Cab Fare Prediction

Santosh Kamat

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.

Variables	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 pandas as pd
```

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

```
# set the current working directory
os.chdir("C:/Users/Pranav/Desktop/Cab Fare Project")
```

```
#Loading the data:
train  = pd.read_csv("train_cab.csv")
test   = pd.read_csv("test.csv")
```

Following are the data types of the independent variables

```
train.dtypes
```

```
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

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

```
# 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')
```

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

Now we can check the missing values as follows

```
# checking missing values
train.isnull().sum()
```

```
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.

```
# drop null values
train = train.dropna(subset=["fare_amount","pickup_datetime","passenger_count"])
```

R code

First, we need to set the current working directory.

```
setwd("C:/Users/Pranav/Desktop/Cab Fare Project")
```

We read the training and test dataset **csv** files as follows using **read.csv()** function

```
## Read the data from csv file
train = read.csv("train_cab.csv", header = T)
test = read.csv("test.csv" , header = T)
```

Following are the data types of independent variables

```
> str(train)
'data.frame':   16067 obs. of  7 variables:
 $ fare_amount      : Factor w/ 469 levels "","-2.5","-2.9",..: 302 59 374 433 371 27 432 57 1 454 ...
 $ pickup_datetime  : Factor w/ 16021 levels "2009-01-01 01:31:49 UTC",..: 1115 2509 6550 8252 2919 4986 9743 7479 9838 1606 ...
 $ pickup_longitude : num  -73.8 -74 -74 -74 -74 ...
 $ pickup_latitude  : num  40.7 40.7 40.8 40.7 40.8 ...
 $ dropoff_longitude: num  -73.8 -74 -74 -74 -74 ...
 $ dropoff_latitude : num  40.7 40.8 40.8 40.8 40.8 ...
 $ passenger_count  : num  1 1 2 1 1 1 1 1 1 2 ...
```

We will convert the independent variable '**fare_amount**' from factor to numeric data type and '**pickup_datetime**' from factor to date/time object

```
train$fare_amount = as.numeric(paste(train$fare_amount))

train$pickup_datetime = as.POSIXlt(train$pickup_datetime, format = "%Y-%m-%d %H:%M:%S")
```

Now, we can check the missing values as follows.

```
> sapply(train, function(x) sum(is.na(x)))
     fare_amount   pickup_datetime  pickup_longitude   pickup_latitude dropoff_longitude  dropoff_latitude   passenger_count
              25                 1                 0                 0                 0                 0                55
```

We will drop the rows containing these missing values

```
#drop rows where pickup_datetime contain NA values
train = train[!is.na(train$pickup_datetime),]

#drop rows where passenger_count contain NA values
train = train[!is.na(train$passenger_count),]

#drop rows where fare_amount contain NA values
train = train[!is.na(train$fare_amount),]
```

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

$a = \sin^2(\Delta\varphi/2) + \cos\varphi_1 \cdot \cos\varphi_2 \cdot \sin^2(\Delta\lambda/2)$
$c = 2 \cdot \text{atan2}(\sqrt{a}, \sqrt{(1-a)})$
$d = R \cdot c$
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!

Python code

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

```
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"])
```

```
# 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

```
# unique values of year
train['year'].unique()
```

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

```
# 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

```
def distance(pickup_latitude, pickup_longitude, dropoff_latitude, dropoff_longitude):
    data = [train, test]
    for i in data:
        R = 6371  #radius of earth in kilometers
        #R = 3959 #radius of earth in miles
        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

```
distance('pickup_latitude', 'pickup_longitude', 'dropoff_latitude', 'dropoff_longitude')
```

Following is the updated training dataset

```
train.head()
```

	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

R code

We extract year, hour, day, month from the pickup_datetime for training and test data

```
# splitting the pickup_datetime into year,month,hour and day for training data
train$year = as.numeric(format(train$pickup_datetime,"%Y"))
train$year = as.factor(train$year)
train$year = as.numeric(train$year)

train$month = as.numeric(format(train$pickup_datetime,"%m"))


train$hour = as.numeric(format(train$pickup_datetime, "%H"))
train$day = train$pickup_datetime$wday

train$pickup_datetime = NULL
```

```
# doing same tranformations for test data

test$pickup_datetime = as.POSIXlt(test$pickup_datetime, format = "%Y-%m-%d %H:%M:%S")
test$year = as.numeric(format(test$pickup_datetime,"%Y"))
test$year = as.factor(test$year)
test$year = as.numeric(test$year)

test$month = as.numeric(format(test$pickup_datetime,"%m"))


test$hour = as.numeric(format(test$pickup_datetime, "%H"))
test$day = test$pickup_datetime$wday

test$pickup_datetime = NULL
```

Next, we create a function that calculates the **haversine** distance between the pickup coordinates and the dropoff coordinates

```
# function to calculate distance
haversine <- function(lat1, long1, lat2, long2){

  R = 6371 # Radius of the earth

  phi1 = DegToRad(lat1)
  phi2 = DegToRad(lat2)

  delta_phi = DegToRad(lat2 - lat1)
  delta_lambda = DegToRad(long2 - long1)

  a = sin(delta_phi/2) ** 2 + cos(phi1) * cos(phi2) * sin(delta_lambda/2) **2

  c = 2 * atan2(sqrt(a), sqrt(1-a))

  d = R * c

  return (d)
}
```

We create a column H_Distance which represents the distance between the coordinates

```
# creating new column H_Distance for the train data
for(i in 1:nrow(train)){
  train$H_Distance[i] = haversine(train$pickup_latitude[i],
                                  train$pickup_longitude[i],
                                  train$dropoff_latitude[i],
                                  train$dropoff_longitude[i])
}

# also creating H_Distance column for test data
for(i in 1:nrow(test)){
  test$H_Distance[i] = haversine(test$pickup_latitude[i],
                                 test$pickup_longitude[i],
                                 test$dropoff_latitude[i],
                                 test$dropoff_longitude[i])
}
```

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.

Python code

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

```
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

```
train.sort_values(by = "Distance")
```

12228	16.50	-73.993360	40.721749	-73.977370	39.610604	2.0	2	4	1	23	123.561157
11619	11.30	-73.945106	39.603178	-73.976636	40.749643	1.0	2	5	4	21	127.509261
14536	4.10	-73.982155	40.775187	-73.976399	39.610030	1.0	2	4	6	12	129.560455
10710	3.70	-73.955411	39.604164	-73.955647	40.772837	1.0	2	2	4	21	129.950482
7014	4.50	-73.988980	40.721697	-74.001073	0.728087	3.0	2	6	5	4	4447.086698
5864	8.50	-73.995030	40.744945	-7.986640	40.729937	1.0	3	3	6	1	5420.988959
2280	8.90	-73.936667	40.757815	0.000000	40.757815	1.0	2	8	0	8	6026.494216
15749	10.90	-73.967183	40.772403	0.000000	40.740677	1.0	3	5	5	17	6028.926779
15783	26.50	0.000000	0.000000	-73.867056	40.768823	1.0	5	9	6	8	8656.714168

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.

```
# 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.

```
train["passenger_count"].unique()
```

```
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, 5.312e+02, 1.300e+00,
       1.200e-01, 5.570e+02])
```

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

```
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.

```
# 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.

```
train.sort_values(by = "fare_amount")
```

9431	88.00	-73.989335	40.744471	-74.183504	40.688282	1.0	5	4	4	14	17.516711
7810	95.00	-73.862951	40.768771	-73.652193	40.657995	2.0	2	10	0	8	21.616360
12915	96.00	-73.652179	40.801690	-73.652179	40.801690	1.0	2	9	2	1	0.000000
12349	104.67	-73.797225	40.639720	-73.840545	41.030892	1.0	1	5	5	15	43.648755
14142	108.00	-74.013570	40.705930	-74.000740	40.738960	1.0	5	12	4	10	3.828609
6630	128.83	0.000000	0.000000	0.000000	0.000000	1.0	5	5	4	1	0.000000
1483	165.00	-73.633363	41.032490	-73.633671	41.032598	1.0	0	8	4	21	0.028489
1335	180.00	-74.429332	40.500046	-74.429332	40.500046	1.0	4	1	2	20	0.000000
980	434.00	-73.990602	40.761100	-73.960025	40.779580	2.0	2	10	0	22	3.294366
607	453.00	-74.007816	40.733536	-73.986556	40.740040	1.0	2	3	3	7	1.931731
1072	4343.00	-73.976309	40.751634	-74.014854	40.709044	1.0	3	1	6	20	5.742518
1015	54343.00	-74.003319	40.727455	-73.964470	40.764378	1.0	6	2	4	17	5.250392

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

```
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.

```
from collections import Counter
```

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

```
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

```
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]
```

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

```
train.update(subset)
```

R code

Latitudes and longitudes are bounded between (-90, 90) and (-180,180) respectively. This is taken care by the following code

```
# Latitudes are bounded between (-90,90)
# Longitudes are bounded between (-180,180)
train = train[(!train$pickup_latitude < -90 & !train$pickup_latitude >90)  ,]
train = train[(!train$pickup_longitude < -180 & !train$pickup_longitude >180)  ,]

train = train[(!train$dropoff_latitude < -90 & !train$dropoff_latitude >90)  ,]
train = train[(!train$dropoff_longitude < -180 & !train$dropoff_longitude >180)  ,]
```

We remove distances which are greater than 130km as they are highly unlikely.

```
train = train[!train$H_Distance > 130 ,]
```

We also remove the records where passenger_count has decimal values and do not consider the values greater than 6

```
train = train[(((train$passenger_count * 10)%% 10)== 0) , ]
```

```
# the cab won't be able to occupy more than 6 passengers
train = train[! train$passenger_count > 6 ,]
```

As the base fare_amount is $2.5 and cannot be greater than $453 as we saw above we do the following

```
# in New York the minimum fare is $2.5
train = train[train$fare_amount >= 2.5 ,]

# we observe the fare amount value and decide to exclude values which are above 453
train = train[train$fare_amount <= 453 ,]
```

We impute the distances in those records where the distances are 0 but the fare amount is non-zero

```
train$H_Distance = ifelse( (train$H_Distance == 0) & (train$fare_amount !=0) , ((train$fare_amount -2.5)/1.56), train$H_Distance)
```

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.

Python code for EDA on continuous variables

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

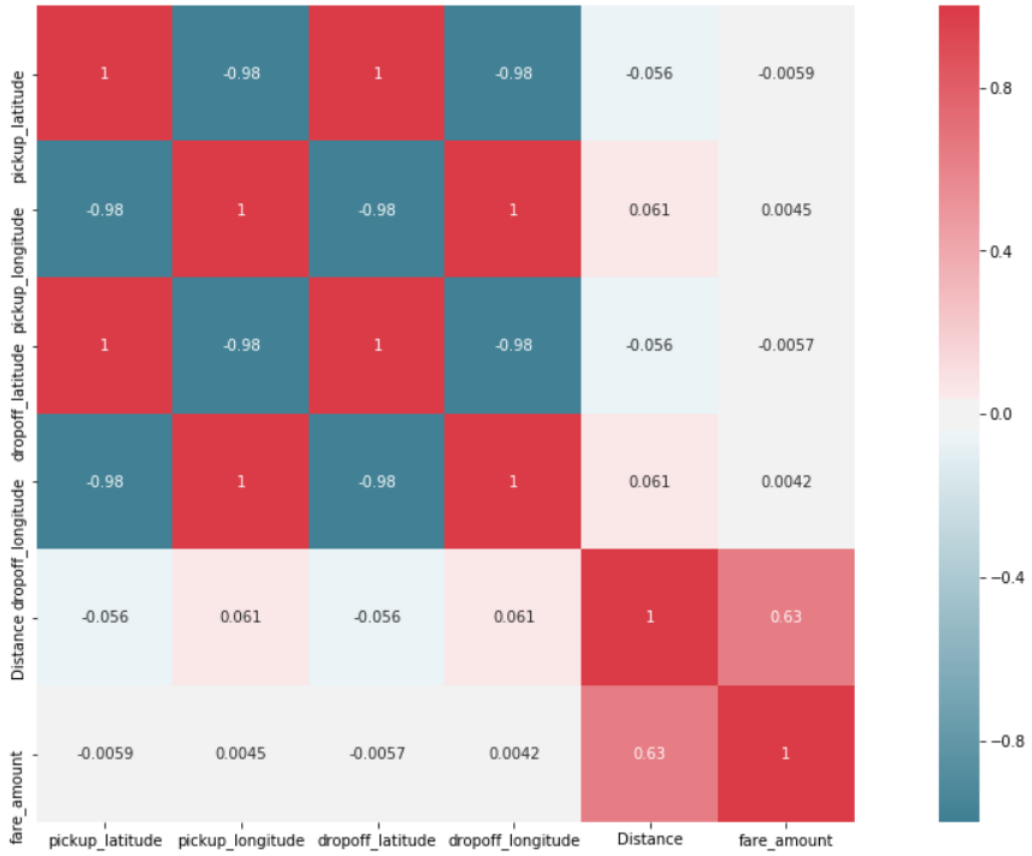
```
import matplotlib.pyplot as plt
import seaborn as sn
```

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

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

```
%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)
```

Following is the heatmap generated

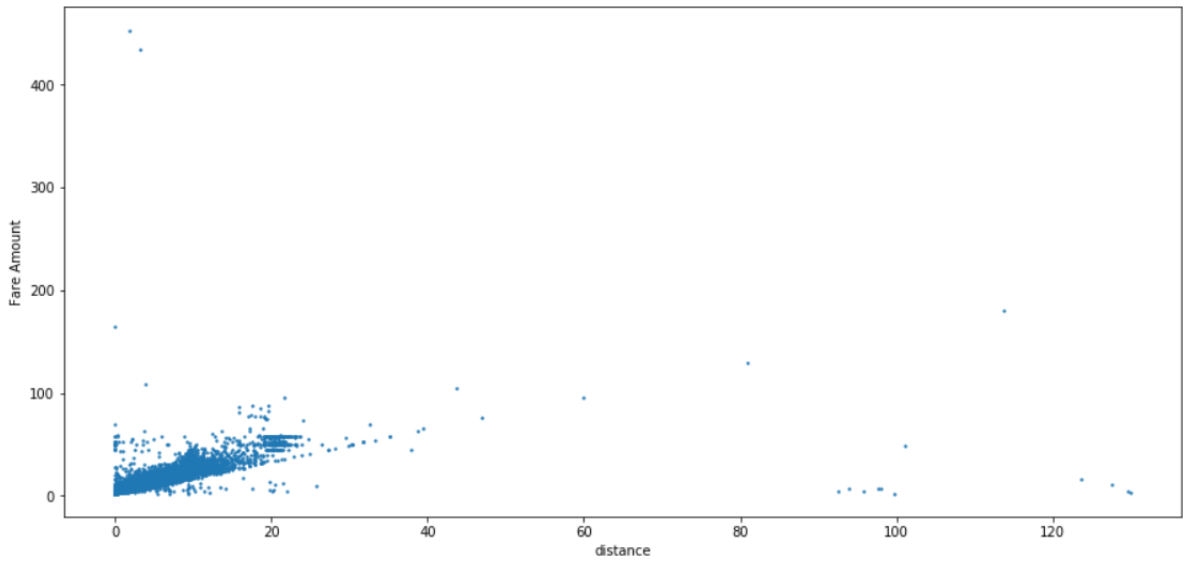


We observe that all the coordinates of latitudes and longitudes are highly correlated to each other. This will negatively impact the model. We will have to drop the columns which represent these coordinates.

```
# 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

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



R code for EDA on continuous variables

The correlation plot for continuous variables can be plotted as follows using **corrgram** library

```
library(corrgram)


numerical_variables = c("H_Distance","fare_amount","pickup_latitude","pickup_longitude","dropoff_latitude","dropoff_longitude")

# plot correlation matrix
# correlation plots can only be done on continuous variables

corrgram(train[numerical_variables], order=FALSE,
         main="correlation plot",upper.panel=panel.pie,text.panel=panel.txt)
```

From correlation plot, we see that the coordinates of latitudes and longitudes are highly correlated to each other. We have to remove all the columns representing these coordinates since they will negatively affect the model.

```
# drop pickup_latitude, pickup_longitude, dropoff_latitude, dropoff_longitude

train = train[,-c(2:5)]

# do smae for test data
test = test[, -c(1:4)]
```

The scatter plot of H_Distance vs fare_amount can be plotted as follows using **ggplot2** library

```
library(ggplot2)

# scatter plot of H_Distance vs fare_amount
s = ggplot(train, aes(x = H_Distance, y = fare_amount)) +
  geom_point()+
  scale_x_continuous(breaks = pretty_breaks(n = 10))

plot(s)
```

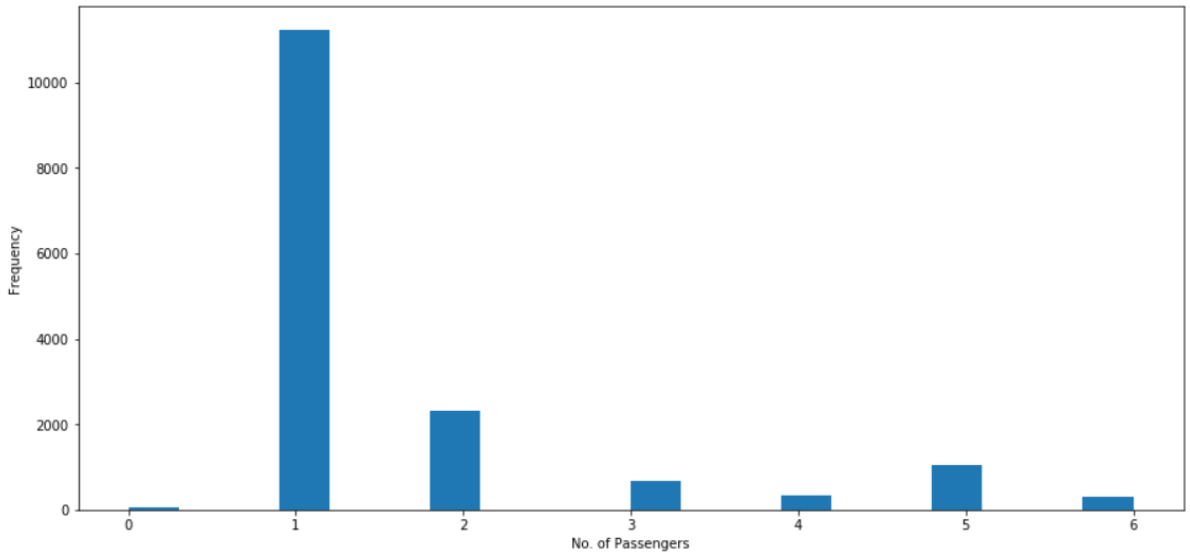
EDA on Categorical variables

We need to understand the relation between the categorical variables and the dependent variable ‘fare_amount’ which we do using scatter plots and histograms.

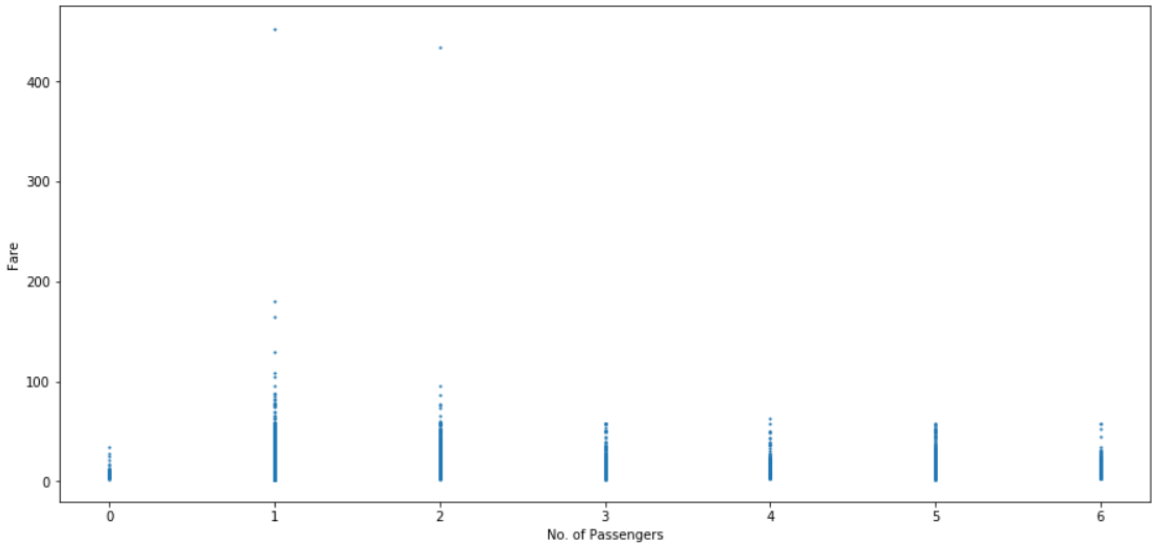
Python code for EDA on Categorical variables

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

```
%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));
```



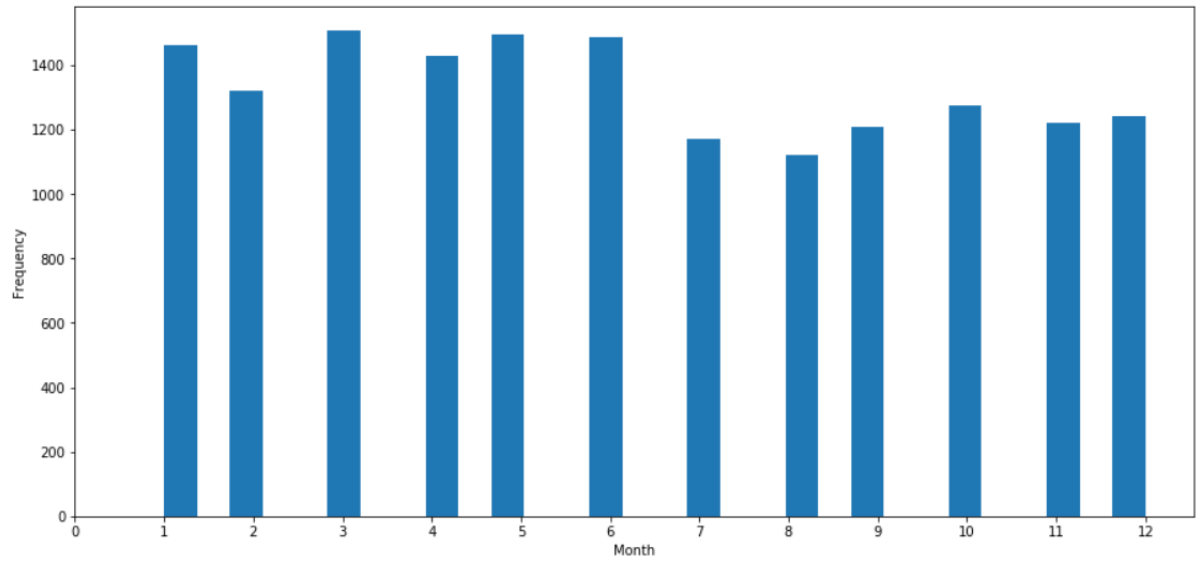

```
plt.figure(figsize=(15,7))
plt.scatter(x=train['passenger_count'], y=train['fare_amount'], s=1.5)
plt.xlabel('No. of Passengers')
plt.ylabel('Fare');
```



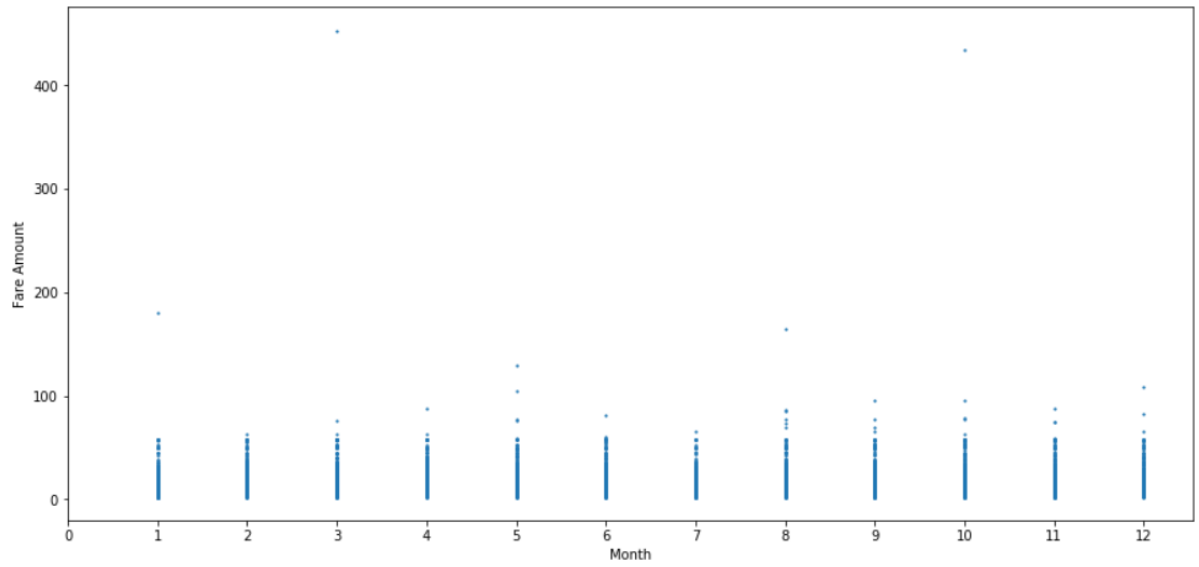
From the above two graphs we observe that the single passengers are the frequent travellers. The highest cab fare is paid by them only

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));
```




```
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));
```

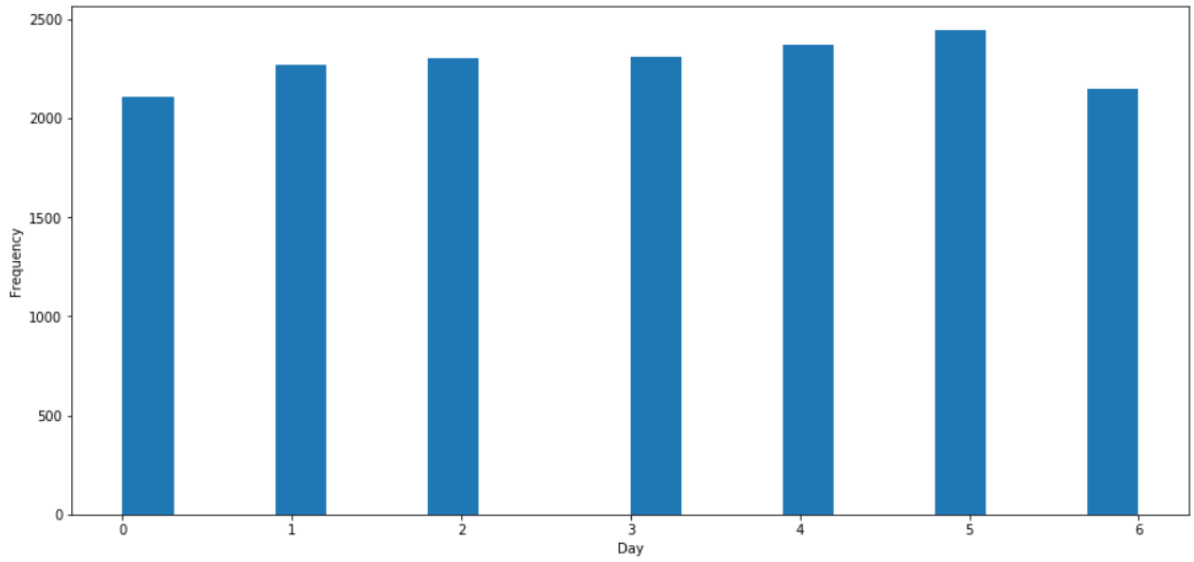


From the two plots above we see that, the frequency of cabs is highest in the month of March and the highest cab fare was also noted in the same month

Let us check how each Day of the week affects the fare amount and the frequency of cabs

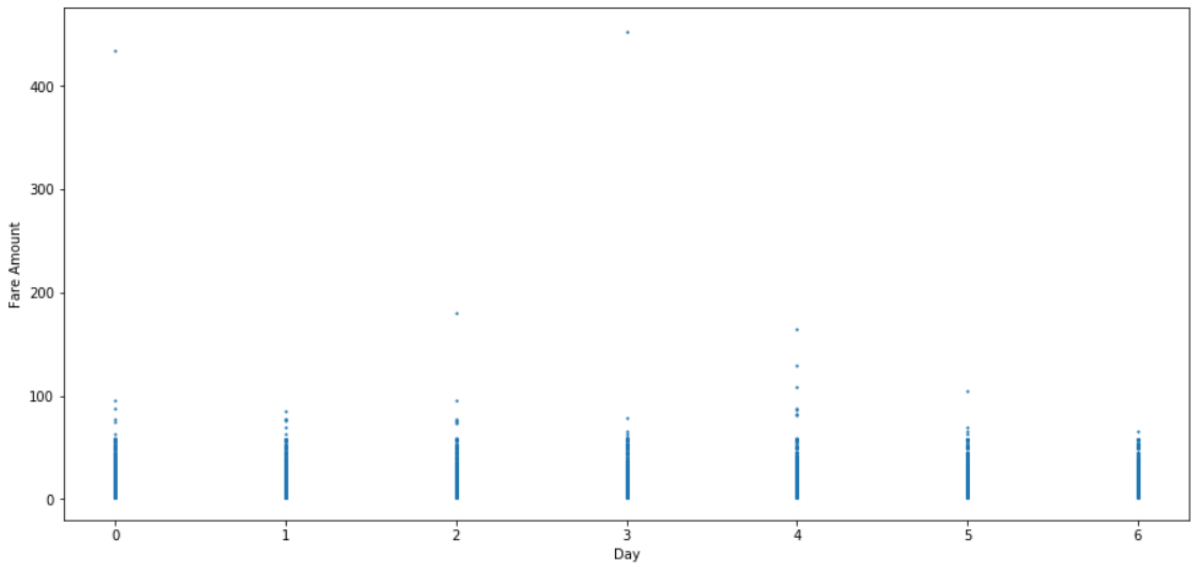
```
plt.figure(figsize=(15,7))
plt.hist(train['Day'], bins=20)
plt.xlabel('Day')
plt.ylabel('Frequency')
```

```
Text(0, 0.5, 'Frequency')
```




```
plt.figure(figsize=(15,7))
plt.scatter(x=train['Day'], y=train['fare_amount'], s=1.5)
plt.xlabel('Day')
plt.ylabel('Fare Amount')
```

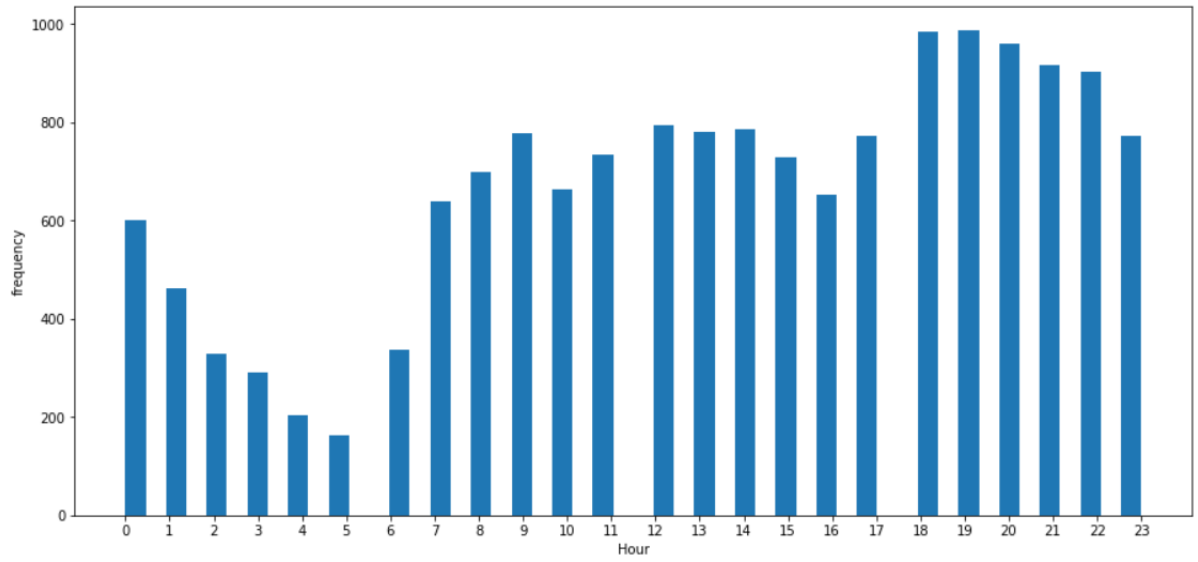
```
Text(0, 0.5, 'Fare Amount')
```



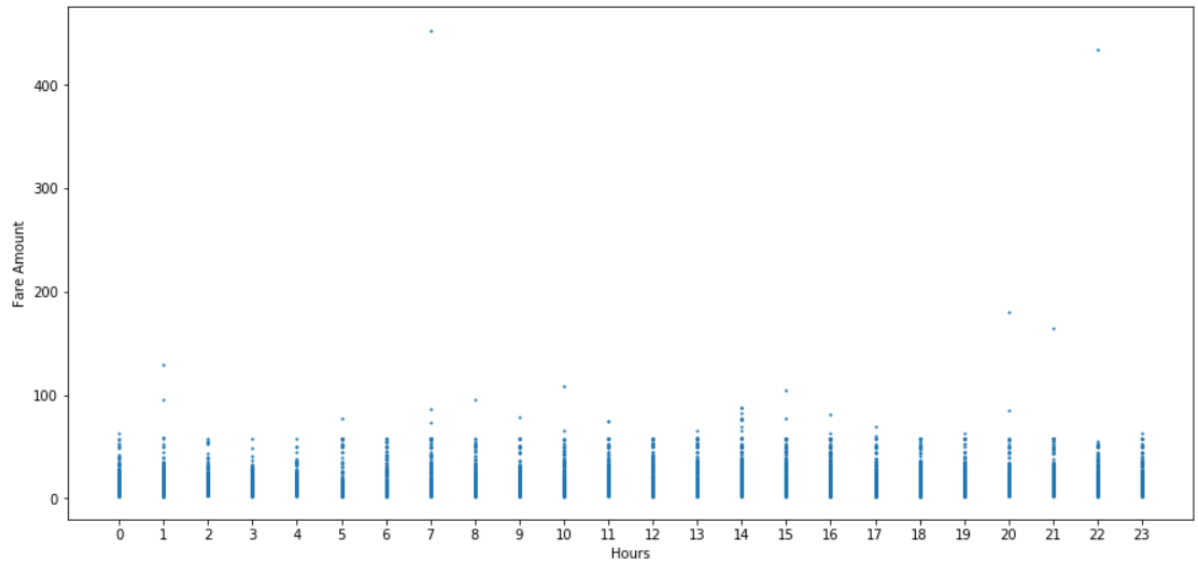
From the two plots we observe that, the frequency of cabs is highest on Saturday and the highest cab fare was noted on Thursday.

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

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




```
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));
```

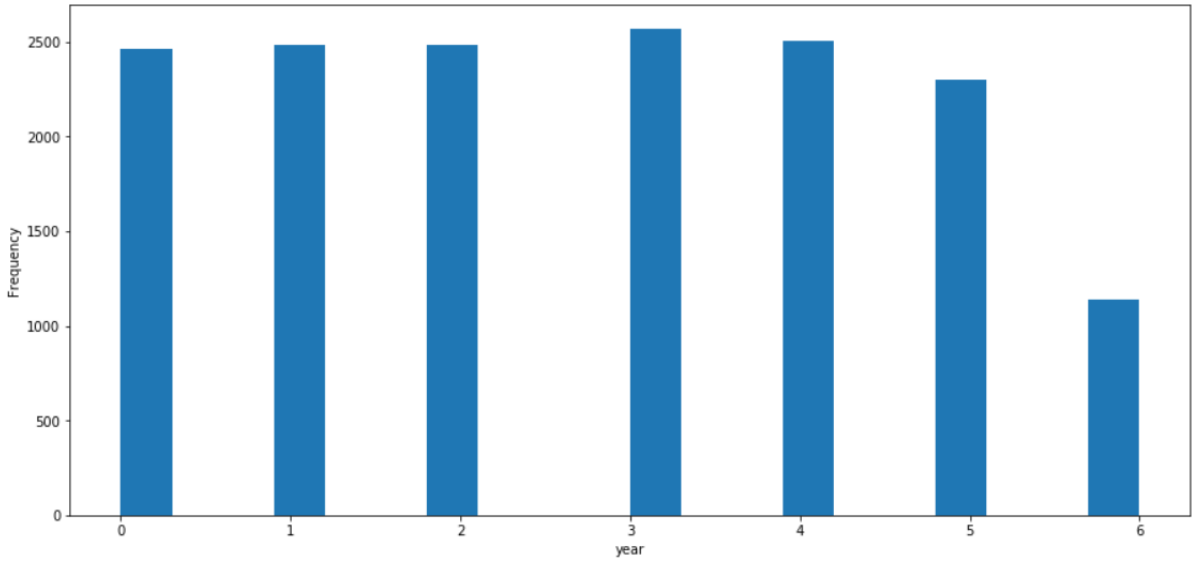


From the two graphs above we see that, frequency of cabs is lowest at 5 AM in the morning and highest at 7 PM in the evening. The highest cab fare was noted at 7 AM in the morning

Let us check how year has affected the frequency of cabs and the fare amount

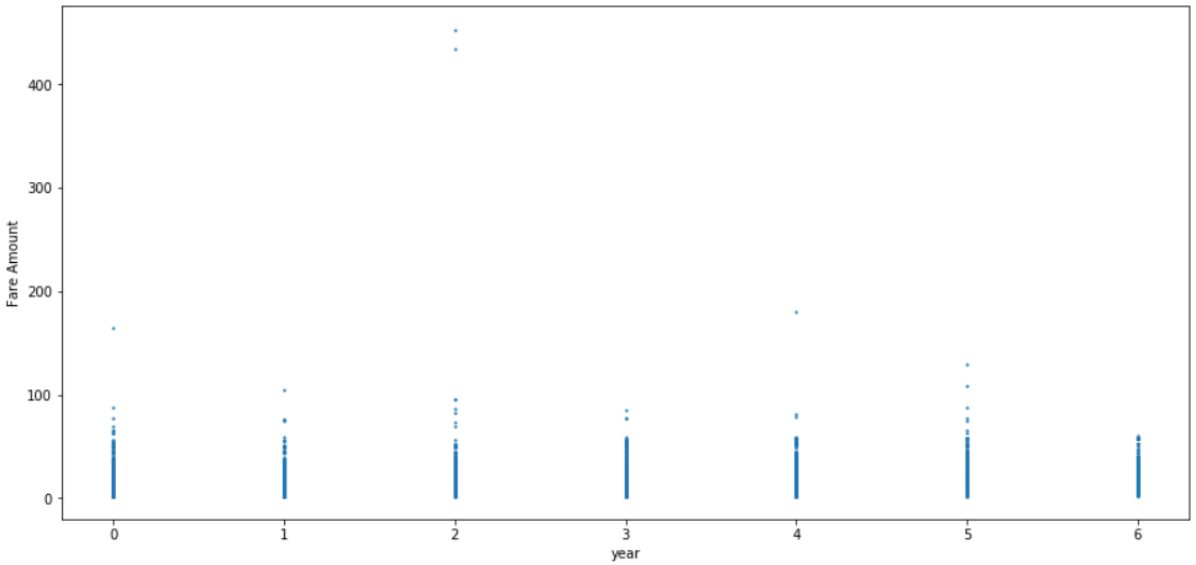
```
plt.figure(figsize=(15,7))
plt.hist(train['year'], bins=20)
plt.xlabel('year')
plt.ylabel('Frequency')
```

```
Text(0, 0.5, 'Frequency')
```




```
plt.figure(figsize=(15,7))
plt.scatter(x=train['year'], y=train['fare_amount'], s=1.5)
plt.xlabel('year')
plt.ylabel('Fare Amount')
```

```
Text(0, 0.5, 'Fare Amount')
```



The frequency of cabs seem to be decreasing from 2014-2015. In the previous years it has been more or less the same. The highest fare amount was noted in 2011.

R code for EDA on categorical variables

The effect of categorical variables like passenger_count, hour, day, month, year on the frequency of cabs and the fare amounts can be visualised by plotting the histograms and scatter plots as follows.

```
### Does passenger_count affect the fare_amount

p<-ggplot(train, aes(x=passenger_count)) +
  geom_histogram(fill="blue" ) +
  scale_x_continuous(breaks = pretty_breaks(n = 6))

plot(p)


s = ggplot(train, aes(x = passenger_count, y = fare_amount)) +
  geom_point()+
  scale_x_continuous(breaks = pretty_breaks(n = 6))

plot(s)
```

The plots for the others variables can be visualised in the similar manner.

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.

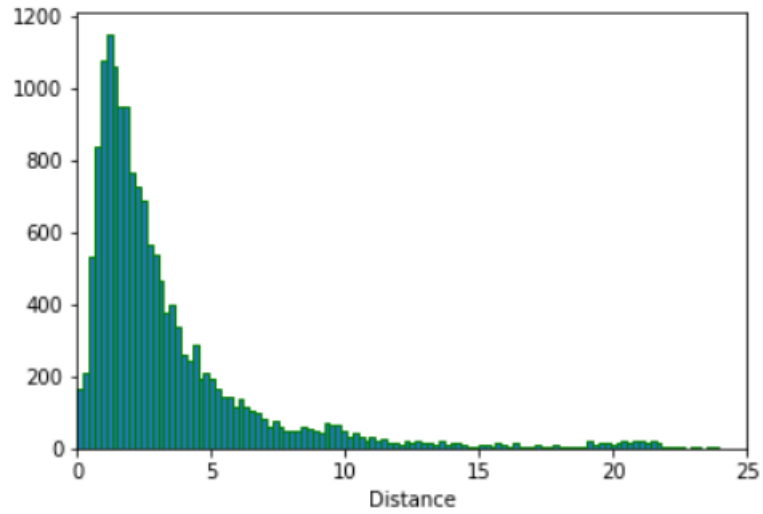
Common transformations to the data include square root, log, etc.

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

Python code for handling skewness of data

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()
```

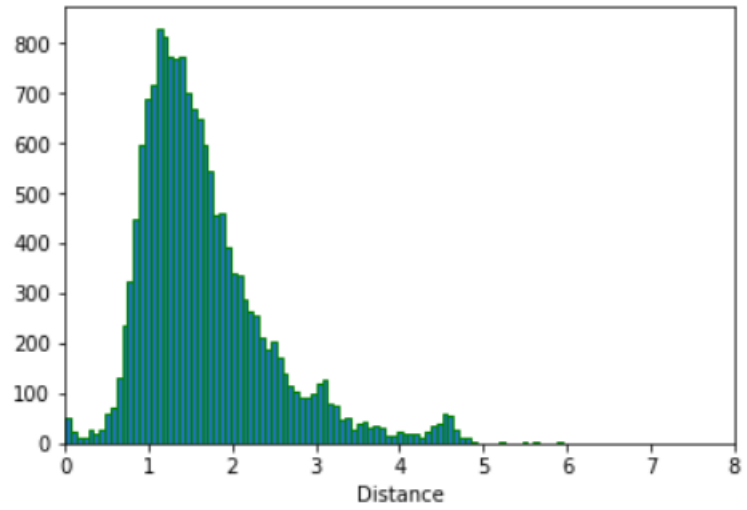


We observe that it is positively skewed. We will apply square root transformation on it.

```
# 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

R code for handling skewness of data

Following code is used for visualising the histograms and transforming the skewed H_Distance data

```
##### Histogram plot of numerical variable H_Distance

p<-ggplot(train, aes(x=H_Distance)) +
  geom_histogram(color="black", fill="green") +
  xlim(0,25)

plot(p)

# We see that the data is skewed . Let's perform square root transformation

train$H_Distance = sqrt(train$H_Distance)
test$H_Distance = sqrt(test$H_Distance)

p<-ggplot(train, aes(x=H_Distance)) +
  geom_histogram(color="black", fill="green") +
  xlim(0,8)

plot(p)
```

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 code

In python, we need to import **train_test_split()** function from **scikit learn** (**sklearn**) library to split the data into training and validation data

```
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**)

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

We also import **mean_squared_error()** , **mean_absolute_error** to compute **Mean Square Error** and **Mean Absolute Error** values respectively

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

R code

In R, using the **sample()** function, we can convert the data into training and validation data.

```
set.seed(1234)
# Divide the data set into train and test
train_index = sample(1:nrow(train) , 0.80 * nrow(train))

trainSplit = train[train_index,]
validationSplit = train[-train_index,]
```

2.2.2. Linear Regression

Python code

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

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

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

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

Following are the model evaluation metrics we obtain

```
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
```

R code

To apply Linear Regression, we need to use the predefined function **lm()** in R

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

```
### Linear Regression

linearModel = lm( fare_amount ~ . , data = trainSplit)

predictions_LR = predict(linearModel,validationSplit[,-1])
```

We use the **eval()** function from **DMwR** library to obtain evaluation metrics

```
library(DMwR)

> regr.eval(validationSplit[,1] ,predictions_LR , stats = c('mae','rmse','mape','mse') )
       mae       rmse       mape        mse
 2.9943245  9.3862861  0.2971989 88.1023660
```

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.

R code for Decision Tree Regression

We have considered CART (Decision Tree Regression) algorithm. For that we have to use **rpart()** package

```
library(rpart)
```

Next, we build the model using **rpart()** function and make predictions for the validation set using **predict()** function

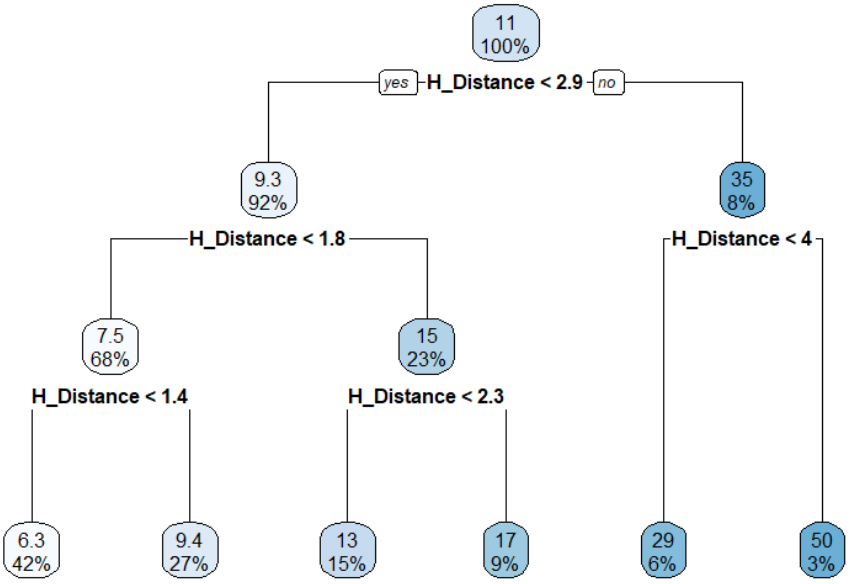
```
set.seed(1234)

fit = rpart(fare_amount ~. , data = trainSplit , method = "anova")

# Predict for new test cases
predictions = predict(fit,validationSplit[,-1])
```

We obtain the following evaluation metrics

```
> regr.eval(validationSplit[,1], predictions , stats = c('mae','rmse','mape','mse'))
      mae      rmse      mape       mse
 2.751829  8.939539  0.269907 79.915352
```

We plot the generated decision tree

```
rpart.plot::rpart.plot(fit)
```



Next, we perform **hyperparameter tuning** using the caret package in R. Method '**rpart**' is capable of tuning only **cp** parameter. Here **cp** is the complexity parameter used by **rpart** to determine when to prune the tree. The **caret** package contains **train()** function which is used to fit predictive models over different tuning parameters. The **trainControl()** function controls the training parameters.

```
# tuning the model using caret
library(caret)
rpart_grid = expand.grid(cp = c(0.1,0.2,0.01,0.02,0.001,0.002))

# Define the control
trControl <- trainControl(method = "cv", number = 3, search = "grid",
                          verboseIter = TRUE)

set.seed(1234)

tuned_tree_model = caret::train(fare_amount ~ .,
                                data = trainSplit,
                                method = "rpart",
                                trControl = trControl,
                                tuneGrid = rpart_grid)
```

We get optimal **cp** value as 0.001

```
> tuned_tree_model$bestTune$cp
[1] 0.001
```

Taking optimal parameter, we make the predictions on validation set

```
predictions = predict(tuned_tree_model,validationSplit[,-1])
```

```
> regr.eval(validationSplit[,1], predictions , stats = c('mae','rmse','mape','mse'))
       mae       rmse       mape        mse
 2.4856209  8.7878333  0.2348219 77.2260142
```

Here, we see that there is an improvement in **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.

R code for Random Forest Regression

We use the **randomForest()** function from **randomForest** library and make predictions using **predict()** function

```
library(randomForest)
set.seed(1234)
rf_model = randomForest(fare_amount~.,data = trainSplit)

# Predict for new test cases
rf_predictions = predict(rf_model,validationSplit[,-1])
```

We get the following evaluation metrics

```
> regr.eval(validationSplit[,1], rf_predictions , stats = c('mae','rmse','mape','mse'))
       mae       rmse       mape        mse
 2.3868076  8.6900050  0.2345371 75.5161873
```

Further, we do **hyperparameter tuning** using the caret package. We experiment with different **mtry** values Here **mtry** is the number of variables randomly sampled as a candidate at each split node. We perform a repeated 3-fold cross validation and find the optimal parameters.

```
set.seed(1234)

# default ntrees = 500
rf_tuned_model = caret::train(fare_amount~.,
                              data = trainSplit,
                              method = 'rf',
                              trControl = trainControl(method = "repeatedcv",
                                                       number = 3,
                                                       verboseIter = TRUE
                              ),
                              tuneGrid = expand.grid(.mtry = c(1:6)),
                              ntree = 500
)


rf_tuned_predictions = predict(rf_tuned_model,validationSplit[,-1])
```

We get the following optimal parameter

```
rf_tuned_model$bestTune
mtry
   3
```

Now we make the predictions

```
rf_tuned_predictions = predict(rf_tuned_model,validationSplit[,-1])
```

We get the following evaluation metrics

```
> regr.eval(validationSplit[,1], rf_tuned_predictions , stats = c('mae','rmse','mape','mse'))
       mae       rmse       mape        mse
 2.3444731  8.6563479  0.2251159 74.9323584
```

We see that there is improvement in RMSE and other 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

R code for Gradient Boosting Regression

We are making use of **caret** package for tuning the parameters in Gradient Boost Regression. We have used different combinations **of interaction.depth** , **n.trees** , **shrinkage** and **n.minobsinnode.** Here **interaction.depth** indicates maximum depth of each tree, **n.trees** indicates total number of trees to fit, **shrinkage** is the learning rate, **n.minobsinnode** indicates number of observations in the terminal node. We are performing 3-fold cross validation on our training data.

```
gbmGrid = expand.grid(interaction.depth = c(3,4,5),
                      n.trees = c(130,140,150) ,
                      shrinkage = 0.1 ,
                      n.minobsinnode = c(2,4,6))

# Define the control
trControl <- trainControl(method = "cv", number = 3, search = "grid",
                          verboseIter = TRUE)

set.seed(1234)
gbr_tuned_model = caret::train(fare_amount ~ .,data = trainSplit, method = "gbm",
                               trControl = trControl, tuneGrid = gbmGrid ,
                               verbose = FALSE)
```

We get the following optimal parameter

```
gbr_tuned_model$bestTune
 n.trees interaction.depth shrinkage n.minobsinnode
     150                 4       0.1              2
```

Now we make the predictions and compute the evaluation metrics

```
> regr.eval(validationSplit[,1], gbr_tuned_predictions , stats = c('mae','rmse','mape','mse'))
       mae       rmse       mape        mse
 2.2816108  8.6242734  0.2186601 74.3780912
```

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.

Python

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

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)
```

R programming

The following table summarizes the results obtained by different regression models in R

Regression Model	MAE	MAPE	MSE	RMSE
Linear Regression	2.9943	0.2971	88.1023	9.3862
Decision Tree	2.4856	0.2348	77.2260	8.7878
Random Forest	2.3444	0.2251	74.9323	8.6563
Gradient Boosting	2.2816	0.2186	74.37	8.6242

We observe that Gradient Boosting Regression achieves the lowest RMSE among all the algorithms. Hence we choose it to make predictions on the test dataset and save it to csv file.

```
#### making final predictions and storing to csv file
final_predictions = predict(gbr_tuned_model , test)
test$fare_amount = final_predictions
test$fare_amount = ifelse( test$fare_amount < 2.5 , 2.5, test$fare_amount)
write.csv(test,"Predictions in R.csv", row.names = FALSE)
```