Project - Cab Fare Prediction

Ayush Saini
Apr 9 · 11 min read

This is a supervised machine learning project. There's a cab rental company where they have already run a pilot program in city and now are ready to launch across the country. We have the historical data collected from the pilot program (train_cab.csv). We now have to make machine learning models to predict the cab fares.

Data provided:

- Train_cab.csv (1.1MB)
- Test.csv (667KB)

Data Exploration and Cleaning

```
##python code


#Load Libraries
import numpy as np
import pandas as pd
import os


#Visualisation
import matplotlib.pyplot as plt
```

```
import seaborn as sns


#setting working directory
os.chdir("F:/analytics_basics")
```

Getting Started

We start by importing libraries and setting up the working directory in both R and python.

```
##R code


#remove all the objects stored
rm(list=ls())


#set current working directory
setwd("F:/analytics_basics")
```

Now, we will load the *train_cab.csv* file and try to get a gist of data provided.

```
##Python code
# Load CSV
train_df = pd.read_csv("train_cab.csv")
train_df.head()
```

Out[266]:

	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

train_df.head()

```
##R code
train_df = read.csv("train_cab.csv", header = TRUE)
```

Describe Data

```
#python
train_df.dtypes
```

```
Out[237]: fare_amount          object
          pickup_datetime      object
          pickup_longitude    float64
          pickup_latitude     float64
          dropoff_longitude   float64
          dropoff_latitude    float64
          passenger_count     float64
          dtype: object
```

train_df.dtypes

```
#python
train_df['fare_amount'] =
pd.to_numeric(train_df['fare_amount'],errors='coerce')
train_df['pickup_datetime'] =
pd.to_datetime(train_df['pickup_datetime'], errors='coerce')


train_df.describe()
```

Out[239]:

	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

train_df.describe()

An effective method for catching outliers and anomalies is to find the summary statistics for the data using the `.describe()` method in python and `summary(train_df` in R.

```
##R
train_df$fare_amount =
as.numeric(as.character(train_df$fare_amount))


summary(train_df)
```

```
> summary(train_df)
  fare_amount                      pickup_datetime  pickup_longitude pickup_latitude  dropoff_longitude
 Min.   :   -3.00   2009-04-18 20:44:00 UTC:    2   Min.   :-74.44   Min.   :-74.01   Min.   :-74.43
 1st Qu.:    6.00   2009-05-10 17:57:00 UTC:    2   1st Qu.:-73.99   1st Qu.: 40.73   1st Qu.:-73.99
 Median :    8.50   2009-07-01 15:55:00 UTC:    2   Median :-73.98   Median : 40.75   Median :-73.98
 Mean   :   15.02   2009-07-28 13:37:00 UTC:    2   Mean   :-72.46   Mean   : 39.91   Mean   :-72.46
 3rd Qu.:   12.50   2009-12-10 15:37:00 UTC:    2   3rd Qu.:-73.97   3rd Qu.: 40.77   3rd Qu.:-73.96
 Max.   :54343.00   2009-12-11 11:56:00 UTC:    2   Max.   : 40.77   Max.   :401.08   Max.   : 40.80
 NA's   :25         (Other)                 :16055
 dropoff_latitude passenger_count
 Min.   :-74.01   Min.   :   0.000
 1st Qu.: 40.73   1st Qu.:   1.000
 Median : 40.75   Median :   1.000
 Mean   : 39.90   Mean   :   2.625
 3rd Qu.: 40.77   3rd Qu.:   2.000
 Max.   : 41.37   Max.   :5345.000
                  NA's   :55
> |
```

summary(train_df)

So, here we can see that there are some major anomalies in the data provided.
Firstly, fare_amount cannot be negative and max fare_amount is too large
the max pickup/dropoff latitude and longitude are too skewed.
maximum passenger count is also unrealistic

Removing missing values

We start off by firstly checking the missing data in our data provided.

```
print(df_train.isnull().sum())
```

```
fare_amount          25
pickup_datetime       1
pickup_longitude      0
pickup_latitude       0
dropoff_longitude     0
dropoff_latitude      0
passenger_count      55
dtype: int64
```

print(df_train.isnull().sum())

```
df_train = df_train.dropna(how = 'any', axis = 'rows')
#replace 0's in coordinates with null values
coord = ['pickup_longitude','pickup_latitude',
         'dropoff_longitude', 'dropoff_latitude']


for i in coord :
    train_df[i] = train_df[i].replace(0,np.nan)
    train_df    = train_df[train_df[i].notnull()]
```

Since the missing data is very less (around ~50) as compared to the total number of rows (around ~16k), I decided to drop them instead of calculating them through different methods like KNN, etc.
Also, I replaced 0s in lat/long with NA and removed the rows corresponding to that too.

```
#R Code
train_df[train_df == 0] <- NA
train_df<-na.omit(train_df)
```

Managing outliers

fare_amount

```
train_df['fare_amount'].value_counts().hist(color = 'b',
edgecolor = 'k');
plt.title('Fare amount distibution');
plt.xlabel('fare_amount in $'); plt.ylabel('Count');
```

We saw by using `.describe()` method that the fare_amount has the negative values which were impossible to have and while 3rd quartile was around $12.5, the maximum value was $54353 which is quite skewed keeping in mind the data provided is for a city.
We remove the outliers of our target variable by:

```
train_df = train_df[ (train_df["fare_amount"] > 0 ) &
                     (train_df["fare_amount"]  <

train_df["fare_amount"].quantile(.9999))]
```

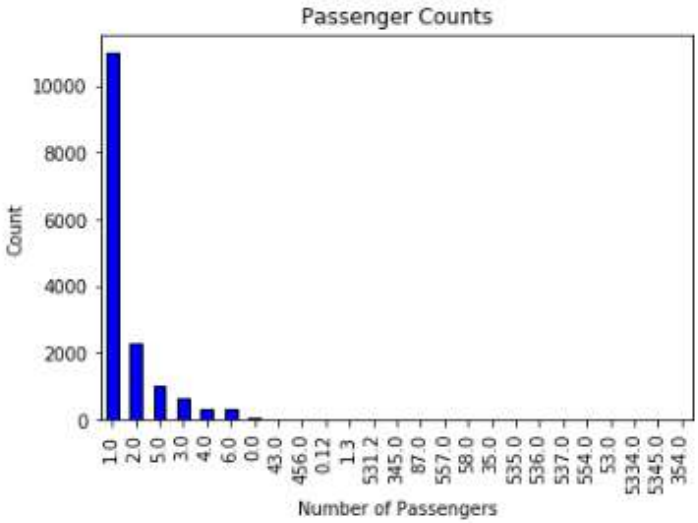
Here, we remove the rows having values less than 0 and greater than 99.99% of data.
Instead of using the usual method we are only removing the top 0.01% data, as the remaining data may contain useful information, we have only removed the which seemed impossible.

	fare_amount
count	15656.000000
mean	11.356744
std	10.775650
min	0.010000
25%	6.000000
50%	8.500000
75%	12.500000
max	453.000000

```
#R code
train_df = subset(train_df, (train_df$fare_amount > 0 ) &
(train_df$fare_amount <
quantile(train_df$fare_amount,.9999)))
```

passenger_count

```
train_df['passenger_count'].value_counts().plot.bar(color =
'b', edgecolor = 'k');
plt.title('Passenger Counts'); plt.xlabel('Number of
Passengers'); plt.ylabel('Count');
```



Based on the plot, we'll be removing rows with passenger_count >6. Also, keeping in mind the practicality, the average cab has up to 6 seats for passengers. So, we'll be removing all the rows in data having passenger_count greater than 6 and less than 1.

```
train_df = train_df[(train_df["passenger_count"] >=1 ) &
                          (train_df["passenger_count"] < 7) ]
```

	fare_amount	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
count	15889.000000	15889.000000	15889.000000	15889.000000	15889.000000	15889.000000
mean	11.243173	-72.478446	39.923339	-72.468662	39.901188	1.650028
std	9.235908	10.526993	6.812848	10.554969	6.180617	1.266244
min	0.010000	-74.438233	-74.006893	-74.227047	-74.006377	1.000000
25%	6.000000	-73.992139	40.734959	-73.991172	40.734765	1.000000
50%	8.500000	-73.981693	40.752640	-73.980157	40.753582	1.000000
75%	12.500000	-73.966824	40.767382	-73.963646	40.768026	2.000000
max	79.000000	40.766125	401.083332	40.802437	41.366138	6.000000

```
#R code
train_df = subset(train_df, (train_df$passenger_count >=1)
&(train_df$passenger_count < 7) )
```

here, we see that now variable passenger_count is a lot under control and practical.

location variables

As seen from the summary of train_df, the max values of lat/long are a bit skewed, therefore are the outliers in our data.

```
#python code
coords = ['pickup_longitude','pickup_latitude',
```

```
          'dropoff_longitude', 'dropoff_latitude']
for i in coord  :
    train_df = train_df[(train_df[i]   >
train_df[i].quantile(.001)) &
                        (train_df[i] <
train_df[i].quantile(.999))]
```

	fare_amount	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
count	15452.000000	15452.000000	15452.000000	15452.000000	15452.000000	15452.000000
mean	11.187415	-73.975549	40.751014	-73.974329	40.751677	1.650291
std	10.237500	0.033917	0.026412	0.032454	0.029324	1.266501
min	0.010000	-74.039047	40.640172	-74.177115	40.598020	1.000000
25%	6.000000	-73.992375	40.736726	-73.991352	40.736632	1.000000
50%	8.500000	-73.982093	40.753366	-73.980622	40.754278	1.000000
75%	12.500000	-73.968227	40.767784	-73.965661	40.768268	2.000000
max	453.000000	-73.422692	40.840073	-73.746250	40.873954	6.000000

```
#R code
coords =
c("pickup_longitude","pickup_latitude","dropoff_longitude","
dropoff_latitude")
for (i in coords){
  print(i)
  train_df = subset(train_df, (train_df[,i] <
quantile(train_df[,i],.999)))
}
```

```
> summary(train_df)
  fare_amount                  pickup_datetime  pickup_longitude pickup_latitude dropoff_longitude dropoff_latitude
 Min.   :  0.01   2009-04-18 20:44:00 UTC:    2   Min.   :-74.43   Min.   :39.60   Min.   :-74.43    Min.   : 0.7281
 1st Qu.:  6.00   2009-05-10 17:57:00 UTC:    2   1st Qu.:-73.99   1st Qu.:40.74   1st Qu.:-73.99    1st Qu.:40.7363
 Median :  8.50   2009-07-01 15:55:00 UTC:    2   Median :-73.98   Median :40.75   Median :-73.98    Median :40.7542
 Mean   : 11.29   2009-07-28 13:37:00 UTC:    2   Mean   :-73.98   Mean   :40.75   Mean   :-73.97    Mean   :40.7483
 3rd Qu.: 12.50   2009-12-10 15:37:00 UTC:    2   3rd Qu.:-73.97   3rd Qu.:40.77   3rd Qu.:-73.97    3rd Qu.:40.7682
 Max.   :453.00   2009-12-11 11:56:00 UTC:    2   Max.   :-73.42   Max.   :40.84   Max.   :-73.75    Max.   :40.8775
                  (other)                 :15505
 passenger_count
 Min.   :1.00
 1st Qu.:1.00
 Median :1.00
 Mean   :1.65
 3rd Qu.:2.00
 Max.   :6.00
```

The variables have started to look good and are within the expected range.

Feature engineering of location data

We now use feature engineering to create new variables *diff_longitude* and *diff_latitude* which gives the absolute difference between dropoff and pickup location.

```
#python code
def add_travel_vector_features(df):
    df['diff_longitude'] = (df.dropoff_longitude -
df.pickup_longitude).abs()
    df['diff_latitude'] = (df.dropoff_latitude -
df.pickup_latitude).abs()


add_travel_vector_features(train_df)
```

	fare_amount	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count	diff_longitude	diff_latitude
count	15452.000000	15452.000000	15452.000000	15452.000000	15452.000000	15452.000000	15452.000000	15452.000000
mean	11.187415	-73.975549	40.751014	-73.974329	40.751677	1.650291	0.022616	0.020971
std	10.237500	0.033917	0.026412	0.032454	0.029324	1.266501	0.033290	0.022028
min	0.010000	-74.039047	40.640172	-74.177115	40.598020	1.000000	0.000000	0.000000
25%	6.000000	-73.992375	40.736726	-73.991352	40.736632	1.000000	0.006083	0.007011
50%	8.500000	-73.982093	40.753366	-73.980622	40.754278	1.000000	0.012769	0.014183
75%	12.500000	-73.968227	40.767784	-73.965661	40.768268	2.000000	0.024106	0.026631
max	453.000000	-73.422692	40.840073	-73.746250	40.873954	6.000000	0.447625	0.208828

```
train_df$diff_longitude= abs(train_df$dropoff_longitude -
train_df$pickup_longitude)
train_df$diff_latitude= abs(train_df$dropoff_latitude -
train_df$pickup_latitude)
```

distance

We will be calculating the distance between the dropoff and pickup co-ordinates.
This uses the '**haversine**' formula to calculate the great-circle distance between two points—that is, the shortest distance over the earth's surface—giving an 'as-the-crow-flies' distance between the points (ignoring any hills they fly over, of course!).

Haversine formula:
The formula for Haversine distance is:

$$= 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos(\varphi_1)\cos(\varphi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

where r is the radius of the Earth. The end units will be in km.

```
#python code
#getting distance between pickup and dropoff location using
haversine function


#defining distance function
def getDistanceFromLatLonInKm(lat1,lon1,lat2,lon2) :
  R = 6371 #radius of earth
  dLat = deg2rad(lat2-lat1)
  dLon = deg2rad(lon2-lon1)
  a = np.sin(dLat/2) * np.sin(dLat/2) +
np.cos(deg2rad(lat1)) * np.cos(deg2rad(lat2)) *
np.sin(dLon/2) * np.sin(dLon/2)

  c = 2 * np.arctan2(np.sqrt(a), np.sqrt(1-a))
  d = R * c
  return d


def deg2rad(deg) :
    PI=22/7


return deg * (PI/180)


#calculating disatnce and storing in the distance variable
train_df['distance']=
getDistanceFromLatLonInKm(train_df['pickup_longitude'],train
_df['pickup_latitude'],train_df['dropoff_longitude'],train_d
f['dropoff_latitude'])
```

	fare_amount	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count	diff_longitude	diff_latitude	distance
count	15452.000000	15452.000000	15452.000000	15452.000000	15452.000000	15452.000000	15452.000000	15452.000000	15452.000000
mean	11.187415	-73.975549	40.751014	-73.974329	40.751677	1.650291	0.022616	0.020971	2.679946
std	10.237500	0.033917	0.026412	0.032454	0.029324	1.266501	0.033290	0.022028	3.705748
min	0.010000	-74.039047	40.640172	-74.177115	40.598020	1.000000	0.000000	0.000000	0.000000
25%	6.000000	-73.992375	40.736726	-73.991352	40.736632	1.000000	0.006083	0.007011	0.846936
50%	8.500000	-73.982093	40.753366	-73.980622	40.754278	1.000000	0.012769	0.014183	1.546984
75%	12.500000	-73.968227	40.767784	-73.965661	40.768268	2.000000	0.024106	0.026631	2.838392
max	453.000000	-73.422692	40.840073	-73.746250	40.873954	6.000000	0.447625	0.208828	49.800373

the minimum value of the distance variable is 0

```
#R code
#getting distance between pickup and dropoff location using
haversine function


getDistanceFromLatLonInKm <- function(lat1,lon1,lat2,lon2) {
  R = 6371 #radius of earth
  dLat = deg2rad(lat2-lat1)
  dLon = deg2rad(lon2-lon1)
  a = sin(dLat/2) * sin(dLat/2) + cos(deg2rad(lat1)) *
cos(deg2rad(lat2)) * sin(dLon/2) * sin(dLon/2)

  c = 2 * atan2(sqrt(a), sqrt(1-a))
  d = R * c
  return (d)
}


deg2rad <- function(deg)
{
  PI=22/7

  return (deg * (PI/180))
}


train_df$distance =
getDistanceFromLatLonInKm(train_df$pickup_longitude,train_df
$pickup_latitude,train_df$dropoff_longitude,train_df$dropoff
_latitude)
```

here we can see that the minimum value of the distance is ‘0’, we’ll be removing the rows containing the distance=0.

```
#python code
train_df = train_df[ (train_df["distance"] > 0 )]
```

	fare_amount	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count	diff_longitude	diff_latitude	distance
count	15312.000000	15312.000000	15312.000000	15312.000000	15312.000000	15312.000000	15312.000000	15312.000000	15312.000000
mean	11.207975	-73.975647	40.751022	-73.974416	40.751692	1.651274	0.022822	0.021163	2.704449
std	10.237703	0.033802	0.026377	0.032323	0.029318	1.267742	0.033371	0.022036	3.713740
min	0.010000	-74.039047	40.640172	-74.177115	40.598020	1.000000	0.000000	0.000000	0.000127
25%	6.000000	-73.992389	40.736726	-73.991354	40.736632	1.000000	0.006237	0.007208	0.862090
50%	8.500000	-73.982102	40.753429	-73.980634	40.754298	1.000000	0.012881	0.014367	1.561084
75%	12.500000	-73.968334	40.767793	-73.965760	40.768297	2.000000	0.024324	0.026806	2.859151
max	453.000000	-73.422692	40.839893	-73.746250	40.873954	6.000000	0.447625	0.208828	49.800373

```
#R code
train_df = subset(train_df, (train_df$distance > 0  ))
```

. . .

Developing Machine Learning models

Now, after treating our data with various methods and cleaning in many ways we are left with about 15k rows to work on.
Now that we have built a few potentially useful features, we can use them for machine learning: training an algorithm to predict the target from the features. We’ll start off with a basic model—Linear

Regression—only using a few features and then move on to more complex models and more features. There is a reason to believe that for this problem, even a simple linear model will perform well because of the strong linear correlation of the distances with the fare. We generally want to use the simplest—and hence most interpretable—model that is above an accuracy threshold (dependent on the application) so if a linear model does the job, there's no need to use a highly complex ensemble model. It's a best practice to start out with a simple model for just this reason!

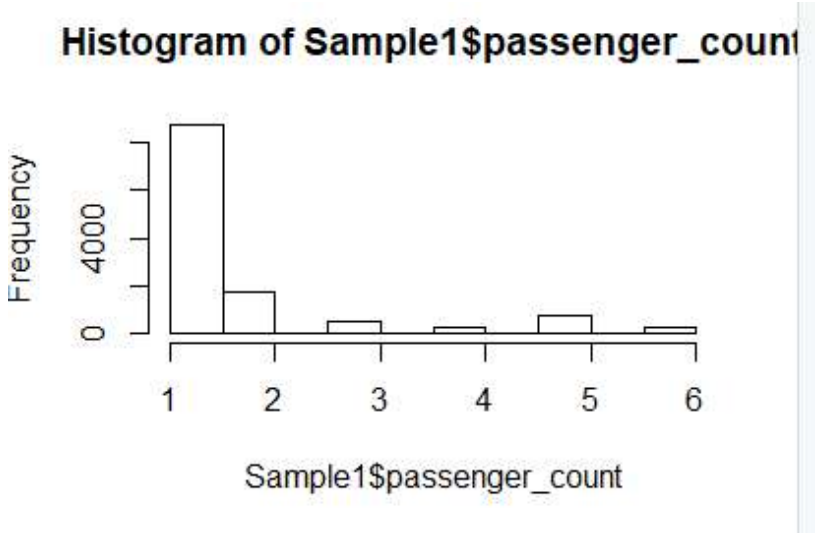
First Model: Linear Regression

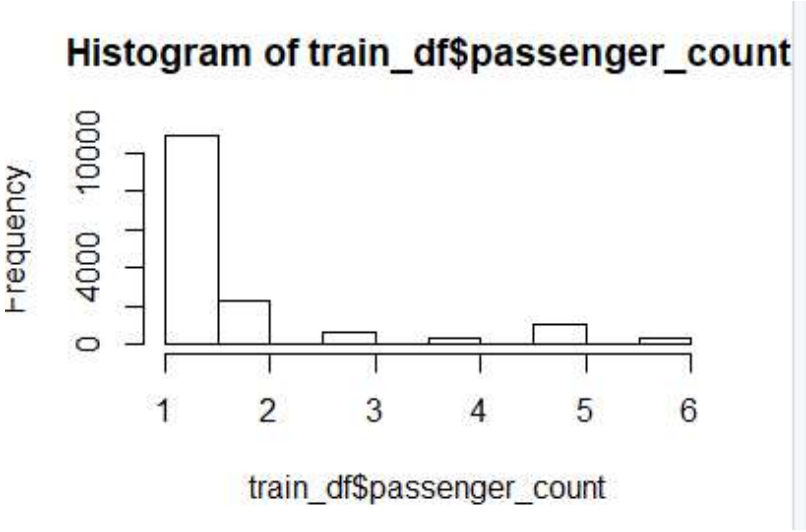
The first model we are using is Linear Regression. We'll be using diff_latitude, diff_longitude, passenger_count variables to train our model.
We'll first be dividing our given data in the ratio of 8:2. We will use 80% of data to train our model and rest 20% to validate and check the accuracy of our model.
We'll firstly form a new variable fare_bin to bin our fare_amount into levels each of level 5. This will help to sample and divide our data in a balanced way.

```
# Bin the fare and convert to string
train_df['fare_bin'] = pd.cut(train_df['fare_amount'], bins
= list(range(0, 50, 5))).astype(str)
# Uppermost bin
train_df.loc[train_df['fare_bin'] == 'nan', 'fare_bin'] =
'[45+]'


Rest, Sample = train_test_split(train_df, test_size = 0.8,
stratify = train_df['fare_bin'])
```

Sample(training data):80% data (~12290 rows)
Rest(test data):20% data (~3080 rows)





proving the sampled data has the same distribution of data

Now, training the model with “Sample”, which is our training data.

```
model = sm.OLS(Sample.iloc[:,0],
Sample.iloc[:,6:9].astype(float)).fit()
model.summary()
```

OLS Regression Results

Dep. Variable:	fare_amount	**R-squared:**	0.775
Model:	OLS	**Adj. R-squared:**	0.775
Method:	Least Squares	**F-statistic:**	1.409e+04
Date:	Tue, 09 Apr 2019	**Prob (F-statistic):**	0.00
Time:	15:37:46	**Log-Likelihood:**	-41735.
No. Observations:	12250	**AIC:**	8.348e+04
Df Residuals:	12247	**BIC:**	8.350e+04
Df Model:	3		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
passenger_count	1.2481	0.038	32.709	0.000	1.173	1.323
diff_longitude	185.1085	2.363	78.323	0.000	180.476	189.741
diff_latitude	178.1468	3.358	53.055	0.000	171.565	184.729

Omnibus:	34353.001	**Durbin-Watson:**	1.976
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	2367937010.690
Skew:	36.266	**Prob(JB):**	0.00
Kurtosis:	2155.664	**Cond. No.**	119.

```
#R code
#run regression model
lm_model = lm(fare_amount ~ passenger_count  +
abs_diff_longitude + abs_diff_latitude + distance, data =
Sample1)
```

now, predictions using the model.

```
#python code
predictions_LR = model.predict(Rest.iloc[:,6:9])


#R code
#Predict
predictions_LR = predict(lm_model, Rest1[,7:9])
```

now, calculating error metrics for linear regression

```
In [29]: #calculating different error metrics to check the accuracy of model
         #Calculate MAPE
         def MAPE(y_true, y_pred):
             mape = np.mean(np.abs((y_true - y_pred) / y_true))*100
             return mape
         #Calculate MAPE
         MAPE(Rest.iloc[:,0], predictions_LR)

Out[29]: 27.60284389340309

In [30]: #Calculate RMSE
         def RMSE(y_true, y_pred):
             rmse = np.sqrt(np.mean((y_true-y_pred)**2))
             return rmse
         #Calculate RMSE
         RMSE(Rest.iloc[:,0], predictions_LR)

Out[30]: 9.61241746077035
```

Python error metrics Output:
MAPE: 27.60284389340309
RMSE: 9.61241746077035

```
> MAPE(Rest1[,1], predictions_LR)
[1] 28.67132
>
> #Calculate RMSE
> RMSE = function(y_true, y_pred){
+   sqrt(mean((y_true-y_pred)**2))
+ }
> RMSE(Rest1[,1], predictions_LR)
[1] 11.98033
>
```

R error metrics Output:
MAPE: 28.67132
RMSE: 11.98033

Now, this model act as our base model to compare to. Without anything to compare these results to, we can't say if they are good. This is the reason for establishing a basic baseline with no machine learning!

Decision Tree

Decision Tree is a decision-making tool that uses a flowchart-like tree structure or is a model of decisions and all of their possible results, including outcomes, input costs and utility.

Decision-tree algorithm falls under the category of supervised learning algorithms. It works for both continuous as well as categorical output variables.
But here we are using "DecisionTreeRegressor" for predicting continuous variable "fare_amount"

making model

```
fit_DT =
DecisionTreeRegressor(max_depth=5).fit(Sample.iloc[:,6:9],
Sample.iloc[:,0])
```

predictions

```
predictions_DT = fit_DT.predict(Rest.iloc[:,6:9])
```

calculating error metrics

```
In [34]: #Calculate MAPE
         MAPE(Rest.iloc[:,0], predictions_DT)

Out[34]: 23.08186063052852

In [35]: #Calculate RMSE
         RMSE(Rest.iloc[:,0], predictions_DT)

Out[35]: 9.21390544843174
```

Python error metrics Output:
MAPE: 23.08186063052852
RMSE: 9.21390544843174

```
> MAPE(Rest1[,1], predictions_DT)
[1] 26.26407
> RMSE(Rest1[,1], predictions_DT)
[1] 11.97397
```

R error metrics output:
MAPE: 26.26407
RMSE: 11.97397

There's a significant improvement in the error metrics over the linear regression model.

Random Forest

For another non-linear model, we'll use the Random Forest regressor. This is a powerful ensemble of regression trees that has good performance and generalization ability because of its low variance. This model should have better error metrics as compared to the decision tree model.

making model

```
RF_model = RandomForestRegressor(n_estimators =
200).fit(Sample.iloc[:,6:9], Sample.iloc[:,0])
```

We're taking no of trees as 200 for both R and python model development.

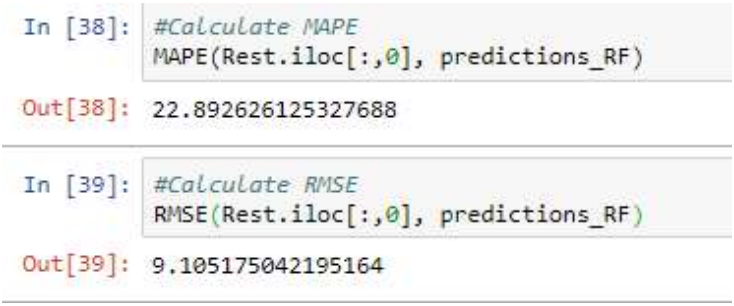
```
# Create a Random Forest model with default parameters
rf_model <- randomForest(fare_amount ~ passenger_count  +
diff_longitude + diff_latitude , data = Sample1, importance
= TRUE,ntree = 200)
```

predictions

```
#python code
predictions_RF = RF_model.predict(Rest.iloc[:,6:9])


#R code
#Predict for new test cases
predictions_RF = predict(rf_model, Rest1[,7:9])
```

calculating error metrics



Python error metrics Output:
MAPE: 22.892626125327688
RMSE: 9.105175042195164

```
> MAPE(Rest1[,1], predictions_RF)
[1] 26.96952
> RMSE(Rest1[,1], predictions_RF)
[1] 11.98548
>
```

R error metrics output:
MAPE: 26.96952
RMSE: 11.98548

The random forest does much better than the simple linear regression and Decision Tree. This indicates that the problem is not completely linear, or at least is not linear in terms of the features we have constructed.

KNN regression

In pattern recognition, the ***k*-nearest neighbors' algorithm** (***k*-NN**) is a non-parametric method used for classification and regression.[1] In both cases, the input consists of the *k* closest training examples in the feature space. The output depends on whether *k*-NN is used for classification or regression.
In *k-NN regression*, the output is the property value for the object. This value is the average of the values of *k* nearest neighbors.
Here, we are using "KNeighborsRegressor" for predicting "fare_amount".

making model

```
#python
KNN_model = KNeighborsRegressor(n_neighbors =
110).fit(Sample.iloc[:,6:9], Sample.iloc[:,0])
```

Here in KNN regression model, it is suggested to take no. of neighbors as the square root of the number of rows in the training set.
Here in Sample data set we have about 12290 rows which give square root of 10.8… , so we have taken k=110 for python model and k=111 for R model.

```
#R code
KNN_predictions = FNN::knn.reg(train = Sample1[,7:9], test =
Rest1[,7:10], y = Sample1$fare_amount, k = 111)
```

predictions

```
#python
predictions_KNN = KNN_model.predict(Rest.iloc[:,6:9])
```

in R the predictions are saved in KNN_predictions$pred

calculating error metrics

```
In [42]: #Calculate MAPE
         MAPE(Rest.iloc[:,0], predictions_KNN)

Out[42]: 21.375485769006175

In [43]: #Calculate RMSE
         RMSE(Rest.iloc[:,0], predictions_KNN)

Out[43]: 9.368085366033474
```

Python error metrics output:
MAPE: 21.375485769006175
RMSE: 9.368085366033474

```
> ###################################   KNN Regression ###################################################################
> KNN_predictions = FNN::knn.reg(train = Sample1[,7:9], test = Rest1[,7:9], y = Sample1$fare_amount, k = 111)
> MAPE(Rest1[,1], KNN_predictions$pred)
[1] 20.84076
> RMSE(Rest1[,1], KNN_predictions$pred)
[1] 12.04125
```

R error metrics output:
MAPE: 20.84076
RMSE: 12.04125

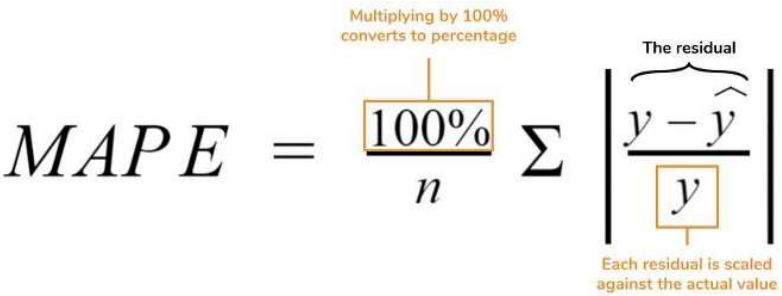
KNN regression model gives significantly better results than the other models.

Conclusion:

Lets firstly talk about error metrics. There are two major error metrics used by industry.

Mean absolute percentage error (**MAPE**), also known as **mean absolute percentage deviation** (**MAPD**), is a measure of prediction

accuracy of a forecasting method in statistics, for example in trend estimation, also used as a loss function for regression problems in machine learning. It usually expresses accuracy as a percentage, and is defined by the formula:



$$MAPE = \frac{100\%}{n} \sum \left| \frac{y - \hat{y}}{y} \right|$$

```
#python code
#Calculate MAPE
def MAPE(y_true, y_pred):
    mape = np.mean(np.abs((y_true - y_pred) / y_true))*100
    return mape
```

Root-mean-square deviation (RMSD) or **root-mean-square error (RMSE)** (or sometimes **root-mean-squared error**) is a frequently used measure of the differences between values (sample or population values) predicted by a model or an estimator and the values observed.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N} (Predicted_i - Actual_i)^2}{N}}$$

```
#python code
#Calculate RMSE
def RMSE(y_true, y_pred):
    rmse = np.sqrt(np.mean((y_true-y_pred)**2))
    return rmse
```

Similarities: Both MAPE and RMSE express average model prediction error in units of the variable of interest. They are negatively-oriented scores, which means lower values are better.

Differences: 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

MAPE is steady and RMSE increases as the variance associated with the frequency distribution of error magnitudes also increase. Also, RMSE is usually used for the predictions including the time-based data. MAPE is usually seen as more interpretable.
Therefore, we'll use MAPE error metric to score our algorithms.

Using MAPE error metric, we can see that **KNN regression model** gives the best results in terms of high accuracy and low error as compared to other models.

Predicting the values of fare_amount in test.csv

Now, we'll be predicting the unknown values of fare_amount of the other data set for which we have prepared our model

Python:

```
#Since KNN has minimum MAPE value, it is accepted as the
model to predict values of test.csv
# Load CSV
test = pd.read_csv("test.csv")
#checking null values in the data
print(test.isnull().sum())
```

```
pickup_datetime      0
pickup_longitude     0
pickup_latitude      0
dropoff_longitude    0
dropoff_latitude     0
passenger_count      0
dtype: int64
```

```
#adding diff_lat/long to test
add_travel_vector_features(test)
```

```
#predicting values
test['fare_amount'] = KNN_model.predict(test.iloc[:,5:8])
```

and in the end, saving it in a csv file.

```
# Writing a csv (output)
test.to_csv("test_new_py.csv", index = False)
```

R:

```
# Load CSV
test = read.csv("test.csv", header = TRUE)
#adding diff_lat/long to test
test$diff_longitude= abs(test$dropoff_longitude -
test$pickup_longitude)
test$diff_latitude= abs(test$dropoff_latitude -
test$pickup_latitude)
KNN_predictions2 = FNN::knn.reg(train = Sample1[,7:9], test
= test[,6:8], y = Sample1$fare_amount, k = 111)
test$fare_amount = KNN_predictions2$pred
write.csv(test, "test_new_R.csv", row.names = F)
```

In end, we have two files of predicted data from both Python and R.
test_new_py.csv
test_new_R.csv