

CAB FARE PREDICTION

Submitted by: Vanusha Baregal

12/23/19



Contents

S.No.	Contents	Page no.
1	Introduction	
	1.1 Problem Statement	2
	1.2 Data	2
2	Methodology	
	2.1 Pre-Processing	4
	2.1.1 Data Cleaning and Missing Value Analysis	7
	2.1.2 Feature Engineering	14
	2.1.3 Outlier Analysis	19
	2.1.4 Data Visualization	21
	2.1.5 Feature Selection	25
	2.1.6 Feature Scaling	28
3	Modelling	
	Linear Regression	34
	Decision Tree	35
	Random Forest	37
	Gradient Boosting/XG Boost	38/41
	Hyper Parameters Tuning for optimising the results (Grid Search CV & Random Search CV)	39
4	Conclusion	
	4.1 Model Evaluation	44
	4.2 Model Selection	46
References		
Appendix- R and Python Code		

Chapter 1

Introduction

Background

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. The objective of this project is to predict the fare amount for the taxi ride given the pickup and drop off locations

1.1 Problem Statement

In this project, we need to predict the fare amount for the taxi ride given the pickup and drop off locations.

1.2 Data

Our task is to build a regression model which will predict the fare of the cab facility based on the customer attributes and all of the information during the journey and general information available to the company about them.

Table 1.1: Train dataset (Columns:1-7)

	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 (Columns:1-6)

	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

In the train dataset there are 16067 observations and 7 variables using which we have to predict the cab fare amount and the test data has 9914 observations and 6 variables

train contains:

- fare_amount
- 6 numerical variables, named pickup_datetime, pickup_longitude, pickup_latitude, dropoff_longitude, dropoff_latitude, passenger_count

test contains:

- 6 numerical variables, named pickup_datetime, pickup_longitude, pickup_latitude, dropoff_longitude, dropoff_latitude, passenger_count

Variables

- **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.
- **fare_amount** - an integer indicating the amount incurred for the cab ride.

Chapter 2

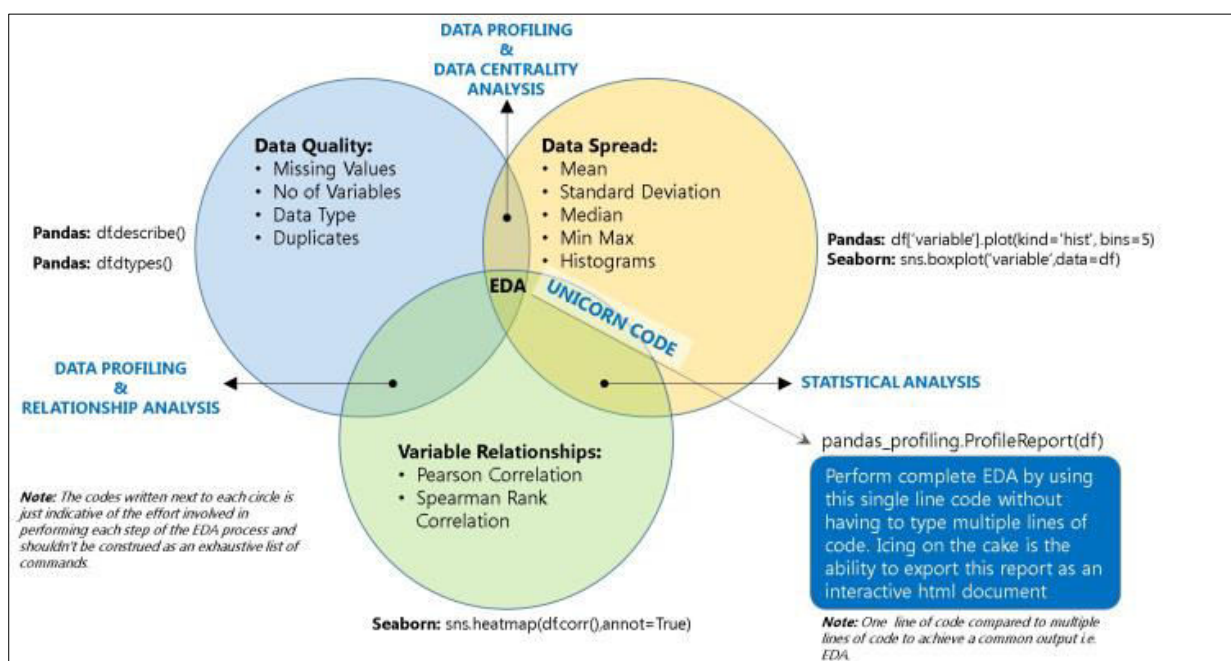
2.1 Methodology

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. Most analysis like regression, require the data to be normally distributed. We can visualize that in a glance by looking at the probability distributions or probability density functions of the variable. To start this process, we will first try pandas profiling.

Python Pandas Profiling

The pandas-profiling Python package is a great tool to create HTML profiling reports. For a given dataset, it computes the following statistics:

- Essentials: type, unique values, missing values.
- Quantile statistics like minimum value, Q1, median, Q3, maximum, range, interquartile range.
- Descriptive statistics like mean, mode, standard deviation, sum, median absolute deviation, coefficient of variation, kurtosis, skewness.
- Most frequent values.
- Histogram.
- Correlations highlighting of highly correlated variables, Spearman and Pearson matrixes.



Source: zone.com/articles/pandas-one-line-magical-code-for-eda-pandas-profil

Output of Pandas Profiling in Python

Dataset info

Number of variables	7
Number of observations	16067
Missing cells	80 (0.1%)
Duplicate rows	0 (0.0%)
Total size in memory	878.7 KiB
Average record size in memory	56.0 B


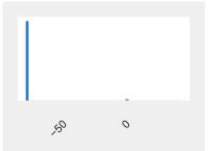

Variables types

Numeric	5
Categorical	1
Boolean	0
Date	0
URL	0
Text (Unique)	0
Rejected	1
Unsupported	0

Warnings

`dropoff_latitude` has 312 (1.9%) zeros
`dropoff_longitude` has 314 (2.0%) zeros
`fare_amount` is highly skewed ($\gamma_1 = 125.404254$)
`passenger_count` is highly skewed ($\gamma_1 = 84.62251802$)
`pickup_datetime` has a high cardinality: 16021 distinct values
`pickup_latitude` has 315 (2.0%) zeros
`pickup_longitude` is highly correlated with `dropoff_longitude` ($\rho = 0.9637700432$)

Zeros
Zeros
Skewed
Skewed
Warning
Zeros
Rejected

dropoff_latitude Numeric	Distinct count	14263	Mean	39.897906	
	Unique (%)	88.8%	Minimum	-74.006377	
	Missing (%)	0.0%	Maximum	41.366138	
	Missing (n)	0	Zeros (%)	1.9%	
	Infinite (%)	0.0%			
	Infinite (n)	0			
dropoff_longitude Numeric	Distinct count	13887	Mean	-72.46232767	
	Unique (%)	86.4%	Minimum	-74.429332	
	Missing (%)	0.0%	Maximum	40.802437	
	Missing (n)	0	Zeros (%)	2.0%	
	Infinite (%)	0.0%			
	Infinite (n)	0			
fare_amount Numeric	Distinct count	468	Mean	15.01500436	
	Unique (%)	2.9%	Minimum	-3	
	Missing (%)	0.2%	Maximum	54343	
	Missing (n)	25	Zeros (%)	< 0.1%	
	Infinite (%)	0.0%			
	Infinite (n)	0			

Statistics

Histogram

Common values

Extreme values

Quantile statistics

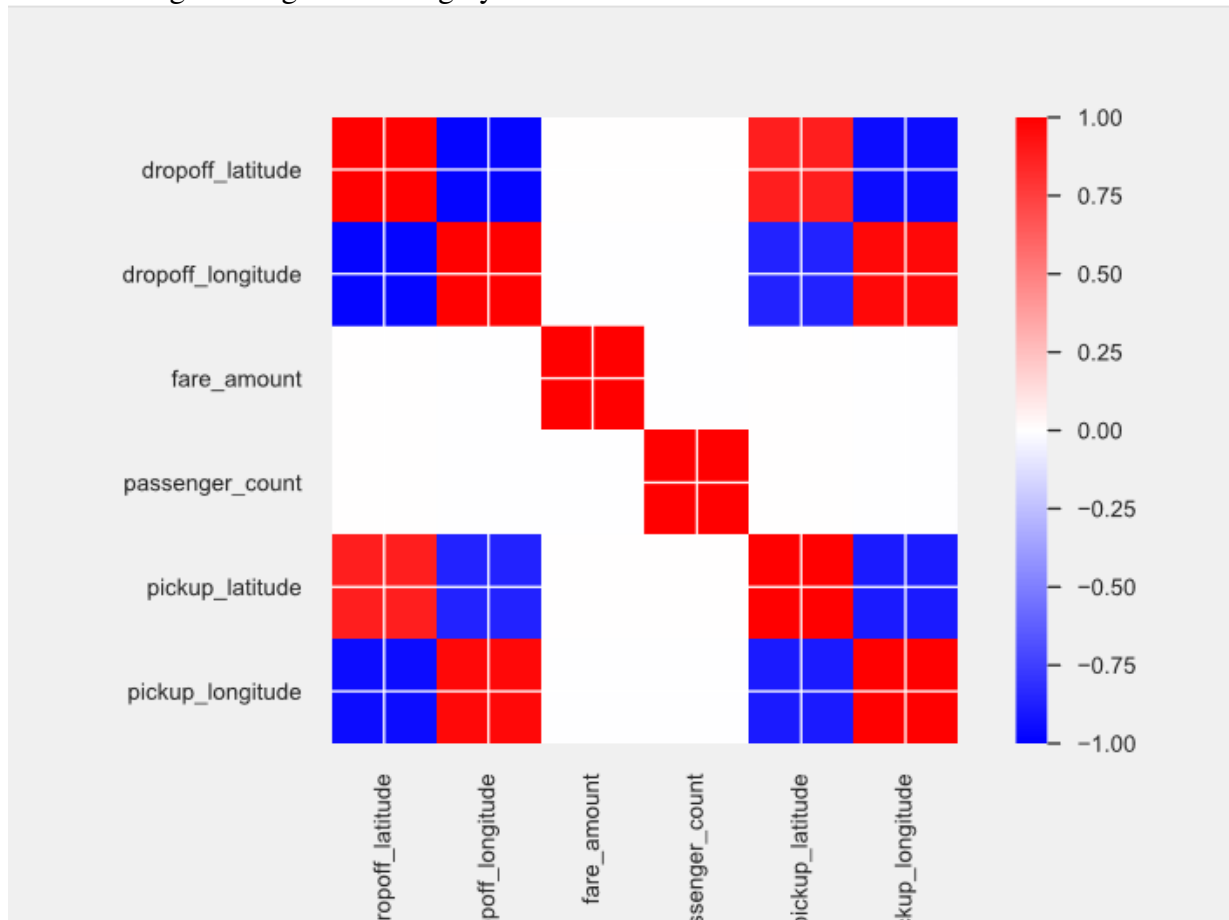
Minimum	-3
5-th percentile	4.1
Q1	6
Median	8.5
Q3	12.5
95-th percentile	30.5
Maximum	54343
Range	54346
Interquartile range	6.5

Descriptive statistics

Standard deviation	430.4609448
Coef of variation	28.66871926
Kurtosis	15820.6933
Mean	15.01500436
MAD	11.70740838
Skewness	125.404254
Sum	240870.7
Variance	185296.625
Memory size	125.6 KiB

Toggle details of target variable “Fare_amount”

Below table gives us glance on highly correlated variables



2.1 Pre Processing

EDA process begins with loading of data into the environment, getting quick look at it along with count of records and number of attributes. We will be making heavy use of pandas and numpy to perform data manipulation and related tasks. For visualisation purposes, we will use matplotlib and seaborn along with pandas visualisation capabilities wherever possible in Python and ggplots in R.

Descriptive Statistics

In Descriptive Statistics you are describing, presenting, summarizing and organizing your data (population), either through numerical calculations or graphs or tables.

Summary statistics of train dataset

	fare_amount	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
count	16042.000000	16067.000000	16067.000000	16067.000000	16067.000000	16012.000000
mean	15.015004	-72.462787	39.914725	-72.462328	39.897906	2.625070
std	430.460945	10.578384	6.826587	10.575062	6.187087	60.844122
min	-3.000000	-74.438233	-74.006893	-74.429332	-74.006377	0.000000
25%	6.000000	-73.992156	40.734927	-73.991182	40.734651	1.000000
50%	8.500000	-73.981698	40.752603	-73.980172	40.753567	1.000000
75%	12.500000	-73.966838	40.767381	-73.963643	40.768013	2.000000
max	54343.000000	40.766125	401.083332	40.802437	41.366138	5345.000000

Summary statistics of test dataset

	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
count	9914.000000	9914.000000	9914.000000	9914.000000	9914.000000
mean	-73.974722	40.751041	-73.973657	40.751743	1.671273
std	0.042774	0.033541	0.039072	0.035435	1.278747
min	-74.252193	40.573143	-74.263242	40.568973	1.000000
25%	-73.992501	40.736125	-73.991247	40.735254	1.000000
50%	-73.982326	40.753051	-73.980015	40.754065	1.000000
75%	-73.968013	40.767113	-73.964059	40.768757	2.000000
max	-72.986532	41.709555	-72.990963	41.696683	6.000000

Findings:

There are 7 variables in the dataset.

From the summary of the dataset, we can see:

1. The minimal fare_amount is negative. maximum is USD 54,343 and median is USD 8.50, As this does not seem to be realistic, I will drop the negative values from the dataset.
2. Some of the minimum and maximum longitude/latitude coordinates are way off. Range should be brought within a set boundary
3. Min passenger count is 0 and max is 208, both seem errorneus. Passenger count should not exceed 6 (even if we consider SUV)

2.1.1 Data cleaning and Missing Value Analysis

The following steps have been done in data cleaning on both train and test dataset:

- Convert fare_amount from object to numeric
- Dropping NA values in 'pickup_datetime' in train_dataset

- Converting 'pickup_datetime' from object to datetime
- Separate the Pickup_datetime column into separate field like year, month, day of the week, hour, minute.
- Maximum number of passenger count is 5345 (Summary) which is not possible. Here, we will be limiting the maximum as 6 (Considering SUV) and minimum is 1 removing variables which are (Count > 6 or =1)
- Fare_amount minimum amount is negative, which cannot be true so we will remove the observations which are (<1 and > 1000)
- Drop the rows having latitude and longitude out the range (**Latitude (-90 to 90)**
Longitude (-180 to 180))

Python Code

1. Convert fare_amount from object to numeric

```
train_cab["fare_amount"] = pd.to_numeric(train_cab["fare_amount"], errors = "coerce")
#Using errors='coerce'. It will replace all non-numeric values with NaN.
```

#2.dropping NA values in datetime column

```
train_cab.dropna(subset= ["pickup_datetime"])
```

3. Here pickup_datetime variable is in object so we need to change its data type to datetime

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

4. Separate the Pickup_datetime column into separate field like year, month, day of the week, etc

```
train_cab['year'] = train_cab['pickup_datetime'].dt.year
train_cab['Month'] = train_cab['pickup_datetime'].dt.month
train_cab['Date'] = train_cab['pickup_datetime'].dt.day
train_cab['Day'] = train_cab['pickup_datetime'].dt.dayofweek
train_cab['Hour'] = train_cab['pickup_datetime'].dt.hour
train_cab['Minute'] = train_cab['pickup_datetime'].dt.minute
```

5. We can see maximum number of passenger count is 5345 which is actually not possible. So reducing the passenger count to 6 (even if we consider the SUV)

```
train_cab = train_cab.drop(train_cab[train_cab["passenger_count"]> 6 ].index, axis=0)
```

#Also removing the values with passenger count of 0.

```
train_cab = train_cab.drop(train_cab[train_cab["passenger_count"] == 0 ].index, axis=0)
```

6. remove the row where fare amount is zero

```
train_cab = train_cab.drop(train_cab[train_cab["fare_amount"]<1].index, axis=0)
train_cab.shape
```

so we will remove the rows having fare amounting more than 1000 as considering them as outliers

```
train_cab = train_cab.drop(train_cab[train_cab["fare_amount"]> 1000 ].index, axis=0)
train_cab.shape
```

7. Drop observations outside the range of latitude and longitude

#Latitude----(-90 to 90)

#Longitude----(-180 to 180)

```
train_cab[train_cab['pickup_latitude']<-90]
train_cab[train_cab['pickup_latitude']>90]
train_cab = train_cab.drop((train_cab[train_cab['pickup_latitude']<-90]).index, axis=0)
train_cab = train_cab.drop((train_cab[train_cab['pickup_latitude']>90]).index, axis=0)
train_cab[train_cab['pickup_longitude']<-180]
train_cab[train_cab['pickup_longitude']>180]
train_cab[train_cab['dropoff_latitude']<-90]
train_cab[train_cab['dropoff_latitude']>90]
train_cab[train_cab['dropoff_longitude']<-180]
train_cab[train_cab['dropoff_longitude']>180]
```

R Code

#1. Data Manipulation

Changing the data types of variables into required data types

```
train$fare_amount = as.numeric(as.character(train$fare_amount))
train$passenger_count=round(train$passenger_count)
```

#Removing values which are not within desired range(outlier) depending upon basic understanding of dataset.

#1.Fare amount has a negative value, which doesn't make sense. A price amount cannot be -ve and also cannot be 0. So we will remove these fields.

```
train[which(train$fare_amount < 1 ),]
nrow(train[which(train$fare_amount < 1 ),])
train = train[-which(train$fare_amount < 1 ),]
```

#2.Passenger_count variable

```
for (i in seq(4,11,by=1)){
  print(paste('passenger_count above ' ,i,nrow(train[which(train$passenger_count > i ),])))
}
```

#so 20 observations of passenger_count is consistently above from 6,7,8,9,10 passenger_counts, let's check them.

```
train[which(train$passenger_count > 6 ),]
# Also we need to see if there are any passenger_count==0
train[which(train$passenger_count <1 ),]
nrow(train[which(train$passenger_count <1 ),])
```

We will remove these 58 observations and 20 observation which are above 6 value because a cab cannot hold these number of passengers.

```
train = train[-which(train$passenger_count < 1 ),]
train = train[-which(train$passenger_count > 6),]
```

```
# There's only one outlier which is in variable pickup_latitude. So we will remove it with nan.
# Also we will see if there are any values equal to 0.
nrow(train[which(train$pickup_longitude == 0 ),])
nrow(train[which(train$pickup_latitude == 0 ),])
nrow(train[which(train$dropoff_longitude == 0 ),])
nrow(train[which(train$pickup_latitude == 0 ),])
# there are values which are equal to 0. we will remove them.
train = train[-which(train$pickup_latitude > 90),]
train = train[-which(train$pickup_longitude == 0),]
train = train[-which(train$dropoff_longitude == 0),]
```

Missing Value Analysis

In statistics, **missing data**, or **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. If missing values are present in the data, then according to the percentage of missing we can either delete the variable or impute the values using mean, median and KNN imputation method.

Missing value analysis is done on both train and test dataset in this project.

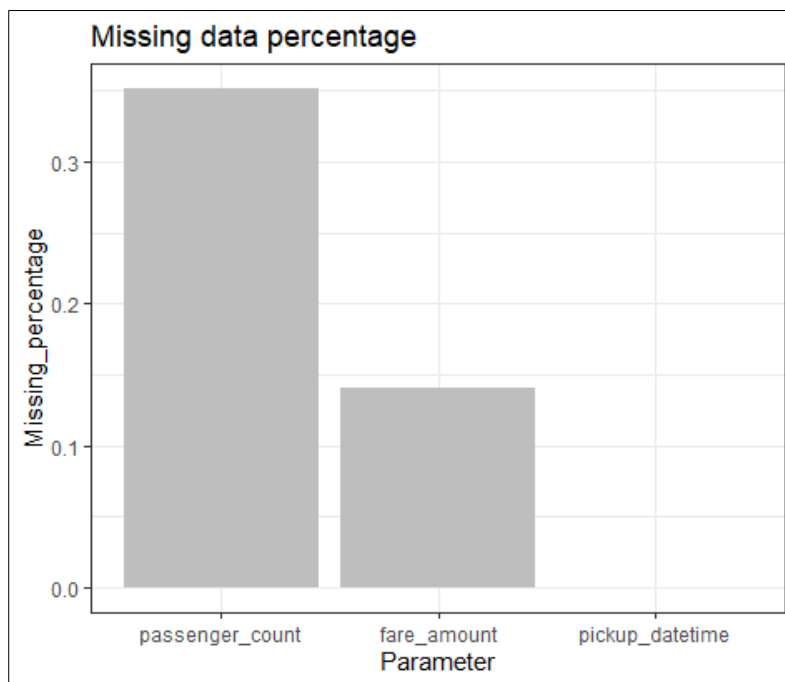
Missing values in train dataset

	index	0
0	fare_amount	24
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	Date	0
10	Day	0
11	Hour	0
12	Minute	0

Findings:

As we can see that in train dataset we find missing values in

1. passenger_count-55 (0.34%) and
2. fare_amount- 24(0.15%).



Missing values in test dataset

Index	test_dataset attributes	
0	pickup_datetime	0
1	pickup_longitude	0
2	pickup_latitude	0
3	dropoff_longitude	0
4	dropoff_latitude	0
5	passenger_count	0

Findings:

1. No missing values in test data

One of the most common problems in Data Cleaning/Exploratory Data Analysis is handling the missing values. Firstly, there is no good way to deal with missing data. But still missing value analysis helps address several concerns caused by incomplete data. If cases with missing values are systematically different from cases without missing values, the results can be misleading. Also, missing data may reduce the precision of calculated statistics because there is less information than originally planned.

Another concern is that the assumptions behind many statistical procedures are based on complete cases, and missing values can complicate the theory required. So, in our data, there

are plenty of missing values available in different variables. So, after computing the percentage of missing data that is available to us in the dataset, it accounts to around 1% of the data. So we will impute the missing data,

Imputation Using (Mean/Median) Values:

This works by calculating the mean/median of the non-missing values in a column and then replacing the missing values within each column separately and independently from the others. It can only be used with numeric data.

Imputation Using k-NN:

The k nearest neighbours is an algorithm that is used for simple classification. The algorithm uses 'feature similarity' to predict the values of any new data points. This means that the new point is assigned a value based on how closely it resembles the points in the training set. This can be very useful in making predictions about the missing values by finding the k's closest neighbours to the observation with missing data and then imputing them based on the non-missing values in the neighbourhood.

Framework to choose the best technique to impute

- Create a small subset of data with complete observations
- Delete some values manually
- Use multiple methods to fill
- See where they are failing
- Choose the best method

We impute the missing values in other columns using Median imputation in Python and KNN in R, because that fits the best after trying various other imputation techniques (Mean, Median, KNN)

Missing value analysis in R and Python is pasted below:

Python code

```
#Create dataframe with missing percentage
missing_val = pd.DataFrame(train_cab.isnull().sum())
#Reset index
missing_val = missing_val.reset_index()
missing_val
#Rename variable
missing_val = missing_val.rename(columns = {'index': 'Variables', 0: 'Missing_percentage'})
missing_val
#Calculate percentage
missing_val['Missing_percentage'] = (missing_val['Missing_percentage']/len(train_cab))*100
#descending order
missing_val = missing_val.sort_values('Missing_percentage', ascending =
False).reset_index(drop = True)
```

```

##### Imputation using mode, mean and median (categorical=mode,
numerical=mean/median)
# 1.For Passenger_count:
# Actual value = 1 Mode = 1
# Choosing a random values to replace it as NA
train_cab['passenger_count'].loc[1000]
# Replacing 1.0 with NA
train_cab['passenger_count'].loc[1000] = np.nan
train_cab['passenger_count'].loc[1000]
# Impute with mode
train_cab['passenger_count'].fillna(train_cab['passenger_count'].mode()[0]).loc[1000]
# We can't use mode method because data will be more biased towards passenger_count=1

# ##### 2.For fare_amount:

# Actual value = 7.0,
# Mean = 15.117,
# Median = 8.5,

# Choosing a random values to replace it as NA
a=train_cab['fare_amount'].loc[1000]
print('fare_amount at loc-1000:{}'.format(a))
# Replacing 1.0 with NA
train_cab['fare_amount'].loc[1000] = np.nan
print('Value after replacing with nan:{}'.format(train_cab['fare_amount'].loc[1000]))
# Impute with mean
print('Value if imputed with
mean:{}'.format(train_cab['fare_amount'].fillna(train_cab['fare_amount'].mean()).loc[1000]))
# Impute with median
print('Value if imputed with
median:{}'.format(train_cab['fare_amount'].fillna(train_cab['fare_amount'].median()).loc[1000]
))

# As we can say that the median value has given values with great approximation, we will go
ahead by using median method to impute

##### Imputation of missing values with median method

def missin_val(df):
    total = df.isnull().sum().sort_values(ascending=False)
    percent = (df.isnull().sum() * 100 /df.isnull().count()).sort_values(ascending=False)
    missing_data = pd.concat([total, percent], axis=1, keys=['Total', 'Percent'])
    return(missing_data)

train_cab["passenger_count"] =
train_cab["passenger_count"].fillna(train_cab["passenger_count"].median())
train_cab["fare_amount"] =
train_cab["fare_amount"].fillna(train_cab["fare_amount"].median())

```


R code

```
missing_val = data.frame(apply(train,2,function(x){sum(is.na(x))}))
missing_val$Columns = row.names(missing_val)
names(missing_val)[1] = "Missing_percentage"
missing_val$Missing_percentage = (missing_val$Missing_percentage/nrow(train)) * 100
missing_val = missing_val[order(-missing_val$Missing_percentage),]
row.names(missing_val) = NULL
missing_val = missing_val[,c(2,1)]
missing_val
```

Imputation of missing values in R using KNN

Replace all outliers with NA and impute

```
vals = train[, "fare_amount"] %in% boxplot.stats(train[, "fare_amount"])$out
train[which(vals), "fare_amount"] = NA
```

#lets check the NA's

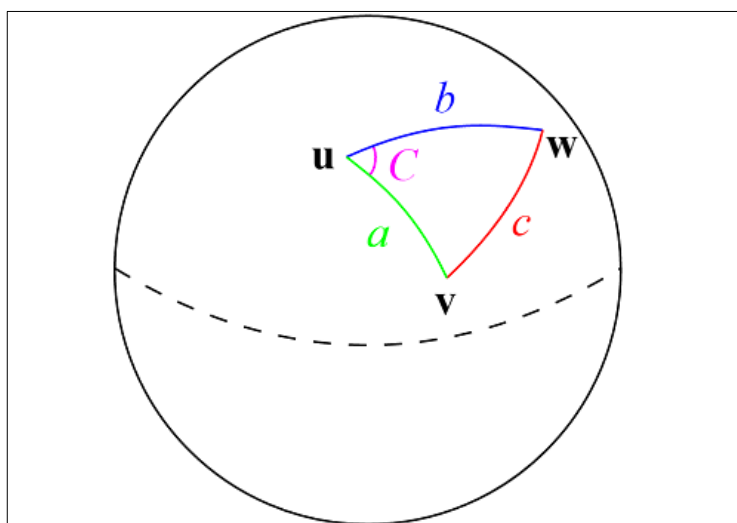
```
sum(is.na(train$fare_amount))
```

#Imputing with KNN

```
train = knnImputation(train,k=3)
```

2.1.2 Feature Engineering

After cleaning both train and test dataset now, we tried to find out the distance using the haversine formula which says: The haversine formula determines the great-circle distance between two points on a sphere given their 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.



So, our new extracted variables are:

- fare_amount
- pickup_datetime
- pickup_longitude
- pickup_latitude
- dropoff_longitude
- dropoff_latitude
- passenger_count
- year ▪ Month ▪ Date ▪ Day of Week ▪ Hour ▪ Minute
- Distance

Selection of variables

Now as we know that all above variables are of now use so we will drop the redundant variables:

- pickup_datetime ▪ pickup_longitude ▪ pickup_latitude ▪ dropoff_longitude ▪ dropoff_latitude
- Minute

Only below variables will be used for further analysis

	fare_amount	passenger_count	year	Month	Date	Day	Hour	distance
0	4.5	1.0	2009.0	6.0	15.0	0.0	17.0	1.030764
1	16.9	1.0	2010.0	1.0	5.0	1.0	16.0	8.450134
2	5.7	2.0	2011.0	8.0	18.0	3.0	0.0	1.389525
3	7.7	1.0	2012.0	4.0	21.0	5.0	4.0	2.799270
4	5.3	1.0	2010.0	3.0	9.0	1.0	7.0	1.999157

Now, we need to look at the unique number in the variables which help us to decide whether the variable is categorical or numeric. So, by using python script 'nunique' we tried to find out the unique values in each variable.

Variable Name	Unique Counts
fare_amount	450
passenger_count	7
year	7
Month	12
Date	31
Day of Week	7

Hour	24
distance	15424

Dividing the variables into two categories basis their data types:

Continuous variables - 'fare_amount', 'distance'.

Categorical Variables - 'year', 'Month', 'Date', 'Day of Week', 'Hour', 'passenger_count'

Feature Engineering in Python and R

Python

```
#As we know that we have given pickup longitude and latitude values and same for drop.
#So we need to calculate the distance Using the haversine formula and we will create a new
variable called distance
from math import radians, cos, sin, 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 * asin(sqrt(a))
    # Radius of earth in kilometers is 6371
    km = 6371 * c
    return km

# 1min

train_cab['distance'] =
train_cab[['pickup_longitude','pickup_latitude','dropoff_longitude','dropoff_latitude']].apply(ha
versine,axis=1)
```

```

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

train_cab.head()

test_cab.head()

train_cab.nunique()

test_cab.nunique()

##finding decending order of fare to get to know whether the outliers are presented or not
train_cab['distance'].sort_values(ascending=False)

# As we can see that top 23 values in the distance variables are very high It means more than
8000 Kms distance they have travelled
# Also just after 23rd value from the top, the distance goes down to 127, which means these
values are showing some outliers
# We need to remove these values

Counter(train_cab['distance'] == 0)

Counter(test_cab['distance'] == 0)

Counter(train_cab['fare_amount'] == 0)

###we will remove the rows whose distance value is zero

train_cab = train_cab.drop(train_cab[train_cab['distance']== 0].index, axis=0)
train_cab.shape

#we will remove the rows whose distance values is very high which is more than 129kms
train_cab = train_cab.drop(train_cab[train_cab['distance'] > 130 ].index, axis=0)
train_cab.shape

# Now we have splitted the pickup date time variable into different variables like month, year,
day etc so now we dont need to have that pickup_Date variable now. Hence we can drop that,
Also we have created distance using pickup and drop longitudes and latitudes so we will also
drop pickup and drop longitudes and latitudes variables.

drop = ['pickup_datetime', 'pickup_longitude', 'pickup_latitude','dropoff_longitude',
'dropoff_latitude', 'Minute']

```

```

train_cab = train_cab.drop(drop, axis = 1)

train_cab.head()

train_cab['passenger_count'] = train_cab['passenger_count'].astype('int64')

train_cab['year'] = train_cab['year'].astype('int64')
train_cab['Month'] = train_cab['Month'].astype('int64')
train_cab['Date'] = train_cab['Date'].astype('int64')
train_cab['Day'] = train_cab['Day'].astype('int64')
train_cab['Hour'] = train_cab['Hour'].astype('int64')

drop_test = ['pickup_datetime', 'pickup_longitude', 'pickup_latitude', 'dropoff_longitude',
'dropoff_latitude', 'Minute']
test_cab = test_cab.drop(drop_test, axis = 1)

```

R code

```

##### Feature Engineering #####
#Convert pickup_datetime from factor to date time
train$pickup_date = as.Date(as.character(train$pickup_datetime))
train$pickup_weekday = as.factor(format(train$pickup_date,"%u"))# Monday = 1
train$pickup_mnth = as.factor(format(train$pickup_date,"%m"))
train$pickup_yr = as.factor(format(train$pickup_date,"%Y"))
pickup_time = strptime(train$pickup_datetime,"%Y-%m-%d %H:%M:%S")
train$pickup_hour = as.factor(format(pickup_time,"%H"))

#Add same features to test set
test$pickup_date = as.Date(as.character(test$pickup_datetime))
test$pickup_weekday = as.factor(format(test$pickup_date,"%u"))# Monday = 1
test$pickup_mnth = as.factor(format(test$pickup_date,"%m"))
test$pickup_yr = as.factor(format(test$pickup_date,"%Y"))
pickup_time = strptime(test$pickup_datetime,"%Y-%m-%d %H:%M:%S")
test$pickup_hour = as.factor(format(pickup_time,"%H"))

sum(is.na(train))# there was 1 'na' in pickup_datetime which created na's in above feature
engineered variables.
train = na.omit(train) # we will remove that 1 row of na's

train = subset(train,select = -c(pickup_datetime,pickup_date))
test = subset(test,select = -c(pickup_datetime,pickup_date))

```

```

# 2.Calculate the distance travelled using longitude and latitude
deg_to_rad = function(deg){
  (deg * pi) / 180
}
haversine = function(long1,lat1,long2,lat2){
  #long1rad = deg_to_rad(long1)
  phi1 = deg_to_rad(lat1)
  #long2rad = deg_to_rad(long2)
  phi2 = deg_to_rad(lat2)
  delphi = deg_to_rad(lat2 - lat1)
  dellamda = deg_to_rad(long2 - long1)

  a = sin(delphi/2) * sin(delphi/2) + cos(phi1) * cos(phi2) *
    sin(dellamda/2) * sin(dellamda/2)

  c = 2 * atan2(sqrt(a),sqrt(1-a))
  R = 6371e3
  R * c / 1000 #1000 is used to convert to meters
}
# Using haversine formula to calculate distance fr both train and test
train$dist =
haversine(train$pickup_longitude,train$pickup_latitude,train$dropoff_longitude,train$dropoff_
latitude)
test$dist =
haversine(test$pickup_longitude,test$pickup_latitude,test$dropoff_longitude,test$dropoff_latit
ude)

# We will remove the variables which were used to feature engineer new variables
train = subset(train,select = -
c(pickup_longitude,pickup_latitude,dropoff_longitude,dropoff_latitude))
test = subset(test,select = -
c(pickup_longitude,pickup_latitude,dropoff_longitude,dropoff_latitude))

```

2.1.3 Outlier Analysis

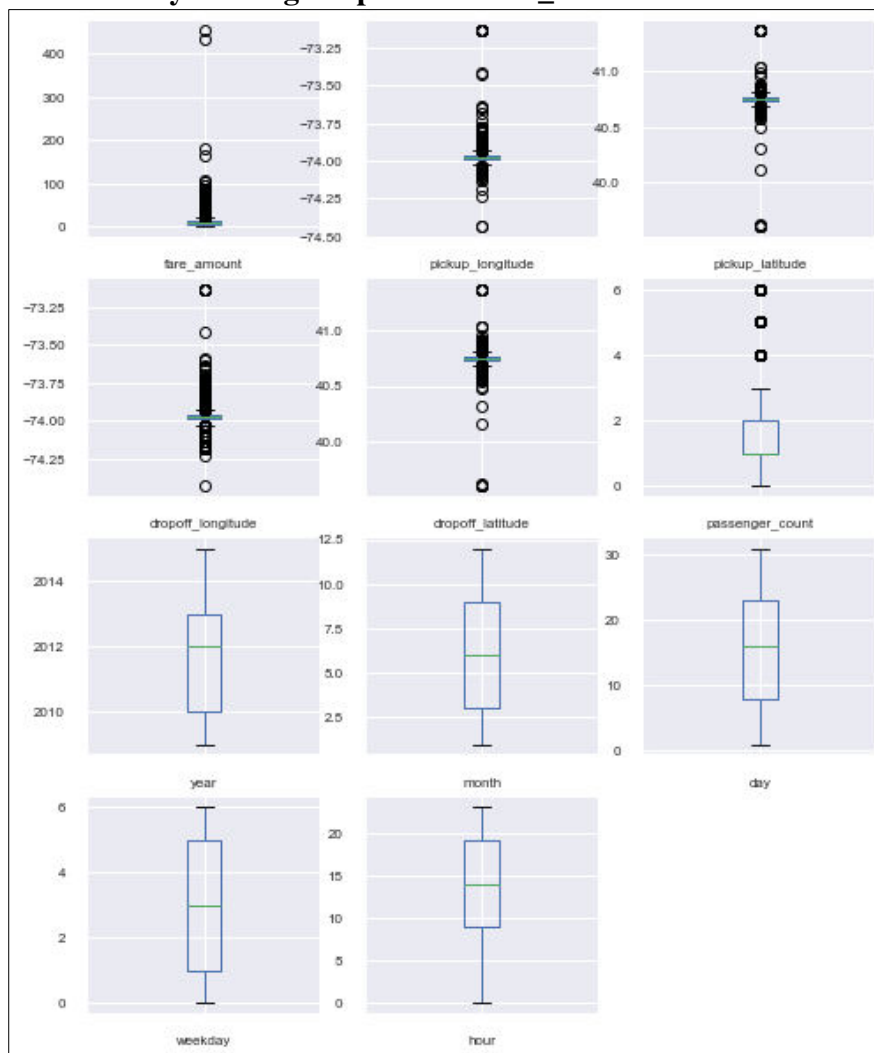
Outliers are extreme values that deviate from other observations on data, they may indicate a variability in a measurement, experimental errors or a novelty. In other words, an outlier is an observation that diverges from an overall pattern on a sample.

In layman terms, we can say that an outlier is something which is separated/different from the crowd. Also, Outlier analysis is very important because they affect the mean and median which in turn affects the error (absolute and mean) in any data set. When we plot the error we might get big deviations if outliers are in the data set. In Box plots analysis of individual features, we can clearly observe from these boxplots that, not every feature contains outliers and many of them even have very few outliers.

Outliers analysis using boxplots

In Box plots analysis of individual features, we can clearly observe from below boxplots that, not every feature contains outliers and many of them even have very few outliers. We have only 16k data-points and after removing the outliers, the data gets decreased by almost 15%. So, dropping the outliers is probably not the best idea. Instead we will try to visualise and find out the outliers using box plots and will fill them with NA, that means we have created 'missing values' in place of outliers within the data. Now, we can treat these outliers like missing values and impute them using standard imputation techniques. In our case, we use Median imputation in Python and KNN imputation in R to impute these missing values.

Outlier analysis using boxplots on train_dataset



Outlier analysis in R and Python is pasted below:

Python code

```
##Detect and delete outliers from data
def outliers_analysis(df):
    for i in df.columns:
        print(i)
```

```

q75, q25 = np.percentile(df.loc[:,i], [75 ,25])
iqr = q75 - q25

min = q25 - (iqr*1.5)
max = q75 + (iqr*1.5)
print(min)
print(max)

df = df.drop(df[df.loc[:,i] < min].index)
df = df.drop(df[df.loc[:,i] > max].index)
return(df)
#Imputation with median

def eliminate_rows_with_zero_value(df):
    df= df[df!= 0]
    df=df.fillna(df.median())
    return(df)

R code

# Boxplot for fare_amount
pl1 = ggplot(train,aes(x = factor(passenger_count),y = fare_amount))
pl1 + geom_boxplot(outlier.colour="red", fill = "grey" ,outlier.shape=18,outlier.size=1,
notch=FALSE)+ylim(0,100)

# Replace all outliers with NA and impute
vals = train[, "fare_amount"] %in% boxplot.stats(train[, "fare_amount"])$out
train[which(vals), "fare_amount"] = NA

#lets check the NA's
sum(is.na(train$fare_amount))

#Imputing with KNN
train = knnImputation(train,k=3)

```

2.1.4 Data Visualisation

Data visualisation helps us to get better insights of the data. By visualising data, we can identify areas that need attention or improvement and clarifies which factors influence fare of the cab and how the resources are used to determine it.

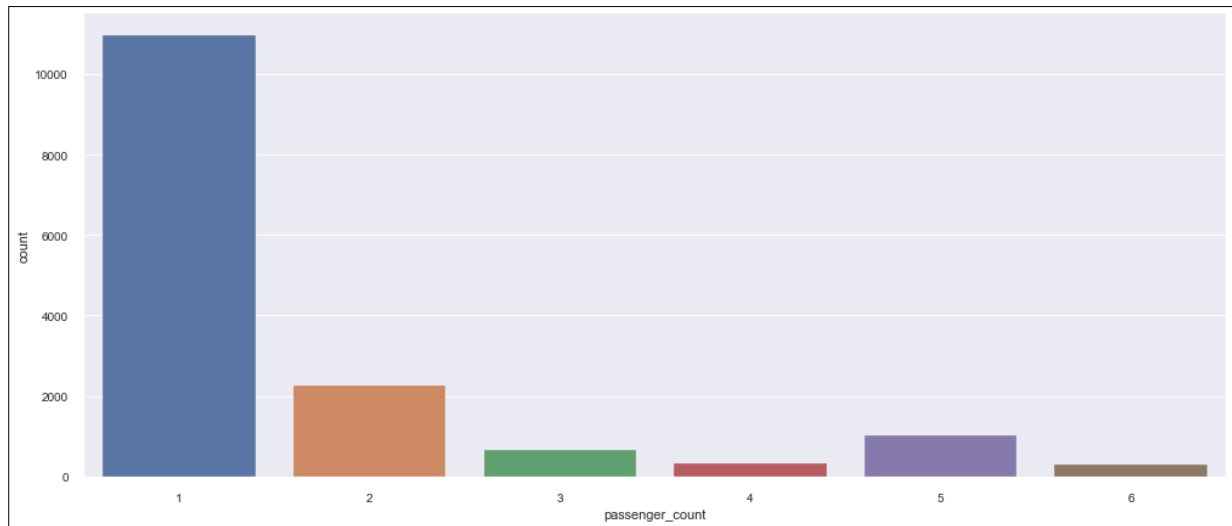
Univariate analysis

Univariate analysis is the simplest form of data analysis where the data being analysed contains only one variable. Since it's a single variable it doesn't deal with causes or relationships. The

main purpose of univariate analysis is to describe the data and find patterns that exist within it. So, Let's have a look at histogram plot, to identify the characteristic of the features and the data.

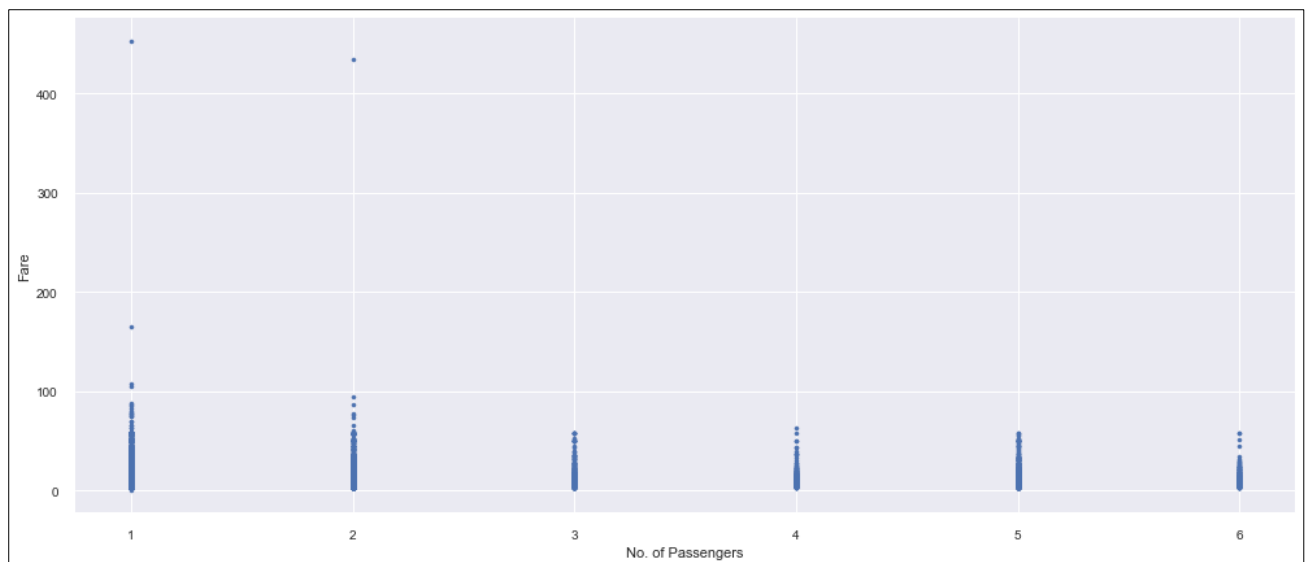
At first we will construct Histogram

Count plot on passenger count

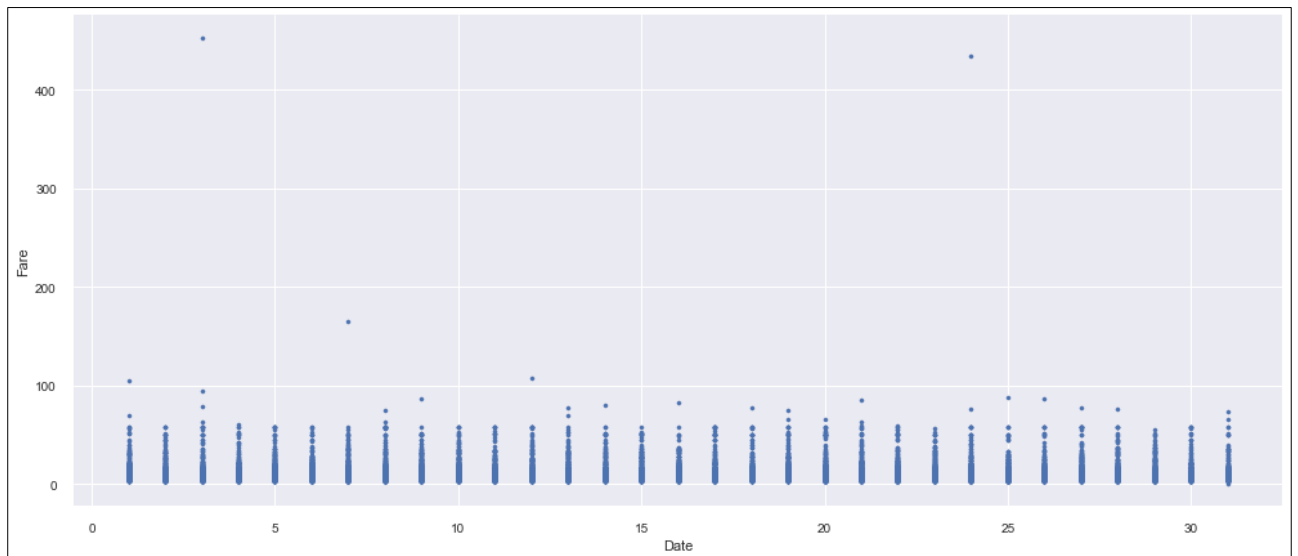


Single passengers are the most frequent travellers

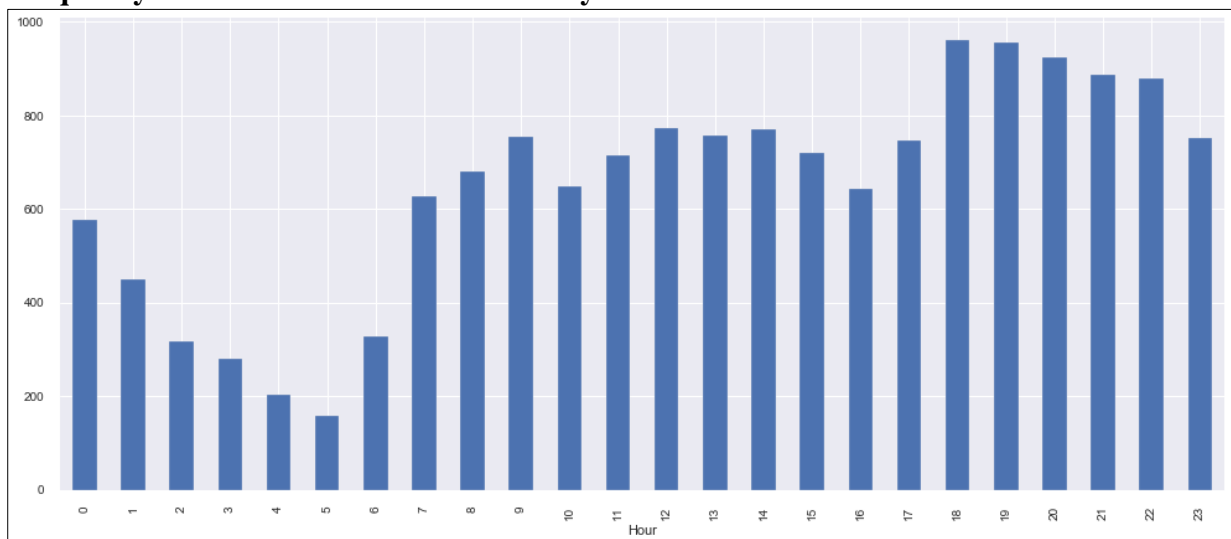
Relationship between number of passengers and Fare amount



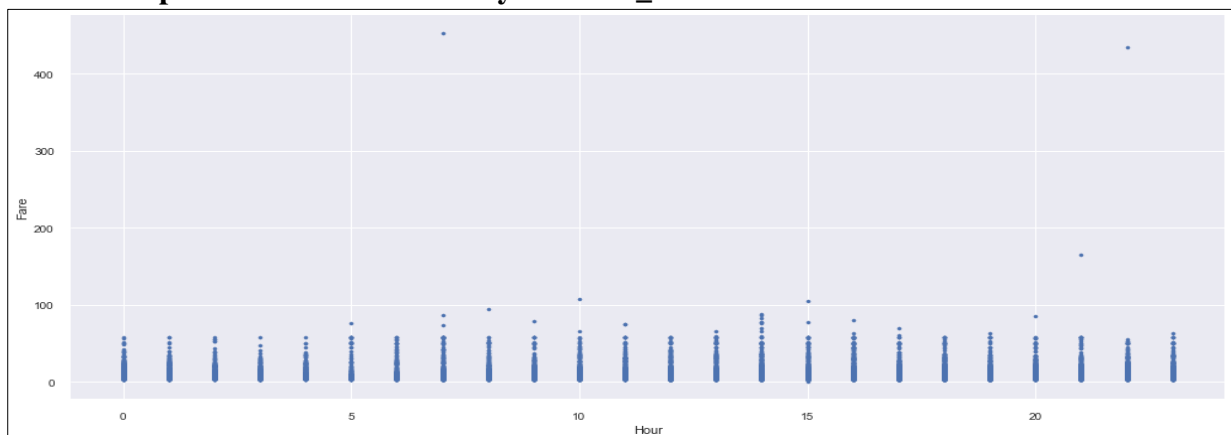
Relationship between date and Fare amount



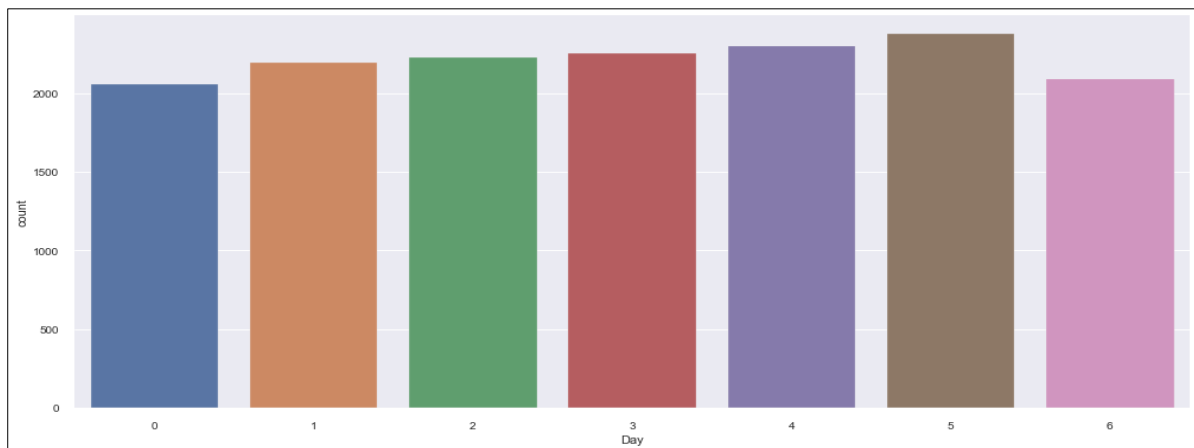
Frequency of cab rides in 'Hour' of the day



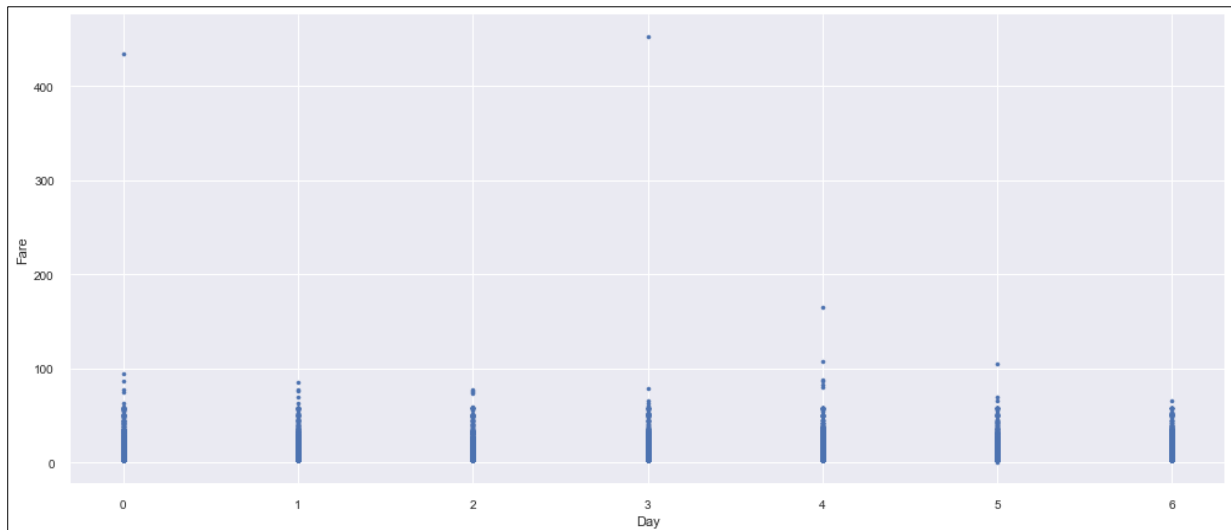
Relationship between hour of the day and fare_amount



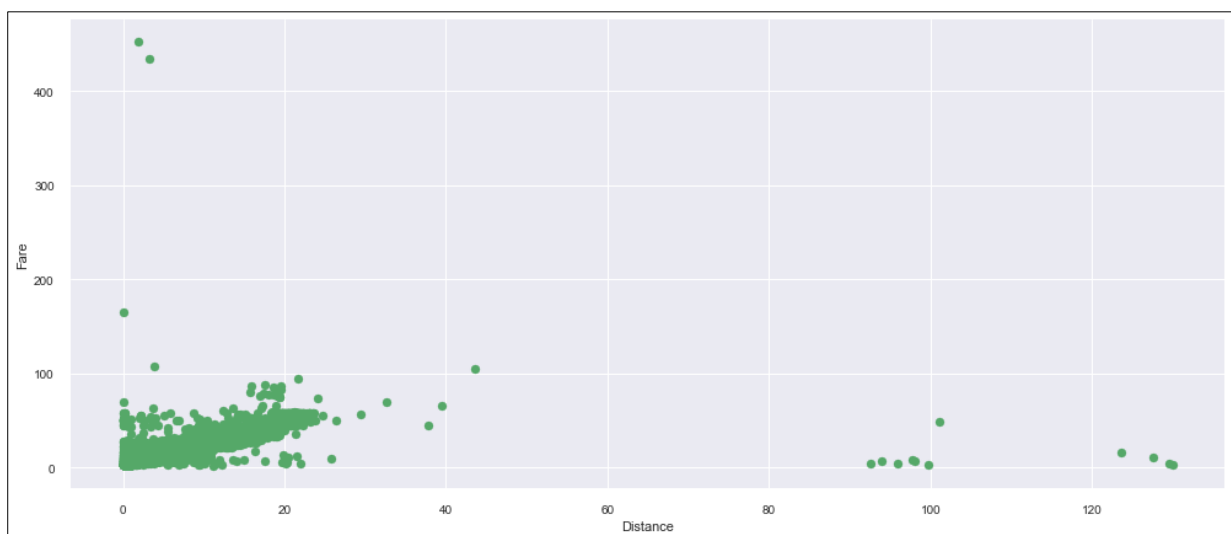
Impact of Day on the number of cab rides



Relationships between day and Fare_amount



Relationship between distance and fare_amount



Findings:

- We can see in graph that single passengers are the most frequent travellers, and the highest fare also seems to come from cabs which carry just 1 passenger.
- The fares throughout the month mostly seem uniform
- During hours 6 PM to 11PM the frequency of cab boarding is very high due to peak hours.
- Fare prices during 2PM to 8PM is bit high compared to all other time might be due to high demands.
- Cab fare is high on Friday, Saturday and Monday, may be during weekend and first day of the working day they charge high fares because of high demands of cabs.
- The day of the week does not seem to have much influence on the number of cabs ride

2.1.5 Feature selection

Feature Selection is the process where you automatically or manually select those features which contribute most to your prediction variable or output in which you are interested in. Having irrelevant features in your data can decrease the accuracy of the models and make your model learn based on irrelevant features.

- **Reduces Overfitting:** Less redundant data means less opportunity to make decisions based on noise.
- **Improves Accuracy:** Less misleading data means modeling accuracy improves.
- **Reduces Training Time:** fewer data points reduce algorithm complexity and algorithms train faster.

There are few Feature selection techniques that can be used viz., Correlation analysis, Chi-Square test, ANOVA, Univariate analysis, Feature Importance. In this project as there are no categorical variables, we will perform Correlation analysis and Feature importance.

Before performing any type of modeling we need to assess the importance of each predictor variable in our analysis. There is a possibility that many variables in our analysis are not important at all to the problem of class prediction. Selecting subset of relevant columns for the model construction is known as Feature Selection. We cannot use all the features because some features may be carrying the same information or irrelevant information which can increase overhead. To reduce overhead, we adopt feature selection technique to extract meaningful features out of data. This in turn helps us to avoid the problem of multi collinearity. In this project we have selected **Correlation Analysis** for numerical variable and **ANOVA** (Analysis of variance) for categorical variable.

Correlation analysis

Correlation helps us understand relationships between different attributes of the data. Since we have to predict the fare_amount in this project (Focus is on forecasting), so correlations help us understand and exploit relationships to build better models.

Findings:

1. The value for collinearity is between -1 to 1. So, any value close to -1/1 will result in high collinearity.
2. It seems all right in our train and test data the situations nothing is more than 0.4 is positive direction and nothing is less than -0.1 is negative direction.
3. Therefore, the datasets are free from collinearity problem.

Python code

```
def Correlation(df):
    df_corr = df.loc[:,df.columns]
    sns.set()
    plt.figure(figsize=(9, 9))
    corr = df_corr.corr()
    sns.heatmap(corr, annot= True,fmt = " .3f", linewidths = 0.5,
                square=True)
```

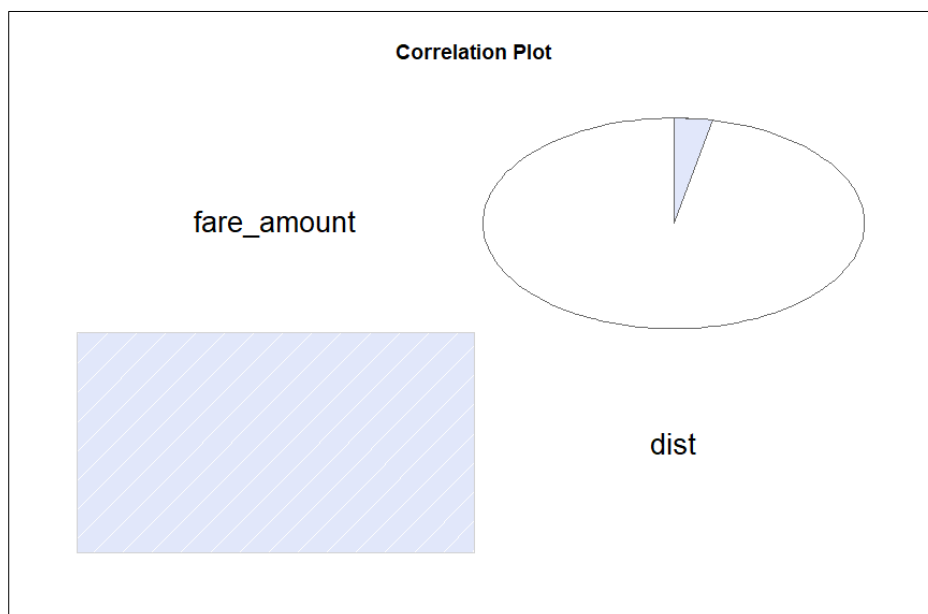
```
Correlation(train_cab)
```

```
Correlation(test_cab)
```

R code

```
numeric_index = sapply(train,is.numeric) #selecting only numeric
numeric_data = train[,numeric_index]
cnames = colnames(numeric_data)
#Correlation analysis for numeric variables
corrgram(train[,numeric_index],upper.panel=panel.pie, main = "Correlation Plot")
```

Correlation plot in R



ANOVA in R

Analysis of Variance (ANOVA) is a statistical technique, commonly used to studying differences between two or more group means. ANOVA test is centred on the different sources of variation in a typical variable. ANOVA in R primarily provides evidence of the existence of the mean equality between the groups.

Hypothesis testing:

- Null Hypothesis: mean of all categories in a variable are same.
- Alternate Hypothesis: mean of at least one category in a variable is different.
- If p-value is less than 0.05 then we reject the null hypothesis.
- And if p-value is greater than 0.05 then we accept the null hypothesis.

```
#ANOVA for categorical variables with target numeric variable
#aov_results = aov(fare_amount ~ passenger_count * pickup_hour * pickup_weekday,data = train)
aov_results = aov(fare_amount ~ passenger_count + pickup_hour + pickup_weekday + pickup_mnth + pickup_yr,data = train)

> summary(aov_results)

# pickup_weekday has p value greater than 0.05
train = subset(train,select=-pickup_weekday)

#remove from test set
test = subset(test,select=-pickup_weekday)
```

```
>
> summary(aov_results)
              Df Sum Sq Mean Sq F value    Pr(>F)
passenger_count    5     252    50.5    2.656   0.0209 *
pickup_hour       23    2560   111.3    5.858 < 2e-16 ***
pickup_weekday     6      59     9.8    0.518   0.7954
pickup_mnth       11     985    89.6    4.715 2.98e-07 ***
pickup_yr         6    7107  1184.6   62.353 < 2e-16 ***
Residuals       15606 296478    19.0
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

>
```

Multicollinearity

Multicollinearity is a state of very high intercorrelations or inter-associations among the independent variables. It is therefore a type of disturbance in the data, and if present in the data the statistical inferences made about the data may not be reliable.

```
R
#check multicollarity
```

```
library(usdm)
vif(train[,-1])
vifcor(train[,-1], th = 0.9)
```

Findings:

We have checked for multicollinearity in our Dataset and all VIF values are below 5.

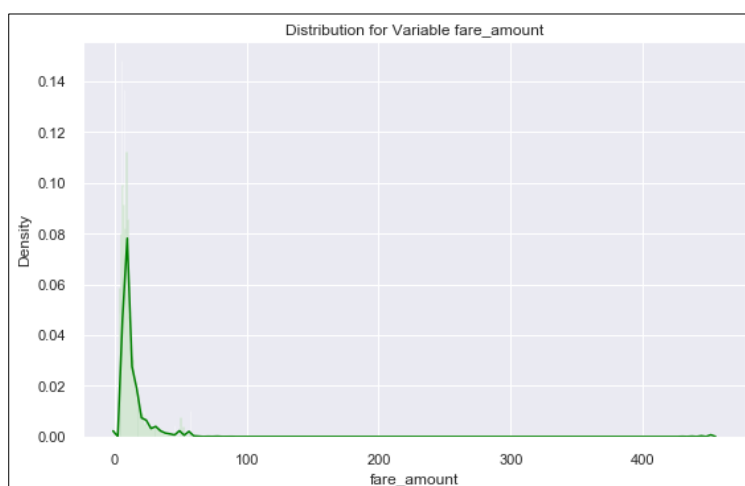
2.1.6 Feature Scaling

Feature scaling is a method used to standardize the range of independent variables or features of data. In data processing, it is also known as data normalization and is generally performed during the data pre-processing step.

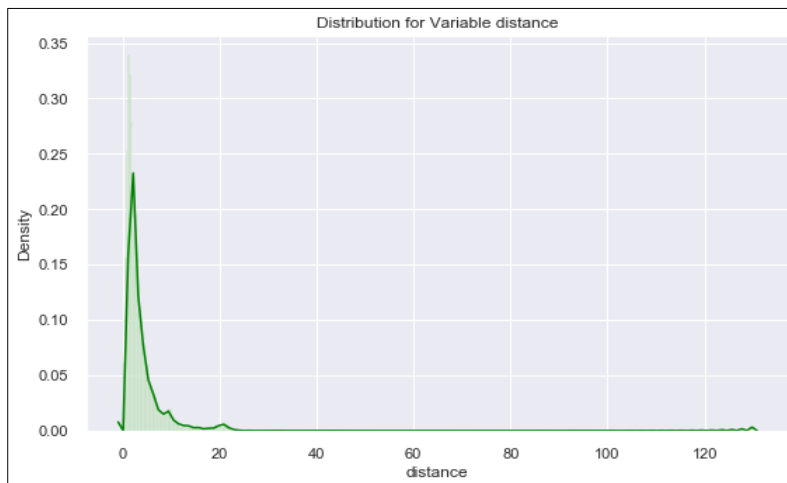
Using the raw values as input features might make models biased toward features having high magnitude values. These models are typically sensitive to the magnitude or scale of features like linear or logistic regression. Therefore, it is recommended to normalize and scale down the features with feature scaling. Here we use Normalisation technique as the train & test _dataset are not normally distributed. Therefore, the range of all features should be normalized so that each feature contributes approximately proportionately to the final distance.

Skewness is asymmetry in a statistical distribution, in which the curve appears distorted or skewed either to the left or to the right. Skewness can be quantified to define the extent to which a distribution differs from a normal distribution. Here we tried to show the skewness of our variables and we find that our target variable absenteeism in hours having is one sided skewed so by using log transform technique we tried to reduce the skewness of the same. Below mentioned graphs shows the probability distribution plot to check distribution before log transformation

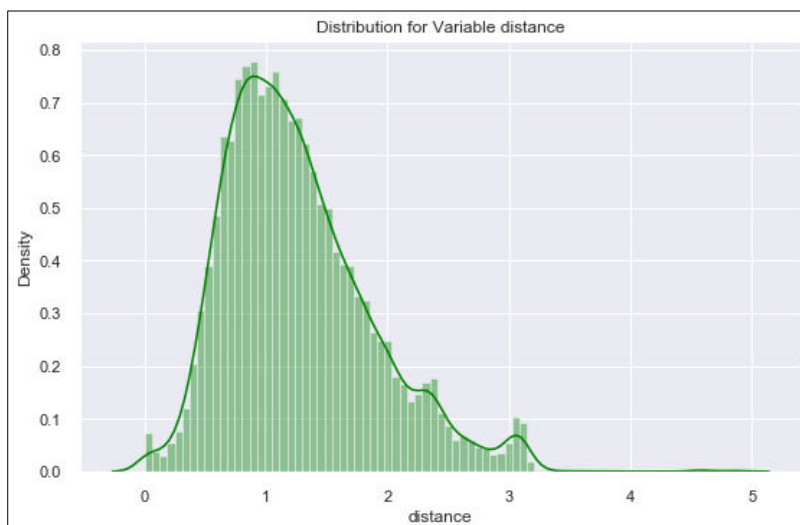
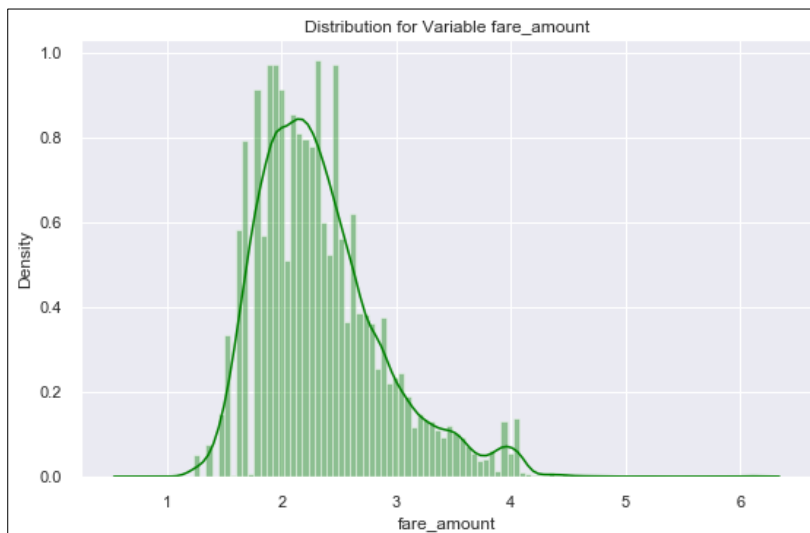
Probability distribution plot of fare_amount



Probability distribution plot of distance



After Log transformation



As our continuous variables appears to be normally distributed so we don't need to use feature scaling techniques like normalization and standardization for the same in Python

Python

#Normality check of training data is uniformly distributed or not-

```
for i in ['fare_amount', 'distance']:
```

```
    print(i)
    sns.distplot(train_cab[i],bins='auto',color='green')
    plt.title("Distribution for Variable "+i)
    plt.ylabel("Density")
    plt.show()
```

#since skewness of target variable is high, apply log transform to reduce the skewness-

```
train_cab['fare_amount'] = np.log1p(train_cab['fare_amount'])
```

#since skewness of distance variable is high, apply log transform to reduce the skewness-

```
train_cab['distance'] = np.log1p(train_cab['distance'])
```

#Normality Re-check to check data is uniformly distributed or not after log transformation

```
for i in ['fare_amount', 'distance']:
```

```
    print(i)
    sns.distplot(train_cab[i],bins='auto',color='green')
    plt.title("Distribution for Variable "+i)
    plt.ylabel("Density")
    plt.show()
```

Here we can see bell shaped distribution. Hence our continuous variables are now normally distributed, we will use not use any Feature Scaling technique. i.e, Normalization or Standardization for our training data

#Normality check for test data is uniformly distributed or not-

```
sns.distplot(test_cab['distance'],bins='auto',color='green')
plt.title("Distribution for Variable "+i)
plt.ylabel("Density")
plt.show()
```

#since skewness of distance variable is high, apply log transform to reduce the skewness-

```
test_cab['distance'] = np.log1p(test_cab['distance'])
```

#rechecking the distribution for distance

```
sns.distplot(test_cab['distance'],bins='auto',color='green')
```

```
plt.title("Distribution for Variable "+i)
```

```
plt.ylabel("Density")
```

```
plt.show()
```

As we can see a bell shaped distribution. Hence our continuous variables are now normally distributed, we will use not use any Feature Scaling technique. i.e, Normalization or Standardization for our test data

Normalisation in R

#Normality check

```
qqnorm(train$fare_amount)
```

```
histogram(train$fare_amount)
```

```
library(car)
```

```
# dev.off()
```

```
par(mfrow=c(1,2))
```

```
qqPlot(train$fare_amount)
```

qqPlot, it has a x values derived from gaussian distribution, if data is distributed normally then the sorted data points should lie very close to the solid reference line

```
truehist(train$fare_amount)
```

truehist() scales the counts to give an estimate of the probability density.

```
lines(density(train$fare_amount))
```

Right skewed

lines() and density() functions to overlay a density plot on histogram

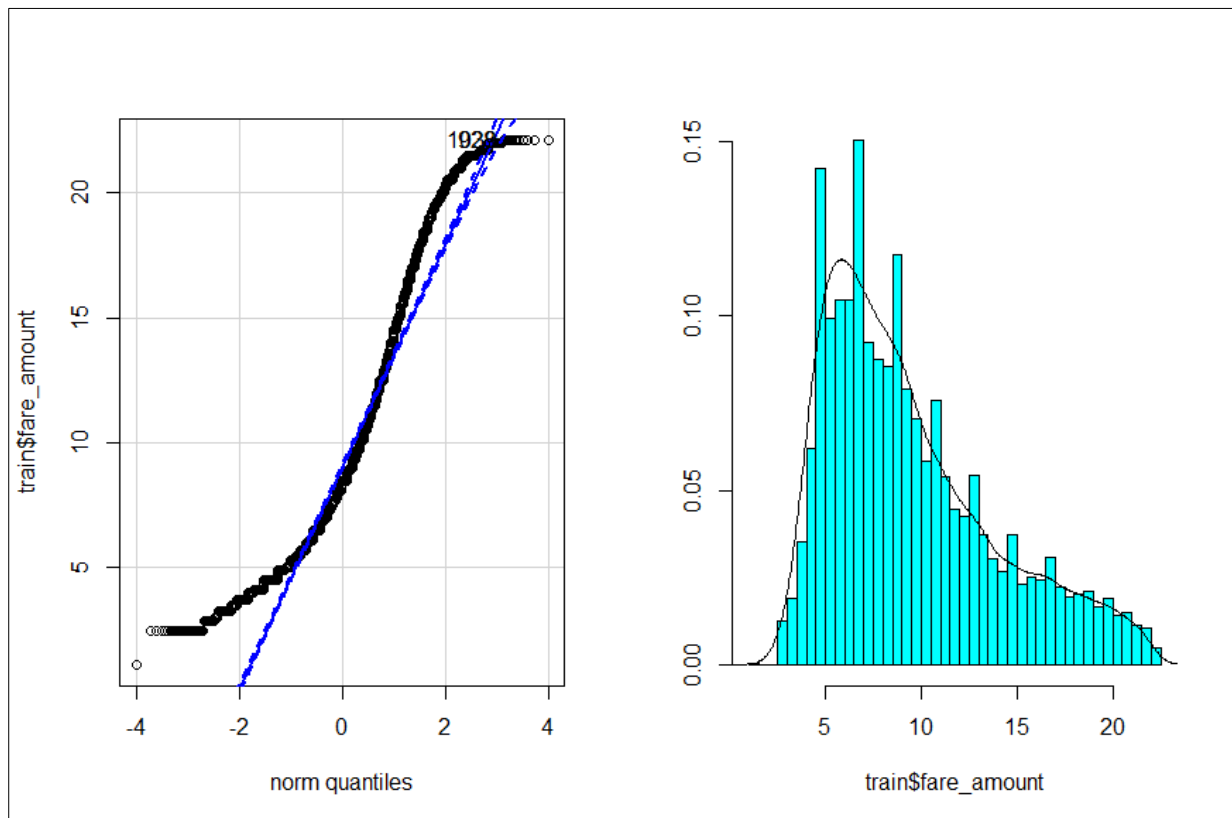
#Normalisation

```
print('dist')
```

```
train[, 'dist'] = (train[, 'dist'] - min(train[, 'dist']))/
```

```
(max(train[, 'dist'] - min(train[, 'dist'])))
```


Normalisation plot in R



Chapter 3

Modeling

As in the pre-processing steps we understood that the target variable is continuous, it is a regression problem. Once all the Pre-Processing steps has been done on our data set, we will now further move to our next step which is modelling. Modelling plays an important role to find out the good inferences from the data. Choice of models depends upon the problem statement and data set. As per our problem statement and dataset, we will try some models on our pre-processed data and post comparing the output results we will select the best suitable model for our problem. As per our data set following models need to be tested:

1. **Linear regression**
2. **Decision Tree**
3. **Random forest,**
4. **Gradient Boosting in Python and XG Boost in R**

Hyperparameter tuning

In contrast to model parameters which are learned during training, model hyper parameters are set by data scientist ahead of training and control implementation aspects of the model. The weights learned during training of a linear regression model are parameters while the number of trees in a random forest is a model hyperparameter because this is set by the data scientist. Hyperparameter can be thought of as model settings. These settings need to be tuned for each problem because the best model hyperparameters for one particular dataset will **not be** the best across all datasets. The process of hyperparameter tuning (also called hyperparameter optimization) means finding the combination of hyperparameter values for a machine learning model that performs the best - as measured on a validation dataset - for a problem.

There are several approaches to hyperparameter tuning

Manual: select hyperparameters based on intuition/experience/guessing, train the model with the hyperparameters, and score on the validation data. Repeat process until you run out of patience or are satisfied with the results.

Grid Search: set up a grid of hyperparameter values and for each combination, train a model and score on the validation data. In this approach, every single combination of hyperparameters values is tried which can be very inefficient!

Random search: set up a grid of hyperparameter values and select *random* combinations to train the model and score. The number of search iterations is set based on time/resources.

Automated Hyperparameter Tuning: use methods such as gradient descent, Bayesian Optimization, or evolutionary algorithms to conduct a guided search for the best hyperparameters.

We have used hyper parameter tunings to check the parameters on which our model runs best. Following are two techniques of hyper parameter tuning we have used:

- Grid Search CV
- Random Search CV

3.1 Multiple Linear Regression

Multiple linear regression is the most common form of linear regression analysis. Multiple regression is an extension of simple linear regression. It is used as a predictive analysis, when we want to predict the value of a variable based on the value of two or more other variables. The variable we want to predict is called the dependent variable (or sometimes, the outcome, target or criterion variable).

In python

```
# Building model on top of training dataset
fit_LR = LinearRegression().fit(X_train , y_train)

#prediction on train data
pred_train_LR = fit_LR.predict(X_train)

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

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

Root Mean Squared Error For Train data = 0.2761331342349163
Root Mean Squared Error For Test data = 0.24330150021027375

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

0.7436717513591988

r2_score(y_test, pred_test_LR)

0.7990769280403993
```

In R

```
lm_model = lm(fare_amount ~.,data=train_data)

summary(lm_model)

str(train_data)

plot(lm_model$fitted.values,rstandard(lm_model),main = "Residual plot",

xlab = "Predicted values of fare_amount",

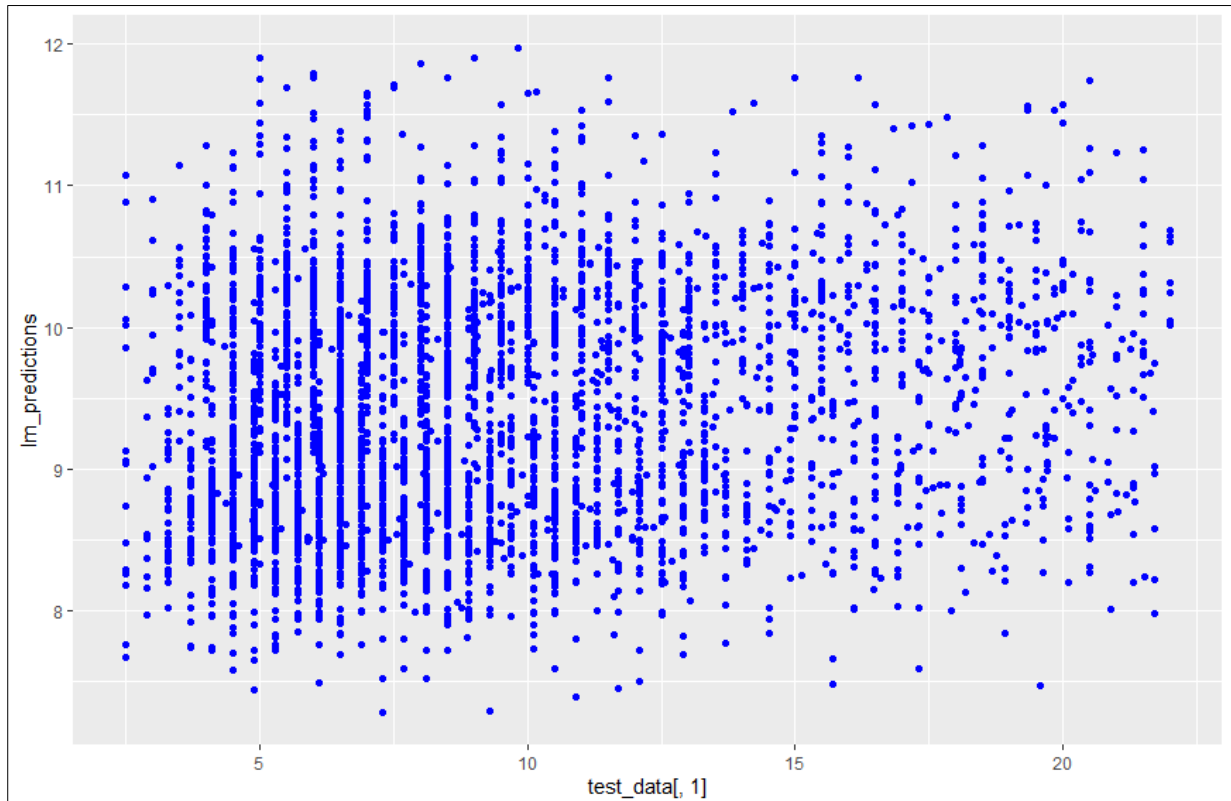
ylab = "standardized residuals")
```

```
lm_predictions = predict(lm_model,test_data[,2:6])

qplot(x = test_data[,1], y = lm_predictions, data = test_data, color = I("blue"), geom = "point")

regr.eval(test_data[,1],lm_predictions)

# mae    mse    rmse    mape
# 3.5022600 19.1274501 4.3734940 0.4503374
```



3.2 Decision Trees

Decision trees are supervised learning algorithms, simple yet powerful in modeling non-linear relationships. Being a non-parametric model, the aim of this algorithm is to learn a model that can predict outcomes based on simple decision rules based on features. In decision analysis, a decision tree can be used to visually and explicitly represent decisions and decision making. As the name goes, it uses a tree-like model of decisions.

```
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)
```

```
##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 = 0.29963421167463683
Root Mean Squared Error For Test data = 0.28376814264939254
```

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

0.6981840250065832
```

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

0.7266824550035069
```

#2.Decision Tree

#Fit the model

```
Dt_model = rpart(fare_amount ~ ., data = train_data, method = "anova")
```

#Summary of the model

```
summary(Dt_model)
```

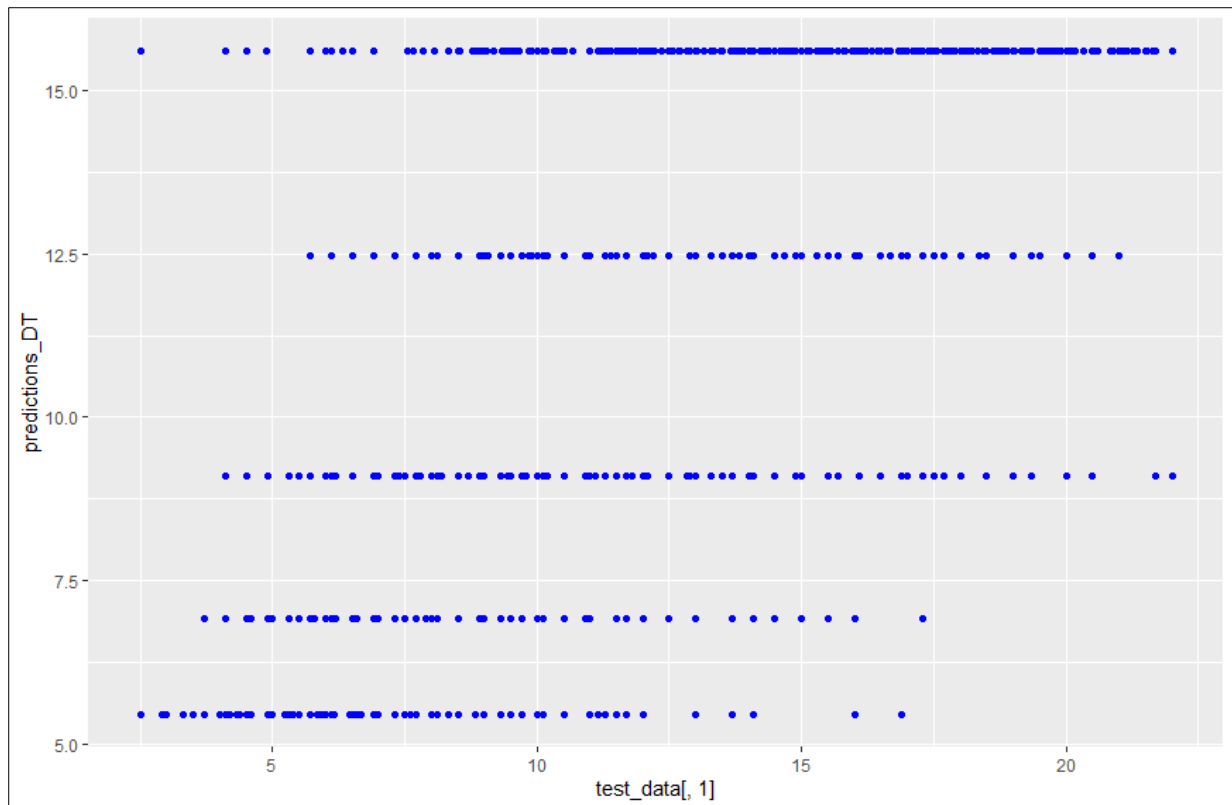
#Predict for new test cases

```
predictions_DT = predict(Dt_model, test_data[,2:6])
```

```
qplot(x = test_data[,1], y = predictions_DT, data = test_data, color = I("blue"), geom = "point")
```

```
regr.eval(test_data[,1],predictions_DT)
```

# mae	mse	rmse	mape
#1.9001012	6.6918569	2.5868624	0.2192849



3.3 Random Forest

Random forests or random decision forests are an ensemble learning method for classification, regression and other tasks that operates by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes or mean prediction of the individual trees

```
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)
```

```
##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 = 0.09575056699976202
Root Mean Squared Error For Test data = 0.23902267147354572
```

```
## calculate R^2 for train data
```

```
r2_score(y_train, pred_train_RF)
```

```
0.9691793258279341
```

```
#calculate R^2 for test data
```

```
r2_score(y_test, pred_test_RF)
```

```
0.8060818639628967
```

#3.Random forest

#Fit the model

```
rf_model = randomForest(fare_amount ~.,data=train_data)
```

#summary of the model

```
summary(rf_model)
```

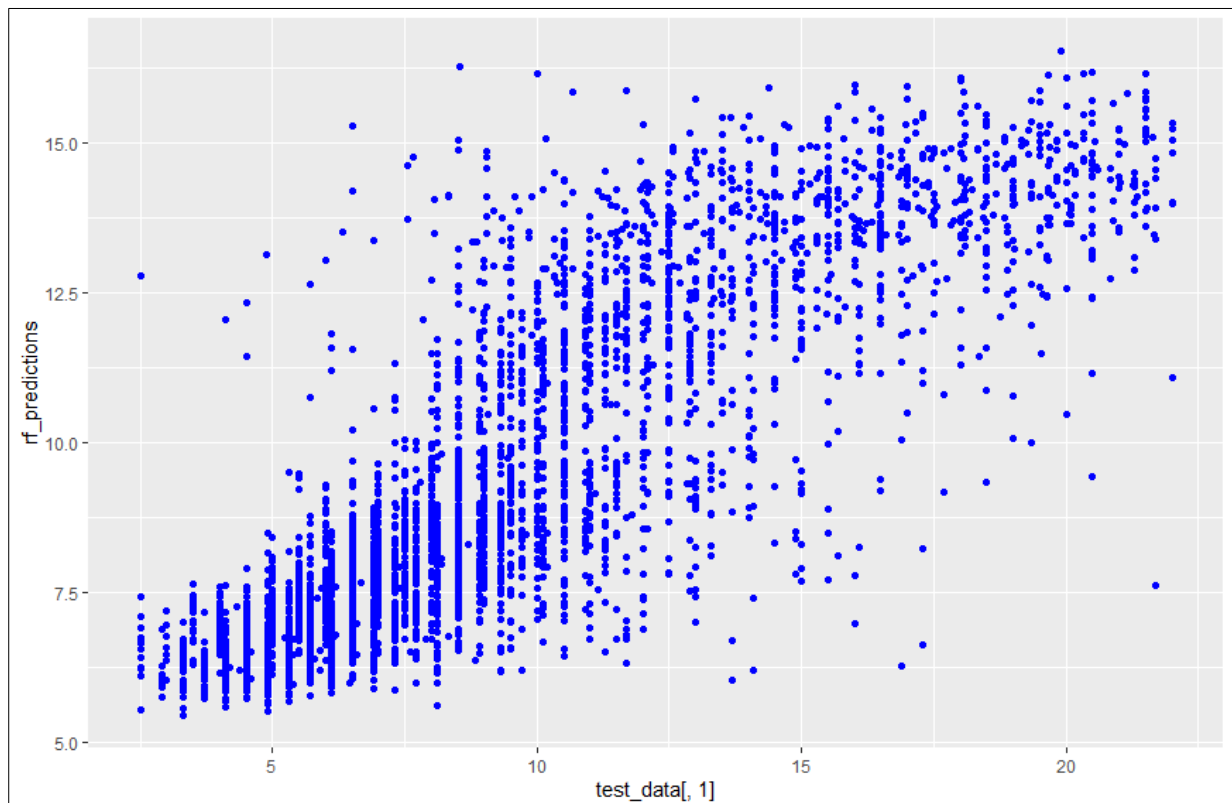
#predict the model on test data

```
rf_predictions = predict(rf_model,test_data[,2:6])
```

```
qplot(x = test_data[,1], y = rf_predictions, data = test_data, color = I("blue"), geom = "point")
```

```
regr.eval(test_data[,1],rf_predictions)
```

# mae	mse	rmse	mape
#1.9103915	6.4214597	2.5340599	0.2342449



3.4 Gradient Boosting

Regression 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 generalises them by allowing optimisation of an arbitrary differentiable loss function.

```

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)

##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 = 0.2283322057954371
Root Mean Squared Error For Test data = 0.2275740475846608

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

0.824213425351429

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

0.8247355760276998

```

Optimising the results using hyperparameters using random search CV and Grid search CV on two best models (Random forest and Gradient boosting-based on low rmse)

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

Optimising the results with parameter tuning

```

In [139]: from sklearn.ensemble import RandomForestRegressor
rf = RandomForestRegressor(random_state = 42)
from pprint import pprint
# Look at parameters used by our current forest
print('Parameters currently in use:\n')
pprint(rf.get_params())

```

Parameters currently in use:

```

{'bootstrap': True,
 'criterion': 'mse',
 'max_depth': None,
 'max_features': 'auto',
 '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': 'warn',
 'n_jobs': None,
 'oob_score': False,
 'random_state': 42,
 'verbose': 0,
 'warm_start': False}

```


1. Random search CV on Random forest model

```
##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)

#R^2
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 = {:.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 = 0.81.
RMSE = 0.2370753020902656
```

2. 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)

#R^2
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 = {:.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 = 0.78.
RMSE = 0.25693659513229167
```

3. Grid search CV on Random Forest

```
from sklearn.model_selection import GridSearchCV
## 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 = {:.2}'.format(GRF_r2))
print('RMSE = ',(GRF_rmse))

Grid Search CV Random Forest Regressor Model Performance:
Best Parameters = {'max_depth': 7, 'n_estimators': 17}
R-squared = 0.81.
RMSE = 0.23498889812429072
```

4. Grid search CV on Gradient Boosting model

```
]: ## Grid Search CV for gradient boosting
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 = {:.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 = 0.8.
RMSE = 0.2422018444250436
```

XG Boost in R

XGBoost is an implementation of gradient boosted decision trees designed for speed and performance. The XGBoost library implements the gradient boosting decision tree algorithm.

This algorithm goes by lots of different names such as gradient boosting, multiple additive regression trees, stochastic gradient boosting or gradient boosting machines.

Boosting is an ensemble technique where new models are added to correct the errors made by existing models. Models are added sequentially until no further improvements can be made. A popular example is the AdaBoost algorithm that weights data points that are hard to predict.

Gradient boosting is an approach where new models are created that predict the residuals or errors of prior models and then added together to make the final prediction. It is called gradient boosting because it uses a gradient descent algorithm to minimize the loss when adding new models.

This approach supports both regression and classification predictive modeling problems.

Improving Accuracy by using Ensemble technique ---- XGBOOST

```
train_data_matrix = as.matrix(sapply(train_data[-1],as.numeric))
```

```
test_data_data_matrix = as.matrix(sapply(test_data[-1],as.numeric))
```

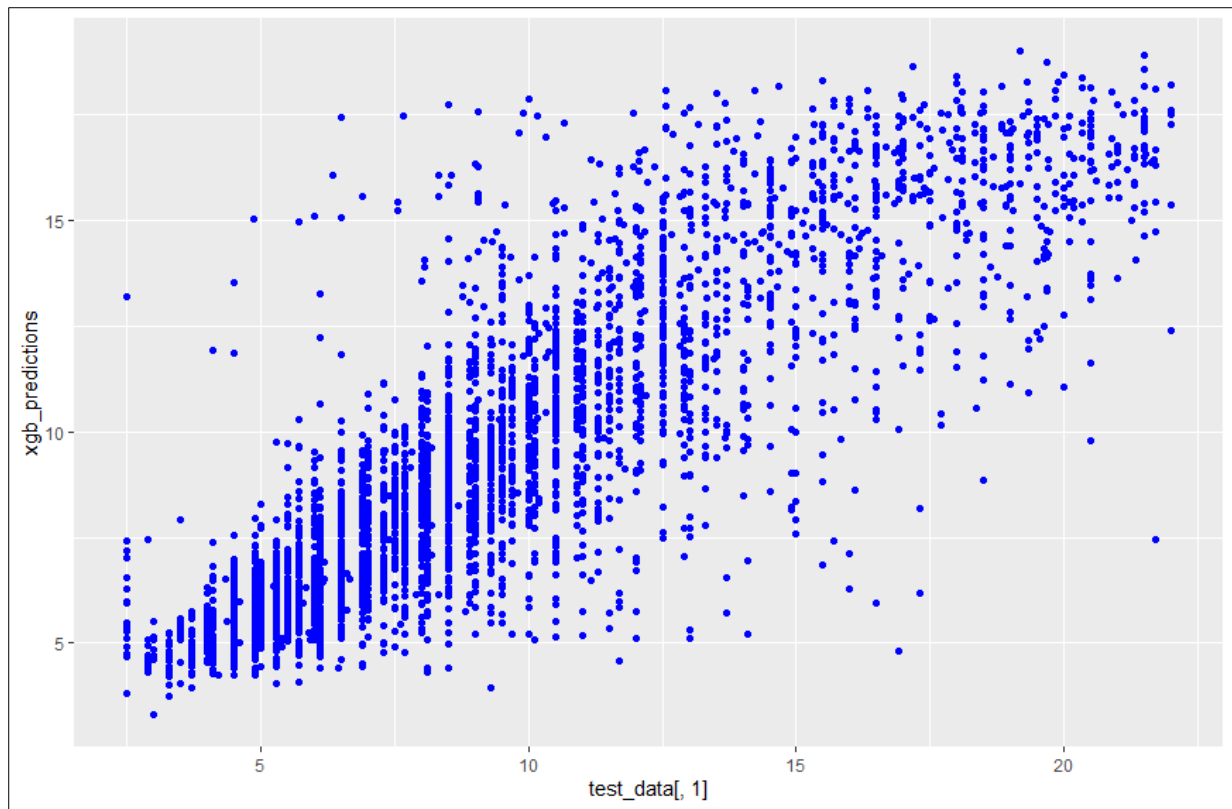
```
xgboost_model = xgboost(data = train_data_matrix,label = train_data$fare_amount,nrounds = 15,verbose = FALSE)
```

```
summary(xgboost_model) xgb_predictions = predict(xgboost_model,test_data_data_matrix)
```

```
qplot(x = test_data[,1], y = xgb_predictions, data = test_data, color = I("blue"), geom = "point")
```

```
regr.eval(test_data[,1],xgb_predictions)
```

# mae	mse	rmse	mape
# 1.6193378	5.2082001	2.2821481	0.1836901



Chapter 4

Conclusion

4.1 Model Evaluation

Now that we have built few models for predicting the target variable, we need to decide which one to choose. There are several criteria that exist for evaluating and comparing models. We can compare the models.

2. Predictive Performance
3. Interpretability
4. Computational Efficiency

In our case of Cab fare, the latter two, Interpretability and Computation Efficiency, do not hold much significance. Therefore, we will use Predictive performance as the criteria to compare and evaluate models. Predictive performance can be measured by comparing Predictions of the models with real values of the target variables and calculating some average error measure.

4.1.1 Root Mean Square Error

Root Mean Square Error (RMSE) is the standard deviation of the residuals (prediction errors). Residuals are a measure of how far from the regression line data points are, RMSE is a measure of how spread out these residuals are. In other words, it tells you how concentrated the data is around the line of best fit. Also, Since the errors are squared before they are averaged, the RMSE gives a relatively high weight to large errors. So, RMSE becomes more useful when large errors are particularly undesirable. So, Root Mean Square value seems like a perfect choice for our problem at hand.

4.1.2 R Squared(R^2)

R Squared(R^2) 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 coefficient of multiple determination for multiple regression. In other words, we can say it explains as to how much of the variance of the target variable is explained.

The results of both train and test data are shown to check whether our model is over fitting.

Results and Discussion

1. Explanation in Python'
2. Explanation in R

Model Evaluation in Python

Results of RMSE and R Squared on test and train data

Model	RMSE		R Squared	
	Train	Test	Train	Test
Multiple Linear Regression	0.27	0.24	0.74	0.80
Decision Trees	0.29	0.28	0.70	0.72
Random Forest	0.09	0.23	0.96	0.80
Gradient boosting	0.23	0.23	0.83	0.83

Results of RMSE and R squared on test and train dataset after hyper tuning the parameters

Model	Parameters	RMSE	R Squared
		Test	Test
Random Search CV	Random Forest	0.24	0.81
	Gradient boosting	0.25	0.78
Grid Search CV	Random Forest	0.23	0.81
	Gradient boosting	0.24	0.80

Above table shows the results after tuning the parameters of our 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.

Model Evaluation in R

Results of RMSE on train data

Model	RMSE	MAE	MSE
Multiple Linear Regression	4.37	3.50	19.12
Decision Trees	2.58	1.90	6.69
Random Forest	2.53	1.91	6.42
XG Boost	2.28	1.62	5.21

3.1 Model Selection

On the basis RMSE and R Squared results a good model should have least RMSE and max R Squared value. So, from above tables we can see:

- From the observation of all RMSE Value and R-Squared Value we have concluded that,
- Both the models- Gradient Boosting Default and Random Forest perform comparatively well while comparing their RMSE and R-Squared value.
- After this, I chose Random Forest CV and Grid Search CV to apply cross validation technique and see changes brought about by that.
- After applying tunings Random forest model shows best results compared to gradient boosting.
- So finally, we can say that **Random forest model** is the best method to make prediction for this project with highest explained variance of the target variables and lowest error chances with parameter tuning technique Grid Search CV. –In Python
- We choose Gradient boosting technique in R as RMSE value is low compared to the other models in R

References

1. <https://www.kaggle.com/>
2. <http://rprogramming.net/>
3. <https://medium.com/>
4. <https://stackoverflow.com/>
5. <https://www.rdocumentation.org/>
6. <https://www.analyticsvidhya.com/blog/>

Appendix – Complete R Code with Explanation

Exploratory data analysis

```
#Clear the environment
rm(list = ls())

#Set working directory
setwd("F:/EdwisorVanusha/Project/Cab Fare Prediction")
getwd()

#####Exploratory Data Analysis#####
#Importing Libraries
x = c("ggplot2", "corrgram", "DMwR", "usdm", "caret", "randomForest", "e1071",
      "DataCombine", "doSNOW", "inTrees", "rpart.plot", "rpart", 'MASS', 'xgboost', 'stats')

#load Packages
lapply(x, require, character.only = TRUE)
rm(x)

#Loading the train and test dataset
train = read.csv("train_cab.csv", header = T, na.strings = c(" ", "", "NA"))
test = read.csv("test.csv")

test_pickup_datetime = test["pickup_datetime"]

# Structure of data
str(train)
str(test)
#Loading the head of loaded dataframes
head(train,5)
head(test,5)

Findings: In the train dataset there are 16067 observations and 7 variables using which we have to predict the cab fare amount and the test data has 9914 observations and 6 variables

#Summary statistics of Train and Test dataset
summary(train)
summary(test)

#Findings:
#1.The minimal fare_amount is negative. maximum is USD 54,343 and median is USD 8.50, As this does not seem to be realistic, I will drop the negative values from the dataset.
#2.Some of the minimum and maximum longitude/latitude coordinates are way off. Latitudes range should be brought within a set boundaries
#3.Min passenger count is 0 and max is 208, both seem errorneous. Passenger count should not exceed 6(even if we consider SUV)
```

#1. Data Manipulation

Changing the data types of variables into required data types

```
train$fare_amount = as.numeric(as.character(train$fare_amount))
train$passenger_count=round(train$passenger_count)
```

#Removing values which are not within desired range(outlier) depending upon basic understanding of dataset.

1.Fare amount has a negative value, which doesn't make sense. A price amount cannot be -ve and also cannot be 0. So we will remove these fields.

```
train[which(train$fare_amount < 1 ),]
nrow(train[which(train$fare_amount < 1 ),])
train = train[-which(train$fare_amount < 1 ),]
```

#2.Passenger_count variable

```
for (i in seq(4,11,by=1)){
  print(paste('passenger_count above ',i,nrow(train[which(train$passenger_count > i ),])))
}
```

so 20 observations of passenger_count is consistently above from 6,7,8,9,10 passenger_counts, let's check them.

```
train[which(train$passenger_count > 6 ),]
# Also we need to see if there are any passenger_count==0
train[which(train$passenger_count <1 ),]
nrow(train[which(train$passenger_count <1 ),])
```

We will remove these 58 observations and 20 observation which are above 6 value because a cab cannot hold these number of passengers.

```
train = train[-which(train$passenger_count < 1 ),]
train = train[-which(train$passenger_count > 6),]
# There's only one outlier which is in variable pickup_latitude.So we will remove it with nan.
```

Also we will see if there are any values equal to 0.

```
nrow(train[which(train$pickup_longitude == 0 ),])
nrow(train[which(train$pickup_latitude == 0 ),])
nrow(train[which(train$dropoff_longitude == 0 ),])
nrow(train[which(train$pickup_latitude == 0 ),])
# there are values which are equal to 0. we will remove them.
train = train[-which(train$pickup_latitude > 90),]
train = train[-which(train$pickup_longitude == 0),]
train = train[-which(train$dropoff_longitude == 0),]
```

Make a copy

```
df=train
```

```
# train=df
```

#2. Missing Value Analysis

```
missing_val = data.frame(apply(train,2,function(x){sum(is.na(x))}))
missing_val$Columns = row.names(missing_val)
names(missing_val)[1] = "Missing_percentage"
```

```

missing_val$Missing_percentage = (missing_val$Missing_percentage/nrow(train)) * 100
missing_val = missing_val[order(-missing_val$Missing_percentage),]
row.names(missing_val) = NULL
missing_val = missing_val[,c(2,1)]
missing_val

```

#Findings:

#1. As we can see that in train dataset we find missing values in passenger_count-55 (0.34%) and fare_amount- 25 (0.15%).

#As the dataset is small with 16k observations, if we delete the missing values, it may probably have effect on the model being trained, so here we will impute the missing values using mean method, KNN and median method for a sample and the method with value that is closer to the actual value will be used for imputation on the train dataset

#Plot of number of missing values in the train dataset

```

ggplot(data = missing_val[1:3,], aes(x=reorder(Columns, -Missing_percentage), y =
Missing_percentage))+
  geom_bar(stat = "identity", fill = "grey")+xlab("Parameter")+
  ggtitle("Missing data percentage") + theme_bw()

```

```

unique(train$passenger_count)
unique(test$passenger_count)
train[, 'passenger_count'] = factor(train[, 'passenger_count'], labels=(1:6))
test[, 'passenger_count'] = factor(test[, 'passenger_count'], labels=(1:6))
# 1.For Passenger_count:
# Actual value = 1
# Mode = 1
# KNN = 1
train$passenger_count[1000]
train$passenger_count[1000] = NA
getmode <- function(v) {
  uniqv <- unique(v)
  uniqv[which.max(tabulate(match(v, uniqv)))]
}

```

Mode Method

```

getmode(train$passenger_count)
# We can't use mode method because data will be more biased towards passenger_count=1

```

2.For fare_amount:

```

# Actual value = 18.1,
# Mean = 15.117,
# Median = 8.5,
# KNN = 18.57
sapply(train, sd, na.rm = TRUE)

```

```

train$fare_amount[1000]
train$fare_amount[1000]= NA

```

Mean Method

```

mean(train$fare_amount, na.rm = T)

#Median Method
median(train$fare_amount, na.rm = T)

# kNN Imputation
train = knnImputation(train, k = 181)
train$fare_amount[1000]
train$passenger_count[1000]
sapply(train, sd, na.rm = TRUE)
sum(is.na(train))
str(train)
summary(train)

df1=train

#3.Outlier Analysis

# We Will do Outlier Analysis only on Fare_amount just for now and we will do outlier
analysis after feature engineering latitudes and longitudes.

# Boxplot for fare_amount
p11 = ggplot(train,aes(x = factor(passenger_count),y = fare_amount))
p11 + geom_boxplot(outlier.colour="red", fill = "grey",outlier.shape=18,outlier.size=1,
notch=FALSE)+ylim(0,100)

# Replace all outliers with NA and impute
vals = train[, "fare_amount"] %in% boxplot.stats(train[, "fare_amount"])$out
train[which(vals), "fare_amount"] = NA

#lets check the NA's
sum(is.na(train$fare_amount))

#Imputing with KNN
train = knnImputation(train,k=3)

# lets check the missing values
sum(is.na(train$fare_amount))
str(train)

df2=train
# train=df2

##### Feature Engineering #####
#Convert pickup_datetime from factor to date time
train$pickup_date = as.Date(as.character(train$pickup_datetime))
train$pickup_weekday = as.factor(format(train$pickup_date,"%u"))# Monday = 1
train$pickup_mnth = as.factor(format(train$pickup_date,"%m"))
train$pickup_yr = as.factor(format(train$pickup_date,"%Y"))
pickup_time = strptime(train$pickup_datetime,"%Y-%m-%d %H:%M:%S")
train$pickup_hour = as.factor(format(pickup_time,"%H"))

```

#Add same features to test set

```
test$pickup_date = as.Date(as.character(test$pickup_datetime))
test$pickup_weekday = as.factor(format(test$pickup_date,"%u"))# Monday = 1
test$pickup_mnth = as.factor(format(test$pickup_date,"%m"))
test$pickup_yr = as.factor(format(test$pickup_date,"%Y"))
pickup_time = strptime(test$pickup_datetime,"%Y-%m-%d %H:%M:%S")
test$pickup_hour = as.factor(format(pickup_time,"%H"))
```

sum(is.na(train))# there was 1 'na' in pickup_datetime which created na's in above feature engineered variables.

train = na.omit(train) # we will remove that 1 row of na's

```
train = subset(train,select = -c(pickup_datetime,pickup_date))
```

```
test = subset(test,select = -c(pickup_datetime,pickup_date))
```

2.Calculate the distance travelled using longitude and latitude

```
deg_to_rad = function(deg){
```

```
  (deg * pi) / 180
```

```
}
```

```
haversine = function(long1,lat1,long2,lat2){
```

```
  #long1rad = deg_to_rad(long1)
```

```
  phi1 = deg_to_rad(lat1)
```

```
  #long2rad = deg_to_rad(long2)
```

```
  phi2 = deg_to_rad(lat2)
```

```
  delphi = deg_to_rad(lat2 - lat1)
```

```
  dellamda = deg_to_rad(long2 - long1)
```

```
  a = sin(delphi/2) * sin(delphi/2) + cos(phi1) * cos(phi2) *
```

```
    sin(dellamda/2) * sin(dellamda/2)
```

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

```
  R = 6371e3
```

```
  R * c / 1000 #1000 is used to convert to meters
```

```
}
```

```
# Using haversine formula to calculate distance fr both train and test
```

```
train$dist =
```

```
haversine(train$pickup_longitude,train$pickup_latitude,train$dropoff_longitude,train$dropoff_latitude)
```

```
test$dist =
```

```
haversine(test$pickup_longitude,test$pickup_latitude,test$dropoff_longitude,test$dropoff_latitude)
```

We will remove the variables which were used to feature engineer new variables

```
train = subset(train,select = -
```

```
c(pickup_longitude,pickup_latitude,dropoff_longitude,dropoff_latitude))
```

```
test = subset(test,select = -
```

```
c(pickup_longitude,pickup_latitude,dropoff_longitude,dropoff_latitude))
```

```
str(train)
```

```

summary(train)

##### Feature selection #####
numeric_index = sapply(train,is.numeric) #selecting only numeric

numeric_data = train[,numeric_index]

cnames = colnames(numeric_data)

#Correlation analysis for numeric variables
corrgram(train[,numeric_index],upper.panel=panel.pie, main = "Correlation Plot")

#ANOVA for categorical variables with target numeric variable

#aov_results = aov(fare_amount ~ passenger_count * pickup_hour * pickup_weekday,data =
train)
aov_results = aov(fare_amount ~ passenger_count + pickup_hour + pickup_weekday +
pickup_mnth + pickup_yr,data = train)

summary(aov_results)

# pickup_weekday has p value greater than 0.05
train = subset(train,select=-pickup_weekday)

#remove from test set
test = subset(test,select=-pickup_weekday)

#4.Feature Scaling

#Normality check
qqnorm(train$fare_amount)
histogram(train$fare_amount)
library(car)
# dev.off()
par(mfrow=c(1,2))
qqPlot(train$fare_amount)
# qqPlot, it has a x values derived from gaussian distribution, if data is distributed normally
then the sorted data points should lie very close to the solid reference line
truehist(train$fare_amount)
# truehist() scales the counts to give an estimate of the probability density.
lines(density(train$fare_amount))
# Right skewed
# lines() and density() functions to overlay a density plot on histogram

#Normalisation

print('dist')
train[, 'dist'] = (train[, 'dist'] - min(train[, 'dist']))/
(max(train[, 'dist'] - min(train[, 'dist'])))

```

```

# #check multicollarity
library(usdm)
vif(train[,-1])
vifcor(train[,-1], th = 0.9)

# Splitting train into train and validation subsets ##
set.seed(1000)
tr.idx = createDataPartition(train$fare_amount,p=0.75,list = FALSE) # 75% in trainin and 25%
in Validation Datasets
train_data = train[tr.idx,]
test_data = train[-tr.idx,]

rmExcept(c("test","train","df","df1","df2","df3","test_data","train_data","test_pickup_datetime"))

##Model Development#####

#1Multiple Linear Regression

#Fit the model
lm_model = lm(fare_amount ~.,data=train_data)

summary(lm_model)
str(train_data)
plot(lm_model$fitted.values,rstandard(lm_model),main = "Residual plot",
      xlab = "Predicted values of fare_amount",
      ylab = "standardized residuals")

lm_predictions = predict(lm_model,test_data[,2:6])

qplot(x = test_data[,1], y = lm_predictions, data = test_data, color = I("blue"), geom = "point")

regr.eval(test_data[,1],lm_predictions)

# mae    mse    rmse    mape
# 3.5022600 19.1274501 4.3734940 0.4503374

#2.Decision Tree

#Fit the model
Dt_model = rpart(fare_amount ~ ., data = train_data, method = "anova")

#Summary of the model
summary(Dt_model)

#Predict for new test cases
predictions_DT = predict(Dt_model, test_data[,2:6])

qplot(x = test_data[,1], y = predictions_DT, data = test_data, color = I("blue"), geom = "point")

```

```

regr.eval(test_data[,1],predictions_DT)
# mae    mse    rmse    mape
#1.9001012 6.6918569 2.5868624 0.2192849

#3.Random forest #

#Fit the model
rf_model = randomForest(fare_amount ~.,data=train_data)
#summary of the model
summary(rf_model)
#predict the model on test data
rf_predictions = predict(rf_model,test_data[,2:6])

qplot(x = test_data[,1], y = rf_predictions, data = test_data, color = I("blue"), geom = "point")

regr.eval(test_data[,1],rf_predictions)
# mae    mse    rmse    mape
#1.9103915 6.4214597 2.5340599 0.2342449

## Improving Accuracy by using Ensemble technique ---- XGBOOST #

train_data_matrix = as.matrix(sapply(train_data[-1],as.numeric))
test_data_data_matrix = as.matrix(sapply(test_data[-1],as.numeric))

xgboost_model = xgboost(data = train_data_matrix,label = train_data$fare_amount,nrounds =
15,verbose = FALSE)

summary(xgboost_model)
xgb_predictions = predict(xgboost_model,test_data_data_matrix)

qplot(x = test_data[,1], y = xgb_predictions, data = test_data, color = I("blue"), geom =
"point")

regr.eval(test_data[,1],xgb_predictions)

# mae    mse    rmse    mape
# 1.6193378 5.2082001 2.2821481 0.1836901

```

As rmse value for XGBoost model is 2.28 which is lower than all other models so we select

Finalizing and Saving Model##

In this step we will train our model on whole training Dataset and save that model for later use

```

train_data_matrix2 = as.matrix(sapply(train[-1],as.numeric))
test_data_matrix2 = as.matrix(sapply(test,as.numeric))

xgboost_model2 = xgboost(data = train_data_matrix2,label = train$fare_amount,nrounds =
15,verbose = FALSE)

```


Saving the trained model

```
saveRDS(xgboost_model2, "/final_Xgboost_model_using_R.rds")
```

loading the saved model

```
super_model <- readRDS("/final_Xgboost_model_using_R.rds")  
print(super_model)
```

Lets now predict on test dataset

```
xgb = predict(super_model, test_data_matrix2)
```

```
xgb_pred = data.frame(test_pickup_datetime, predictions = xgb)
```

Now lets write(save) the predicted fare_amount in disk as .csv format

```
write.csv(xgb_pred, "xgb_predictions_R.csv", row.names = FALSE)
```

Python

Background

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.

The objective of this project is to predict the fare amount for the taxi ride given the pickup and drop off locations

Exploratory Data Analysis

#

Exploratory data analysis is the first step where we explore and understand the data. In this section, we will load the data into our analysis environment and explore its properties. EDA is one of the most important phases in the whole workflow and can help with not just understanding the dataset, but also in presenting certain fine points that can be useful in the coming steps

Import libraries

Ignore the warnings

```
import warnings
```

```
warnings.filterwarnings('always')
```

```
warnings.filterwarnings('ignore')
```

data visualisation and manipulation

```
import os #getting access to input files
```

```
import pandas as pd # Importing pandas for performing EDA
```

```
import numpy as np # Importing numpy for Linear Algebraic operations
```

```
import matplotlib.pyplot as plt # Importing for Data Visualization
```

```
import matplotlib as plt
```

```
get_ipython().run_line_magic('matplotlib', 'inline')
```

```
from matplotlib import style
```

```

from scipy import stats
import seaborn as sns # Importing for Data Visualization
plt.style.use('seaborn-whitegrid')

# configure
# sets matplotlib to inline and displays graphs below the corresponding cell.
get_ipython().run_line_magic('matplotlib', 'inline')
style.use('fivethirtyeight')
sns.set(style='whitegrid',color_codes=True)

# import the necessary modelling algos.

from sklearn.decomposition import PCA
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from math import sqrt
from sklearn.neighbors import KNeighborsRegressor
from sklearn.model_selection import train_test_split,RandomizedSearchCV
from sklearn.ensemble import RandomForestRegressor
from collections import Counter
import statsmodels.api as sm
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import GradientBoostingRegressor
from pprint import pprint
from sklearn.model_selection import GridSearchCV

#evaluation metrics
from sklearn.metrics import r2_score
from sklearn.metrics import mean_squared_error

# ## Preprocessing
#
# EDA process begins with loading of data into the environment, getting quick look at it along
with count of records and number of attributes. We will be making heavy use of pandas and
numpy to perform data manipulation and related tasks. For visualisation purposes, we will use
matplotlib and seaborn along with pandas visualisation capabilities wherever possible

# ##### Set working directory
os.chdir("D:\EdwisorVanusha\Project\Cab Fare Prediction")
os.getcwd()

# ##### Load dataset
# We begin with loading the train_cab.csv and test.csv and checking the shape of loaded
dataframe

```

```

train_cab = pd.read_csv("train_cab.csv",na_values={"pickup_datetime":"43"})
test_cab = pd.read_csv ("test.csv")

#Shape of the train dataset
train_cab.shape

#Shape of the test dataset
test_cab.shape

# In the train dataset there are 16067 observations and 7 variables using which we have to
predict the cab fare amount and the test data has 9914 observations and 6 variables

# ##### Checking the head of the dataframes
# Lets check the top few rows to see how the data looks. We use the head() utility from pandas
for the same to get the output

train_cab.head(5)

# ##### The details of data attributes in the dataset are as follows:
# 1. pickup_datetime - timestamp value indicating when the cab ride started.
# 2. pickup_longitude - float for longitude coordinate of where the cab ride started.
# 3. pickup_latitude - float for latitude coordinate of where the cab ride started.
# 4. dropoff_longitude - float for longitude coordinate of where the cab ride ended.
# 5. dropoff_latitude - float for latitude coordinate of where the cab ride ended.
# 6. passenger_count - an integer indicating the number of passengers in the cab ride.

test_cab.head(5)

# Well, we see fare amount, pickup_datetime and latitude and longitude information. Let's take
look into each of the following features. But, we need to check what data types pandas has
inferred and if any of the attributes require type conversions. The following snippet helps us
check the data types of all attributes

#### check datatypes
train_cab.dtypes

# ##### Findings:
# 1. The attribute datetime would require type conversion from object (or String type) to
timestamp

# 2. The attribute fare_amount is inferred as object by pandas, and they would require
conversion to numeric for proper understanding

# ##### Descriptive statistics
#check statistics of the features
train_cab.describe()

```

Findings:

There are 7 variables in the dataset.

From the summary of the dataset, we can see:

1. The minimal fare_amount is negative. maximum is USD 54,343 and median is USD 8.50, As this does not seem to be realistic, I will drop the negative values from the dataset.

2. Some of the minimum and maximum longitude/latitude coordinates are way off. Latitudes range should be brought within a set boundaries

3. Min passenger count is 0 and max is 208, both seem erroroneus. Passenger count should not exceed 6(even if we consider SUV)

check datatypes

test_cab.dtypes

#check statistics of the features

test_cab.describe()

Pandas_profiling

Pandas Profiling is python package which is a simple and fast way to perform exploratory data analysis of a Pandas Dataframe.

import pandas_profiling as pp

pfr=pp.ProfileReport(train_cab)

pfr

view profile report generated in the saved repository as a html file

pfr.to_file("profile.html")

Data Cleaning and Missing Value Analysis

#Convert fare_amount from object to numeric

train_cab["fare_amount"] = pd.to_numeric(train_cab["fare_amount"],errors = "coerce")

#Using errors='coerce'. It will replace all non-numeric values with NaN.

train_cab.dtypes

train_cab.shape

train_cab.dropna(subset= ["pickup_datetime"]) #dropping NA values in datetime column

Here pickup_datetime variable is in object so we need to change its data type to datetime

train_cab['pickup_datetime'] = pd.to_datetime(train_cab['pickup_datetime'], format='%Y-%m-%d %H:%M:%S UTC')

we will saperate the Pickup_datetime column into separate field like year, month, day of the week, etc

```

train_cab['year'] = train_cab['pickup_datetime'].dt.year
train_cab['Month'] = train_cab['pickup_datetime'].dt.month
train_cab['Date'] = train_cab['pickup_datetime'].dt.day
train_cab['Day'] = train_cab['pickup_datetime'].dt.dayofweek
train_cab['Hour'] = train_cab['pickup_datetime'].dt.hour
train_cab['Minute'] = train_cab['pickup_datetime'].dt.minute

train_cab.dtypes #Re-checking datatypes after conversion

test_cab["pickup_datetime"] = pd.to_datetime(test_cab["pickup_datetime"],format= "%Y-%m-%d %H:%M:%S UTC")
#### we will saperate the Pickup_datetime column into separate field like year, month, day of
the week, etc

test_cab['year'] = test_cab['pickup_datetime'].dt.year
test_cab['Month'] = test_cab['pickup_datetime'].dt.month
test_cab['Date'] = test_cab['pickup_datetime'].dt.day
test_cab['Day'] = test_cab['pickup_datetime'].dt.dayofweek
test_cab['Hour'] = test_cab['pickup_datetime'].dt.hour
test_cab['Minute'] = test_cab['pickup_datetime'].dt.minute

test_cab.dtypes #Re-checking test datatypes after conversion

# Checking the Datetime Variable :

#removing datetime missing values rows
train_cab = train_cab.drop(train_cab[train_cab['pickup_datetime'].isnull()].index, axis=0)
print(train_cab.shape)
print(train_cab['pickup_datetime'].isnull().sum())

train_cab["passenger_count"].describe()

# We can see maximum number of passanger count is 5345 which is actually not possible. So
reducing the passenger count to 6 (even if we consider the SUV)

train_cab = train_cab.drop(train_cab[train_cab["passenger_count"]> 6 ].index, axis=0)

#Also removing the values with passenger count of 0.
train_cab = train_cab.drop(train_cab[train_cab["passenger_count"] == 0 ].index, axis=0)

train_cab["passenger_count"].describe()

# Next checking the Fare Amount variable :
train_cab = train_cab.drop(train_cab[train_cab["passenger_count"] == 0.12 ].index, axis=0)

```

```

train_cab.shape

##finding decending order of fare to get to know whether the outliers are present or not
train_cab["fare_amount"].sort_values(ascending=False)

## The fare amount column is having some negative values, Lets Check it
print(Counter(train_cab['fare_amount']<0))
print(Counter(train_cab['fare_amount']>1000))

#Also remove the row where fare amount is zero
train_cab = train_cab.drop(train_cab[train_cab["fare_amount"]<1].index, axis=0)
train_cab.shape

#Now we can see that there is a huge difference in 1st 2nd and 3rd position in decending order
of fare amount
# so we will remove the rows having fare amounting more that 454 as considering them as
outliers

train_cab = train_cab.drop(train_cab[train_cab["fare_amount"]> 1000 ].index, axis=0)
train_cab.shape

train_cab["fare_amount"].describe()

# Now checking the pickup latitude and longitude :

#Latitude----(-90 to 90)
#Longitude----(-180 to 180)

# we need to drop the rows having pickup latitute and longitute out the range mentioned
above

#train = train.drop(train[train['pickup_latitude']<-90])
train_cab[train_cab['pickup_latitude']<-90]
train_cab[train_cab['pickup_latitude']>90]

#Hence dropping one value of >90
train_cab = train_cab.drop((train_cab[train_cab['pickup_latitude']<-90]).index, axis=0)
train_cab = train_cab.drop((train_cab[train_cab['pickup_latitude']>90]).index, axis=0)

train_cab[train_cab['pickup_longitude']<-180]
train_cab[train_cab['pickup_longitude']>180]

train_cab[train_cab['dropoff_latitude']<-90]
train_cab[train_cab['dropoff_latitude']>90]

```

```

train_cab[train_cab['dropoff_longitude']<-180]
train_cab[train_cab['dropoff_longitude']>180]

train_cab.shape

# ## Missing Value Analysis
# In statistics, missing data, or 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.

#Create dataframe with missing percentage
missing_val = pd.DataFrame(train_cab.isnull().sum())
#Reset index
missing_val = missing_val.reset_index()
missing_val

# ### Findings:
# As we can see that in train dataset we find missing values in
#
# 1.passenger_count-55 (0.34%) and
#
# 2.fare_amount- 25 (0.15%).
#
# As the dataset is small with 16k observations, if we delete the missing values, it may
probably has effect on the model being trained, so here we will impute the missing values
using mean method and median method for a sample and the method with value that is closer
to the actual value will be used for imputation on the train dataset

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

# ##### Imputation using mode, mean and median (categorical=mode,
numerical=mean/median)

# 1.For Passenger_count:
# Actual value = 1 Mode = 1

```

```

# Choosing a random values to replace it as NA
train_cab['passenger_count'].loc[1000]

# Replacing 1.0 with NA
train_cab['passenger_count'].loc[1000] = np.nan
train_cab['passenger_count'].loc[1000]

# Impute with mode
train_cab['passenger_count'].fillna(train_cab['passenger_count'].mode()[0]).loc[1000]

# We can't use mode method because data will be more biased towards passenger_count=1

# ##### 2.For fare_amount:
#
# Actual value = 7.0,
# Mean = 15.117,
# Median = 8.5,

# Choosing a random values to replace it as NA
a=train_cab['fare_amount'].loc[1000]
print('fare_amount at loc-1000:{ }'.format(a))
# Replacing 1.0 with NA
train_cab['fare_amount'].loc[1000] = np.nan
print('Value after replacing with nan:{ }'.format(train_cab['fare_amount'].loc[1000]))
# Impute with mean
print('Value if imputed with
mean:{ }'.format(train_cab['fare_amount'].fillna(train_cab['fare_amount'].mean()).loc[1000]))
# Impute with median
print('Value if imputed with
median:{ }'.format(train_cab['fare_amount'].fillna(train_cab['fare_amount'].median()).loc[1000]
))

# As we can say that the median value has given values with great approximation, we will go
ahead by using median method to impute

# ##### Imputation of missing values with median method

def missin_val(df):
    total = df.isnull().sum().sort_values(ascending=False)
    percent = (df.isnull().sum() * 100 /df.isnull().count()).sort_values(ascending=False)
    missing_data = pd.concat([total, percent], axis=1, keys=['Total', 'Percent'])
    return(missing_data)

train_cab["passenger_count"] =
train_cab["passenger_count"].fillna(train_cab["passenger_count"].median())

```



```

train_cab["fare_amount"] =
train_cab["fare_amount"].fillna(train_cab["fare_amount"].median())

print("missing values after imputation using median method:\n\n",missin_val(train_cab))
print("\n")

# ##### Now we have successfully cleared our both datasets. Thus proceeding for further
operations:

# Calculating distance based on the given coordinates :
#As we know that we have given pickup longitude and latitude values and same for drop.
#So we need to calculate the distance Using the haversine formula and we will create a new
variable called distance
from math import radians, cos, sin, 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 * asin(sqrt(a))
    # Radius of earth in kilometers is 6371
    km = 6371* c
    return km
# 1min

train_cab['distance'] =
train_cab[['pickup_longitude','pickup_latitude','dropoff_longitude','dropoff_latitude']].apply(ha
versine,axis=1)

test_cab['distance'] =
test_cab[['pickup_longitude','pickup_latitude','dropoff_longitude','dropoff_latitude']].apply(hav
ersine,axis=1)

```

```

train_cab.head()
test_cab.head()
train_cab.nunique()
test_cab.nunique()

##finding decending order of fare to get to know whether the outliers are presented or not
train_cab['distance'].sort_values(ascending=False)

# As we can see that top 23 values in the distance variables are very high It means more than
8000 Kms distance they have travelled
# Also just after 23rd value from the top, the distance goes down to 127, which means these
values are showing some outliers

# We need to remove these values
#
Counter(train_cab['distance'] == 0)
Counter(test_cab['distance'] == 0)
Counter(train_cab['fare_amount'] == 0)

###we will remove the rows whose distance value is zero

train_cab = train_cab.drop(train_cab[train_cab['distance']== 0].index, axis=0)
train_cab.shape

#we will remove the rows whose distance values is very high which is more than 129kms
train_cab = train_cab.drop(train_cab[train_cab['distance'] > 130 ].index, axis=0)
train_cab.shape
train_cab.head()

# Now we have splitted the pickup date time variable into different varaibles like month, year,
day etc so now we dont need to have that pickup_Date variable now. Hence we can drop that,
Also we have created distance using pickup and drop longitudes and latitudes so we will also
drop pickup and drop longitudes and latitudes variables.

drop = ['pickup_datetime', 'pickup_longitude', 'pickup_latitude', 'dropoff_longitude',
'dropoff_latitude', 'Minute']
train_cab = train_cab.drop(drop, axis = 1)

train_cab.head()

train_cab['passenger_count'] = train_cab['passenger_count'].astype('int64')
train_cab['year'] = train_cab['year'].astype('int64')
train_cab['Month'] = train_cab['Month'].astype('int64')
train_cab['Date'] = train_cab['Date'].astype('int64')
train_cab['Day'] = train_cab['Day'].astype('int64')
train_cab['Hour'] = train_cab['Hour'].astype('int64')

```

```
train_cab.dtypes
```

```
drop_test = ['pickup_datetime', 'pickup_longitude', 'pickup_latitude', 'dropoff_longitude',  
'dropoff_latitude', 'Minute']
```

```
test_cab = test_cab.drop(drop_test, axis = 1)
```

```
test_cab.head()
```

```
test_cab.dtypes
```

```
df=train_cab
```

Outlier Analysis

While exploring and learning about any dataset, it is imperative we check for extreme and unlikely values. Though we handle missing and incorrect information while preprocessing the dataset, outliers are usually caught during EDA. Outliers can severely and adversely impact the downstream steps like modelling and the results.

We usually utilize boxplots to check outliers in the data. In the following snippet, we analyze outliers for numeric attributes

```
import matplotlib.pyplot as plt
```

```
df.plot(kind='box', subplots=True, layout=(8,3), sharex=False, sharey=False, fontsize=8)
```

```
plt.subplots_adjust(left=0.125, bottom=0.1, right=0.9, top= 3, wspace=0.2, hspace=0.2)
```

```
plt.show()
```

In Box plots analysis of individual features, we can clearly observe from these boxplots that, not every feature contains outliers and many of them even have very few outliers. We have only 16k data-points and after removing the outliers, the data gets decreased by almost 15%. So, dropping the outliers is probably not the best idea. Instead we will try to visualise and find out the outliers using box plots and will fill them with NA, that means we have created 'missing values' in place of outliers within the data. Now, we can treat these outliers like missing values and impute them using standard imputation techniques. In our case, we use Median imputation to impute these missing values.

##Detect and delete outliers from data

```
def outliers_analysis(df):
```

```
    for i in df.columns:
```

```
        print(i)
```

```
        q75, q25 = np.percentile(df.loc[:,i], [75 ,25])
```

```
        iqr = q75 - q25
```

```
        min = q25 - (iqr*1.5)
```

```
        max = q75 + (iqr*1.5)
```

```
        print(min)
```

```
        print(max)
```

```

df = df.drop(df[df.loc[:,i] < min].index)
df = df.drop(df[df.loc[:,i] > max].index)
return(df)

def eliminate_rows_with_zero_value(df):
    df= df[df!= 0]
    df=df.fillna(df.median())
    return(df)

df.shape
train_cab = outliers_analysis(train_cab)
train_cab = eliminate_rows_with_zero_value(train_cab)

train_cab.plot(kind='box', subplots=True, layout=(8,3), sharex=False, sharey=False,
fontsize=8)
plt.subplots_adjust(left=0.125, bottom=0.1, right=0.9, top= 3,wspace=0.2, hspace=0.2)
plt.show()

# ##### Outlier analysis in test data

test_cab.plot(kind='box', subplots=True, layout=(8,3), sharex=False, sharey=False, fontsize=8)
plt.subplots_adjust(left=0.125, bottom=0.1, right=0.9, top= 3,wspace=0.2, hspace=0.2)
plt.show()

test_cab = outliers_analysis(test_cab)
test_cab = eliminate_rows_with_zero_value(test_cab)

test_cab.plot(kind='box', subplots=True, layout=(8,3), sharex=False, sharey=False, fontsize=8)
plt.subplots_adjust(left=0.125, bottom=0.1, right=0.9, top= 3,wspace=0.2, hspace=0.2)
plt.show()

train_cab=df

# ### Data Visualisation
#
# Data visualisation helps us to get better insights of the data. By visualising data, we can
identify areas that need attention or improvement and also clarifies which factors influence fare
of the cab and how the resources are used to determine it.

# ##### Visualization of following:
#
# 1. Number of Passengers effects the the fare
# 2. Pickup date and time effects the fare
# 3. Day of the week does effects the fare

```

```

# 4. Distance effects the fare

# ### Univariate analysis
#
# Univariate analysis is the simplest form of data analysis where the data being analysed
contains only one variable. Since it's a single variable it doesn't deal with causes or
relationships. The main purpose of univariate analysis is to describe the data and find patterns
that exist within it. So, Lets have a look at histogram plot, to identify the characteristic of the
features and the data.

# ### Bivariate analysis
# In bivariate analysis, we will look at the relationship between target variable and predictor.

# Count plot on passenger count
plt.figure(figsize=(15,7))
sns.countplot(x="passenger_count", data=train_cab)

#Relationship between number of passengers and Fare

plt.figure(figsize=(15,7))
plt.scatter(x=train_cab['passenger_count'], y=train_cab['fare_amount'], s=10)
plt.xlabel('No. of Passengers')
plt.ylabel('Fare')
plt.show()

# ##### Findings :
# By seeing the above plots we can easily conclude that:
#
# 1. single travelling passengers are most frequent travellers.
#
# 2. At the sametime we can also conclude that highest Fare are coming from single & double
travelling passengers.

#Relationship between date and Fare
plt.figure(figsize=(15,7))
plt.scatter(x=train_cab['Date'], y=train_cab['fare_amount'], s=10)
plt.xlabel('Date')
plt.ylabel('Fare')
plt.show()

plt.figure(figsize=(15,7))
train_cab.groupby(train_cab["Hour"])[ 'Hour'].count().plot(kind="bar")
plt.show()

# Lowest cabs at 5 AM and highest at and around 7 PM i.e the office rush hours

```

```
#Relationship between Time and Fare
```

```
plt.figure(figsize=(15,7))
```

```
plt.scatter(x=train_cab['Hour'], y=train_cab['fare_amount'], s=10)
```

```
plt.xlabel('Hour')
```

```
plt.ylabel('Fare')
```

```
plt.show()
```

```
# From the above plot We can observe that the cabs taken at 7 am and 23 Pm are the costliest.
```

```
# Hence we can assume that cabs taken early in morning and late at night are costliest
```

```
#
```

```
#impact of Day on the number of cab rides
```

```
plt.figure(figsize=(15,7))
```

```
sns.countplot(x="Day", data=train_cab)
```

```
# Observation :
```

```
# The day of the week does not seem to have much influence on the number of cabs ride
```

```
#Relationships between day and Fare
```

```
plt.figure(figsize=(15,7))
```

```
plt.scatter(x=train_cab['Day'], y=train_cab['fare_amount'], s=10)
```

```
plt.xlabel('Day')
```

```
plt.ylabel('Fare')
```

```
plt.show()
```

```
# The highest fares seem to be on a Sunday, Monday and Thursday, and the low on Wednesday and Saturday. May be due to low demand of the cabs on Saturdays the cab fare is low and high demand of cabs on Sunday and Monday shows the high fare prices
```

```
#Relationship between distance and fare
```

```
plt.figure(figsize=(15,7))
```

```
plt.scatter(x = train_cab['distance'], y = train_cab['fare_amount'], c = "g")
```

```
plt.xlabel('Distance')
```

```
plt.ylabel('Fare')
```

```
plt.show()
```

```
# ### Correlation Analysis
```

```
# Correlation helps us understand relationships between different attributes of the data. Since we have to predict the fare_amount in this project (Focus is on forecasting), so correlations helps us understand and exploit relationships to build better models
```

```
##Correlation analysis
```

```
#Correlation plot
```

```
def Correlation(df):
```

```

df_corr = df.loc[:,df.columns]
sns.set()
plt.figure(figsize=(9, 9))
corr = df_corr.corr()
sns.heatmap(corr, annot= True,fmt = " .3f", linewidths = 0.5,
            square=True)

Correlation(train_cab)
Correlation(test_cab)

# 1. The value for collinearity is between -1 to 1. So, any value close to -1/1 will result in high
collinearity.
# 2. It seems all right in our train and test data the situations nothing is more than 0.4 in
positive direction and nothing is less than -0.1 in negative direction.
#
# Therefore the datasets are free from collinearity problem

# ## Feature Scaling
#
# Using the raw values as input features might make models biased toward features having
really high magnitude values. These models are typically sensitive to the magnitude or scale of
features like linear or logistic regression. Therefore it is recommended to normalize and scale
down the features with feature scaling. Here we use Normalisation technique as the train & test
_dataset are not normally distributed

# ### Normalisation
# Normalization rescales the values into a range of [0,1]. This might be useful in some cases
where all parameters need to have the same positive scale.

#Normality check of training data is uniformly distributed or not-

for i in ['fare_amount', 'distance']:
    print(i)
    sns.distplot(train_cab[i],bins='auto',color='green')
    plt.title("Distribution for Variable "+i)
    plt.ylabel("Density")
    plt.show()

#since skewness of target variable is high, apply log transform to reduce the skewness-
train_cab['fare_amount'] = np.log1p(train_cab['fare_amount'])

#since skewness of distance variable is high, apply log transform to reduce the skewness-
train_cab['distance'] = np.log1p(train_cab['distance'])

#Normality Re-check to check data is uniformly distributed or not after log transformation

```

```

for i in ['fare_amount', 'distance']:
    print(i)
    sns.distplot(train_cab[i],bins='auto',color='green')
    plt.title("Distribution for Variable "+i)
    plt.ylabel("Density")
    plt.show()

# Here we can see bell shaped distribution. Hence our continuous variables are now normally
distributed, we will use not use any Feature Scalling technique. i.e, Normalization or
Standardization for our training data

#Normality check for test data is uniformly distributed or not-

sns.distplot(test_cab['distance'],bins='auto',color='green')
plt.title("Distribution for Variable "+i)
plt.ylabel("Density")
plt.show()

#since skewness of distance variable is high, apply log transform to reduce the skewness-
test_cab['distance'] = np.log1p(test_cab['distance'])

#rechecking the distribution for distance
sns.distplot(test_cab['distance'],bins='auto',color='green')
plt.title("Distribution for Variable "+i)
plt.ylabel("Density")
plt.show()

# As we can see a bell shaped distribution. Hence our continuous variables are now normally
distributed, we will use not use any Feature Scalling technique. i.e, Normalization or
Standardization for our test data

## Evaluating Regression Models

# Predictive performance can be measured by comparing Predictions of the models with real
values of the target variables, and calculating some average error measure.

##### Root Mean Square Value
# Root Mean Square Error (RMSE) is the standard deviation of the residuals (prediction
errors). Residuals are a measure of how far from the regression line data points are, RMSE is a
measure of how spread out these residuals are. In other words, it tells you how concentrated the
data is around the line of best fit. Also, Since the errors are squared before they are averaged,
the RMSE gives a relatively high weight to large errors. So, RMSE becomes more useful when
large errors are particularly undesirable. So, Root Mean Square value seems like a perfect
choice for our problem at hand.

##### Splitting the data into train and validation sets

```



```

##train test split for further modelling
X_train, X_test, y_train, y_test = train_test_split( train_cab.iloc[:, train_cab.columns !=
'fare_amount'],
            train_cab.iloc[:, 0], test_size = 0.20, random_state = 1)

print(X_train.shape)
print(X_test.shape)

# ### Regression analysis
#
# It is the process of investigating the relationships between dependent and independent
variables. It is widely used for the predictive analysis, forecasting and time series analysis.The
regression line is a straight line in this technique. The aim here is to minimize the error(sum of
squared error for instance)

# ## Modeling

# As the process of building the model should move from simple to complex. I am starting the
modeling with Multiple Linear Regression

# Building model on top of training dataset
fit_LR = LinearRegression().fit(X_train , y_train)

#prediction on train data
pred_train_LR = fit_LR.predict(X_train)

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

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

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

# ##### Decision Tree
#

```

```

# Decision trees are supervised learning algorithms, simple yet powerful in modeling non-
linear relationships. Being a non-parametric model, the aim of this algorithm is to learn a
model that can predict outcomes based on simple decision rules based on features.

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)

##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))

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

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

# ##### Random Forest
# Random forests or random decision forests are an ensemble learning method for
classification, regression and other tasks that operates by constructing a multitude of decision
trees at training time and outputting the class that is the mode of the classes or mean prediction
of the individual trees

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)

##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))

## calculate R^2 for train data

r2_score(y_train, pred_train_RF)

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

# #### GBR Modelling
# Regression 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 generalises them by allowing optimisation of an arbitrary
differentiable loss function.

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)

##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))

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

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

# ### Optimising the results with parameter tuning

from sklearn.ensemble import RandomForestRegressor
rf = RandomForestRegressor(random_state = 42)
from pprint import pprint
# Look at parameters used by our current forest

```

```

print('Parameters currently in use:\n')
pprint(rf.get_params())

##Random Hyperparameter Grid
from sklearn.model_selection import train_test_split,RandomizedSearchCV

##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)

#R^2
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 = {:.02}'.format(RRF_r2))
print('RMSE = ',RRF_rmse)

gb = GradientBoostingRegressor(random_state = 42)
from pprint import pprint
# Look at parameters used by our current forest
print('Parameters currently in use:\n')
pprint(gb.get_params())

##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)

#R^2
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)


from sklearn.model_selection import GridSearchCV
## 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 = {:.2}'.format(GRF_r2))
print('RMSE = ',(GRF_rmse))

## Grid Search CV for gradinet boosting
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 = {:.2}'.format(Ggb_r2))
print('RMSE = ',(Ggb_rmse))

# ### Model Selection
# So, It is obvious from above model performance comparison table GBM performed better
than other models comparatively on RMSE (Root Mean Square Error) and can be used for
deployment.

```

```

## Prediction of fare from provided test dataset :

# We have already cleaned and processed our test dataset along with our training dataset.
Hence we will be predicting using grid search CV for 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_test_Df = gridcv_rf.predict(test_cab)

predictions_GRF_test_Df

test_cab['Predicted_fare'] = predictions_GRF_test_Df

```