INTRODUCTION

This report provides a detailed analysis on the work done in participation of the Kaggle competition. All the coding work for this assignment was done in a Linux Ubuntu 14.04 operating system using Python 3.4. The machine learning libraries scikit and LightBGM were used. Other python libraries that were included are numpy, pandas, matplotlib and seaborn. The overall aim of this assignment was to develop a machine learning system for a Kaggle competition. In this competition the players are tasked with predicting the taxi fare for cabs operating in New York. A huge dataset with over 55 million rows which consisted of a number of features were provided. Part 1 of this assignment is to learn about the properties of the dataset and find interesting patterns in it. The second part is to develop a machine learning system based on the patterns observed previously. Part 3 involves forming a team with other Kaggle community members to explore the benefits of teamwork in data mining. This part also involves analysing the ethical consequences in using the chosen features for the machine learning system.

Link to my Kaggle profile :- https://www.kaggle.com/lordvader6023

PART 1

The objective of this part is to analyse the given dataset and examine the quality, completeness and representation of all the features. The various characteristics of the different features must be properly understood in order to do feature engineering for the machine learning system. Before exploring the dataset it is important to have a business understanding of the data. Since the goal here is to find the fare amount from various features it is reasonable to assume that the system would be of most use to taxi service providers like Uber, Ola cabs etc. The obvious use of this system would be to calculate the fare for a taxi ride. But apart from that the system may be used to find which neighbourhoods require more taxis and at what time of the day the rides would be more expensive. It can also provide insight into which day of the week has the highest number of rides and which day yields maximum profit. The exploration performed here builds on the work done by Will Koehrsen in his kernel (https://www.kaggle.com/willkoehrsen/a-walkthrough-and-a-challenge).

The training set provided here is a csv file that consists of about 55 million rows and the following input features.

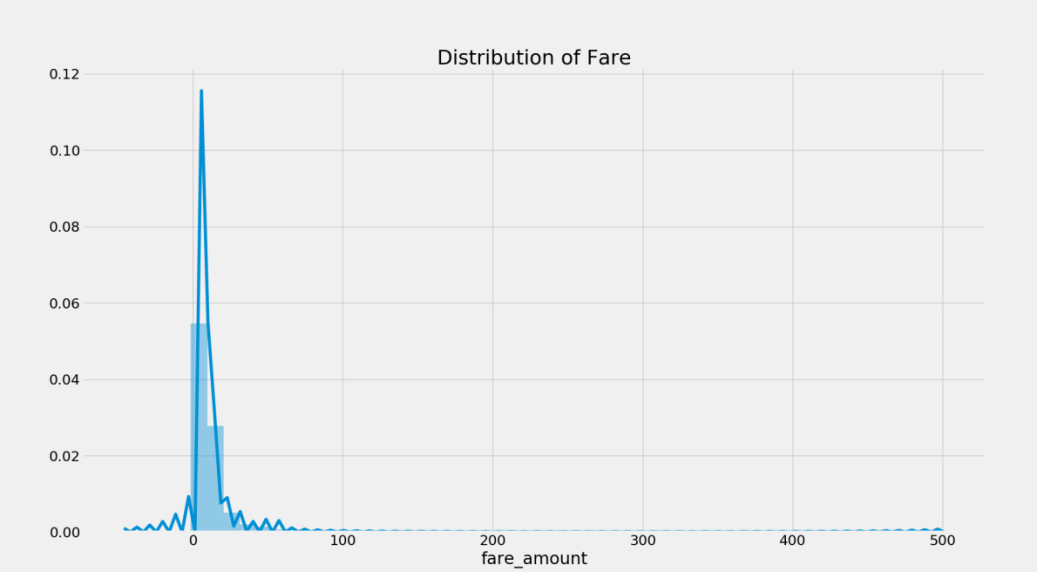
1. Pickup datetime
2. Pickup longitude
3. Pickup latitude
4. Dropoff longitude
5. Dropoff latitude
6. Passenger Count

The target variable here is the fare amount. In order to do data exploration more easily by limiting the computation time, the huge csv file was split and relatively small chunk

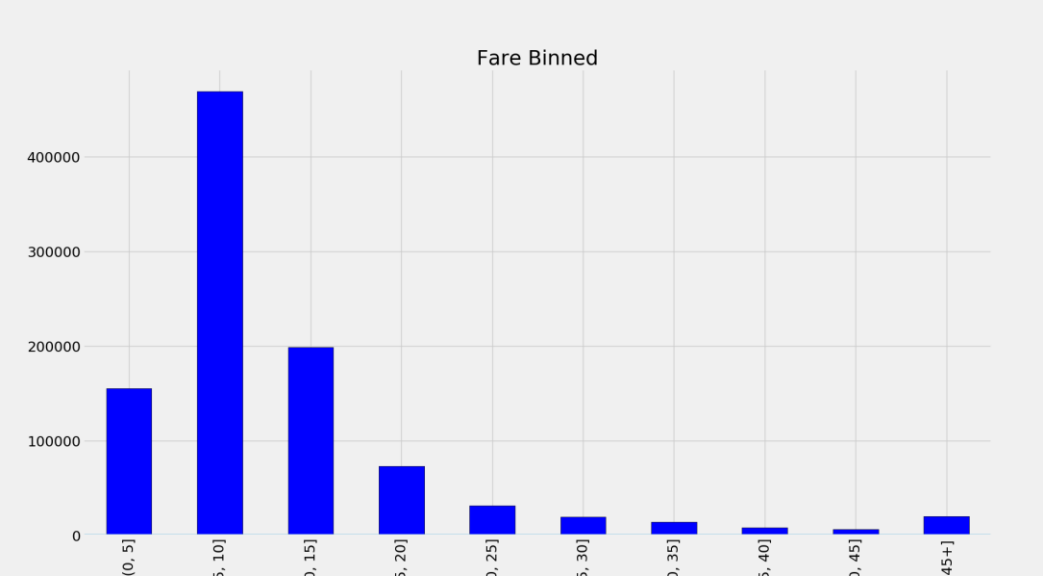
containing 1 million rows was used here. All the tests for finding patterns in the dataset were done using python scripts.

The .describe() method in python was used to get a rough idea of the data. It showed that the passenger count had a maximum value of 208 which does not seem likely for a cab. The maximum and minimum values of latitude and longitude showed a huge deviation from its average value. The target variable fare amount also had huge values that seemed improbable and negative values which are obviously impossible.

The first variable that was examined in detail was the fare amount. The distplot function in the seaborn library was used to get a kernel density estimate plot.

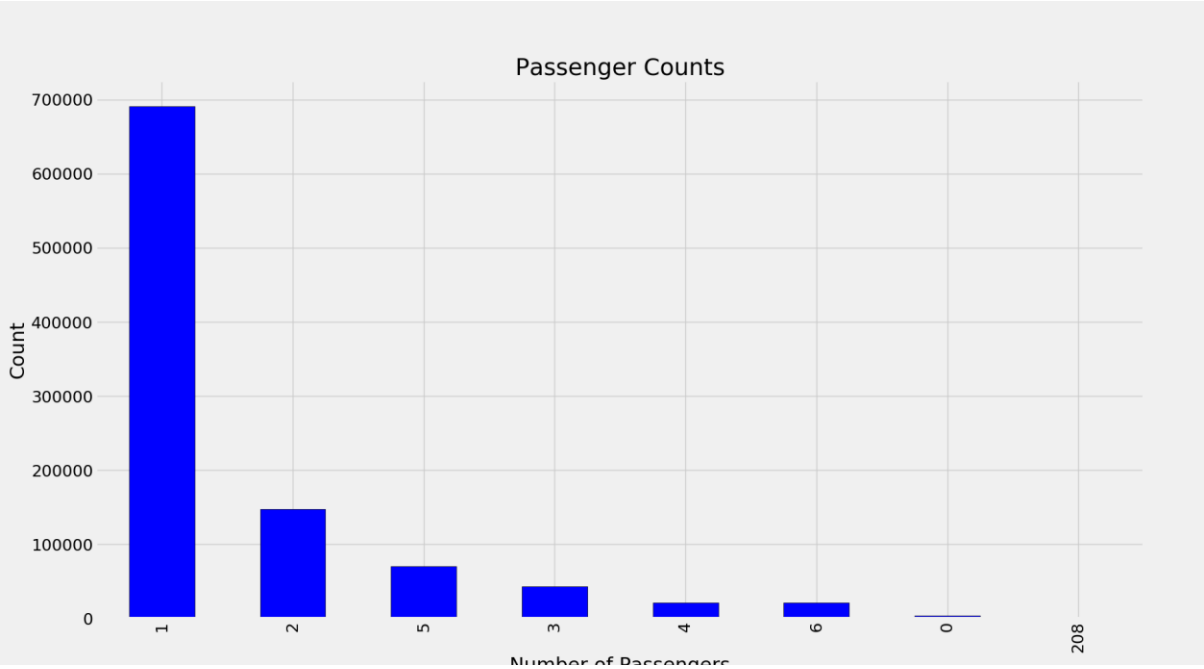


The graph shows that the vast majority of values range from 0 to 100 dollars. There is also a range of values which falls below zero. This indicates incorrect data in the set. A histogram was also created for the fare amount variable using the displot function in the seaborn library. The fare was binned in order to convert a continuous variable into a discrete variable.



Each bin can be said to contain all the instances from that fare range. The binned histogram highlights another important characteristic about the data. There are only significantly fewer instances for some bins than for others which mean that the dataset is unbalanced. This is a scenario where the number of observations belonging to one class is significantly lower than those belonging to the other classes. In this situation, the predictive model developed using conventional machine learning algorithms could be biased and inaccurate.

The next feature that was put under examination was the passenger count. A similar histogram was created for this as well.



This shows that the vast majority of rides have one or two passengers. It also shows a value of 208 at one point. So it is reasonable to assume that if the entire 55 million rows were examined there would probably be more incorrect values. Another interesting aspect to note here is that there seems to be a significant number of rides with zero passengers. The number of instances seems to be a little high for this to simply be an error in the data.

The latitude and longitude values had also indicated the presence of outliers when the .describe() method was used. To find the range of correct latitude and longitude values a python script was used to display the 2.5 and 97.5 percentile range of values.

Pickup_latitude 2.5% 40.64 97.5% 40.8

Pickup_longitude 2.5% -74.01 97.5% -73.78

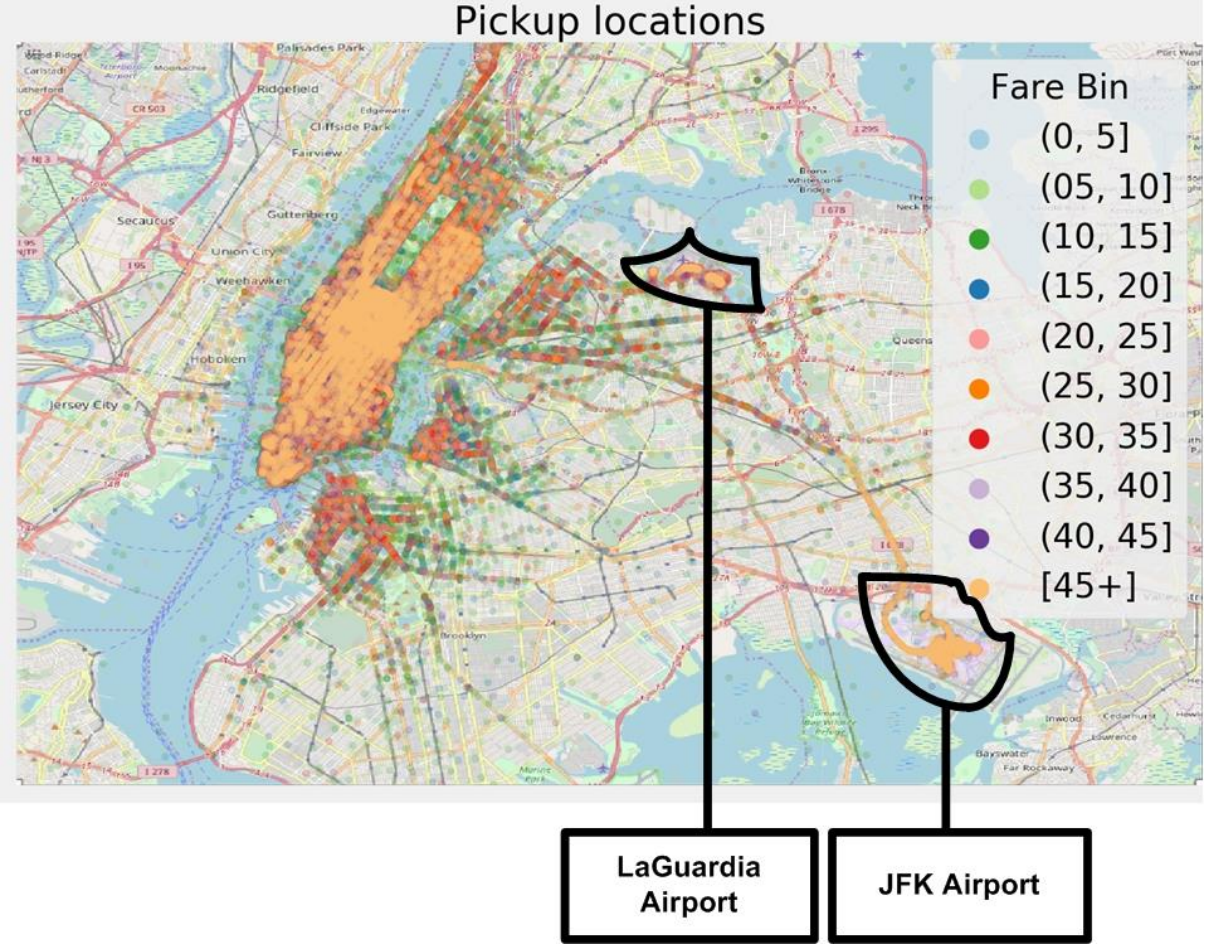
Dropoff_latitude 2.5% 40.64 97.5% 40.81

Dropoff_longitude 2.5% -74.01 97.5% -73.78

An internet search was also carried out to find the latitude and longitude boundaries of New York. The latitude ranges from 40 to 42 and the longitude ranges from -75 to -72.

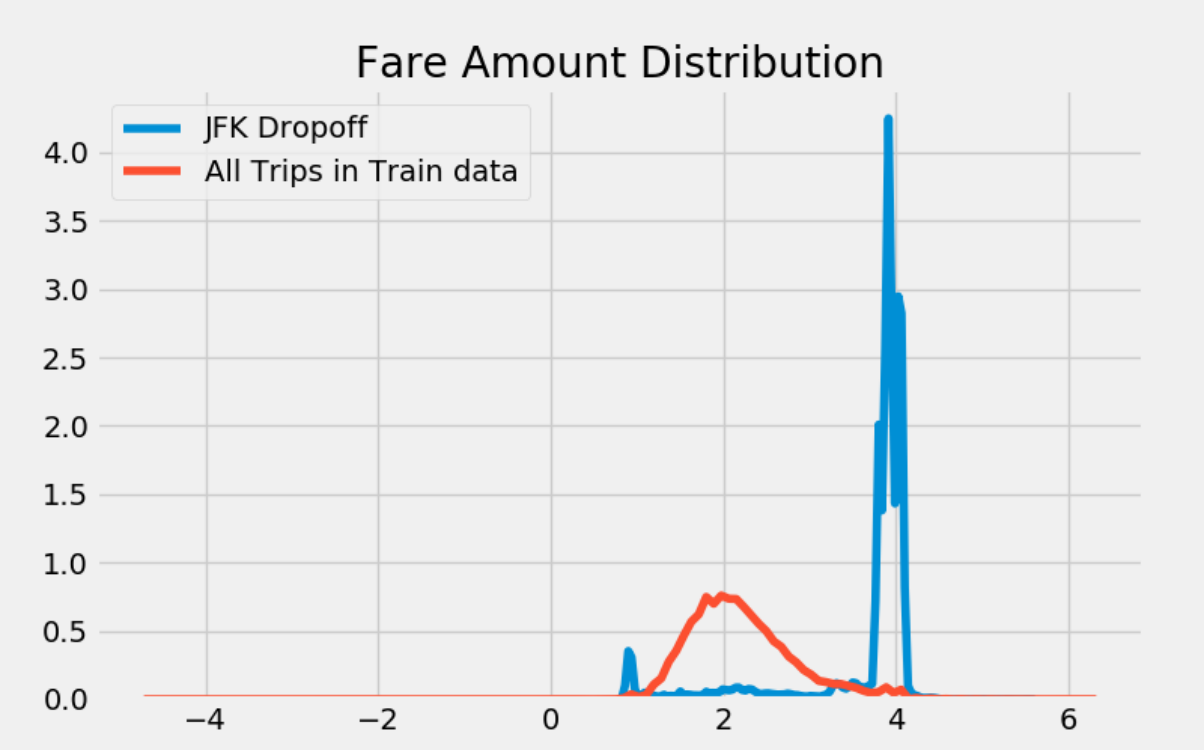
The latitude and longitude values of those instances with passengers count zero were analysed. It was found that for some of those instances the pickup and dropoff co-ordinates had the exact same values. This indicates a cab was called to a location and later cancelled, so the pickup and dropoff co-ordinates become the same but a cancellation fee gets charged.

The next test was carried out find if there would be any interesting patterns in the fare amount values over a map of New York. Each of different fare bins made earlier was assigned a colour. A random sample of 100000 rows was taken. Each instance was marked on the map as a spot with the colour of the spot indicating its fare bin. The map generated by using pickup latitudes and longitudes are given below. The map generated using drop off co-ordinates also looks similar.



At first glance it can be observed that the entire region of Manhatten has higher fares compared to the rest of the map. In addition to this high fare clusters can be observed at the two major airports in New York. A pickup or dropoff at an airport can usually add an overhead due to the waiting time spent at the airport.

This cluster was only spotted by a simple visual inspection. It was verified in a more rigorous manner by a simple python script to calculate the average fare amount for the overall dataset and the average fare amount for just the airport rides.



This graph clearly shows the spike in fares for dropping off passengers at the JFK airport. A similar graph was obtained for the other airport as well.

PART 2

- **Initial Design of the System**

The machine learning problem here is to create a system that can accurately predict the fare amount of a New York cab by using the features mentioned earlier. Since the fare amount is a continuous variable this falls under the category of regression.

Before selecting the appropriate regression algorithm the data should be cleaned and some necessary feature engineering be performed. Feature engineering is the process of using domain knowledge of the data to create features that make machine learning algorithms work. The success of any machine learning algorithm depends on how the data is presented. Feature engineering converts the inputs into things the algorithm can understand more easily.

The target variable fare amount can be cleaned by removing all instances with fare amount less than $2.5 and greater than $500. The lower limit was selected as $2.5 because this is the minimum fare for a cab in New York. Similarly with a cab fare of more than $500, the passenger would easily have left New York. The binned histogram plot for the fare amount as mentioned before clearly indicates the dataset is unbalanced. This can be resolved by making sure that both the test set and validation set would have instances from all the fare bins. The stratify option from the sklearn.model_selection.train_test_split() in the scikit machine learning library was used for this purpose. This ensures that the system does not end up with a lot of outliers in terms of the target in either the validation or training set.

As previously mentioned the passenger count variable was shown to have some incorrect values in the training set. The actual range of likely values for number of passengers can be obtained by analysing the given test set. It was shown to have values ranging from 0 to 8. So instances in the training set having values greater than 8 can be removed.

The co-ordinate variables i.e pickup latitude, pickup longitude, dropoff latitude and dropoff longitude have to be cleaned next. The range of latitude and longitude values that corresponds to New York geographically has been observed earlier. All the instances that fall outside this range can also be deleted. Some interesting feature engineering operations can be performed on these co-ordinate variables. It would be logical to assume that in order to predict the fare, the difference between pickup and dropoff co-ordinates would be a useful feature. Hence two additional features were created, representing the latitude and longitude differences respectively. Another useful feature for predicting the fare amount would the distance between the pickup and dropoff points. While it would impossible to predict the exact distance, a rough approximation may be obtained by calculating the Euclidean distance. The Euclidean distance can be found from the latitude and longitude values. This was added as another feature to the dataset. The datetime column in the dataset can be parsed using the pandas library in python. But this column was not used in this initial design.

The next step is to select an appropriate machine learning algorithm for this task. A random forest regression algorithm was chosen here. Random forests are ensemble learning methods used for classification and regression, that operate by creating a number of different decision trees during training time and merges them together later to produce a better model. Since the dataset used here has unbalanced data a single decision tree can lead to overfitting. Since this ensemble learning method better handles for overfitting and creates a non-linear model as compared to a simple linear regression algorithm, it was deemed suitable for this task. The low variance of this model could yield good results. The RandomForestRegressor algorithm from the scikit machine learning library was used here. The hyperparameters in the model were kept to their default values except for n_estimators and max_depth; both of which were set to 20.

For evaluating the quality of a machine learning system it is important to choose the right metric for examining the models. In this instance the Root Mean Squared Error (RMSE) and Mean Absolute Percentage Error (MAPE) were selected as the metrics for evaluation. The RMSE is the metric used in the competition but MAPE was also selected because it is more interpretable. These two metrics can be used to check how effectively machine learning works on this task. A naïve guess is made by calculating the average of the target variable in the training data. If machine learning is actually taking place the RMSE and MAPE of the learned model will be less than that obtained by the naïve guess.

The final python script called ‘randomforest.py’ used for this regression algorithm is provided under Part2 directory in the Source Code. For this initial design the algorithm was run with 100000 rows of training data. It obtained a score of 4.25722 in the Kaggle competition leaderboard.

- **Intermediate Designs**

1. Random Forest Regression

The random forest regression machine learning system was improved to see if a higher score could be obtained using this system. In order to find out how to improve the feature list it was necessary to ascertain which features performed well and which ones did poorly. The random_forest.feature_importances_ attribute from scikit was used to make this assessment. It was found that Euclidean distance performed better than all the other features by a large margin. The Euclidean distance only provides a rough approximation which is proportional to the distance between starting and ending point of a taxi ride. It would be impossible to find the exact distance travelled since that requires the route taken, roads available and street layout. Since none of these factors are available in the dataset an exact distance calculation cannot be done. However the distance parameter can be improved if the curvature of the earth is taken into account. A new feature which considers this called haversine distance was added to the list of features.

Other new useful features added to the system improved the model by providing an insight into a characteristic that was mentioned in Part 1. It was found that picking up or dropping off a passenger at the airport yields a higher fare. A function was written to return the minimum distance from pickup or dropoff co-ordinates to each airport. These features when added to the system can take into account another factor that affects the fare. The algorithm was first run for 1 million rows of training data. During training time 10% of the instances were used for validation alone. Validation is necessary to determine the performance of a system before using it on the actual test set and submitting. As mentioned before the RMSE and MAPE metrics were used for this purpose. When the test set for this system was uploaded to the Kaggle competition it received a score of 3.63277. The training data was increased to 10 million rows; which received a score of 3.47142. However even though 55 million rows were available the training dataset was not increased beyond this point since it may lead to overfitting.

2. Gradient Boosting Decision Trees

At this point I had found another interesting kernel by Sylas which provided a different algorithm and a different approach to feature engineering (https://www.kaggle.com/jsylas/python-version-of-top-ten-rank-r-22-m-2-88). This algorithm was improved upon to see if a better system could be created.

The ‘gbdt’ algorithm from the LightBGM library in python was used here. This algorithm also produces an ensemble of decision trees which are later merged. However unlike the random forest algorithm it builds the model in a stage-wise fashion like other boosting methods do and it generalizes them by allowing optimization of an arbitrary differentiable loss function. GBDTs require only minimal fine tuning unlike deep neural networks which frequently requires a lot of adjustments of the architecture. They can be trained in a few hours

but deep neural network training time is measured in days. All these factors have led me to the conclusion that this algorithm is highly suitable for this task.

This kernel had a few features which had not been considered in the random forest model that was previously built. A prominent feature that was present here was the date and time of taxi rides. It took the datetime column of the dataset and parsed it to obtain separate columns for hour, day, month, weekday and year. With these features this model can take into account any correlation between the datetime and the fare amount. This model had also converted the latitude and longitude into radians in order to normalise the values. Apart from haversine distance this model also considered the bearing of the taxi.

During a test run with a training data of 1 million rows this model outperformed the random forest model. Therefore I attempted to improve this model in order to increase its performance even more. This model had not converted the fare amount into a discrete categorical variable as had been done for the random forest. So the fare amount was binned in order to stratify the data and make sure that even though the dataset was unbalanced all bins got a fair representation in the model. The latitude and longitude columns were not engineered to remove outlier values. So the instances whose latitude and longitude values did not fall in the range specified in Part 1 was deleted. Just as in random forest 10% of the data during training time was used for validation only. This model was finally run with training data of 10 million rows and a score of 2.96372.

- **Final Design**

The two intermediary models specified above both managed to perform well in the leaderboard. But the gradient boosting model had a slight edge over the random forest because it works better than the other when the data is unbalanced. A prominent difference between the two is that in gradient boosting trees are built one at a time, where each new tree helps to correct errors made by previously trained tree; whereas in random forest each tree is trained independently, using a random sample of the data. This normally gives you better accuracy with fewer trees in gradient boosting. Gradient boosting can be used to solve almost all objective function that we can write gradient out. This means that if the parameters are tuned properly gradient boosting can provide optimal results. Even though gradient boosting takes longer to train it's faster for real time prediction than random forests. The reason for this is the large numbers of trees random forest algorithms are prone to building. Gradient boosting is on the other hand much faster to train as compared to deep neural networks. Hence it may be observed that this algorithm strikes a balance between training time requirements and execution speed during real-world prediction. In addition random forests are more prone to overfitting than gradient boosted trees. So a larger portion of training instances could be used to train the boosted trees. The RMSE and MAPE were calculated for both the training data and the validation data. It showed that for the random forest algorithm overfitting starts to take place after 5 to 10 million rows for this dataset. The gradient boosting algorithm can take in much more training instances before it starts to exhibit this problem. But only 10 million instances were used for training the gradient boosting algorithm. This was due to the computation time required for training. The system used here

had 8 GB of RAM and an Intel I5 processor. The training time for 10 million rows took approximately 3 hours. With some more computation time 25 to 30 million rows could probably be used and the performance improved further. Therefore the gradient boosting algorithm has been selected as my final model. The python codes for both the models are given in the source code zip file that is attached.