train_df['day_of_week'].isin(['Saturday', 'Sunday'])), 1, 0)


test_df['office_hours'] = np.where((test_df['hour']>=8) &
                                    (test_df['hour']<=18) &
                                    (~test_df['day_of_week'].isin(['Saturday', 'Sunday'])), 1, 0)
```

Since the month and date don't have any impact on the fare_amount, we can drop these features from the train and test dataset

```
train_df.drop(['day', 'month'], axis=1, inplace=True)
test_df.drop(['day', 'month'], axis=1, inplace=True)
```

Add Clusters for Pickup and Drop

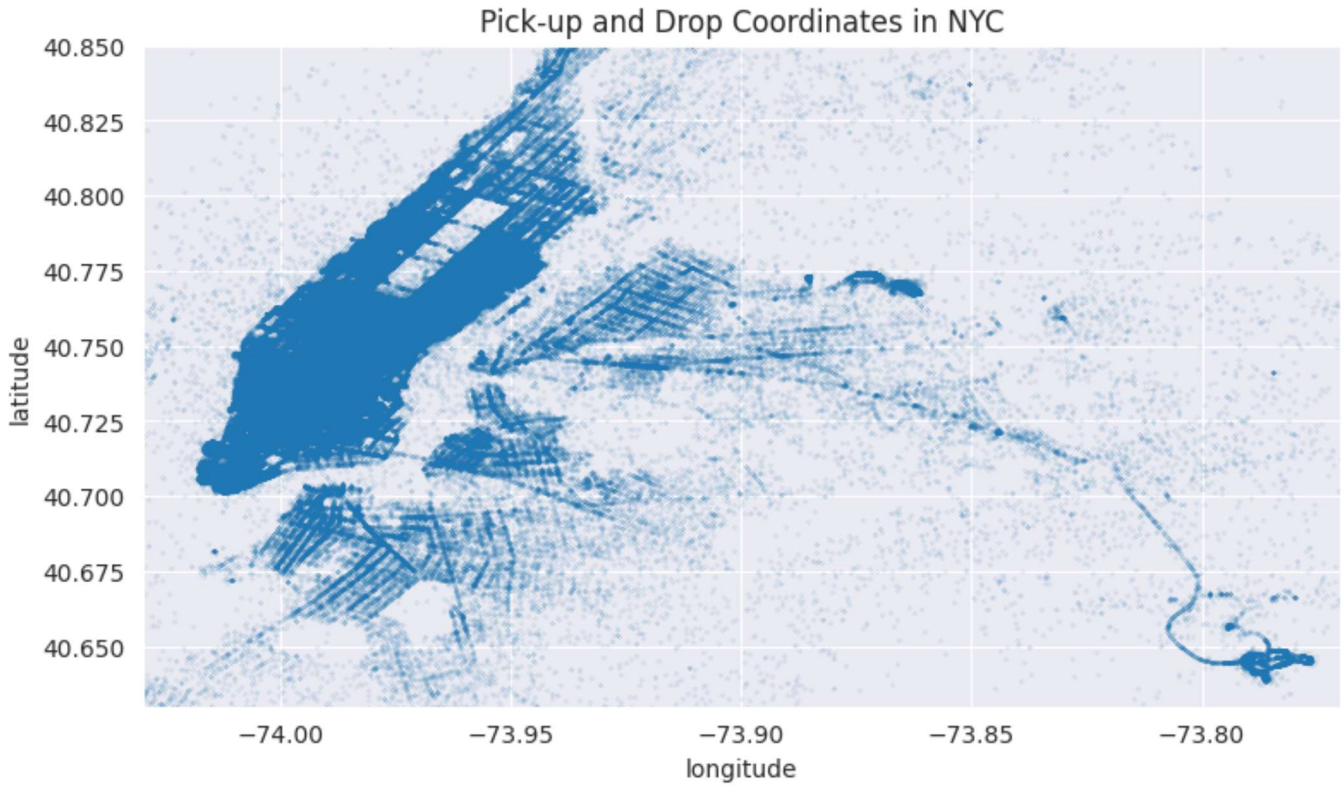
Lets start by visualising the spread of pickup and drop points across the city:

```
# Visualise the geographical spread of pickup and dropoff coordinates

lon_list = list(train_df['pickup_longitude']) + list(train_df['dropoff_longitude'])
lat_list = list(train_df['pickup_latitude']) + list(train_df['dropoff_latitude'])

lon_lat_data = {'longitude':lon_list, 'latitude':lat_list}

lon_lat_df = pd.DataFrame(lon_lat_data)

lon_lat_df.head()
```

	longitude	latitude
0	-73.966202	40.767502
1	-73.950974	40.785633
2	-73.975868	40.790142
3	-73.983849	40.761852
4	-74.001968	40.734997

```
ny_lat_border = [40.63, 40.85]
ny_lon_border = [-74.03, -73.77]

matplotlib.rcParams['figure.figsize'] = (9,5)

lon_lat_df.plot(kind='scatter', x = 'longitude', y = 'latitude', alpha=0.6, s=0.02)
plt.title('Pick-up and Drop Coordinates in NYC')
```

```
plt.ylim(ny_lat_border)
plt.xlim(ny_lon_border)

plt.show()
```

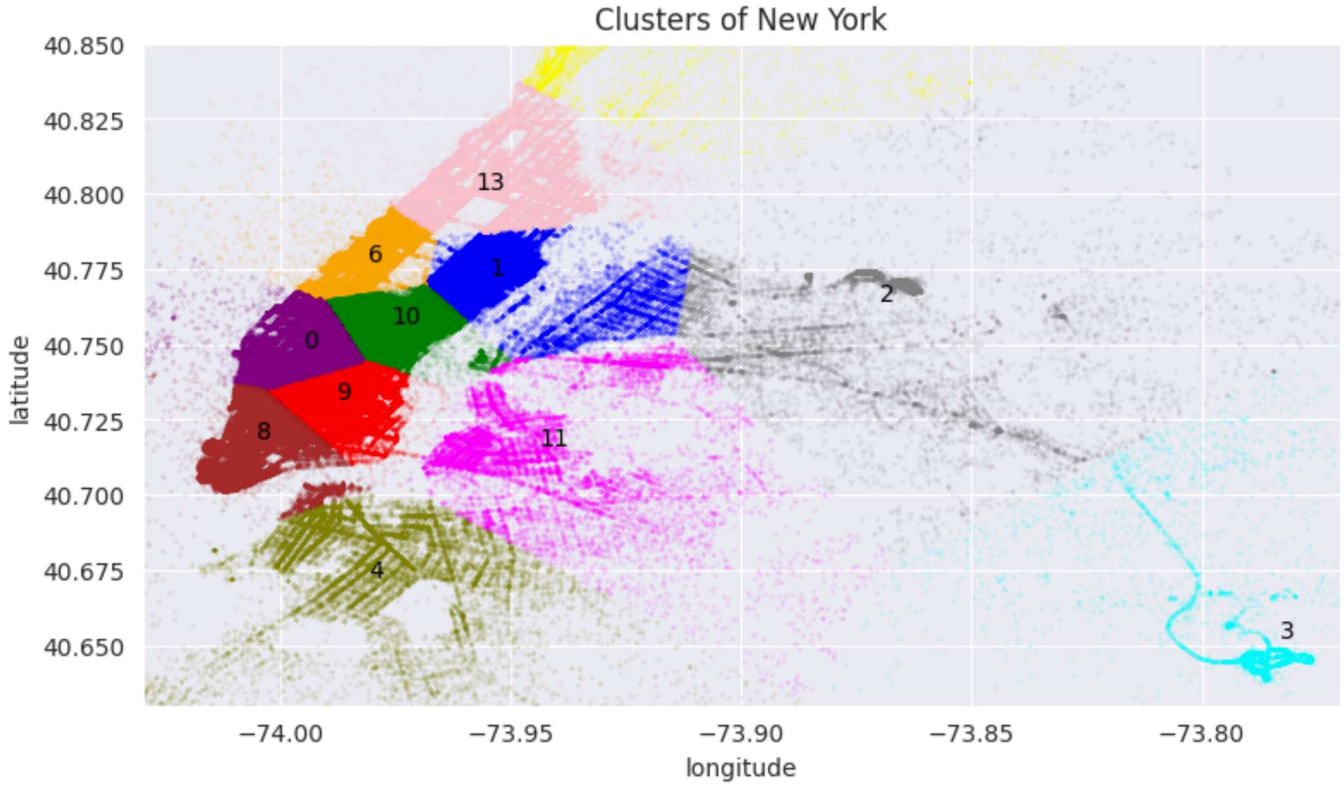


We see that the most dense area in terms of the number of pickups and drops is Manhattan.
Below is the map of 5 Boroughs of NYC for reference:

Other than the 5 boroughs of NYC (Manhattan, Bronx, Queens, Brooklyn and staten island), we also need to further divide Manhattan into major clusters. We'll use KMeans clustering algorithm to divide the city into 15 clulsters

```
# Clustering of coordinates using Kmeans
nyc_clusters = KMeans(n_clusters=15, random_state=42, n_init=10).fit(lon_lat_df)
```

```
lon_lat_df['cluster'] = nyc_clusters.labels_


matplotlib.rcParams['figure.figsize'] = (9,5)
colors = ['blue', 'orange', 'green', 'red', 'purple', 'brown', 'pink', 'gray', 'olive', 'cyan', 'magenta', 'y
clusters = list(lon_lat_df.cluster.unique())

fig, ax = plt.subplots()
for cluster, color in zip(clusters, colors):
  cluster_data = lon_lat_df[lon_lat_df['cluster']==cluster]
  cluster_data.plot(kind='scatter', x = 'longitude', y = 'latitude',alpha=0.6, s=0.02, color=color, ax=ax)

  centroid_lon = cluster_data['longitude'].mean()
  centroid_lat = cluster_data['latitude'].mean()
  ax.annotate(f'{cluster}', (centroid_lon, centroid_lat), fontsize=10, color='black')

plt.ylim(ny_lat_border)
plt.xlim(ny_lon_border)
plt.title('Clusters of New York')
plt.show()
```



Here, we see manhattan divided into its major neighbourhoods: harlem on top in pink; upper east side and upper west side on either side of central park in blue and orange; Wallstreet, lower east side, Chelsea and Midtown in brown, red, purple and green respectively.

JFK airport seems to have its own cluster(cyan) and so does Queens, brooklyn and Bronx. Staten islands doesn't seem to have enough number of rides to warrant a separate cluster.

Following are the indicative names of these clusters:

Please note: The names of the clusters have been manually given basis the positioning of cluster numbers on the map. In case the Kmeans algorithm is run again, the numbering of the clusters might change - resulting in mismatch

of the below names with the numbers

```
cluster_mapping= {0: 'Chelsea',
                  1:'Upper East Side',
                  2:'LaGuardia Airport',
                  3: 'JFK Airport',
                  4: 'Brooklyn',
                  5:  'Bronx',
                  6: 'Upper West Side',
                  7: 'Out of NYC1',
                  8: 'Soho Wallstreet',
                  9: 'Lower East Side',
                  10: 'Midtown',
                  11: 'Queens',
                  12: 'New Jersey',
                  13: 'Harlem',
                  14: 'Out of NYC2'}
```

Lets add these clusters as a feature to our training and test data

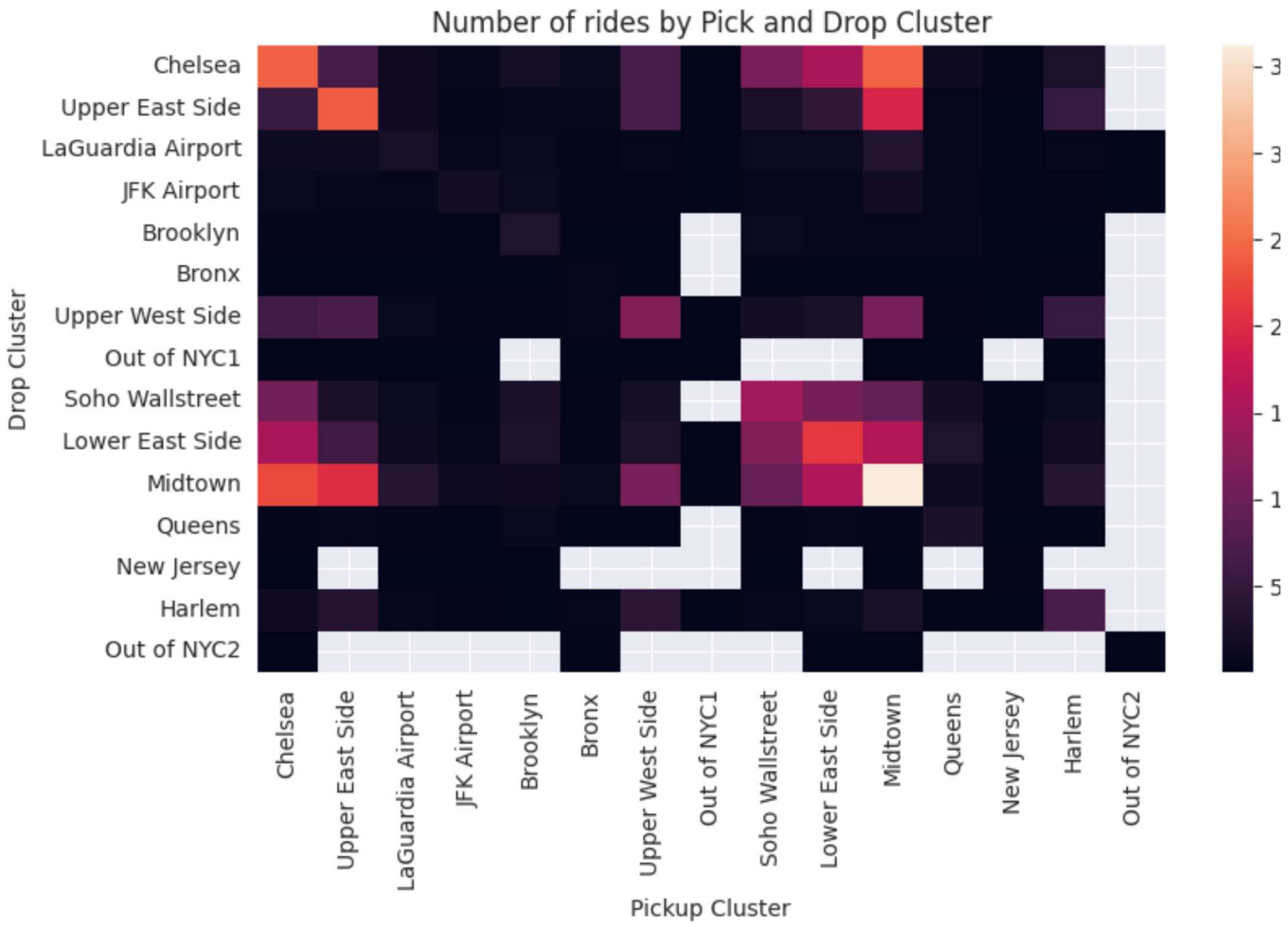
```
def add_clusters(x):
  '''
  To add pickup cluster and drop cluster as features to input dataframe(x)
  '''
  x2 = x[['pickup_longitude', 'pickup_latitude']].rename(columns={'pickup_longitude':'longitude', 'pickup_lat
  x3 = x[['dropoff_longitude', 'dropoff_latitude']].rename(columns={'dropoff_longitude':'longitude', 'dropoff
  x['pick_cluster'] = nyc_clusters.predict(x2)
  x['drop_cluster'] = nyc_clusters.predict(x3)


add_clusters(train_df)
add_clusters(test_df)


train_df[['pick_cluster', 'drop_cluster']].describe()
```

	pick_cluster	drop_cluster
count	540358.000000	540358.000000
mean	6.021382	6.164393
std	4.262289	4.277963
min	0.000000	0.000000
25%	1.000000	1.000000
50%	8.000000	8.000000
75%	10.000000	10.000000
max	14.000000	14.000000

Lets analyse the impact of these clusters on the number of rides, trip distance and fares:

```
#Number of rides across clusters

rides_by_cluster = pd.pivot_table(data = train_df, index = 'pick_cluster', columns='drop_cluster', values='ke
```

```
x_labels = cluster_mapping.values()
y_labels = cluster_mapping.values()
sns.heatmap(rides_by_cluster, xticklabels=x_labels, yticklabels=y_labels)
plt.title('Number of rides by Pick and Drop Cluster')
plt.ylabel('Drop Cluster')
plt.xlabel('Pickup Cluster')
plt.show()
```

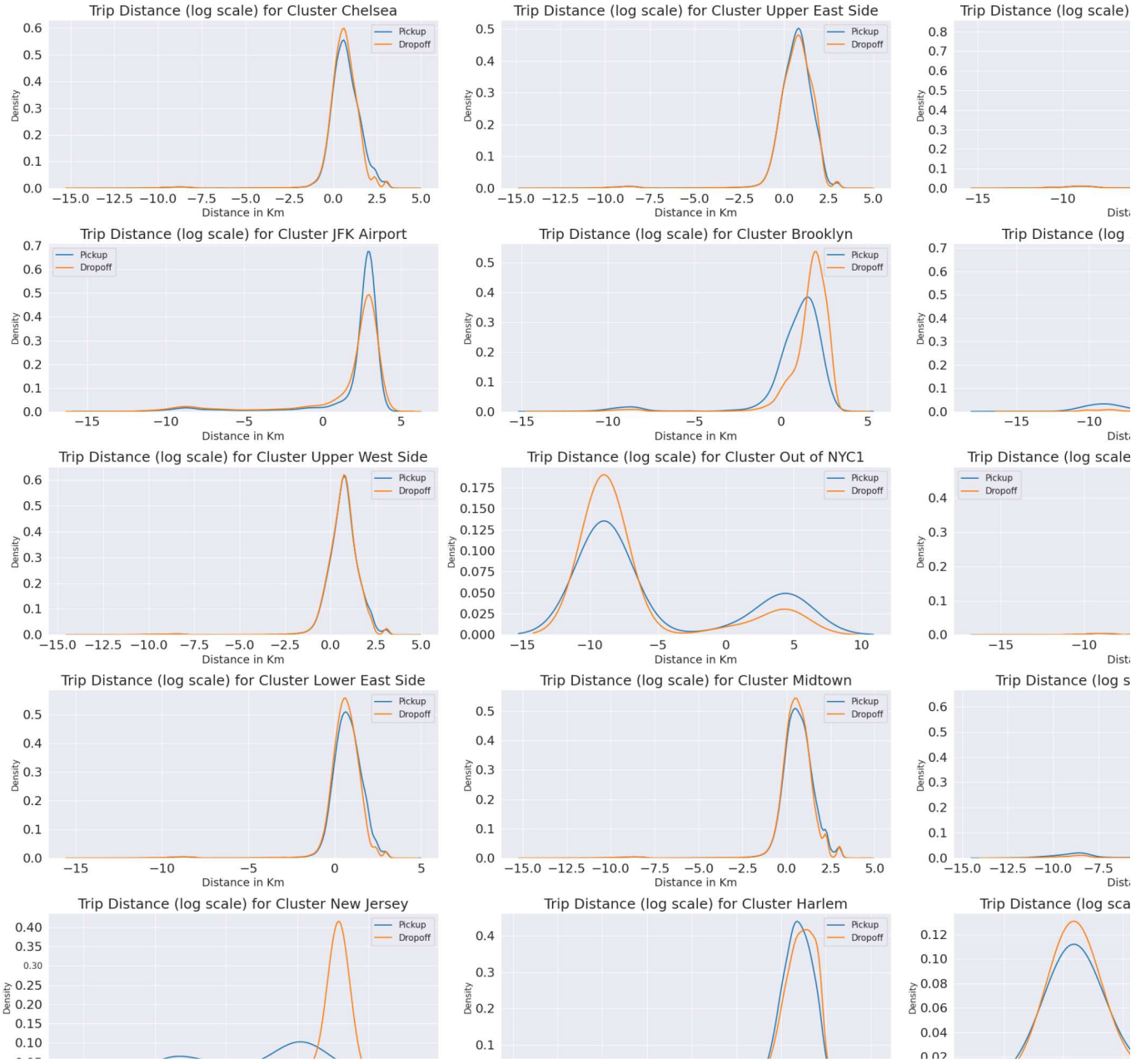


- Chelsea, Upper East Side, Upper West Side, Soho, Lower East Side and Midtown (Manhattan clusters) have the highest number of taxi rides. Even among them, clusters Midtown, Soho, Lower East Side, Chelsea (Lower Manhattan) are much busier than the rest of Manhattan.

```
# Trip Distances accross Clusters

plt.figure(figsize=(24,20))
plt.title("Distribution of Trip Distance Across Clusters")
i=1

for cluster in np.sort(train_df.pick_cluster.unique()):
  ax = plt.subplot(5,3,i)
  sns.kdeplot(np.log(train_df.loc[train_df['pick_cluster']==cluster]['distance'].values),label='Pickup')
  sns.kdeplot(np.log(train_df.loc[train_df['drop_cluster']==cluster]['distance'].values),label='Dropoff')
  plt.title("Trip Distance (log scale) for Cluster "+ cluster_mapping[cluster], fontdict={'fontsize':18})
  plt.xlabel("Distance in Km", fontsize=14)
  plt.xticks(fontsize=16)
```

```
    plt.yticks(fontsize=16)
    plt.legend()
    i += 1

plt.tight_layout()
plt.show()
```

```
<ipython-input-102-0293d995dd38>:8: MatplotlibDeprecationWarning:

Auto-removal of overlapping axes is deprecated since 3.6 and will be removed two minor releases later;
```

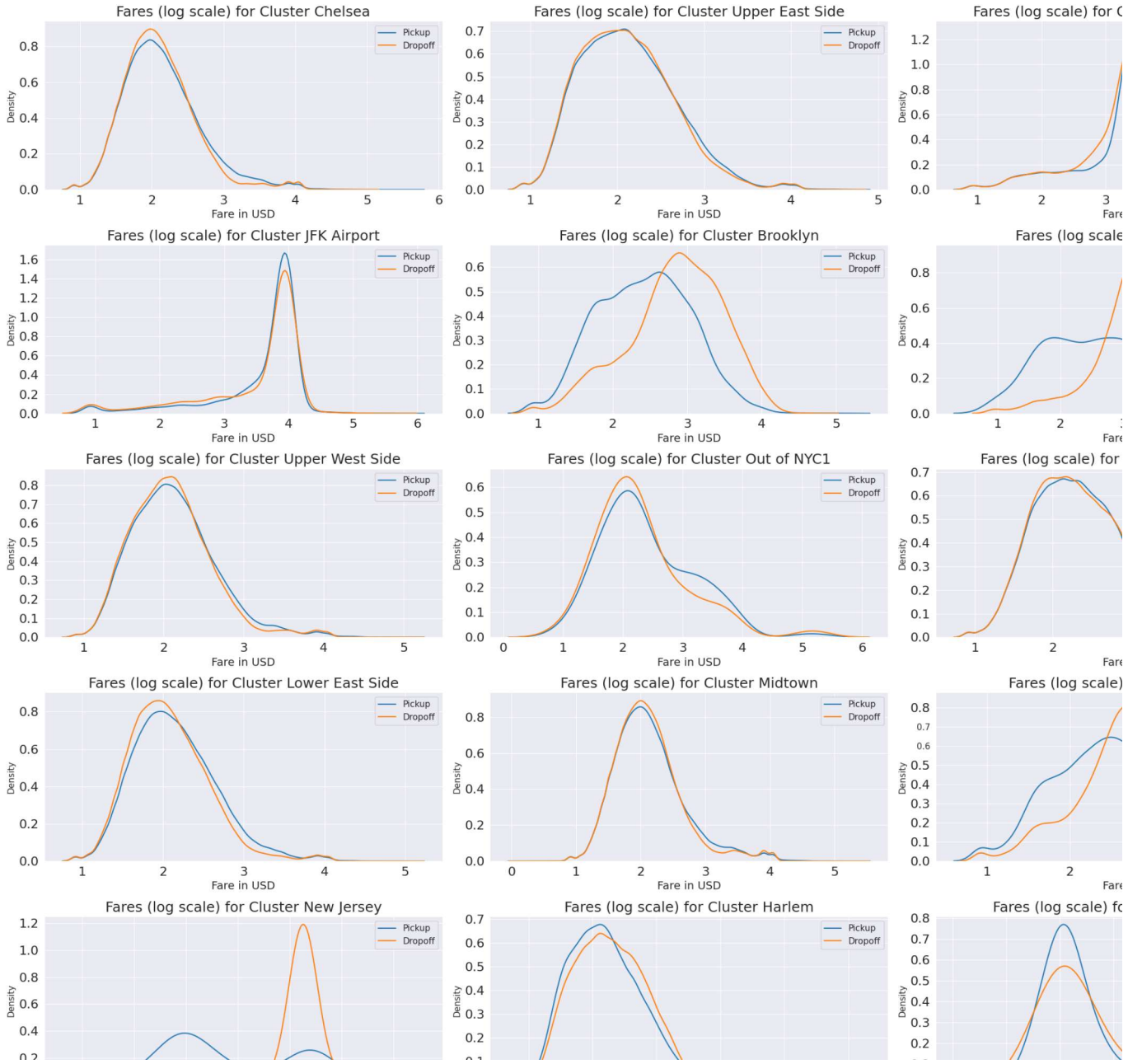


- Most of the Neighbourhoods within Manhattan (Chelsea, Upper East Side, Upper West Side, Lower East Side, Midtown, Harlem, Soho) and Queens have trip distances ranging from ~0.2km-12 km
- Clusters outside NYC have varied trip distances peaking at 0.3m (should be roundtrips) and 20km
- The airport clusters (JFK and LaGuardia) have similar spread of trip distances ~8km-50km

```
# Fares accross clusters
plt.figure(figsize=(24,20))
```

```
plt.title("Distribution of Fares Across Clusters")
i=1

for cluster in np.sort(train_df.pick_cluster.unique()):
  ax = plt.subplot(5,3,i)
  sns.kdeplot(np.log(train_df.loc[train_df['pick_cluster']==cluster]['fare_amount'].values),label='Pickup')
  sns.kdeplot(np.log(train_df.loc[train_df['drop_cluster']==cluster]['fare_amount'].values),label='Dropoff')
  plt.title("Fares (log scale) for Cluster "+ cluster_mapping[cluster], fontdict={'fontsize':18})
  plt.xlabel("Fare in USD", fontsize=14)
  plt.xticks(fontsize=16)
  plt.yticks(fontsize=16)
  plt.legend()
  i += 1

plt.tight_layout()
plt.show()
```

```
<ipython-input-103-aca75f9d41c0>:7: MatplotlibDeprecationWarning:

Auto-removal of overlapping axes is deprecated since 3.6 and will be removed two minor releases later;
```



- For all the clusters in Manhattan, the distribution of fare seems to be similar ranging from ~$2.7-$33 (peaking at ~$7). Wallstreet & Harlem have a much higher spread probabaly because they are the southern-most and northern-most neighbourhoods of Manhattan making trip distance to other parts much higher.
- For bigger boroughs like Queens, Bronx and Brooklyn, the spread is much wider ($2-$55) due to a much larger area of the borough
- The airport clusters have higher fares ranging from ~$33-$90 for JFK and ~$20-$50 for La Guardia

Train-Test Split

In this section we will split the training dataset into train and validation using the `train_test_split` functionality from `sklearn`

```
from sklearn.model_selection import train_test_split
```

```
train_data, val_data = train_test_split(train_df, test_size=0.2, random_state=42)
```

```
len(train_data), len(val_data)
```

```
(432286, 108072)
```

```
train_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 432286 entries, 371481 to 125013
Data columns (total 16 columns):
 #   Column             Non-Null Count   Dtype
---  ------             --------------   -----
 0   key                432286 non-null  object
 1   fare_amount        432286 non-null  float32
 2   pickup_datetime    432286 non-null  datetime64[ns, UTC]
 3   pickup_longitude   432286 non-null  float32
 4   pickup_latitude    432286 non-null  float32
 5   dropoff_longitude  432286 non-null  float32
 6   dropoff_latitude   432286 non-null  float64
 7   passenger_count    432286 non-null  int64
 8   distance           432286 non-null  float64
 9   fare_per_km        432286 non-null  float64
 10  year               432286 non-null  int64
 11  day_of_week        432286 non-null  object
 12  hour               432286 non-null  int64
 13  office_hours       432286 non-null  int64
 14  pick_cluster       432286 non-null  int32
 15  drop_cluster       432286 non-null  int32
dtypes: datetime64[ns, UTC](1), float32(4), float64(3), int32(2), int64(4), object(2)
memory usage: 46.2+ MB
```

Lets convert `office_hours`, `pick_cluster` and `drop_cluster` to categorical columns

```
train_data[['office_hours', 'pick_cluster', 'drop_cluster', 'day_of_week']] = train_data[['office_hours', 'pi
val_data[['office_hours', 'pick_cluster', 'drop_cluster', 'day_of_week']] = val_data[['office_hours', 'pick_c
test_df[['office_hours', 'pick_cluster', 'drop_cluster', 'day_of_week']] = test_df[['office_hours', 'pick_clu
```

```
train_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 432286 entries, 371481 to 125013
Data columns (total 16 columns):
 #   Column             Non-Null Count   Dtype
---  ------             --------------   -----
 0   key                432286 non-null  object
 1   fare_amount        432286 non-null  float32
 2   pickup_datetime    432286 non-null  datetime64[ns, UTC]
 3   pickup_longitude   432286 non-null  float32
 4   pickup_latitude    432286 non-null  float32
 5   dropoff_longitude  432286 non-null  float32
 6   dropoff_latitude   432286 non-null  float64
 7   passenger_count    432286 non-null  int64
 8   distance           432286 non-null  float64
 9   fare_per_km        432286 non-null  float64
 10  year               432286 non-null  int64
 11  day_of_week        432286 non-null  category
 12  hour               432286 non-null  int64
 13  office_hours       432286 non-null  category
 14  pick_cluster       432286 non-null  category
 15  drop_cluster       432286 non-null  category
dtypes: category(4), datetime64[ns, UTC](1), float32(4), float64(3), int64(3), object(1)
memory usage: 37.9+ MB
```

Baseline Model - Linear Regression

We'll try our baseline linear regression model with minimal features. This will help us understand if the features we have added are helpful.

```
train_df.columns
```

```
Index(['key', 'fare_amount', 'pickup_datetime', 'pickup_longitude',
       'pickup_latitude', 'dropoff_longitude', 'dropoff_latitude',
       'passenger_count', 'distance', 'fare_per_km', 'year', 'day_of_week',
       'hour', 'office_hours', 'pick_cluster', 'drop_cluster'],
      dtype='object')
```

```
#Identifying input and target columns

input_cols = ['pickup_longitude', 'pickup_latitude', 'dropoff_longitude', 'dropoff_latitude', 'passenger_cour
target_col = 'fare_amount'
```

```
#Preparing Training and Validation data

lr1_train_x = train_data[input_cols].copy()
lr1_val_x = val_data[input_cols].copy()
lr1_train_y = train_data[target_col].copy()
lr1_val_y = val_data[target_col].copy()
```

```
# Scale Numerical Values to (0,1) range using MinMaxScaler

scaler = MinMaxScaler()
scaler.fit(train_df[input_cols])
```

```
MinMaxScaler()
```

```
lr1_train_x.loc[:,input_cols] = scaler.transform(lr1_train_x.loc[:,input_cols])
lr1_val_x.loc[:,input_cols] = scaler.transform(lr1_val_x.loc[:,input_cols])


# Check the range of all the input columns
lr1_train_x.describe().loc[['min','max']]
```

	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
min	0.0	0.000000	0.0	0.053763	0.0
max	1.0	0.983533	1.0	1.000000	1.0

```
#Train a Linear Regression Model
from sklearn.linear_model import LinearRegression
lr1_model = LinearRegression().fit(lr1_train_x, lr1_train_y)

#Generate Predictions on training and validation data
lr1_pred_train = lr1_model.predict(lr1_train_x)
lr1_pred_val = lr1_model.predict(lr1_val_x)

#Compute training and validation loss
lr1_train_rmse = mean_squared_error(lr1_train_y, lr1_pred_train, squared=False)
lr1_val_rmse = mean_squared_error(lr1_val_y, lr1_pred_val, squared=False)

print(f'The RMSE loss for the training set is $ {lr1_train_rmse}')
print(f'The RMSE loss for the validation set is $ {lr1_val_rmse}')
```

```
The RMSE loss for the training set is $ 8.357364591301394
The RMSE loss for the validation set is $ 8.341472143530488
```

The Root Mean Squared Error(RMSE) for our base model is approximately 8.35

Ridge Regression with Additional Features

Identifying input and target columns:

```
input_cols = ['pickup_longitude',
              'pickup_latitude',
              'dropoff_longitude',
              'dropoff_latitude',
              'passenger_count',
              'distance',
              'year',
              'day_of_week',
```

```
                'hour',
                'office_hours',
                'pick_cluster',
                'drop_cluster']
target_col = 'fare_amount'
```

Prepare test and validation data:

```
lr2_train_x = train_data[input_cols].copy()
lr2_val_x = val_data[input_cols].copy()
lr2_train_y = train_data[target_col].copy()
lr2_val_y = val_data[target_col].copy()
```

Identify Numeric and categorical columns

```
numeric_cols = list(lr2_train_x.select_dtypes(exclude=['category']).columns)
categorical_cols = list(lr2_train_x.select_dtypes(include=['category']).columns)

print('Numeric Columns: ', numeric_cols)
print('Categorical Columns: ', categorical_cols)
```

```
Numeric Columns:  ['pickup_longitude', 'pickup_latitude', 'dropoff_longitude', 'dropoff_latitude', 'pas
Categorical Columns:  ['day_of_week', 'office_hours', 'pick_cluster', 'drop_cluster']
```

Scale numeric columns to (0,1) range:

```
scaler = MinMaxScaler()
scaler.fit(train_df[numeric_cols])
```

```
MinMaxScaler
MinMaxScaler()
```

```
lr2_train_x.loc[:,numeric_cols] = scaler.transform(lr2_train_x.loc[:,numeric_cols])
lr2_val_x.loc[:,numeric_cols] = scaler.transform(lr2_val_x.loc[:,numeric_cols])
```

```
lr2_train_x.describe().loc[['min','max']]
```

	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count	distance
min	0.0	0.000000	0.0	0.053763	0.0	0.000000
max	1.0	0.983533	1.0	1.000000	1.0	0.990515

Encode categorical columns:

```
from sklearn.preprocessing import OneHotEncoder
```

```
encoder = OneHotEncoder(sparse=False, handle_unknown='ignore')
encoder.fit(train_df[categorical_cols])
encoded_cols = list(encoder.get_feature_names_out(categorical_cols))
lr2_train_x[encoded_cols] = encoder.transform(lr2_train_x[categorical_cols])
lr2_val_x[encoded_cols] = encoder.transform(lr2_val_x[categorical_cols])
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_encoders.py:868: FutureWarning: `sparse`
  warnings.warn(
```

Train a ridge regression model:

```
from sklearn.linear_model import Ridge
lr2_model = Ridge()
lr2_model.fit(lr2_train_x[numeric_cols + encoded_cols], lr2_train_y)
```

```
Ridge
Ridge()
```

Make predictions on training and validation data:

```
lr2_pred_train = lr2_model.predict(lr2_train_x[numeric_cols + encoded_cols])
lr2_pred_val = lr2_model.predict(lr2_val_x[numeric_cols + encoded_cols])
```

Calculate the RMSE and evaluate the model's performance:

```
lr2_train_rmse = mean_squared_error(lr2_train_y, lr2_pred_train, squared=False)
lr2_val_rmse = mean_squared_error(lr2_val_y, lr2_pred_val, squared=False)

print(f'The RMSE loss for the training set is $ {lr2_train_rmse}')
print(f'The RMSE loss for the validation set is $ {lr2_val_rmse}')
```

```
The RMSE loss for the training set is $ 4.8185444485644116
The RMSE loss for the validation set is $ 4.937705898804966
```

The Ridge regression model is off by $4.94 which is much better than our baseline model RMSE of 8.35.

Lets make predictions on test data for a submission on Kaggle:

```
#Prepare test data
lr2_test_df = test_df.copy()
lr2_test_df[numeric_cols] = scaler.transform(lr2_test_df[numeric_cols])
lr2_test_df[encoded_cols] = encoder.transform(lr2_test_df[categorical_cols])
```

```
# Make predictions on test data
lr2_pred_test = lr2_model.predict(lr2_test_df[numeric_cols + encoded_cols])
```

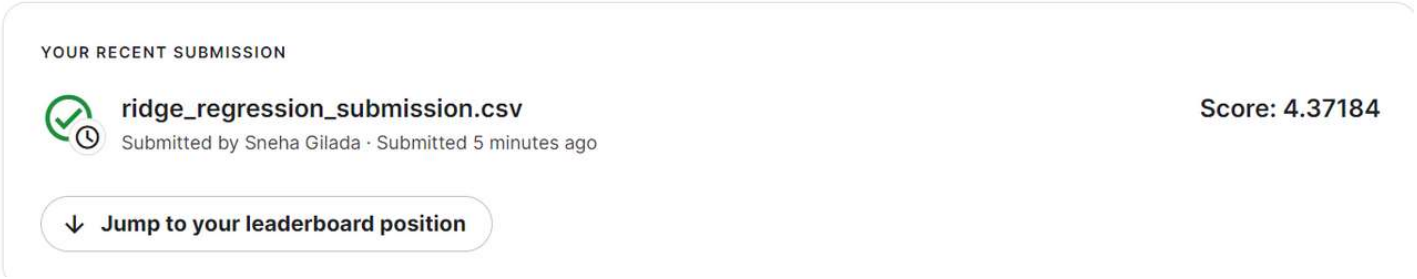
```
lr2_pred_test
```

```
array([11.30450341, 10.99796128,  5.07629634, ..., 48.17756685,
       20.79173661,  8.74352944])
```

```
# Sample submission format
submission_df = pd.read_csv('sample_submission.csv')


submission_df.head()
```

	key	fare_amount
0	2015-01-27 13:08:24.0000002	11.35
1	2015-01-27 13:08:24.0000003	11.35
2	2011-10-08 11:53:44.0000002	11.35
3	2012-12-01 21:12:12.0000002	11.35
4	2012-12-01 21:12:12.0000003	11.35

```
# Replace fare_amount column with our predictions
submission_df['fare_amount'] = lr2_pred_test


# Save results and upload on Kaggle
submission_df.to_csv('ridge_regression_submission.csv', index = None)
```



Random Forest

Since we have already scaled numeric features/encoded categorical features for the previous model, we will use them as the standard datasets for training, validation and testing for all future models:

```
# Standardising the training, validation and test data for all models

train_x = lr2_train_x[numeric_cols + encoded_cols].copy()
val_x = lr2_val_x[numeric_cols + encoded_cols].copy()
train_y = train_data[target_col].copy()
val_y = val_data[target_col].copy()
test = lr2_test_df[numeric_cols + encoded_cols].copy()


from sklearn.ensemble import RandomForestRegressor
```

Now we'll see how a basic random forest model performs on our data without any hyperparameter tuning:

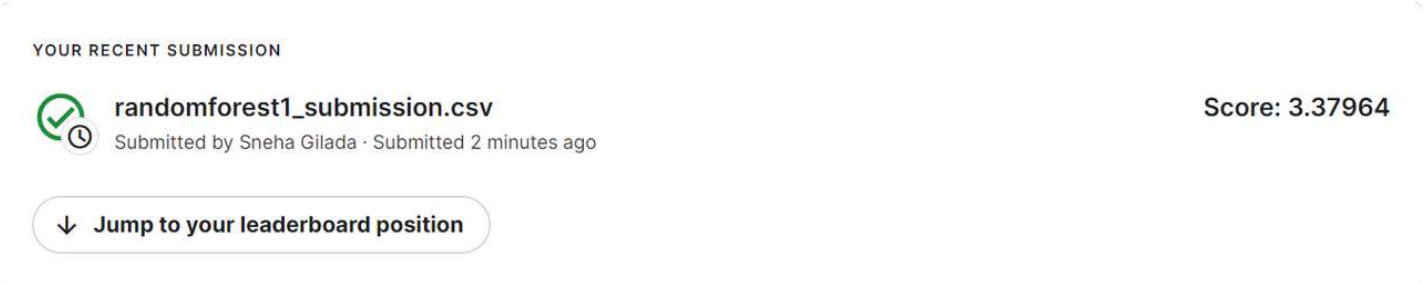
```
#Train a RandomForestRegressor Model
rf1_model = RandomForestRegressor(max_depth=10, n_jobs=-1, n_estimators=100, min_samples_leaf=5, max_features
rf1_model.fit(train_x, train_y)
```

```
RandomForestRegressor(max_depth=10, max_features=0.5, min_samples_leaf=5,
                      n_jobs=-1, random_state=42)
```

```
#Make predictions on train and validation data - calculate RMSE
rf1_pred_train = rf1_model.predict(train_x)
rf1_pred_val = rf1_model.predict(val_x)
rf1_train_rmse = mean_squared_error(train_y, rf1_pred_train, squared=False)
rf1_val_rmse = mean_squared_error(val_y, rf1_pred_val, squared=False)
print(f'The RMSE loss for the training set is $ {rf1_train_rmse}')
print(f'The RMSE loss for the validation set is $ {rf1_val_rmse}')
```

```
The RMSE loss for the training set is $ 3.8075258042291122
The RMSE loss for the validation set is $ 3.8823788008805864
```

```
# Predict on Test data and submit on Kaggle

rf1_pred_test = rf1_model.predict(test)


submission_df['fare_amount'] = rf1_pred_test
submission_df.to_csv('randomforest1_submission.csv', index=None)
```



- The training rmse has improved from 4.82 to 3.81
- The validation rmse has improved from 4.94 to 3.88
- The test score has improved from 4.37 to 3.37

Hyperparameter Tuning with Random Forest

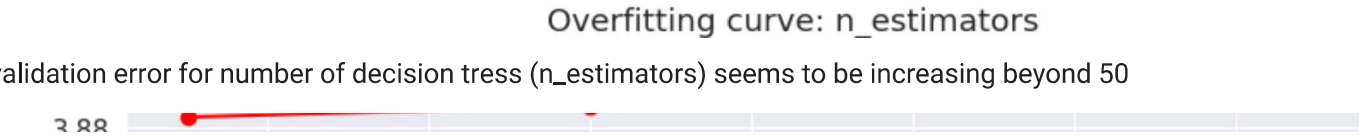
Lets create a function to create submission csv so that we don't have to repeat the same commands:

```
def submission(model, filename):
  '''
  To predict the fare_amount for test data and export
  the final csv file for submission on kaggle
  '''
  test_preds = model.predict(test)
  submission_df = pd.read_csv('sample_submission.csv')
```

```
  submission_df['fare_amount'] = test_preds
  submission_df.to_csv(filename, index=None)
  return submission_df
```

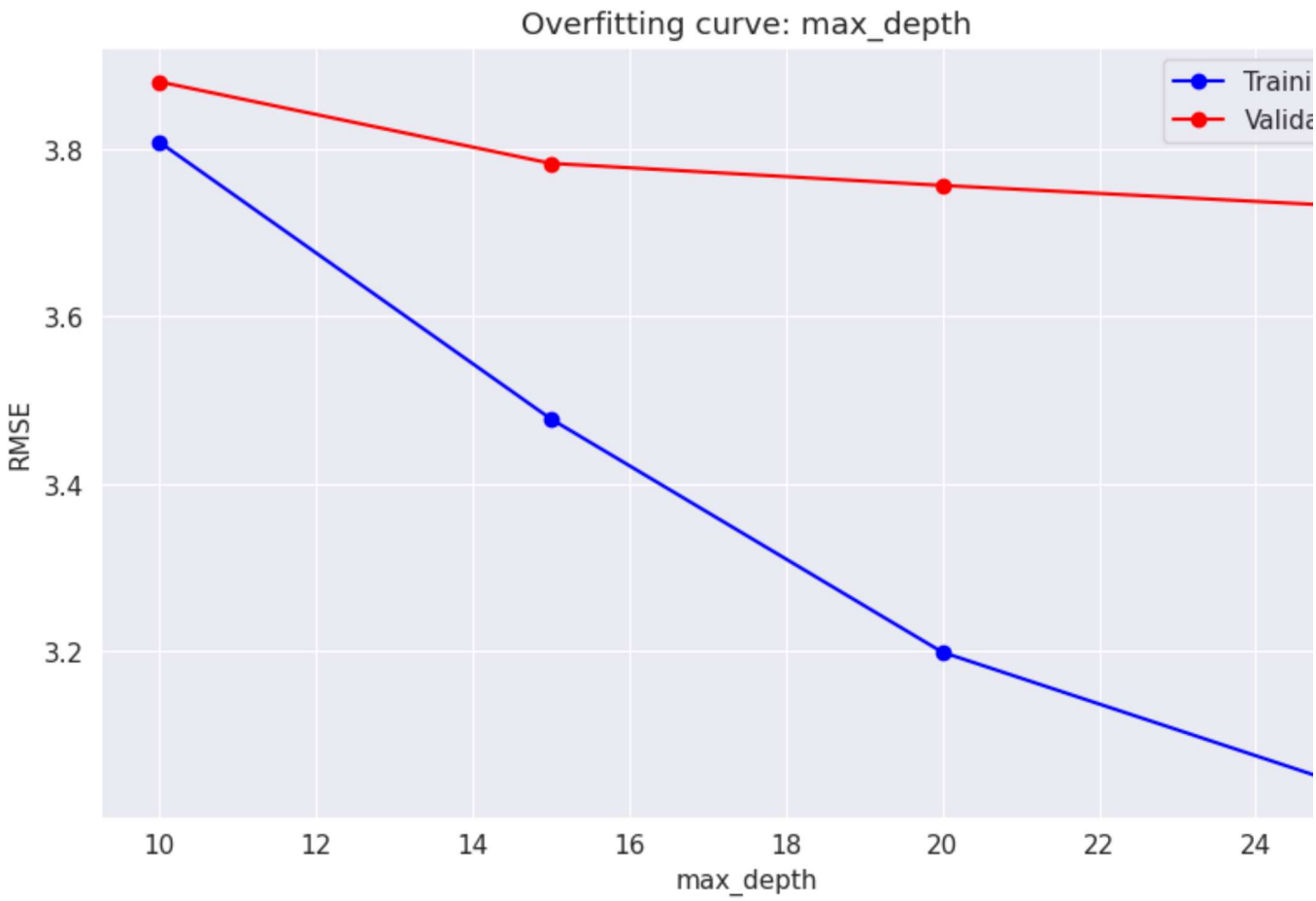
Lets define functions to experiment with hyperparameters of RandomForestRegressor:

```
def test_params(**params):
    model = RandomForestRegressor(random_state=42, n_jobs=-1, **params).fit(train_x, train_y)
    train_rmse = mean_squared_error(model.predict(train_x), train_y, squared=False)
    val_rmse = mean_squared_error(model.predict(val_x), val_y, squared=False)
    return train_rmse, val_rmse


def test_param_and_plot(param_name, param_values):
    train_errors, val_errors = [], []
    for value in param_values:
        params = {param_name: value}
        train_rmse, val_rmse = test_params(**params)
        train_errors.append(train_rmse)
        val_errors.append(val_rmse)
    plt.figure(figsize=(10,6))
    plt.title('Overfitting curve: ' + param_name)
    plt.plot(param_values, train_errors, 'b-o')
    plt.plot(param_values, val_errors, 'r-o')
    plt.xlabel(param_name)
    plt.ylabel('RMSE')
    plt.legend(['Training', 'Validation'])


test_param_and_plot('n_estimators', [50, 100, 150, 200])
```



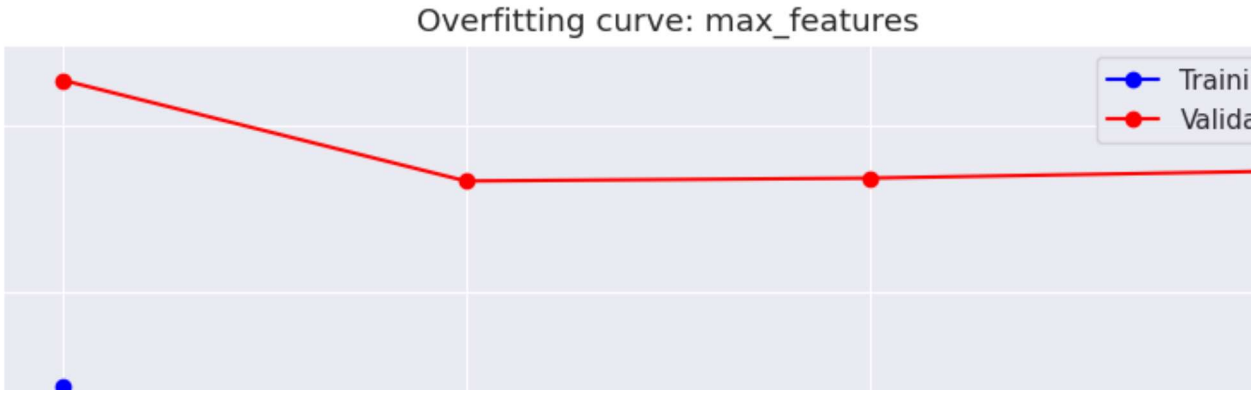
The validation error for number of decision tress (n_estimators) seems to be increasing beyond 50

```
test_param_and_plot('max_depth', [10, 15, 20, 25])
```



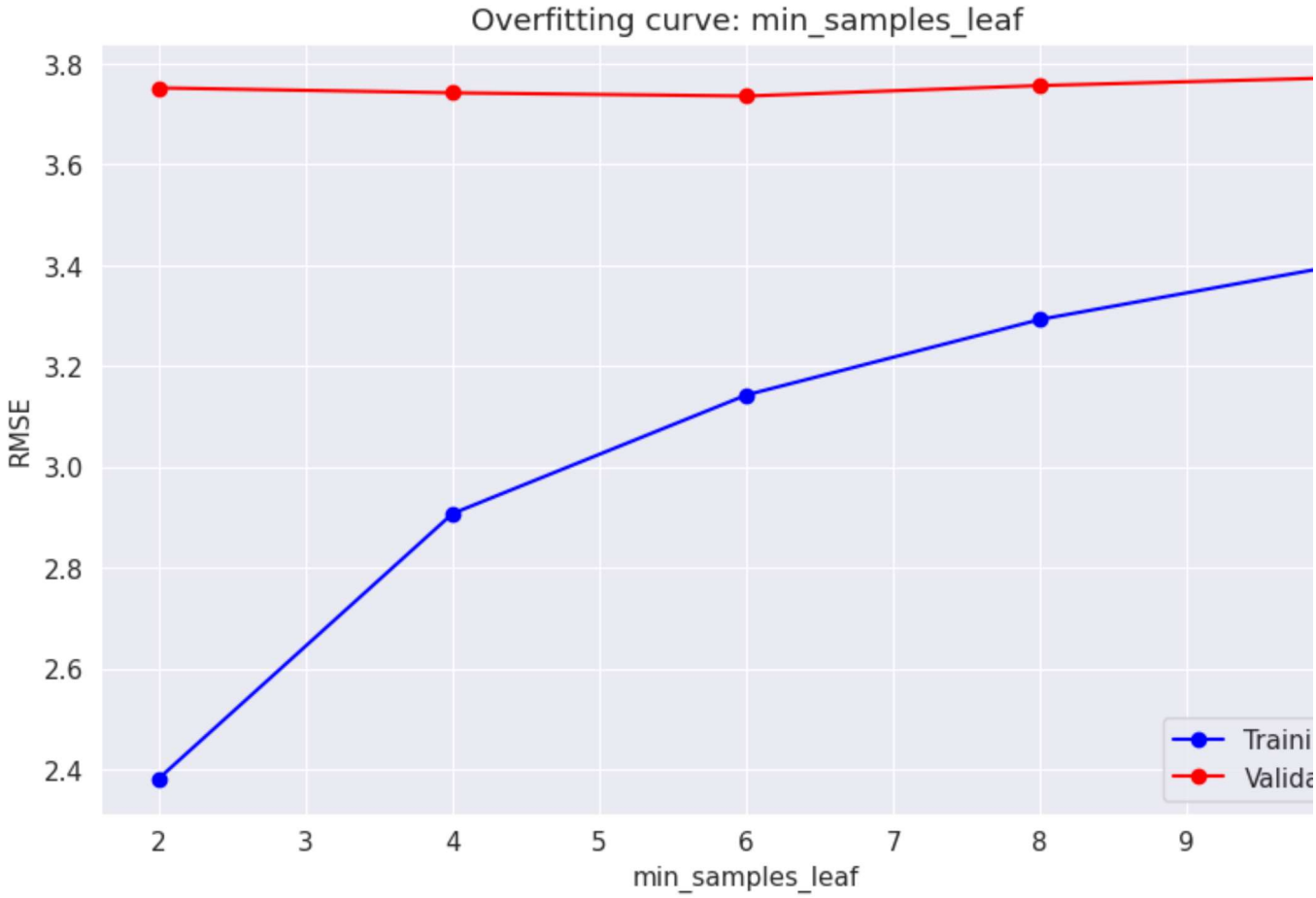
The validation error has a decresing trend with increase in max_depth of the decision trees in random forest

```
test_param_and_plot('max_features', ['sqrt', 0.5, 0.6, 0.7])
```



Half the total number of features (0.5) seems to be the optimal value for max_features to be considered for each split in the decision trees

```
test_param_and_plot('min_samples_leaf', [2, 4, 6, 8, 10])
```



The minimum number of samples required to be at a leaf node (min_samples_leaf) doesn't seem to impact the validation error much.
(4-6) seems to be the optimal value

After experimentation, we have arrived at a set of following hyperparameters which minimise the validation error:

```
test_params(n_estimators = 50, max_depth=30, min_samples_leaf=6, max_features=0.5)
```

```
(3.1102845237556385, 3.7436076902859647)
```

```
rf2_model = RandomForestRegressor(random_state=42, n_jobs=-1, n_estimators = 50, max_depth=30, min_samples_le
rf2_model.fit(train_x, train_y)
```

```
RandomForestRegressor(max_depth=30, max_features=0.5, min_samples_leaf=6,
                      n_estimators=50, n_jobs=-1, random_state=42)
```

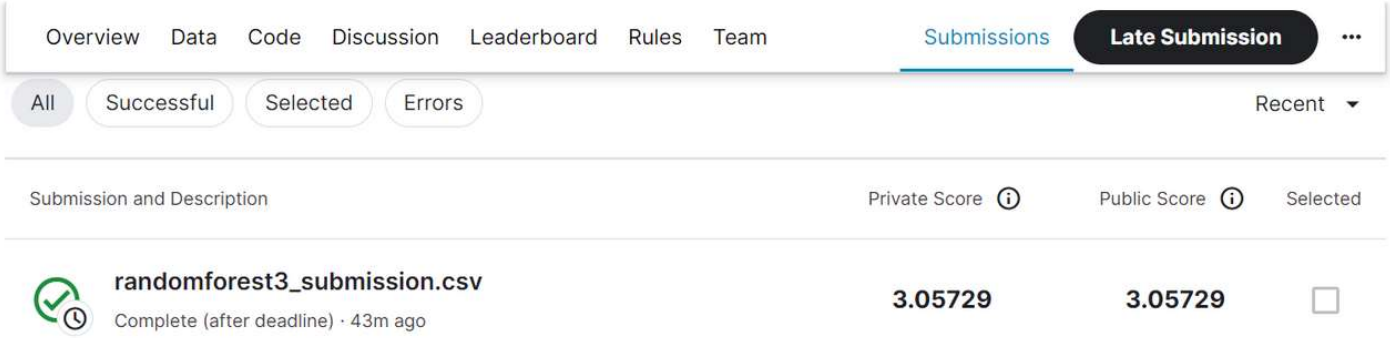
```
submission(rf2_model, 'randomforest2_submission.csv')
```

	key	fare_amount
0	2015-01-27 13:08:24.0000002	10.967622
1	2015-01-27 13:08:24.0000003	10.341950
2	2011-10-08 11:53:44.0000002	4.383888
3	2012-12-01 21:12:12.0000002	7.338774
4	2012-12-01 21:12:12.0000003	14.140467
...	...	...
9909	2015-05-10 12:37:51.0000002	8.382881
9910	2015-01-12 17:05:51.0000001	12.409642
9911	2015-04-19 20:44:15.0000001	52.795055
9912	2015-01-31 01:05:19.0000005	19.938247
9913	2015-01-18 14:06:23.0000006	7.816824

9914 rows × 2 columns

The optimised rf2_model yielded a test score of 3.21, placing us at 447th position on the Kaggle leaderboard out of 1480 submissions (top 30%)

This is the result from training just 1% of the training data.
We also tried to train the same randomforest model on **20% of the training data**.
It resulted in a **test score of 3.05** placing us at **310th rank (top 20%)** of the leaderboard.
Below is a screenshot of submissions with improvment in the test scores successively:



Saving the Model

```
import joblib
```

3.37964 3.37964

```
nyc_taxi_fare_rf = {
    'model': rf2_model,
    'scaler': scaler,
    'encoder': encoder,
    'input_cols': input_cols,
    'target_col': target_col,
    'numeric_cols': numeric_cols,
    'categorical_cols': categorical_cols,
    'encoded_cols': encoded_cols
}
```

```
joblib.dump(nyc_taxi_fare_rf, 'nyc_taxi_fare_rf2.joblib')
```

```
['nyc_taxi_fare_rf2.joblib']
```

References

- How to approach ML problems by Aakash NS: https://jovian.com/aakashns/how-to-approach-ml-problems
- Build a Machine Learning Project from Scratch by Aakash NS: https://youtu.be/Qr9iONLD3Lk
- Kaggle competition notebook: https://www.kaggle.com/code/madhurisivalenka/cleansing-eda-modelling-lgbm-xgboost-starters
- Jovian notebook: https://jovian.com/pritesh/ny-taxi-fare-predictions-1001
- Kaggle competition notebook: https://www.kaggle.com/code/breemen/nyc-taxi-fare-data-exploration
- Kaggle competition notebook: https://www.kaggle.com/code/drgilermo/dynamics-of-new-york-city-animation/notebook
- Haversine_formula to calculate distance between two points on a sphere: https://community.esri.com/t5/coordinate-reference-systems-blog/distance-on-a-sphere-the-haversine-formula/ba-p/902128
- Linear Regression by Aakash NS: https://jovian.com/aakashns/python-sklearn-linear-regression
- Random Forests by Aakash NS: https://jovian.com/aakashns/sklearn-decision-trees-random-forests