Predicting Car Fare

Albert Abraham

26 August 2019

Contents

Introduction

1.1 Problem Statement

Predicting car fares based on historical data on car trips. In this problem we are given a training set of 16067 taxi trips in the train data and 9914 records in the test data. The goal of this challenge is to predict the fare of a taxi trip given information about the pickup and drop off locations, the pickup date time and number of passengers travelling.

1.2 Data

Our task is to build a regression model that will predict the cab fare based on multiple factors like: pickup and drop off locations, the pickup date time and number of passengers travelling.
Given below is a sample of dataset that we are going to use to build a regression model to predict cab fares:

```
## List first few rows (datapoints)
train_df.head()
```

	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

Below is some info about the data:

```
test_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9914 entries, 0 to 9913
Data columns (total 6 columns):
pickup_datetime      9914 non-null datetime64[ns, UTC]
pickup_longitude     9914 non-null float64
pickup_latitude      9914 non-null float64
dropoff_longitude    9914 non-null float64
dropoff_latitude     9914 non-null float64
passenger_count      9914 non-null int64
dtypes: datetime64[ns, UTC](1), float64(4), int64(1)
memory usage: 464.8 KB
```

Below is the statistical summary of our dataset:

```
train_df.describe()
```

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

Methodology

2.1 Data Wrangling

The data provided needs to be cleaned in order to be used for analysis and modeling.Data Wrangling deals with cleaning the data i.e dealing with missing values, NANs, wrong values and outliers. After cleaning the data the data can be pre-processed to make it ready for modelling i.e providing proper structure to data, finding new and useful variables for modeling.

- There are negative fares
- There is missing data (NA).
- How can there be 0 passengers?
- To be useful for exploration and prediction, I'll need to convert the dates from a timestamp object to individual columns for different aspects like, hour of the day, month, year, etc.
- We can cluster the geo-locations to get pickup and dropoff area
- We can derive the distance travelled in each trip

2.1.1 Missing Value Analysis

Missing values can be handled in different ways:

- Remove entire rows with missing values
- Derive missing values from other variables
- Replace missing values with mean of that variable
- Replace missing value with median value of that variable

We will replace the passenger_count with median value because it is the least influenced by outliers.
Fare_amount is an important variable as it the target variable, so we will not replace missing values in this variable, instead we will remove the rows with NA in fare_amount column.

```python
# Replace missing passenger_count values with median of passenger_count column
train_df[train_df.passenger_count.isnull()].head()
```

	fare_amount	pickup_datetime	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
31	22.54	2015-06-21 21:46:34 UTC	-74.010483	40.717667	-73.985771	40.660366	NaN
64	7.3	2011-11-07 10:47:40 UTC	-74.003919	40.753019	-73.992368	40.735362	NaN
82	8.5	2013-06-14 08:27:43 UTC	-73.953710	40.790813	-73.957015	40.777676	NaN
97	9	2014-12-07 12:26:00 UTC	-73.984977	40.752122	-74.000925	40.757982	NaN
112	35	2012-12-06 18:05:00 UTC	-73.953310	40.787772	-73.944352	40.719772	NaN

```python
train_df.loc[train_df.passenger_count.isnull(),['passenger_count']] = train_df.passenger_count.median()
```

```python
### remove records that contain NaN value
train_df = train_df.dropna(how = 'any', axis = 'rows')
test_df = test_df.dropna(how = 'any', axis = 'rows')
```

2.1.2 Fixing Data Types & Removing Incorrect Data

- We will convert the fare_amount into float data type
- Remove rows with passenger count < 1 and passenger_count > 7
- Remove rows with fare_amount < 0

```python
### change dtype of fare_amount to float
train_df['fare_amount'] = train_df['fare_amount'].values.astype(np.float64)
```

```python
#Remove rows fare_amount less than 0 and passenger count less than 0
train_df = train_df[(train_df['fare_amount'] > 0) & (train_df['passenger_count'] >= 1)]
test_df = test_df[test_df['passenger_count'] > 0]
```

```python
train_df.count()
```

```
fare_amount          15980
pickup_datetime      15980
pickup_longitude     15980
pickup_latitude      15980
dropoff_longitude    15980
dropoff_latitude     15980
passenger_count      15980
dtype: int64
```

2.1.3 Outlier Analysis

In statistics, an **outlier** is an observation point that is distant from other observations.
The above definition suggests that outlier is something which is separate/different from the crowd.
Outliers can skew the data so it is good to remove it before training the model.
For detection of outliers we can use mathematical methods, like z-scores, IQR, and graphical methods like box plots and scatter plots.
For Detection we have used boxplots and for outlier removal we have used IQR.

The **interquartile range** (**IQR**), also called the **midspread** or **middle 50%**, or technically **H-spread**, is a measure of statistical dispersion, being equal to the difference between 75th and 25th percentiles, or between upper and lower quartiles, IQR = $Q_3 - Q_1$.In other words, the IQR is the first quartile subtracted from the third quartile; these quartiles can be clearly seen on a box plot on the data.

It is a measure of the dispersion similar to standard deviation or variance, but is much more robust against outliers.

```
# Calculating Inter Quartile Range of train_df
Q1 = train_df.quantile(0.25)
Q3 = train_df.quantile(0.75)
IQR = Q3 - Q1
print(IQR)
```

```
fare_amount          6.500000
pickup_longitude     0.025320
pickup_latitude      0.032420
dropoff_longitude    0.027535
dropoff_latitude     0.033295
passenger_count      1.000000
dtype: float64
```

```
# Removing outliers from train_df
df_out = train_df[~((train_df < (Q1 - 1.5 * IQR)) |(train_df > (Q3 + 1.5 * IQR))).any(axis=1)]
```

2.1.4 Pre processing Data: Deriving Meaningful Variables

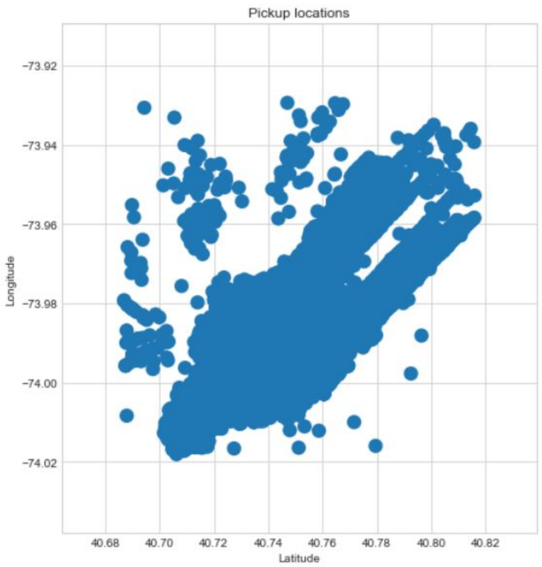
- We will split the pickup_datetime variable and derive several out of it that can be more useful like:
  - Year
  - Month
  - Day
  - DayOfWeek
  - Hour

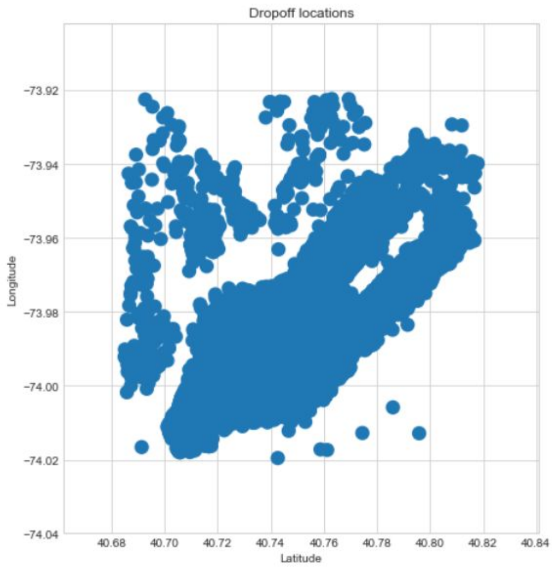
```
### function that splits datetime into categorical data
def add_datetime(df):
    df['pickup_datetime'] = pd.to_datetime(df['pickup_datetime'], format="%Y-%m-%d %H:%M:%S UTC")
    df['year'] = df.pickup_datetime.dt.year
    df['month'] = df.pickup_datetime.dt.month
    df['day'] = df.pickup_datetime.dt.day
    df['hour'] = df.pickup_datetime.dt.hour
    df['dayOfWeek'] = df.pickup_datetime.dt.dayofweek

    return df
```

- We will derive 'distance' variable from the pickup and dropoff geo-locations i.e using variables 'pickup_latitude', 'pickup_longitude', 'dropoff_latitude', 'dropoff_longitude'.

```
### function that calcuates distance between two locations
def getDistance(lat1,lon1,lat2,lon2):
    r = 6373 # earth's radius
    lat1 = np.deg2rad(lat1)
    lon1 = np.deg2rad(lon1)
    lat2 = np.deg2rad(lat2)
    lon2 = np.deg2rad(lon2)

    dlat = lat2 - lat1
    dlon = lon2 - lon1

    a = np.sin(dlat/2)**2 + np.cos(lat1) * np.cos(lat2) * np.sin(dlon/2)**2
    c = 2 * np.arctan2(np.sqrt(a), np.sqrt(1-a))
    distance = r*c

    return distance
```

- We will derive 'pickup_area' and 'dropoff_area' by clustering clustering our pickup and dropoff locations. We could have used k-means clustering but instead HDBSCAN clustering algo, which is an extension of DBSCAN clustering algorithm is used because it has an option to cluster geo-locations using haversine function, which helps in calculating the distance between two points on earth's surface.






```
### function that get a clusterer, using HDBScan technique
def add_cluster(df):

    ### predict cluster
    pickup_area = hdbscan.approximate_predict(clusterer, np.radians(df[['pickup_latitude', 'pickup_longitude']].values)
    dropoff_area = hdbscan.approximate_predict(clusterer, np.radians(df[['dropoff_latitude', 'dropoff_longitude']].value

    df['pickup_area'] = pickup_area
    df['dropoff_area'] = dropoff_area

    del pickup_area
    del dropoff_area

    return df
```

- We will convert the geo-locations to radians from degree.

```
### function that convert latitudes and longtitudes to radians format
def convert_to_radians(df):
    df['pickup_latitude'] = np.deg2rad(df['pickup_latitude'].values)
    df['pickup_longitude'] = np.deg2rad(df['pickup_longitude'].values)
    df['dropoff_latitude'] = np.deg2rad(df['dropoff_latitude'].values)
    df['dropoff_longitude'] = np.deg2rad(df['dropoff_longitude'].values)
    return df
```

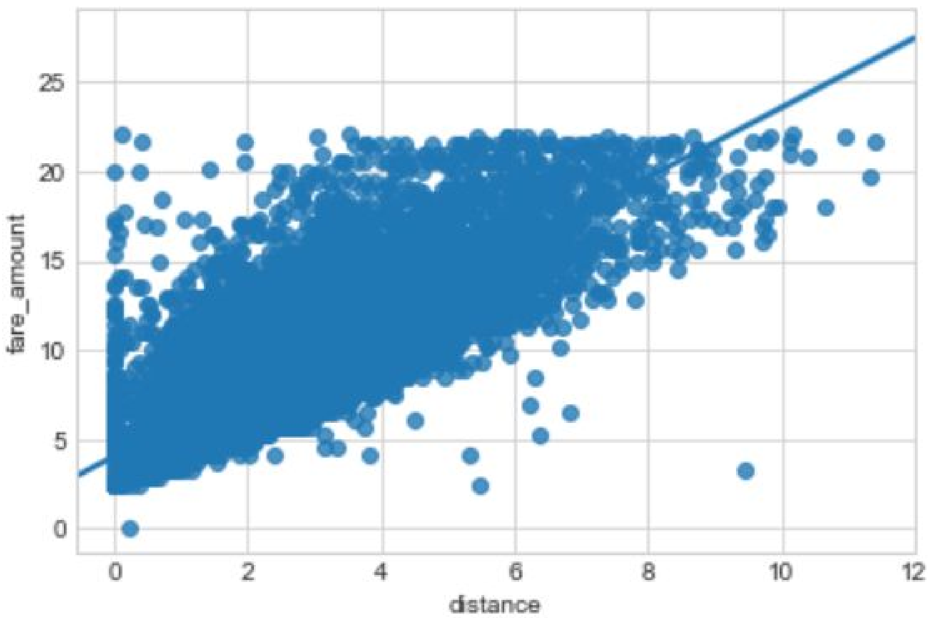
2.2 Exploratory Data Analysis

Exploratory Data Analysis refers to the critical process of performing initial investigations on data so as to discover patterns,to spot anomalies,to test hypothesis and to check assumptions with the help of summary statistics and graphical representations.

By exploratory data analysis suggested that distance is linearly correlated to fare_amount.

```
# Distance as potential predictor variable of price
sns.regplot(x="distance", y="fare_amount", data=df)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0xdac0609828>
```



2.3 Descriptive Data Analysis

Descriptive statistics are brief **descriptive** coefficients that summarize a given data set, which can be either a representation of the entire or a sample of a population. **Descriptive statistics** are broken down into measures of central tendency and measures of variability (spread).
We experimented with p-values of different variables to find their correlation with the target variable i.e fare_amount.

The P-value is the probability value that the correlation between these two variables is statistically significant. Normally, we choose a significance level of 0.05, which means that we are 95% confident that the correlation between the variables is significant.
By convention, when the:

- p-value is <0.001: we say there is strong evidence that the correlation is significant.
- the p-value is < 0.05: there is moderate evidence that the correlation is significant.
- the p-value is < 0.1: there is weak evidence that the correlation is significant.
- the p-value is > 0.1: there is no evidence that the correlation is significant.

2.3.1 Feature Selection

In machine learning and statistics, **feature selection**, also known as **variable selection**, **attribute selection** or **variable** subset **selection**, is the process of **selecting** a subset of relevant **features** (variables, predictors) for use in model construction.
We implemented feature selection using feature selection using feature selection library of sklearn, Recursive Feature Elimination and Ridge Regression.

```
# Import the necessary libraries first
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_classif
```

```
X = df[df.columns[2:]]
Y = df['fare_amount']
```

```
# Feature extraction
test = SelectKBest(score_func=f_classif, k=4)
fit = test.fit(X, Y)

# Summarize scores
np.set_printoptions(precision=3)
print(fit.scores_)

features = fit.transform(X)
# Summarize selected features
print(features[0:5,:])
```

```
[   1.345    1.731    1.294    2.241    1.003    1.249    1.096 225.843 114.529
    1.764    0.983    1.579    0.88 ]
[[7.118e-01 8.453e+00 2.010e+03 1.000e+00]
 [7.112e-01 1.390e+00 2.011e+03 8.000e+00]
 [7.114e-01 2.800e+00 2.012e+03 4.000e+00]
 [7.118e-01 2.000e+00 2.010e+03 3.000e+00]
 [7.114e-01 3.788e+00 2.011e+03 1.000e+00]]
```

Recursive Feature Elimination

```
# Import your necessary dependencies
from sklearn.feature_selection import RFE
from sklearn.linear_model import LinearRegression
```

```
# Feature extraction
model = LinearRegression()
rfe = RFE(model, 3)
fit = rfe.fit(X, Y)
print("Num Features: %s" % (fit.n_features_))
print("Selected Features: %s" % (fit.support_))
print("Feature Ranking: %s" % (fit.ranking_))
```

```
Num Features: 3
Selected Features: [ True False  True  True False False False False False False False False
 False]
Feature Ranking: [ 1  2  1  1  5 11 10  3  4  6  9  8  7]
```

Ridge Regression

```
# First things first
from sklearn.linear_model import Ridge
```

```
ridge = Ridge(alpha=1.0)
ridge.fit(X,Y)
```

```
Ridge(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=None,
   normalize=False, random_state=None, solver='auto', tol=0.001)
```

```
# A helper method for pretty-printing the coefficients
def pretty_print_coefs(coefs, names = None, sort = False):
    if names == None:
        names = ["X%s" % x for x in range(len(coefs))]
    lst = zip(coefs, names)
    if sort:
        lst = sorted(lst,  key = lambda x:-np.abs(x[0]))
    return " + ".join("%s * %s" % (round(coef, 3), name)
                               for coef, name in lst)
```

```
print ("Ridge model:", pretty_print_coefs(ridge.coef_))
```

```
Ridge model: -0.134 * X0 + -0.264 * X1 + -0.395 * X2 + -1.005 * X3 + 0.127 * X4 + 0.0 * X5 + 0.0 * X6 + 1.964 * X7 +
0.366 * X8 + 0.056 * X9 + -0.001 * X10 + 0.011 * X11 + -0.03 * X12
```

2.4 Model Development

After wrangling, exploring, analysing and preprocessing the data now it's finally time to model the data. As discussed earlier this is a regression problem where we want to generate a model that can predict car fares. We will use three models:

1. Multiple Linear Regression Model
2. Random Forest Model
3. Gradient Boosting Model

2.4.1 Multiple Linear Regression Model

Multiple linear regression (MLR), also known simply as multiple regression is a statistical technique that uses several explanatory variables to predict the outcome of a response variable. The goal of multiple linear regression (MLR) is to model the linear regression between the explanatory (independent) variables and response (dependent) variable.

The Formula for Multiple Linear Regression Is

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_p x_{ip} + \epsilon$$

where, for $i = n$ observations:

$y_i = \text{dependent variable}$

$x_i = \text{expanatory variables}$

$\beta_0 = \text{y-intercept (constant term)}$

$\beta_p = \text{slope coefficients for each explanatory variable}$

$\epsilon = \text{the model's error term (also known as the residuals)}$

1. Multiple Linear Regression ¶

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

```
from sklearn.model_selection import train_test_split
```

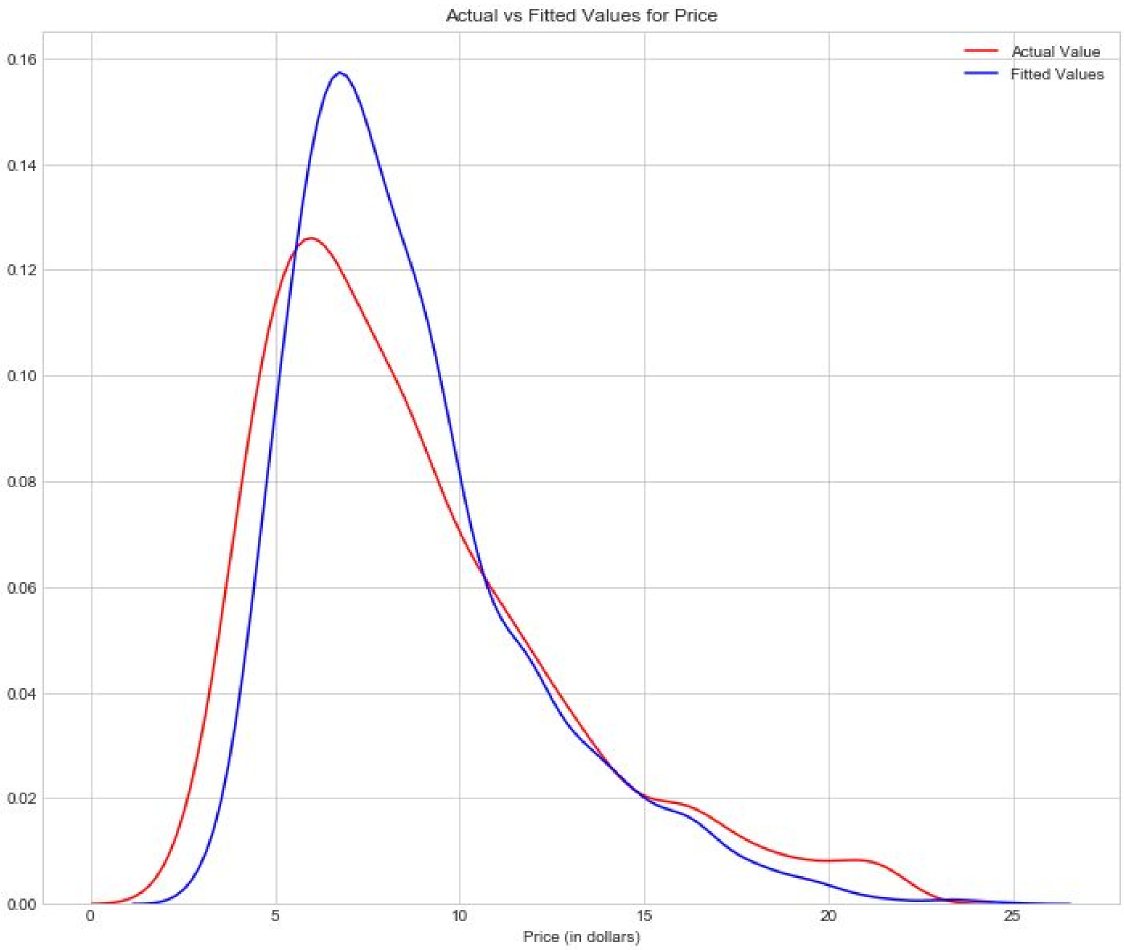
```
lm = LinearRegression()
lm
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize=False)
```

```
x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=0.25, random_state=0)
```

```
lm.fit(x_train, y_train)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize=False)
```



```
Yhat = lm.predict(x_test)
```

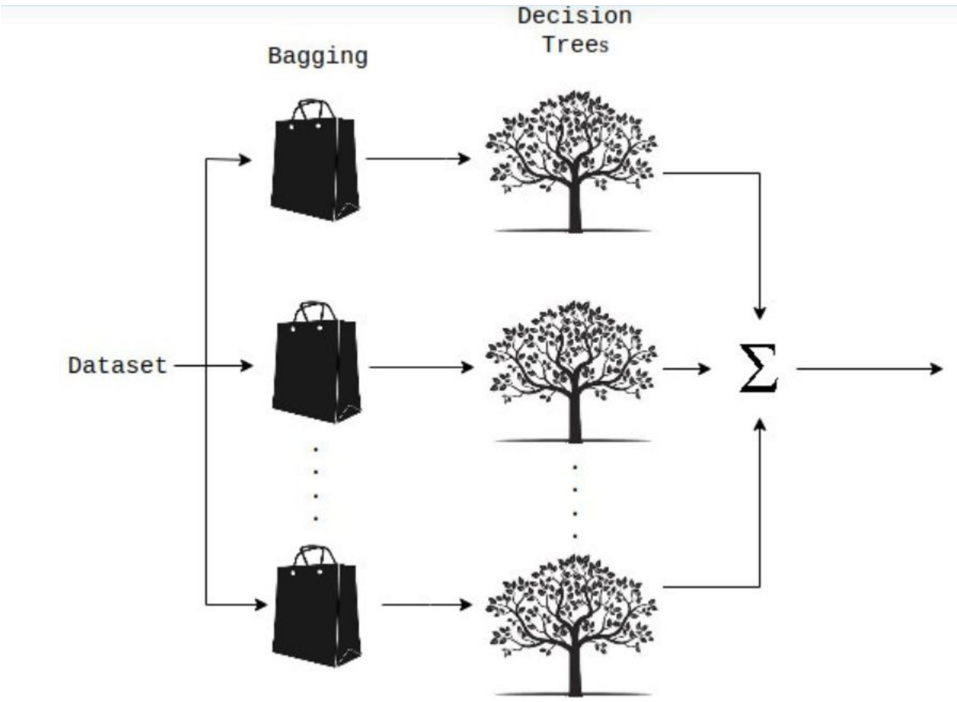
```
# Use score method to get accuracy of model
score = lm.score(x_test, y_test)
print(score)
```

```
0.713794747092104
```

2.4.2 Random Forest

A random forest is a meta estimator that fits a number of classifying decision trees on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting.

A Random Forest is an ensemble technique capable of performing both regression and classification tasks with the use of multiple decision trees and a technique called **Bootstrap Aggregation**, commonly known as **bagging**. Bagging, in the Random Forest method, involves training each decision tree on a different data sample where sampling is done with replacement.



Random Forest Regression: Process

2. Random Forest Regression

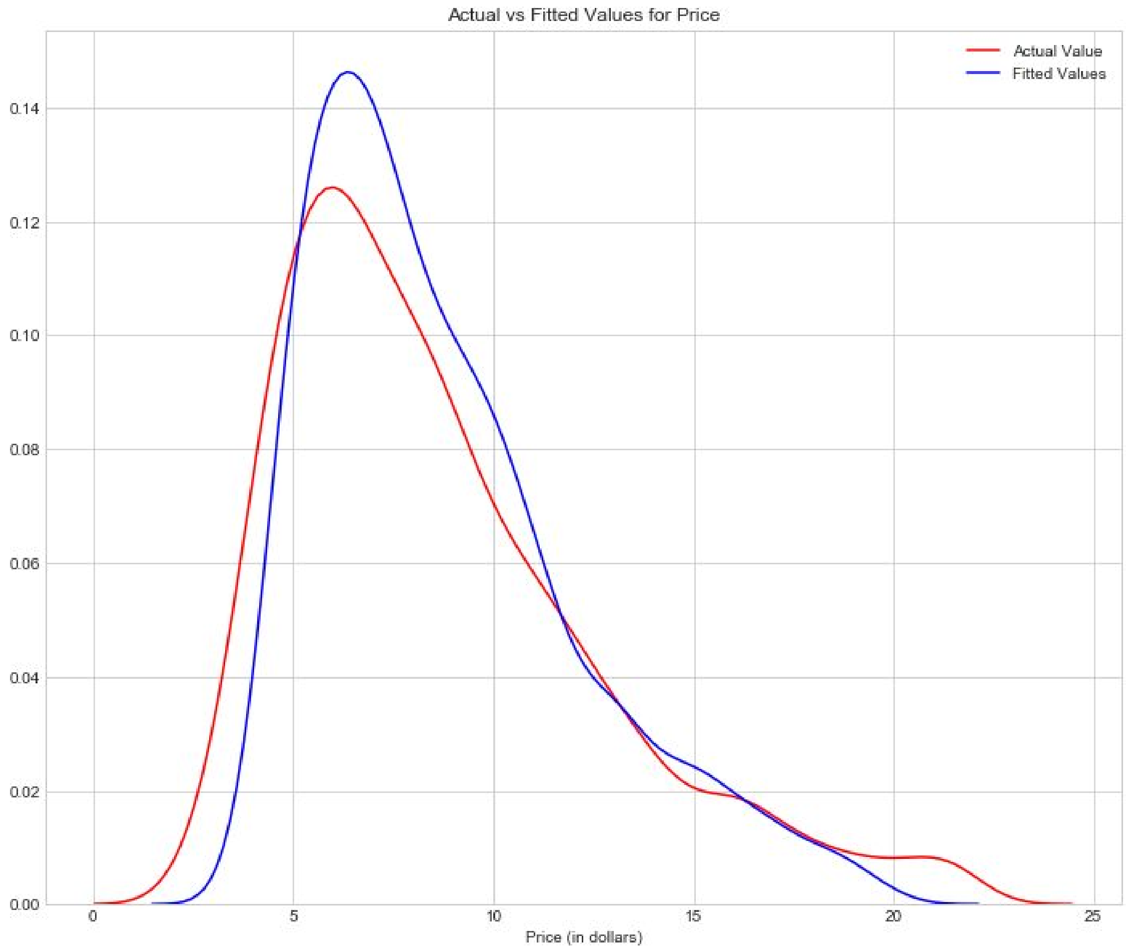
```
# Fitting Random Forest Regression to the dataset
# import the regressor
from sklearn.ensemble import RandomForestRegressor
```

```
x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=0.25, random_state=0)
```

```
# create regressor object
regressor = RandomForestRegressor(n_estimators = 100, random_state = 0)
```

```
# fit the regressor with x and y data
regressor.fit(x_train, y_train)
```

```
RandomForestRegressor(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=100, n_jobs=None,
           oob_score=False, random_state=0, verbose=0, warm_start=False)
```



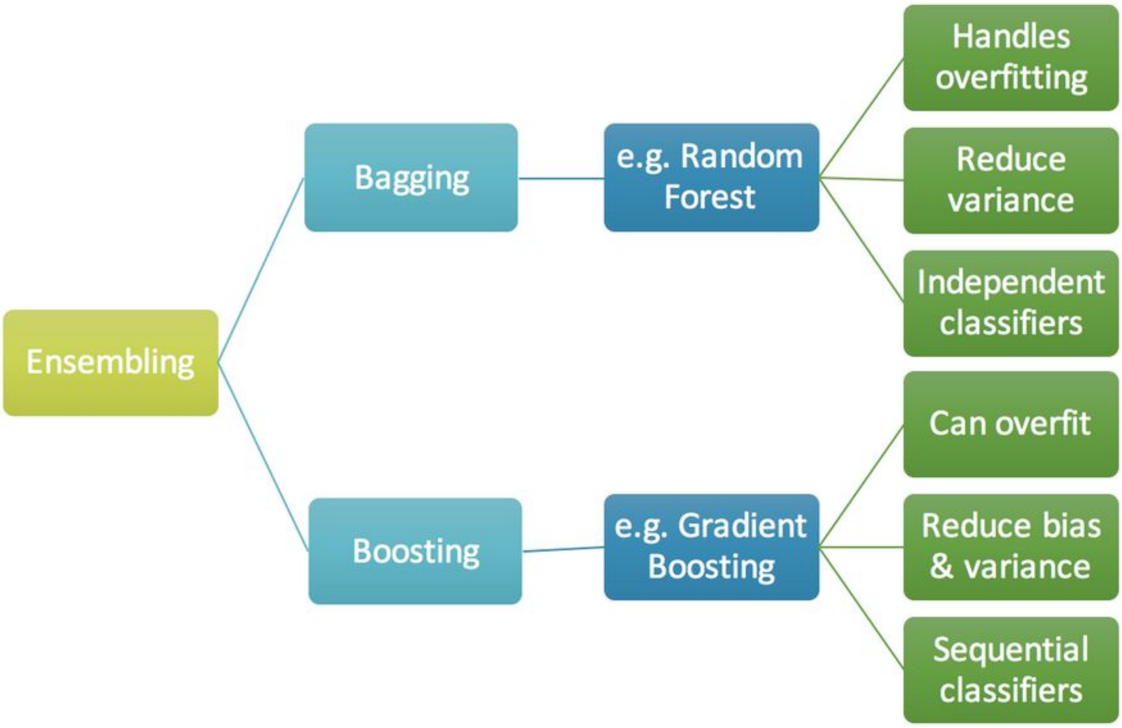
```
predictions = regressor.predict(x_test)
```

```
# Use score method to get accuracy of model
score = regressor.score(x_test, y_test)
print(score)
```

```
0.7405938411585025
```

2.4.3 Gradient Boosting

Gradient boosting is a machine learning technique for regression and classification problems, which produces a prediction model in the form of an ensemble of weak prediction models, typically decision trees.

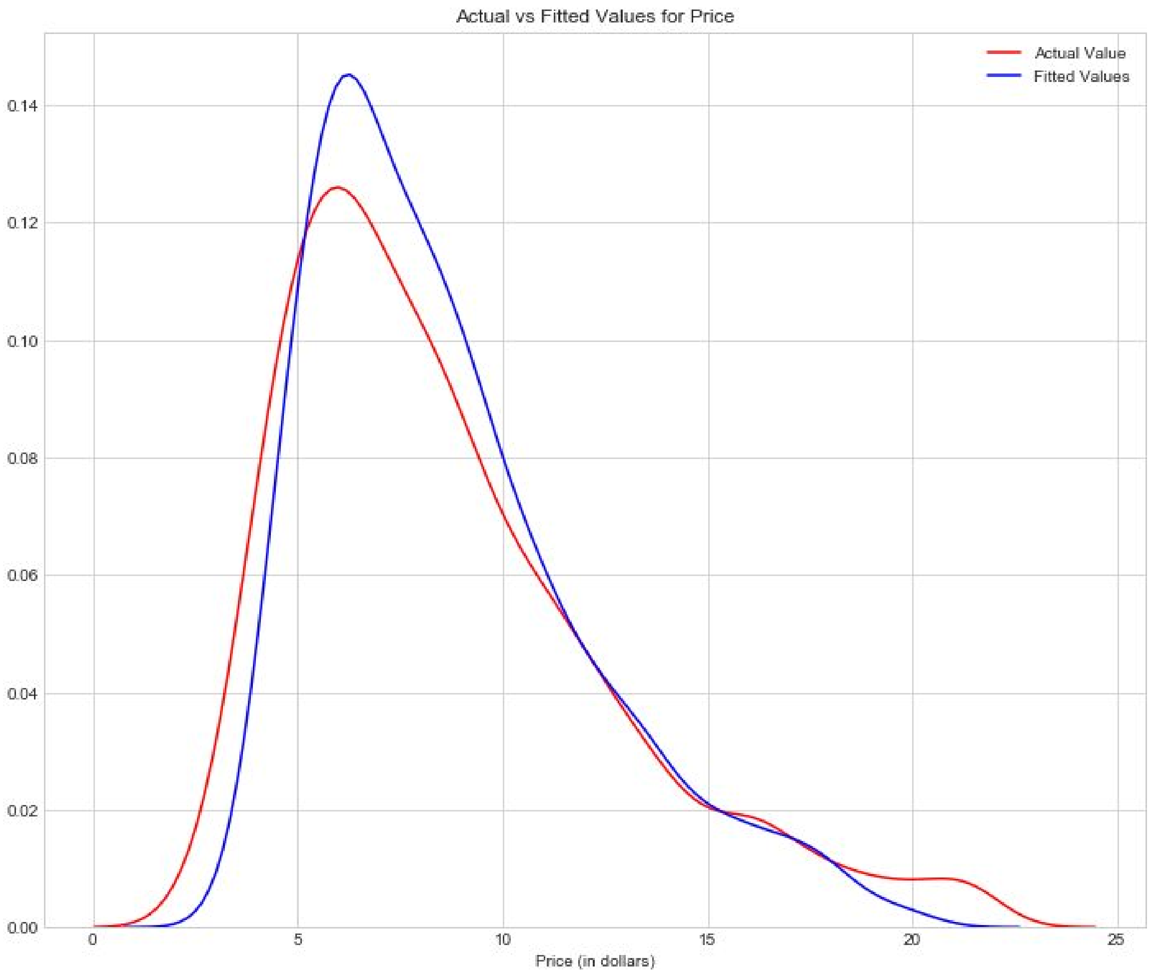


- In Python we have used lightgbm which is an implementation of gradient boosting, it's faster than XGBoost.
- In R we have used XGBoost as lihgtgbm package was not readily available from cran.

```python
### create lightgbm dataset
train_set = lgbm.Dataset(x_train, y_train, silent=False, categorical_feature=['year','month','day','dayOfWeek','hour'])
test_set = lgbm.Dataset(x_test, y_test, silent=False, categorical_feature=['year','month','day','dayOfWeek','hour'])
```

```python
### parameter for lightgbm model
params = {
    'boosting_type':'gbdt',
    'objective': 'regression',
    'nthread':16,
    'num_leaves': 31,
    'learning_rate': 0.03,
    'max_depth': 500,
    'subsample': 0.8,
    'bagging_fraction' : 1,
    'max_bin' : 5000 ,
    'bagging_freq': 30,
    'colsample_bytree': 0.6,
    'metric': 'rmse',
    'min_split_gain': 0.5,
    'min_child_weight': 1,
    'min_child_samples': 10,
    'scale_pos_weight':1,
    'zero_as_missing': False,
    'seed':0,
    'num_rounds':60000,
}
```

```python
### train the model
model = lgbm.train(params, train_set = train_set, valid_sets=test_set, num_boost_round=5000,early_stopping_rounds=500,ve
```



```
Training until validation scores don't improve for 500 rounds.
[100]	valid_0's rmse: 2.11222
[200]	valid_0's rmse: 1.95722
[300]	valid_0's rmse: 1.9307
[400]	valid_0's rmse: 1.91772
[500]	valid_0's rmse: 1.9133
[600]	valid_0's rmse: 1.91138
[700]	valid_0's rmse: 1.90678
[800]	valid_0's rmse: 1.90382
[900]	valid_0's rmse: 1.90163
[1000]	valid_0's rmse: 1.9014
[1100]	valid_0's rmse: 1.9014
[1200]	valid_0's rmse: 1.9014
[1300]	valid_0's rmse: 1.9014
[1400]	valid_0's rmse: 1.9014
Early stopping, best iteration is:
[905]	valid_0's rmse: 1.90135
```

```
model.best_score
```

```
defaultdict(dict, {'valid_0': {'rmse': 1.9013540632844097}})
```

```
predict = model.predict(x_test)
```

Conclusion

3.1 Model Evaluation

Mean Absolute Error (MAE) and Root mean squared error (RMSE) are two of the most common metrics used to measure accuracy for continuous variables.

Mean Absolute Error (MAE): MAE measures the average magnitude of the errors in a set of predictions, without considering their direction. It's the average over the test sample of the absolute differences between prediction and actual observation where all individual differences have equal weight.

$$\mathrm{MAE} = \frac{1}{n}\sum_{j=1}^{n}|y_j - \hat{y}_j|$$

Root mean squared error (RMSE): RMSE is a quadratic scoring rule that also measures the average magnitude of the error. It's the square root of the average of squared differences between prediction and actual observation.

$$\mathrm{RMSE} = \sqrt{\frac{1}{n}\sum_{j=1}^{n}(y_j - \hat{y}_j)^2}$$

Taking the square root of the average squared errors has some interesting implications for RMSE. Since the errors are squared before they are averaged, the RMSE gives a relatively high weight to large errors. This means the RMSE should be more useful when large errors are particularly undesirable.

Hence, we will use Root Mean Square Error (RMSE) as the evaluation metric as it fits our case and we want to avoid large errors.

1. Linear Regression

>> R-square Error

```
print('The R-square is: ', lm.score(x_test, y_test))
```

```
The R-square is:  0.713794747092104
```

>>Cross Validation

```
Rcross = cross_val_score(lm, X, Y, cv=4)
```

```
print("The mean of the folds are", Rcross.mean(), "and the standard deviation is" , Rcross.std())
```

```
The mean of the folds are 0.7024363685502308 and the standard deviation is 0.004322344286l6818
```

>>Mean Square Error(MSE)

```
mse = mean_squared_error(y_test, lm.predict(x_test))
print('The mean square error(MSE) of price and predicted value using multifit is: ', mse)
```

```
The mean square error(MSE) of price and predicted value using multifit is:  4.540229932701916
```

>>Root Mean Square Error (RMSE)

```
rmse = math.sqrt(mse)
print('The Root mean square error(RMSE) of price and predicted value using multifit is: ', rmse)
```

```
The Root mean square error(RMSE) of price and predicted value using multifit is:  2.130781530965086
```

>> RMSE for Linear Regression is **2.130781530965086**

2. Random Forest

>>R-Square Error

```
print('The R-square is: ', regressor.score(x_test, y_test))
```

```
The R-square is:  0.7405938411585025
```

>>Cross Validation

```
Rcross = cross_val_score(regressor, X, Y, cv=4)
```

```
print("The mean of the folds are", Rcross.mean(), "and the standard deviation is" , Rcross.std())
```

```
The mean of the folds are 0.7347222833972608 and the standard deviation is 0.004699891430294816
```

>>Mean Square Error(MSE)

```
mse = mean_squared_error(y_test, regressor.predict(x_test))
print('The mean square error(MSE) of price and predicted value using multifit is: ', mse)
```

```
The mean square error(MSE) of price and predicted value using multifit is:  4.115101295776747
```

>>Root Mean Square Error (RMSE)

```
rmse = math.sqrt(mse)
print('The Root mean square error(RMSE) of price and predicted value using multifit is: ', rmse)
```

```
The Root mean square error(RMSE) of price and predicted value using multifit is:  2.0285712449349043
```

>> RMSE for Random Forest is **2.0285712449349043**

3. Gradient Boosting: lightgbm

>> Mean Square Error(MSE) ¶

```
mse = mean_squared_error(y_test, model.predict(x_test))
print('The mean square error(MSE) of price and predicted value using multifit is: ', mse)
```

```
The mean square error(MSE) of price and predicted value using multifit is:  3.615147251668539
```

>>Root Mean Square Error (RMSE)

```
import math
rmse = math.sqrt(mse)
print('The Root mean square error(RMSE) of price and predicted value using multifit is: ', rmse)
```

```
The Root mean square error(RMSE) of price and predicted value using multifit is:  1.901354057420274
```

>> RMSE for Gradient Boosting is **1.901354057420274**

3.2 Final Verdict

We will select the **Gradient Boosting regression model** for **CAR FARE PREDICTION**, since it has the least Root Mean Square Error(RMSE).

We will use gradient boosting regression model for our prediction for the test data.

```
key = test_df['pickup_datetime']
test = test_df[test_df.columns[1:]]
```

```
### predict value
prediction = model.predict(test, num_iteration = model.best_iteration)

submission = pd.DataFrame({
    "key": key,
    "fare_amount": prediction
})

submission.to_csv('submission.csv',index=False)
```

The predictions will be saved in ‘submission.csv’ file.