Project 1 – Predicting cab rental fares

<u>Problem background:</u>

With disruptive business ideas in daily commute such as Uber & Ola have completely changed the daily commute scenario for locals in the city, the previously popular local cab providers are struggling to meet the trip quality as well as the price which competitors such as Uber & Ola offer. The market is totally dominated by Uber & Ola, which was the case with local cab companies few years ago. The use of available technology & the willingness to adopt it are potentially the key parameters to provide efficient & affordable trips within the city. **Thus, a local cab company wants to employ a fare-prediction model which will predict a fare prior to beginning the trip, which does have a significant decision inclination of a passenger whether or not to ride the cab**. In order to predict accurate fares, it is important to determine the factors responsible, which is exactly what the goal of this project is & eventually predict the fares considering these parameters.

Problem Statement - After running a successful pilot phase, a cab company wants to employ a model which predicts the cab fares for various trips within the city.

<u>Train data</u>
Variables - 7 (6 independent & 1 dependent)
Dep. Variable - Fare Amount (Continuous)
Observations – 16067

<u>Test data</u>
Variables – 6 (6 independent)
Observations – 9914

Data description:

- Fare_amount
- Pickup_datetime
- Pickup_longitude
- Pickup_latitude
- Dropoff_longitude
- Dropoff_latitude
- Passenger_count

The detailed description about each variable is mentioned below:

Fare amount – This is the dependent variable which we have to predict, we have been provided with some historical data of previous trips, which will help us build and train our models with for better predictions on unseen data.

Pickup datetime – This is one of our independent variables, which has date and time information when the trip began.

Pickup longitude – Longitudinal coordinate of the location from where taxi picked up the fare.

Pickup latitude – Latitudinal coordinate of the location from where taxi picked up the fare.

Dropoff longitude - Longitudinal coordinate of the location from where taxi dropped off the fare.

Dropoff latitude - Latitudinal coordinate of the location from where taxi dropped off the fare.

Passenger count – Last of our independent variables, passenger count gives the no. of passengers riding in each cab during each trip.

The test data provided has all the variables except the **fare_amount**, which essentially, we're going to predict. All other variables are identical.

- **Phase 1 - Data Preprocessing**

Approach - In order to predict the cab fares, the most important parameters are **distance**, **time of day**. As these variables are not included in the dataset as direct entities, we need to extract the meaningful parameters out of the variables.

We'll breakup the pickup_datetime variable & extract year, month, day of month, day of week & time of day & create new variables for each of them.

Thus, breaking up the pickup datetime variable will give us:

Pickup year – The year in which the trip was picked up.
Pickup month – The month in which the trip was picked up.
Pickup day – The day of the month in which the trip was picked up.
Pickup day of the week – The day of the week in which the trip was picked up.
Pickup hour – The hour of the day in which the trip was picked up.

Each of this information is critical in analyzing multiple scenarios to help us better predict fares.

Similarly, we'll calculate the distance from latitudes & longitudes information from our dataset using a user-defined haversine formula function.

The **Haversine formula** determines the **spherical distance** (shortest path on a surface of a sphere – considering Earth as a sphere) **between 2 points** on a sphere given their **longitude & latitude**.

Given the nature of our problem & the data, which is provided, using the haversine formula is the best suited approach. First, we'll calculate the difference between all longitudes as delta longitudes, then the same for latitudes as delta latitudes, we know the radius of the earth to be ~6371 kms. We'll need to convert the angles from degrees to radians.

$$a = \sin^2(\Delta latDifference/2) + \cos(lat1).\cos(lt2).\sin^2(\Delta lonDifference/2)$$
$$c = 2.\mathrm{atan2}(\sqrt{a}, \sqrt{(1-a)})$$
$$d = R.c$$

where,

ΔlatDifference = lat1 – lat2 (difference of latitude)

ΔlonDifference = lon1 – lon2 (difference of longitude)

R is radius of earth i.e., **6371 KM** or **3961 miles**

and d is the distance computed between two points.

We'll loop this function over the dataset to calculate the distance in kms.

With the derivation of new variables completed, the dataset now has:

- Fare_amount
- Pickup_datetime
- Pickup_year
- Pickup_month
- Pickup_day
- Pickup_dow
- Pickup_hour
- Pickup_longitude
- Pickup_latitude
- Dropoff_longitude
- Dropoff_latitude
- Passenger_count
- distance

However, as the newly created variables are derived from the existing ones, there's redundancy, which we must remove before we proceed. Thus we'll remove these variables:

- Pickup_datetime
- Pickup_longitude
- Pickup_latitude
- Dropoff_longitude
- Dropoff_latitude

With the removal of redundant variables, our new dataset is comprising of:

- Fare_amount
- Pickup_year
- Pickup_month
- Pickup_day
- Pickup_dow
- Pickup_hour
- Passenger_count
- distance

One final check before we proceed to next phase is the data type check and update. These variables are representing pieces of information which might be important in predicting the fare, therefore, them being in appropriate data type is important.

- Fare amount – Must be a number, however, can be a decimal, therefore a **float**
- Pickup year – Must be a number, cannot be a decimal, therefore an **integer**
- Pickup month – Must be a number, cannot be a decimal, therefore an **integer**
- Pickup day – Must be a number, cannot be a decimal, therefore an **integer**
- Pickup day of the week – Is a number, cannot be decimal, thus an **integer**
- Pickup hour – Must be a number, cannot be a decimal, therefore an **integer**
- Passenger count - Must be a number, cannot be a decimal, therefore an **integer**
- Distance – Must be a number, however, can be a decimal, therefore a **float**

One important thing is not all variables have infinite valid inputs. For e.g.,

- **month** values can only range between 0-11 (12 valid levels of input)
- **day** values can only range between 0-30 (31 valid levels of input)
- **day of week** values can only range between 0-6 (7 valid levels of input)
- **hour** values can only range between 0-23 (24 valid levels of input)
- **passenger** count values cannot exceed 6 (6 valid levels of input)

Ideally, all these are classification variables with finite valid inputs. However, none of these variables are classified as nominal, all of them are ordinal. Meaning each variable has a level of increment or decrement in which you can classify these inputs. i.e., for months, March comes after February etc. Thus, in order to predict the fare, the ordinality won't affect our prediction.

- **Phase 2 – Data Cleaning**

Approach – Now that we have created our data frame which we'll be using it for our analysis, we'll need to clean the data & treat any data anomalies present in our dataset.

Missing data check

From initial observations, there are **55 observations** missing from the passenger count variable.

Other variables have anomalies, but they will be removed.

From checking for missing values, our data had **55 missing values** in the passenger count column. If we compare these 55 values with our dataset size, we find that the percentage of missing values compared to the data is less than **1%**. Thus, we'll remove the missing values. Now, we'll check the dataset for abnormal values or anomalies.

Outlier detection - manual

For any **passenger** vehicle such as taxi, maximum seating capacity will not exceed 6 including a driver & if a trip is considered to be complete, then minimum the cab must haul at least 1 passenger. Therefore, we have now established our threshold values of **1 & 6**. So, we can safely remove all the observations which have passenger count values less than **1** or greater than or equal to **6**.

- Remove passenger count values < 1
- Remove passenger count values > 6
- Remove passenger count values <= 0
- Remove passenger count values in decimals

For **distance** variable, we have considered the maximum trip distance to be 500 kms. Thus, any values for distance which are negative & exceeding 500 are also removed from the dataset.

- Remove distance values > 500
- Remove distance values <= 0

For **fare** amount, our dependent variable, we have considered the upper limit to be $400 and quite obviously we have removed values which are less than 0 and greater than 400.

- Remove fare values > 400
- Remove fare values <= 0
- Remove fare values with characters

For latitudes & longitudes, before we applied the haversine distance formula, we ensured to remove any unwanted co-ordinate values.

- Remove longitude values < -180 & > 180
- Remove longitude values < -90 & > 90
- Remove longitude values at (0,0) as the co-ordinates are not on land, but in the ocean.

We have used the capping method to detect and remove outliers from our data in a manual way because the dataset has only 2 continuous variables & all others as categorical.

Now we have checked & removed missing values from our data & also detected & capped manually the outliers from our data. Let's list down the hypothesis & assumptions before we explore the data.

Hypothesis & Assumptions:

- The pickup hour considers, the peak hours & the rest hours, & thus charged accordingly
- The pickup day of week considers the distribution of trips over weekdays & weekends
- The pickup year considers the distribution of trips over seasons, holidays & routine life
- The pickup day of the month considers the distribution of 22 workdays & 8 weekends
- The passenger count shows no impact on the fare charged per trip

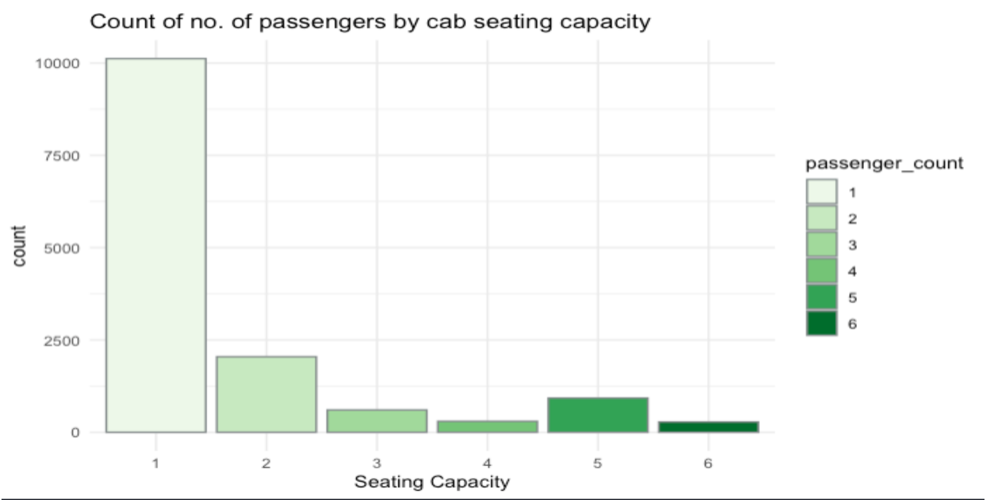
- The distance solely determines the trip fare & no external factor influences it
- **Phase 3 – Exploratory Data Analysis**

As previously mentioned, our dataset has a combination of continuous & categorical variables, thus, we'll first explore by category in both univariate & bivariate exploration. We have considered the following explorations:

- Univariate
- Bivariate

First moving to univariate categorical data exploration.

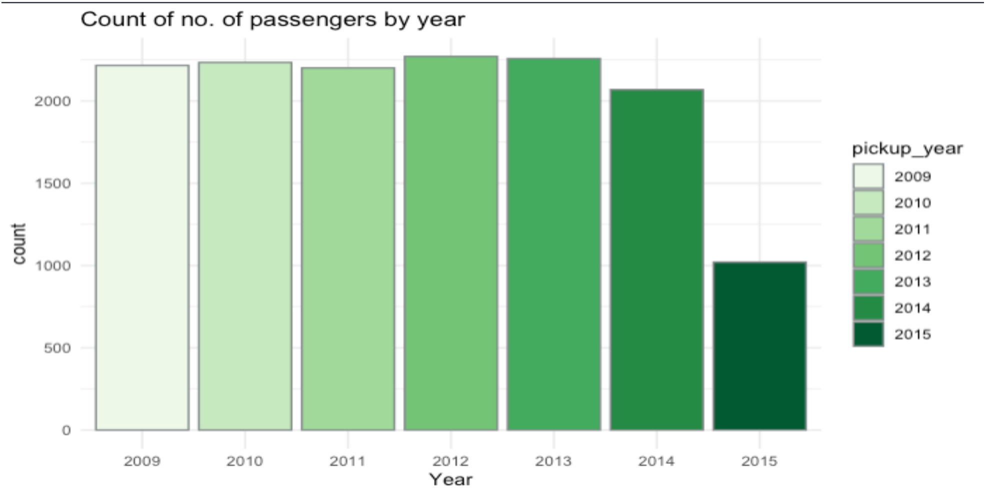
Passenger count



As depicted in the graph, the count of passengers is highest with only one seat occupied during the trip. As we move forward, the count drops, except with seating capacity 5 where we see a medium spike in passenger count. This spike could be attributed to families.

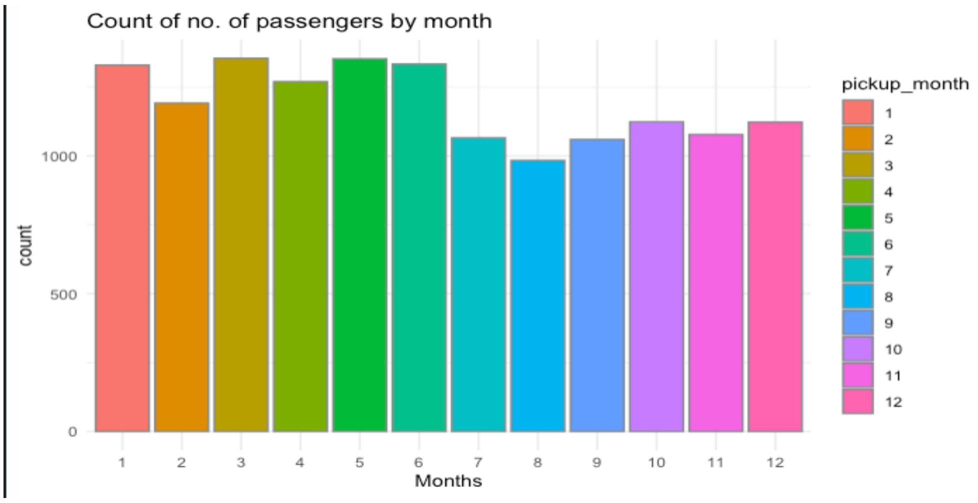
The second observation points us towards a sudden drop of passengers with seating capacity as 3 & 4 which might account to time of the day as explored later.

Pickup Year



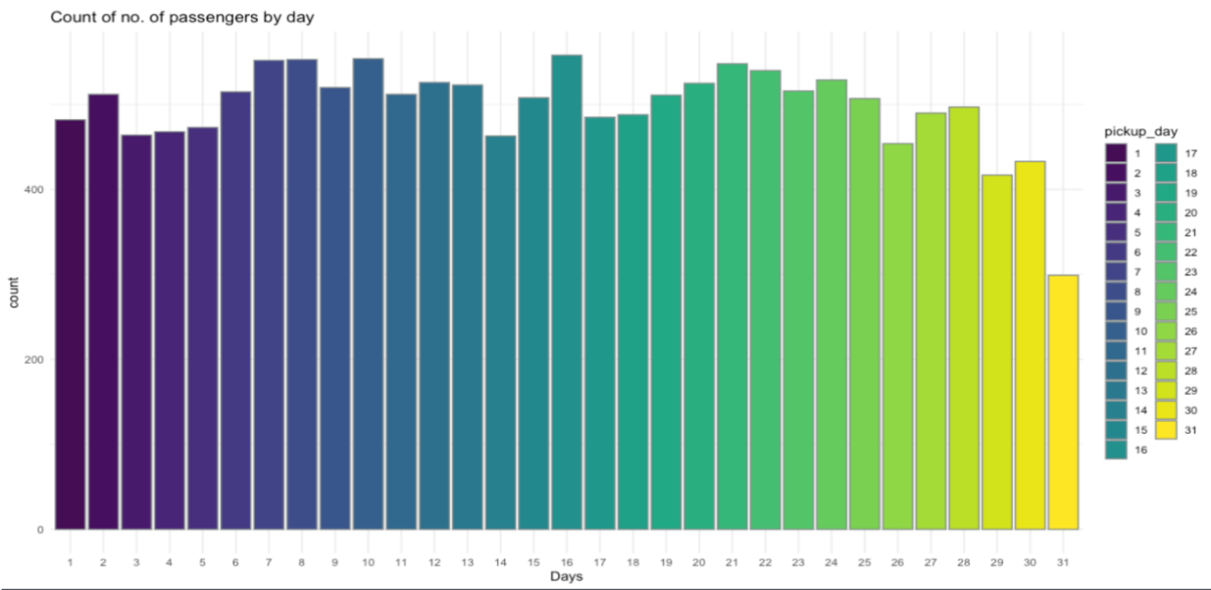
As depicted in the graph, the count of passengers over the past years has been pretty much constant, except in 2015 where we see a **drop** by more than **50%**. However, it is unclear whether the drop in number is due to any involved factors. On an average the taxi company is retaining **~2200** passengers every year.

Pickup month



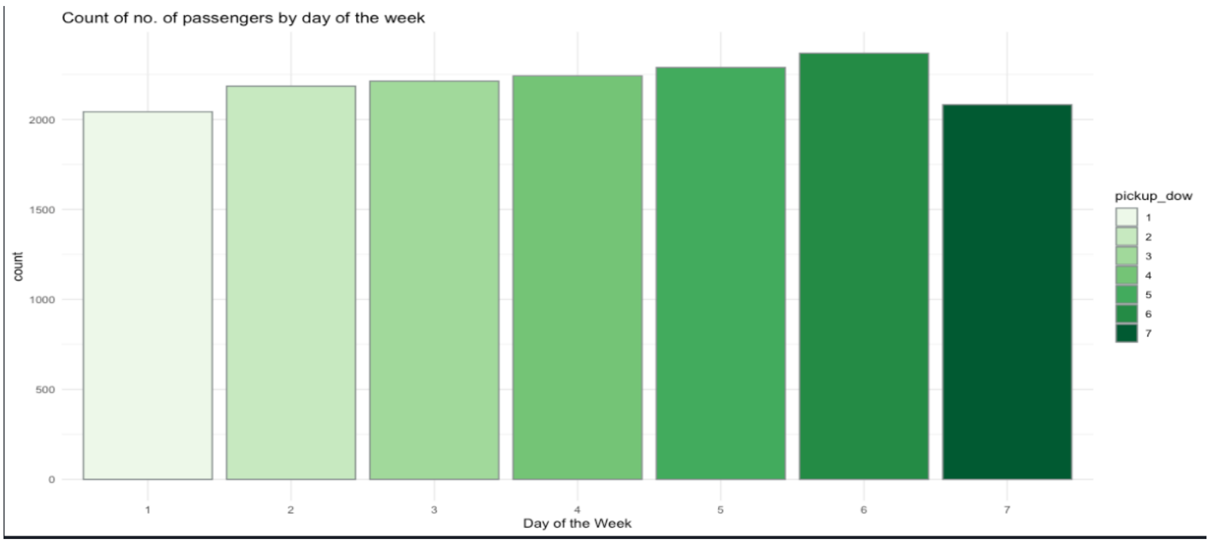
As we see from the above graph, the count of passengers is on the higher side for the first 6 months of every year, and then drops in the latter part of the year. Possible reasons could be, work routine life for professionals, routine life for family & friends, which is followed by school vacations from July to September and then the holiday season begins back from November.

Pickup day



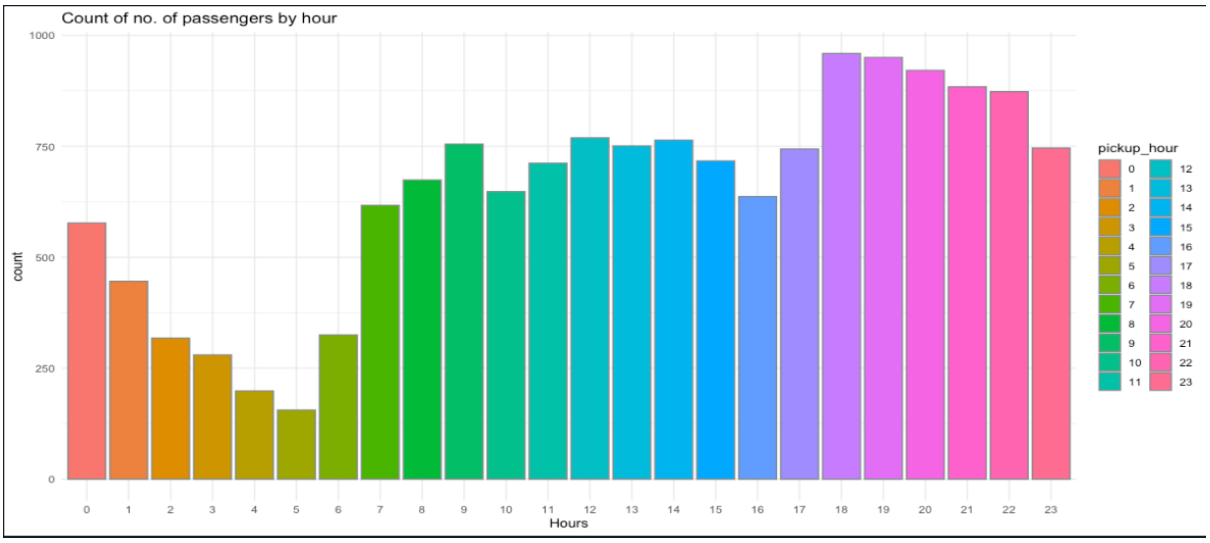
The above graph represents the distribution of pickups over the month, we observe no specific stand out pattern to draw any conclusions. On an average, ~410 pickups are made every day of the month, the 2nd week shows high distribution with 16th being the highest no. of pickups in a day. The pickups reduce toward the end of the month and drop on 31st, because not every month has 31 days & this is a collated distribution of pickups of 6 years.

Day of the week



Perhaps this is an important finding, of distribution of pickups over the week. The numeric digit 1 represents Monday and 7 being a Sunday. We see a gradual increase from Monday to Saturday with each day adding some number of pickups. The lowest we see is on a Sunday, where most people prefer resting after a hectic week and also on Monday where the week begins. To no surprise Fridays & Saturdays show the highest no. of pickups in a week, mostly because it is the end of the week and beginning of the much-awaited weekend.

Hour of the day

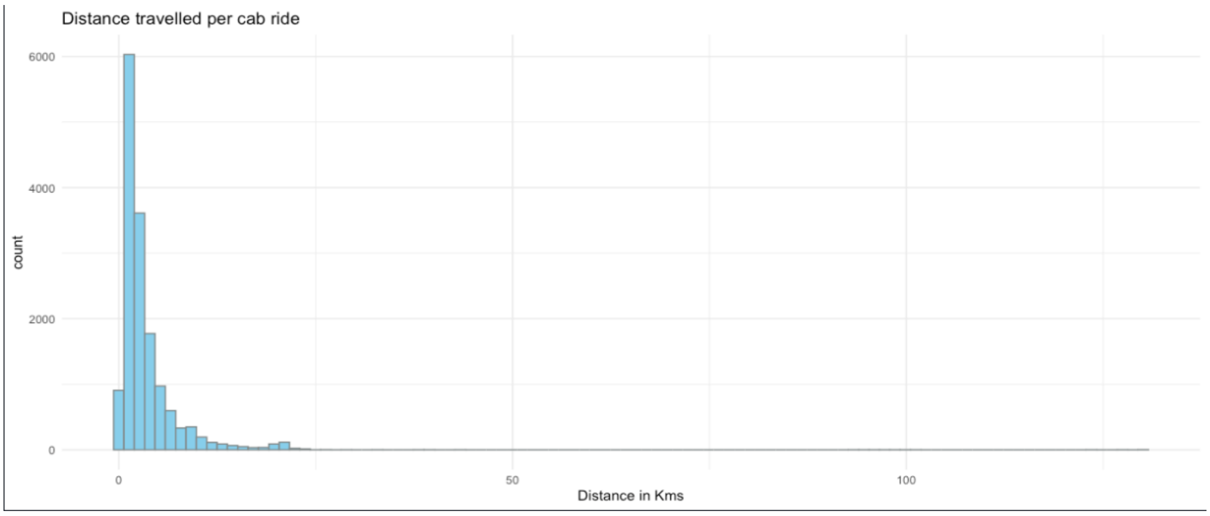


The above graph represents the distribution of pickups at different hours in a single day. This potentially is an important predictor of fare in Apps of Uber & Ola cabs. We'll try to determine if we can fetch any crucial information.

- To no surprise, pickups are lowest during the hours from **1 AM to 5 AM**.
- The count of pickups increases from 6 AM, marking the beginning of the day.
- We observe a sudden spike from 7 AM to 9 AM, beginning of work routine.
- A constant ups & down are observed between 10 AM to 3 PM, casual & work routine.
- The lowest during the day is observed at 6 PM, the hibernation period.

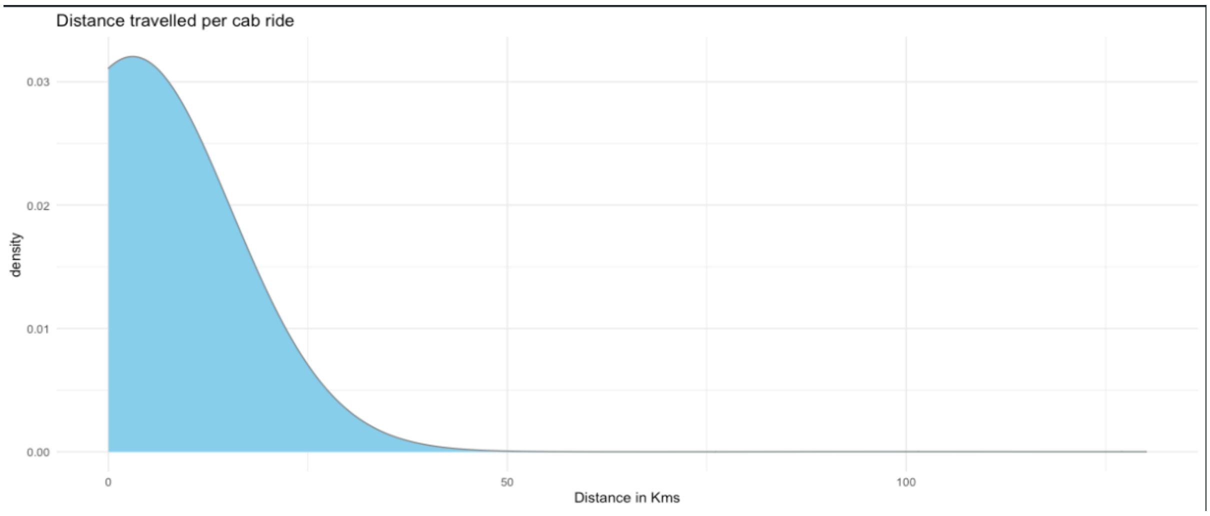
- Again, we observe a sudden spike from 5 PM to 10 PM, the returning work routine crowd.
- The maximum pickups happen between 5-6 PM, early returning crowd.
- Constant rush hours between 7 PM to 10 PM, which finally settles at 10 PM.
- We observe relatively high pickups between 11 PM & midnight, maybe nightlife crowds.

Distance



The above graph plots the distance travelled per trip. It is pretty common that the maximum no. of trips taken are short distance trips which are less than 10 kms. From there on we see a decreasing tail until 30 kms and then it is nothing but a straight line from there on forward. This indicates that this cab company is preferred for their **short haul** trips up to **30 kms.** However, there are passengers who have used it for longer distances, the frequency of such is very low.

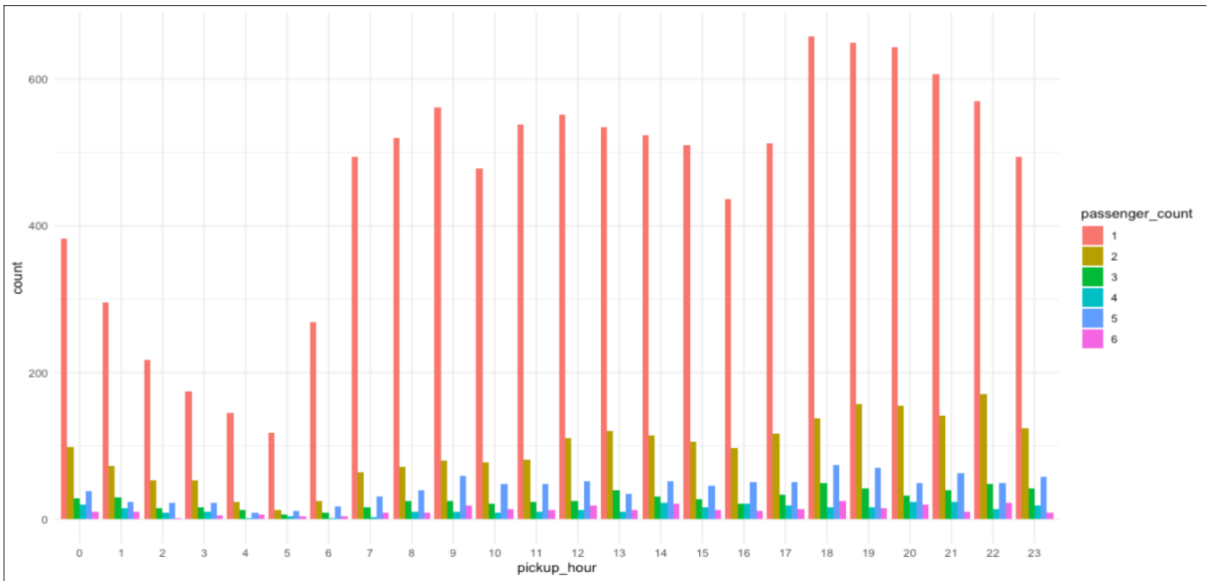
The same can be better observed in this density plot.



Bi-variate plots

The independent variables for our analysis are limited as the variables we are left with are derived variables, therefore, each variable will have a high correlation with the other. Thus we'll see the two most important bi-variate comparisons to conclude our analysis.

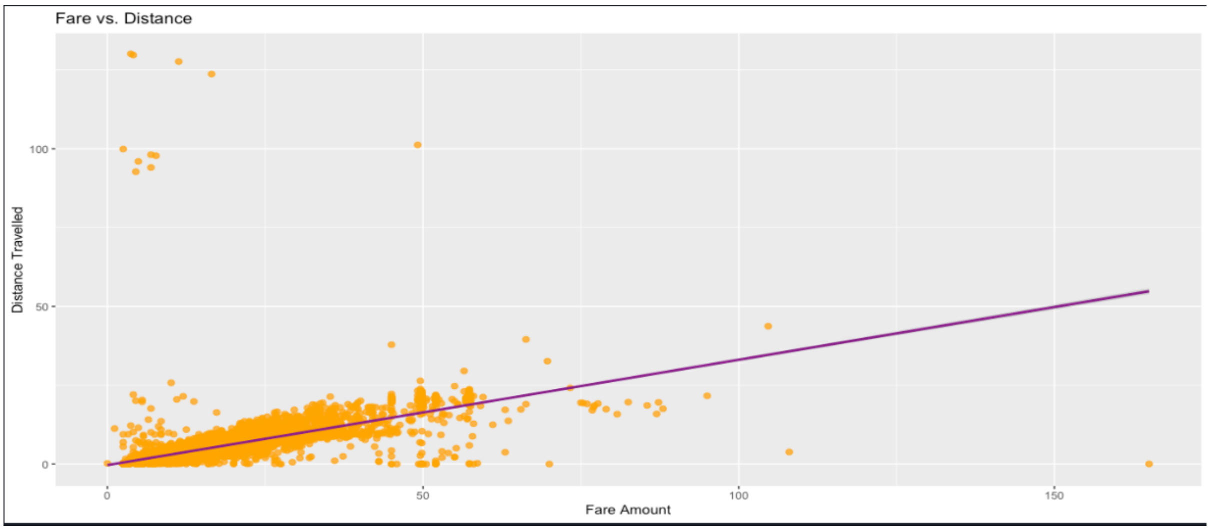
Passenger count vs time of the day



In the above graph, we observe the time of the day and along with it, the passenger count. This is to determine at what hours, the demand for larger cabs is the most as well as how can the single passenger trips be made more efficient.

It is obvious that passenger occupancy of 1 & 2 will dominate the pickup hours throughout the day, but instead we'll turn our attention to 4-6 passengers in a trip. A full occupancy trip or a single passenger trip as such makes no difference in terms of trip fare, but it could impact the trip duration & reduce the efficiency. The demand for full occupancy is highest during the midafternoon to late afternoon time slot. It could be the families, or group of friends sharing a cab to hop some place. It could also be the school children along with their guardians who could account for high occupancy.

Fare vs Distance

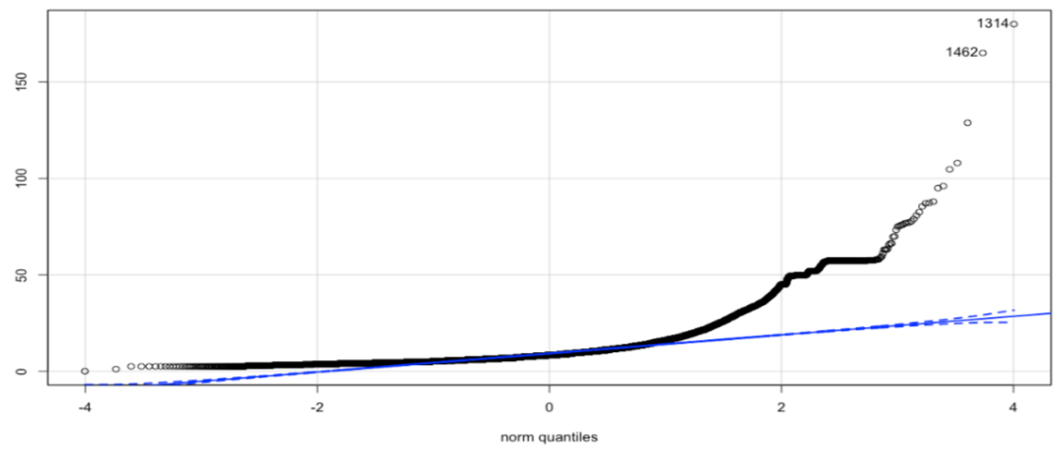


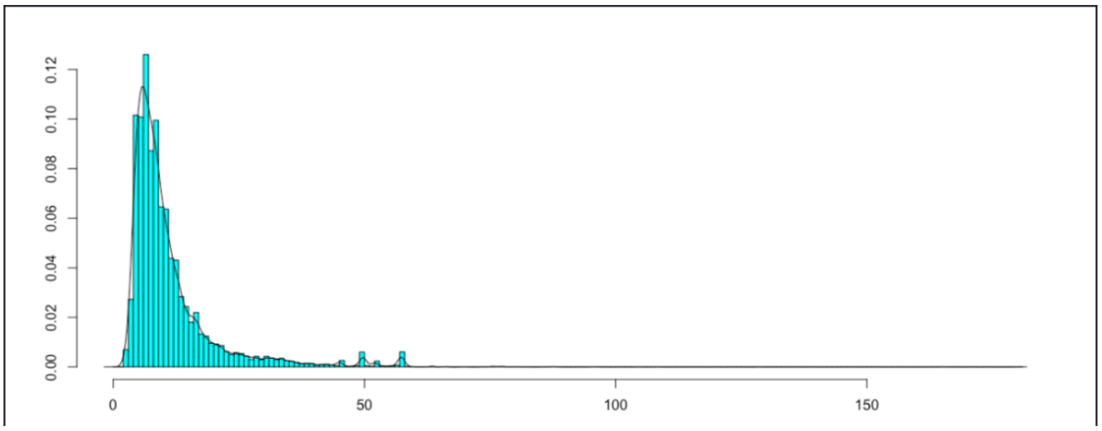
The most important comparison between our predictor **fare** & **distance** travelled. As of now all our analysis could not conclude whether the company wants time-based service or passenger-based service. As we observe this correlation is directly proportional & even confirmed by a linear model. As distance increases, the fare increases. No other parameters are contributing towards fare amount

- **Phase 4 – Feature Scaling & Transformation**

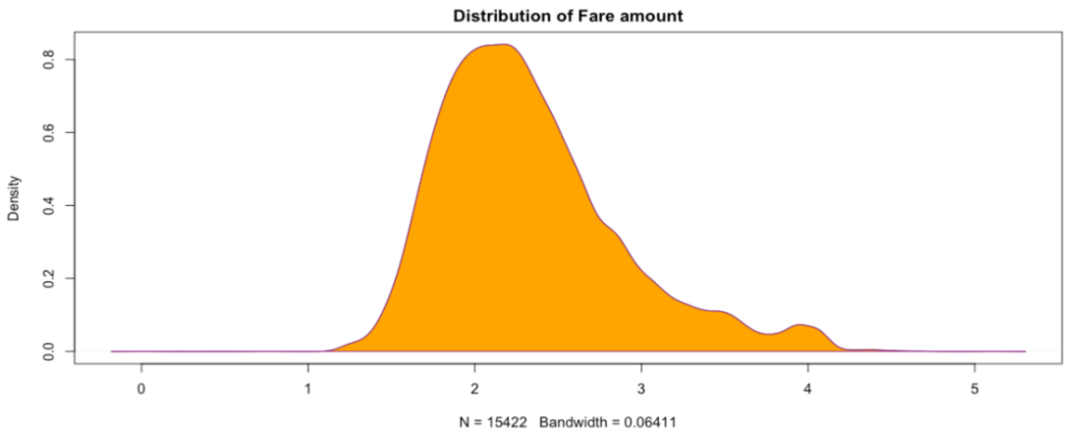
Approach – In this phase, we'll complete the final checks before proceeding towards modelling. Here we'll focus on normalizing our datasets and also checking the distributions to see if any tails (skewness or kurtosis) exist. As our dataset has limited features and only 2 of them are continuous variables, we'll begin with fare amount & distance.

Fare amount

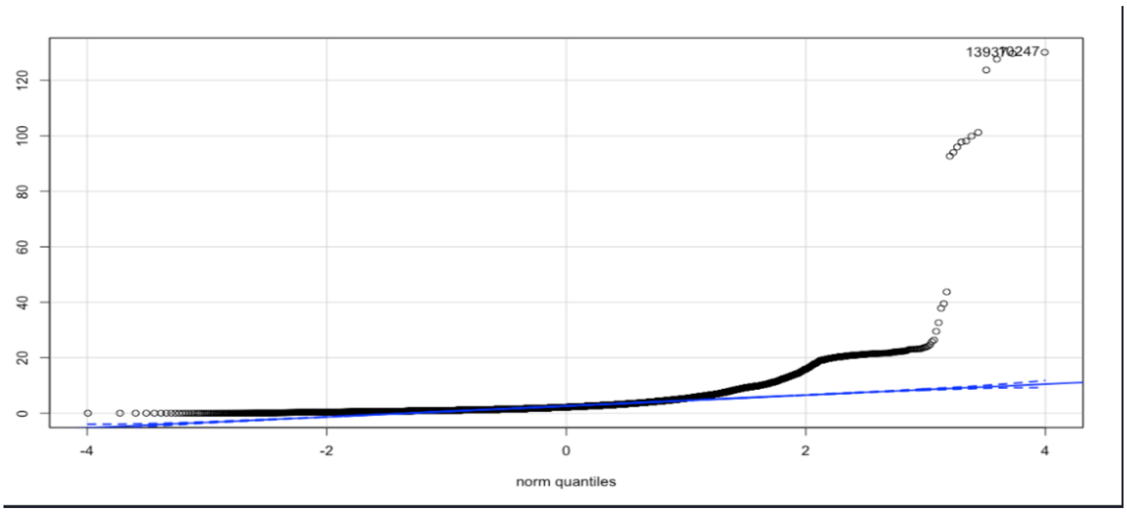


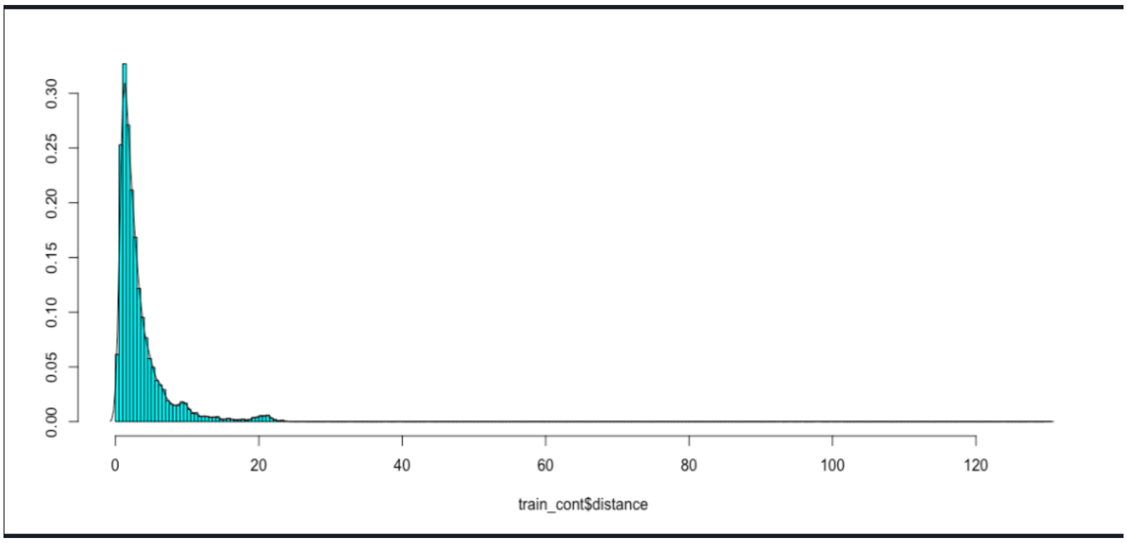


As we observe from both the plots that, there is an exceeding right tail to our fare variable, which indicates positive skew. Therefore, we'll need to apply log transformation to it.

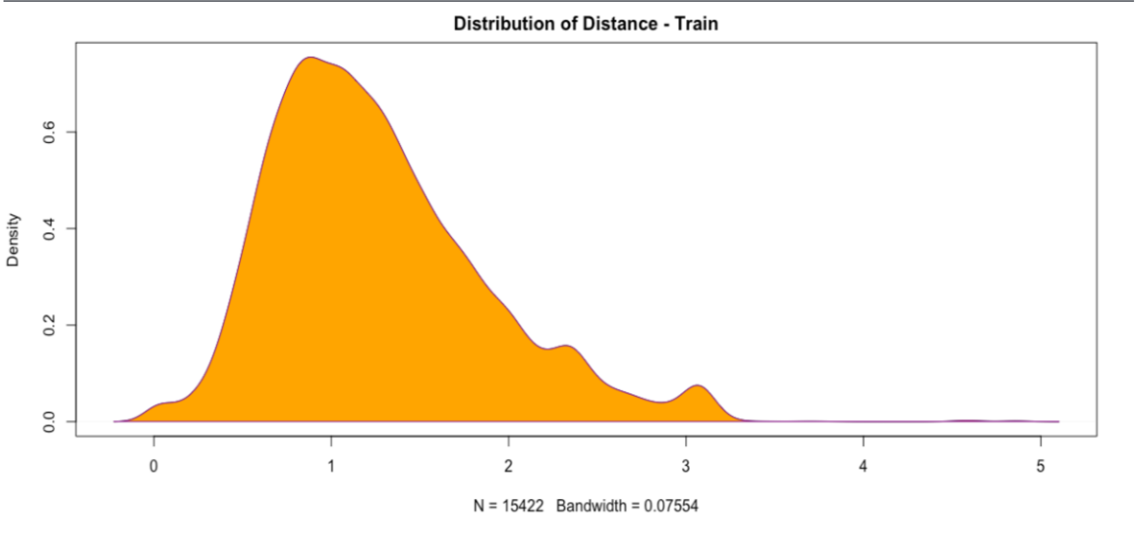


Distance





Again, like fare amount, our distance variable also shows right positive skew, thus we'll apply log transformation to both train as well as test datasets.



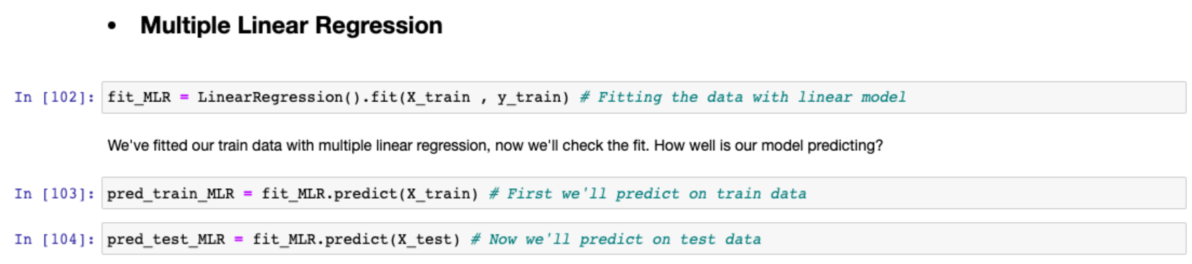
- **Phase 5 – Model Building & Evaluation**

The problem we're dealing with is a regression problem, where we have to predict a cab fare amount for a trip for a cab company. So, we have started the basic model building with a multiple linear regression model & moved on to ensemble techniques such a regression trees, random forest gradient boosting and XG boosting.

In order to train our model, we have used a 75% split with training data for the model to learn as much as possible from the observations and 25% as the test data on which we'll validate our model performance.

Linear Regression

We've built the multiple regression model with feeding our response variable with all other independent variables. Here's the model.



- **Multiple Linear Regression**

```
In [102]: fit_MLR = LinearRegression().fit(X_train , y_train) # Fitting the data with linear model
```

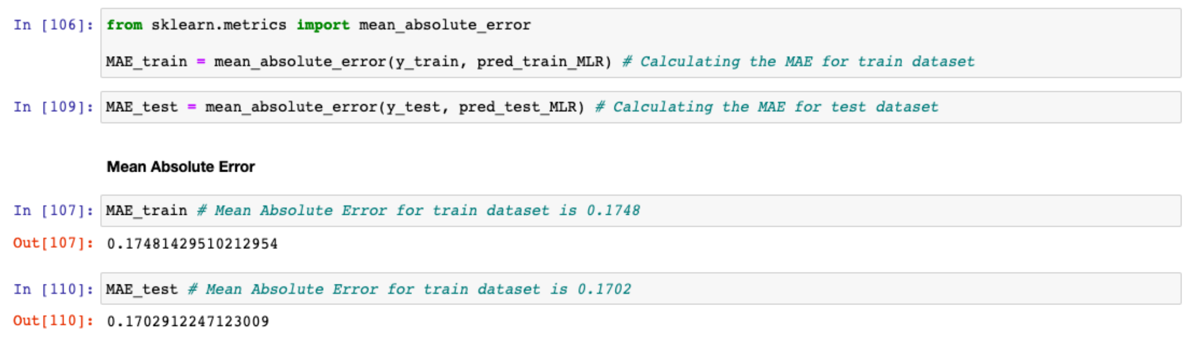
We've fitted our train data with multiple linear regression, now we'll check the fit. How well is our model predicting?

```
In [103]: pred_train_MLR = fit_MLR.predict(X_train) # First we'll predict on train data

In [104]: pred_test_MLR = fit_MLR.predict(X_test) # Now we'll predict on test data
```

We've used 3 metrics to evaluate model performance & R2 to check goodness of fit.

- Mean Absolute Error
- Mean Squared Error
- Root Mean Squared Error
- R-Squared

The mean absolute error tells us the difference between our forecasted value against the actual value. The bigger the deviation (error) larger the scope for improvement.



```
In [106]: from sklearn.metrics import mean_absolute_error

          MAE_train = mean_absolute_error(y_train, pred_train_MLR) # Calculating the MAE for train dataset

In [109]: MAE_test = mean_absolute_error(y_test, pred_test_MLR) # Calculating the MAE for test dataset
```

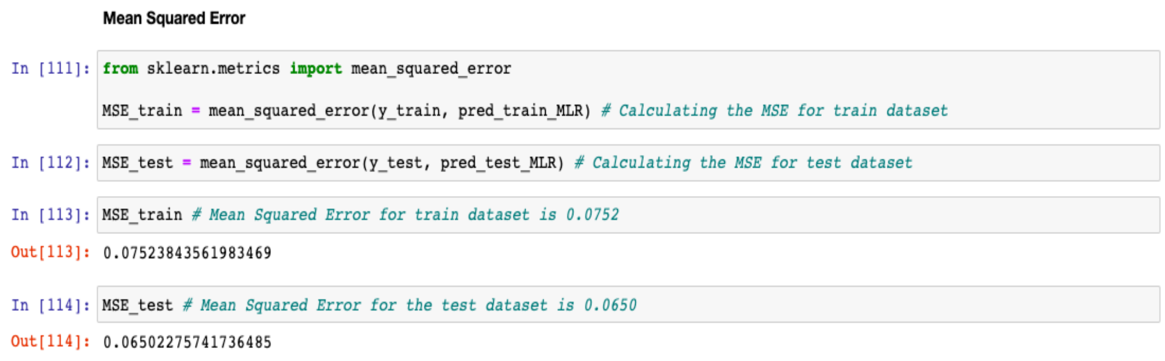
Mean Absolute Error

```
In [107]: MAE_train # Mean Absolute Error for train dataset is 0.1748
Out[107]: 0.17481429510212954

In [110]: MAE_test # Mean Absolute Error for train dataset is 0.1702
Out[110]: 0.1702912247123009
```

The **MAE** for our linear regression model for the train set is: **0.1748 i.e., ~17%,** which means that our model was able to predict with ~**83% accuracy**

Similarly, the **MAE** for our test set is: **0.1702 i.e., ~17%,** pretty much the same, but slight improvement on test (unseen data) which is a sign of a good learner model.

The mean squared error measures how close is your predicted observation from the regression line. It squares these differences in order to eliminate any negative values.

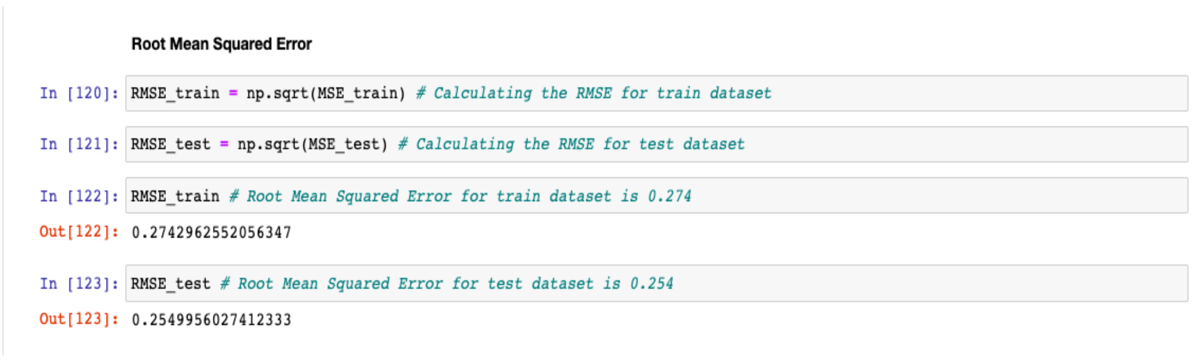

Mean Squared Error

```
In [111]: from sklearn.metrics import mean_squared_error

          MSE_train = mean_squared_error(y_train, pred_train_MLR) # Calculating the MSE for train dataset

In [112]: MSE_test = mean_squared_error(y_test, pred_test_MLR) # Calculating the MSE for test dataset

In [113]: MSE_train # Mean Squared Error for train dataset is 0.0752

Out[113]: 0.07523843561983469

In [114]: MSE_test # Mean Squared Error for the test dataset is 0.0650

Out[114]: 0.06502275741736485
```

The **MSE** for our linear regression model for the train set is: **0.0752 i.e., ~7%,** which means that our model was able to predict with ~**93% accuracy.**

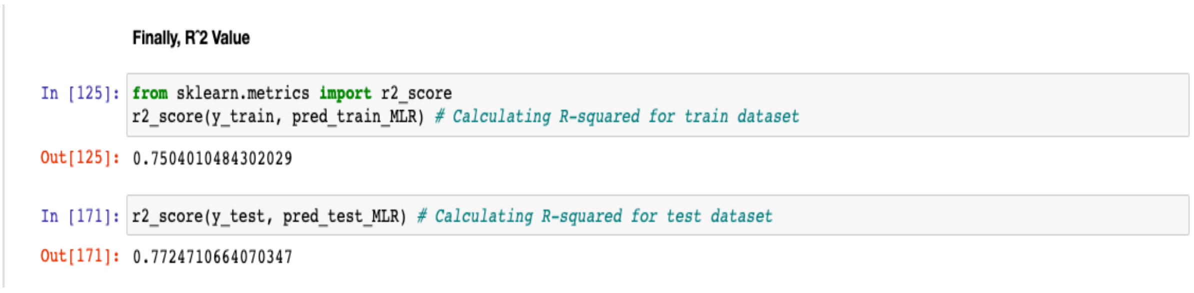
Similarly, the **MSE** for our test set is: **0.0650 i.e., ~6%,** even better than the train set, which is essentially an excellent prediction.

The root mean squared error is a square root of the mean squared error. This is essentially a comparison between models.


Root Mean Squared Error

```
In [120]: RMSE_train = np.sqrt(MSE_train) # Calculating the RMSE for train dataset

In [121]: RMSE_test = np.sqrt(MSE_test) # Calculating the RMSE for test dataset

In [122]: RMSE_train # Root Mean Squared Error for train dataset is 0.274

Out[122]: 0.2742962552056347

In [123]: RMSE_test # Root Mean Squared Error for test dataset is 0.254

Out[123]: 0.2549956027412333
```

The RMSE for train set is: **0.2742, i.e., ~27%**
The RMSE for test set is: **0.2549, i.e., ~25%**


Finally, R^2 Value

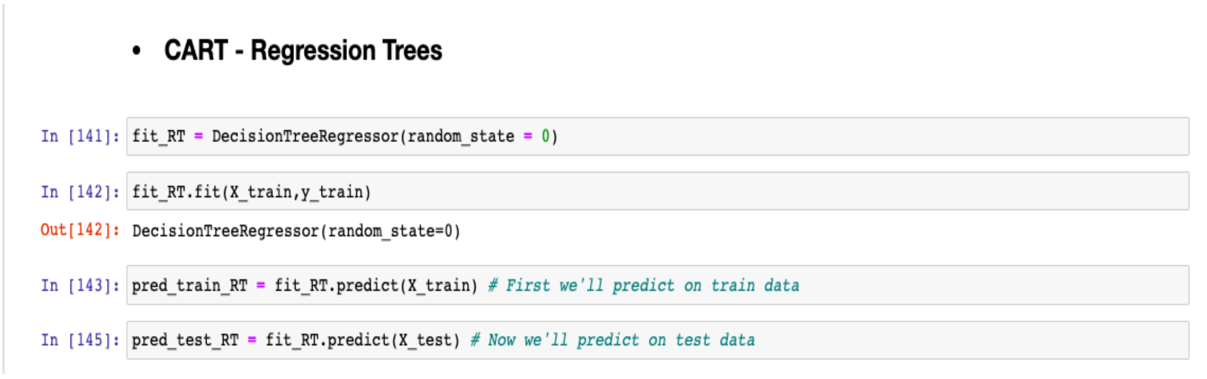
```
In [125]: from sklearn.metrics import r2_score
          r2_score(y_train, pred_train_MLR) # Calculating R-squared for train dataset

Out[125]: 0.7504010484302029

In [171]: r2_score(y_test, pred_test_MLR) # Calculating R-squared for test dataset

Out[171]: 0.7724710664070347
```

The R-Squared is a goodness of fit measure which determines how well have the parameters fitted the model. Here for a linear regression model, the values are:

R-Squared for train set: **0.7504 i.e., ~75%**
R-Squared for test set: 0.7724, **i.e., ~77%**

Regression Tree

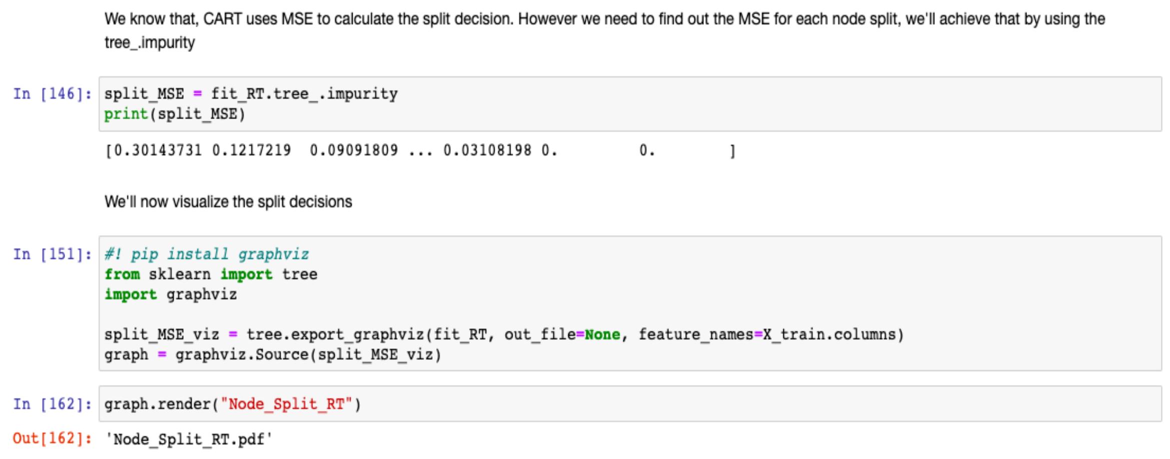
Essentially the RT of CART is the regression tree. It is an ensemble learning algorithm which can predict both classification as well as regression variables. Here's our model



- **CART - Regression Trees**

```
In [141]: fit_RT = DecisionTreeRegressor(random_state = 0)

In [142]: fit_RT.fit(X_train,y_train)
Out[142]: DecisionTreeRegressor(random_state=0)

In [143]: pred_train_RT = fit_RT.predict(X_train) # First we'll predict on train data

In [145]: pred_test_RT = fit_RT.predict(X_test) # Now we'll predict on test data
```

We've modelled CART initially with default parameters. We'll check the performance and then depending on the need, we'll tune the parameters as we may need them.



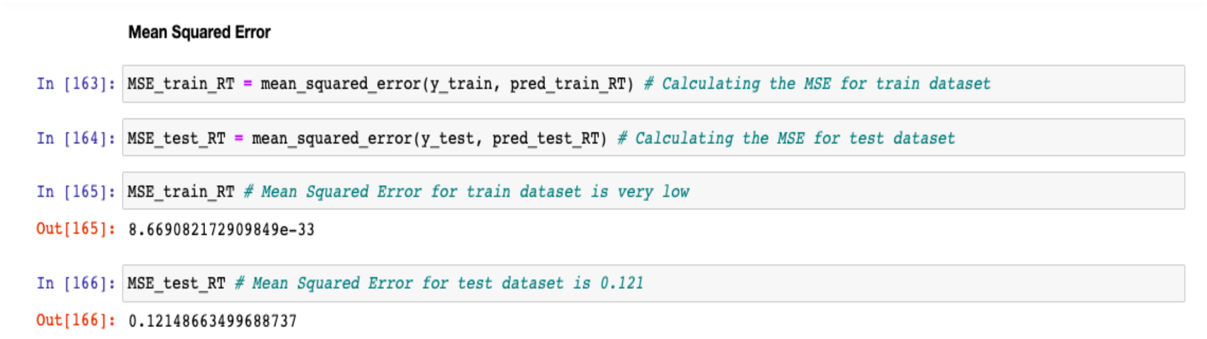
We know that, CART uses MSE to calculate the split decision. However we need to find out the MSE for each node split, we'll achieve that by using the tree_.impurity

```
In [146]: split_MSE = fit_RT.tree_.impurity
          print(split_MSE)
          [0.30143731 0.1217219  0.09091809 ... 0.03108198 0.         0.        ]
```

We'll now visualize the split decisions

```
In [151]: #! pip install graphviz
          from sklearn import tree
          import graphviz

          split_MSE_viz = tree.export_graphviz(fit_RT, out_file=None, feature_names=X_train.columns)
          graph = graphviz.Source(split_MSE_viz)

In [162]: graph.render("Node_Split_RT")
Out[162]: 'Node_Split_RT.pdf'
```

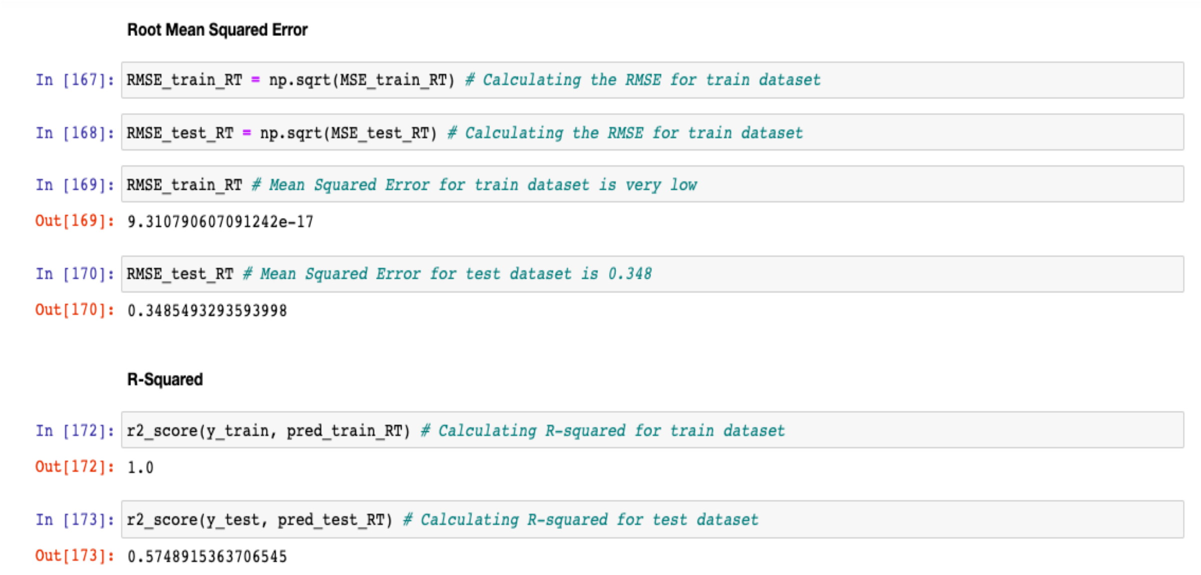
CART uses MSE to calculate the split decisions. Depending on the MSE value a node is further split into leaf nodes and the entire split decisions are saved in the pdf file (attached).



Mean Squared Error

```
In [163]: MSE_train_RT = mean_squared_error(y_train, pred_train_RT) # Calculating the MSE for train dataset

In [164]: MSE_test_RT = mean_squared_error(y_test, pred_test_RT) # Calculating the MSE for test dataset

In [165]: MSE_train_RT # Mean Squared Error for train dataset is very low
Out[165]: 8.669082172909849e-33

In [166]: MSE_test_RT # Mean Squared Error for test dataset is 0.121
Out[166]: 0.12148663499688737
```

As we have used the default parameters, we notice an extremely low, almost negligible MSE value, however it means the model is performing with 100% accuracy, **which cannot be true**. If you further observe the same model performs well with unseen data with only ~12% error. This clearly indicates that the splits have a very high correlation between them which is making the model overfit the data. Thus, we would need to fine tune our regression tree model.



Root Mean Squared Error

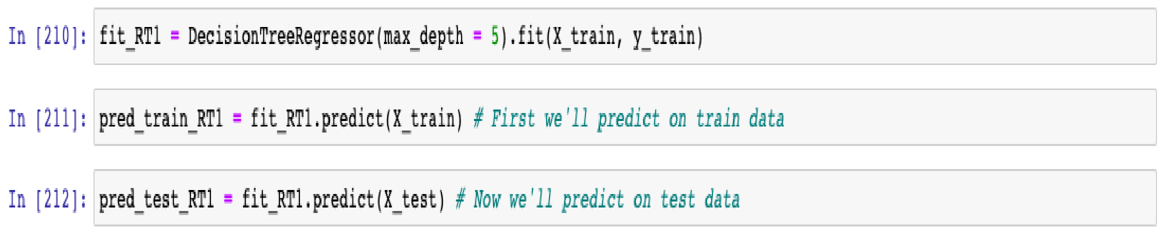
```
In [167]: RMSE_train_RT = np.sqrt(MSE_train_RT) # Calculating the RMSE for train dataset

In [168]: RMSE_test_RT = np.sqrt(MSE_test_RT) # Calculating the RMSE for train dataset

In [169]: RMSE_train_RT # Mean Squared Error for train dataset is very low
Out[169]: 9.310790607091242e-17

In [170]: RMSE_test_RT # Mean Squared Error for test dataset is 0.348
Out[170]: 0.3485493293593998
```

R-Squared

```
In [172]: r2_score(y_train, pred_train_RT) # Calculating R-squared for train dataset
Out[172]: 1.0

In [173]: r2_score(y_test, pred_test_RT) # Calculating R-squared for test dataset
Out[173]: 0.5748915363706545
```

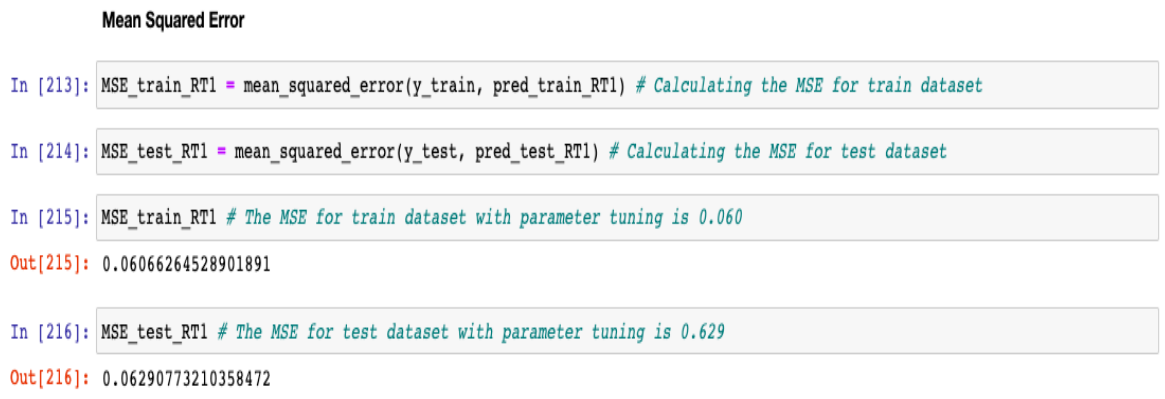
The exact same thing is seen in RMSE, where the model is overfitting the data and working relatively better with test data. If you observe the R-Squared value, it is ~57 not a great model by any means.

Let us know add a few parameters such as **max_depth**, which restricts the growth of the tree and avoids overfitting the model.



```
In [210]: fit_RT1 = DecisionTreeRegressor(max_depth = 5).fit(X_train, y_train)

In [211]: pred_train_RT1 = fit_RT1.predict(X_train) # First we'll predict on train data

In [212]: pred_test_RT1 = fit_RT1.predict(X_test) # Now we'll predict on test data
```

We have rebuilt a model with max_depth = 5, which means the tree will restrict its growth after 5 splits deep. Let us compare how this model compared to the previous one.



Mean Squared Error

```
In [213]: MSE_train_RT1 = mean_squared_error(y_train, pred_train_RT1) # Calculating the MSE for train dataset

In [214]: MSE_test_RT1 = mean_squared_error(y_test, pred_test_RT1) # Calculating the MSE for test dataset

In [215]: MSE_train_RT1 # The MSE for train dataset with parameter tuning is 0.060
Out[215]: 0.06066264528901891

In [216]: MSE_test_RT1 # The MSE for test dataset with parameter tuning is 0.629
Out[216]: 0.06290773210358472
```

If we observe carefully, the MSE values for train dataset has improved drastically. From almost negligible it is now at **~6%** which means model is **94%** accurate. A similar MSE value is seen for the test dataset, with **~6.2%** error i.e., **93.8%** accuracy.

Root Mean Squared Error

```
In [217]: RMSE_train_RT1 = np.sqrt(MSE_train_RT1) # Calculating the RMSE for train dataset

In [218]: RMSE_test_RT1 = np.sqrt(MSE_test_RT1) # Calculating the RMSE for test dataset

In [219]: RMSE_train_RT1 # The RMSE for train dataset with parameter tuning is 0.246

Out[219]: 0.24629787918091967

In [220]: RMSE_test_RT1 # The RMSE for test dataset with parameter tuning is 0.250

Out[220]: 0.2508141385639668
```

R-Squared

```
In [221]: r2_score(y_train, pred_train_RT1) # Calculating R-squared for train dataset

Out[221]: 0.7987553497244966

In [222]: r2_score(y_train, pred_train_RT1) # Calculating R-squared for train dataset

Out[222]: 0.7987553497244966
```

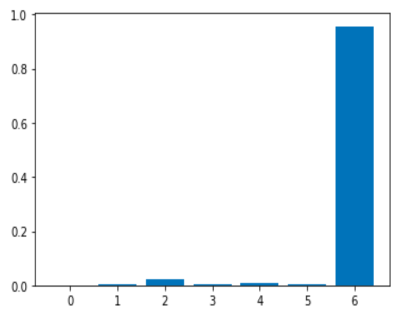
Compared to the previous model, the RMSE value as well as the R-Squared value has increased. The R-Squared value sits at ~79.87 which is an improvement of over ~20% that's great.

With further tweaking the max_depth value of our tree, we get even better results.

At max_depth = **7**, we get a R-Squared value of **82.4,** we'll discuss this more in hyperparameter tuning section later.

Also, plotting the variable importance plot from Regression tree shows:

```
In [254]: importance = fit_RT2.feature_importances_ # Get importance of each feature in the model

In [255]: for i,v in enumerate(importance):
              print('Feature: %0d, Score: %.5f' % (i,v))

          Feature: 0, Score: 0.00041
          Feature: 1, Score: 0.00453
          Feature: 2, Score: 0.02091
          Feature: 3, Score: 0.00456
          Feature: 4, Score: 0.00886
          Feature: 5, Score: 0.00356
          Feature: 6, Score: 0.95717

In [257]: plt.bar([x for x in range(len(importance))], importance)
          plt.show()
```

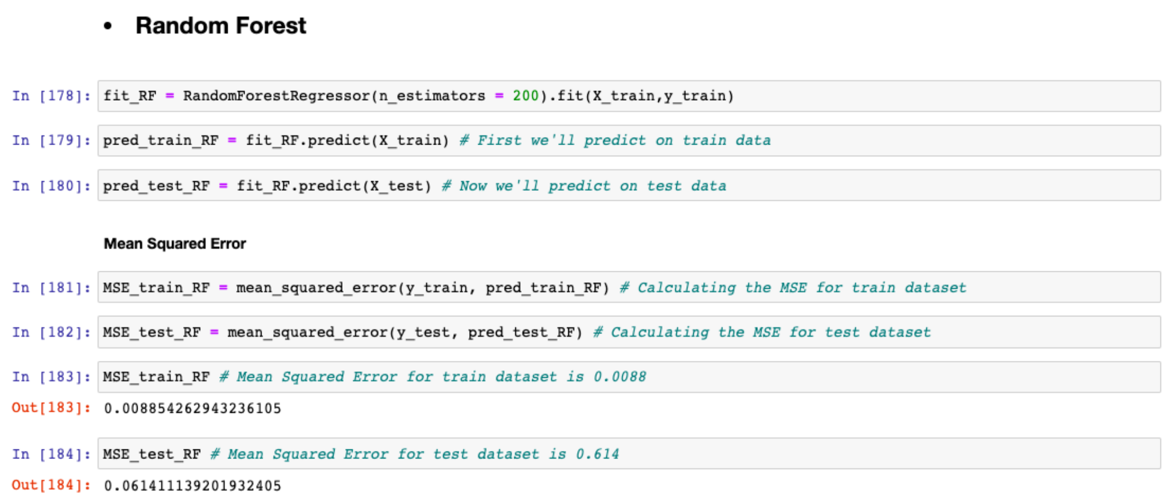


Out of the 7 independent variables, 1 variable is contributing for more than 95% of the weight, and the other 6 variables combined account for remaining 5% weight.

That 1 variable is **distance**, which we initially presumed to be decisive.

Random Forest:

Random Forest, again one of the ensemble learning model, used to predict both classification as well as regression variables.



- **Random Forest**

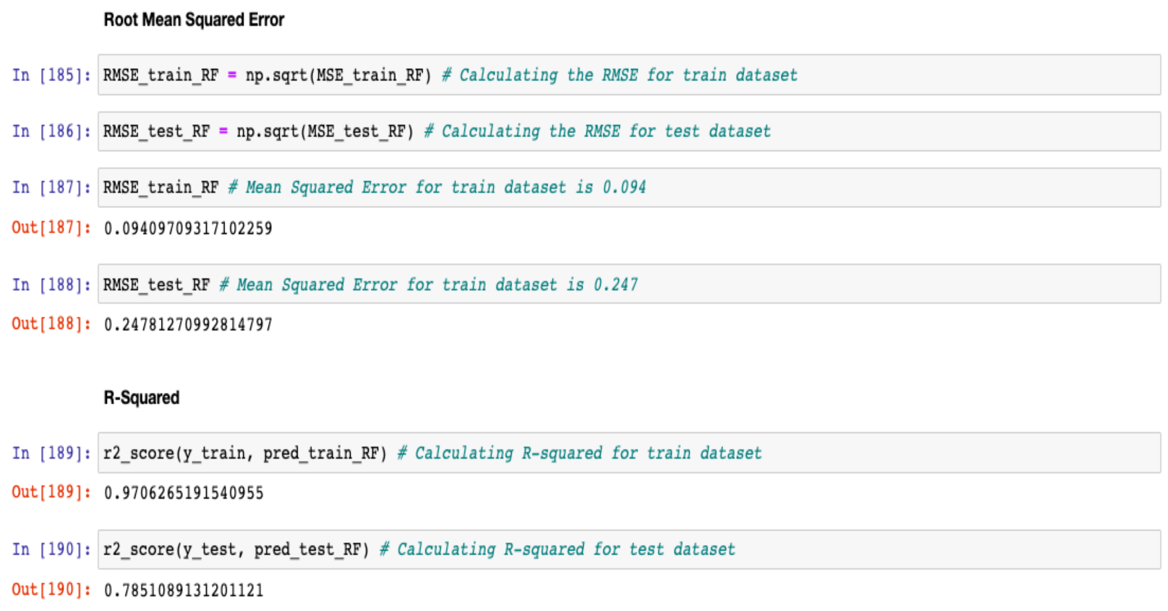
```
In [178]: fit_RF = RandomForestRegressor(n_estimators = 200).fit(X_train,y_train)

In [179]: pred_train_RF = fit_RF.predict(X_train) # First we'll predict on train data

In [180]: pred_test_RF = fit_RF.predict(X_test) # Now we'll predict on test data
```

Mean Squared Error

```
In [181]: MSE_train_RF = mean_squared_error(y_train, pred_train_RF) # Calculating the MSE for train dataset

In [182]: MSE_test_RF = mean_squared_error(y_test, pred_test_RF) # Calculating the MSE for test dataset

In [183]: MSE_train_RF # Mean Squared Error for train dataset is 0.0088
Out[183]: 0.008854262943236105

In [184]: MSE_test_RF # Mean Squared Error for test dataset is 0.614
Out[184]: 0.061411139201932405
```

As we observed from Regression tree that a model with default parameters do not yield a good model, thus we have added a **n_estimators** parameter with **200 trees**.

The 200-tree forest gives us a MSE value of 0.008, which again is too good to be true. However, on the test dataset it performs at 0.0614 i.e., **~6%.**



Root Mean Squared Error

```
In [185]: RMSE_train_RF = np.sqrt(MSE_train_RF) # Calculating the RMSE for train dataset

In [186]: RMSE_test_RF = np.sqrt(MSE_test_RF) # Calculating the RMSE for test dataset

In [187]: RMSE_train_RF # Mean Squared Error for train dataset is 0.094
Out[187]: 0.09409709317102259

In [188]: RMSE_test_RF # Mean Squared Error for train dataset is 0.247
Out[188]: 0.24781270992814797
```

R-Squared

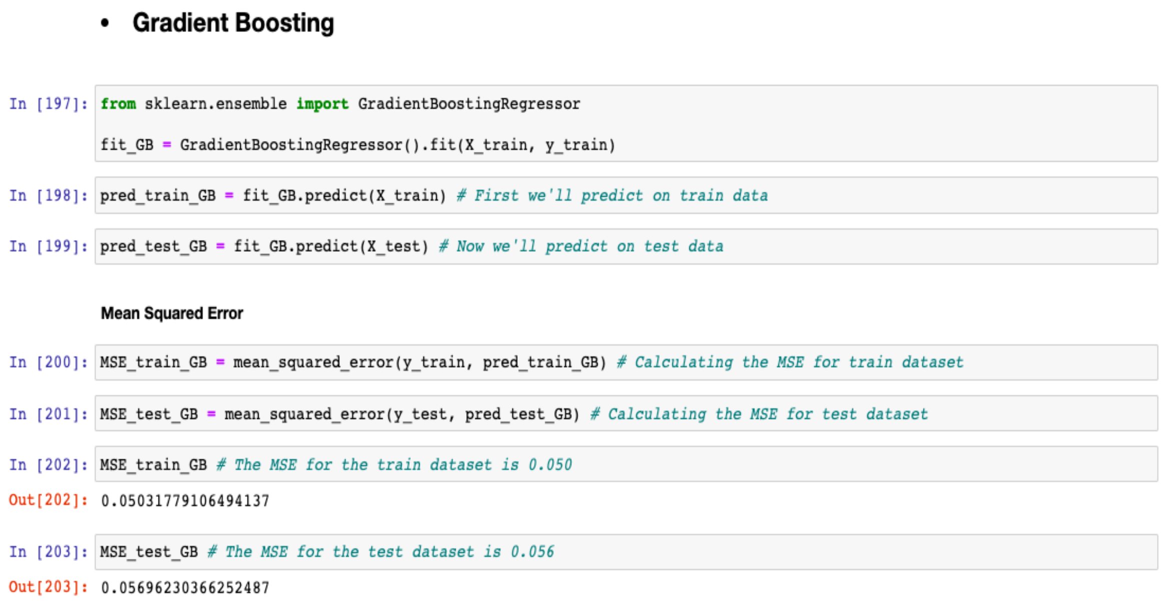
```
In [189]: r2_score(y_train, pred_train_RF) # Calculating R-squared for train dataset
Out[189]: 0.9706265191540955

In [190]: r2_score(y_test, pred_test_RF) # Calculating R-squared for test dataset
Out[190]: 0.7851089131201121
```

Moving on to RMSE and R-Squared, the model is performing much better on the training data with RMSE at 9%. However, there’s a distinct drop in performance on the test set.

Overall, the random forest model performs better only because we’ve not yet fully explored all of its hyperparameters and despite that is performing so well.

Moving on to our last model before moving on to hyperparameter tuning.

Gradient Boosting



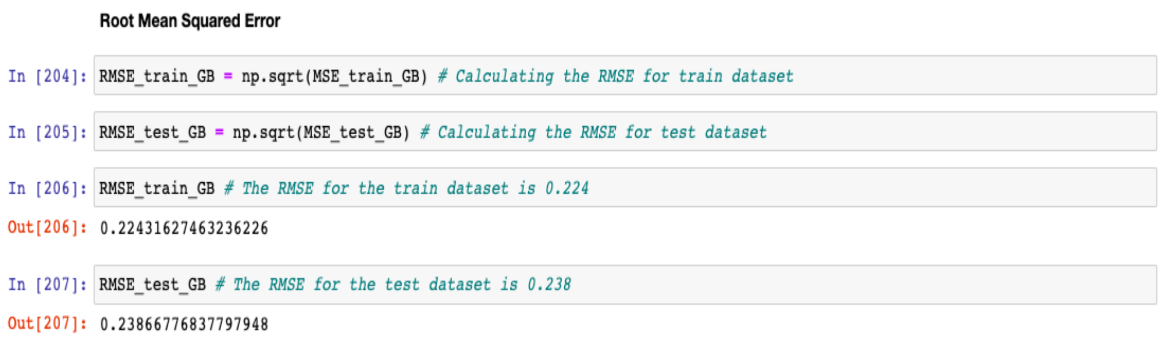
- **Gradient Boosting**

```
In [197]: from sklearn.ensemble import GradientBoostingRegressor

          fit_GB = GradientBoostingRegressor().fit(X_train, y_train)

In [198]: pred_train_GB = fit_GB.predict(X_train) # First we'll predict on train data

In [199]: pred_test_GB = fit_GB.predict(X_test) # Now we'll predict on test data
```

Mean Squared Error

```
In [200]: MSE_train_GB = mean_squared_error(y_train, pred_train_GB) # Calculating the MSE for train dataset

In [201]: MSE_test_GB = mean_squared_error(y_test, pred_test_GB) # Calculating the MSE for test dataset

In [202]: MSE_train_GB # The MSE for the train dataset is 0.050
Out[202]: 0.05031779106494137

In [203]: MSE_test_GB # The MSE for the test dataset is 0.056
Out[203]: 0.05696230366252487
```

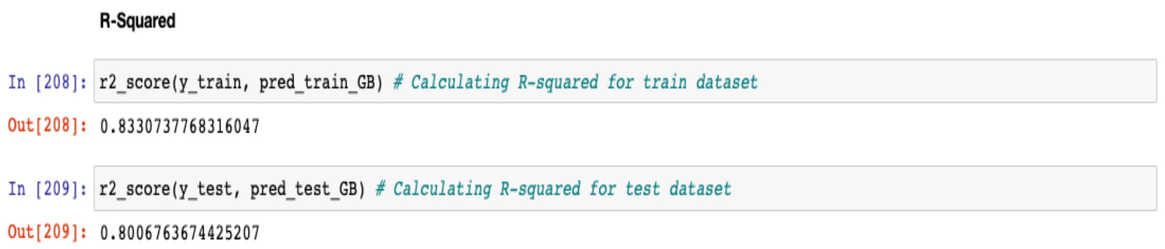
We'll fit the GB model with default parameters and tune the parameters later. With the default parameters the model is performing at 5% MSE on train dataset and about the same on test data.



Root Mean Squared Error

```
In [204]: RMSE_train_GB = np.sqrt(MSE_train_GB) # Calculating the RMSE for train dataset

In [205]: RMSE_test_GB = np.sqrt(MSE_test_GB) # Calculating the RMSE for test dataset

In [206]: RMSE_train_GB # The RMSE for the train dataset is 0.224
Out[206]: 0.22431627463236226

In [207]: RMSE_test_GB # The RMSE for the test dataset is 0.238
Out[207]: 0.23866776837797948
```

The RMSE on train set is: **22%** and RMSE on test set is: **~24%.** Let's check the R-Squared value**.**



R-Squared

```
In [208]: r2_score(y_train, pred_train_GB) # Calculating R-squared for train dataset
Out[208]: 0.8330737768316047

In [209]: r2_score(y_test, pred_test_GB) # Calculating R-squared for test dataset
Out[209]: 0.8006763674425207
```

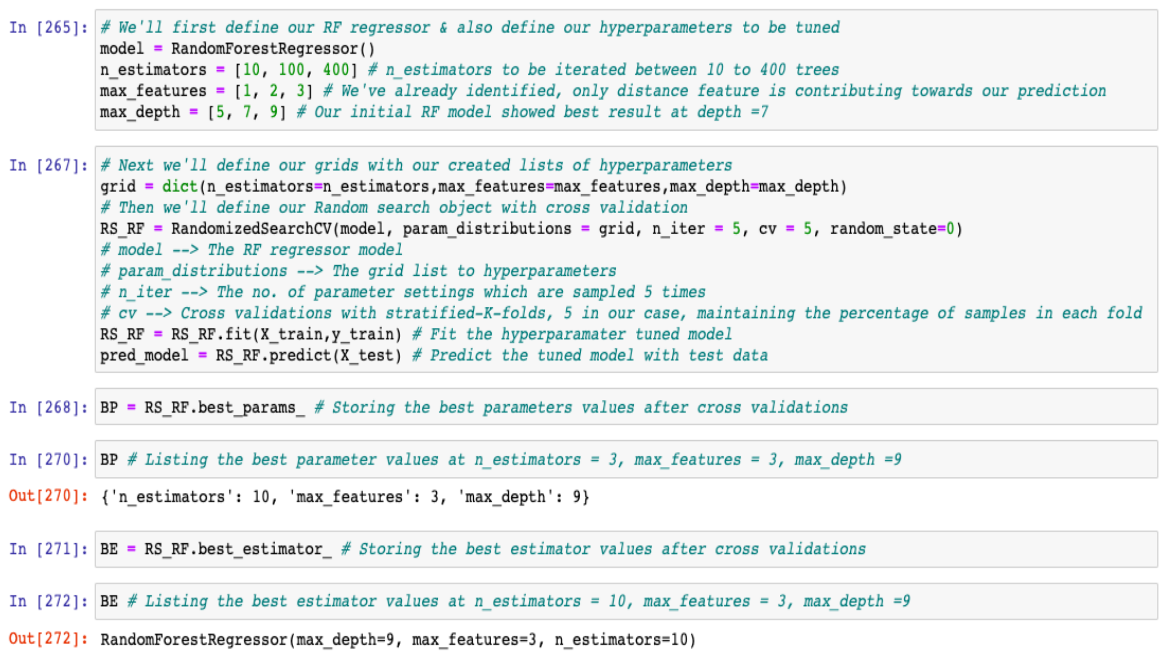
The R-Squared value on train data is: **83%** which marks an excellent fit on the data.

The R-Squared value on test data is: **80%** which considering unseen data is also very good.

This completes the initial modelling of basic as well as advanced models. As we had tuned the RT model in this section itself, we'll hyper tune the parameters with Random Search & Grid Search for Random Forest & Gradient Boost algorithms & then we'll come to a conclusion.

Hyperparameter Tuning

Random Forest – Random Search CV


```
In [265]: # We'll first define our RF regressor & also define our hyperparameters to be tuned
          model = RandomForestRegressor()
          n_estimators = [10, 100, 400] # n_estimators to be iterated between 10 to 400 trees
          max_features = [1, 2, 3] # We've already identified, only distance feature is contributing towards our prediction
          max_depth = [5, 7, 9] # Our initial RF model showed best result at depth =7

In [267]: # Next we'll define our grids with our created lists of hyperparameters
          grid = dict(n_estimators=n_estimators,max_features=max_features,max_depth=max_depth)
          # Then we'll define our Random search object with cross validation
          RS_RF = RandomizedSearchCV(model, param_distributions = grid, n_iter = 5, cv = 5, random_state=0)
          # model --> The RF regressor model
          # param_distributions --> The grid list to hyperparameters
          # n_iter --> The no. of parameter settings which are sampled 5 times
          # cv --> Cross validations with stratified-K-folds, 5 in our case, maintaining the percentage of samples in each fold
          RS_RF = RS_RF.fit(X_train,y_train) # Fit the hyperparamater tuned model
          pred_model = RS_RF.predict(X_test) # Predict the tuned model with test data

In [268]: BP = RS_RF.best_params_ # Storing the best parameters values after cross validations

In [270]: BP # Listing the best parameter values at n_estimators = 3, max_features = 3, max_depth =9

Out[270]: {'n_estimators': 10, 'max_features': 3, 'max_depth': 9}

In [271]: BE = RS_RF.best_estimator_ # Storing the best estimator values after cross validations

In [272]: BE # Listing the best estimator values at n_estimators = 10, max_features = 3, max_depth =9

Out[272]: RandomForestRegressor(max_depth=9, max_features=3, n_estimators=10)
```

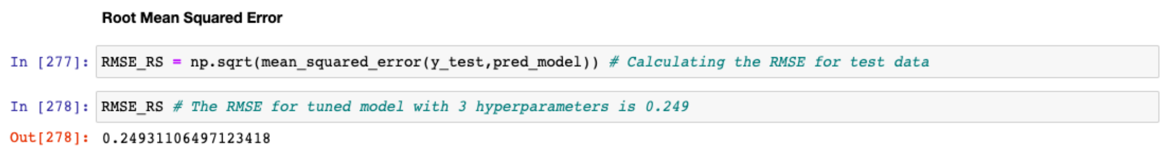
We've built the model with 3 hyperparameters. The n_estimator, max_features and max_depth. The reason behind choosing these are:

The n_estimators parameter determines the number of trees in a forest, which essentially means it controls the spread of our model building which means greater control.

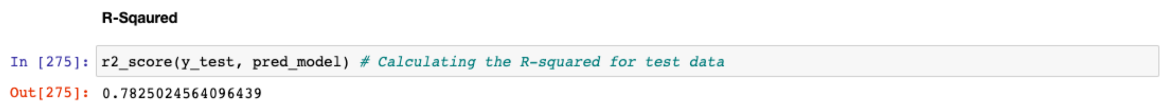
The max_features parameter determines the number of features to be included in the building of random forest. Here we know that distance is only the decisive parameter so it would be a good idea to combine it with the others to test how effective others are.

The max_depth parameter determines the growth of a tree in a random forest. Even though it is different from pruning, it essentially avoids overfitting of data as it uses the MSE for deciding on splits.

Let us check the model performance.


Root Mean Squared Error

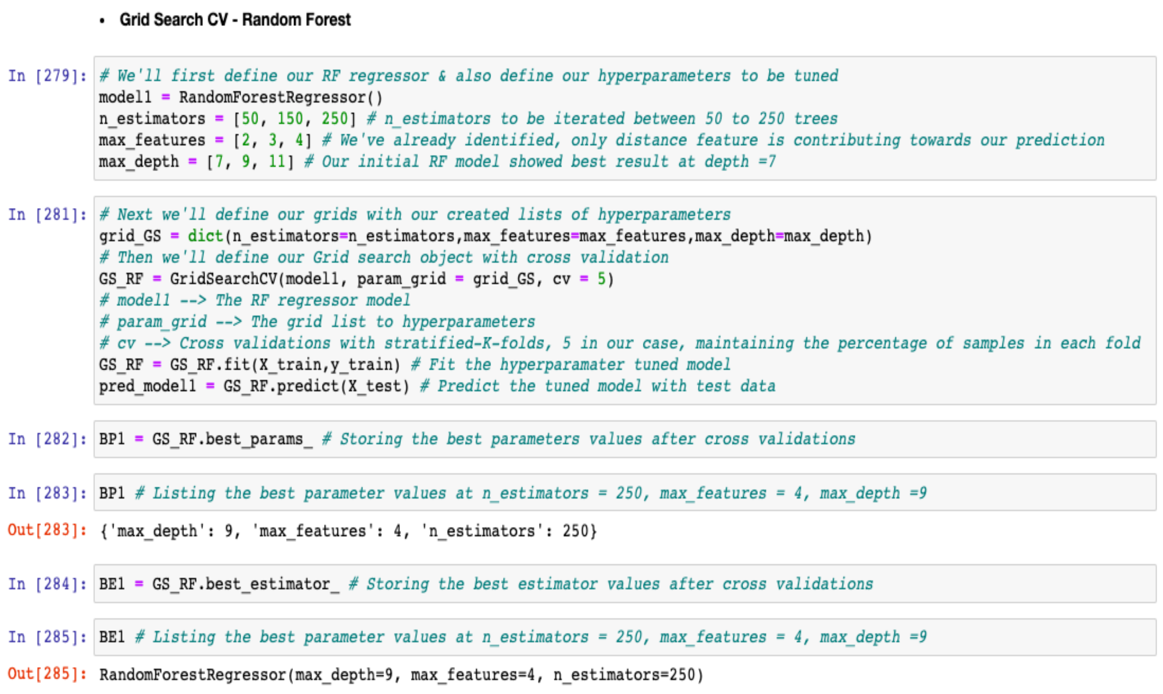
```
In [277]: RMSE_RS = np.sqrt(mean_squared_error(y_test,pred_model)) # Calculating the RMSE for test data

In [278]: RMSE_RS # The RMSE for tuned model with 3 hyperparameters is 0.249

Out[278]: 0.24931106497123418
```


R-Sqaured

```
In [275]: r2_score(y_test, pred_model) # Calculating the R-squared for test data

Out[275]: 0.7825024564096439
```

The RMSE value is: **24%** and R-Squared is: **78%** both of which are very good and correct estimates. Any values bettering more or less these values could mean an overfitted model.

Random Forest – Grid Search CV

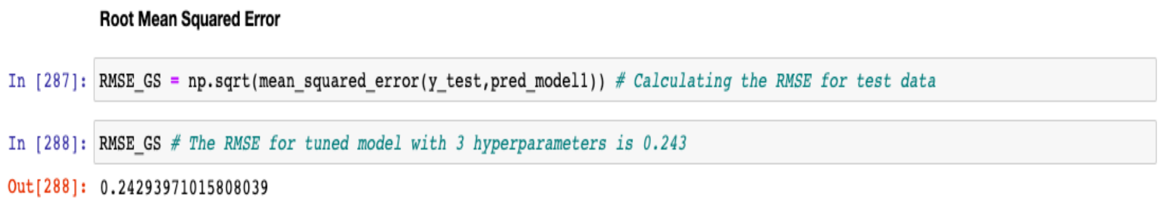


- **Grid Search CV - Random Forest**

```
In [279]: # We'll first define our RF regressor & also define our hyperparameters to be tuned
          model1 = RandomForestRegressor()
          n_estimators = [50, 150, 250] # n_estimators to be iterated between 50 to 250 trees
          max_features = [2, 3, 4] # We've already identified, only distance feature is contributing towards our prediction
          max_depth = [7, 9, 11] # Our initial RF model showed best result at depth =7

In [281]: # Next we'll define our grids with our created lists of hyperparameters
          grid_GS = dict(n_estimators=n_estimators,max_features=max_features,max_depth=max_depth)
          # Then we'll define our Grid search object with cross validation
          GS_RF = GridSearchCV(model1, param_grid = grid_GS, cv = 5)
          # model1 --> The RF regressor model
          # param_grid --> The grid list to hyperparameters
          # cv --> Cross validations with stratified-K-folds, 5 in our case, maintaining the percentage of samples in each fold
          GS_RF = GS_RF.fit(X_train,y_train) # Fit the hyperparamater tuned model
          pred_model1 = GS_RF.predict(X_test) # Predict the tuned model with test data

In [282]: BP1 = GS_RF.best_params_ # Storing the best parameters values after cross validations

In [283]: BP1 # Listing the best parameter values at n_estimators = 250, max_features = 4, max_depth =9

Out[283]: {'max_depth': 9, 'max_features': 4, 'n_estimators': 250}

In [284]: BE1 = GS_RF.best_estimator_ # Storing the best estimator values after cross validations

In [285]: BE1 # Listing the best parameter values at n_estimators = 250, max_features = 4, max_depth =9

Out[285]: RandomForestRegressor(max_depth=9, max_features=4, n_estimators=250)
```

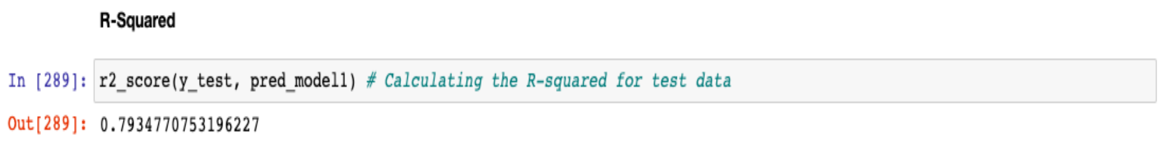
Here, we have retained the earlier 3 hyperparameters but have varied their ranges. Also, in the grid search function, we've fed the model with cross validations as 5, i.e., stratified k-folds. It divides the dataset into k-folds and iterates through the entire dataset unlike random sampling. Thus, ensuring that unbiased information is fed to the model and therefore greatly reducing the chances of overfitting the model.

We'll then find the best parameters as **max_depth = 9**, **max_features = 4** and **n_estimators as 250**. Let us now evaluate the performance



Root Mean Squared Error

```
In [287]: RMSE_GS = np.sqrt(mean_squared_error(y_test,pred_model1)) # Calculating the RMSE for test data

In [288]: RMSE_GS # The RMSE for tuned model with 3 hyperparameters is 0.243

Out[288]: 0.24293971015808039
```



R-Squared

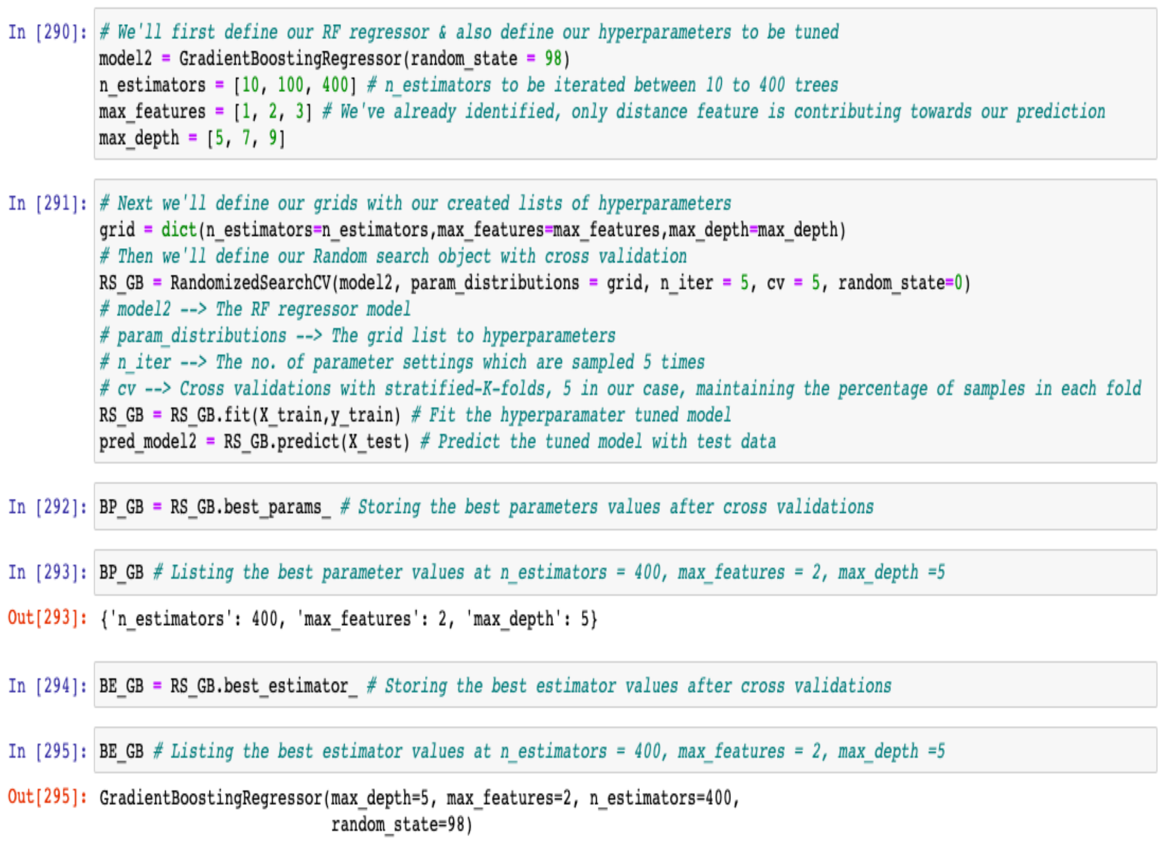
```
In [289]: r2_score(y_test, pred_model1) # Calculating the R-squared for test data

Out[289]: 0.7934770753196227
```

The RMSE for test data is **24% -** which is again pretty good keeping in mind the 5 iterations of cross validations on the dataset of ~12K observations.

The R-Squared value is **79%,** slight improvement from the Random Search CV.

This completes the modelling of Random Forest. Now we'll model for Gradient Boosting.

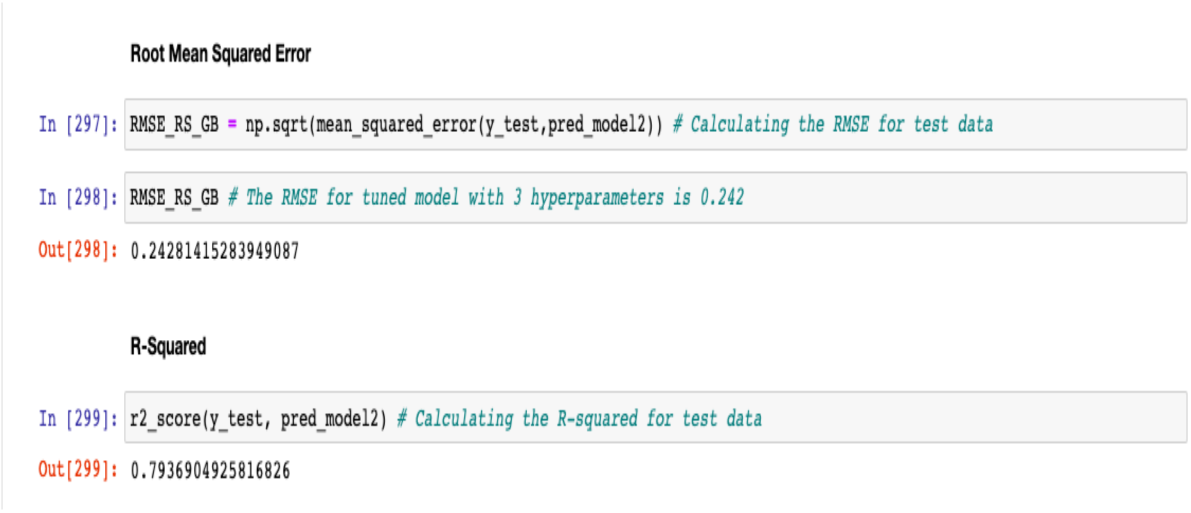
Gradient Boosting – Random Search CV



```
In [290]: # We'll first define our RF regressor & also define our hyperparameters to be tuned
          model2 = GradientBoostingRegressor(random_state = 98)
          n_estimators = [10, 100, 400] # n_estimators to be iterated between 10 to 400 trees
          max_features = [1, 2, 3] # We've already identified, only distance feature is contributing towards our prediction
          max_depth = [5, 7, 9]

In [291]: # Next we'll define our grids with our created lists of hyperparameters
          grid = dict(n_estimators=n_estimators,max_features=max_features,max_depth=max_depth)
          # Then we'll define our Random search object with cross validation
          RS_GB = RandomizedSearchCV(model2, param_distributions = grid, n_iter = 5, cv = 5, random_state=0)
          # model2 --> The RF regressor model
          # param_distributions --> The grid list to hyperparameters
          # n_iter --> The no. of parameter settings which are sampled 5 times
          # cv --> Cross validations with stratified-K-folds, 5 in our case, maintaining the percentage of samples in each fold
          RS_GB = RS_GB.fit(X_train,y_train) # Fit the hyperparamater tuned model
          pred_model2 = RS_GB.predict(X_test) # Predict the tuned model with test data

In [292]: BP_GB = RS_GB.best_params_ # Storing the best parameters values after cross validations

In [293]: BP_GB # Listing the best parameter values at n_estimators = 400, max_features = 2, max_depth =5

Out[293]: {'n_estimators': 400, 'max_features': 2, 'max_depth': 5}

In [294]: BE_GB = RS_GB.best_estimator_ # Storing the best estimator values after cross validations

In [295]: BE_GB # Listing the best estimator values at n_estimators = 400, max_features = 2, max_depth =5

Out[295]: GradientBoostingRegressor(max_depth=5, max_features=2, n_estimators=400,
                                    random_state=98)
```

The parameters are same and for the random search CV function we've passed the grid and again the 5 iterations along with 5 cross validations.

The best parameters are **max_features – 2**, **max_depth – 5.** Let's evaluate the performance.



Root Mean Squared Error

```
In [297]: RMSE_RS_GB = np.sqrt(mean_squared_error(y_test,pred_model2)) # Calculating the RMSE for test data

In [298]: RMSE_RS_GB # The RMSE for tuned model with 3 hyperparameters is 0.242

Out[298]: 0.24281415283949087
```

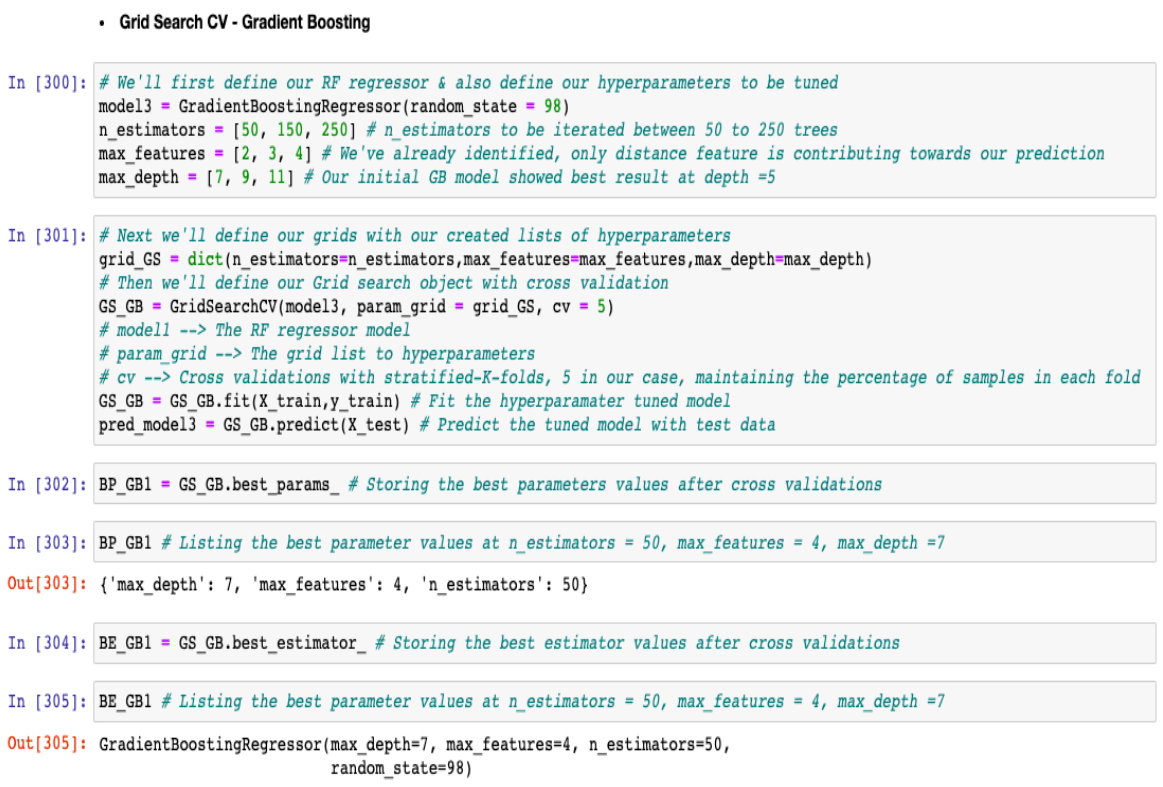
R-Squared

```
In [299]: r2_score(y_test, pred_model2) # Calculating the R-squared for test data

Out[299]: 0.7936904925816826
```

The RMSE is: **24%** and R-Squared is: **~80%.**

The model is performing similar to random forest – random search CV.

Gradient Boosting – Grid Search CV

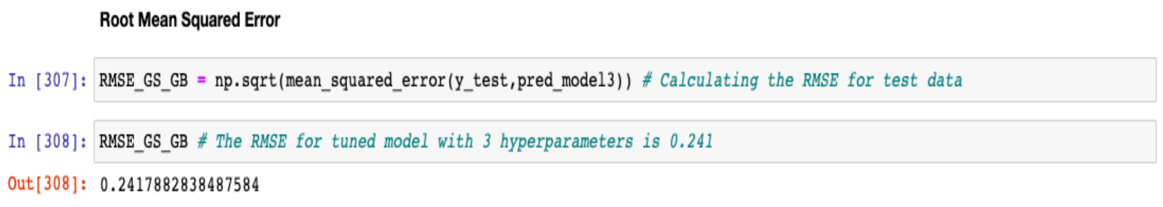


- **Grid Search CV - Gradient Boosting**

```
In [300]: # We'll first define our RF regressor & also define our hyperparameters to be tuned
          model3 = GradientBoostingRegressor(random_state = 98)
          n_estimators = [50, 150, 250] # n_estimators to be iterated between 50 to 250 trees
          max_features = [2, 3, 4] # We've already identified, only distance feature is contributing towards our prediction
          max_depth = [7, 9, 11] # Our initial GB model showed best result at depth =5

In [301]: # Next we'll define our grids with our created lists of hyperparameters
          grid_GS = dict(n_estimators=n_estimators,max_features=max_features,max_depth=max_depth)
          # Then we'll define our Grid search object with cross validation
          GS_GB = GridSearchCV(model3, param_grid = grid_GS, cv = 5)
          # model1 --> The RF regressor model
          # param_grid --> The grid list to hyperparameters
          # cv --> Cross validations with stratified-K-folds, 5 in our case, maintaining the percentage of samples in each fold
          GS_GB = GS_GB.fit(X_train,y_train) # Fit the hyperparamater tuned model
          pred_model3 = GS_GB.predict(X_test) # Predict the tuned model with test data

In [302]: BP_GB1 = GS_GB.best_params_ # Storing the best parameters values after cross validations

In [303]: BP_GB1 # Listing the best parameter values at n_estimators = 50, max_features = 4, max_depth =7

Out[303]: {'max_depth': 7, 'max_features': 4, 'n_estimators': 50}

In [304]: BE_GB1 = GS_GB.best_estimator_ # Storing the best estimator values after cross validations

In [305]: BE_GB1 # Listing the best parameter values at n_estimators = 50, max_features = 4, max_depth =7

Out[305]: GradientBoostingRegressor(max_depth=7, max_features=4, n_estimators=50,
                                    random_state=98)
```

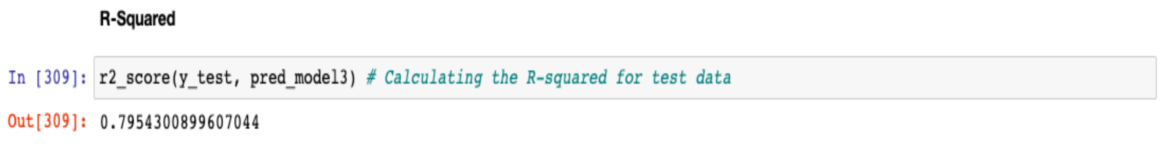
The parameters again are same as with previous model with 5 cross validations. The best parameters are **max_depth = 7**, **max_features = 4** and **n_estimators = 50**

Evaluating the performance.



Root Mean Squared Error

```
In [307]: RMSE_GS_GB = np.sqrt(mean_squared_error(y_test,pred_model3)) # Calculating the RMSE for test data

In [308]: RMSE_GS_GB # The RMSE for tuned model with 3 hyperparameters is 0.241

Out[308]: 0.2417882838487584
```



R-Squared

```
In [309]: r2_score(y_test, pred_model3) # Calculating the R-squared for test data

Out[309]: 0.7954300899607044
```

The RMSE for test data is: **24.17%**

The R-Squared for test data is: **79.54%**

This essentially completes the modelling phase. Moving on to summarizing all models.

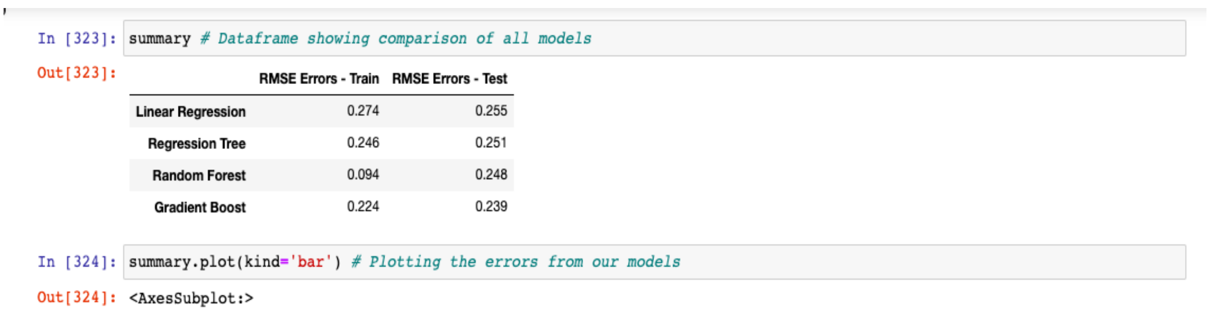
Model Summary

After running all the models, we've reached to conclude which one performed better and can be chosen over the others.

So, the evaluation criteria are the **RMSE values**, the **R-Squared values** and most importantly the **scope of further tuning** the model.

Linear Regression model performed very well, but there's little scope for further tuning thus improving on the metrics.

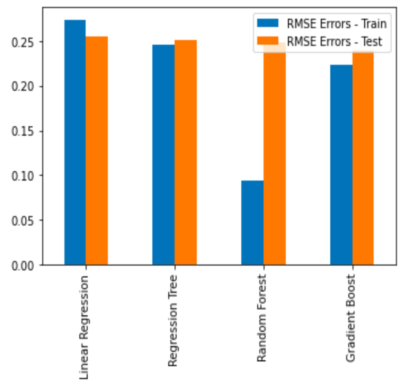
The ensemble models – all of them performed well with tuning the hyperparameters. However, the Random forest with hyper parameter tuning was perhaps the best. Let us quickly compare:



```
In [323]: summary # Dataframe showing comparison of all models
Out[323]:
```

	RMSE Errors - Train	RMSE Errors - Test
Linear Regression	0.274	0.255
Regression Tree	0.246	0.251
Random Forest	0.094	0.248
Gradient Boost	0.224	0.239

```
In [324]: summary.plot(kind='bar') # Plotting the errors from our models
Out[324]: <AxesSubplot:>
```



The Random Forest model performed well on the training data from which it learnt, and it is the only model to be outperforming others in that aspect.

After tuning the hyperparameters, the improvements by Random Forest are better.

Model	RMSE	R-Squared
Random Forest – RS-CV	~24%	~79%
Random Forest – GS-CV	~24%	~79%
Gradient Boosting – RS-CV	~24%	79.78%
Gradient Boosting – GS-CV	24.17%	79.54%

---End---

Instructions: Run the .rmd & .ipynb files.