Cab fare Prediction

Abhay Kumar Singh

Contents

Chapter 1

Introduction

1.1. Problem Statement

You are a cab rental company. You have successfully run the pilot project and now want to launch your cab service across the country. You have collected the historical data from your pilot project and now have a requirement to apply analytics for fare prediction. You need to design a system that predicts the fare amount for a cab ride in the city.

1.2. Data

Whenever we start a machine learning project, we will have data from different sources compiled into a single source. Having done that, we now need a better understanding of all the features to predict a target variable.

Value	**Description**
fare_amount	Fare Amount
pickup_datetime	Cab pickup date with time
pickup_longitude	Coordinate of pickup location longitude
pickup_latitude	Coordinate of pickup location latitude
dropoff_longitude	Coordinate of drop location longitude
dropoff_latitude	Coordinate of drop location latitude
passenger_count	Number of passengers in the cab

Chapter 2

Methodology

2.1. Pre-Processing

Pre-processing refers to the transformations applied to the data before feeding it to machine learning algorithm. Data Pre-processing is a technique that is used that is used to convert raw data into clean dataset. It involves Missing Value Analysis, Outliers Analysis, and Feature Engineering, Exploratory Data Analysis and Feature Selection.

2.1.1. Missing Value Analysis

Missing Value Analysis is done to ensure that there are no missing values in the dataset. There are many ways to deal with the missing values. One way is to remove the rows or records containing the missing values or drop the column depending on the count of missing values. Other way is to impute the missing values using mean, median, KNN imputation, etc.

Python code

In python, we make use of **pandas** library to work with data frames. Also we use the **os** library to set the current working directory.

```
import os
import math
import pandas as pd
```

We read the **csv** file from the current working directory.

```
In [3]: # set the current working directory
        os.chdir("C:/Users/abhay/Desktop/Data Science/Cab_Fare")
        print(os.getcwd())

        C:\Users\abhay\Desktop\Data Science\Cab_Fare
```

Following are the data types of the independent variables

```
In [5]: # checking data types
        train.dtypes

Out[5]: fare_amount           object
        pickup_datetime       object
        pickup_longitude     float64
        pickup_latitude      float64
        dropoff_longitude    float64
        dropoff_latitude     float64
        passenger_count      float64
        dtype: object
```

Before we calculate the missing values, we will convert "**fare_amount**" from object to numeric form and "**pickup_datetime**" from object to datetime format.

```
In [7]: #Convert fare_amount from object datatype to numeric datatype
        train["fare_amount"] = pd.to_numeric(train["fare_amount"],errors = "coerce")

In [8]: # Here for pickup_datetime variable , we need to change its data type to datetime
        train['pickup_datetime'] =  pd.to_datetime(train['pickup_datetime'], format='%Y-%m-%d %H:%M:%S UTC' , errors = 'coerce')

In [9]: test["pickup_datetime"] = pd.to_datetime(test["pickup_datetime"],format= "%Y-%m-%d %H:%M:%S UTC" , errors = 'coerce')

In [10]: train.dtypes
```

Now we can check the missing values as follows.

```
In [11]: # checking missing values
         train.isnull().sum()

Out[11]: fare_amount          25
         pickup_datetime       1
         pickup_longitude      0
         pickup_latitude       0
         dropoff_longitude     0
         dropoff_latitude      0
         passenger_count      55
         dtype: int64
```

We see that there are few values missing. The „**fare_amount**" variable has 25, „**pickup_datetime**" has 1 and „**passenger_count**" has 55 missing values. Imputation is clearly not a good idea in this case. Better strategy would be to drop the records containing missing values.

2.1.2. Feature Engineering

Feature Engineering, also known as feature creation, is the process of constructing new features from the existing data to train machine learning model

In the dataset, we have been provided with date/time values. We will extract day, month, year, hour from them. Also from the pickup coordinates (**pickup_latitude** and **pickup_longitude**) and drop location coordinates (**dropoff_latitude** and **dropoff_longitude**), we will be calculating the distance which will also play a crucial role in determining the output variable **fare_amount**

We will be using the **haversine** formula to calculate the great-circle

distance between the two points on a sphere given their latitudes and

longitudes $a = \sin^2(\Delta\varphi/2) + \cos\varphi_1 \cdot \cos\varphi_2 \cdot \sin^2(\Delta\lambda/2)$

$c = 2 \cdot \operatorname{atan2}(\sqrt{a}, \sqrt{(1-a)})$

$d = R \cdot c$

φ is latitude, λ is longitude, R is earth's radius (mean radius = 6,371km);

note that angles need to be in radians to pass to trig functions!

We extract year, Month, Day, and Hour from the pickup_datetime

```python
In [14]: train['year'] = train['pickup_datetime'].dt.year

         train['Month'] = train['pickup_datetime'].dt.month
         train['Day'] = train['pickup_datetime'].dt.dayofweek
         train['Hour'] = train['pickup_datetime'].dt.hour
         train = train.drop(columns = ["pickup_datetime"])
```

```python
In [15]: # Doing the same thing for test dataset
         test['year'] = test['pickup_datetime'].dt.year

         test['Month'] = test['pickup_datetime'].dt.month
         test['Day'] = test['pickup_datetime'].dt.dayofweek
         test['Hour'] = test['pickup_datetime'].dt.hour
         test = test.drop(columns = ["pickup_datetime"])
```

We label encode the year values.

```python
In [16]: # unique values of year
         train['year'].unique()

Out[16]: array([2009, 2010, 2011, 2012, 2013, 2014, 2015], dtype=int64)
```

```python
In [17]: # label encoding the year values for train as well as test data
         train['year'] = train['year'].astype('category')
         train['year'] = train["year"].cat.codes

         test['year'] = test['year'].astype('category')
         test['year'] = test["year"].cat.codes
```

We create a function that gives **haversine** distance as output.

```python
In [19]: def distance(pickup_latitude, pickup_longitude, dropoff_latitude, dropoff_longitude):
             data = [train, test]
             for i in data:
                 R = 6371  #radius of earth in kilometers

                 phi1 = np.radians(i[pickup_latitude])
                 phi2 = np.radians(i[dropoff_latitude])

                 delta_phi = np.radians(i[dropoff_latitude]-i[pickup_latitude])
                 delta_lambda = np.radians(i[dropoff_longitude]-i[pickup_longitude])

                 #a = sin²((φB - φA)/2) + cos φA . cos φB . sin²((λB - λA)/2)
                 a = np.sin(delta_phi / 2.0) ** 2 + np.cos(phi1) * np.cos(phi2) * np.sin(delta_lambda / 2.0) ** 2

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

                 #d = R*c
                 #distance in kilometers
                 d = (R * c)
                 i['Distance'] = d
```

We call this function to see a new column getting added to train and test datasets

```python
In [20]: distance('pickup_latitude', 'pickup_longitude', 'dropoff_latitude', 'dropoff_longitude')
```

Following is the updated training dataset

```
In [21]: train.head()
```

Out[21]:

	fare_amount	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count	year	Month	Day	Hour	Distance
0	4.5	-73.844311	40.721319	-73.841610	40.712278	1.0	0	6	0	17	1.030764
1	16.9	-74.016048	40.711303	-73.979268	40.782004	1.0	1	1	1	16	8.450134
2	5.7	-73.982738	40.761270	-73.991242	40.750562	2.0	2	8	3	0	1.389525
3	7.7	-73.987130	40.733143	-73.991567	40.758092	1.0	3	4	5	4	2.799270
4	5.3	-73.968095	40.768008	-73.956655	40.783762	1.0	1	3	1	7	1.999157

2.1.3. Outlier Analysis

Outliers are extreme values that deviate from other observations in the data. The rows containing the outliers can either be deleted or imputed.

The coordinates of latitudes lie in the range of (-90, 90), while the coordinates of the longitude lies in the range (-180, 180). We ensure this by using following code.

```
In [22]: train = train.drop(train.loc[(train["pickup_latitude"] < -90) | (train["pickup_latitude"] > 90)].index , axis = 0)
         train = train.drop(train.loc[(train["pickup_longitude"] < -180) | (train["pickup_longitude"] > 180)].index , axis = 0)

         train = train.drop(train.loc[(train["dropoff_latitude"] < -90) | (train["dropoff_latitude"] > 90)].index , axis = 0)
         train = train.drop(train.loc[(train["dropoff_longitude"] < -180) | (train["dropoff_longitude"] > 180)].index , axis = 0)
```

Next, we check for distances which are very large. We take the threshold value based on our observation

```
In [24]: train.sort_values(by = "Distance")
```

Out[24]:

	fare_amount	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count	year	Month	Day	Hour	Distance
4086	45.00	-73.776293	40.645693	-73.776293	40.645693	2.0	0	6	0	12	0.000000
11437	5.30	-73.919393	40.731357	-73.919393	40.731357	2.0	1	10	2	14	0.000000
13264	9.00	0.000000	0.000000	0.000000	0.000000	6.0	4	2	3	22	0.000000
799	3.00	0.000000	0.000000	0.000000	0.000000	5.0	4	12	6	23	0.000000
10959	5.30	-73.974772	40.759440	-73.974772	40.759440	1.0	1	11	2	13	0.000000
10964	4.10	0.000000	0.000000	0.000000	0.000000	1.0	0	9	3	9	0.000000
6744	14.90	0.000000	0.000000	0.000000	0.000000	1.0	0	7	1	18	0.000000
808	6.10	-73.984433	40.771084	-73.984433	40.771084	1.0	1	10	4	2	0.000000
9863	6.50	0.000000	0.000000	0.000000	0.000000	1.0	1	12	3	6	0.000000
7458	3.70	0.000000	0.000000	0.000000	0.000000	1.0	1	9	1	8	0.000000
2143	6.50	0.000000	0.000000	0.000000	0.000000	1.0	0	1	4	8	0.000000

Clearly, there is a drastic increase in distance from 129.95 to 4447.08. The distances above 130 seem unlikely and are likely the resultant of missing values of the coordinates. We will remove these large distances.

```
In [25]: # distances above 130 km look unlikely.
         train = train.drop(train.loc[(train["Distance"] > 130)].index , axis = 0)
```

Next, we observe that there are many inconsistencies in the passenger_count values.

```
In [29]: train["passenger_count"].unique()

Out[29]: array([1.000e+00, 2.000e+00, 3.000e+00, 6.000e+00, 5.000e+00, 4.000e+00,
                2.360e+02, 4.560e+02, 5.334e+03, 0.000e+00, 5.350e+02, 3.540e+02,
                5.540e+02, 5.300e+01, 3.500e+01, 3.450e+02, 5.345e+03, 5.360e+02,
                4.300e+01, 5.800e+01, 5.370e+02, 8.700e+01, 5.570e+02])
```

There are many decimal values in the passenger_count variable which we need to remove.

```
In [28]: train.drop(train[((train["passenger_count"] * 10) % 10) != 0].index , inplace = True)
```

We know that a cab cannot occupy more than 6 people in the cab. There are many values in the passenger_count variable which are greater than 6. Such values shouldn‟t exist in the training data.

```
In [30]: # maximum 6 persons can be there in the cab
         train.drop(train[(train["passenger_count"] > 6)].index , inplace = True)
```

Next, we check for outliers in the fare_amount variable.

```
In [32]: train.sort_values(by = "fare_amount")
```

Out[32]:

	fare_amount	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count	year	Month	Day	Hour	Distance
13032	-3.00	-73.995062	40.740755	-73.995885	40.741357	4.0	4	8	4	8	0.096377
2039	-2.90	-73.789450	40.643498	-73.788665	40.641952	1.0	1	3	1	23	0.184225
2486	-2.50	-74.000031	40.720631	-73.999809	40.720539	1.0	6	3	6	5	0.021244
10002	0.00	-73.987115	40.738808	-74.005911	40.713960	1.0	1	2	0	14	3.184763
2780	0.01	-73.939041	40.713963	-73.941673	40.713997	1.0	6	5	4	15	0.221878
1427	1.14	-73.862829	40.769014	-73.982075	40.723854	1.0	5	5	5	15	11.230687
6226	2.50	-73.982657	40.731395	-73.982282	40.731852	1.0	0	11	2	23	0.059839
14530	2.50	-74.010751	40.702421	-74.010743	40.702385	1.0	5	6	5	1	0.004059
6007	2.50	0.000000	0.000000	0.000000	0.000000	1.0	3	11	2	12	0.000000
503	2.50	-73.998720	40.624708	-73.998720	40.624708	1.0	1	1	1	1	0.000000
6297	2.50	0.000000	0.000000	0.000000	0.000000	1.0	1	11	0	9	0.000000
6002	2.50	-73.937357	40.758250	-73.937397	40.758217	1.0	4	6	4	10	0.004982
922	2.50	-73.959008	40.712517	-73.959132	40.712184	1.0	3	8	1	23	0.038475
9621	2.50	0.000000	0.000000	0.000000	0.000000	1.0	6	3	6	22	0.000000

By checking the coordinates of locations, we can understand that the locations are in **New York** and its neighbourhood. We have made some assumptions accordingly. The fare_amount given to us is in dollars. Also the base fare in **New York** is $2.5

Clearly, there is a drastic change in fare_amount from $453 to $54343 which is not possible. Hence we delete the observations which have fare_amount greater than $453

```
In [33]: train = train.drop(train.loc[(train["fare_amount"] < 2.5) | (train["fare_amount"] > 453) ].index , axis = 0)
```

Next, we analyse distances which have been calculated as 0 but the fare_amount is positive. There may be a couple of reasons for the same. One reason is that the pickup location and the drop location may be same. The other reason could be that pickup or the drop location coordinates may be missing or may not have been entered by the passenger.

```
In [34]: Counter(((train["Distance"] == 0) |
                (train["pickup_latitude"] == 0)|
                (train["pickup_longitude"] == 0)|
                (train["dropoff_latitude"] == 0)|
                (train["dropoff_longitude"] == 0))
                & train["fare_amount"] !=0)

Out[34]: Counter({False: 15476, True: 455})
```

There are 455 such values in total. We cannot delete these observations. We use a strategy to impute the distance in such cases using the formula below.

Distance = (fare_amount – 2.5)/1.56

Here 2.5 is the base fare in dollars for a cab in New York. 1.56 is the amount in dollars for each extra kilometre travelled. Following code demonstrates the imputation using the above formula\

```
In [36]: subset = train.loc[((train["Distance"] == 0) |
                (train["pickup_latitude"] == 0)|
                (train["pickup_longitude"] == 0)|
                (train["dropoff_latitude"] == 0)|
                (train["dropoff_longitude"] == 0))
                & train["fare_amount"] !=0]

In [37]: subset["Distance"] = subset.apply(lambda x: (x["fare_amount"] -2.5)/1.56 , axis =1)

In [38]: # updating the values in the training set
         train.update(subset)
```

2.1.4. Exploratory Data Analysis and Feature Selection

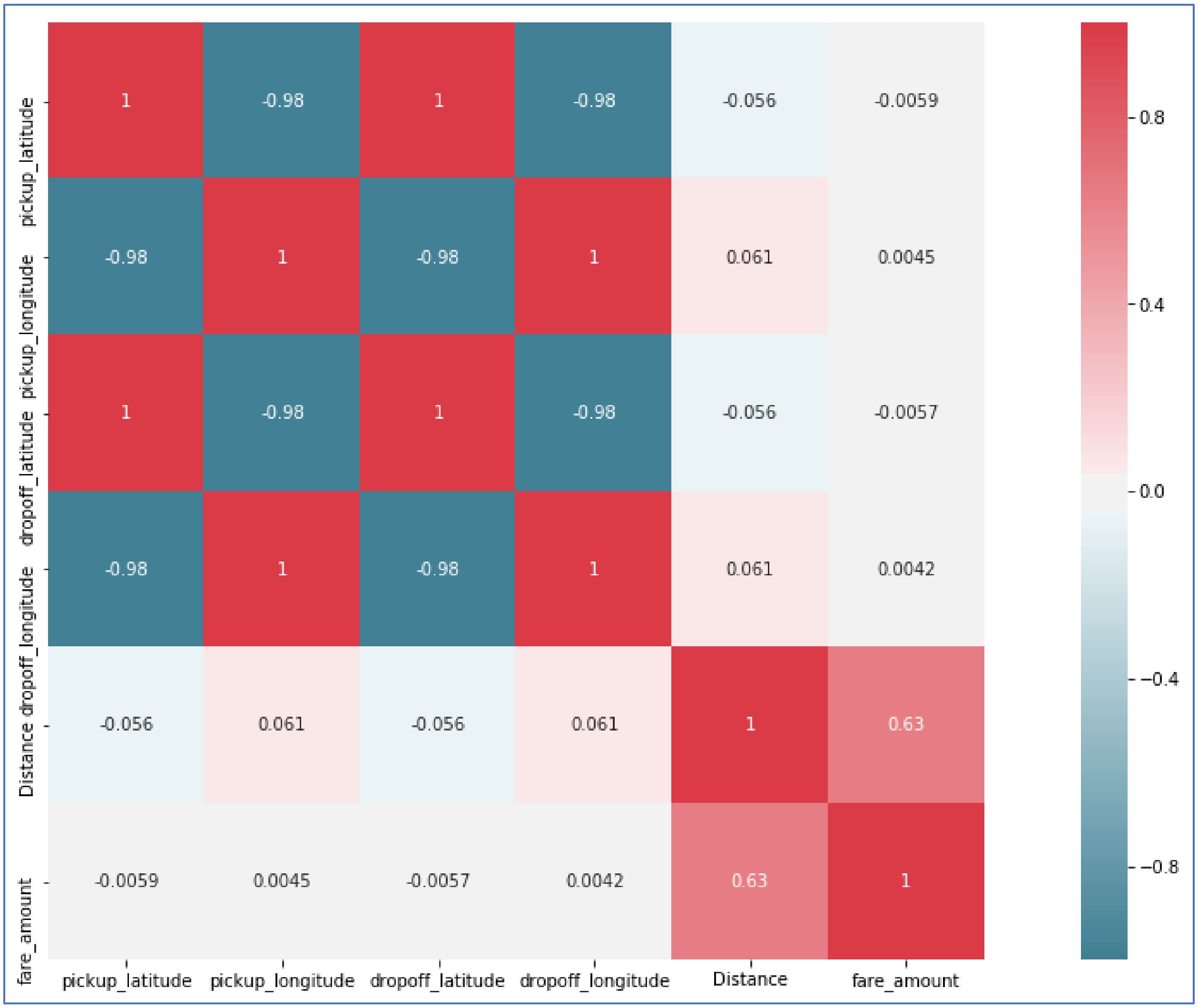
Exploratory Data Analysis is an approach to analyse datasets to summarize the main characteristics, often with visual methods. EDA is seeing what the data can tell us beyond the formal modelling or hypothesis testing task. Based on findings of EDA, we select the relevant features

EDA on Continuous variables

We visualize the heatmap of correlation matrix between continuous variables. Here we make observations if there is any kind of multicollinearity among the features.

We plot the heatmap of correlation matrix for continuous variables using the **matplotlib** and **seaborn** libraries.

```
In [39]: numerical_variables = ["pickup_latitude",
                               "pickup_longitude",
                               "dropoff_latitude",
                               "dropoff_longitude",
                               "Distance",
                               "fare_amount"]

In [40]: # Correlation analysis
         # correlation plot
         df_corr = train.loc[:,numerical_variables]

In [41]: %matplotlib inline
         # Set the width and height of the plot
         f , ax = plt.subplots(figsize = (20,10))

         # Generate correlation matrix
         corr = df_corr.corr()

         # Plot using seaborn library
         sn.heatmap(corr,mask = np.zeros_like(corr,dtype = np.bool), cmap = sn.diverging_palette(220,10,as_cmap = True),
                    square = True , ax = ax , vmin = -1 , vmax = 1 , annot = True)

Out[41]: <matplotlib.axes._subplots.AxesSubplot at 0x208f2e247b8>
```

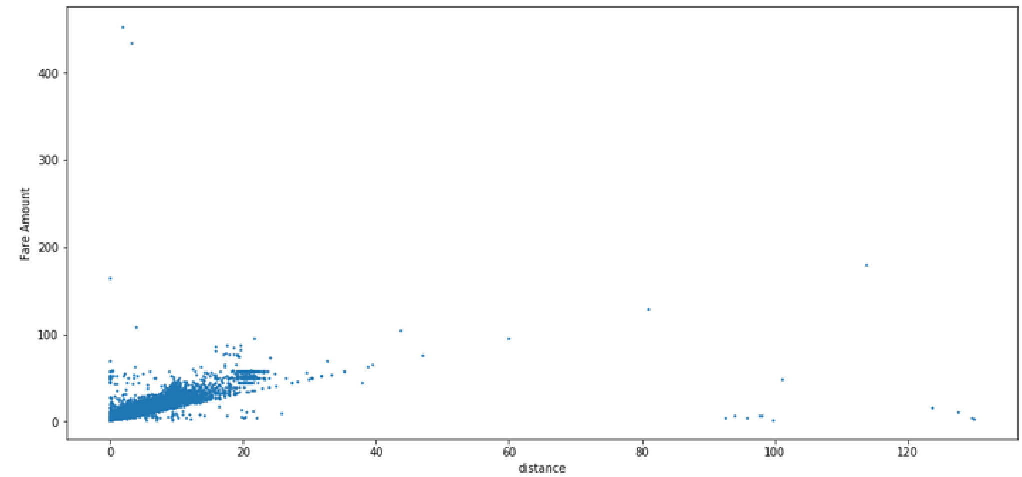


We observe that all the coordinates of latitudes and longitudes are highly correlated to each other. This will negatively impact the model. We will have to drop the columns which represent these coordinates.

```
In [42]: # we drop all latitudes and longitudes because they are highly correlated with each other
         # they will negatively affect the model
         # moreover we have calculated the haversine distance and we don't need these variables

         train = train.drop(columns = ["pickup_longitude","pickup_latitude","dropoff_longitude", "dropoff_latitude"])
         test = test.drop(columns = ["pickup_longitude","pickup_latitude","dropoff_longitude", "dropoff_latitude"])
```

We use the following code to plot a scatter plot of Distance vs fare_amount

```
In [44]: plt.figure(figsize=(15,7))
         plt.scatter(x=train['Distance'], y=train['fare_amount'], s=2)
         plt.xlabel('distance')
         plt.ylabel('Fare Amount');
```

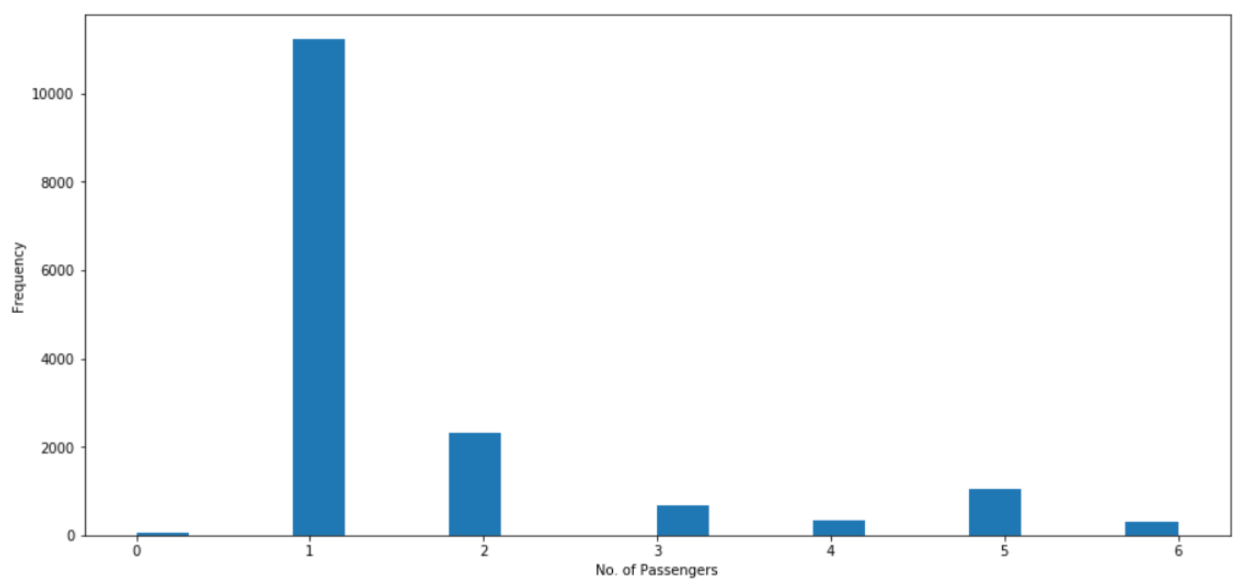


Python code for EDA on Categorical variables

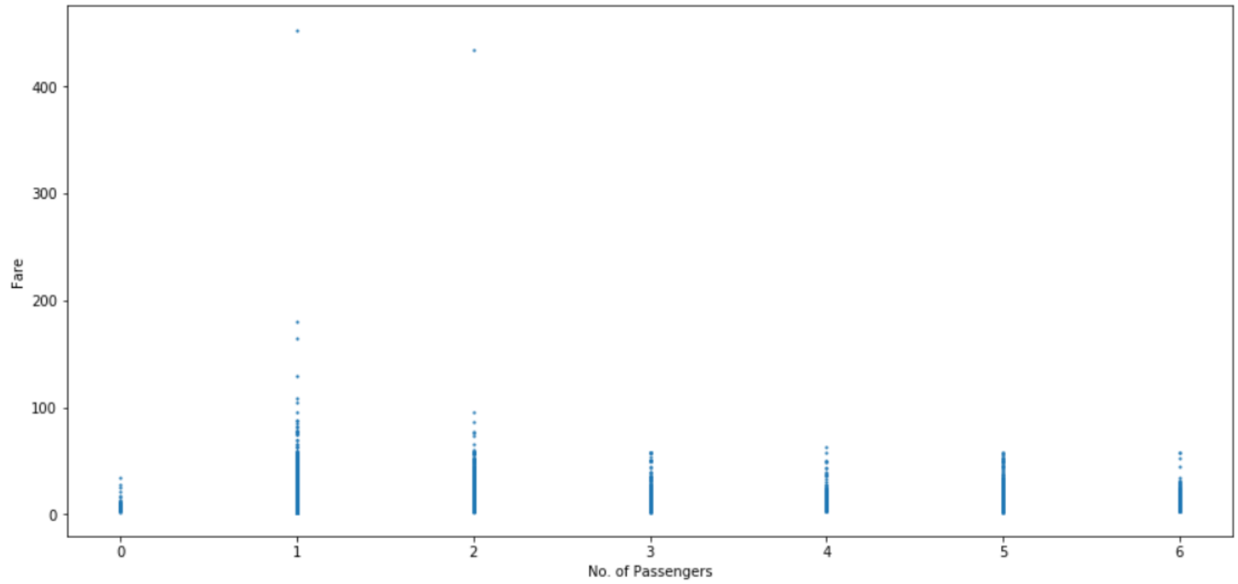
Let us check how the numbers of passengers affect the fare amount and the frequency of cabs

From the

```
%matplotlib inline
plt.figure(figsize=(15,7))
plt.hist(train['passenger_count'], bins=20)
plt.xlabel('No. of Passengers')
plt.ylabel('Frequency')
plt.xticks(range(0, 7));
```



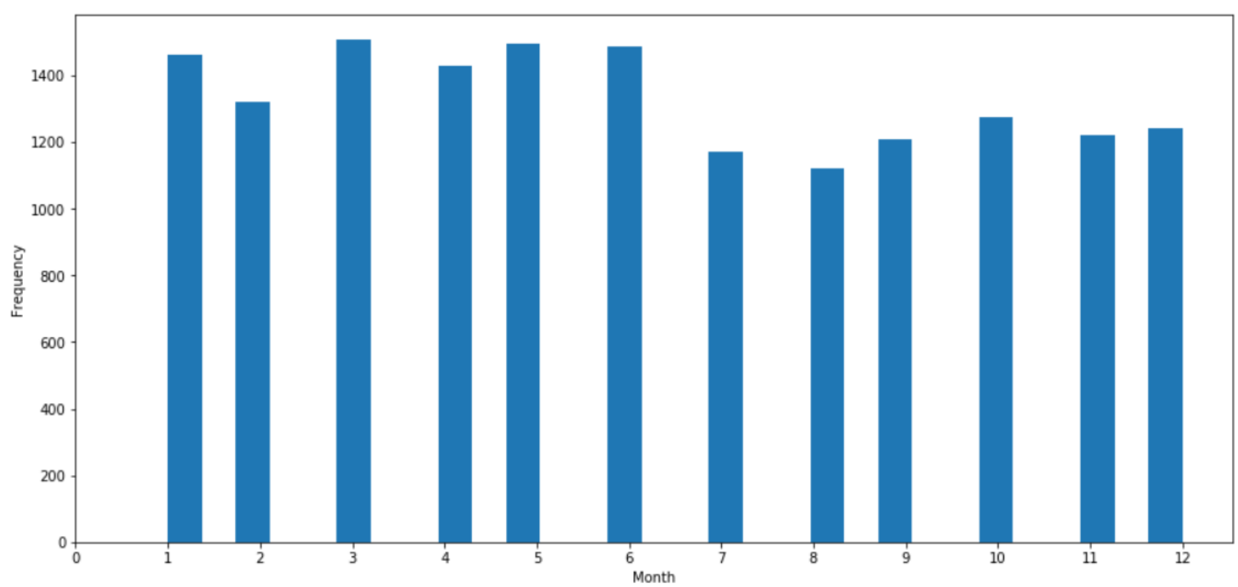
```
plt.figure(figsize=(15,7))
plt.scatter(x=train['passenger_count'], y=train['fare_amount'], s=1.5)
plt.xlabel('No. of Passengers')
plt.ylabel('Fare');
```



From the above two graphs we observe that the single passengers are the frequent travellers. The highest cab fare is paid by them only

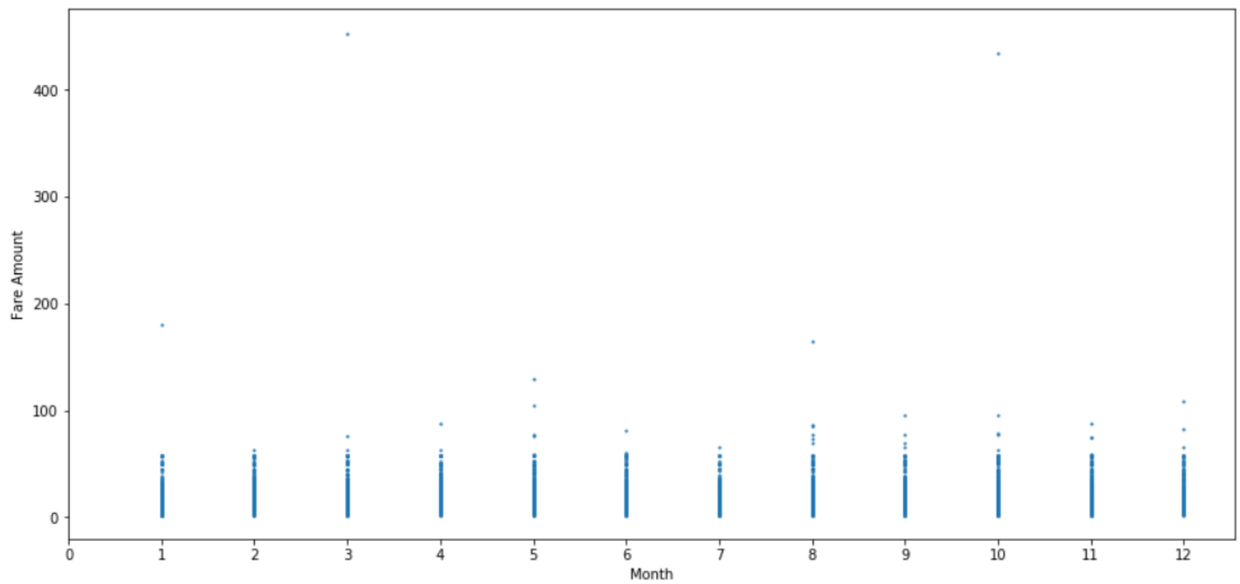
Now let us check how the month affects the fare amount and the frequency of cabs

```
plt.figure(figsize=(15,7))
plt.hist(train['Month'], bins=30)
plt.xlabel('Month')
plt.ylabel('Frequency')
plt.xticks(range(0, 13));
```



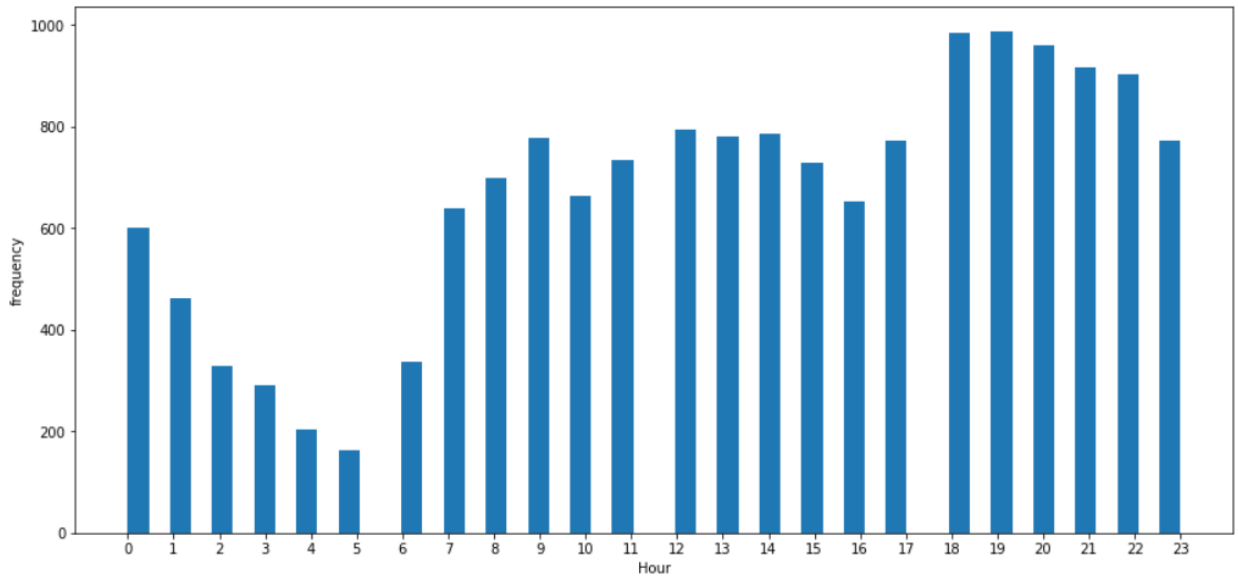
From the two plots above we see that, the frequency of cabs is highest in the month of March and the highest cab fare was also noted in the same month

```
plt.figure(figsize=(15,7))
plt.scatter(x=train['Month'], y=train['fare_amount'], s=1.5)
plt.xlabel('Month')
plt.ylabel('Fare Amount')
plt.xticks(range(0, 13));
```

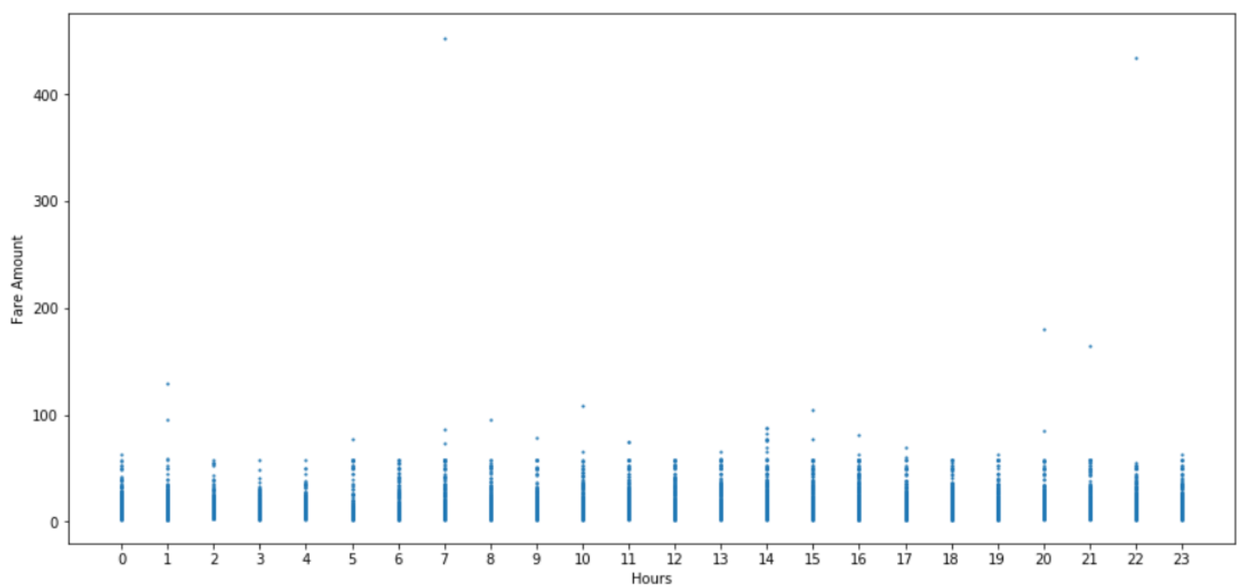


Let us check how each hour affects the frequency of cabs and the fare amount
From the

```python
plt.figure(figsize=(15,7))
plt.hist(train['Hour'] , bins= 50)
plt.xlabel('Hour')
plt.ylabel('frequency')
plt.xticks(range(0,24 ));
```




```python
plt.figure(figsize=(15,7))
plt.scatter(x=train['Hour'], y=train['fare_amount'], s=1.5)
plt.xlabel('Hours')
plt.ylabel('Fare Amount')
plt.xticks(range(0, 24));
```



From the two graphs above we see that, frequency of cabs is lowest at 5 AM in the morning and highest at 7 PM in the evening. The highest cab fare was noted at 7 AM in the morning

2.1.5. Handling skewness of Data

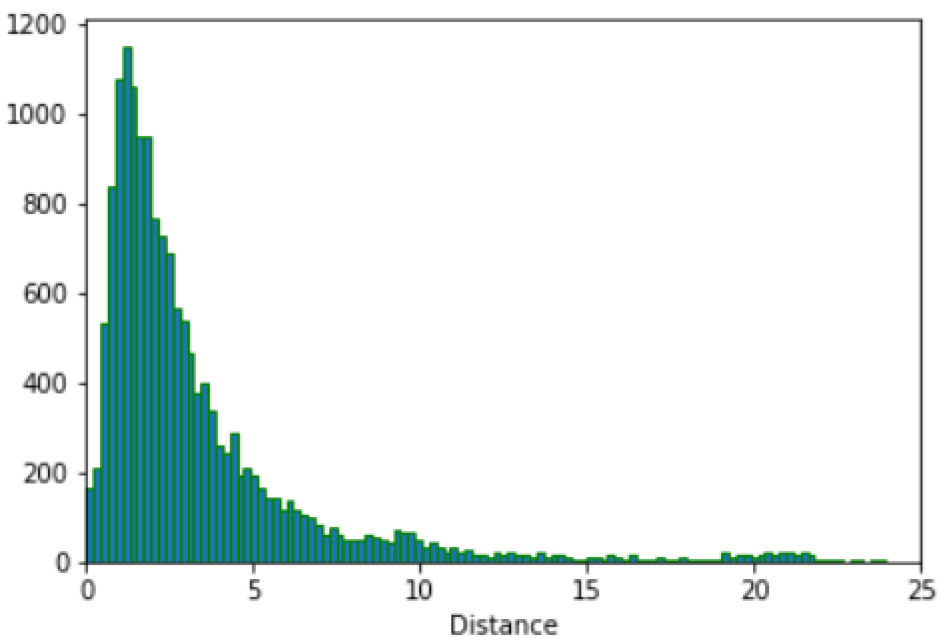
A data transformation may be used to reduce skewness of data. A distribution that is symmetric or nearly so is easier to handle and interpret than a skewed distribution. More specifically, a normal or Gaussian distribution is often regarded as ideal as it is assumed by statistical methods.

Common transformations to the data include square root, log, etc.

We have used square root transformation as some of the values for continuous variables are zero

Following is the histogram plot of continuous variable Distance

```
plt.hist(train['Distance'],bins = 'auto' ,ec='green')
plt.xlim(0,25)
plt.xlabel('Distance')
plt.show()
```

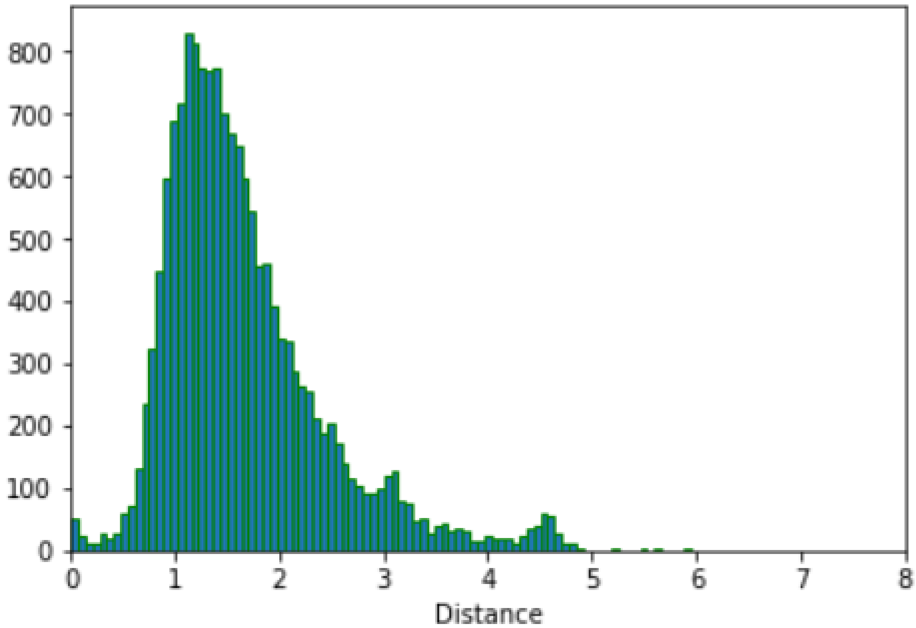


We observe that it is positively skewed. We will apply square root transformation on it.

```
# some of the distances are 0. we cannot take the log. We decide to proceed by taking square root
train["Distance"] = np.sqrt(train["Distance"])
test["Distance"] = np.sqrt(test["Distance"])
```

We will now check the histogram plot of it again

```
plt.hist(train['Distance'],bins = 'auto' ,ec='green')
plt.xlim(0,8)
plt.xlabel('Distance')
plt.show()
```



We see that the skewness of it has reduced to a large extent

2.2. Model Development and Evaluation

2.2.1. Model Selection

We have to predict the fare amount for the cab ride. Our target variable „**fare_amount**" is a continuous variable. Clearly, this is a regression problem. We will choose **RMSE** (Root mean square error) as the final evaluation metric as large errors are undesirable in this case.

We have used the following Regression Algorithms

1. Linear Regression
2. Decision Tree Regression
3. Random Forest Regression
4. Gradient Boost Regression

Before applying the model, we have to divide the dataset into training and validation dataset. We are using 80% data for training and 20% of data for validation purpose.

```python
from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_test=train_test_split(train.drop('fare_amount',axis=1),
                                               train['fare_amount'],
                                               test_size=0.2,
                                               random_state=3)
```

We have created a custom function to calculate Mean Absolute Percentage Error (**MAPE**)

```python
# Calculate MAPE
def MAPE(y_true, y_pred):
    mape = np.mean(np.abs((y_true-y_pred)/y_true))
    return mape
```

```python
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
```

2.2.2. Linear Regression

Python code

To apply Linear Regression, we need to import **LinearRegression** from sklearn

```python
from sklearn.linear_model import LinearRegression
```

Then we build the model and predict fare amount for the validation set

```python
linearRegressionModel = LinearRegression()
linearRegressionModel.fit(x_train,y_train)
linearRegressionModel_predictions = linearRegressionModel.predict(x_test)
```

Following are the model evaluation metrics we obtain

```python
print("MAE for Linear Regression is ",mean_absolute_error(y_test,linearRegressionModel_predictions))
print("MAPE for Linear Regression is ",MAPE(y_test,linearRegressionModel_predictions))
print("MSE for Linear Regression is ",mean_squared_error(y_test,linearRegressionModel_predictions))
print("RMSE for Linear Regression is ",np.sqrt(mean_squared_error(y_test,linearRegressionModel_predictions)))
```

```
MAE for Linear Regression is  2.794141787382295
MAPE for Linear Regression is  0.2769475988349314
MSE for Linear Regression is  24.46193450755993
RMSE for Linear Regression is  4.945900778175795
```

2.2.3. Decision Tree Regression

Python code

To apply Decision Tree Regression, we need to import **DecisionTreeRegression** from **sklearn**

```
from sklearn.tree import DecisionTreeRegressor
```

Then we build the model and predict fare amount for the validation set

```
dtree_model = DecisionTreeRegressor(random_state=42)
dtree_model.fit(x_train,y_train)
dtree_predictions = dtree_model.predict(x_test)
```

Following are the evaluation metrics we obtain

```
print("MAE for decision tree regressor is ",mean_absolute_error(y_test,dtree_predictions))
print("MAPE for decision tree regressor is ",MAPE(y_test,dtree_predictions))
print("MSE for decision tree regressor is ",mean_squared_error(y_test,dtree_predictions))
print("RMSE for decision tree regressor is ",np.sqrt(mean_squared_error(y_test,dtree_predictions)))
```

```
MAE for decision tree regressor is  2.9344116724192033
MAPE for decision tree regressor is  0.2809281968685263
MSE for decision tree regressor is  29.387060338876687
RMSE for decision tree regressor is  5.420983336893473
```

Further, we apply **hyperparameter tuning** to improve our metrics. We have considered different combinations of **min_samples_leaf** and **max_features**. Here **min_samples_leaf** are the minimum number of samples required to be at the leaf node and **max_features** are the number of features to be considered when looking at the best split. We are performing a 3-fold cross validation in this case. Using the optimal parameters we make the predictions. We use **GridSearchCV()** from sklearn which does an exhaustive search over specified parameter values for an estimator. Its important features are **fit** and **predict**

```
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV
```

```
kf = KFold(n_splits= 3 , random_state=42)
params_dict = {
               'min_samples_leaf': list(range(1,20,2)),
               'max_features': list(range(1,7))
               }

dtree_tune=GridSearchCV(estimator=DecisionTreeRegressor(random_state=42),
                        param_grid=params_dict ,
                        cv = kf,
                        scoring='neg_mean_squared_error',
                        verbose=6000)
dtree_tune.fit(x_train,y_train)
```

Following is the best estimator found through grid search

```
dtree_tune.best_estimator_
```

```
DecisionTreeRegressor(criterion='mse', max_depth=None, max_features=6,
                      max_leaf_nodes=None, min_impurity_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=19,
                      min_samples_split=2, min_weight_fraction_leaf=0.0,
                      presort=False, random_state=42, splitter='best')
```

Next, we make the predictions which we obtain by applying the model on the validation set

```
dtree_tuned_predictions=dtree_tune.predict(x_test)

print("MAE after decision tree after hyperparameter tuning is ",mean_absolute_error(y_test,dtree_tuned_predictions))
print("MAPE after decision tree after hyperparameter tuning is ",MAPE(y_test,dtree_tuned_predictions))
print("MSE after decision tree after hyperparameter tuning is ",mean_squared_error(y_test,dtree_tuned_predictions))
print("RMSE after decision tree after hyperparameter tuning is ",
      np.sqrt(mean_squared_error(y_test,dtree_tuned_predictions)))
```

```
MAE after decision tree after hyperparameter tuning is  2.341356860136691
MAPE after decision tree after hyperparameter tuning is  0.231902852899584
MSE after decision tree after hyperparameter tuning is  21.23701243505393
RMSE after decision tree after hyperparameter tuning is  4.608363314133764
```

We see that, there is considerable improvement is **RMSE** as well as other metrics.

2.2.3. Random Forest Regression

Python code

To apply Random Forest Regression, we need to import **RandomForestRegressor** from **sklearn**

```
from sklearn.ensemble import RandomForestRegressor
```

Then we build the model and predict the fare amount for the validation set

```
rf_model = RandomForestRegressor(random_state=42)
rf_model.fit(x_train,y_train)
rf_predictions = rf_model.predict(x_test)
```

Following are the model evaluation metrics which we obtain

```
print("MAE for random forest regressor is ",mean_absolute_error(y_test,rf_predictions))
print("MAPE for random forest regressor is ",MAPE(y_test,rf_predictions))
print("MSE for random forest regressor is ",mean_squared_error(y_test,rf_predictions))
print("RMSE for random forest regressor is ",np.sqrt(mean_squared_error(y_test,rf_predictions)))
```

```
MAE for random forest regressor is  2.3919880765610295
MAPE for random forest regressor is  0.25216728324460813
MSE for random forest regressor is  22.89173483150298
RMSE for random forest regressor is  4.784530784884029
```

Next we go for **hyperparameter optimization** to tune our model and improve the evaluation metrics. We have considered different combinations of **n_estimators** , **max_features** and **max_depth**. Here **n_estimators** are the number of trees in the forest, **max_features** are the number of features to be considered when looking for the best split, **max_depth** is the maximum depth of the tree. We are performing 3-fold cross validation and finding the optimal parameters.

```
#for random forest regresion.

kf = KFold(n_splits= 3 , random_state=42)

param_grid={'n_estimators':[100,200,300,400,500],'max_features':list(range(1,7)),'max_depth': [4,6,8]}

rf_tune=GridSearchCV(RandomForestRegressor(random_state=42),
                     param_grid=param_grid ,
                     cv=kf ,
                     verbose= 6000)

rf_tune.fit(x_train,y_train)
```

Following is the best estimator found through grid search

```
rf_tune.best_estimator_
```

```
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=8,
                      max_features=3, max_leaf_nodes=None,
                      min_impurity_decrease=0.0, min_impurity_split=None,
                      min_samples_leaf=1, min_samples_split=2,
                      min_weight_fraction_leaf=0.0, n_estimators=300,
                      n_jobs=None, oob_score=False, random_state=42, verbose=0,
                      warm_start=False)
```

Next we make the predictions and obtain evaluation metrics

```
rf_tuned_predictions=rf_tune.predict(x_test)

print("MAE after random forest hyperparameter tuning is ",mean_absolute_error(y_test,rf_tuned_predictions))
print("MAPE after random forest hyperparameter tuning is ",MAPE(y_test,rf_tuned_predictions))
print("MSE after random forest hyperparameter tuning is ",mean_squared_error(y_test,rf_tuned_predictions))
print("RMSE after random forest hyperparameter tuning is ",np.sqrt(mean_squared_error(y_test,rf_tuned_predictions)))
```

```
MAE after random forest hyperparameter tuning is  2.1795720880373306
MAPE after random forest hyperparameter tuning is  0.2311267646579907
MSE after random forest hyperparameter tuning is  17.33209256891075
RMSE after random forest hyperparameter tuning is  4.1631829852783016
```

We see that there is significant improvement in **RMSE** and other evaluation metrics.

2.2.4. Gradient Boosting Regression

Python code for Gradient Boosting Regression

To perform Gradient Boosting Regression, we have to use **GradientBoostingRegressor** from **sklearn**

```
from sklearn.ensemble import GradientBoostingRegressor
```

Then we build the model and make predictions on the validation dataset

```
gbr_model = GradientBoostingRegressor(random_state=42)
gbr_model.fit(x_train,y_train)
test_pred = gbr_model.predict(x_test)
```

Following are the model evaluation metrics which we obtain

```
print("MAE for gradient boosting regressor is ",mean_absolute_error(y_test,gbr_predictions))
print("MAPE for gradient boosting regressor is ",MAPE(y_test,gbr_predictions))
print("MSE for gradient boosting regressor is ",mean_squared_error(y_test,gbr_predictions))
print("RMSE for gradient boosting regressor is ",np.sqrt(mean_squared_error(y_test,gbr_predictions)))
```

```
MAE for gradient boosting regressor is  2.090227247178083
MAPE for gradient boosting regressor is  0.21154354785771226
MSE for gradient boosting regressor is  16.780220780515457
RMSE for gradient boosting regressor is  4.096366778074866
```

Next, we perform **hyperparameter tuning** to improve our metrics. We consider different combinations of **learning_rate**, **n_estimators**. The **learning_rate** indicates the learning rate and **n_estimators** indicates the number of boosting stages to perform. We are performing 3-fold cross validation to determine the optimal parameters.

```
# gradient boost hyper parameter tuning

kf = KFold(n_splits= 3 , random_state=42)

gb_grid_params = {'learning_rate': [0.1],
              'n_estimators' : list(range(130,240,10))
              }

gbr_tuned = GridSearchCV(GradientBoostingRegressor(random_state=42),
                        gb_grid_params,
                        cv=kf,
                        verbose = 6000)

gbr_tuned.fit(x_train,y_train)
```

We get the following best estimator

```
gbr_tuned.best_estimator_
```

```
GradientBoostingRegressor(alpha=0.9, criterion='friedman_mse', init=None,
                          learning_rate=0.1, loss='ls', max_depth=3,
                          max_features=None, max_leaf_nodes=None,
                          min_impurity_decrease=0.0, min_impurity_split=None,
                          min_samples_leaf=1, min_samples_split=2,
                          min_weight_fraction_leaf=0.0, n_estimators=130,
                          n_iter_no_change=None, presort='auto',
                          random_state=42, subsample=1.0, tol=0.0001,
                          validation_fraction=0.1, verbose=0, warm_start=False)
```

Following are the evaluation metrics which we obtain

```
gbr_tuned_predictions=gbr_tuned.predict(x_test)

print("MAE after GBR hyperparameter tuning is ",mean_absolute_error(y_test,gbr_tuned_predictions))
print("MAPE after GBR hyperparameter tuning is ",MAPE(y_test,gbr_tuned_predictions))
print("MSE after GBR hyperparameter tuning is ",mean_squared_error(y_test,gbr_tuned_predictions))
print("RMSE after GBR hyperparameter tuning is ",np.sqrt(mean_squared_error(y_test,gbr_tuned_predictions)))
```

```
MAE after GBR hyperparameter tuning is  2.077874495102081
MAPE after GBR hyperparameter tuning is  0.20961845254503678
MSE after GBR hyperparameter tuning is  16.60020708962114
RMSE after GBR hyperparameter tuning is  4.074335171487631
```

We see that there is improvement in RMSE as well as other evaluation metrics

Chapter 3

Conclusion

3.1. Final Best Model Selection

For final Model Selection, we will choose the model which gives us the lowest **Root Mean Square Error** value. After applying all the regression models, we observed that Gradient Boosting Regression performs the best and results in lowest **RMSE** values. We will use Gradient Boosting Regression in both python and R to make predictions on the test dataset.

The following table summarizes the results obtained by different regression algorithms in

Regression Model	MAE	MAPE	MSE	RMSE
Linear Regression	2.7941	0.2769	24.4619	4.9459
Decision Tree	2.3413	0.2319	21.2370	4.6083
Random Forest	2.1795	0.2311	17.3320	4.1631
Gradient Boosting	2.0778	0.2096	16.6002	4.0743

Clearly, Gradient Boosting Regression achieves the lowest **RMSE**. Hence we make the prediction for our test dataset using the same algorithm.

```
gbr_final_test_predictions = gbr_tuned.predict(test)
```

```
test["Predicted Fare Amount"] = gbr_final_test_predictions
```

```
test.loc[(test["Predicted Fare Amount"] < 2.5),'Predicted Fare Amount'] = 2.5
```

```
test.to_csv("Predictions.csv" , index = False)
```