Taxi Demand Prediction In New York City

A Capstone Project
PGP - ML program, Pune
Great Learning

Submitted by:
Archana
Prankur
Shoheb

Table of Contents

Abstract

At the outset we wanted to predict the number of pickups made by yellow taxis in New York city, at a given region and a given time interval. Later, to make the predictions more generalizable we included other factors of weather which could possibly be of importance while predicting the requirement of taxi services in a city. This could be of great interest to companies providing taxi services and city traffic planners. On the other hand, failure to predict the consumer requirement by large gaps could result in problems such as cabs being idle, traffic jams, surge pricing etc. The prediction results from such methods can be conveyed to the taxi drivers by smartphone apps and they can be diverted to their closest locations of highest demand ensuring a smooth flow of traffic and services. We used the NYC TLC trip dataset to build and test several regression models for prediction of demand. The random forest models outperformed all the other models that we tried to implement on this dataset with a coefficient of determination of 0.92 on the test dataset and 0.76 on the 2019 dataset.

Problem Definition

To predict the number of taxi pickups in regions of NYC in a given time interval, using historical data.

Methodology

In this project we proposed a prediction model for taxi demand in NYC based on historical data (2018- 2019). In the first stage we perform data cleaning and Exploratory Data Analysis based on pick-up and drop-off locations, trip duration and distance.

In the final stage the processed data is fed to the various regression models such as least squares regression, decision tree regression, random forest regression, neural network, XGBoost to predict demand as accurately as possible for each region in a specific time interval.

Evaluation Metrics

Root Mean Square Error

To evaluate the performance of our models, we have partitioned the data into a training set and testing set. We have planned a strategy of multiple iterations to evaluate multiple Models which is as below.

- Models performance for data of all Days.
- Models performance for Day wise data. We have evaluated models for Day 3 & 6

After considering a few different error metrics to evaluate our predictions, we have chosen RMSE for the same.
Root Mean Squared Error:
Root mean squared error (RMSE) is the square root of the mean of the square of all of the errors. The use of RMSE is very common, and it is considered an excellent general-purpose error metric for numerical predictions.

$$\mathrm{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(S_i - O_i\right)^2}$$

where Oi are the observations, Si predicted values of a variable, and n the number of observations available for analysis. RMSE is a good measure of accuracy.

R Squared or Coefficient of Determination

R-squared evaluates the scatter of the data points around the fitted regression line. It is also called the coefficient of determination, or the coefficient of multiple determination for multiple regression. For the same data set, higher R-squared values represent smaller differences between the observed data and the fitted values. R-squared is the percentage of the dependent variable variation that a linear model explains.

$$R^2 = \frac{\text{Variance explained by the model}}{\text{Total variance}}$$

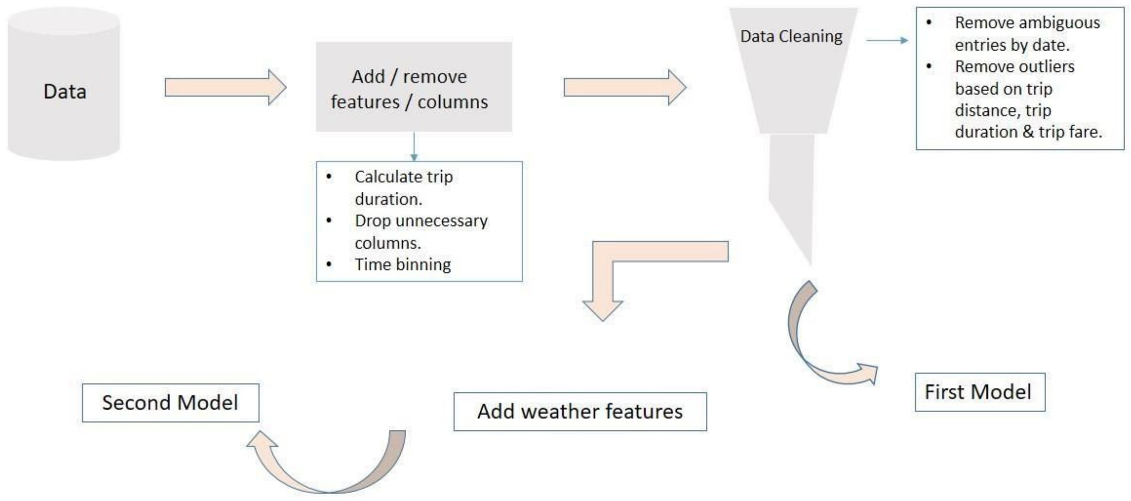
R-squared is always between 0 and 100%:

- 0% represents a model that does not explain any of the variation in the response variable around its mean. The mean of the dependent variable predicts the dependent variable as well as the regression model.

- 100% represents a model that explains all of the variation in the response variable around its mean.

Usually, the larger the R2, the better the regression model fits your observations.

Overview of The Final Process


Data
Add / remove features / columns
Calculate trip duration.
Drop unnecessary columns.
Time binning
Data Cleaning
Remove ambiguous entries by date.
Remove outliers based on trip distance, trip duration & trip fare.
First Model
Add weather features
Second Model

1. The data for the current analysis was retrieved from the NYC Taxi and Limousine Commission website: (https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page). Each row in the dataset corresponds to one trip of Yellow Medallion Taxicabs. These are the famous NYC yellow taxis that provide transportation exclusively through street-hails. The pickups are not pre-arranged. For the current problem we picked the data from January 2018 to December 2018.

2. We first removed the unnecessary features in the dataset which were not useful in predicting the taxi demand at a given location, to reduce the size of the dataset. The size of each month's data was around 700 megabytes. Hence columns like VendorID, passenger_count, RatecodeID, store_and_fwd_flag, tip_amount, tolls_amount, improvement_surcharge, mta_tax, payment_type were dropped from the analysis.

3. Our goal here is to predict the taxi demand in a 30-minute window at the given location, time of the day and day of the week, as accurately as possible. The problem statement for our analysis is formulated as follows:

   Input parameters: Pickup location, pickup time of the day, day of the week.

   Output: Predicted number of taxi pickups at the provided inputs.

4. Next we calculated the duration for each trip by subtracting the pickup time from the drop-off time. According to NYC Taxi & Limousine Commission Regulations the maximum allowed trip duration in a 24-hour interval is 12 hours.

5. The pickup time was binned into intervals of 30 minutes or the entire day. This was done because of our primary objective of predicting demand at a particular location in a thirty-minute time window.

6. For the data cleaning procedure, we first checked the dates contained in each month's dataset. Dates outside of the range of a given month were removed. Outlier removal was done on the basis of trip distance, trip duration and fare amount. The detailed procedure is shown in section *Data Cleaning*.

7. In the next step exploratory data analysis was performed which is shown in detail in section *EDA and Visualizations*. We also mapped the location ids given in the dataset to their respective geographical location on the map, to visualize the pickup and drop-off locations on the map.

8. We formed the first model using this dataset for only the month of April 2018 due to constraints of memory available for computing. We tried various regression models on this data, the details of which are shown in section xxxx. Later on we used the Dask library in Python to work with the entire year's dataset. Dask library is quite useful in scenarios where we have data size larger than the RAM size. Instead of loading all the data into the memory at once like Pandas, Dask divides the task in several partitions and works on them by loading them chunk by chunk and processing them parallelly.

9. To improve the generalizability of models we added the weather data for the same time period in New York city, since we thought that weather conditions might influence the taxi ridership at a given time. The details of the weather API is given in section *Weather data generation* .

10. In the final model we used the **Random Forest Regressor** algorithms to predict the ridership.

Project Repository

Ref: https://github.com/archanavpatil/taxi-demand-prediction

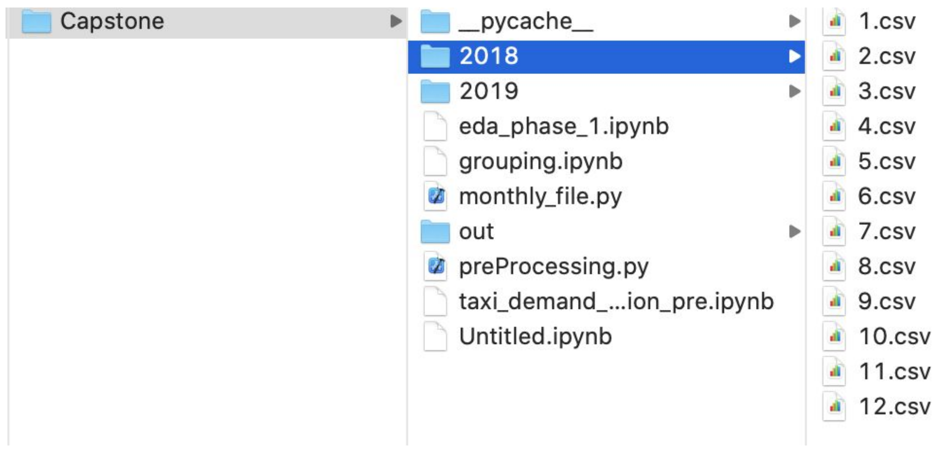
1. preProcessing.py (Month Wise data preprocessing and merging features like weather data)
2. Eda_n_grouping.ipynb (Year data preprocessing and grouping)
3. Ml_model.ipynb (ML Model and visualisation)

Step-by-step Walk Through of the Solution

Data Preparation

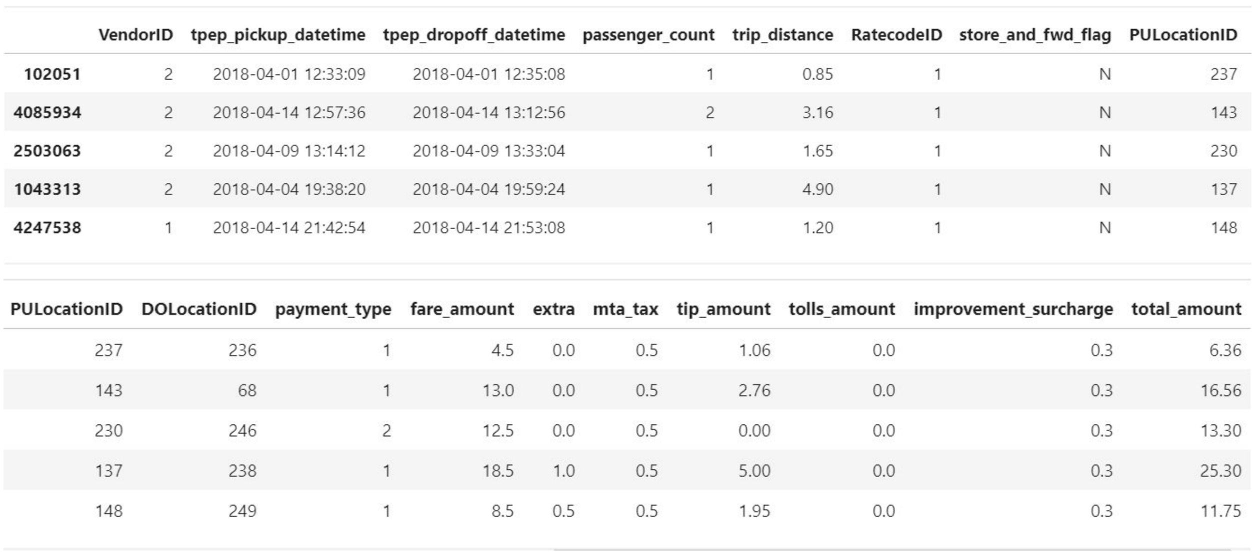
Downloading data files

Data is downloaded from TLC Trip Record Data - TLC year wise. Each month's data is in the form of separate csvs; rename each month file to it's number.



Sample Data

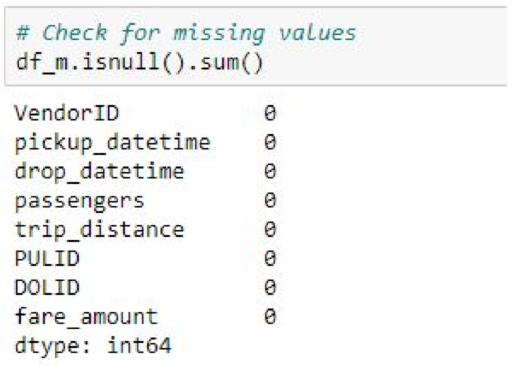
The data obtained from the TLC website consists of the following columns which can be seen in the figure below. The shape of the data frame is for example, (9305515 x 17) for the month of April 2018. Of the 17 columns in the dataset, the variables of interest for our current analysis are: *tpep_pickup_datetime, tpep_dropoff_datetime, trip_distance, PULocationID, DOLocationID, fare_amount*.



	VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count	trip_distance	RatecodeID	store_and_fwd_flag	PULocationID	DOLocationID	payment_type	fare_amount	extra	mta_tax	tip_amount	tolls_amount	improvement_surcharge	total_amount
102051	2	2018-04-01 12:33:09	2018-04-01 12:35:08	1	0.85	1	N	237	236	1	4.5	0.0	0.5	1.06	0.0	0.3	6.36
4085934	2	2018-04-14 12:57:36	2018-04-14 13:12:56	2	3.16	1	N	143	68	1	13.0	0.0	0.5	2.76	0.0	0.3	16.56
2503063	2	2018-04-09 13:14:12	2018-04-09 13:33:04	1	1.65	1	N	230	246	2	12.5	0.0	0.5	0.00	0.0	0.3	13.30
1043313	2	2018-04-04 19:38:20	2018-04-04 19:59:24	1	4.90	1	N	137	238	1	18.5	1.0	0.5	5.00	0.0	0.3	25.30
4247538	1	2018-04-14 21:42:54	2018-04-14 21:53:08	1	1.20	1	N	148	249	1	8.5	0.5	0.5	1.95	0.0	0.3	11.75

Missing Value Analysis

No missing values were found in the current dataset. Since the dataset consists of trip by trip entry of values, it is likely that the dataset will not consist of missing values as far as the features of our interest are considered.

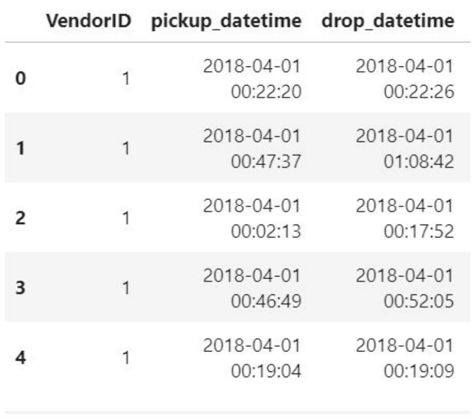


```
# Check for missing values
df_m.isnull().sum()
```

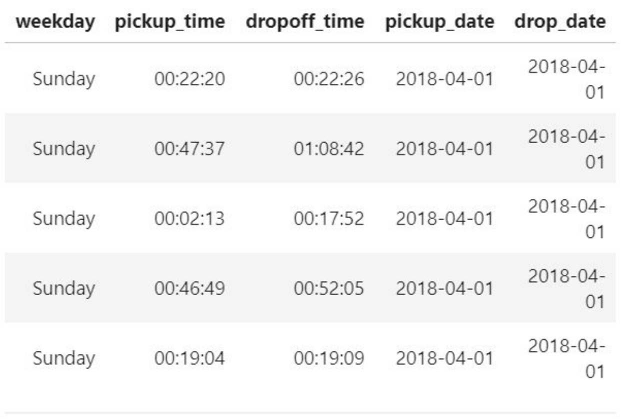
```
VendorID           0
pickup_datetime    0
drop_datetime      0
passengers         0
trip_distance      0
PULID              0
DOLID              0
fare_amount        0
dtype: int64
```

Transformation of date-time format

When we read the data from a csv file and convert it into a data frame, the type of date columns is of type ‘Object’, so first we need to convert them into *date time* format and then we obtain required columns from it, like weekday, time etc.



	VendorID	pickup_datetime	drop_datetime
0	1	2018-04-01 00:22:20	2018-04-01 00:22:26
1	1	2018-04-01 00:47:37	2018-04-01 01:08:42
2	1	2018-04-01 00:02:13	2018-04-01 00:17:52
3	1	2018-04-01 00:46:49	2018-04-01 00:52:05
4	1	2018-04-01 00:19:04	2018-04-01 00:19:09



weekday	pickup_time	dropoff_time	pickup_date	drop_date
Sunday	00:22:20	00:22:26	2018-04-01	2018-04-01
Sunday	00:47:37	01:08:42	2018-04-01	2018-04-01
Sunday	00:02:13	00:17:52	2018-04-01	2018-04-01
Sunday	00:46:49	00:52:05	2018-04-01	2018-04-01
Sunday	00:19:04	00:19:09	2018-04-01	2018-04-01

Time Binning

The time of pickup and drop-off given as date-time format in original data was split and transformed into pickup and drop-off date, time of day and day of week. The exact time values were further categorized into bins of 30 minutes i.e. there are 48 bins in an entire day in which the value can fall. This was done because of our motive to predict demand in a 30-minute interval.

Time Code Generation

Following function has been used for generating time code based on time. Refer Preprocessor class, It is a static function.

```
def get_code_for_time(date_time: datetime) -> int:
     hour = date_time.hour
     code = 1
     if hour != 0:
        code = (hour * 2) + 1
     if date_time.minute >= 30:
        code = code + 1
     return code
```

Generating Start Day of Month

Following function is used to get the first day of the month. Refer to the PreProcessor class, it is a static function.

```
def first_day_of_month(month: int, year: int) -> datetime:
        return datetime.datetime(year, month, 1)
```

Generating Last Day of Month

Following function is used to get the last day of the month. Refer Preprocessor class, It is a static function.

```
def last_day_of_month(any_day: datetime) -> datetime:
     # this will never fail
     # get close to the end of the month for any day, and add 4 days 'over'
     next_month = any_day.replace(day=28) + datetime.timedelta(days=4)
     # subtract the number of remaining 'overage' days to get last day of current
               month,
     # or said programattically said, the previous day of the first of next month
     return next_month - datetime.timedelta(days=next_month.day)
```

Weather data generation

Refer class WeatherData, to generate weather data.

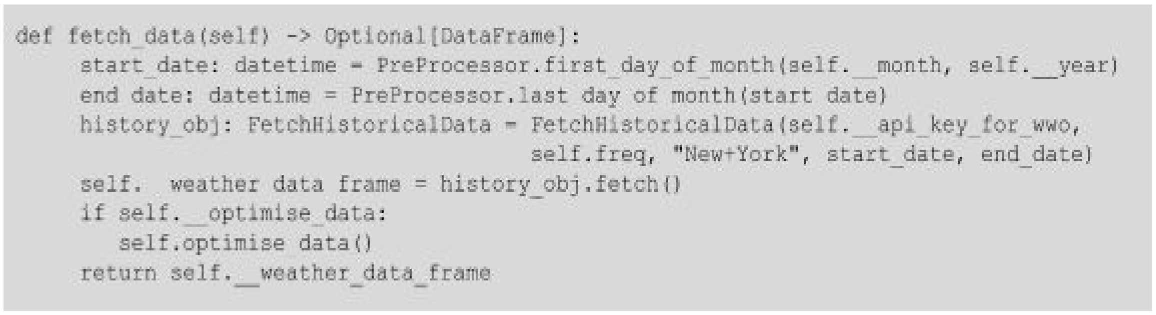
```
    def init(self, output_path: str, month: int, year: int,
                 api_key: str, optimise_data: bool = True, freq: int = 1):
        super().__init__("", output_path)
        self.__month: int = month
        self.__year: int = year
        self.__freq: int = freq
        self.__api_key_for_wwo = api_key
        self.__optimise_data: bool = optimise_data
```

Input required (Weather Data Constructor) to this class is

1. Output path: Path where output files will be generated
2. Month: month index
3. Year: year number

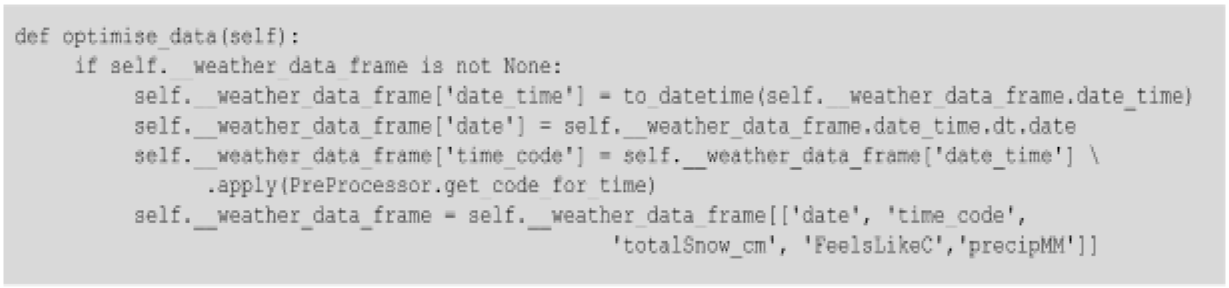
4. Frequency: Hourly, Quarterly … daily. For hourly, the number is 1.
5. Api_key: api key generated from site for historical data (valid for a year)
6. Optimise_data: is a boolean value, if this value is set then will get only following columns
    a. date
    b. time_code
    c. totalSnow_cm
    d. FeelsLikeC
    e. precipMM

To fetch data use WeatherData.fetch_data(), It returns weather data Dataframe as output.
Also Refer class FetchHistoricalData, It calls the API retrieve_hist_data internally.
WeatherData.fetch_data() method internally calls FetchHistoricalData.



```
def fetch_data(self) -> Optional[DataFrame]:
     start_date: datetime = PreProcessor.first_day_of_month(self.__month, self.__year)
     end_date: datetime = PreProcessor.last_day_of_month(start_date)
     history_obj: FetchHistoricalData = FetchHistoricalData(self.__api_key_for_wwo,
                                    self.freq, "New+York", start_date, end_date)
     self.__weather_data_frame = history_obj.fetch()
     if self.__optimise_data:
        self.optimise_data()
     return self.__weather_data_frame
```

Here optimise_data is true, as in we only required few columns, let’s take look at optimise_data()



```
def optimise_data(self):
     if self.__weather_data_frame is not None:
          self.__weather_data_frame['date_time'] = to_datetime(self.__weather_data_frame.date_time)
          self.__weather_data_frame['date'] = self.__weather_data_frame.date_time.dt.date
          self.__weather_data_frame['time_code'] = self.__weather_data_frame['date_time'] \
                .apply(PreProcessor.get_code_for_time)
          self.__weather_data_frame = self.__weather_data_frame[['date', 'time_code',
                                               'totalSnow_cm', 'FeelsLikeC','precipMM']]
```

Optimised data function keeps required data, also generates time code columns.
WeatherData.fetch_data() generates dataframe, to save that dataframe to output path use WeatherData.write_as_csv()

```
def write_as_csv(self):
    if self.__weather_data_frame is not None:
        self.write_dataframe_as_csv(self.__weather_data_frame,
                                    "history_" + str(self.__year)

                                  "_" + str(self.__month))
```

```
taxi_data = YellowTaxiData(monthly_file_path, yellow_car_data_out_path,
                           month, year, whether_data)
taxi_data.read_and_process_csv()
```

Data Cleaning

The data cleaning procedure i.e removal of outliers and the exploratory data analysis were done on files obtained from the above process. For this purpose, we used the dask library in python due to the size of the dataset which was much bigger than the memory available in personal computers.

Removal of outliers based on trip distance

Distribution of trip distance is also heavily skewed towards the right suggesting there are few rides with very long distance. The box plot below clearly shows the presence of outliers.

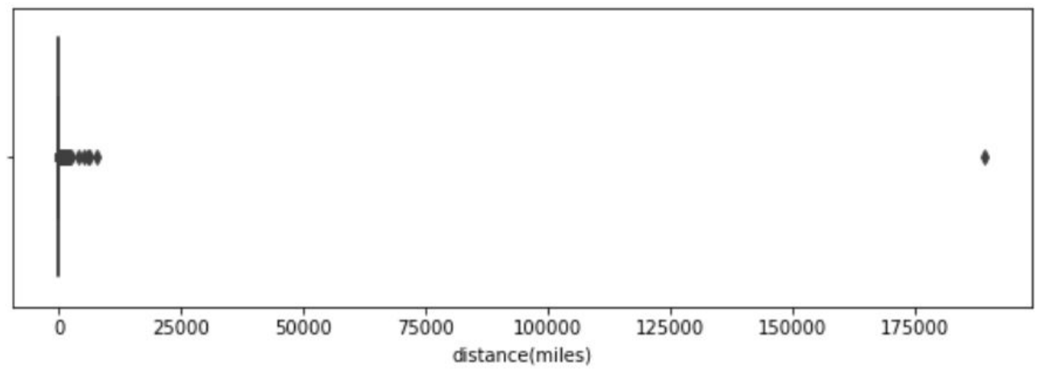


Fig: Box plot of trip distance before removal of outliers

We remove the outliers using the percentile methods. The 25th, 50th and 75th percentiles are not clearly visible in the box plot. Hence the outliers lie far ahead of the IQR range. On looking further than the 99.995th percentile there was a sudden jump in the value of trip distance, hence we removed the corresponding data greater than 60 miles.

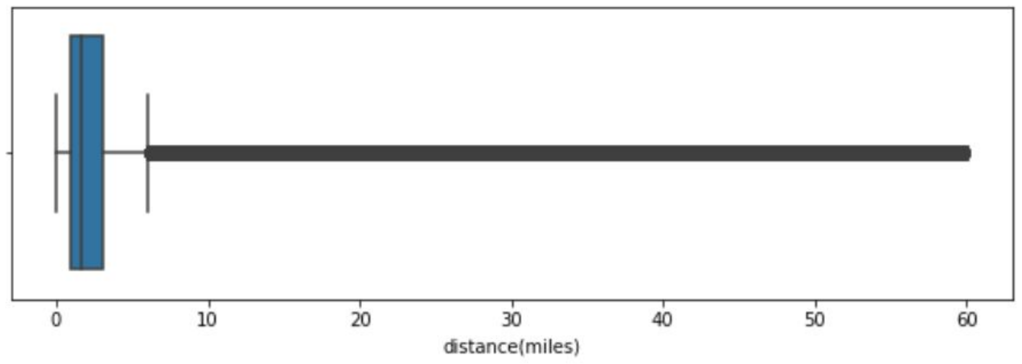


Fig: Box plot of trip distance after removal of outliers

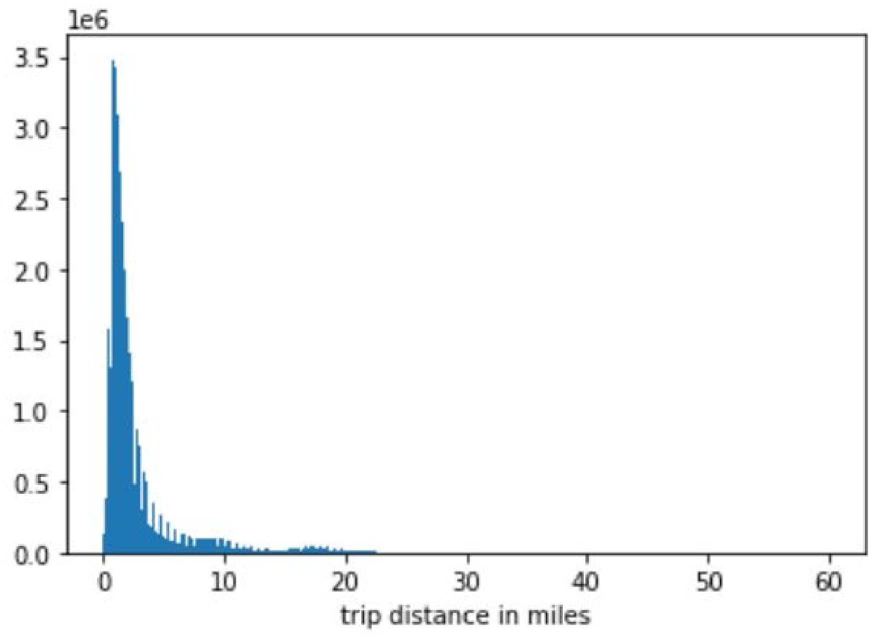


Fig: Distribution of trip distance after removal of outliers

Removal of outliers based on trip duration:

According to NYC Taxi & Limousine Commission Regulations the maximum allowed trip duration in a 24-hour interval is 12 hours or 720 minutes. It is calculated as a difference between drop-off time and pickup time. The boxplot below shows trip durations for the month. As we can see there are several outliers and the distribution is extremely skewed.

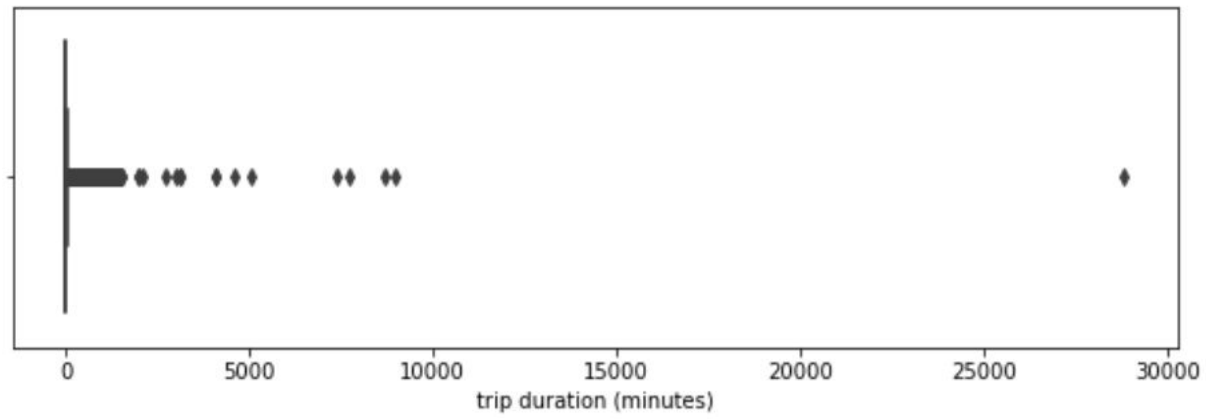


Fig: Box plot of trip duration before removal of outliers

To remove outlier data, we have used the percentile method as well as the TLC criterion. Data corresponding to trip duration beyond 99.995th percentile was removed i.e. trip duration of more than 720 minutes was removed

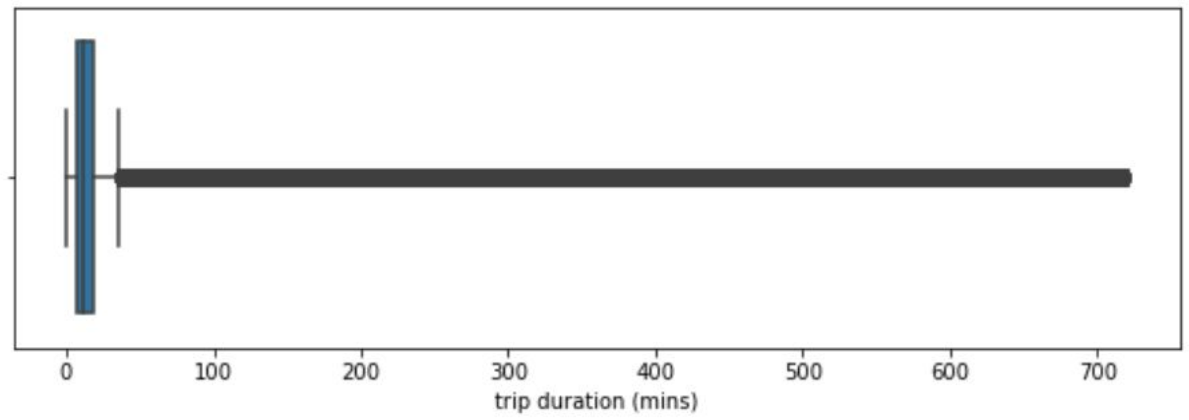


Fig: Box plot of trip duration after removal of outliers

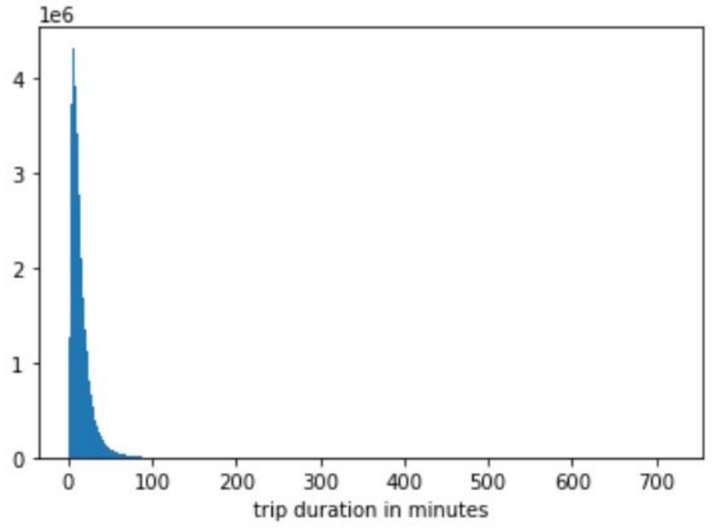


Fig: Distribution of trip duration before removal of outliers

The log transformed value of time duration shows more clearly bell shape and a short tail on the left of the distribution.

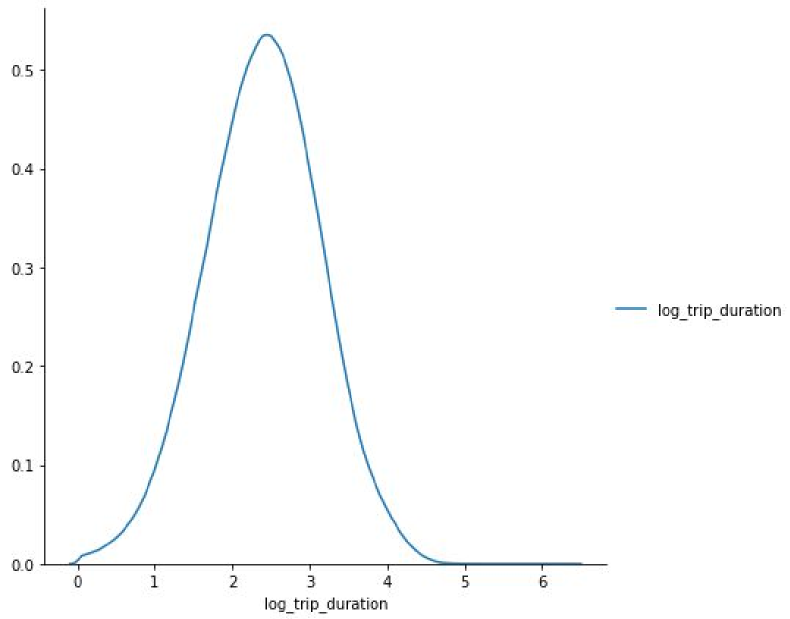


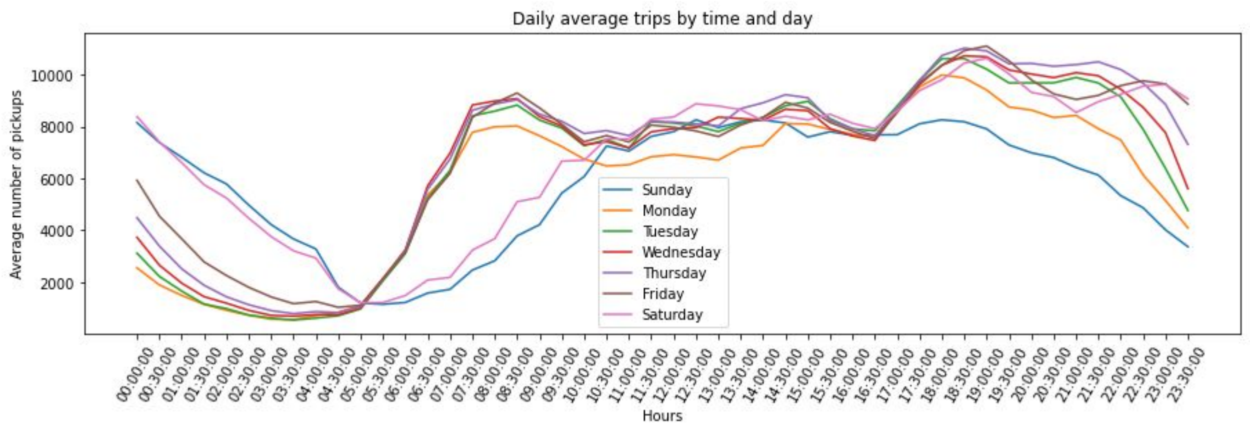
Fig: Log normal distribution of trip duration after removal of outliers

EDA and Visualizations

Daily average trips by time and day

To observe trends of how the taxi demand varies across different weekdays and weekends w.r.t time of the day we plotted the daily average trips by time and day for the sample month of April 2018. For this we arranged the data frame into a dictionary with weekdays as keys and averaged the taxi demand in all 48 time slots of the day for each weekday.

As can be seen from the figure below there are observable differences in demand between regular weekdays and weekends. Taxi demand appears to increase late in the morning on weekends and is higher at around midnight compared to other weekdays.



Distribution of demand at pickup locations

The bar plot below shows the taxi demand at each location during the entire year. It can be observed that some locations are clustered together and have higher net demand for taxis than the rest of the locations. Our target is to correctly identify these locations throughout the city.

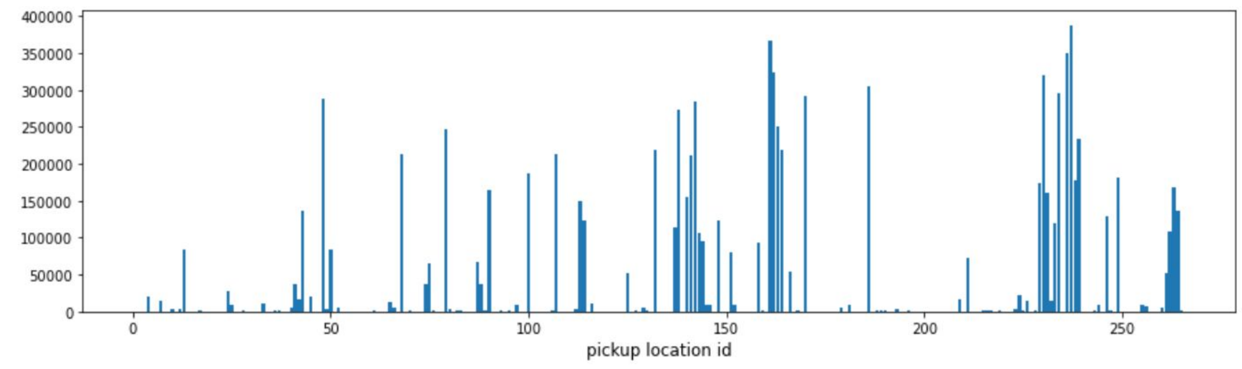
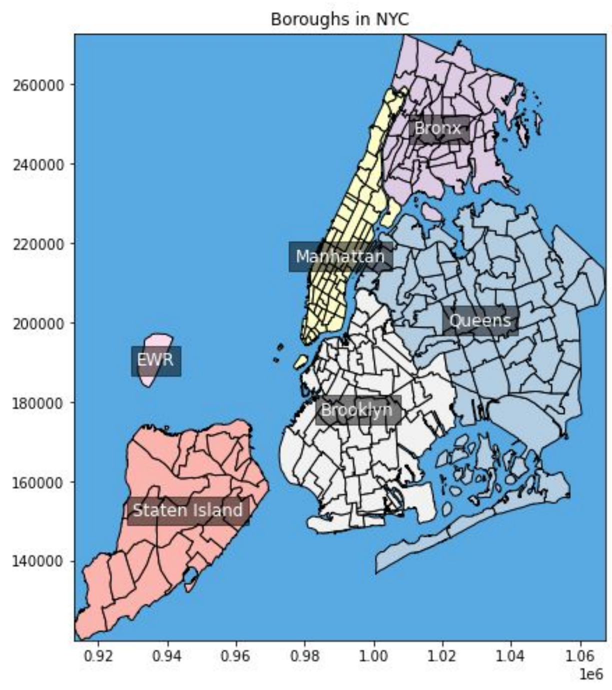
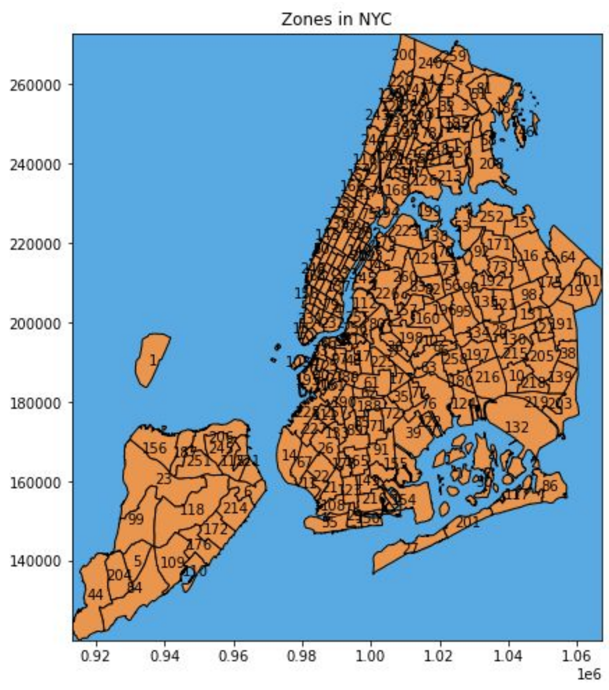


Fig: Number of pickups from location ids

Mapping of location id onto the map of New York

For this we used the shapefile from https://geo.nyu.edu/catalog/nyu_2451_36743 to transform the location ids to geographical coordinates, and overlay them onto the map of New York city. This would be useful in clustering the regions later on in the analysis and also be helpful in drawing insights from the analysis about peak hour traffic at certain important locations in the city.
The first image shows in the city divided into boroughs. The second image shows the location ids in these boroughs. Later on we can cluster the small locations to make more accurate predictions.





Cyclic nature of taxi demand

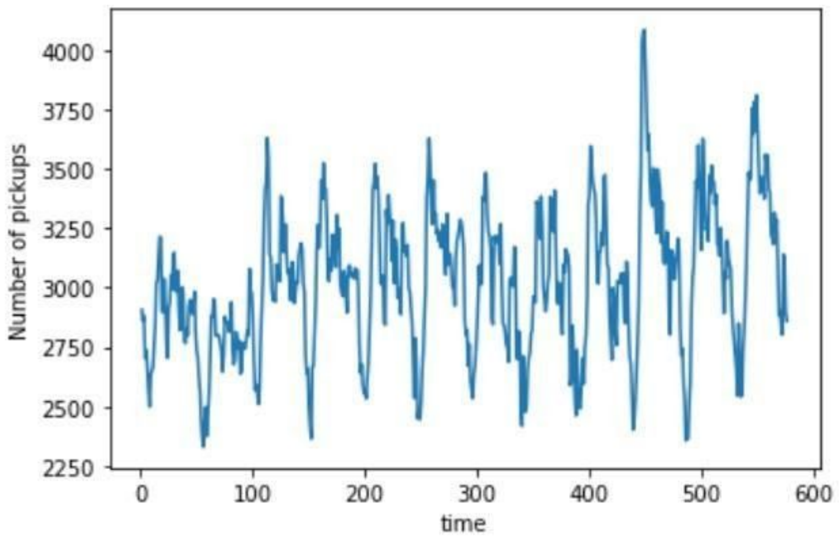


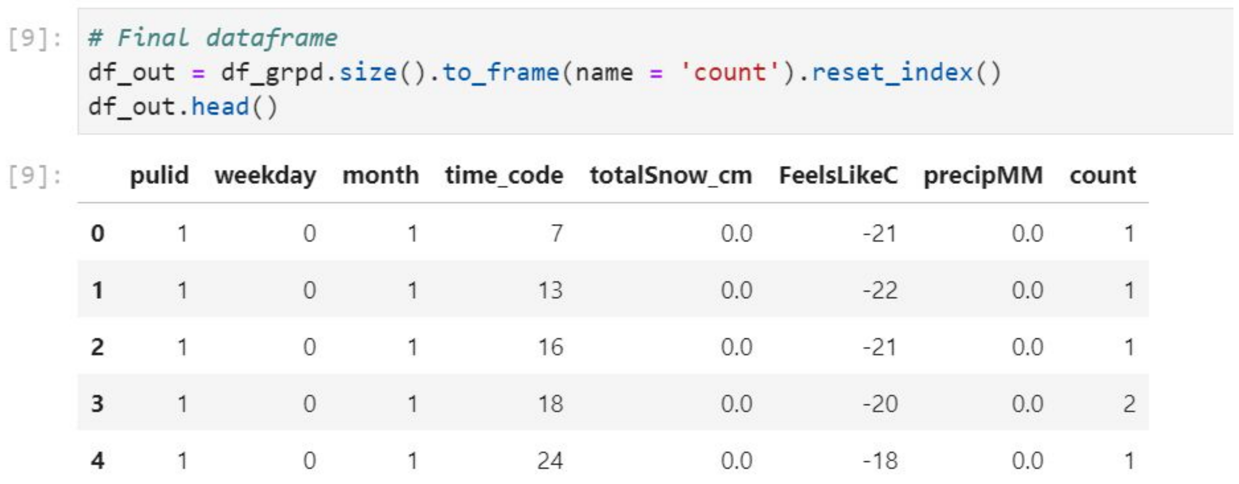
Fig: Taxi demand over the entire year of 2018

It is observed that a pattern is formed if the demand for taxis is plotted for every month of the year. We can see 12 similar patterns with distinct peaks in the above graph for every month. This suggests that our time series data for taxi demand consists of a pattern. The demand for taxis also has a repetition pattern for every 24 hours as can be seen from the figure in section Daily average trips by time and day.

Preparing data for applying Machine Learning algorithms:

After cleaning the data for outliers we used the *group by* operation on the data frame to get the *count* of pickups at each particular time slot, location, day of the week and month. Thus we get the number of pickup done by taxis as a time series data which can be formulated as a regression problem. In the process we also end up reducing the size of data to around 45 megabytes from a massive 5 gigabytes.

Following is the snapshot of the final data frame fed to the models. The dataset has features - pickup location (PULID), day of the week (weekday) and pickup time slot. The target variable for the model is how many pickups were done (count), given the aforementioned features.



```
[9]: # Final dataframe
     df_out = df_grpd.size().to_frame(name = 'count').reset_index()
     df_out.head()
```

[9]:

	pulid	weekday	month	time_code	totalSnow_cm	FeelsLikeC	precipMM	count
0	1	0	1	7	0.0	-21	0.0	1
1	1	0	1	13	0.0	-22	0.0	1
2	1	0	1	16	0.0	-21	0.0	1
3	1	0	1	18	0.0	-20	0.0	2
4	1	0	1	24	0.0	-18	0.0	1

ML models for taxi demand prediction

Training & Test Strategy for models

Training data sets

2018 taxi Data will be partitioned into a ratio of 80-20. And a bigger chunk will be used as training data to model.

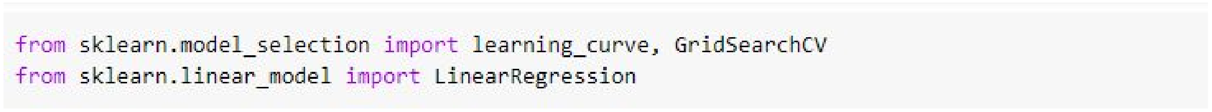
Test data sets

Smaller chunk of the 2018 dataset will be used as testing data.

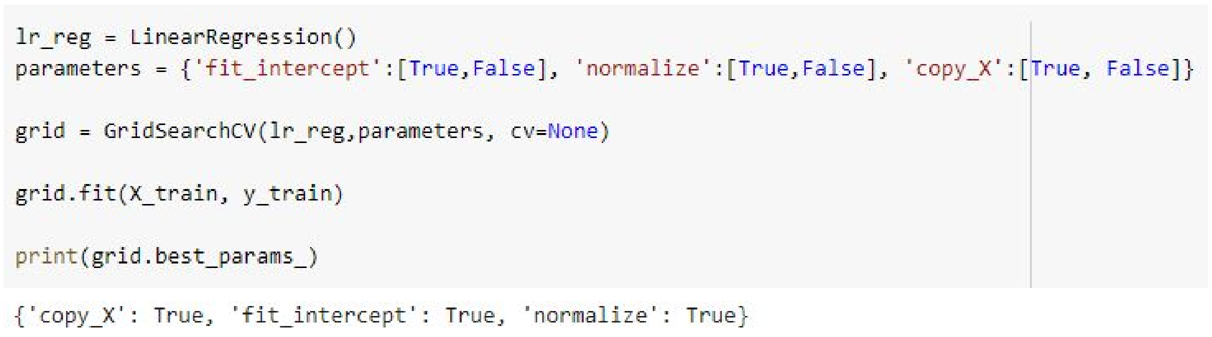
2019 data set will be also used as Test Data

Linear Regression

- Derive Best Estimator parameters



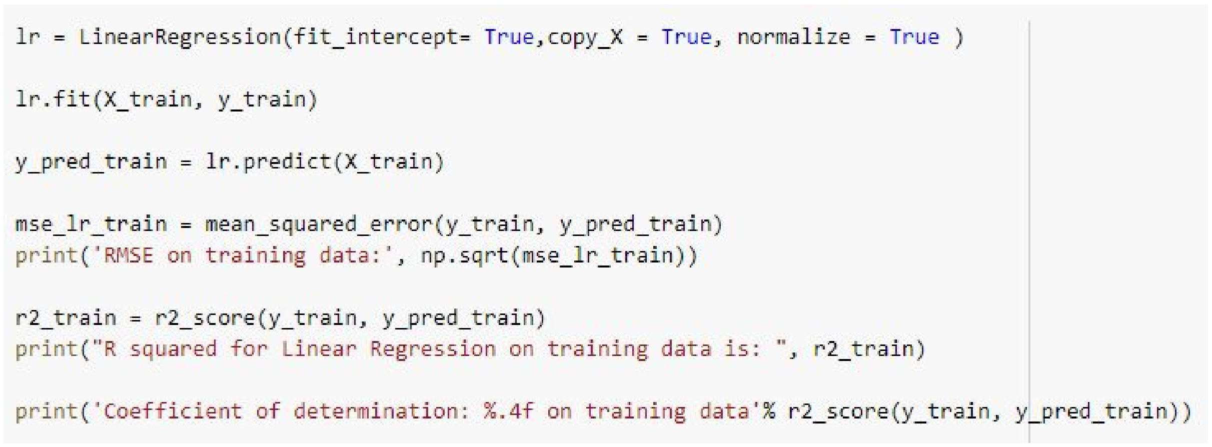
```
from sklearn.model_selection import learning_curve, GridSearchCV
from sklearn.linear_model import LinearRegression
```



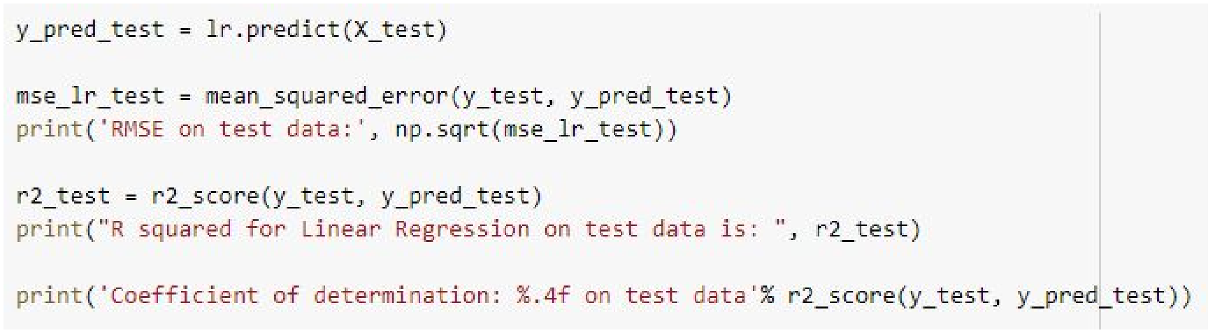
```
lr_reg = LinearRegression()
parameters = {'fit_intercept':[True,False], 'normalize':[True,False], 'copy_X':[True, False]}

grid = GridSearchCV(lr_reg,parameters, cv=None)

grid.fit(X_train, y_train)

print(grid.best_params_)
```

```
{'copy_X': True, 'fit_intercept': True, 'normalize': True}
```

- Model Execution & Accuracy Capture



```
lr = LinearRegression(fit_intercept= True,copy_X = True, normalize = True )

lr.fit(X_train, y_train)

y_pred_train = lr.predict(X_train)

mse_lr_train = mean_squared_error(y_train, y_pred_train)
print('RMSE on training data:', np.sqrt(mse_lr_train))

r2_train = r2_score(y_train, y_pred_train)
print("R squared for Linear Regression on training data is: ", r2_train)

print('Coefficient of determination: %.4f on training data'% r2_score(y_train, y_pred_train))
```



```
y_pred_test = lr.predict(X_test)

mse_lr_test = mean_squared_error(y_test, y_pred_test)
print('RMSE on test data:', np.sqrt(mse_lr_test))

r2_test = r2_score(y_test, y_pred_test)
print("R squared for Linear Regression on test data is: ", r2_test)

print('Coefficient of determination: %.4f on test data'% r2_score(y_test, y_pred_test))
```

Random Forest regression

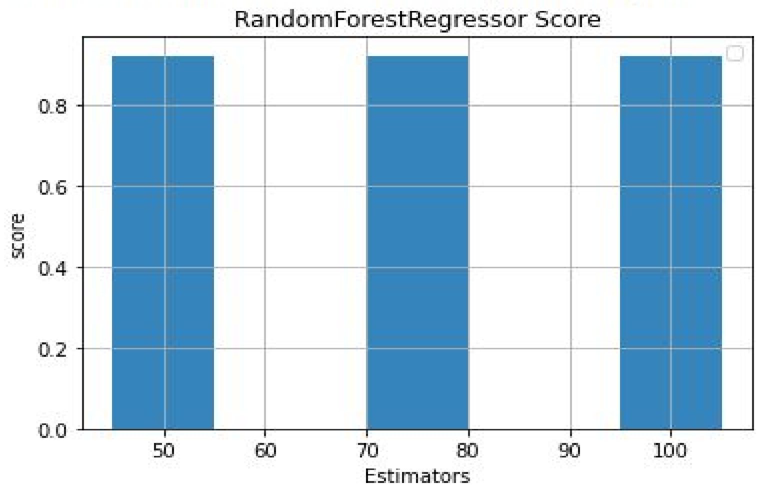
- Derive Best Estimator parameters

```
from sklearn.model_selection import RandomizedSearchCV
from sklearn.ensemble import RandomForestRegressor
import random
```

```
# Establish model
model = RandomForestRegressor(n_jobs=-1)

# Try different numbers of n_estimators - this will take a minute or so
estimators = [50, 75, 100]
scores = []
scoresOfEstimation = []

for n in estimators:
    model.set_params(n_estimators=n)
    model.fit(X_train, y_train)
    scores.append(model.score(X_test, y_test))
    scoresOfEstimation.append(n)
```

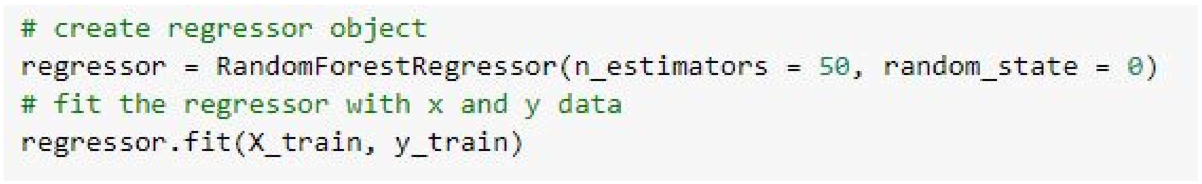
```
plt.bar(scoresOfEstimation,scores,10,alpha=.9)
plt.ylabel('score')
plt.xlabel('Estimators')
plt.title('RandomForestRegressor Score')
plt.grid(True)
plt.legend()
plt.show()
```

```
No handles with labels found to put in legend.
```

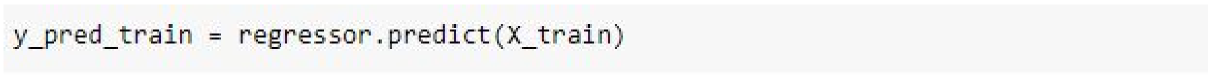


As we can see, very minor improvement with an increased number of estimators. So building a model with 50 estimators only.

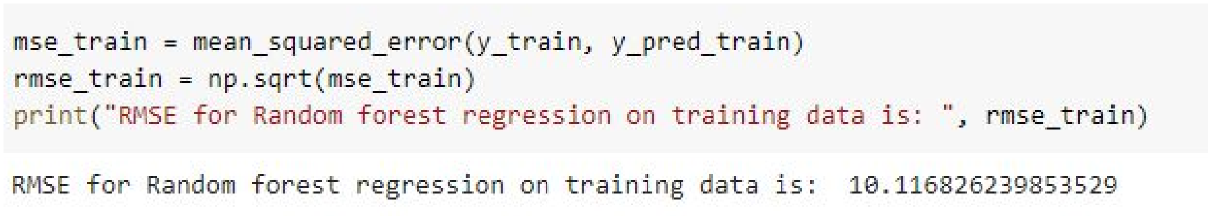
- Model Execution & Accuracy Capture



```
# create regressor object
regressor = RandomForestRegressor(n_estimators = 50, random_state = 0)
# fit the regressor with x and y data
regressor.fit(X_train, y_train)
```

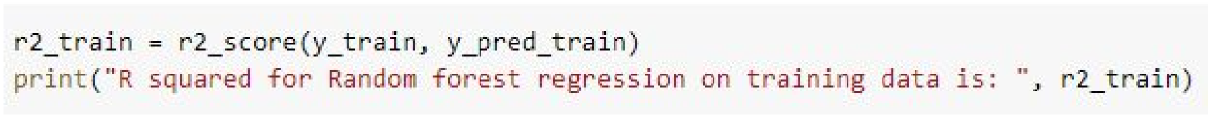


```
y_pred_train = regressor.predict(X_train)
```



```
mse_train = mean_squared_error(y_train, y_pred_train)
rmse_train = np.sqrt(mse_train)
print("RMSE for Random forest regression on training data is: ", rmse_train)
```

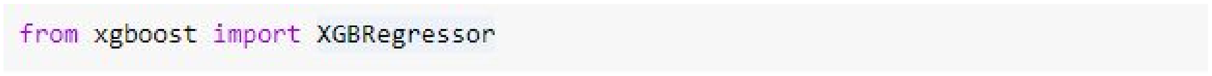
```
RMSE for Random forest regression on training data is:  10.116826239853529
```



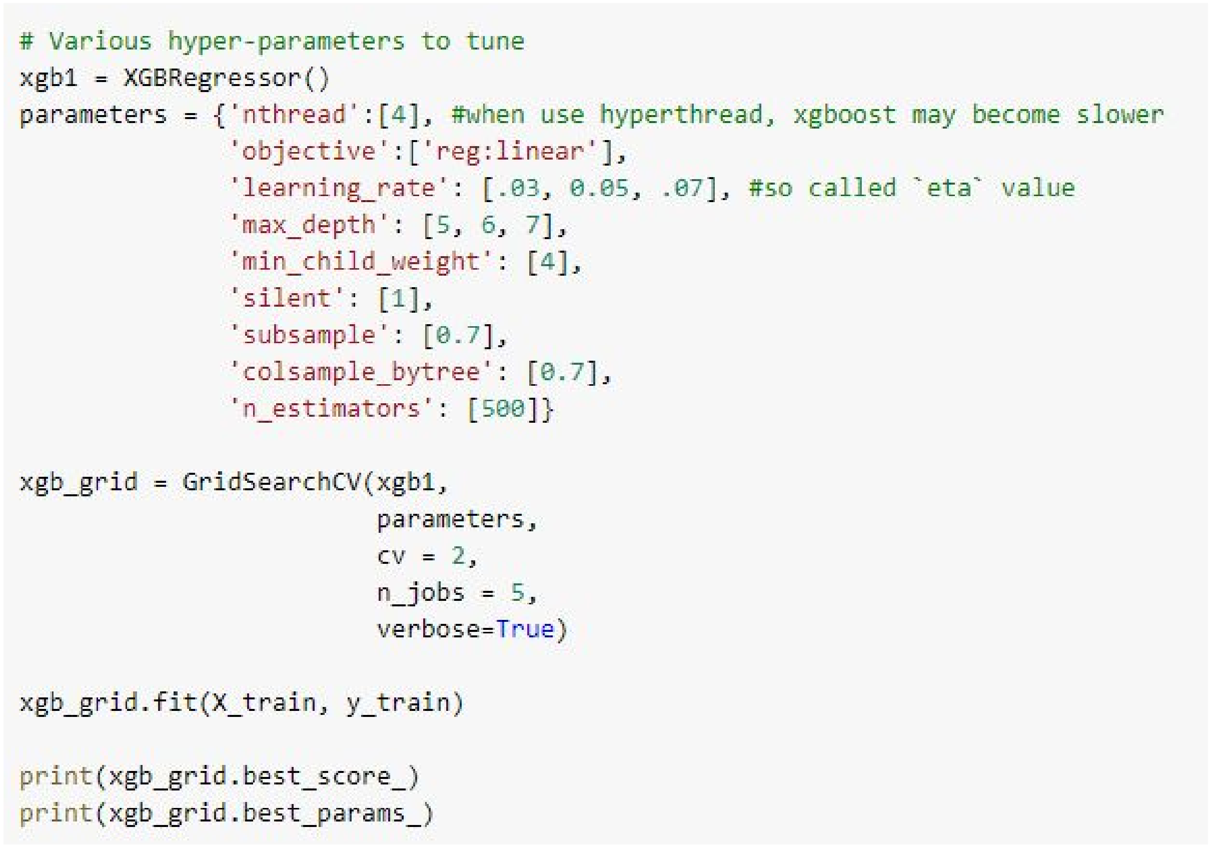
```
r2_train = r2_score(y_train, y_pred_train)
print("R squared for Random forest regression on training data is: ", r2_train)
```

XGBoost Regression

- Derive Best Estimator parameters

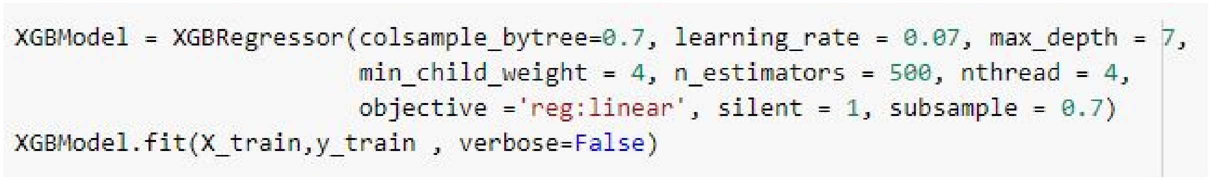


```
from xgboost import XGBRegressor
```

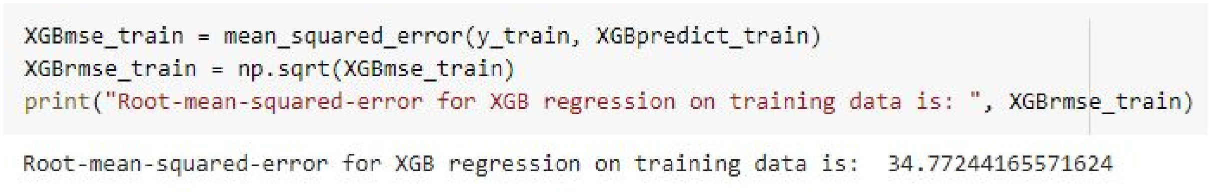


```
# Various hyper-parameters to tune
xgb1 = XGBRegressor()
parameters = {'nthread':[4], #when use hyperthread, xgboost may become slower
              'objective':['reg:linear'],
              'learning_rate': [.03, 0.05, .07], #so called `eta` value
              'max_depth': [5, 6, 7],
              'min_child_weight': [4],
              'silent': [1],
              'subsample': [0.7],
              'colsample_bytree': [0.7],
              'n_estimators': [500]}

xgb_grid = GridSearchCV(xgb1,
                        parameters,
                        cv = 2,
                        n_jobs = 5,
                        verbose=True)

xgb_grid.fit(X_train, y_train)

print(xgb_grid.best_score_)
print(xgb_grid.best_params_)
```

- Model Execution & Accuracy Capture

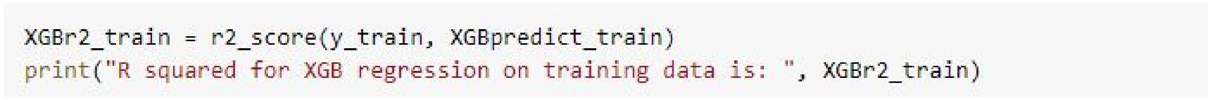


```
XGBModel = XGBRegressor(colsample_bytree=0.7, learning_rate = 0.07, max_depth = 7,
                        min_child_weight = 4, n_estimators = 500, nthread = 4,
                        objective ='reg:linear', silent = 1, subsample = 0.7)
XGBModel.fit(X_train,y_train , verbose=False)
```



```
XGBmse_train = mean_squared_error(y_train, XGBpredict_train)
XGBrmse_train = np.sqrt(XGBmse_train)
print("Root-mean-squared-error for XGB regression on training data is: ", XGBrmse_train)
```

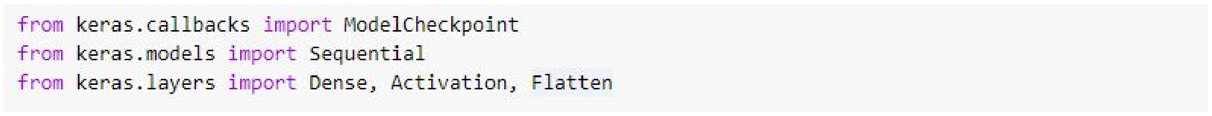
```
Root-mean-squared-error for XGB regression on training data is:  34.77244165571624
```



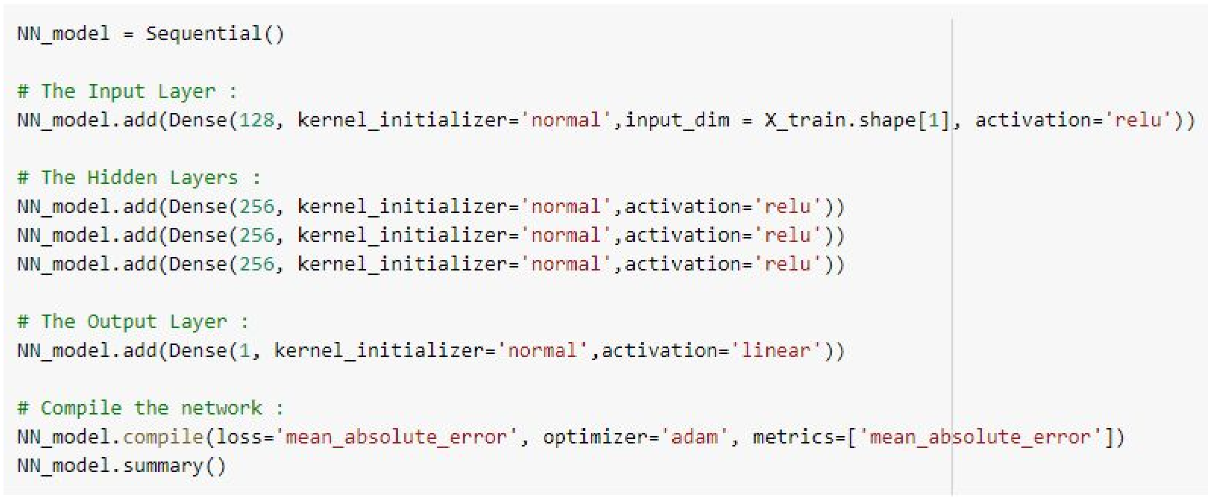
```
XGBr2_train = r2_score(y_train, XGBpredict_train)
print("R squared for XGB regression on training data is: ", XGBr2_train)
```

Neural network regression

- Derive Best Estimator parameters

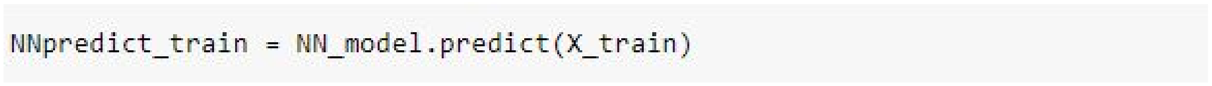


```
from keras.callbacks import ModelCheckpoint
from keras.models import Sequential
from keras.layers import Dense, Activation, Flatten
```

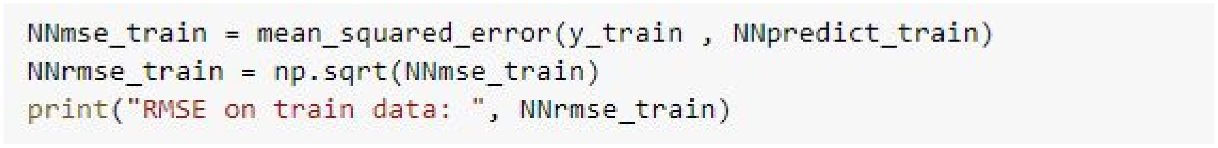


```
NN_model = Sequential()

# The Input Layer :
NN_model.add(Dense(128, kernel_initializer='normal',input_dim = X_train.shape[1], activation='relu'))

# The Hidden Layers :
NN_model.add(Dense(256, kernel_initializer='normal',activation='relu'))
NN_model.add(Dense(256, kernel_initializer='normal',activation='relu'))
NN_model.add(Dense(256, kernel_initializer='normal',activation='relu'))

# The Output Layer :
NN_model.add(Dense(1, kernel_initializer='normal',activation='linear'))

# Compile the network :
NN_model.compile(loss='mean_absolute_error', optimizer='adam', metrics=['mean_absolute_error'])
NN_model.summary()
```

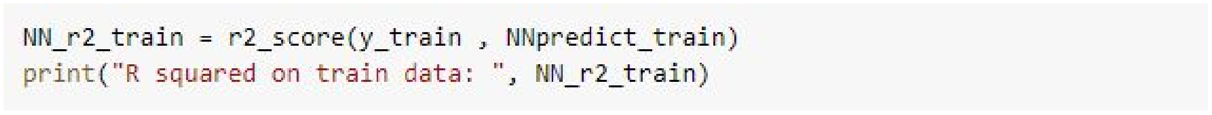
- Model Execution & Accuracy Capture



```
NNpredict_train = NN_model.predict(X_train)
```



```
NNmse_train = mean_squared_error(y_train , NNpredict_train)
NNrmse_train = np.sqrt(NNmse_train)
print("RMSE on train data: ", NNrmse_train)
```



```
NN_r2_train = r2_score(y_train , NNpredict_train)
print("R squared on train data: ", NN_r2_train)
```

Decision Tree regression

- Derive Best Estimator parameters

```
from sklearn.tree import DecisionTreeRegressor
```

```
param_grid = {"criterion": ["mse", "mae"],
              "max_depth": [3, 9, 13],
              "min_samples_leaf": [10, 30, 50],
              "max_leaf_nodes": [10, 20, 30],
              }

grid_cv_dtm = GridSearchCV(dtm, param_grid, cv=5)

grid_cv_dtm.fit(X,y)

print("R-Squared::{}".format(grid_cv_dtm.best_score_))
print("Best Hyperparameters::\n{}".format(grid_cv_dtm.best_params_))
```

- Model Execution & Accuracy Capture

```
regressor = DecisionTreeRegressor(max_depth=13, random_state=1, min_samples_leaf=10)
```

```
regressor.fit(X_train, y_train)
```

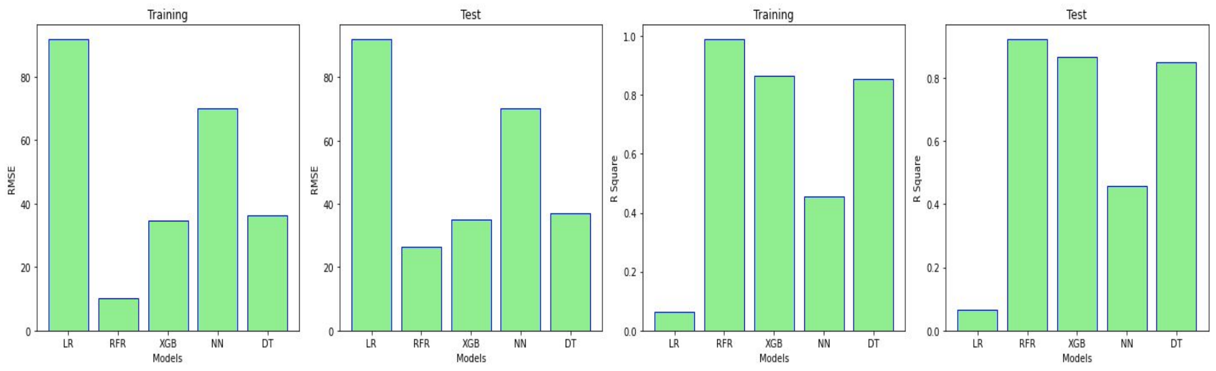
```
y_pred_train = regressor.predict(X_train)
```

```
MAE_train = mean_absolute_error(y_train , y_pred_train)
print('Mean absolute error for training data = ', MAE_train)

DTmse_train = mean_squared_error(y_train, y_pred_train)
DTrmse_train = np.sqrt(DTmse_train)
print("RMSE for training data is: ", DTrmse_train)

DTr2_train = r2_score(y_train, y_pred_train)
print("R squared on training data is: ", DTr2_train)
```

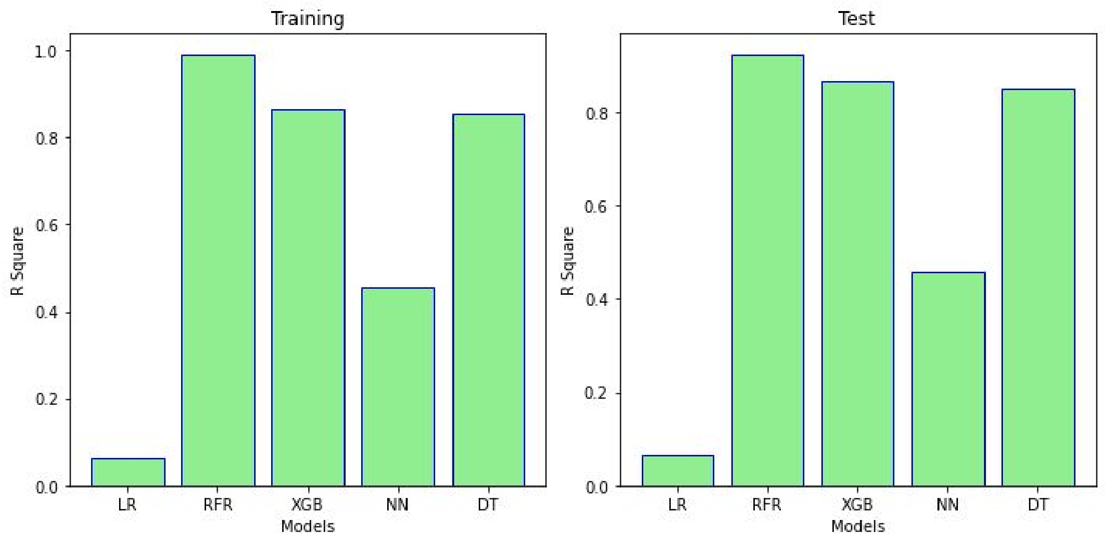
Comparison of models

Model	RMSE			R2			
	Training	Test	2019		Training	Test	2019
Linear Regression	91.8034	91.97	91.97		0.0644	0.065	0.065
Random forest	10.1168	26.568	39.874		0.9886	0.922	0.7512
XGBoost	34.7724	34.973	36.037		0.8657	0.8648	0.7968
Neural network	70.0314	70.125	59.538		0.4555	0.4564	0.4454
Decision Tree	36.4154	36.93	37.327		0.8527	0.8492	0.782
Day 3							
Linear Regression	97.8433	97.729	97.729		0.0774	0.077	0.077
Random forest	10.003	26.084	45.378		0.9903	0.9342	0.7174
XGBoost	29.8889	30.846	38.938		0.9139	0.908	0.7919
Neural network	85.1625	85.539	70.683		0.301	0.2929	0.3145
Decision Tree	33.6219	35.456	40.687		0.891	0.8785	0.7728
Day 6							
Linear Regression	79.7058	79.27	79.27		0.0294	0.0325	0.0325
Random forest	8.3955	22.064	37.648		0.9892	0.925	0.6986
XGBoost	25.0753	25.874	32.619		0.9039	0.8969	0.7737
Neural network	71.7372	70.531	59.911		0.2138	0.234	0.2368
Decision Tree	28.3486	29.772	34.21		0.8772	0.8635	0.7512

Performance on 2018 Data

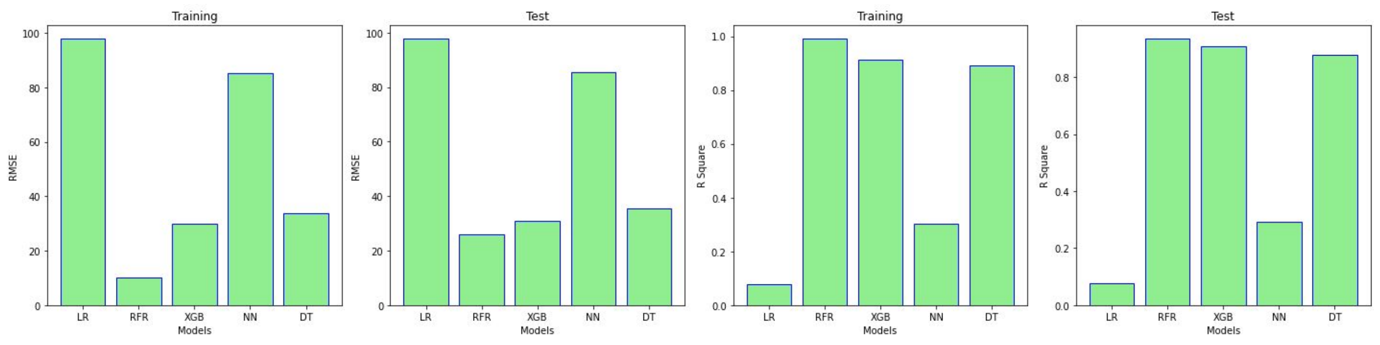

Training
Test
Training
Test
RMSE
R Square
Models
LR
RFR
XGB
NN
DT

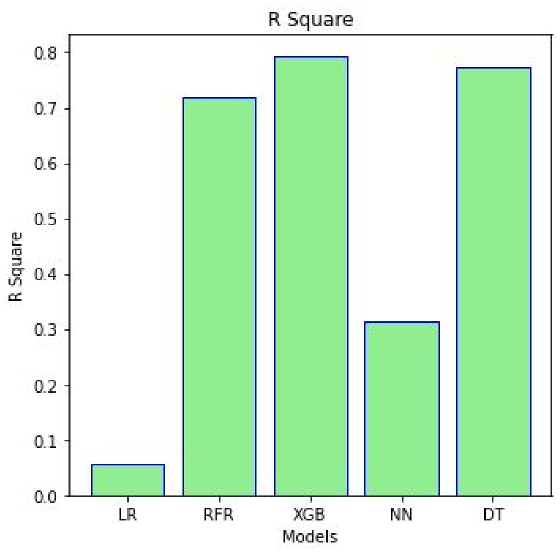
Performance on 2019 Data

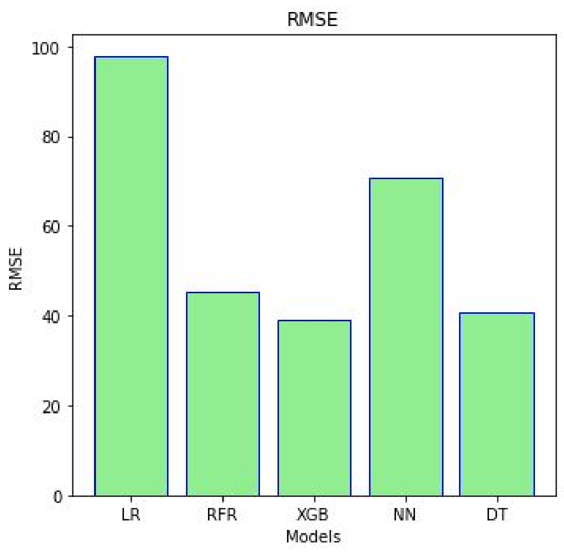


Days Wise Models – 2018

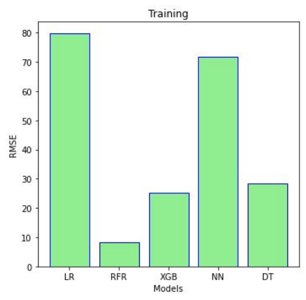
- Day 3

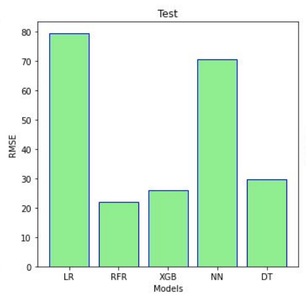


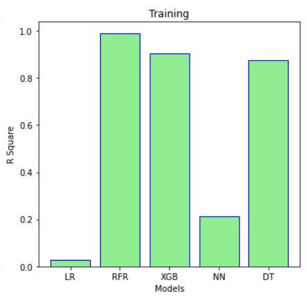

R Square
R Square
Models
LR
RFR
XGB
NN
DT

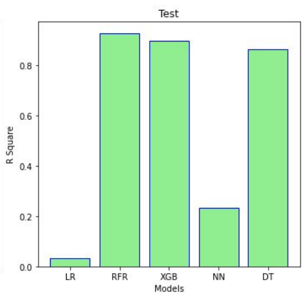

RMSE
RMSE
Models
LR
RFR
XGB
NN
DT

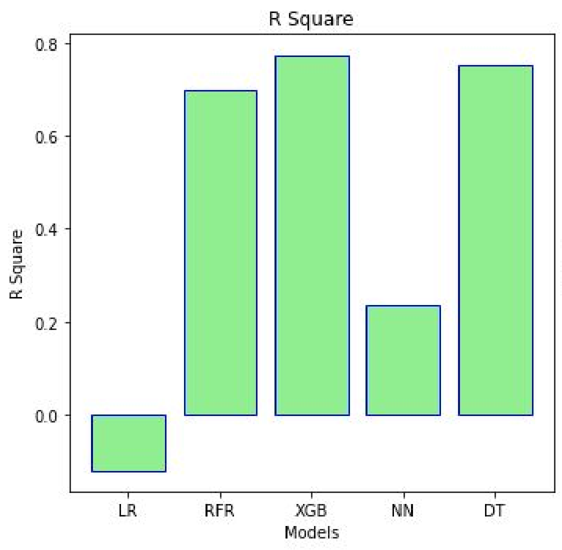
- Day 6

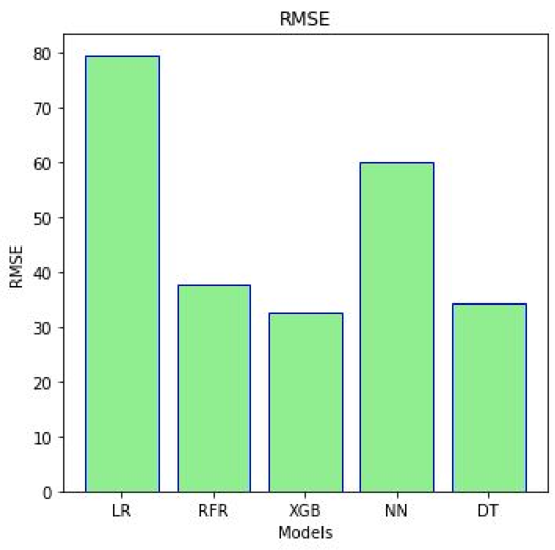

Training
RMSE
Models
LR
RFR
XGB
NN
DT


Test
RMSE
Models
LR
RFR
XGB
NN
DT


Training
R Square
Models
LR
RFR
XGB
NN
DT


Test
R Square
Models
LR
RFR
XGB
NN
DT


R Square
R Square
Models
LR
RFR
XGB
NN
DT


RMSE
RMSE
Models
LR
RFR
XGB
NN
DT

Observation

- Linear Regression shows lowest performance when compared with other Models.
- **Random Forest Regression** seems to be the best Model by considering R2 & RMSE both.
- XgBoost Regressor & Neural Network models are showing RMSE of more than Random Forest which is not a number based on which a prediction model can be considered.

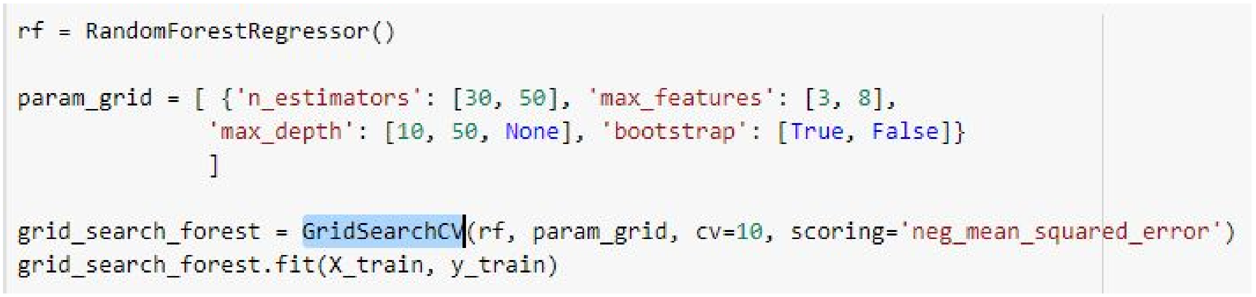
Final Model

Based on a number of metrics, Random Forest Regression looks to be a more prominent choice as a final model to proceed.

Hyper parameter Tuning

The process of searching for the ideal model architecture is referred to as *hyperparameter tuning*.

Grid search is arguably the most basic hyperparameter tuning method. With this technique, we simply build a model for each possible combination of all of the hyperparameter values provided, evaluating each model, and selecting the architecture which produces the best results.



```
rf = RandomForestRegressor()

param_grid = [ {'n_estimators': [30, 50], 'max_features': [3, 8],
              'max_depth': [10, 50, None], 'bootstrap': [True, False]}
              ]

grid_search_forest = GridSearchCV(rf, param_grid, cv=10, scoring='neg_mean_squared_error')
grid_search_forest.fit(X_train, y_train)
```

Output based on RMSE:

```
cvres = grid_search_forest.cv_results_
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    print(np.sqrt(-mean_score), params)
```

```
66.23339209879299 {'bootstrap': True, 'max_depth': 10, 'max_features': 3, 'n_estimators': 30}
67.18984602829268 {'bootstrap': True, 'max_depth': 10, 'max_features': 3, 'n_estimators': 50}
nan {'bootstrap': True, 'max_depth': 10, 'max_features': 8, 'n_estimators': 30}
nan {'bootstrap': True, 'max_depth': 10, 'max_features': 8, 'n_estimators': 50}
29.651223172207814 {'bootstrap': True, 'max_depth': 50, 'max_features': 3, 'n_estimators': 30}
29.29028243133367 {'bootstrap': True, 'max_depth': 50, 'max_features': 3, 'n_estimators': 50}
nan {'bootstrap': True, 'max_depth': 50, 'max_features': 8, 'n_estimators': 30}
nan {'bootstrap': True, 'max_depth': 50, 'max_features': 8, 'n_estimators': 50}
29.89130373236676 {'bootstrap': True, 'max_depth': None, 'max_features': 3, 'n_estimators': 30}
29.245262759908048 {'bootstrap': True, 'max_depth': None, 'max_features': 3, 'n_estimators': 50}
nan {'bootstrap': True, 'max_depth': None, 'max_features': 8, 'n_estimators': 30}
nan {'bootstrap': True, 'max_depth': None, 'max_features': 8, 'n_estimators': 50}
66.20419231594819 {'bootstrap': False, 'max_depth': 10, 'max_features': 3, 'n_estimators': 30}
66.72549821520995 {'bootstrap': False, 'max_depth': 10, 'max_features': 3, 'n_estimators': 50}
nan {'bootstrap': False, 'max_depth': 10, 'max_features': 8, 'n_estimators': 30}
nan {'bootstrap': False, 'max_depth': 10, 'max_features': 8, 'n_estimators': 50}
29.984080350035022 {'bootstrap': False, 'max_depth': 50, 'max_features': 3, 'n_estimators': 30}
29.386613999974177 {'bootstrap': False, 'max_depth': 50, 'max_features': 3, 'n_estimators': 50}
nan {'bootstrap': False, 'max_depth': 50, 'max_features': 8, 'n_estimators': 30}
nan {'bootstrap': False, 'max_depth': 50, 'max_features': 8, 'n_estimators': 50}
29.800953770912656 {'bootstrap': False, 'max_depth': None, 'max_features': 3, 'n_estimators': 30}
29.371094319906444 {'bootstrap': False, 'max_depth': None, 'max_features': 3, 'n_estimators': 50}
nan {'bootstrap': False, 'max_depth': None, 'max_features': 8, 'n_estimators': 30}
nan {'bootstrap': False, 'max_depth': None, 'max_features': 8, 'n_estimators': 50}
```

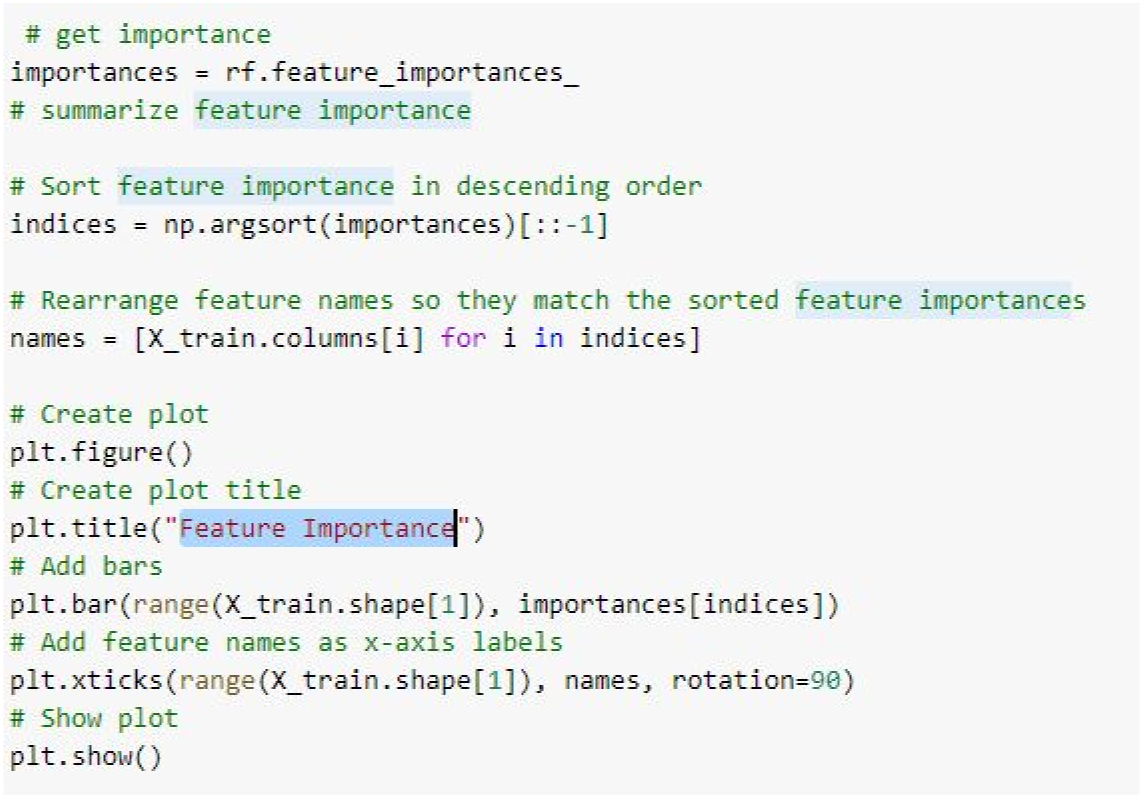
Best Estimator

```
#find the best model of grid search
grid_search_forest.best_estimator_
```

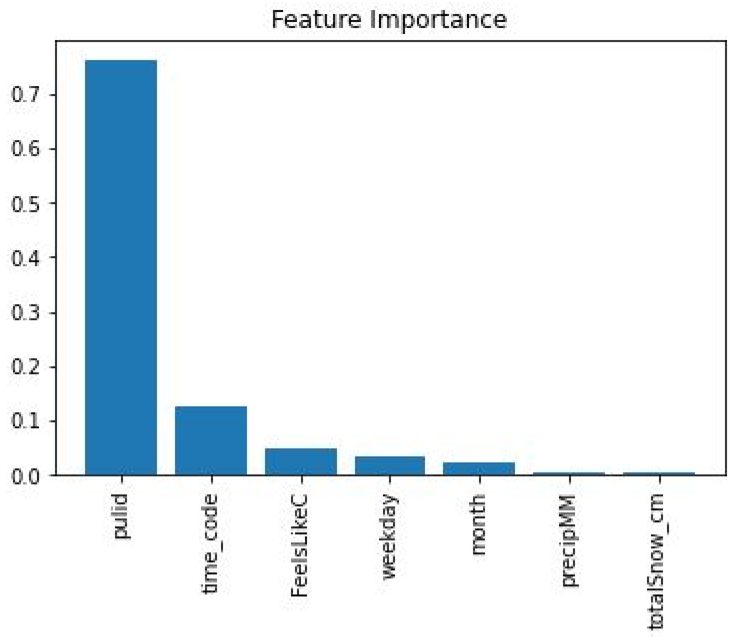
```
RandomForestRegressor(bootstrap=True, ccp_alpha=0.0, criterion='mse',
                      max_depth=None, max_features=3, max_leaf_nodes=None,
                      max_samples=None, min_impurity_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=1,
                      min_samples_split=2, min_weight_fraction_leaf=0.0,
                      n_estimators=50, n_jobs=None, oob_score=False,
                      random_state=None, verbose=0, warm_start=False)
```

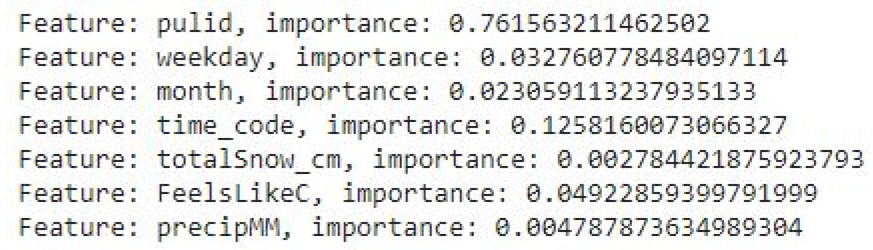
Performance of Tuned Model

- Feature Importance



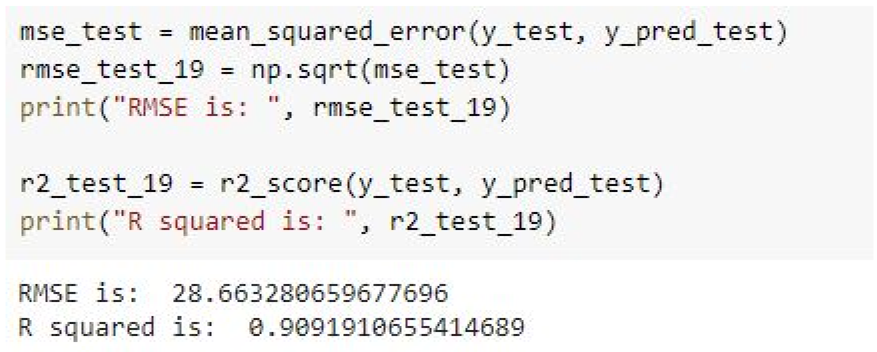
```
 # get importance
importances = rf.feature_importances_
# summarize feature importance

# Sort feature importance in descending order
indices = np.argsort(importances)[::-1]

# Rearrange feature names so they match the sorted feature importances
names = [X_train.columns[i] for i in indices]

# Create plot
plt.figure()
# Create plot title
plt.title("Feature Importance")
# Add bars
plt.bar(range(X_train.shape[1]), importances[indices])
# Add feature names as x-axis labels
plt.xticks(range(X_train.shape[1]), names, rotation=90)
# Show plot
plt.show()
```





```
Feature: pulid, importance: 0.761563211462502
Feature: weekday, importance: 0.032760778484097114
Feature: month, importance: 0.023059113237935133
Feature: time_code, importance: 0.1258160073066327
Feature: totalSnow_cm, importance: 0.002784421875923793
Feature: FeelsLikeC, importance: 0.04922859399791999
Feature: precipMM, importance: 0.004787873634989304
```

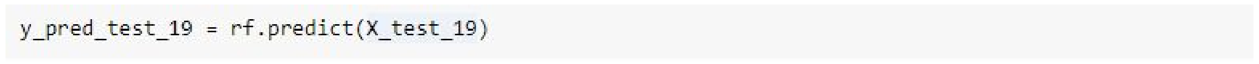
- RMSE & R-squared on Test Data 2018



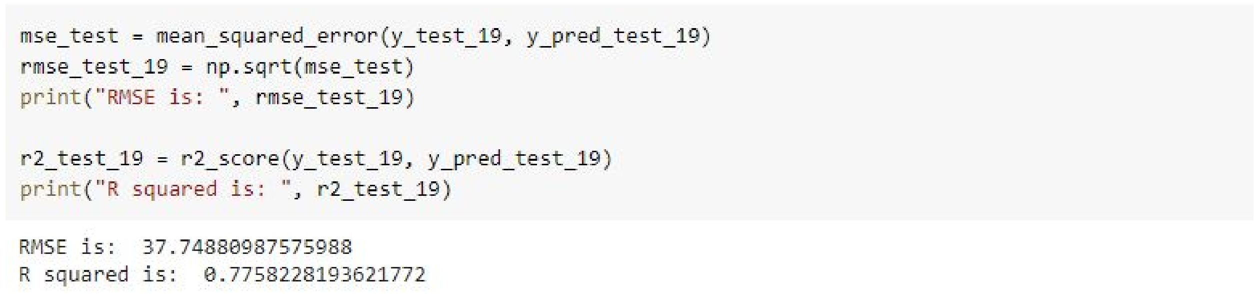
```
mse_test = mean_squared_error(y_test, y_pred_test)
rmse_test_19 = np.sqrt(mse_test)
print("RMSE is: ", rmse_test_19)

r2_test_19 = r2_score(y_test, y_pred_test)
print("R squared is: ", r2_test_19)
```

```
RMSE is:  28.663280659677696
R squared is:  0.9091910655414689
```

- RMSE & R-squared on Test Data 2019



```
y_pred_test_19 = rf.predict(X_test_19)
```



```
mse_test = mean_squared_error(y_test_19, y_pred_test_19)
rmse_test_19 = np.sqrt(mse_test)
print("RMSE is: ", rmse_test_19)

r2_test_19 = r2_score(y_test_19, y_pred_test_19)
print("R squared is: ", r2_test_19)
```

```
RMSE is:  37.74880987575988
R squared is:  0.7758228193621772
```

Comparison to benchmarks

We set the latest research by Wickramasinghe et al. 2019 as our benchmark standard. The authors used TLC trip records from 2017-18 as their training dataset. Hour,Day Of Week, Month, Drop Off ZoneID were used as features for predicting hourly drop offs on the test data. They achieved a root mean squared error of 0.01. As compared to our model’s rmse of 26.6, it is quite low. The reasons behind their results may be the difference in target variable, it’s resolution in terms of time interval. They set a larger prediction time window of one hour as compared to thirty minutes in our case. Also, the features used were limited to the given TLC dataset. We considered weather factors in addition to those given in the dataset, to make our model more generalizable to the real world.

Implications of current model

- Reduction in overfitting: by averaging several trees, there is a significantly lower risk of overfitting.
- It has an effective method for estimating missing data and maintains accuracy when a large proportion of the data are missing
- It is very easy to measure the relative importance of each feature on the prediction.

Limitations of the current model

- Complex and computationally expensive.
- Cannot extrapolate at all to data that are outside the range that they have trained.
- Random forest can feel like a black box approach for statistical modelers. we have very little control on what the model does except few Hyper Parameters.

Closing reflections

It seems that Random Forest Regressor is a good choice of method for this kind of estimation task. The model achieved a clearly better score than trivial averages when attempting to estimate taxi demand, and was able to utilize the addition of different input parameters well. However, the final results are not very strong, and while it is possible that some other feature exists that could increase the accuracy of the predictions even further, it is more likely that the taxi ridership of New York is too random to get good estimates. From previous work it seems like it is easier to make accurate predictions in areas with higher levels of taxi ridership.

A strong belief is that incorporating major and minor events could be a very strong indicator. Events that don't occur on a regular basis weekends or New Year's day is not at all captured by the features attempted in this paper. If data for example sporting events or concerts could be collected and mapped to the input it is possible that prediction rates would increase.

***