edWiser

CAB FARE PREDICTION

Subodh Pradhan

27-Nov-19

Table of Contents

Chapter -1

INTRODUCTION

A car rental business rents vehicles at affordable daily and weekly prices. All sorts of different parties are interested in car rentals. Common customers include business and leisure travelers, those whose vehicles are out of commission and businesses.

Now a day's cab rental services are expanding with the multiple rates. The ease of using the services and flexibility gives their customer a great experience with competitive prices.

1.1 Problem statement

You are a cab rental start-up 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

Here we have given two dataset having 16067 observations and 7 variables including target variable in train data and 9914 observations and 6 variables excluding target variable. Dataset contains missing value also.

The attributes in the dataset are

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

Missing Values : Yes

The data sets are given below

Table 1.1 Train dataset

	fare_amount	pickup_datetime	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
0	4.5	2009-06-15 17:26:21 UTC	-73.844311	40.721319	-73.841610	40.712278	1.0
1	16.9	2010-01-05 16:52:16 UTC	-74.016048	40.711303	-73.979268	40.782004	1.0
2	5.7	2011-08-18 00:35:00 UTC	-73.982738	40.761270	-73.991242	40.750562	2.0
3	7.7	2012-04-21 04:30:42 UTC	-73.987130	40.733143	-73.991567	40.758092	1.0
4	5.3	2010-03-09 07:51:00 UTC	-73.968095	40.768008	-73.956655	40.783762	1.0

Table 1.2 Test dataset

	pickup_datetime	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
0	2015-01-27 13:08:24 UTC	-73.973320	40.763805	-73.981430	40.743835	1
1	2015-01-27 13:08:24 UTC	-73.986862	40.719383	-73.998886	40.739201	1
2	2011-10-08 11:53:44 UTC	-73.982524	40.751260	-73.979654	40.746139	1
3	2012-12-01 21:12:12 UTC	-73.981160	40.767807	-73.990448	40.751635	1
4	2012-12-01 21:12:12 UTC	-73.966046	40.789775	-73.988565	40.744427	1

Chapter- 2

METHODOLOGY

2.1 Data pre processing

Any predictive modeling requires that we look at the data before we start modeling. However, in data mining terms *looking at data* refers to so much more than just looking. Looking at data refers to exploring the data, cleaning the data as well as visualizing the data through graphs and plots. This is often called as **Exploratory Data Analysis**. To start this process we will first try and look at all the distributions of the Numeric variables. Most analysis like regression, require the data to be normally distributed.

2.1.1 Exploratory data analysis

Exploring data the data means understanding what data speaks about. Is the data relevant to our business objective or not, if it is relevant how we can derive the data.

In exploratory data analysis first we will check the data types and structure of the data.

```
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 16067 entries, 0 to 16066
Data columns (total 7 columns):
fare_amount          16043 non-null object
pickup_datetime      16067 non-null object
pickup_longitude     16067 non-null float64
pickup_latitude      16067 non-null float64
dropoff_longitude    16067 non-null float64
dropoff_latitude     16067 non-null float64
passenger_count      16012 non-null float64
dtypes: float64(5), object(2)
memory usage: 878.7+ KB
```

In the train data we have given 16067 observations but here we can see that some values are missing in fare_amount and passenger_count variable. We will check and remove or impute these values in missing value analysis step.

```
test.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9914 entries, 0 to 9913
Data columns (total 6 columns):
pickup_datetime      9914 non-null object
pickup_longitude     9914 non-null float64
pickup_latitude      9914 non-null float64
dropoff_longitude    9914 non-null float64
dropoff_latitude     9914 non-null float64
passenger_count      9914 non-null int64
dtypes: float64(4), int64(1), object(1)
memory usage: 464.8+ KB
```

We can see that in both train and test dataset pickup_datetime is in object format so first we will convert them into proper datetime format.

In exploratory data analysis we have done the following work.

- Converted the pickup_datetime variables in both train and test dataset into proper datetime category.
- Fare_amount can't be negative or zero so we have removed those observations which are less than one and in fare_amount two values i.e 54343 and 4343 are extremely inconsistence with other observations so we have removed these values.
- In a cab passenger count should not be greater than 6 or less than one and passenger count should not be a float so we have removed those observations which are greater than 6 and less than one and also removed fractional values.
- The range of longitude is from -90 to 90 and the range of latitude is from -180 to180 so we have removed those observations which are beyond the ranges.
- Pickup_datetime variable contains date and time so we need to extract extra features like year, month, day_of week, hour to check the effect of these attributes on fare_amount and demand of cab.

After exploring the data we have 10 variables where **'fare_amount'** is the dependent or target variable and **'pickup_datetime', 'pickup_longitude', 'pickup_latitude', 'dropoff_longitude', 'dropoff_latitude', 'passenger_count', 'year', 'month', 'day_of_week', 'hour'** are the independent variables.

2.1.1 Univariate Analysis

In univariate analysis we will check the distributions of numerical variables. Now we will check the distribution of only target variable fare_amount, because now we have given longitude and latitude and after calculating distance from longitude and latitude we will check the distribution and behavior of distance variable.

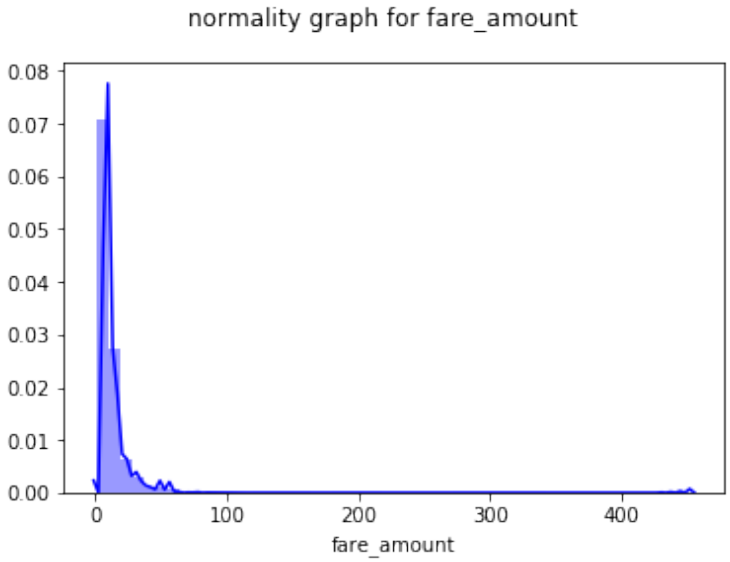
First we will check the summary of target variable.

```
train['fare_amount'].describe()
```

```
count    15635.000000
mean        11.366476
std         10.785706
min          1.140000
25%          6.000000
50%          8.500000
75%         12.500000
max        453.000000
Name: fare_amount, dtype: float64
```

The distribution graph of fare_amount is given below.

Fig 2.1 distribution of the target variable 'fare_amount'




```
skewness= 10.708226936176576
kurtosis= 343.1026732816111
```

Here we can see that the skewness of fare_amount is 10.7 and kurtosis is 343.1 that means fare_amount is not normally distributed, it is highly positively skewed.

The formula of skewness is

Skewness = 3(mean-median)/standard deviation

In python we can directly calculate the skewness and kurtosis by the functions ***skew()*** *for skewness and* ***kurt()*** *for kurtosis* which are present in **scipy.stats** library.
And in R we can calculate the skewness and kurtosis by using the library **(e1071).**

Skewness*:*

It is the degree of distortion *from the symmetrical bell curve or the normal distribution. It measures the lack of symmetry in data distribution.*
It differentiates extreme values in one versus the other tail. A symmetrical distribution will have a skewness of 0.

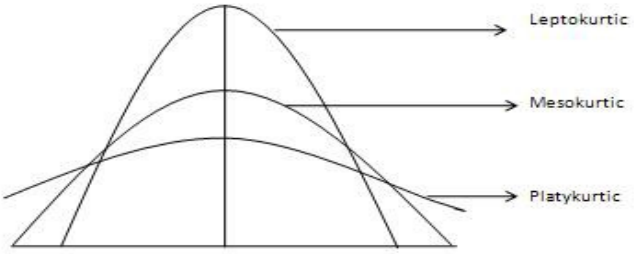
- If the skewness is between -0.5 and 0.5, the data are fairly symmetrical.

- If the skewness is between -1 and -0.5(negatively skewed) or between 0.5 and 1(positively skewed), the data are moderately skewed.
- If the skewness is less than -1(negatively skewed) or greater than 1(positively skewed), the data are highly skewed.

Kurtosis:

Kurtosis is all about the tails of the distribution — not the peakedness or flatness. It is used to describe the extreme values in one versus the other tail. It is actually the measure of outliers present in the distribution.

- **High kurtosis** in a data set is an indicator that data has heavy tails or outliers. If there is a high kurtosis, then, we need to investigate why do we have so many outliers. It indicates a lot of things, maybe wrong data entry or other things. Investigate!
- **Low kurtosis** in a data set is an indicator that data has light tails or lack of outliers. If we get low kurtosis(too good to be true), then also we need to investigate and trim the dataset of unwanted results.



Mesokurtic: This distribution has kurtosis statistic similar to that of the normal distribution. It means that the extreme values of the distribution are similar to that of a normal distribution characteristic. This definition is used so that the standard normal distribution has a *kurtosis of three.*

Leptokurtic (*Kurtosis > 3*): Distribution is longer, tails are fatter. Peak is higher and sharper than Mesokurtic, which means that data are heavy-tailed or profusion of outliers.

Outliers stretch the horizontal axis of the histogram graph, which makes the bulk of the data appear in a narrow ("skinny") vertical range, thereby giving the "skinniness" of a leptokurtic distribution.

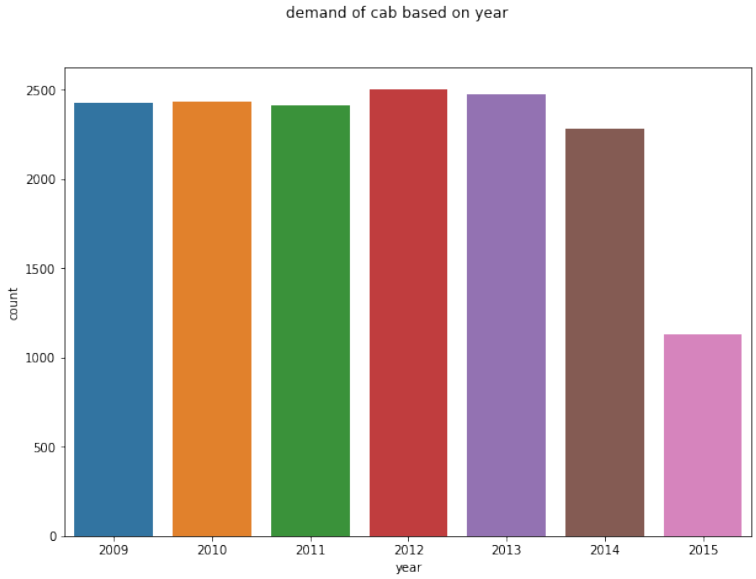
Platykurtic: (*Kurtosis < 3*): Distribution is shorter; tails are thinner than the normal distribution. The peak is lower and broader than Mesokurtic, which means that data are light-tailed or lack of outliers.

The reason for this is because the extreme values are less than that of the normal distribution

2.1.3 Bivariate Analysis

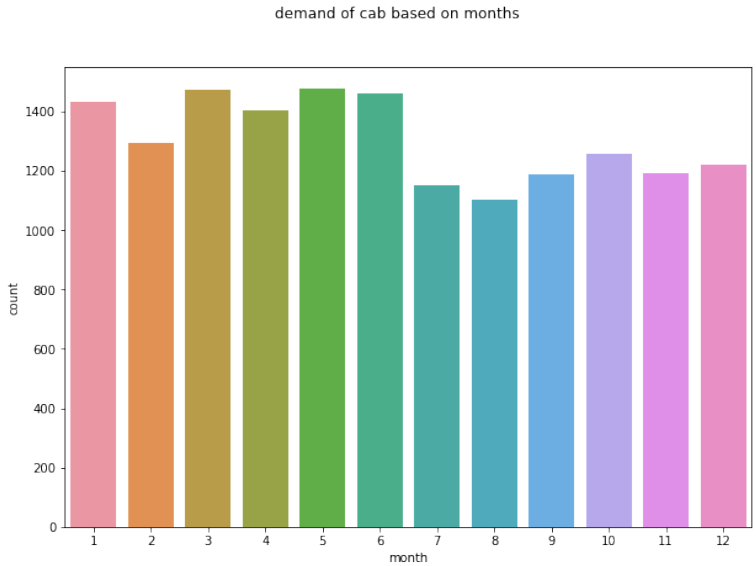
In Bivariate analysis we will determine the relationship between the target variable fare_amount and the independent variables and also determine the demand of cabs based on month, year, weekday/weekend, hour extracted from pickup_datetime variable because demand also effects the fare amount.

Fig 2.2 count plot of the variable 'year'



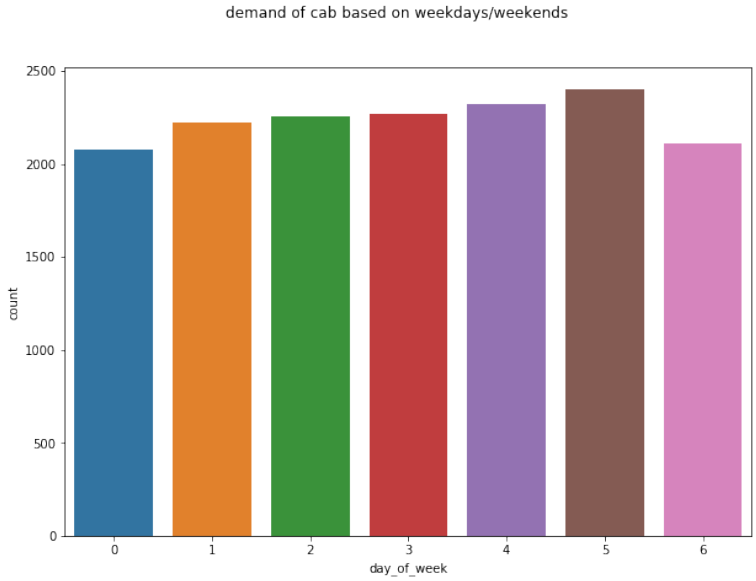
Above fig 2.2 shows that the demand of cabs based on years. In the figure we can see that the demands of cabs from 2009 to 2011 are approximately same but the demand is quit high in the year of 2012 and 2013. And after 2013 the demand s of cabs are gradually decreasing and in the year of 2015 the demand of cabs are very less as compared to the other years which may affect the fare amount.

Fig 2.3 count plot of the variable 'month'



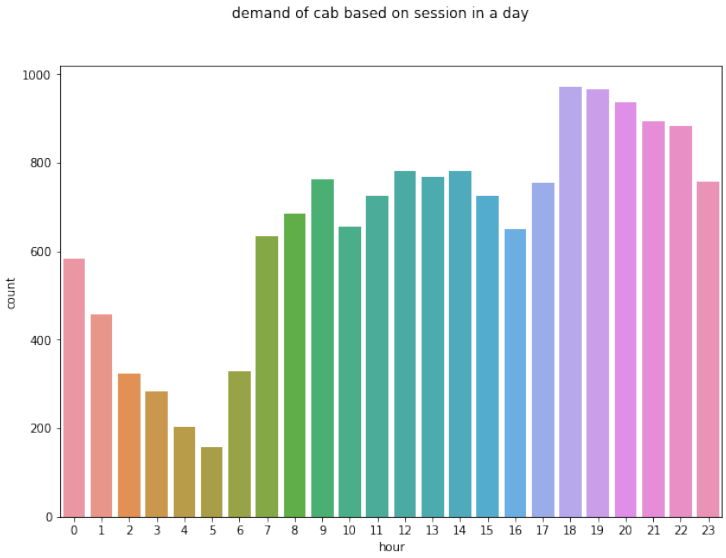
Above figure 2.3 shows the demands of cabs based on months in a year. This plots shows that the demand of cabs in the months of January to June is very high and from July to December the demands of cabs are less as compared to other months. So this is due to the seasonal effect.

Fig 2.4 count plot of the variable 'day_of_week'



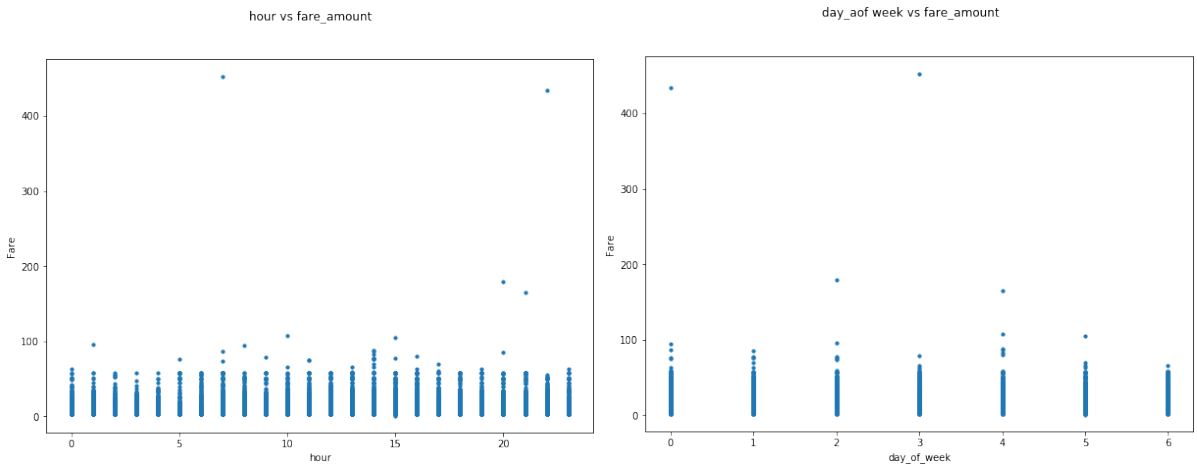
Above figure 2.4 shows the demands of cabs based on day of weeks. The demands of cabs are quit less on weekends (i.e on Saturday and Sunday) and from Monday to Friday most people are preferring cab for their transportation. So the fare might be less in weekends.

Fig 2.5 count plot of the variable 'hour'



Above figure 2.4 shows the demands of cabs based on hours in a day i.e sessions in a day. Here we can see that the demand of cab is very less in 2 am to 6 am it is very high in other hours.

Fig 2.6 Relation between hour, day_of_week and fare_amount



Above fig 2.6 shows that the relationship between hour vs fare amount and day_of week vs fare amount. In these plots we can see that there is not much difference in fare amount based day_of_week and hours. But some outliers are present in these variables so we will remove these observations in outlier analysis step and we will extract the sessions from hour and seasons from months and check their effect on fare amount in feature engineering step.

2.1.4 Missing value analysis

Missing values occur when no data value is stored for the variable in an observation. Missing data are a common occurrence and can have a significant effect on the conclusions that can be drawn from the data.

Missing value analysis is done to check whether any missing value presents in the given dataset or not. Missing values can be treated using various method like mean, median, mode, KNN imputation and prediction method to impute missing values.

In python ***isnull().sum()*** function and in R **sum**(***is.na()***) function is used to check the sum of missing values in a dataset.

Python output for missing values in train data is given in the below table

```
######################################## Missing Value Analysis ##########
#chcking for missing values in train data set.
missing_val = pd.DataFrame(train.isnull().sum())
#Reset index
missing_val = missing_val.reset_index()
missing_val
```

	index	0
0	fare_amount	22
1	pickup_datetime	0
2	pickup_longitude	0
3	pickup_latitude	0
4	dropoff_longitude	0
5	dropoff_latitude	0
6	passenger_count	55
7	year	0
8	month	0
9	day_of_week	0
10	hour	0

In train data there are 22 missing values in fare_amount and in passenger_count 55 values are missing. So we will calculate the percentage of missing values and if the missing percentage is greater than 30 we will impute them with central statistics method or KNN imputation method.

```
# Calculating missing percentage
#Rename variable
missing_val = missing_val.rename(columns = {'index': 'Variables', 0: 'Missing_percentage'})
missing_val['Missing_percentage'] = (missing_val['Missing_percentage']/len(train))*100
#descending order
missing_val = missing_val.sort_values('Missing_percentage', ascending = False).reset_index(drop = True)
missing_val
```

	Variables	Missing_percentage
0	passenger_count	0.351281
1	fare_amount	0.140512
2	pickup_datetime	0.000000
3	pickup_longitude	0.000000
4	pickup_latitude	0.000000
5	dropoff_longitude	0.000000
6	dropoff_latitude	0.000000
7	year	0.000000
8	month	0.000000
9	day_of_week	0.000000
10	hour	0.000000

Here 0.35% of data in passenger_count and 0.14% of data in fare_amount are missing, so we will impute these values.

We have used central statistics and KNN imputation method to impute the values of the variables.

Imputing the values of passenger count

Actual value = 2

Imputed with mode = 1

Imputed with KNN = 2 (2.69)

Since the value after imputing with KNN is equal or closed with the actual value, will select KNN imputation to impute the values of passenger_count.

Imputing the values of passenger count

Actual value = 5.7

Imputed with mean = 11.36

Imputed with mode = 8.5

Imputed with KNN = 5.58

Since the value after imputing with KNN is closed to the actual value, we will select the KNN imputation to impute the values of fare_amount.

We have used elbow method to select the optimal value of k to impute values.

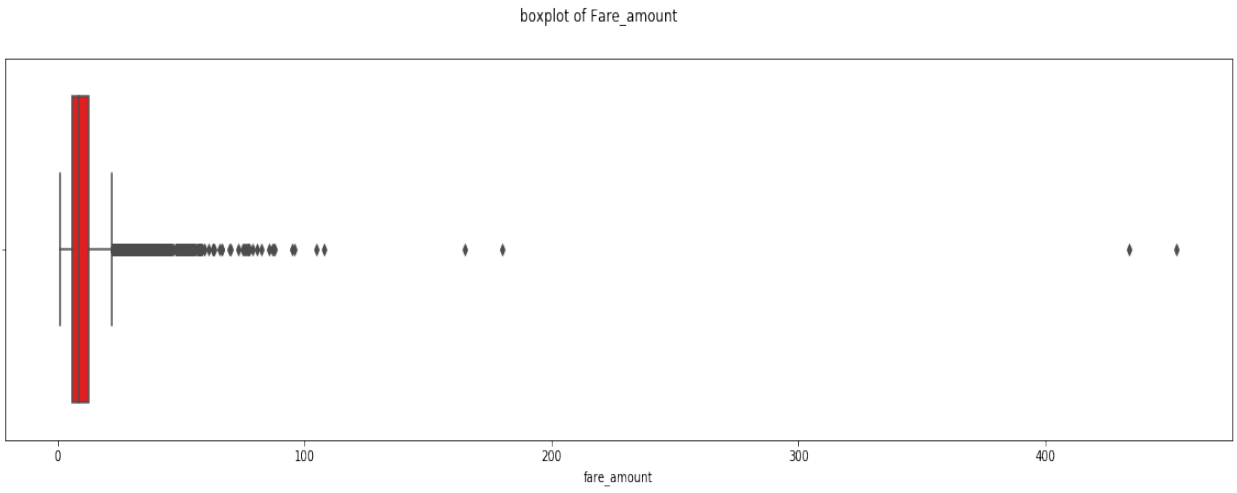
2.1.5 Outlier Analysis

An outlier is an observation which is inconsistent with rest of the dataset. It is an observation which lies an abnormal distance from the other values in the dataset or any value is falling away from the actual bunch is count as an outlier. So we will perform outlier analysis to handle all inconsistent observations present in the given dataset.

Outliers can be present only on continuous variables and not in categorical variable so we will check outliers only for the continuous variables i.e fare_amount and longitude, latitude by using box plot. Now we will check for the outliers only in fare_amount variable and after feature engineering we will check the outliers in longitude and latitude variable.

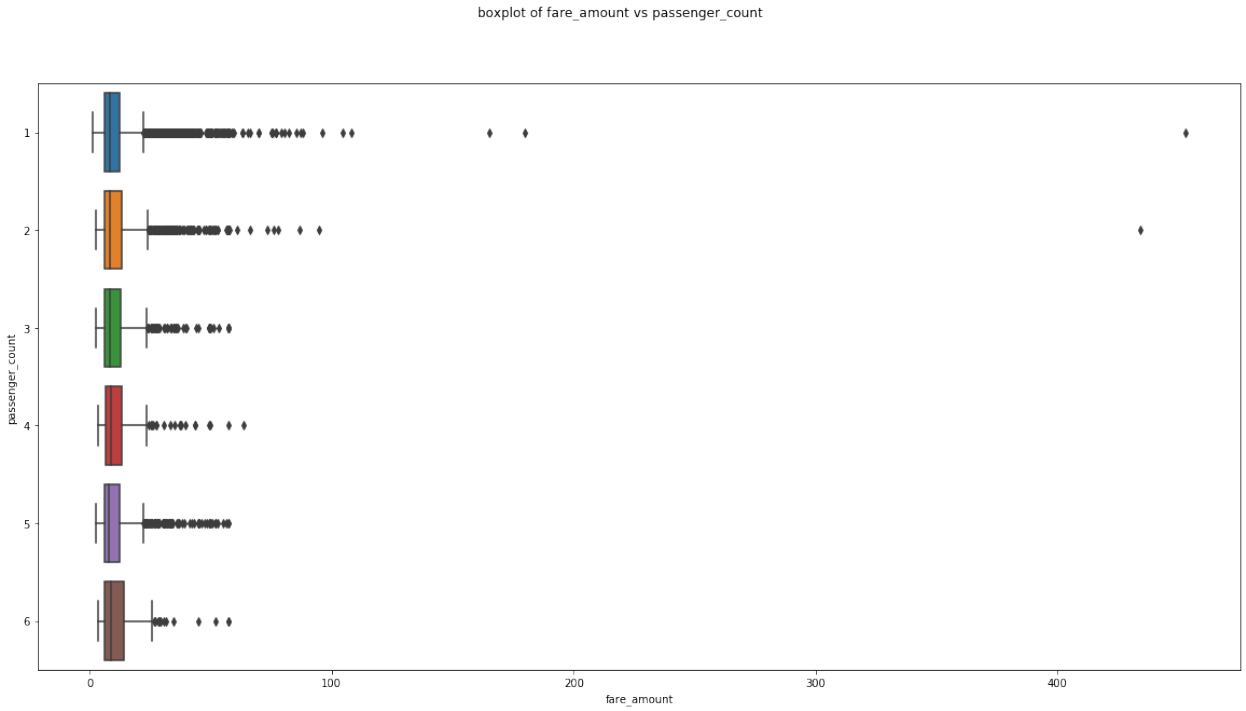
Boxplot : *boxplot is a method for graphically depicting groups of numerical data through their quartiles. Box plots may also have lines extending vertically from the boxes (whiskers) indicating variability outside the upper and lower quartiles*

Fig 2.7 box plot for the target variable 'fare_amount'



From the above figure 2.7 we can see that there are many outliers present in the target variable fare_amount.

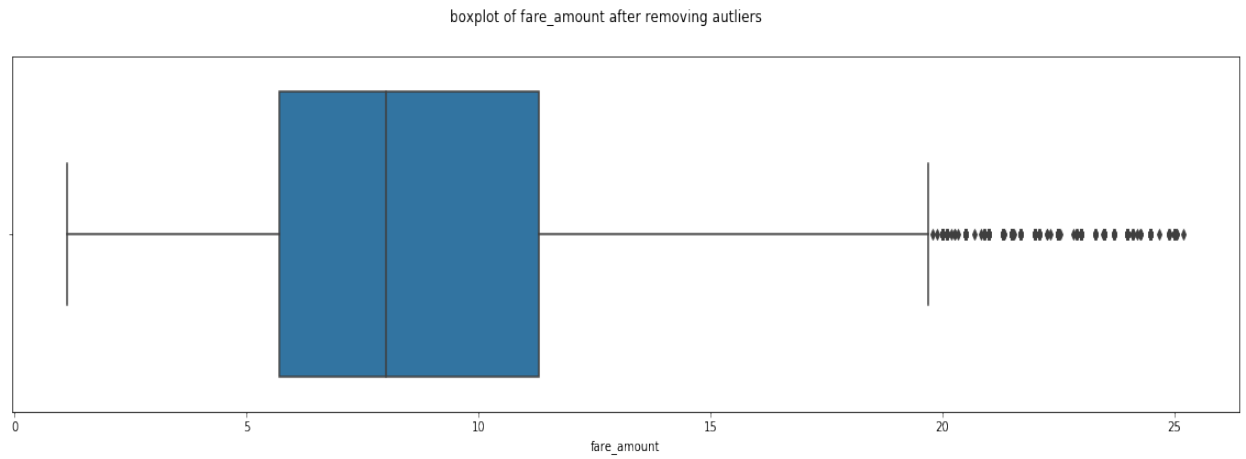
Fig 2.8 box plot for passenger_count vs fare_amount



Above figure 2.8 shows the box plot or bi-variate relation between fare_amount and all values of passenger count. In the figure more outliers are appearing when the passenger count is 1 and 2 and less outlier are present when the passenger count is 3, 4, 5, and 6.

So we have detected and removed these outliers, after removing outliers the box plot for the fare_amount is given below.

Fig 2.9 box plot for the target variable 'fare_amount' after removing outliers



2.1.6 Feature Engineering

Deriving new features from the existing feature is called feature engineering.

In feature engineering step we will extract the following features:

1. seasons from month variable
2. weekday/weekend from day_of_week
3. Sessions in a day from hour variable.
4. Distance from longitude and latitude variable.

1. ***Feature engineering for month variable***

   Here we have extracted seasons in a year from month variable. The python code to extract these features is given below.

```
#Extracting seasons from months
def g(x):
    if (x >=3) and (x <= 5):
        return 'spring'
    elif (x >=6) and (x <=8 ):
        return 'summer'
    elif (x >= 9) and (x <= 11):
        return'fall'
    elif (x >=12)|(x <= 2) :
        return 'winter'
```

```
train['seasons'] = train['month'].apply(g)
test['seasons'] = test['month'].apply(g)
```

2. ***Feature engineering for day_of_week variable***

   Here we have extracted weekdays and weekends from day_of_week variable. The python code is given below.

```
# classfying weekdays and weekends in day_of_week
def h(x):
    if (x >=0) and (x <= 4):
        return 'weekday'
    elif (x >=5) and (x <=6 ):
        return 'weekend'
```

```
train['week'] = train['day_of_week'].apply(h)
test['week'] = test['day_of_week'].apply(h)
```

3. ***Feature engineering for month variable***

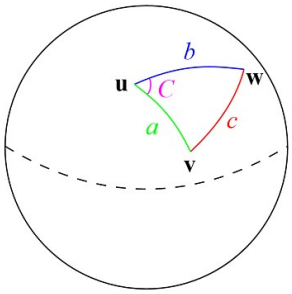
```
# Now we will derive sessions in a day from hour
# Defining a function to extract sessions in a day from hour
def f(x):
    if (x >=5) and (x <= 11):
        return 'morning'
    elif (x >=12) and (x <=16 ):
        return 'afternoon'
    elif (x >= 17) and (x <= 20):
        return'evening'
    elif (x >=21) and (x <= 23) :
        return 'night_PM'
    elif (x >=0) and (x <=4):
        return'night_AM'
```

```
# calling function and applying on train and test data
train['session'] = train['hour'].apply(f)
test['session'] = test['hour'].apply(f)
```

4. ***Feature engineering for longitude and latitude variable.***

We have given pickup_longitude, pickup_latitude, dropoff_longitude and dropoff_latitude. So to calculate the fare amount we have to calculate the distance from longitude and latitude variable using haversine formula.

The haversine formula determine the great-circle distance between two points on a sphere given longitudes and latitudes. Important in navigation, it is a special case of a more general formula in spherical trigonometry, the law of haversines that relates the sides and angles of spherical triangles.



The formula of haversine is given by,

Haversine formula: $a = \sin^2(\Delta\phi/2) + \cos\phi_1 \cdot \cos\phi_2 \cdot \sin^2(\Delta\lambda/2)$

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

$d = R \cdot c$

Where, ϕ is latitude, λ is longitude, R is earth's radius (mean radius = 6,371km) and d is the distance.

Python code to calculate great-circle distance is given below.

```
from math import radians, cos, sin,atan2, asin, sqrt

def haversine(a):
    lon1=a[0]
    lat1=a[1]
    lon2=a[2]
    lat2=a[3]
    """
    Calculate the great circle distance between two points
    on the earth (specified in decimal degrees)
    """
    # convert decimal degrees to radians
    lon1, lat1, lon2, lat2 = map(radians, [lon1, lat1, lon2, lat2])

    # haversine formula
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2
    c =  2 * atan2(sqrt(a), sqrt(1-a))
    # Radius of earth in kilometers is 6371
    km = 6371* c
    return km
```

```
train['distance'] = train[['pickup_longitude','pickup_latitude','dropoff_longitude','dropoff_latitude']].apply(haversine,axis=1)
test['distance'] = test[['pickup_longitude','pickup_latitude','dropoff_longitude','dropoff_latitude']].apply(haversine,axis=1)
```

After feature engineering the variables that we have are

```
train.columns
```

```
Index(['pickup_datetime', 'year', 'month', 'day_of_week', 'hour',
       'fare_amount', 'pickup_longitude', 'pickup_latitude',
       'dropoff_longitude', 'dropoff_latitude', 'passenger_count', 'distance',
       'session', 'seasons', 'week'],
      dtype='object')
```

We will remove those observations which we have used to extract the new features.

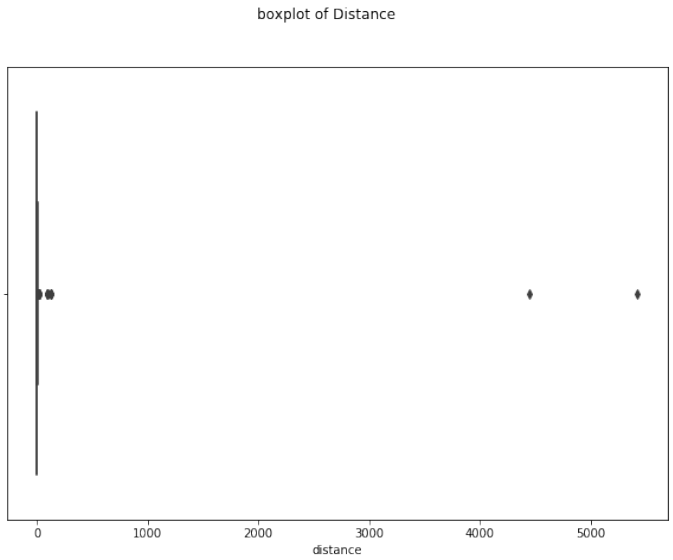
We have removed pickup_datetime, hour, pickup_longitude, pickup_latitude, dropoff_longitude, dropoff_latitude, day_of_week, and month.

After removing these variables we have 5 categorical variables and 2 continuous variables where year, passenger_count, session, seasons and week are categorical variables and fare_amount (target variable) and distance are the continuous variables.

Now we will check for the outliers in distance variable.

```
# boxplot for distance variable
plt.figure(figsize = (10, 7))
sns.boxplot(train['distance'], orient = 'h')
plt.suptitle('boxplot of Distance')
```

Fig 2.10 box plot for distance variable



In the above plot 2.10 we can see that there are some outliers are present in distance variable, so we will detect and remove these outliers.

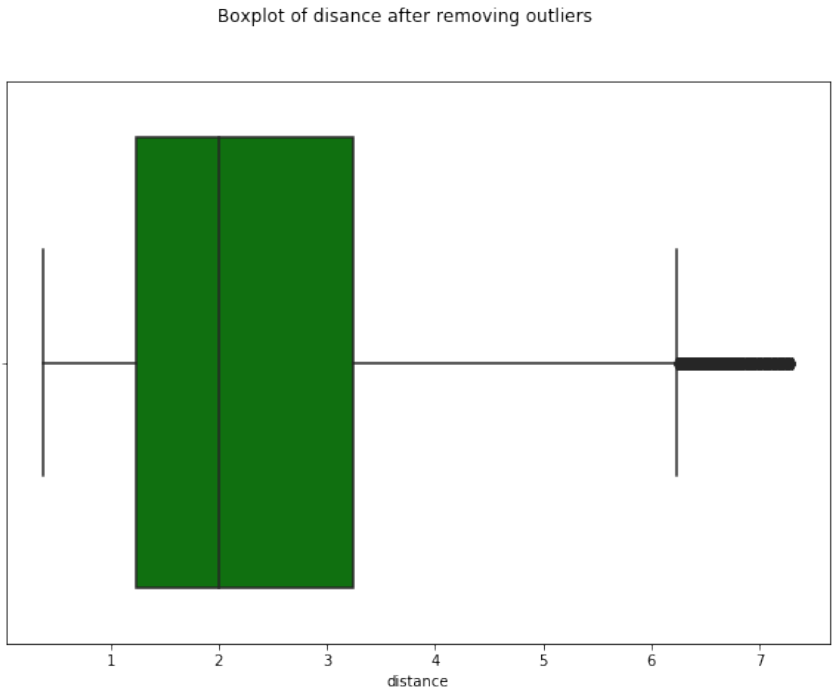
```
# Now we will detect and remove these variables
q75, q25 = np.percentile(train['distance'], [75,25])
iqr = q75-q25
min = iqr - (q25*1.5)
max = iqr + (q75*1.5)
print(min)
print(max)
```

```
0.37747859561037167
7.307643492108459
```

```
# Now we will remove these observations which are greater than upper fence and less than lower fence
train = train.drop(train[train['distance']< min].index)
train = train.drop(train[train['distance']> max].index)
```

After removing outliers in distance variable the box plot is

Fig 2.10 box plot for distance variable after removing outliers



2.1.6 Feature selection

Selecting a subset of relevant and meaningful features from the dataset which contributes much information to develop a model is called feature selection. The agenda of

feature selection is to identify and remove a needed or irrelevant attribute from the data that do not contribute much information.

There are different methods to reduce the dimensions of data or based on which we can select or drop variables.

1. Correlation analysis (For numerical continuous variable)
2. Chi-square test of independence (For categorical variable)
3. Analysis Of Variance (ANOVA) test for categorical variable
4. VIF (variance Inflation Factor) to test the multico-linearity for all the variables.

We have already removed those variables which we have used for feature engineering.

1. **Correlation analysis (For numerical continuous variable)**

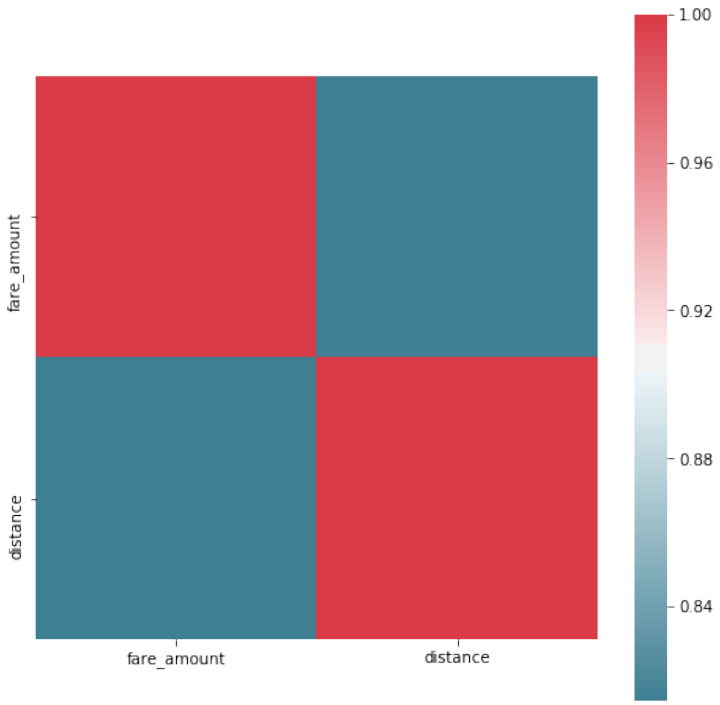
   Correlation analysis requires only numerical variables. Therefore we will filter out numerical variables and feed it to the correlation analysis to calculate the correlation matrix and using that correlation matrix we will plot the heat map to check the correlation between continuous variable. The correlation between independent variables should zero or less and the correlation between independent variable and dependent variable should be high.

Python code for Correletion plot

```
# Now we will check the corrilation between numerical variables through heatmape using correlation matrix
f, ax = plt.subplots(figsize = (8,8))
#Generate correlation matrix
corr_mat = train.corr()
#plot heatmmape using seaborn library
sns.heatmap(corr_mat, mask=np.zeros_like(corr_mat, dtype=np.bool), cmap=sns.diverging_palette(220, 10, as_cmap=True),square=True,
```

Below graph shows the correlation plot of all numeric variable present in the dataset.

Fig 2.11 heat map for numerical variables



In the above heat map we can see that the correlation between distance and fare_amount is 0.84 that means there is a high correlation between distance and fare amount where distance is the independent variable and fare_amount is the dependent or target variable s we will keep distance variable in the dataset to predict the fare amount.

2. Chi-square test of independence (For categorical variable)

The Chi-Square test of independence is used to determine if there is a significant relationship between two nominal (categorical) variables. The frequency of each category for one nominal variable is compared across the categories of the second nominal variable. The data can be displayed in a contingency table where each row represents a category for one variable and each column represents a category for the other variable.

In chi-square test we will test the hypothesis.

Null hypothesis H0: there is no association between 2 independent variables.

Alternative hypothesis H1: there is an association between two independent variables.

Hypothesis testing:

Hypothesis testing for the chi-square test of independence as it is for other tests like ANOVA, where a test statistic is computed and compared to a critical value. The critical value for the chi-square statistic is determined by the level of significance (typically 0 .05) and the degrees of freedom. The degrees of freedom for the chi-square are calculated using the following formula: *df* = (r-1)(c-1) where r is the number of rows and c is the number of columns. If the observed chi-square test statistic is less than the critical value (0.05), the null hypothesis can be rejected and if the test statistic is greater than 0.05 we will accept the null hypothesis.

```
# Applying chi_square test
for i in cat_var:
    for j in cat_var:
        if(i != j):
            chi2, p, dof, ex = chi2_contingency(pd.crosstab(train[i], train[j]))
            if(p < 0.05):
                print(i,"and",j,"are dependent on each other with P value",p,'=  remove')
            else:
                print(i,"and",j,"are independent on each other with P value",p,'= Keep')
```

```
year and seasons are dependent on each other with P value 8.421232210580892e-93 =  remove
year and passenger_count are dependent on each other with P value 3.2996360985544357e-28 =  remove
year and session are independent on each other with P value 0.8445881225252707 = Keep
year and week are dependent on each other with P value 0.023062809124273123 =  remove
seasons and year are dependent on each other with P value 8.42123221058137e-93 =  remove
seasons and passenger_count are independent on each other with P value 0.06353776808063871 = Keep
seasons and session are independent on each other with P value 0.09686602798889939 = Keep
seasons and week are independent on each other with P value 0.12727362700748787 = Keep
passenger_count and year are dependent on each other with P value 3.2996360985544357e-28 =  remove
passenger_count and seasons are independent on each other with P value 0.06353776808063871 = Keep
passenger_count and session are dependent on each other with P value 6.277472327698881e-23 =  remove
passenger_count and week are dependent on each other with P value 1.8142112391333172e-20 =  remove
session and year are independent on each other with P value 0.8445881225252707 = Keep
session and seasons are independent on each other with P value 0.09686602798889939 = Keep
session and passenger_count are dependent on each other with P value 6.277472327698792e-23 =  remove
session and week are dependent on each other with P value 1.015460107623779e-120 =  remove
week and year are dependent on each other with P value 0.023062809124273106 =  remove
week and seasons are independent on each other with P value 0.12727362700748787 = Keep
week and passenger_count are dependent on each other with P value 1.8142112391333037e-20 =  remove
d770717a55e14435ea96371f7f746da7a...nt on each other with P value 1.015460107623779e-120 =  remove
```

1. Analysis Of Variance (ANOVA) test for categorical variable

Analysis of variance (ANOVA) is a statistical technique that is used to check if the means of two or more groups are significantly different from each other. ANOVA checks the impact of one or more factors by comparing the means of different samples.

Null hypothesis H0: means of each category in a variable are same.

Alternative Hypothesis H1: mean of at least one category in a variable

If P value is less than 0.05 we will reject the null hypothesis and if P 0.05 we will accept the null hypothesis.

```
# ANOVA test for the catagorical variable
model = ols('fare_amount ~ C(year)+C(seasons)+C(passenger_count)+C(session)+C(week)',data=train).fit()
ANOVA_table = sm.stats.anova_lm(model)
ANOVA_table
```

	df	sum_sq	mean_sq	F	PR(>F)
C(year)	6.0	7945.859384	1324.309897	85.265106	2.714415e-105
C(seasons)	3.0	545.304474	181.768158	11.703062	1.180187e-07
C(passenger_count)	5.0	219.336386	43.867277	2.824375	1.488847e-02
C(session)	4.0	646.888526	161.722131	10.412408	2.029541e-08
C(week)	1.0	22.352351	22.352351	1.439146	2.302984e-01
Residual	13604.0	211292.903235	15.531675	NaN	NaN

In the above table we can see that p values of all the variables are less than 0.05 so we will reject the null hypothesis.

2. **VIF (variance Inflation Factor) to test the multico-linearity for all the variables**

   A variance inflation factor (VIF) detects multicollinearity in regression analysis. Multicollinearity is when there's correlation between predictors (i.e. independent variables) in a model; it's presence can adversely affect your regression results. The VIF estimates how much the variance of a regression coefficient is inflated due to multicollinearity in the model.

   - VIF is always greater than or equal to 1.
   - If VIF of a variable is 1 then the variable is not correlated to any of the other variable.
   - If VIF is between1-5, then the variable is moderately correlated with other variables.
   - If VIF is above 5 then there is a high correlation between the variables.
   - If there are multiple variables with VIF greater than 5, only remove the variable with the highest VIF.
   - If VIF goes above 10, we can assume that the regression coefficients are poorly estimated due to multicollinearity.

The formula to calculate VIF is given by,

$$\mathrm{VIF} = \frac{1}{1 - R_i^2}$$

Where R^2 is the R-squared value.

The VIF table for each independent variable is given below.

	VIF	features
0	3425.186988	Intercept
1	1.688485	year[T.1]
2	1.690192	year[T.2]
3	1.706162	year[T.3]
4	1.710537	year[T.4]
5	1.665170	year[T.5]
6	1.405393	year[T.6]
7	706.690287	passenger_count[T.1]
8	421.205388	passenger_count[T.2]
9	142.481441	passenger_count[T.3]
10	70.105012	passenger_count[T.4]
11	208.286743	passenger_count[T.5]
12	66.492515	passenger_count[T.6]
13	1.541765	session[T.1]
14	1.568048	session[T.2]
15	1.353245	session[T.3]
16	1.432340	session[T.4]
17	1.662167	seasons[T.1]
18	1.568781	seasons[T.2]
19	1.609899	seasons[T.3]

20	1.051300	week[T.1]
21	1.020764	distance

We have calculated VIF for all the variables in the dataset and VIF's of all the variables except passenger count are below 5.

2.1.7 Feature scaling

Feature scaling is the method to limit the range of variable so they can be compared on the ground and it is perform only on the variable.

To scale the features there are two methods,

1. Normalization
2. Standardization (zee score)

1.Normalization- Normalization is the process of reducing unwanted variation either within or between variable(i.e it converts all the data points of each variable between 0 to 1).

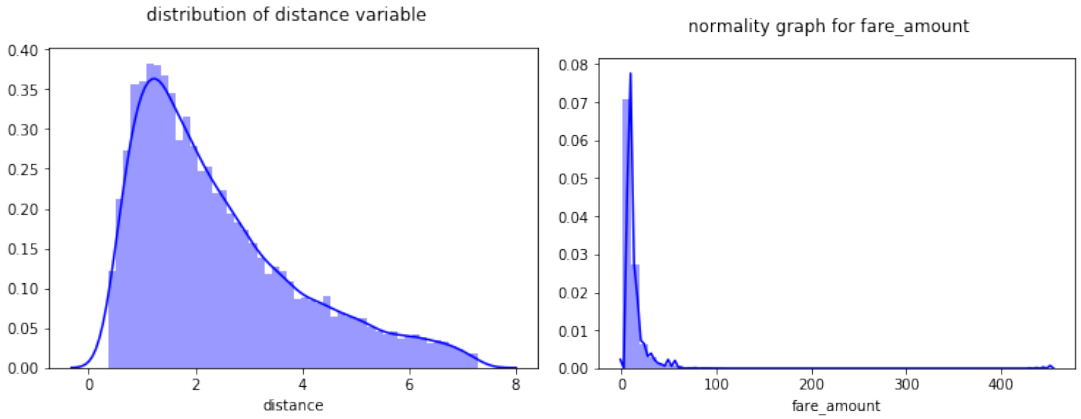
2.Standardization- It transforms the data set to have a zero mean and unique variance.

Linear model assumes that the data we are feeding are related in a linear fashion or can be measured with a linear distance metric.

Our independent variable **distance** is not normally distributed so we will choose normalization over standardization. We have checked the variance for each column in the dataset before normalization. High variance will affect the accuracy of the model, so we need to normalize that variance.

Below graphs shows the distribution of **distance** and **fare_amount** before normalization.

Fig 2.11 distribution of distance and fare_amount




```
# skewness and kurtosis of the of distance
skewness = train['distance'].skew()
kurtosis = train['distance'].kurt()
print('skewness =',skewness,'kurtosis=', kurtosis)
```

```
skewness = 1.0499080421081244 kurtosis= 0.440637353243138
```

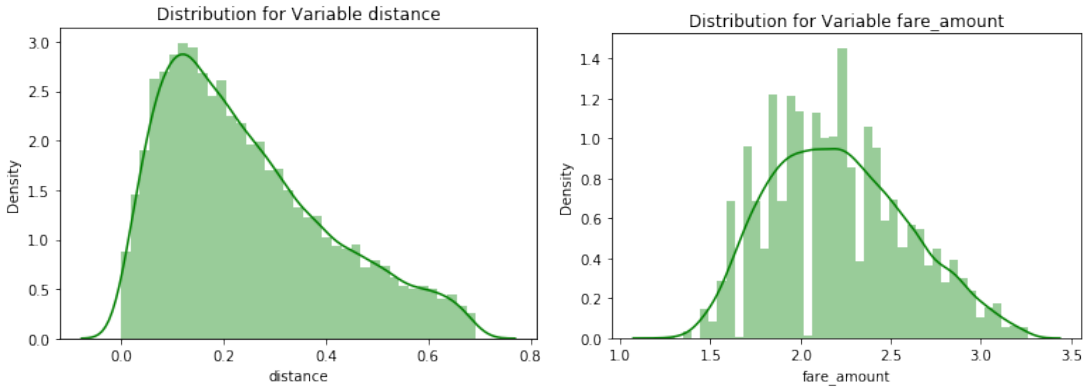
```
#skewness and kurtosis of fare amount variable
skewness = train['fare_amount'].skew()
kurtosis = train['fare_amount'].kurt()
print('skewness =',skewness,'kurtosis=', kurtosis)
```

```
skewness = 1.15666582445666 kurtosis= 1.1759286919168832
```

In the above plot we can see that both distance and fare_amount are not normally distributed since their skewness are greater than 0 and not symmetrically distrbuted also.

After normalization the graph of distance and fare_amount is

Fig 2.12 distribution of distance and fare_amount after normalization



Skewness and kurtosis after normalization.

```
skewness = train['fare_amount'].skew()
print(skewness)
```

```
0.3059395218349297
```

```
skewness = train['distance'].skew()
print(skewness)
```

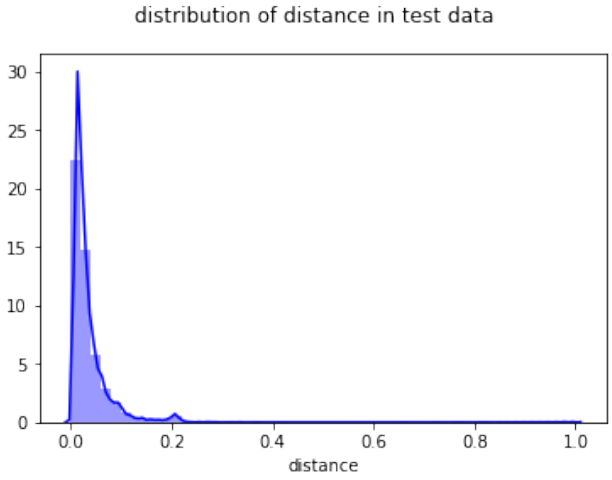
```
0.7461361965361342
```

Here we can see that the skewness is closed to zero. Now these numerical variables are symmetrically distributed.

Now we will check the distribution of distance variable in test data

The distribution of distance variable in test data is given in below graph.

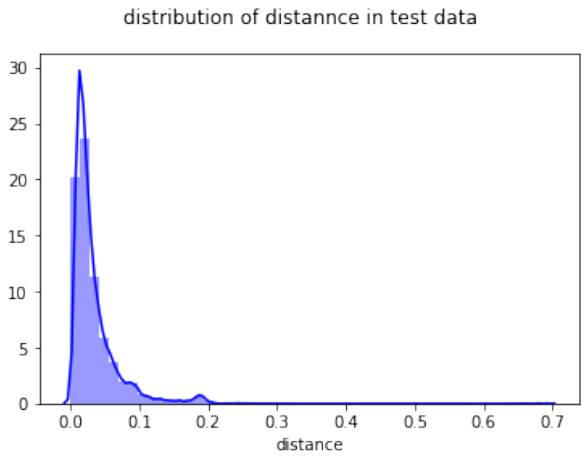
Fig 2.13 distribution of distance in test data



Here in the above plot we can see that the distance variable in test data is also not symmetrically distributed so we will normalize distance of test data to reduce its skewness.

After normalizing the distribution graph of distance in test data is given below.

Fig 2.14 distribution of distance in test data after normalization



Chapter -3

MODELING

Our problem statement is to predict the fare_amount fare amount for a cab ride in the city and here our target variable is a continuous variable. For continuous variable we can go for different Regression models. And the model having less error rate and high accuracy will be the best model for the given dataset.
Using following 4 models we will predict the fare_amount.

- Linear Regression
- Decision tree Regressor(C50)
- Random Forest
- Gradient boosting

Before modeling we have divided the data into train and test data using Simple Random Sampling (SRS). Where train data contains 75% of the dataset and test data contains 25% data of the dataset.

3.1 Multiple Linear Regression

Multiple linear regression (MLR is a statistical technique that uses several explanatory variables to predict the outcome of a response variable. The goal of multiple linear regression (MLR) is to model the linear relationship between the explanatory (independent) variables and response (dependent) variable.
In essence, multiple regression is the extension of ordinary least-squares (OLS) regression that involves more than one explanatory variable.

The equation of multiple regression is

$$Y_i=\beta_0+\beta_1 x_{i1}+\beta_2 x_{i2}+...+\beta_p x_{ip}+\epsilon$$

Y_i =dependent variable

x_i= independent variables

β_0=y-intercept (constant term)

βp = slope coefficients for each explanatory variable

ϵ=the model's error term (also known as the residuals)

Python code for multiple regression is

```
################################### Multiple Linear Regression ##########################################
fit_LR = LinearRegression().fit(X_train , y_train)
```

```
pred_train_LR = fit_LR.predict(X_train)
```

```
#prediction on test data
pred_test_LR = fit_LR.predict(X_test)
```

```
#calculate R^2 for train data
from sklearn.metrics import r2_score
r2_score(y_train, pred_train_LR)
```

```
0.7253814548161668
```

```
##calculating RMSE for test data
RMSE_test_LR = np.sqrt(mean_squared_error(y_test, pred_test_LR))

##calculating RMSE for train data
RMSE_train_LR= np.sqrt(mean_squared_error(y_train, pred_train_LR))
```

```
print("Root Mean Squared Error For Training data = "+str(RMSE_train_LR))
print("Root Mean Squared Error For Test data = "+str(RMSE_test_LR))
```

```
Root Mean Squared Error For Training data = 0.1998078290041233
Root Mean Squared Error For Test data = 0.20607603673502253
```

3.2 Decision tree

Decision tree is a predictive model based on a branching series of Boolean tests. General motive of using Decision tree is to create a training model which can be used to predict the class or a value of target variable by learning distinct rules insert from the past data. It's an algorithm that uses the tree like graph or a model of decisions.

Under the family of decision tree there are different types of Decision tree algorithms are available, like C50, C45, CART, CHECK etc. Here we have used C50 algorithm which works on information gain.

Information gain: it is the statistical approach which helps to select the node and develop a tree like structure to make any conclusion or decision on the data. It represents the expected amount of information that would be needed (i.e out of all the independent variable as a parent node who is contributing much information of all).

Python code for Decision tree is regression

```
############################## Decision Tree ############################3
fit_DT = DecisionTreeRegressor(max_depth = 2).fit(X_train,y_train)
```

```
#prediction on train data
pred_train_DT = fit_DT.predict(X_train)

#prediction on test data
pred_test_DT = fit_DT.predict(X_test)
```

```
## R^2 calculation for train data
r2_score(y_train, pred_train_DT)
```

```
0.6524348301376425
```

```
## R^2 calculation for test data
r2_score(y_test, pred_test_DT)
```

```
0.6229136593210711
```

```
##calculating RMSE for train data
RMSE_train_DT = np.sqrt(mean_squared_error(y_train, pred_train_DT))

##calculating RMSE for test data
RMSE_test_DT = np.sqrt(mean_squared_error(y_test, pred_test_DT))
```

```
print("Root Mean Squared Error For Training data = "+str(RMSE_train_DT))
print("Root Mean Squared Error For Test data = "+str(RMSE_test_DT))
```

```
Root Mean Squared Error For Training data = 0.22475852867550794
Root Mean Squared Error For Test data = 0.23278193107733608
```

3.3 Random Forest

Random forest is an ensemble that consists of many decision trees. To reduce the error and improve the accuracy we will combine the multiple decision trees to prepare a strong classifier or regressor.

Python code for Random Forest

```
######################################## Random Forest ################################################
fit_RF = RandomForestRegressor(n_estimators = 200).fit(X_train,y_train)
```

```
#prediction on train data
pred_train_RF = fit_RF.predict(X_train)
#prediction on test data
pred_test_RF = fit_RF.predict(X_test)
```

```
## calculate R^2 for train data
r2_score(y_train, pred_train_RF)
```

```
0.9599236094662843
```

```
#calculate R^2 for test data
r2_score(y_test, pred_test_RF)
```

```
0.6873742508499638
```

```
##calculating RMSE for train data
RMSE_train_RF = np.sqrt(mean_squared_error(y_train, pred_train_RF))
##calculating RMSE for test data
RMSE_test_RF = np.sqrt(mean_squared_error(y_test, pred_test_RF))
print("Root Mean Squared Error For Training data = "+str(RMSE_train_RF))
print("Root Mean Squared Error For Test data = "+str(RMSE_test_RF))
```

```
Root Mean Squared Error For Training data = 0.0763207019695304
Root Mean Squared Error For Test data = 0.2119538154225453
```

3.3 Gradient Boosting

Gradient boosting is a machine learning technique for regression and classification problems which produces a prediction model in the form of an ensemble of weak prediction models, typically decision trees. It builds the model in a stage-wise fashion like other boosting methods do, and it generalize them by allowing optimization of an arbitrary differentiable loss function.

Python code for gradient boosting

```
######################################  Gradient boosting ################################################
fit_GB = GradientBoostingRegressor().fit(X_train, y_train)
```

```
#prediction on train data
pred_train_GB = fit_GB.predict(X_train)

#prediction on test data
pred_test_GB = fit_GB.predict(X_test)
```

```
#calculate R^2 for test data
r2_score(y_test, pred_test_GB)
```

```
0.7253435498282337
```

```
#calculate R^2 for train data
r2_score(y_train, pred_train_GB)
```

```
0.759093799456726
```

```
##calculating RMSE for train data
RMSE_train_GB = np.sqrt(mean_squared_error(y_train, pred_train_GB))
##calculating RMSE for test data
RMSE_test_GB = np.sqrt(mean_squared_error(y_test, pred_test_GB))
print("Root Mean Squared Error For Training data = "+str(RMSE_train_GB))
print("Root Mean Squared Error For Test data = "+str(RMSE_test_GB))
```

```
Root Mean Squared Error For Training data = 0.18712079126285588
Root Mean Squared Error For Test data = 0.19866610052497302
```

Chapter -4

PARAMETER TUNING

4.1 Hyper Parameter Tuning For Optimizing The Result

Model hyper parameters are set by the data scientist ahead of training and control implementation aspects of the model. The weights learned during training of a linear rregression model are parameters while the number of trees in a random forest is a model hyper parameter because this is set by the data scientist. Hyper-parameters can be thought of as model settings. These settings need to be tuned for each problem because the best hyperparameters for one particular dataset will not be the best across all datasets.

The process of hyper parameter tuning (also called hyper parameter optimization) means finding the combination of hyperparameter values for a machine learning model that performs the best as measured o a validation dataset for a problem.

Here we have used two hyper parameter tuning technique.

- Random Search CV
- Grid Search CV

1. **Random Search CV:** This algorithm setup a grid of hyper parameter values and select random combinations to train the model and score. The number of each search iterations is set based on time/resources.
2. **Grid Search CV:** This algorithm setup a hyper parameter values and for each combination, train a model and score on the validation data. In this approach every single combination of hyper parameters values is tried which can be very efficient.

4.1.1 Check results after Using Random Search CV on Random Forest and gradient boosting model.

- **Random search CV on Random Forest**

```
#Random Search CV on Random Forest Model
RRF = RandomForestRegressor(random_state = 0)
n_estimator = list(range(1,20,2))
depth = list(range(1,100,2))

# Create the random grid
rand_grid = {'n_estimators': n_estimator,
               'max_depth': depth}

randomcv_rf = RandomizedSearchCV(RRF, param_distributions = rand_grid, n_iter = 5, cv = 5, random_state=0)
randomcv_rf = randomcv_rf.fit(X_train,y_train)
predictions_RRF = randomcv_rf.predict(X_test)

view_best_params_RRF = randomcv_rf.best_params_

best_model = randomcv_rf.best_estimator_

predictions_RRF = best_model.predict(X_test)

# Calculating R-squared value
RRF_r2 = r2_score(y_test, predictions_RRF)
#Calculating RMSE
RRF_rmse = np.sqrt(mean_squared_error(y_test,predictions_RRF))

print('Random Search CV Random Forest Regressor Model Performance:')
print('Best Parameters = ',view_best_params_RRF)
print('R-squared = {:0.2}.'.format(RRF_r2))
print('RMSE = ',RRF_rmse)
```

```
Random Search CV Random Forest Regressor Model Performance:
Best Parameters =  {'n_estimators': 15, 'max_depth': 9}
R-squared = 0.71.
RMSE =  0.20255973463860888
```

- **Random search CV on Gradient Boosting model**

```
##Random Search CV on gradient boosting model

gb = GradientBoostingRegressor(random_state = 0)
n_estimator = list(range(1,20,2))
depth = list(range(1,100,2))

# Create the random grid
rand_grid = {'n_estimators': n_estimator,
               'max_depth': depth}

randomcv_gb = RandomizedSearchCV(gb, param_distributions = rand_grid, n_iter = 5, cv = 5, random_state=0)
randomcv_gb = randomcv_gb.fit(X_train,y_train)
predictions_gb = randomcv_gb.predict(X_test)

view_best_params_gb = randomcv_gb.best_params_

best_model = randomcv_gb.best_estimator_

predictions_gb = best_model.predict(X_test)

#calculating R square value
gb_r2 = r2_score(y_test, predictions_gb)
#Calculating RMSE
gb_rmse = np.sqrt(mean_squared_error(y_test,predictions_gb))

print('Random Search CV Gradient Boosting Model Performance:')
print('Best Parameters = ',view_best_params_gb)
print('R-squared = {:0.2}.'.format(gb_r2))
print('RMSE = ', gb_rmse)
```

```
Random Search CV Gradient Boosting Model Performance:
Best Parameters =  {'n_estimators': 15, 'max_depth': 9}
R-squared = 0.68.
RMSE =  0.21415215258104084
```

4.1.2 Check result after applying Grid Search CV on Random Forest and Gradient boosting model

- **Grid Search CV On Random Forest Model**

```
## Grid Search CV for random Forest model
regr = RandomForestRegressor(random_state = 0)
n_estimator = list(range(11,20,1))
depth = list(range(5,15,2))

# Create the grid
grid_search = {'n_estimators': n_estimator,
               'max_depth': depth}

## Grid Search Cross-Validation with 5 fold CV
gridcv_rf = GridSearchCV(regr, param_grid = grid_search, cv = 5)
gridcv_rf = gridcv_rf.fit(X_train,y_train)
view_best_params_GRF = gridcv_rf.best_params_

#Apply model on test data
predictions_GRF = gridcv_rf.predict(X_test)

#R^2
GRF_r2 = r2_score(y_test, predictions_GRF)
#Calculating RMSE
GRF_rmse = np.sqrt(mean_squared_error(y_test,predictions_GRF))

print('Grid Search CV Random Forest Regressor Model Performance:')
print('Best Parameters = ',view_best_params_GRF)
print('R-squared = {:0.2}.'.format(GRF_r2))
print('RMSE = ',(GRF_rmse))
```

```
Grid Search CV Random Forest Regressor Model Performance:
Best Parameters =  {'max_depth': 7, 'n_estimators': 19}
R-squared = 0.72.
RMSE =  0.20086427687097538
```

- **Grid Search CV On Gradient Boosting Model**

```
#Grid Search CV on Gradient boosting model
gb = GradientBoostingRegressor(random_state = 0)
n_estimator = list(range(11,20,1))
depth = list(range(5,15,2))

# Create the grid
grid_search = {'n_estimators': n_estimator,
               'max_depth': depth}

## Grid Search Cross-Validation with 5 fold CV
gridcv_gb = GridSearchCV(gb, param_grid = grid_search, cv = 5)
gridcv_gb = gridcv_gb.fit(X_train,y_train)
view_best_params_Ggb = gridcv_gb.best_params_

#Apply model on test data
predictions_Ggb = gridcv_gb.predict(X_test)

#R^2
Ggb_r2 = r2_score(y_test, predictions_Ggb)
#Calculating RMSE
Ggb_rmse = np.sqrt(mean_squared_error(y_test,predictions_Ggb))

print('Grid Search CV Gradient Boosting regression Model Performance:')
print('Best Parameters = ',view_best_params_Ggb)
print('R-squared = {:0.2}.'.format(Ggb_r2))
print('RMSE = ',(Ggb_rmse))
```

```
Grid Search CV Gradient Boosting regression Model Performance:
Best Parameters =  {'max_depth': 5, 'n_estimators': 19}
R-squared = 0.7.
RMSE =  0.2059846733687027
```

Chapter -5

CONCLUSION

5.1 Model Evaluation

We have developed few models to predict the target variable fare_amount. Now we have to choose the suitable model which is providing best result for the dataset based on predictive performance.

Predictive performance can be measured by comparing predicted values of the models with the actual values of the target variable and calculating some average error measures.

Using following error metrics we will calculate the error rate and then we will select the appropriate model.

5.1.1 RMSE

Root Mean Square Error is a frequently used measure of the difference between values predicted by a model and the values actually observed from the environment that is being modeled. RMSE square the errors, finds their average and take their square root.

Formula of RMSE is

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n} (X_{obs,i} - X_{model,i})^2}{n}}$$

5.1.2 R-Squared (R^2)

R-squared is a statistical measure of how close the data are to the fitted regression line. It is also known as the coefficient of determination or the coefficients of multiple determinations for multiple regression.

In other words R squared value explains how much of the variance of the target variable is explained by target variable.

We have shown the results of both train and test data in a table to check whether our data is over fitted or not.

Table 5.1: Error metric table

MODEL	RMSE		R-Squared	
	Train	Test	Train	Test
Linear Regression	0.199	0.206	0.72	0.70
Decision Tree	0.224	0.232	0.65	0.62
Random Forest	0.076	0.211	0.95	0.68
Gradient Boosting	0.187	0.198	0.75	0.72

From the above table we can see that less RMSE and high R-squared found in Random Forest model.

After performing hyper parameter tuning, the results of the model are given in the below table.

Table 5.2: Error metric table after parameter tuning

Model	Parameter	RMSE(Test)	R-Squared (Test)
Random Search CV	Random Forest	0.202	0.71
	Gradient Boosting	0.214	0.68
Grid search CV	Random Forest	0.200	0.72
	Gradient Boosting	0.205	0.70

Above table 5.2 shows the results after tuning the parameters of the best two best suited models i.e Random Forest and Gradient Boosting. For tuning the parameters we have used

Random search CV and Grid Search CV under which we have given the range of n estimators, depth and CV folds.

5.2 Model Selection

On the basis of RMSE and R squared results a good model should have least RMSE and max R-Squared value.

On the above table we have concluded that

- Both the models i.e Random Forest and Gradient Boosting perform comparatively well while comparing their RMSE and R-squared values.
- After optimizing the parameters Random forest gives the best results as compared to Gradient boosting.

After looking at all these observations we can say that Random forest model is the best model to predict the fare_amount for this model with highest explained variance of the target variable and lowest error chances with parameter tuning technique Grid Search CV.

Finally we have used Random forest model with Grid search CV parameter tuning to predict the target variable for the test data given in the problem statement.