CSE 151B final Project Report

Yifei Wang
yiw085@ucsd.edu
Wenzhou Lyu
wlyu@ucsd.edu
Yunlong Wang
yuw151@ucsd.edu
Chenyang Zhou
chz057@ucsd.edu
Department of Computer Science
University of California San Diego
Github Link: Link
Drive Link: Link

1 Task Description and Exploratory Analysis

1.1 Problem A

1.1.1 Deep Learning and Its Importance

Deep learning is a subset of machine learning that utilizes neural networks to train a model to solve complex problems with huge amounts of data. With improvement in computational power, we are more capable of dealing with large datasets and extracting features from them. For traditional machine learning, we need to select appropriate models for different datasets to obtain a low loss, but for deep learning, we may be able to train the model and extract features from it. Thus, we do not need a specific algorithm in order to train it.

1.1.2 Real World Application

We may use a convolutional neural network (CNN) in deep learning for image processing and voice recognition. In fact, the image recognition algorithm has been adopted in auto driving industries as well as security departments. The recurrent neural network (RNN) algorithm may be used for text generation and voice recognition. We have already seen some very basic text generation algorithms using RNN as their model several years ago. However, more advanced algorithms like transformer is the state-of-the-art model that is used to train natural language processing including chatGPT.

1.2 Problem B

For all of the loss functions below, loss $L(a,b) = \sum_{i=1}^{N}(a_i - b_i)^2$ for mse, and $L(a,b) = \sum_{i=1}^{N}\sqrt{(a_i - b_i)^2}$ for mse

1.2.1 LightGBM

lightGBM model is an open-source framework developed by Microsoft that utilizes boosting algorithm for training. LightGBM combines multiple weak learners such as decision trees to obtain a better model. We input a training dataset, and it will perform boosting.

In general, for boosting the classification algorithm, given training data of $[m \times n]$ dimension, and a set of training data $x_1...x_m$,and corresponding true label $y_1..y_m$, we first create an initial weight

Preprint. Under review.

across all data points called w. By determining some decision boundaries using a set of weak learners $h_1...h_k$, including decision trees or KNN, we may determine the error rate for each learner and adjust the weight of the data points accordingly where error rate $\epsilon = \sum_i^m w_i * (y_i \neq h_j(x_i))$ for jth learner.

Then, we adjust the weight based on the error rate where $\alpha = \frac{1}{2} ln(\frac{1-\epsilon}{\epsilon})$ and $w_{new} =$

$$\begin{cases} w * e^{-\alpha} & \text{if } y_i = h_j(x_i) \\ w * e^{\alpha} & \text{if } y_i \neq h_j(x_i) \end{cases} \tag{1}$$

Finally, we may generate a decision boundary that combines multiple weak ensembles, which is essentially the idea of boosting.

We have used both rmse and mse for the loss function.

1.2.2 Linear Regression

We have used linear regression as our second model. We fitted the training data and then validate it. The predicted value $\hat{y} = f(x|w, b) = w^T x - b$ We want to choose the optimal weight and bias value such that we can minimize the loss $argmin_{w,b} \sum_{i=1}^N L(y_i, \hat{y_i})$

1.2.3 Random Forest

Random Forest is our third model. We use the RandomForestRegressor in sklearn to fit out training data. Using a decision tree as the underlying model, the decision tree train on a subset of data at each node, and generate a forest at a specified depth.

1.2.4 Bayesian Ridge

Bayesian ridge utilizes the Naive Bayes theorem $P(y|x) = P(x)P(x|y) \cdot P(y)$. There can be multiple events in the chain, and finally, we can get the $p(y|x_1, ..., xn)$ given n data points and the actual value.

Therefore, by applying this theorem, we may get the posterior distribution $\mu = \alpha \Sigma X^T y$ where μ is the mean vector of the posterior distribution. α is the precision parameter for the error term. Σ is the covariance matrix of the posterior distribution. X is the matrix of input features. y is the target variable vector. It is notable that the covariance matrix Σ can be computed using $\Sigma = (\lambda I + \alpha X^T X)^{-1}$.

Finally, we may be able to estimate the value of a new set of input features by computing the dot product between X and μ

1.2.5 MLP

Given the taxiId, year, month, day, hour, week, callType, and dayType, we map this input vector into a 128-dim hidden layer using a linear model.

$y = f(x|w, b) = w^T x - b$
where x is [54, 1], w is [128, 54] and y is [128, 1]

We then map this hidden layer into another 64-dim hidden layer using a linear model.

$y = f(x|w, b) = w^T x - b$
where x is [128, 1], w is [64, 128] and y is [128, 1]

We then map this hidden layer into another 32-dim hidden layer using a linear model.

$y = f(x|w, b) = w^T x - b$
where x is [64, 1], w is [32, 64] and y is [32, 1]

Finally, we obtain the output using the same linear model $y = f(x|w, b) = w^T x - b$
where x is [32, 1], w is [1, 32] and y is [1, 1]

2 Exploration Data Analysis

2.1 Problem A

For the provided dataset, the training data size is 1710670, and the test data size is 320. However, this is the data size before we perform any trim operation for outliers. For the raw data, the mean value for the trip length is 716.426, the median is 600.0, with the standard deviation of 684.751. As we notice the huge difference between the mean and the median, we speculate that the distribution of the trip length to be skewed, so we get the following distribution plot of trip length for the raw data.

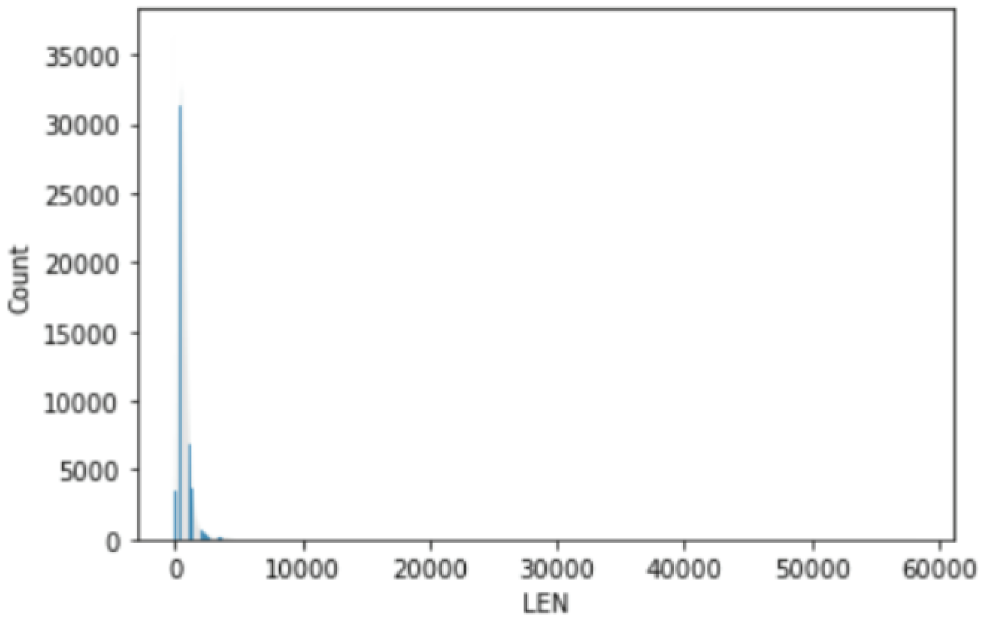


Figure 2.1 trip duration distribution of raw dataset

As we expected, outliers exist so what we first did is to trim the dataset and remove outliers that are 3 standard deviations from the mean, and the training size goes from 1710670 to 1692771, with the following distribution graph.

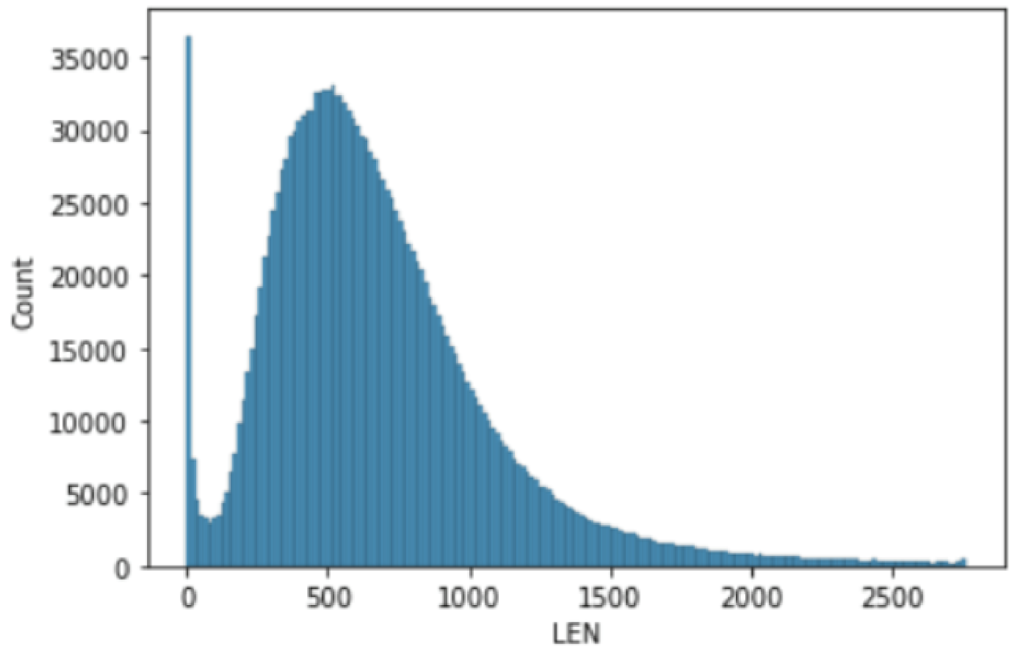


Figure 2.2 trip duration distribution of raw dataset after removing outliers 3 standard deviation from mean

Now, we can see the improvement on the overall distribution of trip duration on the dataset. However, notice there are still a fair number of outliers on the left hand side of the graph. Therefore we decide to remove all the trips that have trip length shorter than 60 seconds, and we get the following, shrinking the training size down to 1637270.

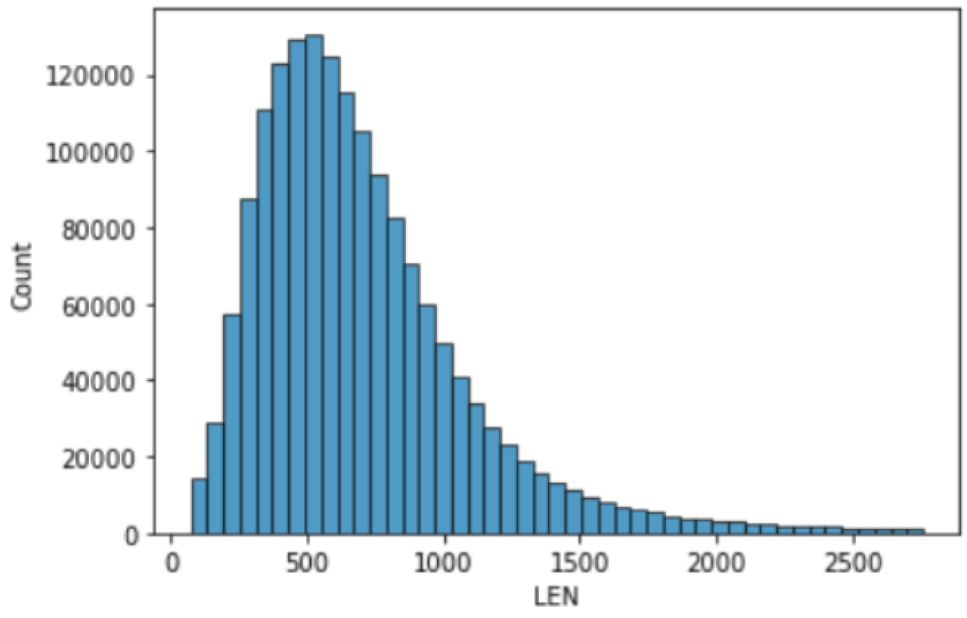
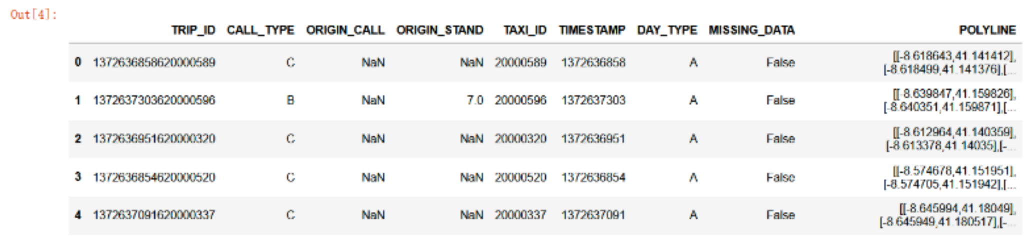


Figure 2.3 trip duration distribution of raw dataset after removing outliers 3 standard deviation from mean and ones smaller than 60

Respectively, the dimension of the inputs is 8, and the dimension of the output is 1, which is the predicted trip length. If we take a look at the raw data set in the following table, we can see that in fact, there are 9 columns in total, and each row represents an individual trip. However, the dimension of the inputs is 8, because the last column POLYLINE, where each entry is a list of pairs, represents the locations recorded for that trip in the form (longitude, latitude) every 15 seconds. Since the trip time is what we are predicting, we can calculate the trip time from POLYLINE and use that as labels (Y value). Therefore, the dimension input is 8 instead of 9, as we extract POLYLINE column from it.


Out[4]:

	TRIP_ID	CALL_TYPE	ORIGIN_CALL	ORIGIN_STAND	TAXI_ID	TIMESTAMP	DAY_TYPE	MISSING_DATA	POLYLINE
0	1372636858620000589	C	NaN	NaN	20000589	1372636858	A	False	[[-8.618643,41.141412], [-8.618499,41.141376],[...
1	1372637303620000596	B	NaN	7.0	20000596	1372637303	A	False	[[-8.639847,41.159826], [-8.640351,41.159871],[...
2	1372636951620000320	C	NaN	NaN	20000320	1372636951	A	False	[[-8.612964,41.140359], [-8.613378,41.14035],[-...
3	1372636854620000520	C	NaN	NaN	20000520	1372636854	A	False	[[-8.574678,41.151951], [-8.574705,41.151942],[...
4	1372637091620000337	C	NaN	NaN	20000337	1372637091	A	False	[[-8.645994,41.18049], [-8.645949,41.180517],[-...

Figure 2.2 raw data frame before data pre-processing

Since the output dimension is 1, the meaning of the only dimension is the trip length, the value we are supposed to predict. For those 8 input dimensions, they have different meanings. First of all, TRIP_ID is a unique 19-digit ID for each trip. It mainly consists of TIMESTAMP concatenated with TAXI_ID. CALL_TYPE are either A, B, or C, where A is for trips dispatched from the central by making a phone call, B is for trips started at specific stands, and C is for the others.
ORIGIN_CALL will be the customer's phone number if CALL_TYPE is A, and null otherwise. ORIGIN_STAND will be a unique identifier for 63 different stands locations if CALL_TYPE is B, and null otherwise. TAXI_ID is simply a unique identifier for each taxi. TIMESTAMP gives information about when the trip happens. DAY_TYPE are also in letters A, B, or C, where A is a normal day, B is for holiday, and C is the day before a B-type day. MISSING_DATA is True when the POLYLINE gps location is completely recorded, and False otherwise.

To visualize a data sample, we use folium package to see the route of the trip. We choose the first trip (index 0) in the dataset, and use its POLYLINE value to get the following route. The POLYLINE for this trip is "[[-8.618643,41.141412], [-8.618499,41.141376], [-8.620326,41.14251], [-8.622153,41.143815], [-8.623953,41.144373], [-8.62668,41.144778], [-8.627373,41.144697], [-8.630226,41.14521], [-8.632746,41.14692], [-8.631738,41.148225], [-8.629938,41.150385], [-8.62911,41.151213], [-8.629128,41.15124], [-8.628786,41.152203], [-8.628687,41.152374], [-8.628759,41.152518], [-8.630838,41.15268], [-8.632323,41.153022], [-8.631144,41.154489], [-8.630829,41.154507], [-8.630829,41.154516], [-8.630829,41.154498], [-8.630838,41.154489]]", which are 23 location points in total.

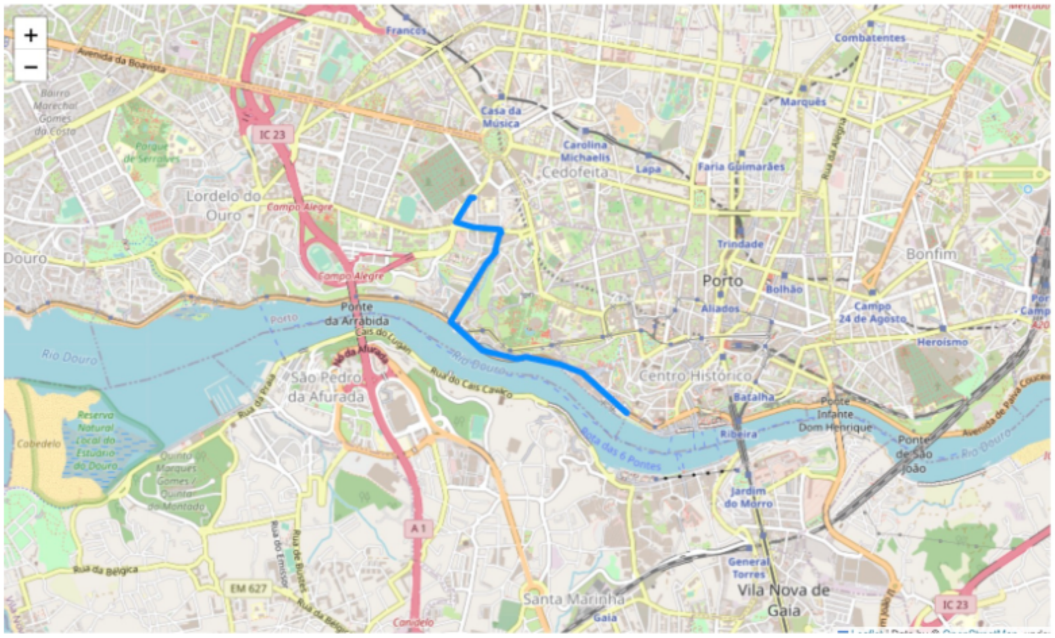


Figure 2.4 visualization of one trip in the dataset

If we want to visualize all the trips on the map, adding frequency for the locations, we get the following heatmap.

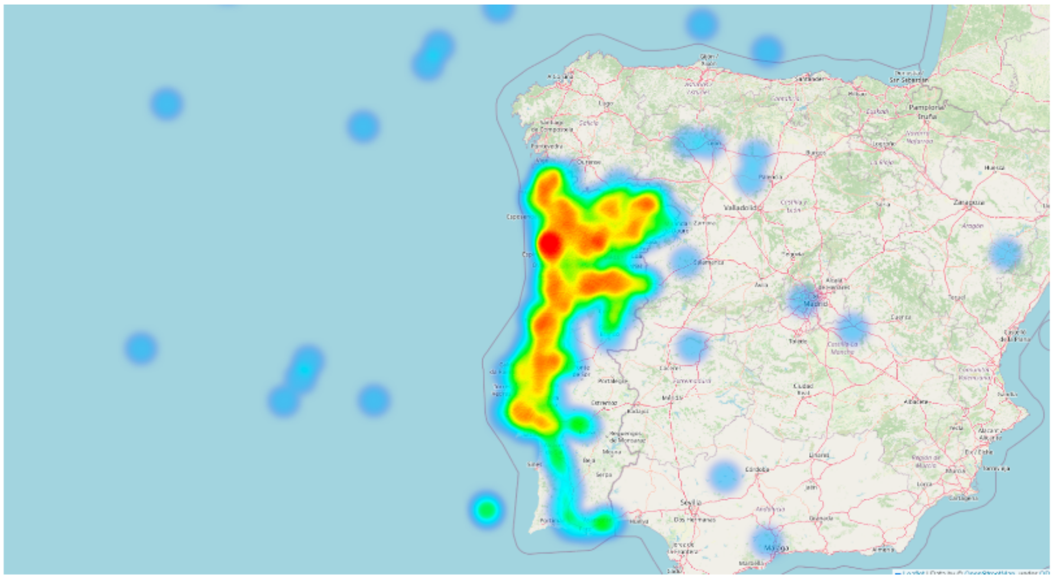


Figure 2.5 visualization of all trip routes on the map

Notice how the trips almost span the entire region of Portugal, and the red-most region in the map is where Porto is, which totally makes sense for the dataset.

Also, we can also visualize the 63 specific stands in Porto with frequencies added to see where some popular stands for cab services are.

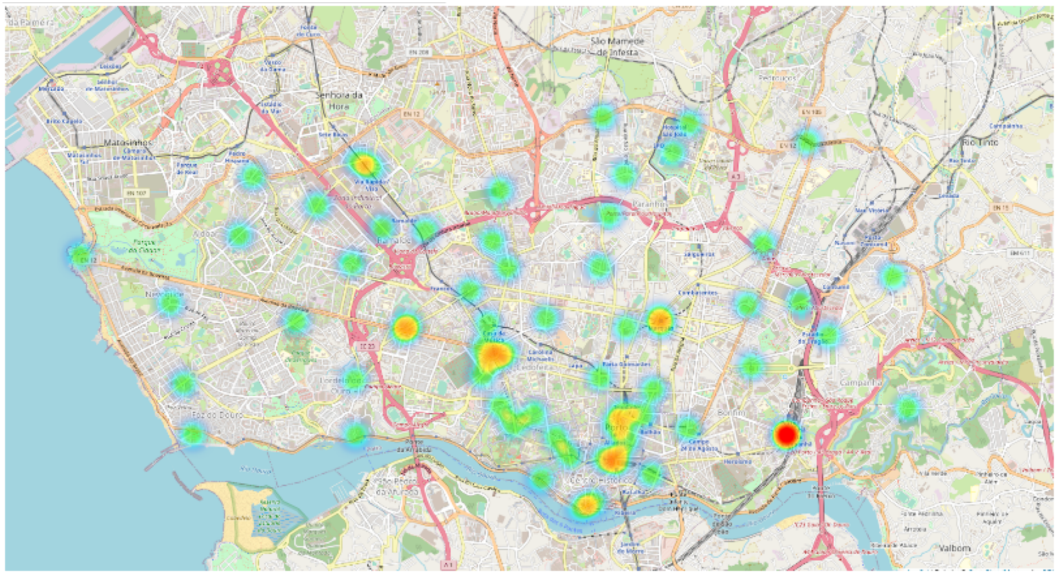


Figure 2.6 visualization of all 63 specific stands on the map

As we can see, the red-color regions indicate a higher frequency of a trip. It seems like Noeda (the region in the southeast corner) has the most amount of taxi business.

2.2 Problem B

We use the train_test_split function from sklearn.model_selection, and we choose 25% of the dataset to the validation set randomly from the training dataset, and the rest 75% will be our actual training set. Since the size of the training dataset after removing the outliers is 1637270. The size of the train dataset is 1227952 and the size of the validation dataset is 409318.

For feature engineering, what we first did is to take a look at how the three call types are distributed and calculate the trip lengths after grouping them by call types, we get the following graphs.

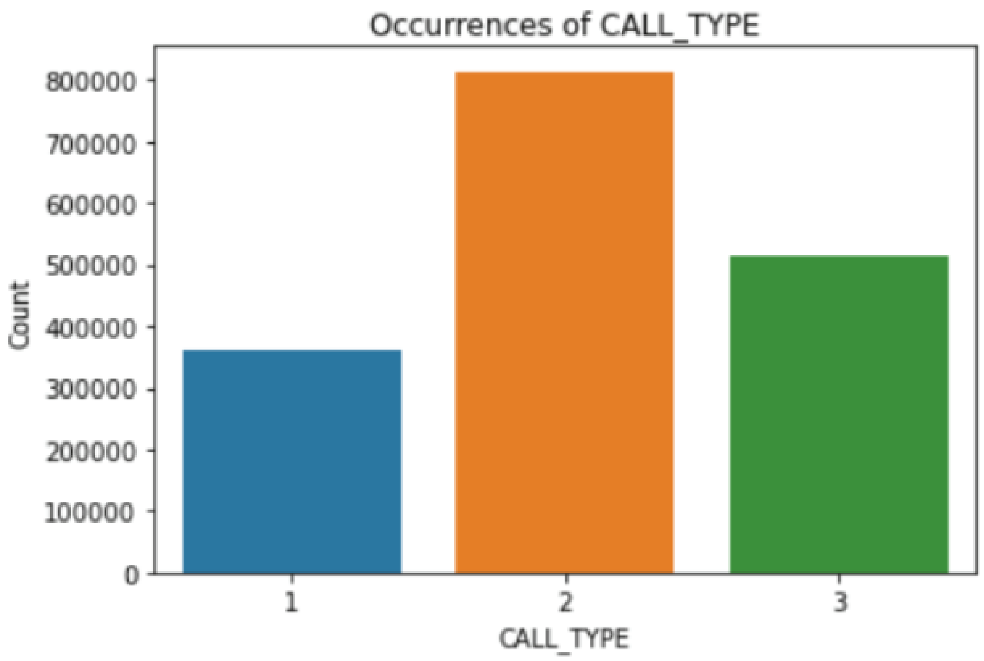


Figure 2.7 Occurrences of 3 different CALL_TYPE

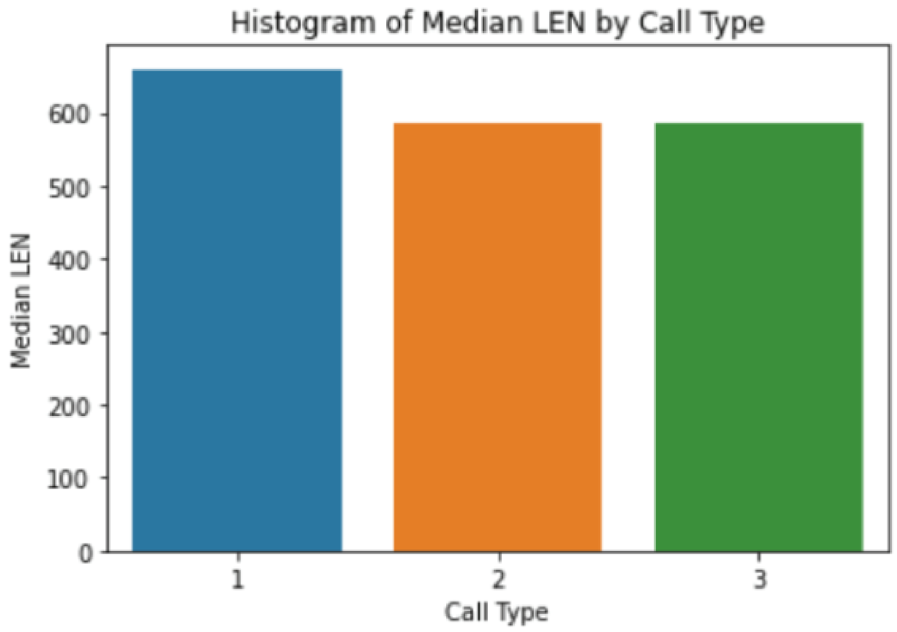


Figure 2.8 Median trip duration of 3 different CALL_TYPE

Interestingly, almost half of the dataset is call-type B, which means the taxis are demanded on those specific stands. Also, for call type A, which are trips dispatched from the central, there is a tendency that these trips are longer than the other 2 call types. This makes sense because if customers choose to make phone calls for a ride (call-type A), they are more likely to be willing to wait for the taxi dispatched to their locations, because there is a larger chance that their trip is planned. Planned trips are usually longer than the trips demanded on a random street (e.g. call-type C).

If we extract the time values from TIMESTAMP, we will get the corresponding YR, MON, DAY, WK, HR values for each trip and add them as new columns for the dataset. Instead of digging into frequencies of how each time category appears in the dataset, we decide to focus on the trip length distribution, because that is what we are predicting. Note that we use unique() function on the dataframe to make sure that each category of these time values do exist in the training set. We also calculate the median trip lengths after grouping the dataset by those different time categories. We get the following distributions.

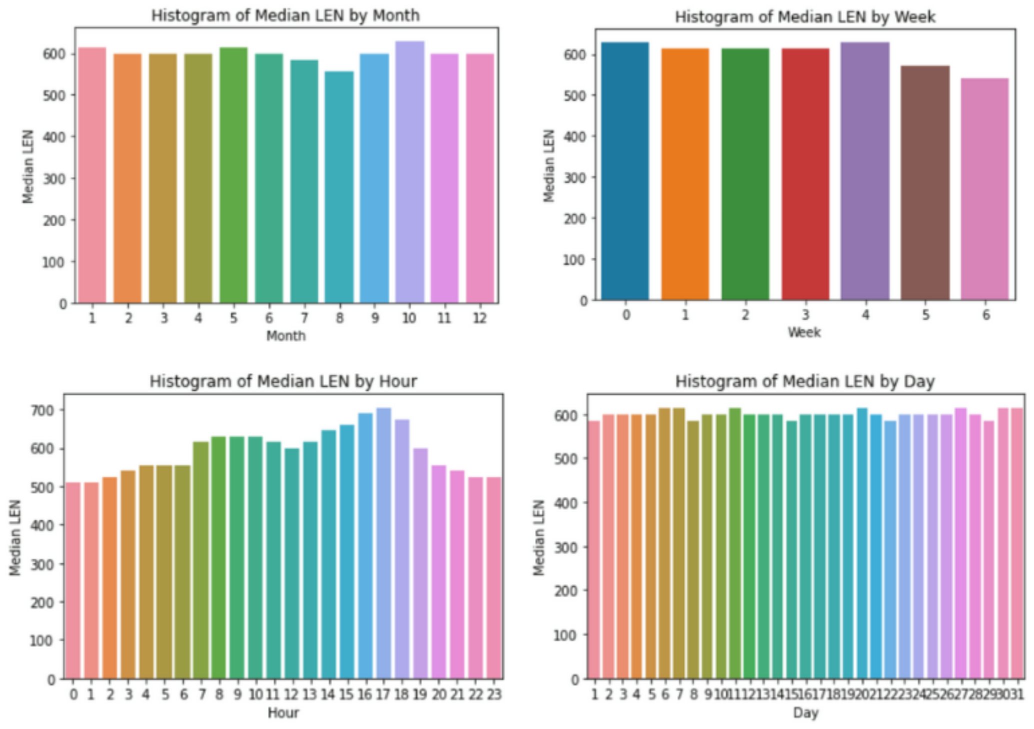


Figure 2.9 Median trip duration of different MON, WK, HR, DAY

For different months, there does not seem to be any relationship between trip duration and the month when the trip occurs, except for the fact that August is the month having the shortest trip duration value in our training dataset. However, we can clearly see some relationship among median trip length for different WK values and median trip length for different HR values. From the distribution above, we can conclude that trips tend to be shorter during weekends than the ones on weekdays. This is probably due to the heavier traffic among weekdays, resulting in longer trips. Similarly, longer trips are more likely to happen from 4 p.m. to 6 p.m., and shorter trips occur more frequently during midnight. While for the last distribution graph, the one showing the median trip duration for different DAY values, we barely observe any patterns. The reason is that it combines all the months altogether. Compared with DAYS, the pattern of median trip duration vs. different WK values is more noticeable. Since in our dataset, all the trips are in 2013, the YR column only has 1 unique value, so if we only look at the trip duration for February, it will be February 2013 only, as the following graph shows.

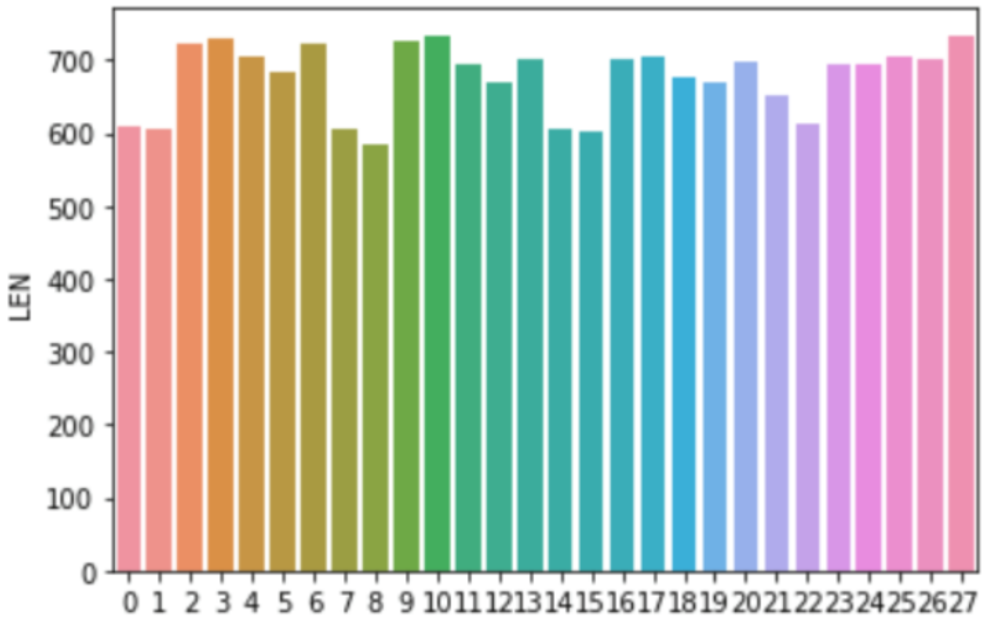


Figure 2.10 Median trip duration of 28 days in Feb 2013

Here, the X-coordinates from 0 to 27, represents Feb 1st, all the way to Feb 28th (2013 is a non leap year). Surprisingly, we can observe a 7-day pattern in the graph which is a continuous 5-day peak followed by a 2-day trough, which once again, indicates the importance of WK values for our training task.

For our DAY_TYPE feature, it is all A in the training set, which contradicts with the data description. We found that there are trip records that happened on Christmas Eve 2013, where DAY_TYPE should be marked as C instead of A. Therefore, we decided to dive into different holidays in Portugal and see if there is any relationship.

For example, as the title of the following 3 graphs show, they show the median trip duration in the three months in the year of 2013 relatively. In December, we can clearly see that during Christmas time, it does not follow the 7-day weekly pattern any more. On Dec 25th., the median trip duration is the shortest of the week for a day having DAY_TYPE of B. Similarly, June 10th 2013, being the Portugal Day as a Monday, has a very low median trip length, being a B DAY_TYPE as well. However, May 12th 2013, being Mother's Day in 2013 and it is a Sunday, which is supposed to have shorter trip duration during the weekend, it turns out that the median trip duration is extremely high, even though it is another B DAY_TYPE. Therefore, we decide not to include DAY_TYPE information in our input data, due to the inconsistency in the median trip duration and DAY_TYPE. Also, we should be expecting the model to learn the relationship if we include both DAY and MON information.

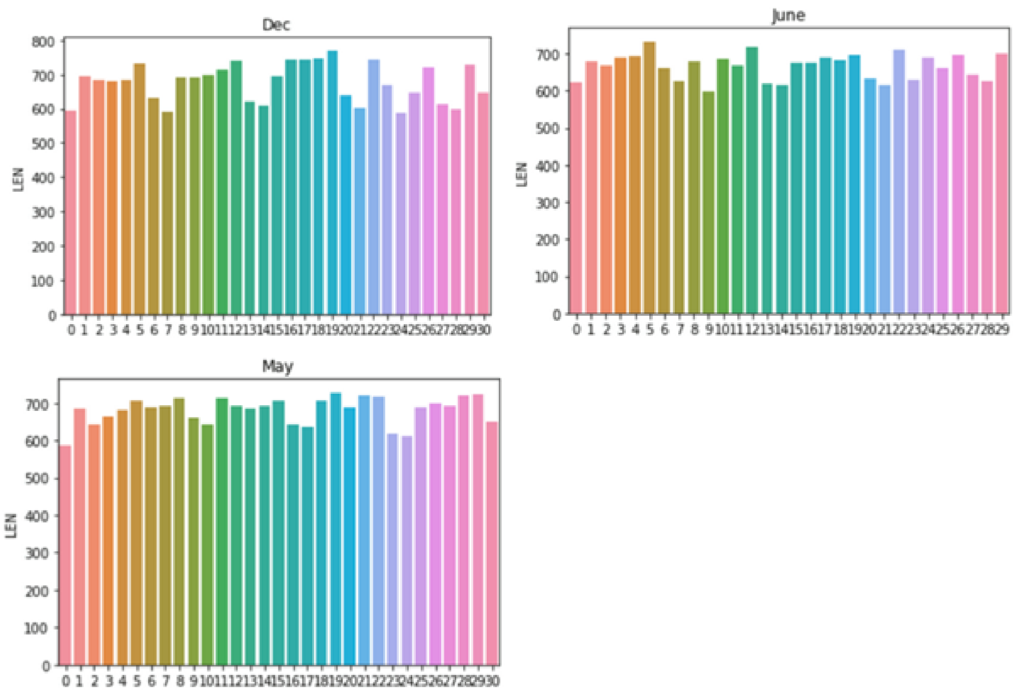
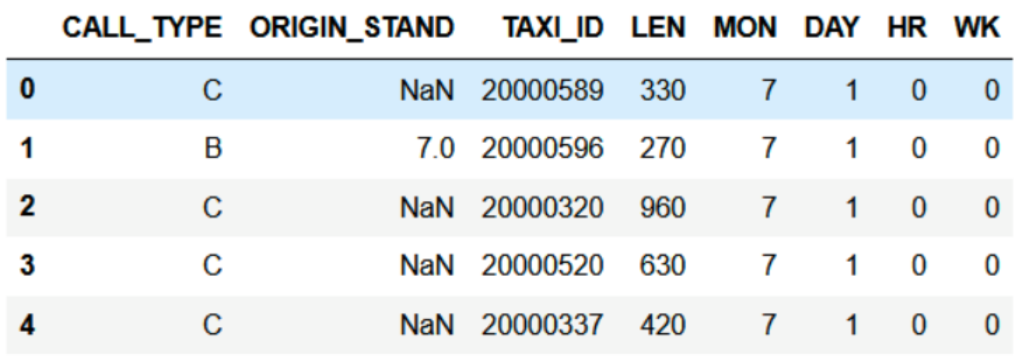


Figure 2.11 Median trip duration of days in Dec 2013, June 2013, May 2013

For TAXI_ID, we found out that there are 448 unique values indicating 448 different taxis recorded in the dataset. Since it is categorical data, if we want to use it as input to train our model, we want to make sure that every trip's TAXI_ID in our public test dataset exists in the training set. If there are way too many new TAXI_IDs in the test set, it may not be a good idea to even include TAXI_ID as an input feature in the first place. Luckily, every TAXI_ID in the public test set does have at least one appearance in the training dataset.

Every trip does have TAXI_ID, but not necessarily an ORIGIN_STAND value. It will only be an integer from 1 to 63 if CALL_TYPE is B. Thus, there are 64 unique values if we do consider the null entries. We decided to include this because 63 unique stands represent 63 different locations for people waiting in lines for taxis. There has to be some relationships so that some stands, for example the airport, have a much longer trip duration than the others.

For the other features, we decided to remove some of them for our training purpose. Firstly, we removed TRIP_ID because it is basically a combination of TIMESTAMP and TAXI_ID, which are redundant information. ORIGIN_CALL, representing the phone number for the customer when the CALL_TYPE is A. We found out that there are more than 10 thousand unique phone numbers in the training set, and since it is categorical data, it is such a memory overhead if we do one hot encoding including this column. Thus, we decided to drop this column. For MISSING_DATA, DAY_TYPE and YR, there is only 1 unique value for these columns, so we dropped them, because the model cannot learn anything if the training dataset has uniform values. After dropping these columns, we end up having the data frame looks like the following.



	CALL_TYPE	ORIGIN_STAND	TAXI_ID	LEN	MON	DAY	HR	WK
0	C	NaN	20000589	330	7	1	0	0
1	B	7.0	20000596	270	7	1	0	0
2	C	NaN	20000320	960	7	1	0	0
3	C	NaN	20000520	630	7	1	0	0
4	C	NaN	20000337	420	7	1	0	0

Figure 2.12 the only 7 categorical data in the data frame before doing one hot encoding

Notice that LEN is the trip duration we calculated from POLYLINE, which will be removed later and saved as Y labels. Now, note that for all the features remaining, they are categorical data. We use one hot encoding method for these 7 features. Basically what one hot encoding does is to create N entries if one original entry has N unique values. For example, CALL_TYPE has 3 different values–A, B, and C. Instead of using the current CALL_TYPE column, we add 3 new columns, where each column can only be either 0 or 1. The 3 new columns is 1 0 0 for CALL_TYPE A, 0 1 0 for CALL_TYPE B, 0 0 1 for CALL_TYPE C. As a result, we removed the numerical relationship within these categorical data and the model can learn fairly for different categorical values. Using this technique, after doing one hot encoding, the number of resulting input features will be 3 (CALL_TYPE) + 64 (ORIGIN_STAND) + 448 (TAXI_ID) + 12 (MON) + 31 (DAY) + 24 (HR) + 7 (WK) = 589, where each number represents how many unique values in that original column. Now, for each row, the input data will look like 589 zero or one, where there are only 7 1's in each row, and their positions indicate the value for the original 7 features. One trick we did with one hot encoding is that we need to call fit() function for the encoder to map how many unique values before we remove outliers. This is because there are 7 TAXI_IDs that only appear in those outliers. If we want to make sure that every TAXI_ID from the public test dataset appears at least once in the input data after doing one hot encoding, we need to call fit() in the first place to avoid the problem.

Since we are doing one hot encoding for all features, they are all 0s and 1s, there does not leave any numerical value. Thus, we do not do any normalization. However, in our previous version of feature engineering. We utilized some normalization methods. We calculated the mean values after grouping MON values and added it as a new column called MON_MEAN_LEN. For example, for 2 trips that are both in January, they should have the same MON_MEAN_LEN, which is the mean value for all the trips in January in the dataset. Then, we do the same thing for WK, HR, and ORIGIN_STAND. After we have these mean value columns, we used min-max normalization method, using the formula: f(x) = (x-min) / (max-min) * 1.0 + (0.0). Which will end up with values between 0 to 1, and it matches our 0s and 1s from one hot encoding. The reason why we gave up on this approach is because there is a tradeoff between using these float numbers which take up much memory space so that we cannot include one hot encoding for TAXI_ID. It turns out that having these mean values does not improve the performance at all. Also, the model should be able to learn the relationship between those categorical values and labels. Thus, we choose to use one hot encoding only.

3 Deep Learning Model

3.1 Problem A

The pipeline for our learning task is the following:

- Do the exploratory data analysis on the raw dataset and decide what to keep as input features.
- Conduct feature engineering on both the training data and the public test data, including removing outliers, dropping unlearnable columns, and doing one hot encoding on all the

categorical features. Save them as csv files so that solves the memory issue and improves the efficiency of testing different models.

- Split the training data into training set and validation set, to test our models on the validation set for the purpose of avoiding overfitting.
- Starts with basic Machine Learning models, like Linear Regression. Then, exploit with other Machine Learning models (LightGBM, Random Forest, BayesianRidge, etc.), as well as Deep Learning models (Multi-Layer Perceptron).
- Record the training loss and validation loss, and exploit with different parameters to get better performance.
- After trying plenty of different models. We are inspired by the idea of boosting in Machine Learning, so that we decide to choose several best models, use their prediction results as new features, and feed them into the Linear Regression model and the MLP, to see if there is any improvement. Surprisingly, it turns out that this method gives even better results, which is used as our final model for this prediction task.

As we have discussed the details of how to decide what to include in the input features, we end up leaving all categorical data in the training data and perform one hot encoding on them, resulting in a total of 589 input features. For the output feature, there is only 1, because it is a regression problem, so that the only output feature is the trip duration we are predicting.

For Deep Learning models, we decide to choose Multi-Layer Perceptron (MLP) over Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN), because the prediction task is the trip duration as a float number, which makes it a regression problem. CNN is usually better at prediction tasks on images and 2D vectors and exploiting spatial locality, whereas RNN does better in exploring temporal locality. Thus, we decided to choose fully-connected MLP as our Deep Learning model, which is suitable for regression problems.

Then, we tried 2 different MLP architectures. Our first intuition is that we want to try both deep but narrow networks and flat but wide networks, so we have the following different architectures to make a balance for this tradeoff.

Architecture 1:

- 6 hidden layers,
- input dimension: 589
- First Hidden Layer: dimension 589 -> 32
- Second Hidden Layer: dimension 32 -> 32
- Third Hidden Layer: dimension 32->32
- Fourth Hidden Layer: dimension 32 -> 32
- Fifth Hidden Layer: dimension 32 -> 32
- Sixth Hidden Layer: dimension 32 -> 32
- Output Layer: dimension 32->1

Architecture 2:

- 3 hidden layers,
- input dimension: 589
- First Hidden Layer: dimension 589 -> 128
- Second Hidden Layer: dimension 128 -> 64
- Third Hidden Layer: dimension 64->32
- Output Layer: dimension 32->1

Architecture 3:

- 4 hidden layers,
- input dimension: 589
- First Hidden Layer: dimension 589 -> 1000
- Second Hidden Layer: dimension 1000 -> 750
- Third Hidden Layer: dimension 750 -> 500'
- Fourth Hidden Layer: dimension 500 -> 250
- Output Layer: dimension 250 -> 1

It turns out that Architecture 2 gives a better result in general. Our assumption on why Architecture 2 is better than Architecture 1 is that even though the input data is extremely sparse, only 7 of 589 columns are 1 and the rest are all 0, wider layers are more helpful than the number of hidden layers. Also, we notice that the training loss for Architecture 1 stops decreasing at around 405, but for Architecture 2, it is still shrinking when it is at about 386. It indicates that complex models seem to have better results. Notice that we did not choose Architecture 3, because it takes us more than 20 minutes to train each epoch, and after training it for 50 epochs and notice that the validation loss begins to ramp up after the 7th epoch. Thus, we upload the results using Architecture 3 and notice that the public test score is about 775, which is higher than the one we got using Architecture 2, which is around 770. Therefore, we choose architecture 2 at the end, and decide to use it as the primary structure for our deep learning model (we also reuse the same model in our own HOM model, which will be discussed later).

The loss function used in our multi-layer perceptron model is MSE (mean squared error), which is the average of squared difference between the predicted values and the corresponding actual values. The activation function used in all the different architectures is the ReLU function.

3.2 Problem B

We tried different feature engineering methods, the previous was discussed in the milestone report, but it didn't produce a good public score on the kaggle leaderboard. Thus, we changed our feature engineering, and trained new models on the new transformed data.

We have tried five models:

- LightGBM Regressor
- Linear Regression
- Random Forest
- BayesianRidge
- MLP

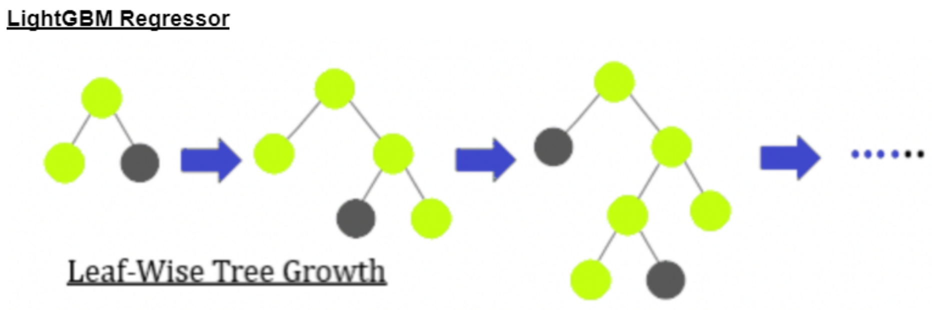


Figure 3.1 Visualization of Leaf-Wise Tree Growth [1].

- Data Processing: One-hot encoding

- Architecture: 100 Leaf-Wise growth gradient boosting Decision Trees.
- Parameters: n_estimators = 100, max_depth = unlimited, num_leaves = 31
- Loss Function: MSE loss
- Regularization: None

Linear Regression

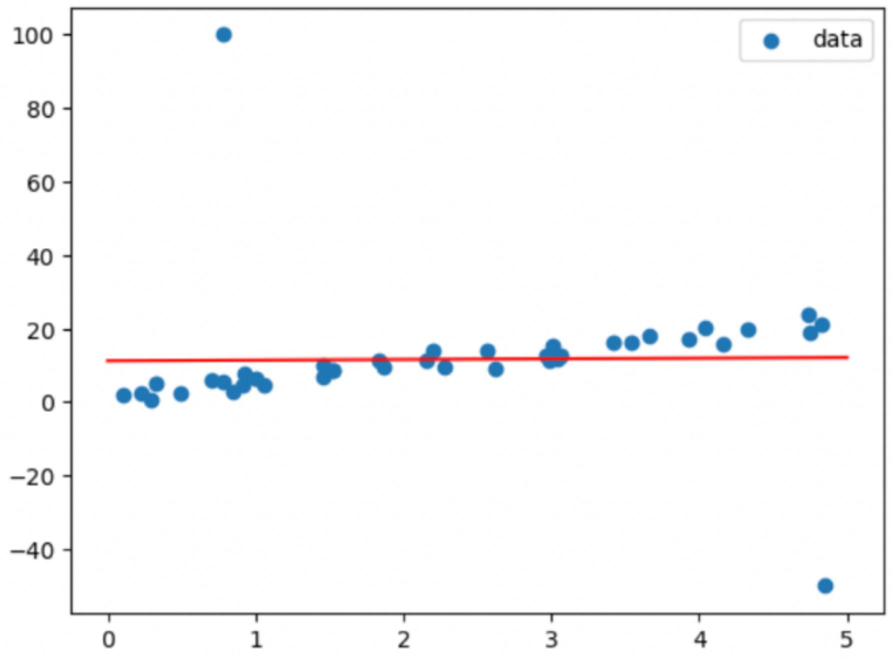


Figure 3.2 Linear Regression in 2D

- Data Processing: One-hot encoding
- Architecture: Fix a hyperplane to the pre-processed training data.
- Parameters: None
- Loss Function: MSE loss
- Regularization: None

Random Forest

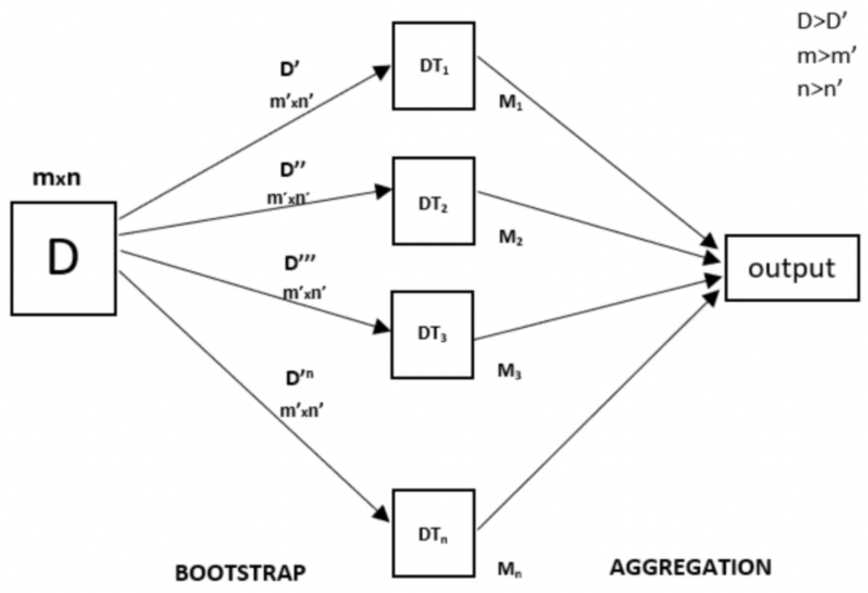


Figure 3.3 Visualization of Random Forest model [2].

- Data Processing: One-hot encoding

- Architecture: 100 Decision Trees combined.

- Parameters: we utilized Sklearn random forest regressor, n_estimators = 100, max_depth = unlimited, min_samples_split = 2, min_samples_leaf = 1

- Loss function: MSE loss

- Regularization: None

BayesianRidge

The last Machine Learning method we tried is Bayesian Ridge. It is quite similar to Linear Regression in terms of generating weights of 589, which matches our input data dimension.

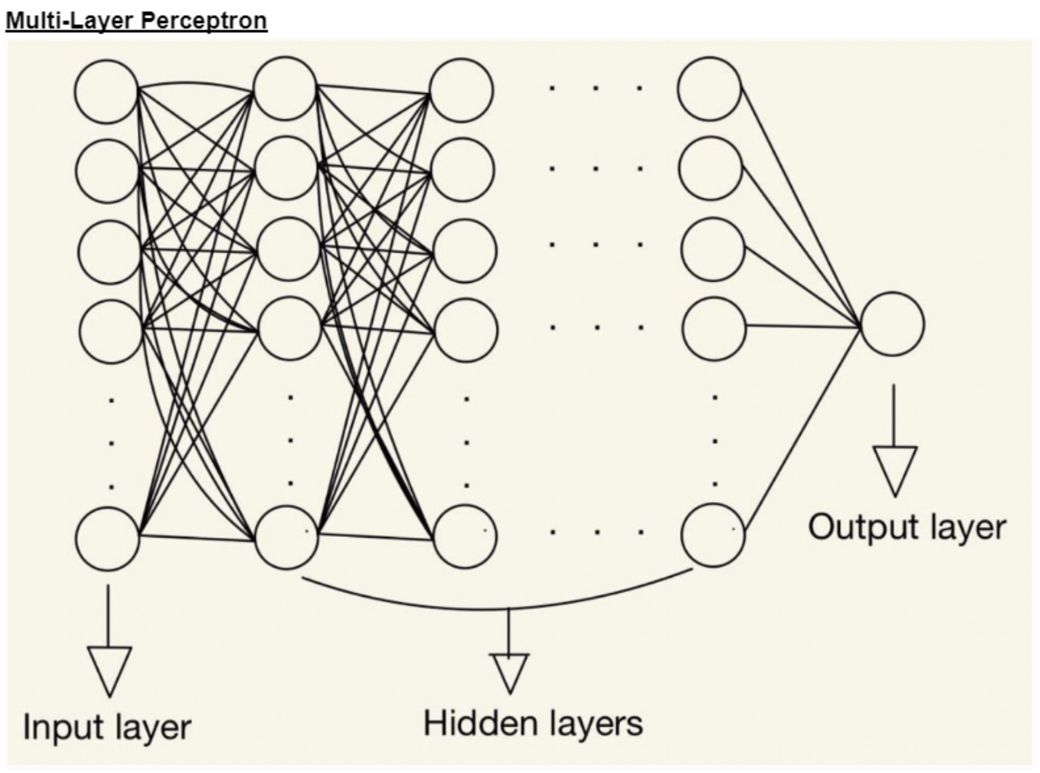
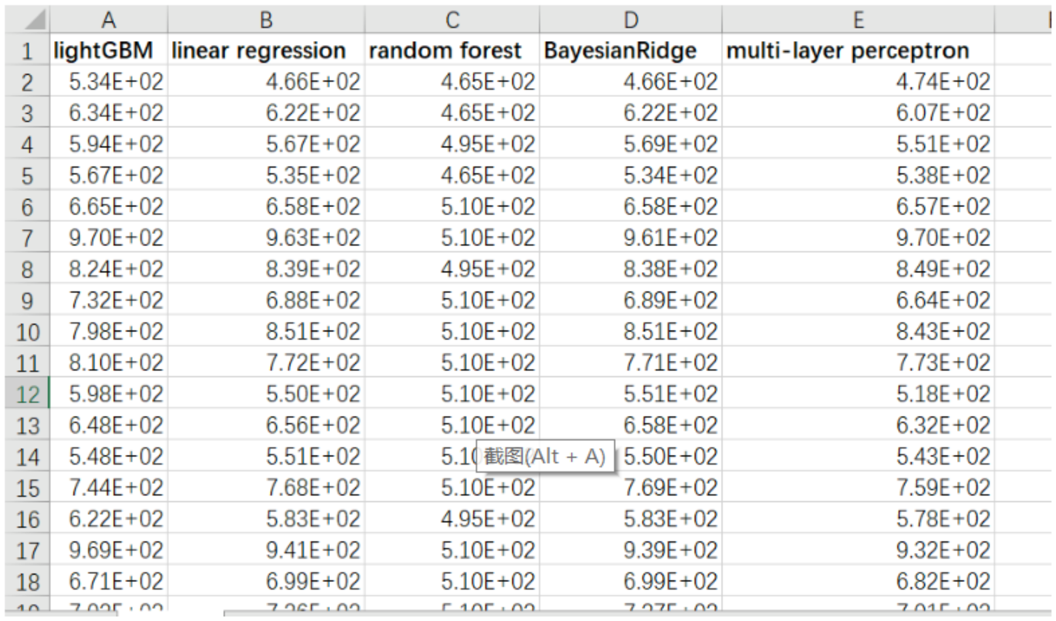


Figure 3.4 Visualization of MLP model

- Data Processing: One-hot encoding
- Architecture:
  - 3 hidden layers,
  - input dimension: 589
  - First Hidden Layer: dimension 589 -> 128
  - Second Hidden Layer: dimension 128 -> 64
  - Third Hidden Layer: dimension 64->32
  - Output Layer: dimension 32->1
- Activation Function: ReLu for each fully connected layer
- Loss Function: MSE Loss
- Regularization: dropout after each fully connected layer with probability of 0.25

heuristic-oriented model (HOM)

Our final model is a combination of all previous models we have tried. We give it a fancy name called "heuristic-oriented model" (HOM). Inspired by the boosting in machine learning, we want to see if we can combine all the five models and feed the predictions using the five models into another model, expecting the new model could produce a better result. The process is described below:
1. We first use each model to predict the original training set. The output of each model is an array containing all the travel time predictions. It has the same length as the training set. We would have five of those arrays, and we save them into a csv file for future use. Also, we do the same operation on the validation set and test set, and save them into csv for future use.



	A	B	C	D	E
1	lightGBM	linear regression	random forest	BayesianRidge	multi-layer perceptron
2	5.34E+02	4.66E+02	4.65E+02	4.66E+02	4.74E+02
3	6.34E+02	6.22E+02	4.65E+02	6.22E+02	6.07E+02
4	5.94E+02	5.67E+02	4.95E+02	5.69E+02	5.51E+02
5	5.67E+02	5.35E+02	4.65E+02	5.34E+02	5.38E+02
6	6.65E+02	6.58E+02	5.10E+02	6.58E+02	6.57E+02
7	9.70E+02	9.63E+02	5.10E+02	9.61E+02	9.70E+02
8	8.24E+02	8.39E+02	4.95E+02	8.38E+02	8.49E+02
9	7.32E+02	6.88E+02	5.10E+02	6.89E+02	6.64E+02
10	7.98E+02	8.51E+02	5.10E+02	8.51E+02	8.43E+02
11	8.10E+02	7.72E+02	5.10E+02	7.71E+02	7.73E+02
12	5.98E+02	5.50E+02	5.10E+02	5.51E+02	5.18E+02
13	6.48E+02	6.56E+02	5.10E+02	6.58E+02	6.32E+02
14	5.48E+02	5.51E+02	5.1 截图(Alt + A)	5.50E+02	5.43E+02
15	7.44E+02	7.68E+02	5.10E+02	7.69E+02	7.59E+02
16	6.22E+02	5.83E+02	4.95E+02	5.83E+02	5.78E+02
17	9.69E+02	9.41E+02	5.10E+02	9.39E+02	9.32E+02
18	6.71E+02	6.99E+02	5.10E+02	6.99E+02	6.82E+02
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Figure 3.5 The predicted results of each model on training set

2. We then use this new data as the training set, and apply different models on this set.

3. We test on different models and choose the best model to do prediction on the test set.

By intuition, we want to assign each model a weight, and take the weighted sum of all five models' output as the final output. So we first tried to use linear regression as the final step before the output (See Figure 3.x). Hence, we fit a Sklearn Linear Regression on the training set, and it took about 1s to complete. Next, we want to make this combined model more complicated, so we decided to use MLP as the final step before the output. We utilized the same architecture as the previous MLP model because that one had the best performance.

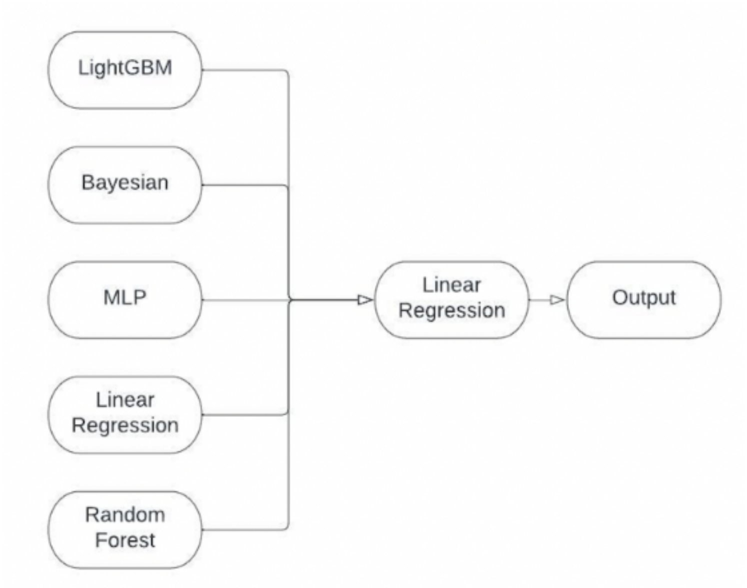


Figure 3.6 HOM model with Linear Regression Output

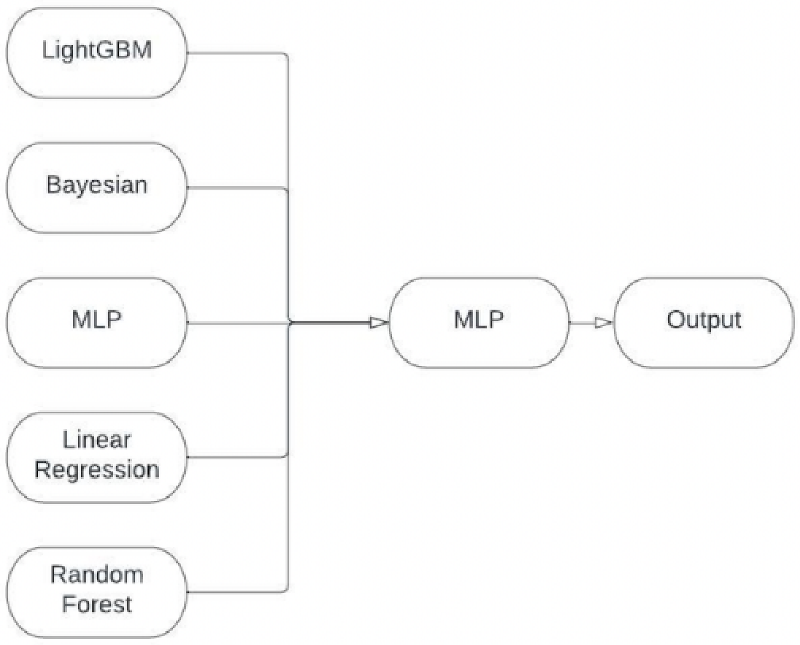


Figure 3.7 HOM model with MLP Output

4 Experiment Design and results

4.1 Problem A

We completed our computation on datahub.ucsd.edu for both training and testing. After we made final decisions on feature engineering and data processing, we saved processed training data and test data into csv files so that multiple teammates can try different models at the same time. Primarily, we relied on one of our teammates' laptop with a NVIDIA 3070 GPU with cuda settings turned on for the majority of our training and testing tasks.

We choose Adam as our optimizer for MLP over SGD, because SGD uses fixed step size, which takes longer to converge even though it guarantees to find the optimum. While for Adagrad, it will adapt the step size dynamically, however, as we have tested in the project milestone, it does not converge as fast as Adam. Also, Adam is considered a more popular optimizer for MLP, which also gives dynamic learning rate decay as adjustments during the training. For the learning rate, we are motivated by our past experience from Homework 4 and the project milestone. We began with the learning rate of 0.0001 and it turns out that converges quicker than we would expect. Then, we decide not to even try a larger learning rate, such as 0.001 since it would only be worse. Thus, we end up using the learning rate of 0.00001, which gives a perfect training loss exponential decay graph. We use 0.9 as the momentum, since it is also a common practice. After we tried different momentum values in the project milestone, we did not notice much difference. Thus, we decide to keep 0.9 as default.

For regularization methods, we tried both batch normalization and dropout. We notice there is a huge increase in validation loss if we add batch normalization before the activation function in each layer. However, it turns out that the dropout method gives a slight improvement in the model. We tried dropout rates of 0.25, 0.5, 0.75, and we found out that 0.25 gives the best result, even though with dropout, the training loss decreases slightly slower. It totally makes sense because adding dropout in all the layers means that we randomly turn off some neurons during training, which increases the time for the loss to converge.

We trained 100 epochs with a batch size of 30, with the number of workers being 2. For the MLP Architecture #2 with the first hidden layer size being 128, it took us about 2 minutes to train 1 epoch. While for Architecture 3 with a first-layer hidden size of 1000, it took more than 20 minutes to train 1 epoch

4.2 Problem B

To compare different results, training time, and number of parameters of different models, we create the following table:

model	training loss	validation loss	public test score	training time estimation	# of parameters
LightGBM	373.9727168608378	374.76647406151574	771.9719	5 min	3000
Linear Regression	376.8693865378112	377.6907599107631	778.9339	40 s	590
Random Forest Regressor	437.48123257725797	438.3986068549048	789.78389	17 min	16300
BayesianRidge	376.87115766346403	377.6935761656919	782.72438	13 min	589
MLP	389.241	376.871	769.66934	5 hr	86021
HOM Linear Regression	372.8751277861822	373.6816272642171	763.3734	1 s	5
HOM MLP	391.13735515802784	375.38357701356085	760.99909	14 min	11269

Conclusion drawn from this table:
We expected that higher validation would result in a high Kaggle test loss. From the results table we generated, we found that most models had a training/validation loss under 400, except for one, which was Random Forest Regressor, which had its training/validation losses both above 400. And its Kaggle public test score is also the worst (789.78389). However, although for other models, validation loss and training loss are very close in each model, and the validation losses are also simiar across models, their Kaggle public scores vary. Therefore, we assumed that the test data set has a different distribution, and we would use the Kaggle score as the metric to evaluate our best model. The final model we selected was the HOM MLP, which has the lowest Kaggle score of 760.99909.

4.3 Problem C

Visualization for MLP training/validation loss (MSE):

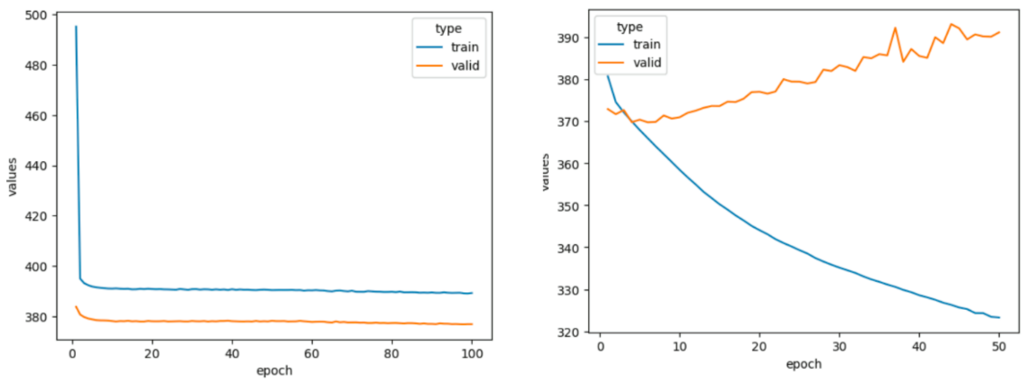


Figure 4.1 Initial MLPs (hidden size 128 100 epochs: left vs. hidden size 1000 50 epochs: right)

As we can see in this comparison of training/validation loss for 2 different MLP architectures, we can observe both exponential decay in training loss. However, the right model, which is the one with much larger hidden layer, shows a much better result in terms of an interception of training loss and validation loss, which is around epoch #7. This interception indicates where we should do early stopping to avoid overfitting. However, we actually chose the left MLP architecture for our fifth model and fed it into the HOM model, because of two reasons. Firstly, the one on the left (hidden size 128) has a lower public test score than the one on the right. Also, the training time for the one on the right is more than 10 times longer than the one on the left, so we have to make this decision. More details on this will be discussed in section 5.

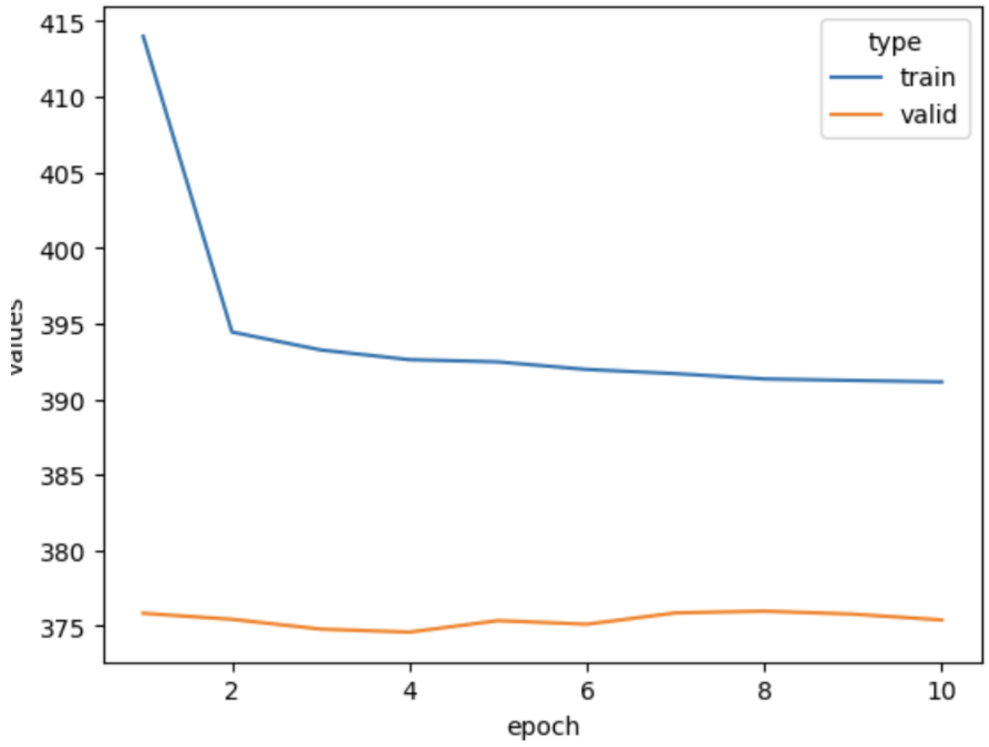


Figure 4.2 HOM final MLP (hidden size 128) 10 epochs

Even though the loss graph of our HOM model using MLP, as shown in figure 4.2 does not look as good as the one before, it actually improves our public test score by quite a lot. This is probably because we reuse the 128 hidden layer size structure for our input data dimension being 5, which is arguably kind of wasteful in some sense. However, this huge model seems to be way more than enough to learn for our HOM model.

Visualization for LightGBM training/validation loss (MSE):

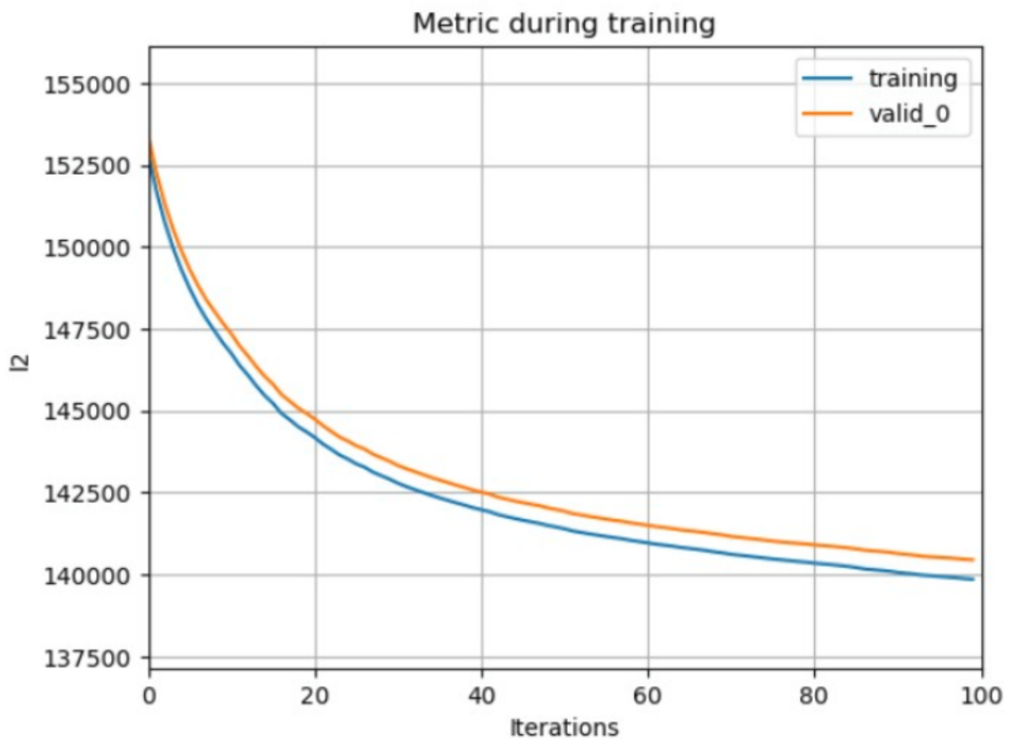


Figure 4.3 LightGBM Regressor Training/Validation Losss

The training loss and validation loss of LightGMB Regressor is pretty standard as both of them decrease exponentially. Also, the validation loss is slightly larger than the training loss, which is what we would expect for this training task.

Comparisons of training/validation loss (MSE) for all models:

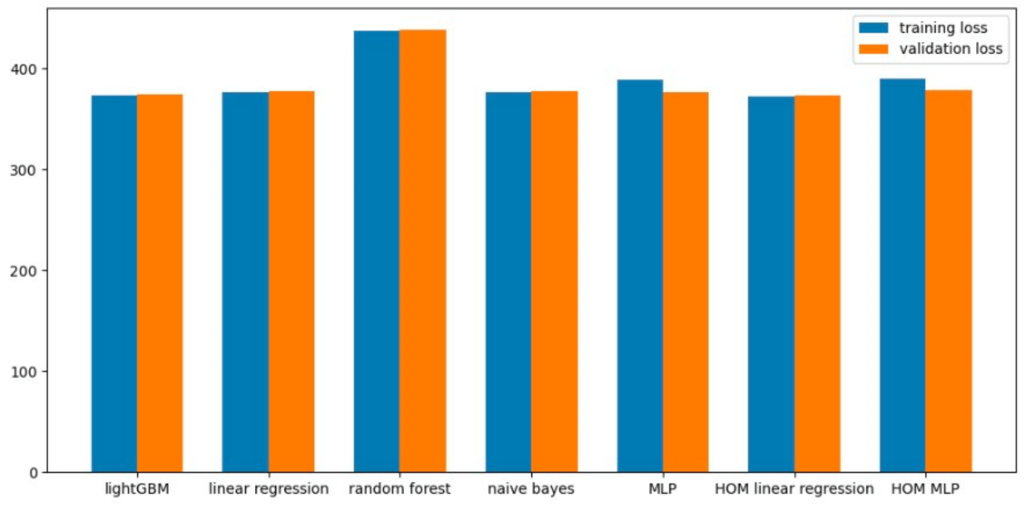


Figure 4.4 Training/Validation Losses across all Models

To quickly summarize figure 4.4 where we compare losses for all the models. We can clearly see that random forest did the worst in this prediction, and for most of them, validation loss is higher than training loss, which totally makes sense.

Visualization of ground truths vs. predictions:

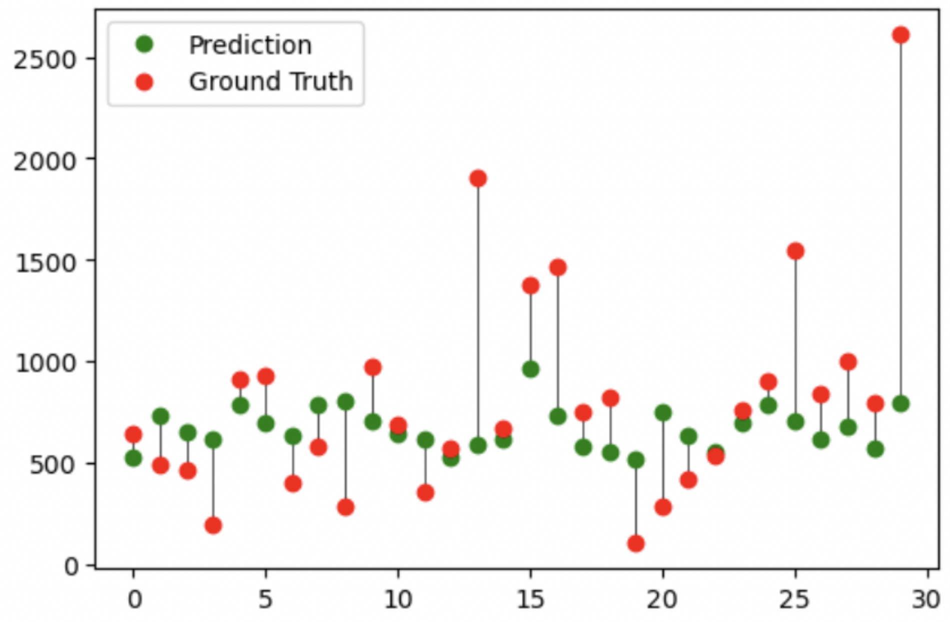


Figure 4.5 Ground Truth vs Prediction with 30 random samples

As we randomly sample 30 data points and compare their ground truth values and prediction values in figure 4.5, we can see that almost half of them being ground truth higher than the prediction and the other half being the opposite and most of them are pretty close. It can be considered one of the indications that our final model did a decent job in predicting the trip duration. Also, we can see that our prediction seems to have smaller variance than ground truths.

Our final ranking on the leaderboard after the release of the entire public test is 32, with the final test MSE being 1060.48328 (This is the one predicted by HOM MLP model).

5 Discussion and Future work

5.1 Problem A

Effective Strategies and Techniques:
The most effective feature engineering strategy is one hot encoding. What we have tried in the project milestone is to remove TAXI_ID and replace it with the median values of trip duration after grouping the data into unique TAXI_IDs. Even though it drastically reduces the input data dimension, the performance was not pleasing. Similarly, we also add those calculated median values for columns like MON, WK, HR, etc. After the project milestone, we decided to only leave 7 useful columns and all 7 of them are categorical data. Then, we do one hot encoding on them, including TAXI_ID and ORIGIN_ID, which increase our input dimension from 54 to 589. It turns out that the result improves by much. Thus, we conclude that using one hot encoding is extremely effective in terms of dealing categorical data, and there is no need to group them and assign median values for them, since it is redundant and the relationship between different categories is something the model can learn. With that being said, including as much useful information from the raw data as possible should give a better result.

The technique we found that improves our ranking the most is coming up with the idea of heuristic-oriented model (HOM). As we have tried a bunch of different Machine Learning models, as well as the Deep Learning mode like Multi-Layer Perceptron, we were inspired by the idea of boosting and Homework 4, where we gave initial weights for a gradient descent algorithm for a regression problem. Utilizing this idea, we assembles the prediction results using the 5 models and feed into simple linear regression model and we get the following result weight: [1.0428565 , 5.59449289, -0.17479039, -6.8341273 , 1.35239159], which corresponds to the weight for LightGBM, linear regression, random forest, bayesian regressor, MLP. As we can see the weight for random forest is only -0.17, which is the smallest. It totally makes sense because from our loss results analysis in section 4, we notice that random forest is the worst in terms of performance. Therefore, small weight means that it contributes the least to the forming of the final HOM model. Notice that for this linear regression, we purposefully choose to turn off the bias value. This is because for our input data, already has 5 different prediction values that should be somewhere in the proper range for the actual trip duration. We do not want to have a bias term to throw things off. While actually, we also tried to add in the bias, and as what we just discussed, it is not as good as the one not using bias. To extend this idea, we thought that why we did not feed the data into a more complex model than simple linear regression. Hence, we reused our MLP structure (hidden size 128) and fed the 5 columns into it. The result is even better, because MLP is a much more complex model than linear regression. Some other small trick we did is to round our final prediction result to multiples of 15, because the POLYLINE column is sampled every 15 seconds. However, it does not work as well as we expected, as it sometimes improves the public test score and sometimes it does not. Thus, we still chose to throw this one off for our final prediction.

Challenges:
The major challenges for this project is hardware limitation, we found ourselves running out of memory and the dead jupyter notebook kernel quite often. The first main trouble we encounter is not being able to calculate validation loss after the training is done. We solved it later by creating a data loader for the validation set as well so that we can calculate the validation loss after every epoch without taking much memory at once. Similarly, since we sharply increase our input data dimension from 54 to 589 after the project milestone as we choose to do one hot encoding on columns like TAXI_ID, where there are 448 unique values. We ran out of memory while moving the training set to the device as tensors. What we did for this issue is to create a new notebook doing all the data preprocessing work and store training data and test data into csv files. It does not only solve the memory issue, but also makes the code more neat, and multiple teammates can test different models at the same time. However, still due to the memory problem, we really wanted to try the grid search method using LightGBM, but it seems like we are kind of limited by our hardwares for this specific model.

Something as advice for deep learning beginners is to trust the results from the validation set, and more complex the model usually gives the better results. The reason we are saying this is because remember we had two architectures for MLP to decide, the one with the first hidden layer size of 128 (A), and the one with 1000 (B). When making the decisions, we chose A over B, because A has a better public test score than B, and B takes more than 10 times to train than A does because it is simply a much larger network. However, if we actually take a look at the training/validation loss of 2 models, model B has much lower validation loss and it converges earlier than model A. After the release of private hidden test scores, model B, the larger MLP, is actually the best model among all. We figure the reason why the previous public test score for model B is disappointing is that there may be a distribution shift among the sampling of the public test data from the entire private hidden test dataset. Thus, we should be trusting our validation results more than the public test score.

Future Work:
The 63 coordinates corresponding to the taxi stands of ORIGIN_STAND, along with the latitude and longitude information inside the polyline, can be used as a 2D input on a map. Then, a CNN (Convolutional Neural Network) can be trained on this 2D map. In our case, the 2D map can represent a grid or image where each coordinate or point is marked or represented in some way. The CNN can then learn spatial patterns and features from this map to make predictions on the route trajectory of the taxi.

We may also utilize the time series information (ordered by TIMESTAMP column), and apply Transformer or LSTM on the encoded information. In this case, Transformer would capture the patterns in the sequential data, and thus historical route information to predict the new route trajectory of the taxi.

We also want to try out Sklrean GridSearch on each sub-models such as LightGBM and RandomForest if we successfully configure the GPU setting for LightGBM or a bigger RAM for our CPU. This would help to find the best hyperparameters combination for LightGBM and RandomForest Regressor with the least cross validation loss , and thus a best model.

References

[1] shreyanshisingh28. (2023) *LightGBM (Light Gradient Boosting Machine)* . GeeksforGeeks.org. Retrieved from https://www.geeksforgeeks.org/lightgbm-light-gradient-boosting-machine.

[2] Avik_Dutta. (2023) *Random Forest Regression in Python* . GeeksforGeeks.org. Retrieved from https://www.geeksforgeeks.org/random-forest-regression-in-python.