NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

CZ4041: Machine Learning

Project: New York City Taxi Trip Duration (Kaggle)

Submitted by: Group 21

Name	**Matric No.**
Chang Heen Sunn	U1920383H
Hasan Mohammad Yusuf	U1822478E
Ng Man Chun	U1722138D
Padhi Abhinandan	U1823860D
Wong Zhen Yan	U1822520G

SCHOOL OF COMPUTER SCIENCE & ENGINEERING

NANYANG TECHNOLOGICAL UNIVERSITY

Contents

1. Introduction

1.1 Motivation

Estimating the duration of trips between locations or places is a common task that both travellers and passengers alike adopt in order to plan their trips. By estimating the time taken for their trips, travellers would be able to factor these timings into their schedules, allowing for more accurate and precise schedule planning. This is important since estimating trip duration provides an indicator on the type of transportation travellers would prefer. For instance, they can choose to take public transportation if they have an abundance of time or opt to take a taxi if they are short of time.

The option of taking taxis instead of public transport is often associated with a shorter travelling time since it is assumed that taxi drivers are familiar with the routes and will take the fastest route from the starting location to the destination. Therefore, with information of taxis that contain the pick-up and drop-off locations and the duration the trip took, the task of predicting future trip duration of taxis between two locations would be possible.

In this project, we will look into various machine learning techniques that can be used for predicting the taxi trip duration, and evaluate the results of each technique.

1.2 Problem Statement

The problem statement for this project is as follows: *Build a model that predicts the total ride duration of Taxi Trips in New York City*.

1.3 Structure of Report

The subsequent sections of this report consist of the following: methodology, experiments and results, and conclusion. The methodology goes into detail about several aspects of the project, which are as follows. Firstly, we will introduce the dataset by providing a description of the data's features/fields. Subsequently, the Exploratory Data Analysis (EDA), Feature Engineering/Selection methods, and the approaches we used to solve the problem of NYC Taxi Trip Duration prediction, i.e., the Machine Learning and Deep Learning models (XGBoost and Neural Networks) will be discussed. The experiments and results of each approach will then be displayed and analysed. The last section of this report is the conclusion where we summarize the project, briefly explain the challenges we faced (and solutions to them), and propose possible improvements to this problem for future studies.

Please note that the screenshots of the Kaggle submissions for this project as well as a breakdown of the team members' work contribution is provided in the Appendix.

2. Methodology

2.1 Dataset Description

For this project, we used two datasets, namely the *New York City Taxi Trip Duration Dataset* and the *New York City Taxi Trip with OSRM Dataset*, both of which are from Kaggle.

The *New York City Taxi Trip Duration Dataset* contains the following fields:

- **id** - a unique identifier for each trip
- **vendor_id** - a code indicating the provider associated with the trip record
- **pickup_datetime** - date and time when the meter was engaged
- **dropoff_datetime** - date and time when the meter was disengaged
- **passenger_count** - the number of passengers in the vehicle (driver entered value)
- **pickup_longitude** - the longitude where the meter was engaged
- **pickup_latitude** - the latitude where the meter was engaged
- **dropoff_longitude** - the longitude where the meter was disengaged
- **dropoff_latitude** - the latitude where the meter was disengaged
- **store_and_fwd_flag** - this flag indicates whether the trip record was held in vehicle memory before sending to the vendor because the vehicle did not have a connection to the server - Y = store and forward; N = not a store and forward trip
- **trip_duration** - duration of the trip in seconds

The *New York City Taxi Trip with OSRM Dataset* contains the following fields:

- **starting_street** - the street where the taxi-trip starts.
- **end_street** - the street where the taxi-trip ends.
- **total_distance** - the total distance is measured between the pickup coordinates and the drop-off coordinates in train.csv and test.csv. he unit is meters.
- **total_travel_time** - the total travel time for that data point in seconds
- **number_of_steps** - the number of steps on that trip. One step consists of some driving and an action the taxi needs to perform. It can be something like a turn or going on to a highway.
- **street_for_each_step** - a list of streets where each step occurs. Multiple steps can be performed on the same street. Therefore the same street might occur multiple times.
- **distance_per_step** - the distance for each step.
- **travel_time_per_step** - the travel time for each step
- **step_maneuvers** - the action (or maneuver) performed in each step.
- **step_direction** - the direction for each action (or maneuver)
- **step_location_list** - the coordinates for each action (or maneuver)

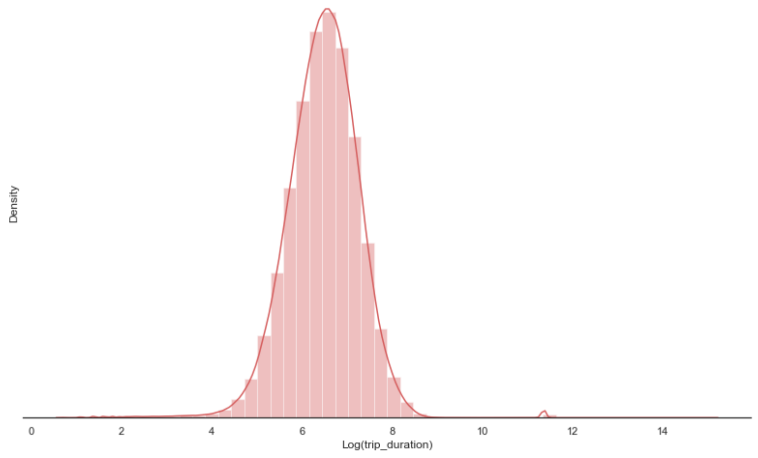
2.2 Exploratory Data Analysis

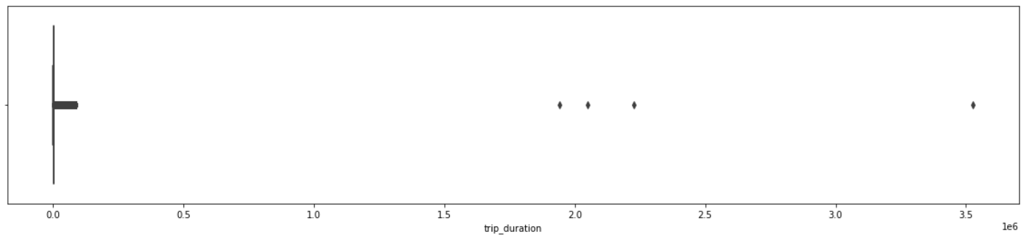
In order to better understand the dataset, we will first perform the task of Exploratory Data Analysis (EDA). Analysing the dataset through the use of visualisations will allow us to identify trends and patterns, outliers or anomalies and find out interesting relations among variables in the data.

The following section is split into 2 parts. For the first part, we will document and illustrate the various EDAs that we have performed on the dataset before feature engineering and selection was done. The second part will document the EDAs that we have performed after the process of feature engineering and selection.

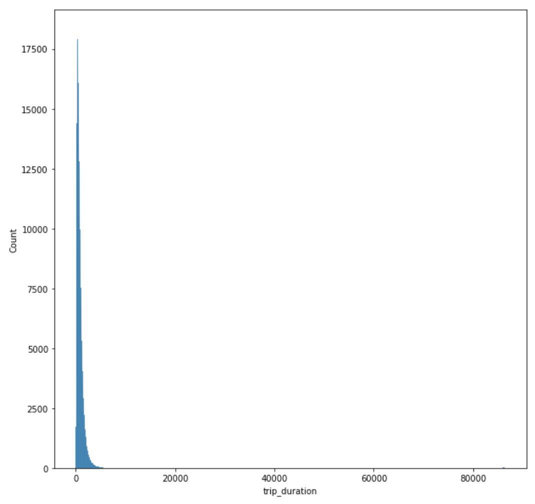
Part 1: EDAs before Feature Engineering and Selection

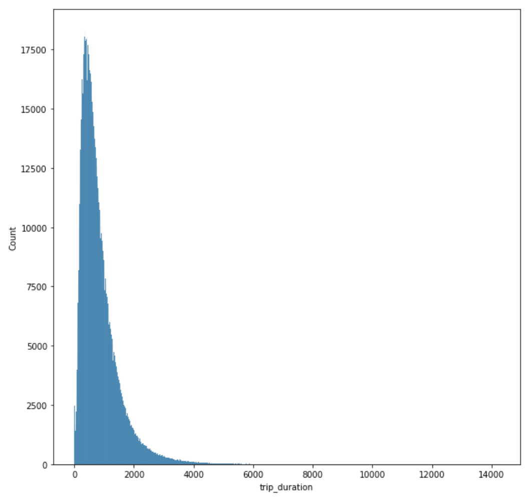
Trip duration





From our visualisation, we can identify outliers where the trip durations are extremely long. The longest trip lasted a duration of approximately 41 days. A reason for this could be due to the driver forgetting to switch off the taxi meter, resulting in such a long trip duration recorded.

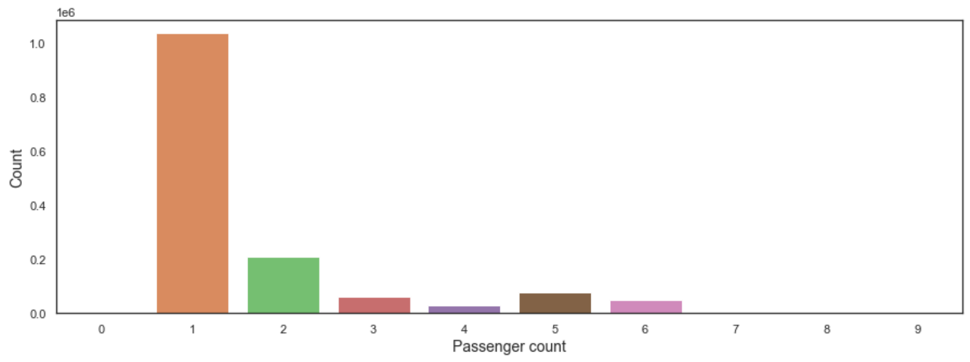




First figure: Trip durations less than 24 hours, second figure: Trip durations less than 4 hours

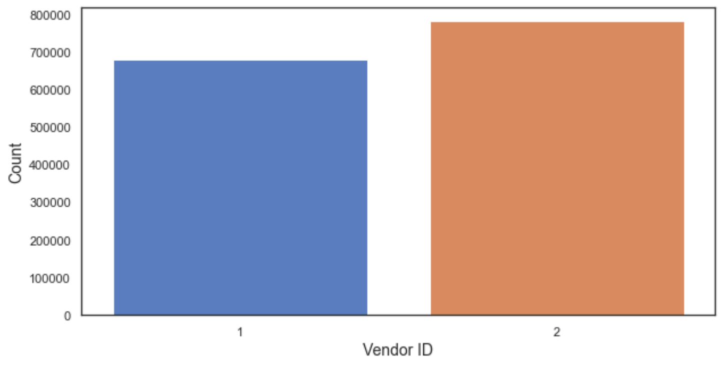
After further analysing trip durations that are less than 4 hours (14400 seconds), we can see that most trips are short trips which are less than an hour (3600 seconds).

Passenger Counts



From the above, we can see that trips with only 1 passenger have the highest count, followed by trips with 2 passengers. The maximum number for passenger count is 9.

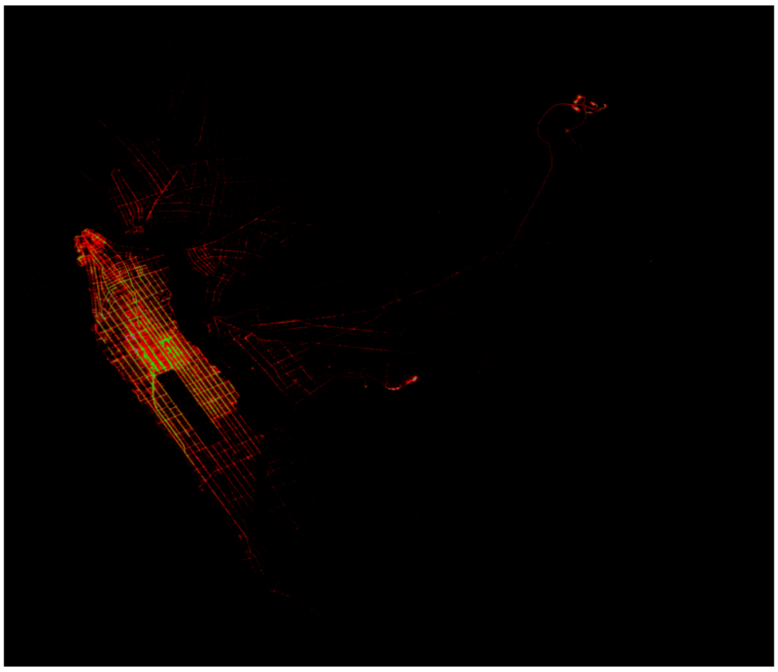
Vendor ID



In this dataset, there are 2 vendor providers for taxis. Vendor ID 2 has a higher trip count compared to Vendor ID 1. Reasons that Vendor ID 2 has a higher count could be due to it being more popular among passengers or that there are more taxis that are under this vendor.

Pick-up Location

In order to visualize the starting point of most trips, we visualised the coordinates of the pick-up location onto a heatmap. Red points signify that 1-10 trips in the given data have that point as the pickup point. Similarly, green points signify 10-50 trips and yellow points signify more than 50 trips.

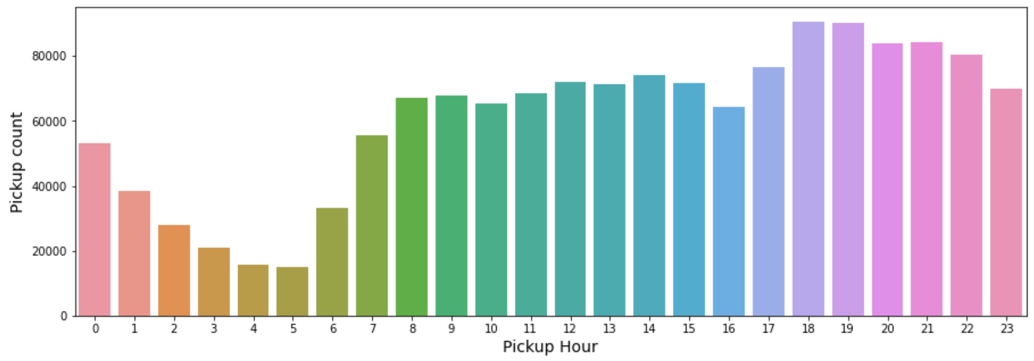


From the heatmap above, Manhattan is mostly yellow with a few specks of green, signifying many yellow points and a few green points. This indicates that the starting point for most taxi trips originate from Manhattan.

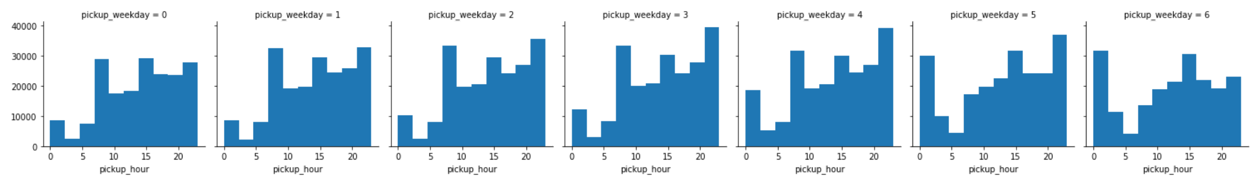
Part 2: EDAs after Feature engineering and selection

Spatial and Temporal behaviour of people based on Trip distributions

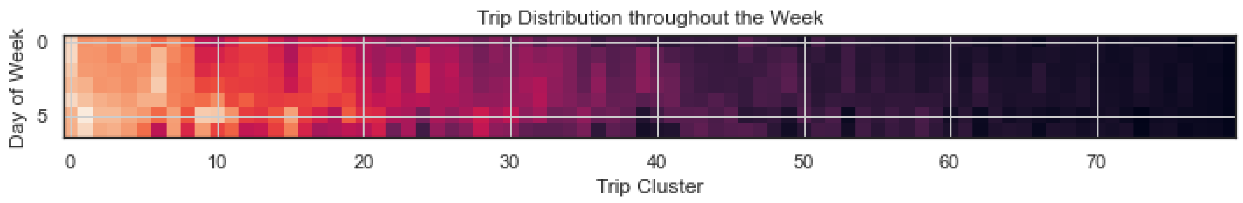
In this section, we will be exploring the spatial and temporal behavior of the people of New York as can be inferred by examining their cab usage.



From the above, we can see that the pickup hours that have the highest pickup count are at 18:00 and 19:00. These timings coincide with the end of work timing, where the majority of the passengers would take taxis as a way of commuting home after work.

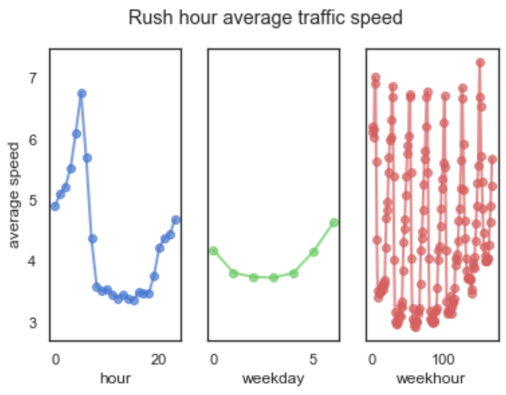


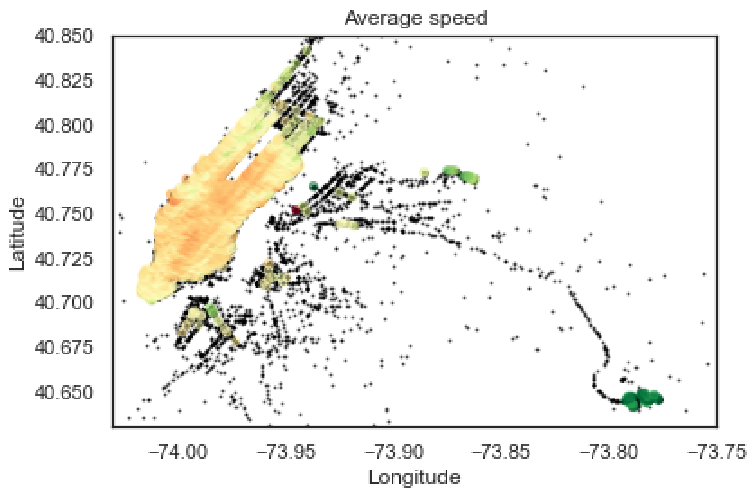
Upon visualising the passenger count by the individual days of the week, we can see that during weekdays, there are generally more pickups during the peak hours (morning and evening) where passengers would take a taxi to work or after work. However on weekends, there are generally more pickups after midnight where more people are out till late.



We can see that the pattern for trip distribution on the weekends differs from the weekdays. This coincides with our inference earlier.

Speed

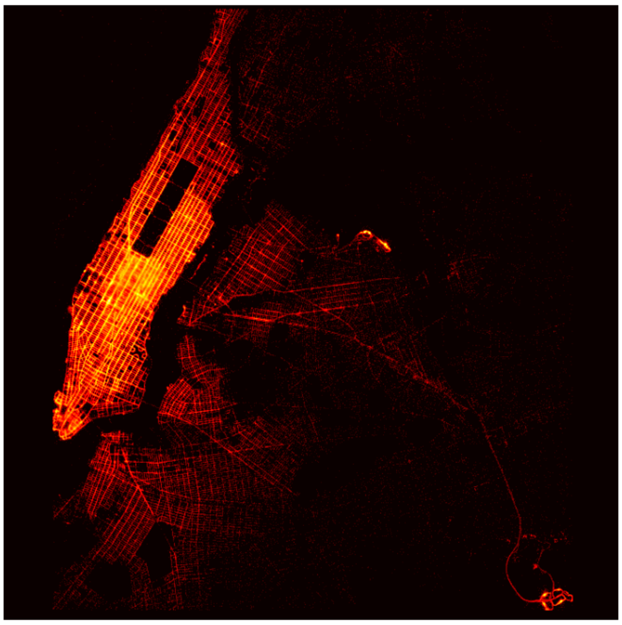




From the above, we are able to visualise the speeds of taxis based on the time of day, day of the week and their locations.

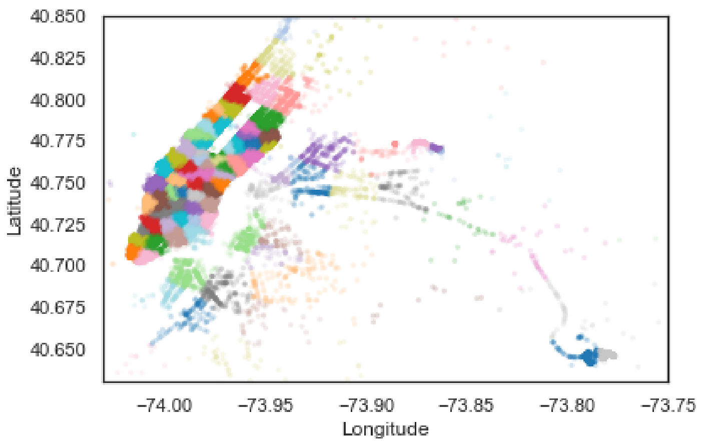
Pick-up and Drop-off Location

To visualise the pickup and dropoff locations, we will plot a spatial density plot.

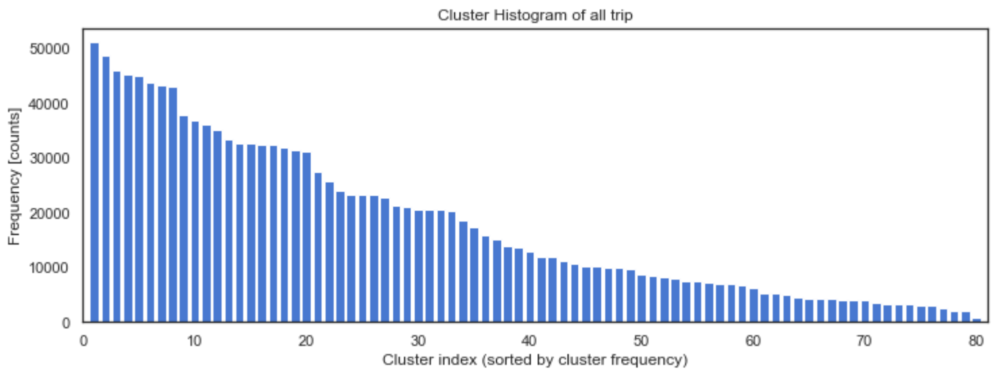


From the plot above, we are able to see which regions have the highest density. These regions indicate locations which have highest pickup and dropoff counts, and are reflected as yellow points on the map.

Pickup location clusters



From our visualisation above, we can see the different clusters for pickup locations. This allows us to visualise the locations of each cluster on the map and the range of latitudes and longitudes belonging to each cluster respectively.



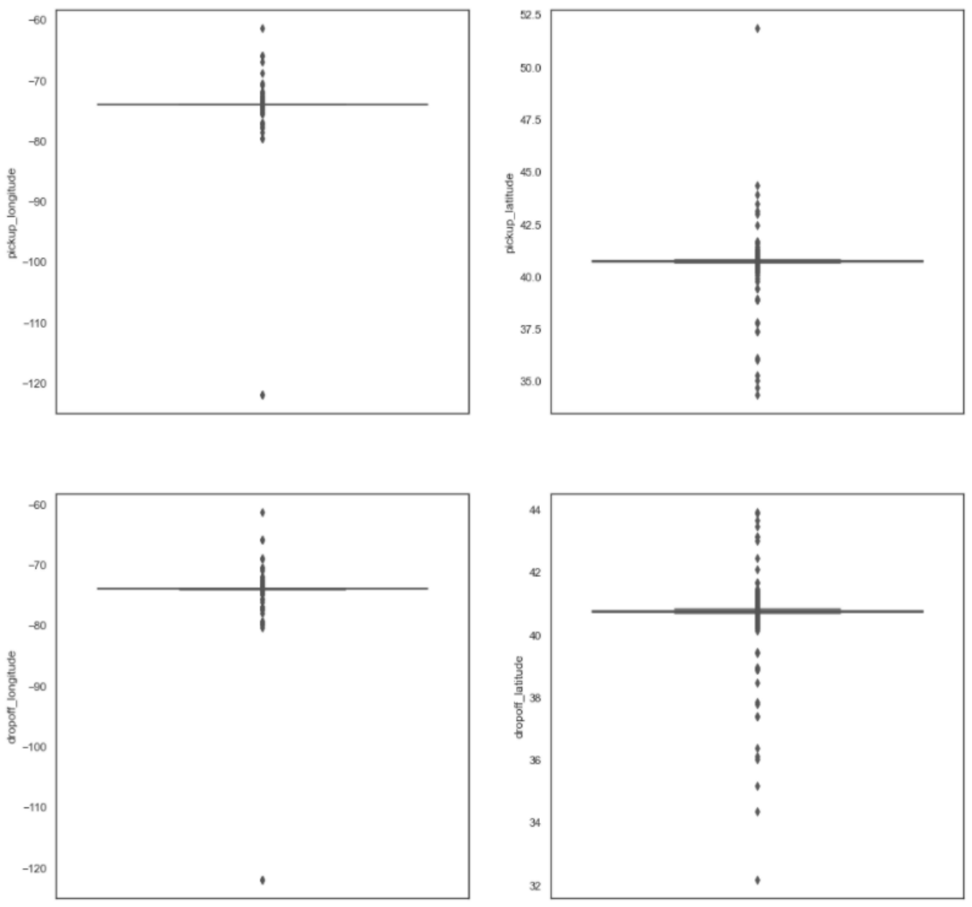
This histogram allows us to view the frequency for each respective cluster. This allows us to know which clusters have the highest and lowest frequency respectively.

2.3 Feature Engineering and Selection

Before implementing our respective models, we performed feature engineering and selection on the respective dataset.

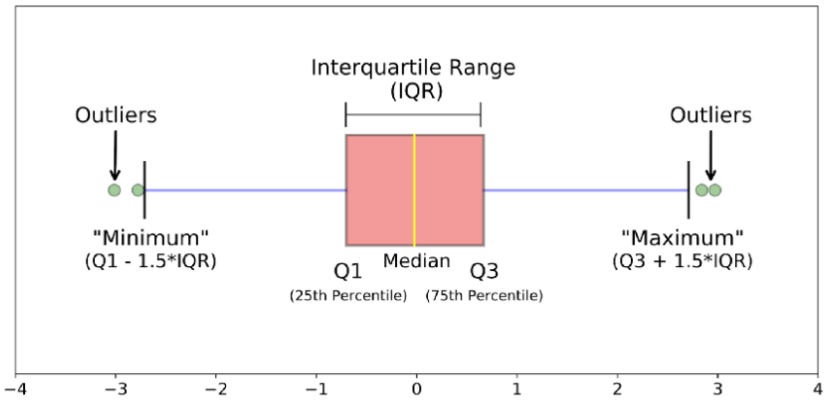
Outlier Detection

In order to remove outliers in the train data, the interquartile range and Z-score approaches were implemented.



Boxplot to observe the outliers for columns present in the dataset

Interquartile Range Approach



Different parts of a boxplot

In the above figure,

- Minimum - Minimum Value of the feature in the data.
- Maximum - Maximum Value of the feature in the data.
- The median is the median (or centre point), also called second quartile, of the data (resulting from the fact that the data is ordered).
- Q2 is the median (50th Percentile) of the feature in the dataset.
- Q1 is the 25th Percentile of the feature in the dataset (25% of the data is between minimum and Q1)
- Q3 is the 75th Percentile of the feature in the dataset (75% of the data is between minimum and Q1)

Interquartile Range is the difference between first and third quartile.

Any point lying outside a specific range (Lower Bound and Upper Bound) is considered an outlier. The range is as given below:

- Lower Bound: (Q1 - 1.5 * IQR)
- Upper Bound: (Q3 + 1.5 * IQR)

If the data value is more than upper bound, it is an outlier. If the data value is less than the lower bound, it is an outlier.

We tweaked the sensitivity range (Change the value of 1.5) to the value of 30 to produce the most optimum result.

Z-Score Approach

Z-score is a measure used in statistics to tell the distance any data point is away from the other data points. In other words, it informs us the number of standard deviations away from the data point is from the mean of the whole data.

For instance, a Z-score of 3 means that the data point is 3 SD far from the mean of the whole dataset.

Z-score is calculated using two values provided by the data — mean and standard deviation.

$$\text{Z-score} = \frac{x - \text{mean}}{\text{Standard Deviation}}$$

We tweaked the value of the threshold to produce the most optimum result.

If the magnitude of the Z-score of any point is greater than the threshold set, we consider it as an outlier.

Ultimately, the interquartile range approach was used in the final model because it provided a lower error rate and a better accuracy score in the prediction of results.

Feature Scaling

Machine learning algorithms that calculate distances between data require feature scaling. When calculating distances, if the feature with a higher value range is not scaled, the feature with a higher value range takes precedence.

Scaling is a must in many algorithms that require faster convergence, such as Neural Networks.

Due to the wide range of values in raw data, objective functions in some machine learning algorithms do not work correctly without normalization. All features should have their ranges normalized so that each contributes roughly proportionately to the final distance.

We tried three different feature scaling techniques:

Min-Max Scaler

$$x_{new} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

X-new = New scaled feature value
X-min = Smallest feature value
X-max = Highest feature value

Scale each feature to a specific range to transform it. This estimator scales and translates each feature individually so that it falls within the training set's given range, such as zero to one. If there are negative values, this Scaler shrinks the data to a range of -1 to 1. The range can be set to [0,1], [0,5], or [-1,1].

When the standard deviation is small and the distribution is not Gaussian, this Scaler performs well. It is sensitive to outliers.

Robust Scaler

This Scaler is robust to outliers, as the name implies. Scaling with the data's mean and standard deviation won't work if there are a lot of outliers.

The median is removed, and the data is scaled according to the quantile range (IQR: Interquartile Range). The interquartile range (IQR) is the range between the first and third quartiles (25th and 75th quantile). Because this Scaler's centering and scaling statistics are based on percentiles, they are unaffected by a few large marginal outliers. It's worth noting that the outliers remain in the transformed data. A non-linear transformation is required if separate outlier clipping is desired.

Standard Scaler

$$z = \frac{x - \mu}{\sigma}$$

$\mu =$ Mean
$\sigma =$ Standard Deviation

The StandardScaler assumes that data within each feature is normally distributed and scales it so that the distribution is centered around 0 with a standard deviation of 1. By computing the relevant statistics on the samples in the training set, each feature is independently centered and scaled. This is not the best Scaler to use if the data is not normally distributed.

We finally used the standard scaler as it improved our model performance the most. Our data was normally distributed and it helped to improve our model performance.

Date-time data columns

We had to convert the object type columns in the dataset to datetime data type. This would help us to extract all the data-time features from the data using the features provided from the datetime module.

```
train['pickup_datetime'] = pd.to_datetime(train.pickup_datetime)
test['pickup_datetime'] = pd.to_datetime(test.pickup_datetime)
```

Feature Extraction

Date-Time features

We used the datetime module to extract features like week, day, date, minute, second, weekofyear, etc from the columns ‘pickup_datetime’. A machine learning model only understands numbers and extracting these features helps create more detailed numeric data which enhances the model training and accuracy.

We also added additional features into the dataset based on the domain of taxis. We added these domain specific features as mentioned below:

- Trips taken at night are taken into account. (Less trips are taken at night and are shorter in time due to less traffic)
- Trips taken during rush hours are taken into account. (Rush hours indicate higher use of taxis, more traffic which leads to longer duration)
- Trips taken during weekday/weekend are taken into account. (More trips maybe taken on weekdays due to office work, More trips maybe taken on weekends where people want to travel, go for leisure activities)
- Trips taken during or near to holidays in the US are taken into account. (On the US Holidays, citizens might want to go out more)
- Trips taken during business days in the US are taken into account. (People travel to their workplace using taxis)

The extra datasets used to get US holidays and business days are as follows:

- from pandas.tseries.holiday import USFederalHolidayCalendar
- from pandas.tseries.offsets import CustomBusinessDay

Principal Component Analysis (PCA)

PCA is a dimensionality-reduction method that is often used to reduce the dimensionality of large data sets, by transforming a large set of variables into a smaller one that still contains most of the information in the large set.

The general benefits of PCA is as follows:

- Principal components capture the most variation in a dataset.
- PCA deals with the curse of dimensionality(Having too many features which reduces accuracy) by capturing the essence of data into a few principal components.

We use PCA to transform longitude and latitude coordinates. In this case it is not about dimension reduction since we transform 2D to 2D. The rotation could help for decision tree splits.

Distance between pickup and dropoff locations

To estimate the distance between the pickup and dropoff locations, we calculated it using various methods such as manhattan distance, haversine distance and OSRM distance to calculate it.

Manhattan Distance Approach

Manhattan distance is the distance between two points measured along axes at right angles. It is calculated using the pickup location latitude and longitude and dropoff location latitude and longitude.

$$\text{Manhattandistance} = \text{haversine}(\text{S}_{lat}, S_{lon}, H_{lat}, H_{lon}) + haversine(H_{lat}, H_{lon}, D_{lat}, D_{lon})$$

In the above formula, S represents pickup location, D represents the dropoff location and H is the hinge point. If we can find H where the longitudinal line running through S and the latitudinal line running through D' intersect, we can find the total distance as the sum of straight-line distances between S' to H' and between H' and D'.

Haversine distance Approach

Haversine formula determines the great-circle distance between two points on a sphere given their longitudes and latitudes.

The central angle between any two points on a sphere is:

$$\Theta = \frac{d}{r}$$

- *d* is the distance between the two points along a great circle of the sphere
- *r* is the radius of the sphere.

In our project, we approximated the earth radius to be 6371km.

The haversine formula allows the haversine of Θ to be computed directly from the latitude (represented by φ) and longitude (represented by λ) of the two points:

$$\text{hav}(\Theta) = \text{hav}(\varphi_2 - \varphi_1) + \cos(\varphi_1)\cos(\varphi_2)\,\text{hav}(\lambda_2 - \lambda_1)$$

- φ_1, φ_2 are the latitude of point 1 and latitude of point 2 (in radians),
- λ_1, λ_2 are the longitude of point 1 and longitude of point 2 (in radians).

To solve for the distance d, the arcsine (inverse sine) function is used:

$$\begin{aligned} d &= 2r \arcsin\left(\sqrt{\text{hav}(\varphi_2 - \varphi_1) + \cos(\varphi_1)\cos(\varphi_2)\,\text{hav}(\lambda_2 - \lambda_1)}\right) \\ &= 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos(\varphi_1)\cos(\varphi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right) \end{aligned}$$

Using Haversine distance is better than the Manhattan distance because the Earth is a sphere and the distance between two points is not calculated as a straight line distance (Manhattan distance).

However, both haversine distance and Manhattan distance are only rough estimates. In order to improve the accuracy, we incorporated the use of Open Source Routing Machine (OSRM).

Open Source Routing Machine (OSRM)
OSRM is a C++ implementation of a high-performance routing engine that is able to calculate the shortest paths in road networks.

For our project, we used the *New York City Taxi Trip with OSRM Dataset* to get the distance between the pickup and dropoff locations.

Using OSRM allowed us to calculate the fastest routes between both the pickup and dropoff locations. Since drivers are assumed to take the fastest routes (routes that have shortest path between locations), using OSRM distance would thus give a more accurate representation of the distance between locations.

2.4 Methodology of Predicting Taxi Trip Duration

In this project, two different approaches/algorithms were used for predicting the NYC Taxi Trip duration based on the original features in both datasets, as well as engineered features. The two approaches are: XGBoost (Extreme Gradient Boosting), and Neural Networks (Deep Feed-Forward Neural Networks).

2.4.1 XGBoost

Extreme Gradient Boosting, or XGBoost, is an enhanced and improved implementation of the gradient boosting decision tree algorithm designed for improved model performance and computational speed.

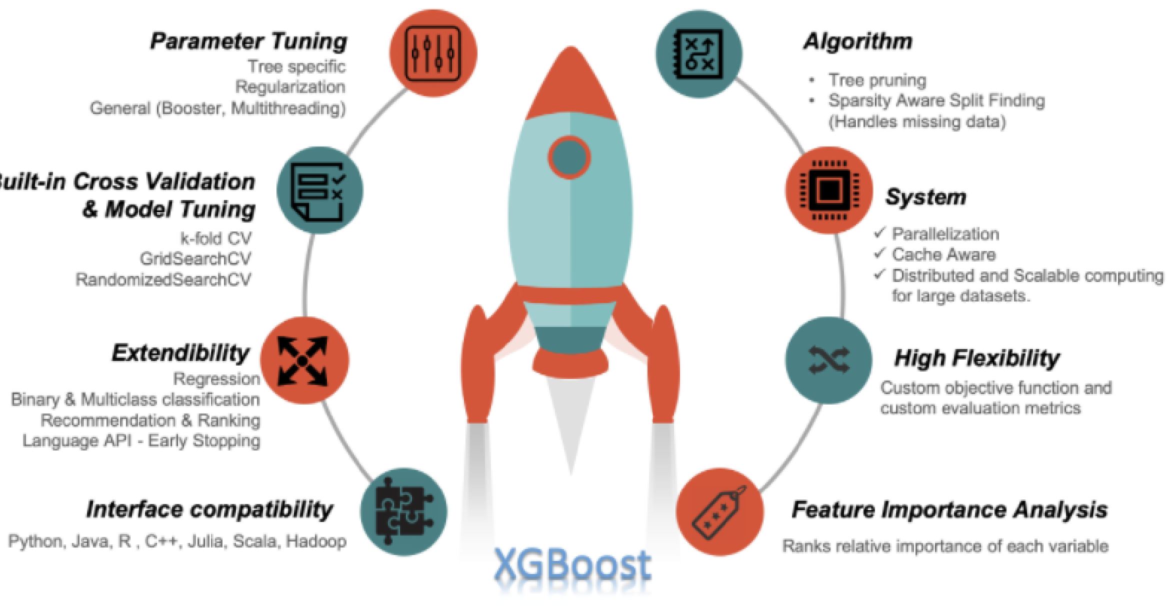
Gradient boosting is a technique that involves creating new models that estimate the residuals or errors of previous models, which are then combined to make the final prediction. Gradient boosting gets its name from the fact that it uses a gradient descent algorithm to mitigate loss while inserting new models. Gradient boosting changes weights by using a Gradient Descent algorithm, which iteratively optimizes the model's loss by updating weights using gradient (a path in the loss function). The discrepancy between the expected and actual value is usually referred to as a loss. We use MSE (Mean Squared Error) loss as an evaluation metric for regression algorithms, and logarithmic loss for classification problems.

$$w = w - \eta \nabla w$$

$$\nabla w = \frac{\partial L}{\partial w} \ \textit{where } L \textit{ is loss}$$

w represents the weight vector, **η** is the learning rate

XGBoost is a unique combination of software and hardware technologies designed to improve existing boosting techniques with precision and speed.


Parameter Tuning
Tree specific
Regularization
General (Booster, Multithreading)
Built-in Cross Validation & Model Tuning
k-fold CV
GridSearchCV
RandomizedSearchCV
Extendibility
Regression
Binary & Multiclass classification
Recommendation & Ranking
Language API - Early Stopping
Interface compatibility
Python, Java, R , C++, Julia, Scala, Hadoop
Algorithm
Tree pruning
Sparsity Aware Split Finding (Handles missing data)
System
Parallelization
Cache Aware
Distributed and Scalable computing for large datasets.
High Flexibility
Custom objective function and custom evaluation metrics
Feature Importance Analysis
Ranks relative importance of each variable
XGBoost

XGBoost Overview

Because of its robustness in handling overfitting, missing values, cross-validation, device updates, and versatility, we chose XGBoost. We'll go into each of the following points in detail below:

Overfitting

Tree Pruning — Pruning is a machine learning strategy for reducing the size of regression trees by removing nodes that don't lead to better leaf classification. The aim of pruning a regression tree is to prevent the training data from being overfit. Cost Complexity or Weakest Link Pruning, which uses mean square error, k-fold cross-validation and learning rate, is the most effective pruning process. XGBoost generates nodes (also known as splits) up to the given max depth and then begins pruning from the backward direction until the loss is below a threshold. In the case of a split with a -3 loss and a corresponding node with a +7 loss, XGBoost can not delete the split simply by looking at one of the negative losses. It will calculate the total loss (-3 + 7 = +4) and hold both if the result is positive.

Missing Values

Sparsity Aware Split Finding — It's not uncommon for the data collected to be sparse (with a lot of missing or empty values) or to become sparse after data engineering (feature encoding). Each tree is given a default direction in order to be aware of the sparsity trends in the data. Missing data is handled by XGBoost by assigning them to the default path and finding the best imputation value to reduce training loss. The optimization here is to only visit missing values, which makes the algorithm 50 times faster than the naive method.

Cross-validation
Built-in Cross-validation — Cross validation is a statistical tool for evaluating machine learning models on data that hasn't been seen before. It prevents overfitting by not taking an independent sample (holdout) from training data for validation when the dataset is small. By reducing the size of the training data, we are jeopardizing the hidden features and trends in the data, potentially leading to further errors in our model. This is close to the scikit-learn library's cross val score feature.

System Enhancements
Parallelization — Tree learning requires sorted data, which requires parallelization. Data is separated into compressed blocks to reduce sorting costs (each column with corresponding feature value). XGBoost sorts each block in parallel, using all CPU cores/threads. Since a large number of nodes are generated regularly in a tree, this optimization is advantageous. XGBoost, in a nutshell, parallelizes the sequential mechanism of tree generation.

Cache Aware — We store gradient statistics (direction/value) for each split node in internal buffers for each thread and accumulate in a mini-batch manner using cache-aware optimization. This reduces the overhead of immediate read/write operations and eliminates cache misses. Cache knowledge is achieved by selecting optimal cache block size (e.g. 2^{16}).

Flexibility
Customized Objective Function — The aim of an objective function is to maximize or decrease the value of something. In machine learning, we aim to minimize the objective function, which is made up of the loss function and the regularization concept.

$$\mathcal{L}(\phi) = \underbrace{\sum_i l(\hat{y}_i, y_i)}_{\text{Loss function}} + \underbrace{\sum_k \Omega(f_k)}_{\text{Regularization}}$$

L(Φ) is objective function

Optimizing the loss function facilitates predictive models, while optimizing regularization reduces uncertainty and increases predictability. The following are some of the objective functions available in XGBoost:

- reg: linear for regression
- reg: logistic, and binary: logistic for binary classification
- Multi: softmax, and multi: softprob for multiclass classification

Customized Evaluation Metric — This is a metric that is used to monitor the model's accuracy when it comes to validation results. The following are the metrics:

- rmse — Root mean squared error (Regression)
- mae — Mean absolute error (Regression)
- error — Binary classification error (Classification)
- logloss — Negative log-likelihood (Classification)
- auc — Area under the curve (Classification)

2.4.2 Neural Networks

Neural networks are a subset of machine learning and form the basis of deep learning algorithms. Each neural network consists of layers of “neurons”, inspired by the human brain, where neurons function as follows: the neurons take in input vectors, multiply them by an internally stored *weight* vector, add an internally stored *bias* vector, and then output the result. Neurons are organized in *layers*, such that neurons in adjacent layers are connected. This means that the outputs of a prior layer are the inputs of a latter layer. Additional *activation* functions can be used to alter the raw output of neurons in order to improve performance (learning).

If the outputs of a layer connect to the inputs of the next layer in only one direction, this network is a *feed forward network*. Often, the last layer contains one neuron with a softmax activation for classification tasks, or one neuron with linear activation for linear regression tasks. In the case of this project, a linear activated neuron is used to predict taxi trip duration.

The passing of inputs from prior layers to latter layers is called *forward propagation* - which results in a prediction of a target based on its inputs. The network is able to learn to make good predictions by refining the weights and biases. The weights and biases are updated in the process of *backpropagation*. Essentially, for each layer and for each neuron, the weight and bias vectors of each neuron are modified by the product of the partial derivative of the activation function and the error vector and the neuron inputs, where the error vector is the difference between the predicted result (previous output of the neuron) and the actual result. The derivations are performed layer by layer, in the opposite direction to the forward pass.

Neural Network Architecture

A standard feed forward network architecture is used in this project. Convolutional Neural Networks (CNN) are not used because the processed features do not have any geometric relationship, and Recurrent Neural Networks (RNN) are not used because the processed features do not have a time dependence. All neuron layers are dense (fully connected), that is, for all layers, each neuron receives the output of each neuron in the previous layer as input.

Batch normalization (BN) layers are introduced to improve the stability of training, to allow the neural network to converge more quickly. Training data is split into batches, and for each batch of inputs to each BN layer, the inputs are normalized with respect to their batch in both batch size and input size dimensions.

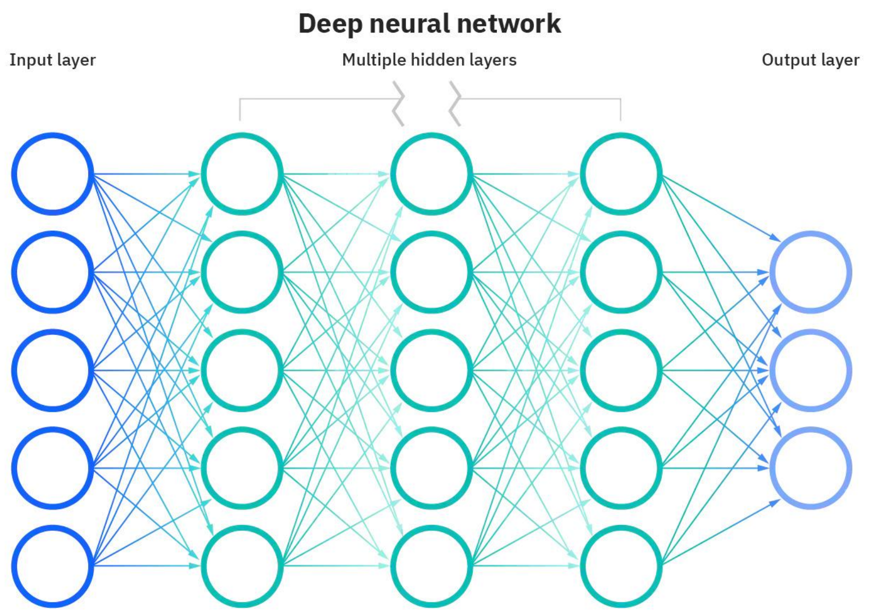


Figure: Our General Neural Network Architecture with Fully Connected Hidden Layers.

Optimizers

While training a neural network, we aim to minimize the difference between the predicted and actual outputs, which is computed using a cost function such as Mean Square Error (MSE) and Root Mean Square Error (RMSE). The cost functions are generally convex functions – functions that can have a global minimum. Thus, to minimize the cost function, the neural network must find the optimal values for the weights/biases in the network. In order to find the optimal weights for a neural network (weights that minimize cost), an optimization algorithm known as an optimizer, is used which iterates through the dataset multiple times.

The optimizers used for the neural networks in this project are Stochastic Gradient Descent (SGD), Momentum, RMSProp, and Adam.

Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD) is a variant of the classic optimizer called Gradient Descent (or Batch GD). SGD uses a single training example at a time to calculate gradients during backpropagation (partial derivatives of the loss for one training example with respect to each weight or bias in the network) and updates the weights once for each example in the training set. In one complete run through the dataset (one epoch), SGD allows for multiple updates to the neural network's weights and biases.

SGD allows for very fast learning (compared to Batch GD) as weights can be updated multiple times per epoch (leading to faster convergence). However, the cost function fluctuates heavily when using SGD as gradients are computed for only one training example. Thus, with SGD it is more difficult to reach the global minimum of the cost function.

In our code, Mini-Batch Gradient Descent is actually used as an optimizer, though it is called SGD in TensorFlow. Mini-Batch GD uses small batches of training examples at each step in the epoch and allows for decently fast learning and convergence, and the cost function does not fluctuate as much as with SGD, so it is better. However, more modern optimizers such as Momentum, RMSProp, and Adam are even better.

Momentum and RMSProp

Momentum and RMSProp are optimizers that build upon SGD. The Momentum optimizer, also known as SGD with Momentum, is based on the idea of exponentially weighted moving averages – an idea similar to the physics concept of momentum. For instance, if we consider a ball rolling down a hill, it can be seen that the ball gains momentum as it rolls down the hill. Similarly, Momentum allows the SGD optimizer to more effectively move in the direction that points towards the global minimum of the cost function by reducing deviations in other (perpendicular) directions.

Effectively, Momentum helps accelerate SGD when the Gradient Descent curve is steeper in one direction than in another. This dampens random oscillations in the GD curve. When updating weights, momentum takes the gradient of the current step as well as the gradient of the previous time steps to determine the update of the weights and biases in the neural network. This helps the model move faster towards convergence.

On the other hand, we have RMSProp, which stands for Root Mean Square Propagation. It was originally proposed by renowned computer scientist and Deep Learning researcher Geoffrey Hinton. Similar to Momentum, RMSProp uses the concept of exponentially weighted moving averages, though it instead uses the squared gradients to compute the new weights and biases. With RMSProp, the learning rate is adjusted automatically for each parameter as RMSProp divides the learning rate by the average of the exponential decay of squared gradients.

SGD

$$\theta_j \leftarrow \theta_j - \epsilon \nabla_{\theta_j} \mathcal{L}(\theta)$$

SGD with Momentum

$$v_{t+1} \leftarrow \rho v_t + \nabla_\theta \mathcal{L}(\theta)$$
$$\theta_j \leftarrow \theta_j - \epsilon v_{t+1}$$

RMSProp

$$v_{dw} = \beta \cdot v_{dw} + (1-\beta) \cdot dw^2$$
$$v_{db} = \beta \cdot v_{dw} + (1-\beta) \cdot db^2$$
$$W = W - \alpha \cdot \frac{dw}{\sqrt{v_{dw}} + \epsilon}$$
$$b = b - \alpha \cdot \frac{db}{\sqrt{v_{db}} + \epsilon}$$

Figure: Update Rules for SGD, Momentum, and RMSProp

Adam (Adaptive Moment Estimation)

The most heavily used optimizer in this project is Adam, which stands for "Adaptive Moment Estimation". Adam is effectively a combination of Momentum and RMSProp as it calculates the individual learning rate for each parameter (adaptive learning rate) from estimates of the first and second moments of the gradients. Here, the first and second moments refer to the exponentially weighted moving averages of the gradient (momentum) and the squared gradient (velocity).

Adam is computationally efficient and has a low memory requirement, which is why it is one of the most popular gradient descent optimization algorithms today. Also, Adam usually allows neural networks to converge faster than with other optimizers such as SGD.

Learning Rate

Learning rate is a hyperparameter that determines how much the weights of a neural network are updated with respect to the loss function. Lower learning rates result in smaller changes to the neural network's weights during each gradient descent update, resulting in a slow movement along the downward slope of the gradient descent curve for the cost function (and vice versa).

Using a low learning rate increases the likelihood that a neural network converges at local or global minima, allowing the model to attain a relatively low training loss. However, it might result in slow training, i.e., the model taking a relatively long time to converge. On the other hand, using a reasonably large learning rate can speed up convergence. However, if the learning rate is too large, the model may be unable to find a minimum for the loss function, or possibly result in divergence – the loss may increase uncontrollably.

How good a learning rate is depends on the optimizer used. For instance, SGD works well using learning rates such as 0.01, though Adam optimizer requires lower learning rates such as 0.001 (1e-3 or lower). Thus, when choosing optimizers for our neural network experiments, we had to carefully consider the base learning rate (and tune learning rate in later experiments).

Regularization

In Machine/Deep Learning tasks such as regression, we aim to minimize loss (MSE/RMSE) on our training set while also being able to generalize well to new data as indicated by test set loss. However, achieving this objective is difficult due to the tendency of complex models to overfit to the training set, resulting in poor performance on the validation and test sets. A model that overfits to the training set can also be considered to have high variance, and it is the opposite of a model that underfits and has high bias. Regularization is one well-proven method to counter overfitting, and it allows a model to have a better balance of bias and variance.

For our neural networks, two types of regularization were tested: L2 Regularization and Dropout Regularization. The following is a brief overview of these two types of regularization.

L2 Regularization

L2 regularization owes its name to the L2 norm of a vector, also known as the Euclidean norm. In the context of a neural network, L2 regularization adds the squared magnitude of the weights in a neural network as a "penalty" term to a loss function. The amount of L2 regularization applied in a model is influenced by a constant called the L2 regularization factor. L2 regularization causes weights in the neural network to take smaller values, causing each weight to have a smaller impact on the neural network's predictions – this is called weight decay.

In this project, L2 regularization was applied on the weights and biases in the neural networks in the form of TensorFlow's L2 kernel and bias regularizers for the dense (fully connected) layers.

Dropout Regularization

"Dropout" in deep learning refers to the process of randomly ignoring certain neurons in a layer during training, i.e., the weights/biases of "dropped" neurons are not considered during training. Dropout is commonly used as a regularization technique as it prevents overfitting by ensuring that neurons in the network are not codependent. This is because learning weights together during training results in some of the neurons having more "predictive capability" than the others, which can lead to overfitting.

Applying dropout regularization to a neural network effectively creates a "thinner" network which has unique combinations of hidden layer neurons being kept and dropped randomly (at different points in time during each epoch of training). The percentage of hidden layer neurons to drop at any point in time is determined by the drop rate, and this is an important hyperparameter to determine the degree of regularization applied on the model.

In this project, we experimented with dropout regularization (specifically, the drop rate and number of dropout layers) by applying Dropout layers after some of the Dense layers.

3. Experiments and Results

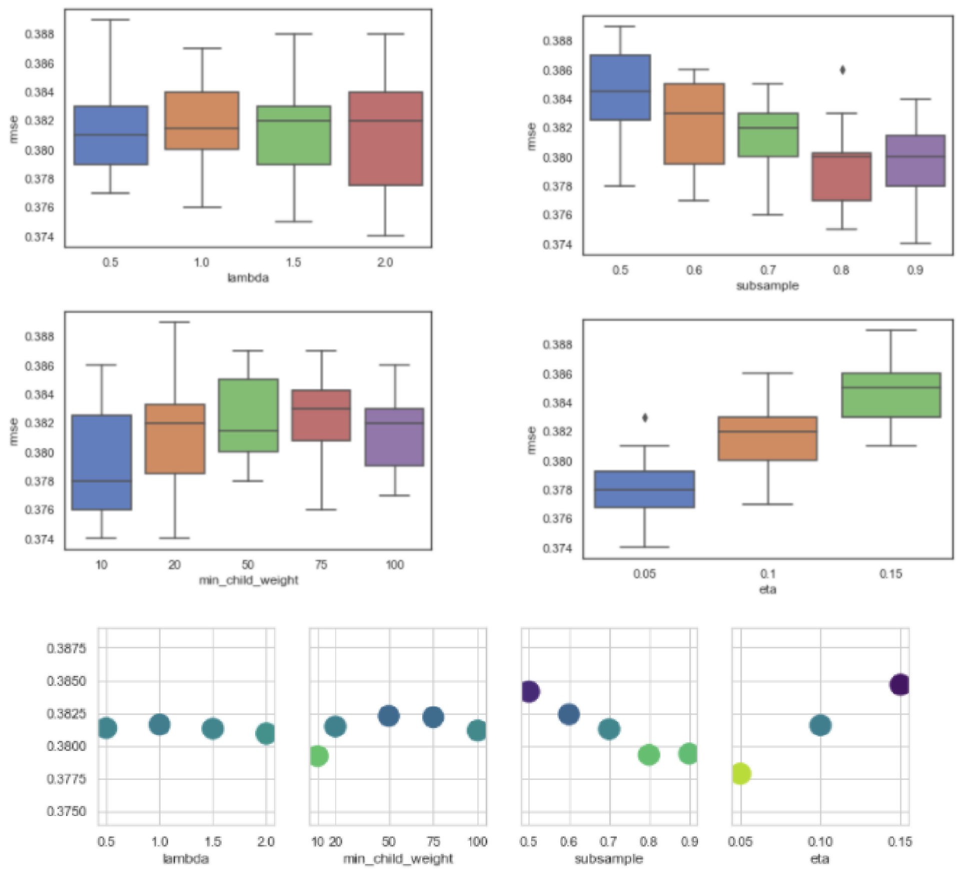
After implementing our XGBoost and Neural Network models, we needed to tune the models in order to maximize performance (minimize RMSE loss). Thus, the main focus of our experiments was to improve model performance, which involved tuning of various hyperparameters.

3.1 XGBoost

For hyperparameter tuning, we used random search. We tried different values for the following hyperparameters:

- min_child_weight
- eta
- colsample_bytree
- max_depth -subsample
- Lambda

Results of hyper parameter optimization visualized:



We can see that lower eta could improve the performance significantly. Higher subsample and smaller child limit leads to better results too. Lambda does not really matter.

Results from our final model:

Train RMSE	0.13582
Validation RMSE	0.36754

RMSE for train and validation set with final XGBoost model

The Kaggle public leaderboard score with our final XGBoost model was 0.37299 (screenshot is shown in appendix).

3.2 Neural Networks

For hyperparameter tuning, we initially tried to use an extensive grid search, although it was too time-consuming due to the long training time required for neural networks. Eventually, we resorted to using a random search to conserve time and effort. The following is the list of hyperparameters we tuned:

- Number of Dense / Fully Connected Layers (network depth - between 4 and 64 layers).
- Number of Neurons per Layer (network breadth - between 16 and 512 neurons per layer)
- Optimizers (Adam vs. SGD vs. Momentum vs. RMSProp)
- Regularization (None vs. L2 vs. Dropout)
- Learning Rate

For each combination of hyperparameters, the model was trained for 100 epochs. It was noted that the neural network was overfitting to the training data relatively quickly, and so training for more epochs was usually not necessary. Initially we also tested with different batch sizes (16, 32, 64, 128, 256), although the batch size was later fixed at 128 as the dataset is relatively large, and so training is slightly faster, albeit with slightly fewer gradient descent updates (steps) per epoch.
Please note that cross-validation (K-Fold cross-validation) has not been utilized because neural networks are highly time-consuming. Instead, 20% of the original training data was used for validation, and 80% for actual training. Moreover, note that the train and validation RMSE values in this section may be difficult to reproduce because of the random nature of the operations within the neural network (random weight initialization, random shuffling, etc.).

Some of the most important combinations of hyperparameters that we tested have been displayed in the table on the next page (not all combinations are shown as we tested for a large number of combinations of hyperparameters, and showing all combinations would be impractical).

Hyperparameters	**Train RMSE**	**Validation RMSE**	**Leaderboard Score (Public)**
16 layers, 256 neurons per layer, SGD, no regularization, learning rate = 1e-2	0.43314	0.43201	0.47360
16 layers, 256 neurons per layer, Adam, no regularization, learning rate = 1e-3	0.42165	0.42784	0.46209
16 layers, 256 neurons per layer, Adam, L2 reg. factor = 1e-5, learning rate = 1e-3	0.41398	0.41239	0.43442
16 layers, 256 neurons per layer, Adam, Dropout w/drop-rate = 0.2, learning rate = 1e-3	0.41887	0.43542	0.45930
32 layers, 256 neurons per layer, Adam, L2 reg. factor = 1e-5, learning rate = 1e-3	0.40480	0.39246	0.41522
40 layers, 400 neurons per layer, Adam, L2 reg. factor = 5e-5, learning rate = 1e-3	**0.39921**	**0.38803**	**0.39963**

Table of Results for Neural Networks with Various Combinations of Hyperparameters

Our best neural network model had the hyperparameters in the row above that is highlighted in green. Our best (final) model obtained a public leaderboard score of 0.39963 (RMSLE for test set). Screenshots showing the Kaggle public leaderboard score using the best (final) neural network model are provided in the appendix (after XGBoost).

3.3 Analysis

From the experiments, it is clear that the final XGBoost model performed better than the final neural network model. This might be due to a few reasons. One reason might be that XGBoost is better suited to the NYC dataset, even after using external OSRM data and data processing. It is a common occurrence for one model to perform better than a different type of model. Secondly, it might be possible that with further hyperparameter tuning, the neural network could outperform XGBoost. However, such extensive tuning was not possible due to computational and time constraints, and it might be more suitable for the future. Nevertheless, we appreciate the success of XGBoost, and its performance might explain why it is such a popular model.

One reason why the models did not perform better could be that the trip duration predictions may be affected by factors such as weather that were not accounted for in the training of the models. For example, trip duration may be longer during rainy weather where drivers would have to reduce their speed while driving. Hence, prediction of trip durations were not as accurate. Using the NYC Weather Data on Kaggle could help improve our results, though this was not used for our project due to time and computational constraints.

4. Conclusion

To sum up, we have explored various types of Exploratory Data Analysis (EDA) to better understand the trends and patterns of the data. We applied outlier detection using the Interquartile range and Z-score approach, and performed feature engineering and selection on our dataset. For prediction, we attempted to experiment with two types of models: XGBoost and Neural Networks. Through our experiments and analysis of each model, we have concluded that the XGBoost approach outperformed the Neural Network approach.

We would also like to mention the challenges we faced when developing a solution for the problem statement, which are explained in the following list:

1. Like all real-world systems, the data was quite noisy and contained both outright errors and non-standard behaviour. Therefore, we had to discover ways to filter out bad data so that the prediction would be more accurate.

2. The trip duration may have been affected by various factors such as the time of the day, speed, and distance between the pickup and dropoff locations. Therefore, we had to explore the factors and create new features to account for these factors.

3. The NYC data had relatively few features per training example, so external data was required. For this we used the NYC-OSRM dataset from Kaggle, though processing this dataset and merging with the original data was challenging.

4. Training Machine Learning and Deep Learning models was computationally expensive (needed a powerful computer) and was time-consuming, which was exacerbated due to the large size of the dataset (nearly 1.4 million records in the training data). Therefore, we had to use cloud services for computation, such as Google Colaboratory and Kaggle GPUs.

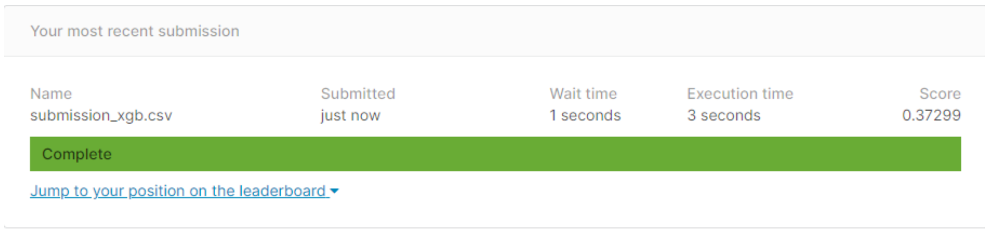
5. The division of work was difficult due to dependencies between different aspects of our project. For instance, training models requires the completion of feature engineering and selection. Therefore, we attempted to collaborate as much as possible on the code by sharing code on a GitHub repository and using Google Colaboratory.

For possible improvements in the future, firstly, we can explore different methods for outlier removal and data cleaning. The inclusion or exclusion of outliers can have a significant effect on the model's performance, and hence we can explore other methods that would improve the model's performance. Next, we can construct more features and apply it on the existing features to find a more optimal set that is used to train the model. Finally, we can experiment with other complex models such as other gradient boosting methods to see if they perform better.

5. Appendix

5.1 Kaggle Submissions

Kaggle Scores for final XGBoost model:

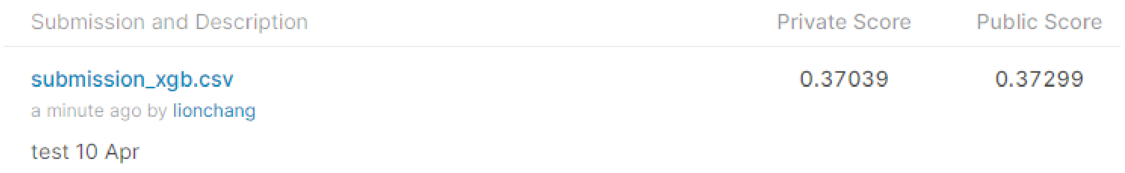


Your most recent submission

Name	Submitted	Wait time	Execution time	Score
submission_xgb.csv	just now	1 seconds	3 seconds	0.37299

Complete

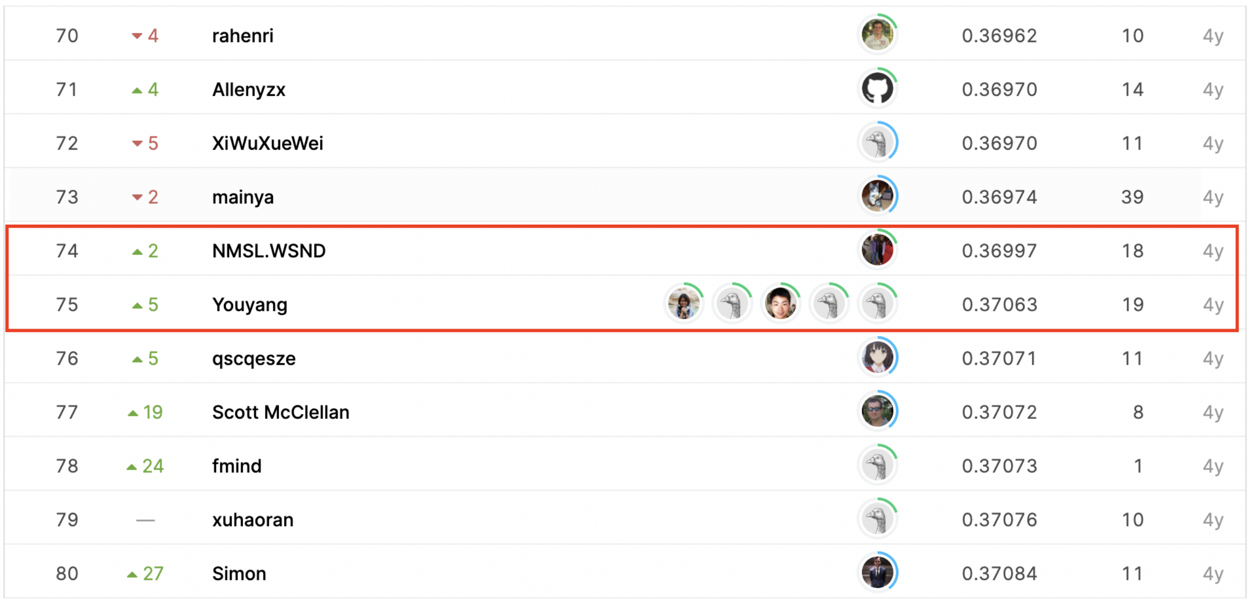
Jump to your position on the leaderboard



Submission and Description	Private Score	Public Score
submission_xgb.csv a minute ago by lionchang test 10 Apr	0.37039	0.37299

Private Leaderboard

Score: 0.37039

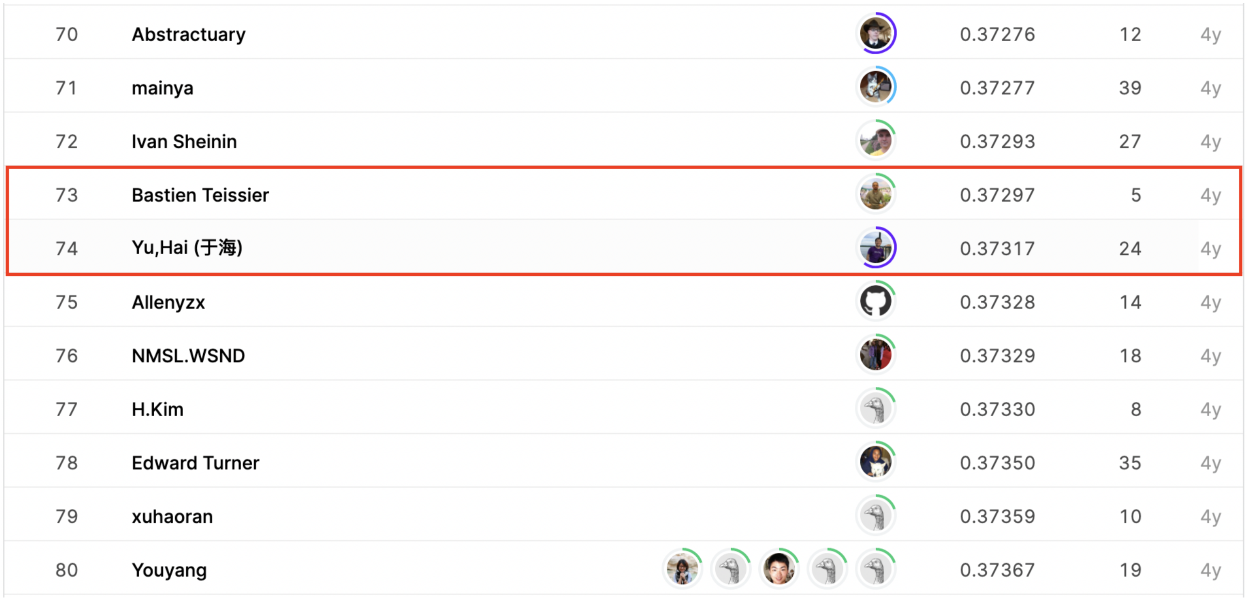


70	▼ 4	rahenri		0.36962	10	4y
71	▲ 4	Allenyzx		0.36970	14	4y
72	▼ 5	XiWuXueWei		0.36970	11	4y
73	▼ 2	mainya		0.36974	39	4y
74	▲ 2	NMSL.WSND		0.36997	18	4y
75	▲ 5	Youyang		0.37063	19	4y
76	▲ 5	qscqesze		0.37071	11	4y
77	▲ 19	Scott McClellan		0.37072	8	4y
78	▲ 24	fmind		0.37073	1	4y
79	—	xuhaoran		0.37076	10	4y
80	▲ 27	Simon		0.37084	11	4y

Rank = 75th

Rank Placement = ~Top 6% (75/1254 = 5.98%)

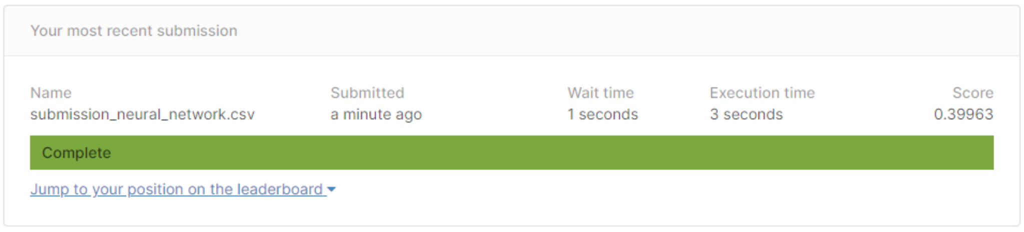
Public Leaderboard
Score: 0.37299



70	Abstractuary	0.37276	12	4y
71	mainya	0.37277	39	4y
72	Ivan Sheinin	0.37293	27	4y
73	Bastien Teissier	0.37297	5	4y
74	Yu,Hai (于海)	0.37317	24	4y
75	Allenyzx	0.37328	14	4y
76	NMSL.WSND	0.37329	18	4y
77	H.Kim	0.37330	8	4y
78	Edward Turner	0.37350	35	4y
79	xuhaoran	0.37359	10	4y
80	Youyang	0.37367	19	4y

Rank = 74th
Rank Placement = ~Top 6% (74/1252 = 5.91)

Kaggle Scores for Final Neural Network model:

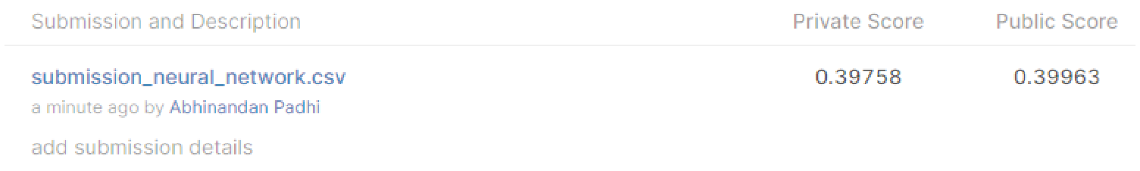


Your most recent submission

Name	Submitted	Wait time	Execution time	Score
submission_neural_network.csv	a minute ago	1 seconds	3 seconds	0.39963

Complete

Jump to your position on the leaderboard ▾



Submission and Description	Private Score	Public Score
submission_neural_network.csv a minute ago by Abhinandan Padhi add submission details	0.39758	0.39963

Since the kaggle leaderboard scores using the neural network model are not as good as the scores for the XGBoost model, we considered the XGBoost model as our final model. Thus, screenshots of the leaderboard and the rank when using the neural network model are not shown.

5.2 Contribution breakdown

Name	**Contribution**
Chang Heen Sunn	Code, Report, Presentation (20%)
Hasan Mohammad Yusuf	Code, Report, Presentation (20%)
Ng Man Chun	Code, Report, Presentation (20%)
Padhi Abhinandan	Code, Report, Presentation (20%)
Wong Zhen Yan	Code, Report, Presentation (20%)

5.3 References

1. New York City Taxi Trip Duration Dataset: https://www.kaggle.com/c/nyc-taxi-trip-duration/
2. New York City Taxi Trip with OSRM Dataset: https://www.kaggle.com/oscarleo/new-york-city-taxi-with-osrm
3. XGBoost Documentation: https://xgboost.readthedocs.io/en/latest/
4. TensorFlow & Keras Documentation: https://www.tensorflow.org/api_docs/
5. XGBoost Figure: https://medium.com/sfu-cspmp/xgboost-a-deep-dive-into-boosting-f06c9c41349
6. Neural Networks Figure: https://www.ibm.com/cloud/learn/neural-networks