Taxi Trip Estimator

Oren Ciolli
ociolli@ucsd.edu

Anthony Dang
andang@ucsd.edu

Carlos Monterosa
cmonterosa@ucsd.edu

Esperanza Rozas
erozas@ucsd.edu

1 Task Description and Background

In this project, we use data from Taxi rides in Porto, Portugal to try to predict how long a trip will take. Among the variables we have access to are the time that the ride began and information about how the ride was booked, including the type of calls, what taxi stand the ride began at (if applicable, as not all rides started at a stand), the unique ID of the taxi itself, and the time and date. For a taxi company, this kind of analysis is hugely important to improving the efficiency of their service. Being able to accurately predict the duration of a ride would allow the dispatcher to schedule rides more efficiently to minimize downtime between rides, and minimize wait time for riders.

Specifically, along with the predicted information about how long people are in the taxis, the information can be used to find:

- How much traveling taxis have to do before picking up their next ride
- If the number of taxis needs to be increased to cover the number of riders daily
- How much traffic patterns and different days affect the service

Increased efficiency with taxi dispatchment could improve traffic in the city greatly and have the most impact on riders who are likely to use taxis the most: tourists. Tourists are an important part of the economy in Porto, especially in summer months (as a coastal Mediterranean city) and has become more so in the last few years). As such, increased efficiency can help ensure tourists get to their destination more quickly and continue to use future taxi services, benefiting both the tourist locations and the taxi company).

Mathematically, the input to our final model is a set $\{\vec{x}_i, y_i\}_{i=1}^{N}$ where $\vec{x}_i \in \mathbb{R}^{143,1}$ and $y_i \in \mathbb{R}^{+} \cup \{0\}$. That is, for each data point (represented by a vector of 143 features, resulting from the onehot encoding which we describe later), there is an associated travel time (in seconds) which belongs to the set of nonnegative real numbers. In this case, $N = 1539594$, as this is the number of unique points in our training dataset (post validation split and preprocessing).

Our model is represented by a prediction function f, and we'll be attempting to minimize root mean squared error $\mathbb{L} = \sqrt{\frac{\sum_{i=1}^{N}(f(\vec{x}_i)-y_i)^2}{N}}$. Formally, we'll attempt to find the optimal weights w and biases b to solve the optimization problem: $argmin_{w,b} \sum_{i=1}^{N} \mathbb{L}(y_i, f)$.

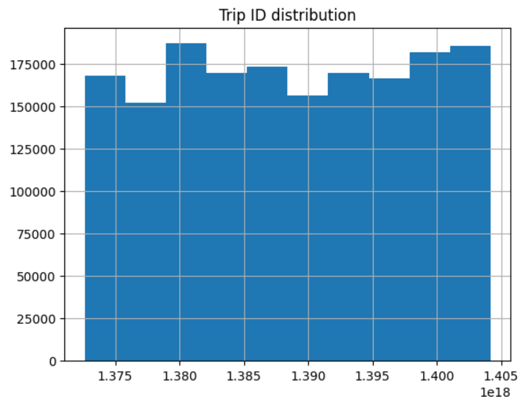


Figure 1: Histogram of Trip IDs in the training dataset.

In theory, solving this problem should be equivalent to solving any task for regression, as the goal of regression is to predict a continuous value (in our case, travel time) while minimizing loss (in our case, root mean squared error). Our model is not the best for regression tasks in general by any means, as it's relatively simple and likely will fail to capture the complexities present in most datasets, including this one. However, an example of a task that may be worth trying with our general framework is predicting the income of an individual given some categorical variables about them (such as their age, education history, home country etc.). Just like we do with our task, it could make sense to use either an embedding or a one-hot encoding to encode these categorical variables, then pass it into an MLP which outputs the person's predicted income. And of course, this model should be applicable in other travel time prediction tasks, such as predicting the time of taxi rides, or even car journeys, in other cities where data of this format are available.

2 Exploratory Data Analysis

2.1 Dataset Sizes

Starting with an overview of the data, there are 1,710,670 trips in the training dataset and 320 trips in the test dataset. The training dataset trip IDs are labeled numerically. We did not find any obvious patterns in numbering for the Trip IDs, other than noticing that some Trip IDs are missing in each of the buckets (Figure 1).

The size of the training data was (1710670, 9), meaning that there were 1,710,670 observations and 9 features, explained later in more detail.

The test dataset trip IDs are labeled 'T1' through 'T327' and the seven missing (since there are only 320 entries in the dataset) are: 'T89', 'T97', 'T105', 'T106', 'T108', 'T150', and 'T165'.

2.2 Dimensions of Outputs/Inputs

Our goal is to predict the length of every single trip in the test dataset, so we want 320 outputs corresponding to a trip length in seconds (based on how many coordinate pairs exist in the "POLYLINE" variable). This means that we create a "length" feature for the training dataset as well.

So the ending dimensions without any feature changes are 1,710,670 rows x 9 (not including length) features for the training dataset and 320 rows x 8 (no polyline or length) features for the test dataset.

2.3 Meanings of Input/Output Dimensions

In the training dataset, there are nine provided input features:

1. trip_id: A unique trip ID.
2. call_type: A letter A, B, or C corresponding to how the taxi was called.

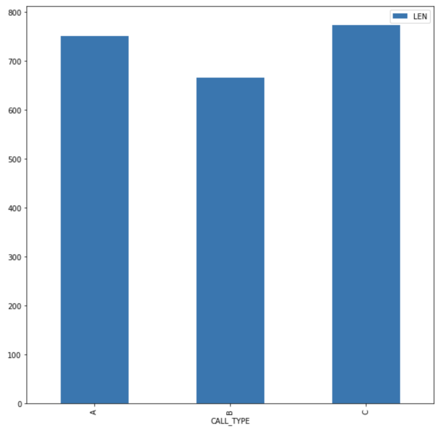


Figure 2: Average Length of Trip (in sec) by Call Type

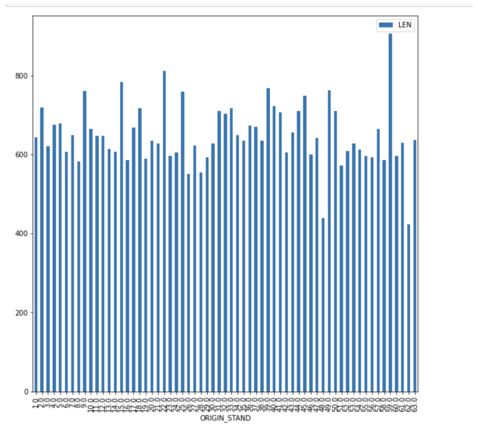


Figure 3: Average Length of Trip (in sec) by Origin Stand

3. origin_call: The caller's phone number if the taxi was called via telephone.
4. origin_stand: A taxi stand identifier if someone called the taxi from a stand.
5. taxi_id: A taxi identifier.
6. timestamp: A UNIX timestamp for the start of the trip.
7. daytype: A letter A, B, or C corresponding to normal days, holidays, and the days before holidays, respectively.
8. missing_data: A true / false value to indicate if the next variable has some missing information in the list of coordinates.
9. polyline: A list of coordinates taken every 15 seconds of the taxi's trip.

For the test dataset, all of these features are provided except for the POLYLINE variable, since that directly ties into the length we want to predict.

Doing early analysis of our training data using histograms, we made a few important findings.

Firstly, the length of trips in the dataset have pretty similar averages across two of the three call-types: dispatched from a central station (A) and hailed on the street (C). We wanted to include this information somehow in our prediction, as the type of call from a specific stand (B) tends to have slightly shorter trips on average compared to the other two (Figure 2).

We also made a similar histogram visualization for the average length of the trips in seconds across the 63 different taxi stands across the city (Figure 3). With this visualization, we found that there are certain stands that seem to have much shorter trips on average (48, 62) and others that have much higher average trip lengths (59).

Finally, we used histograms to analyze the overall distribution of lengths of trips in the dataset (Figures 4, 5, 6). We found that most trips are under 1 hour (in fact, most trips are under 20 minutes long), but there are a few trips that last several hours (the largest trip in the dataset is 16 hours long). We also analyzed the specific lengths (in seconds) of trips up to one hour in duration (Figure 7).

Once we evaluated the lengths of the trips, we used a different type of visualization to evaluate where taxis start their rides - a heatmap (Figure 8). In this figure, the latitude and longitude values were split into 10 bins each and were not defined by equal width in terms of distance of the coordinates, but instead by quantiles for the amount of data in each bin (using Pandas' method qcut). This heatmap demonstrates that there are very specific regular starting points for the taxis - the coordinate centered around [41.146, -8.612] is by far and away the most common starting point for the taxis, with a few other common pockets of ride starts.

To evaluate this intuition, we plotted the taxi stand locations and first Polyline coordinates together to see if these "hubs" generally line up or if there is a common location not represented by a hub (Figure 9). Generally what we find is that even though there are upwards of 60 different taxi locations, they

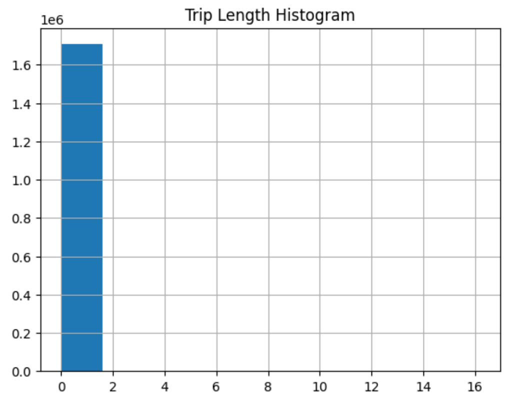


Figure 4: Count of Lengths (by Hours) for All Trips

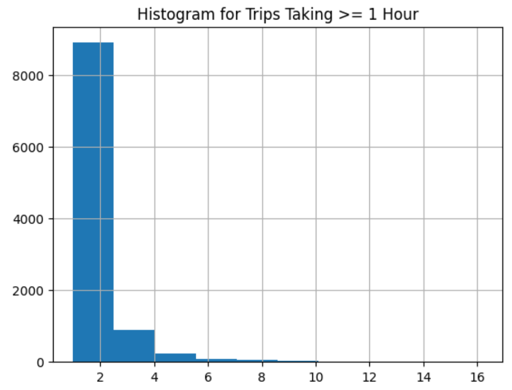


Figure 5: Count of Lengths (by Hours) for Trips >= 1 Hr

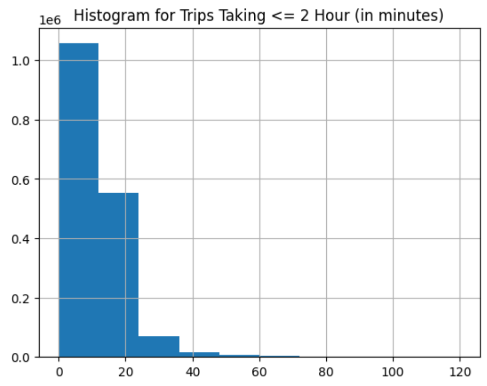


Figure 6: Count of Lengths (by Minutes) for Trips <= 2 Hr

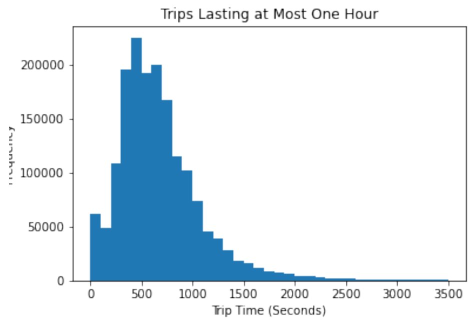


Figure 7: Count of Lengths (by Seconds) for Trips <= 1 Hr

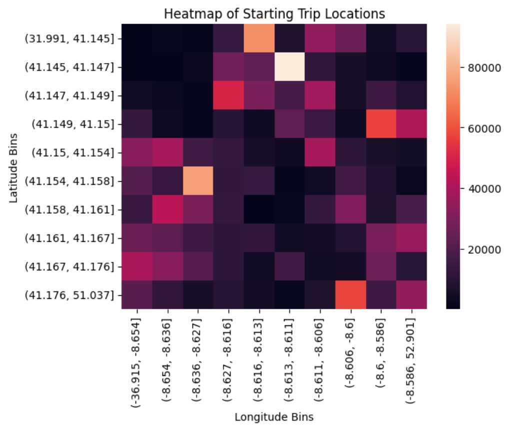


Figure 8: Heatmap of Starting Locations in the Training Dataset

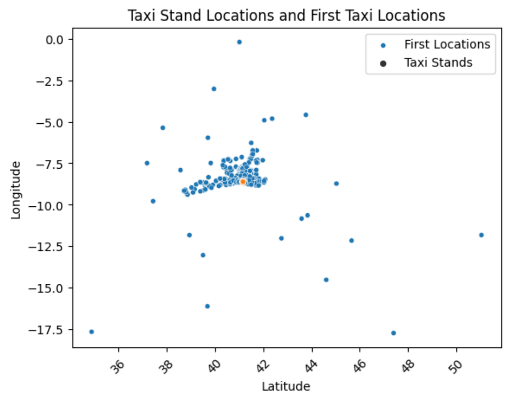


Figure 9: Taxi Stand Locations and First Locations

```
0_2013     0.0
0_2014     1.0
1_1        0.0
1_2        1.0
1_3        0.0
          ...
5_62.0     0.0
5_63.0     0.0
6_1        0.0
6_2        1.0
6_3        0.0
Name: OHE_series_example, Length: 143,
```

Figure 10: One trip represented as a Pandas Series post One Hot Encoding

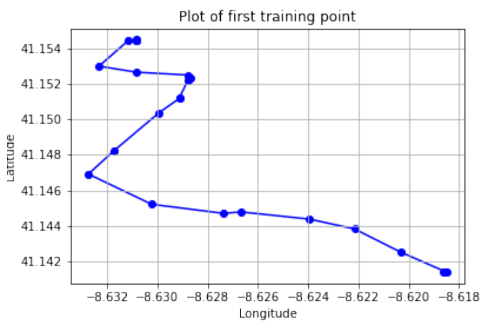


Figure 11: A sample polyline presented in two dimensions

all are clustered around the [-8.6109, 41.1455] coordinate we found earlier, so it is likely that many of the trips starting within those coordinates are from a single stand or a high traffic area.

We did not end up using all of this analysis in our final model, but we found it informative for our later feature engineering steps.

2.3.1 Data Sample Visualization

A visualization of one data sample are Figures 10, which shows the independent variables, and 11, which shows the polyline. For the independent variables, one hot encoding is used to create this vector in 143 dimensions representing the data. And the polyline plot shows an example trip that a taxi took, with the longitude and latitude displayed on the x and y axes.

2.4 Splitting Training and Test Data

We used a 90-10 train-test split after removing the 10 rows in the training dataset that had MISSING_DATA = True (we found it impossible to tell how much data was missing, ie. what the true length was). Our training dataset has 1,539,594 rows, and our validation set has 171,066 data points. We tried some versions of the model with a subset of rows and information such as:

- Taking out any length values that were at the 85th percentile or higher out of the training and validation set (or just the training set).
- Using a smaller subset of the training data as a whole (ie. 40% or 50%, which corresponds to about 684,264 or 855,330 rows).
- Training on data which we believed would be more representative of our test set, by doing things such as excluding months which weren't seen in the test data.

We also found that there were some interesting patterns in the test data, namely that only 5 months were represented, August, September, October, November, December. Additionally, only 5 unique dates were present: 8/14/2014, 9/30/2014, 10/6/2014, 11/1/2014, and 12/21/2014.

Of these days, we found that one was a Sunday (December 21st), another was the day before holiday (August 14th, the day before Assumption Day), one was the day after a holiday (October 6th is the day after Republic Day), and the rest were normal weekdays. We tried training our models on data which we thought would be more representative of the test data, sampling only from the months which were present in our data and choosing weekdays to weekends at a 4:1 ratio (200,000 weekdays, and 5,000 weekends, the number of days was chosen arbitrarily). But, this didn't really yield significant improvements, and we decided not to do this for our final model to avoid overfitting to the test data provided.

As such, most of the models we ended up running on the Kaggle and our most successful runs used the (mostly) full dataset.

2.5 Feature Engineering

We tried quite a bit of feature engineering, not all of which were included in our most successful models. Features we tried included:

- The year, month, day, hour, and weekday (parsed from the UNIX timestamp) from the feature engineering starter code. We ended up including these variables in our final model, as including them helped a lot with prediction.
- Additional time features for: the quarter of the day (midnight-6am, 6am-12pm, 12pm-6pm, and 6pm-midnight), the number of hours offset from noon, whether it was summer (based on June-September), whether it was a weekend or not.
- Starting location x and y coordinates, either based on the stand the taxi was called from or the most common starting location (if the taxi was not called from a stand).
- Average, min, and max lengths that a taxi has traveled (corresponding to taxi IDs).
- The average, min, and max lengths taxis travel depending on the origin stand, using overall values if the trip did not originate at a stand.
- A column to indicate if something was an outlier or not (based on the length values) for the training dataset.
- Caller average lengths and call frequency for each phone number.
- Ranking versions of the taxi average length (ie. 10 if a taxi tends to have a long average trip length, 1 if short), caller average lengths, and caller frequency.
- Average distances covered by taxi and by hour.
- Interaction features in string format: 'DAY-HOUR', 'ORIGIN STAND-HOUR', 'ORIGIN STAND-DAY'. These interaction features were created by concatenating the values of the categorical variables, with an underscore in between. Each of these concatenations in turn represents a unique pair of these categorical variables. For example, a trip taken at day 5 on hour 11 would take the form of "5_11". We then added these variables to the list of variables to use in our models (both the one-hot encoding model and the embedding-based model.) We saw marginal improvements in the loss on our training dataset, but the RMSE on our validation set stayed about the same, and the public score on Kaggle went down. While we believe that these features should be very strong predictors, we decided not to include them in our final model because we couldn't show empirically that they offered any predictive power.

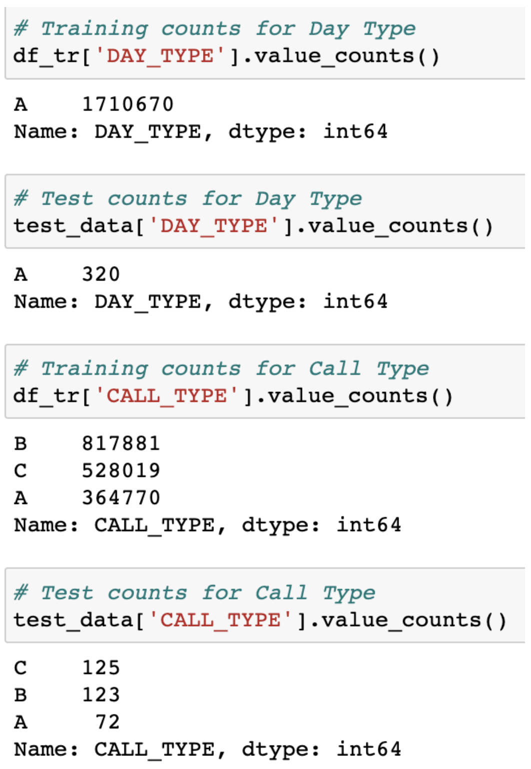
We also filled in null values in our dataset if we needed to for some of these more engineered features.

We then tried two different ways of modifying our categorical features: one hot encoding and embedding. Both of these approaches had ups and downs - one hot encoding was more intuitive and easier to develop, but took a significant amount of space. Embeddings were more space efficient and allowed us to run models faster but were more difficult to implement, and didn't seem to yield better results.

Our rationale generally behind our feature engineering was to take full advantage of the time features since they show up in both the training and test datasets, and then to find a way to include as much length/distance information as possible in the test dataset using the training dataset values since the test dataset has no way of getting any of that information otherwise. We also worked with interaction features because we wanted to represent unique relationships within a feature instead of outside of it, and we thought that the interactions we chose might be some of the more important ones to look at especially since they were likely more common (compared to specific minutes) and allowed us to connect time/time features and time/location features.

2.6 Normalization

For normalization, we did not really perform much normalization on the dataset as we initially used tree-based algorithms like random forests or decision trees to generate our results. As decision trees and random forests are generally insensitive to the scale of the features, we decided it was not necessary to use normalization. If we had more time on this project, we may have experimented



```
# Training counts for Day Type
df_tr['DAY_TYPE'].value_counts()
```

```
A    1710670
Name: DAY_TYPE, dtype: int64
```

```
# Test counts for Day Type
test_data['DAY_TYPE'].value_counts()
```

```
A    320
Name: DAY_TYPE, dtype: int64
```

```
# Training counts for Call Type
df_tr['CALL_TYPE'].value_counts()
```

```
B    817881
C    528019
A    364770
Name: CALL_TYPE, dtype: int64
```

```
# Test counts for Call Type
test_data['CALL_TYPE'].value_counts()
```

```
C    125
B    123
A     72
Name: CALL_TYPE, dtype: int64
```

Figure 12: Variety in Day Types and Call Types for Training and Test Sets

more with this. Additionally, almost all of the features that we used in our multilayer perceptron models were categorical (including all the features in our final model), so normalization was not really appropriate.

2.7 Utilizing Call Type

We did use the call type in our model because we realized it was generally pretty important for determining if a rider has traveled before (allowed us to define rider count/average lengths) or the starting location for a trip (based on if the call type indicated dispatching the taxi from central). We could've done even more with this variable, ie. creating different models based on call type, but we did not have time to implement this - we did want to ensure call type was included though.

What we did not include was the day type variable: all of the day type values in the training and test dataset were 'A' (corresponding a normal day). So we thought it would be redundant to include this piece of information considering there were entirely from the test dataset, and just removed it.

3 Deep Learning Model

Compared to all of the feature engineering we tried (some of which went into other models, such as the Decision Tree and Random Forest, and our embedding-based multilayer perceptron model), the features in our final model were pretty limited. We used a one hot encoding of the year, month, day, hour, call type, and stand of origin. The output of this model was just the length of the trip. In our final model(s), we did not try to predict any intermediate values to help with our travel time prediction as we initially tried to do (such as predicting whether or not a data point would produce a travel time which is considered an outlier).

Once we encoded these values (using Sci-Kit Learn's OneHotEncoder) and told the encoder to ignore values that did not show up in the training set, we used a more advanced Multi-Layer Perceptron as our final model.

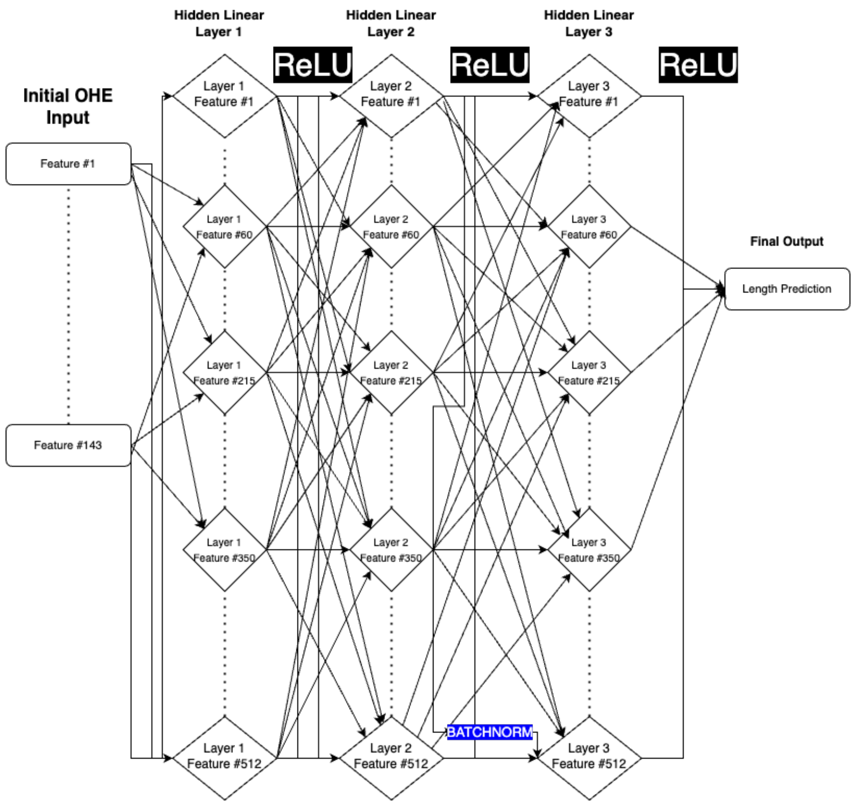


Figure 13: Final Model Architecture Flowchart

In our multi-layer perceptron model, we used a hidden size of 512 and sent the data through 2 linear layers of size 512, before applying a batch norm for regularization, then passing the data through another linear layer of size 512, and ultimately returning a single scalar representing the predicted travel time. The only regularization we used in our final models was the batch norm, which produced better results on our validation set compared to the dropout and other regularization methods we tried. Our loss function was the Mean Squared Error Loss function packaged with PyTorch, since RMSE should be minimized at the same location where MSE is minimized.

We ran a grid search to find hyperparameters for our model, testing batch sizes from the set [32, 64, 128, 256, 512], learning rates from the set [1e-2, 1e-3, 1e-4, 1e-5, 1e-6], and hidden sizes from the set [64, 128, 256, 512, 1024]. The differences in learning rates 1e-4 and 1e-3 were minimal but we found that on our model with one hot encoded features, the Adam optimizer performed best at 1e-3. We found that the model needed at least 128 nodes in the hidden layers to perform well, likely because our data was so high dimensional after the one hot encodings. But, the jump from 512 to 1024 didn't improve the RMSE on our validation set with any choice of other hyperparameters, so we decided it would be wiser to use 512 as our hidden size to speed up training.

We should note that we also tried a variation of this model with one of the hidden layers removed, which produced similar results. The train and validation loss from this model was almost identical, but interestingly the grid search revealed a different choice of optimal hyperparameters. Namely, hidden size of 1024, batch size of 128, and a learning rate of 1e-4. This was the other model which we submitted as our final model on the kaggle competition, and it scored very similarly on the public test data, reflecting its similar performance on the validation set.

Another thing which is important to note is that this model produced somewhat varied results, both on our validation set and on the Kaggle test dataset. We had some submissions with very similar hyperparameters which had much lower validation loss, and there was even more variance in our loss on the Kaggle data. We even produced one trained model which got 750 RMSE on the public test data, and 1061 on the private test data (good for 50th and 31st on the public and private leaderboards, respectively). However, we lost the file that had the weights for this model, and due to the apparent randomness in the results of our model after retraining, we weren't able to reproduce this loss before the competition closed. We suspect that this variance may be the result of very lucky train-test splits and shuffling of points within the dataloader, but we really can't explain such a great variance in RMSE, which has been one of our biggest shortcomings in this project.

We also tried a slightly more complex model based on embeddings, which we can explain more in part B.

The classes of models we tried were: a Decision Tree, a Random Forest, a one-hot encoding based multi-layer perceptron, an embedding-based multi-layer perceptron, and a single-layer LSTM.

- Decision Tree: max depth of 10, a minimum split number of 2, a minimum leaf size of 1, and a minimum weight fraction of 0. Embedding-based.
- Random Forest: 100 trees (default), max depth of 10, a minimum split number of 2, a minimum leaf size of 1, and a minimum weight fraction of 0. Embedding-based.
- One hot encoding MLP: 3 linear layers of size 512, with a batch norm between layers 2 and 3. Trained for 5 epochs.
- Embedding-based MLP: Embedding layer consisting of a separate embedding for each of the 12 categorical variables we passed in, 3 linear layers of size 128. Trained for 5 epochs.
- Single-layer LSTM: One hot encoded model consisting of 6 one hot encoded variables, batch size of 32, with a hidden size of 100 and a learning rate of 1e-3, and one dropout layer of 0.7 right before the output. Trained for 15 epochs.

Our final multi-layer perceptron model, as described above, involved only the one-hot encoded features mentioned in part A, whereas the best decision tree and random forest models uploaded to Kaggle used some additional features from feature engineering (specifically the starting coordinates, the caller average lengths/frequency, and the average lengths for taxi based on taxi ID and stand). The basic LSTM had the same one-hot encoded variables except for the call type (ie. year, day, month, weekday, hour, and stand of origin).

3.1 Decision Tree

A decision tree is a tree-like model where each internal node represents a feature or attribute, each branch represents a decision rule based on that attribute, and each leaf node represents the outcome or class label. The main parameter for a decision tree is the maximum depth, which controls the depth of the tree. Other parameters include the splitting criterion (e.g., Gini index or entropy) and pruning techniques (e.g., minimum samples per leaf).

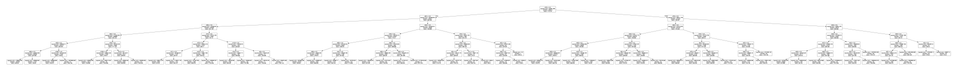


Figure 14: Decision Tree

We utilized Sci-Kit Learn's implementation of the decision tree, which ran on the CPU instead of the GPU (making it slower). Some of the hyperparameters we chose to modify for this project were the max depth of the tree, the minimum number of samples to cause a split, the minimum number of samples for a leaf, and the minimum weight fraction for the leaf. Based on extensive hyperparameter comparison and using Grid Search with Cross-Validation, we ended up using a max depth of 10, a minimum split number of 2, a minimum leaf size of 1, and a minimum weight fraction of 0, as this tended to do best with our validation set. While this model actually performed the best on our validation set, the public Kaggle score wasn't as good as our MLP models. Because of this, and our knowledge of the tendency of decision trees to overfit, we suspected that we may have tuned the hyperparameters in such a way that we ended up overfitting to our validation set. For this reason, we decided that the MLP's would provide a more generalizeable alternative, and decided not to further pursue the decision tree.

We could have also limited the number of features and max number of leaf notes, but we found that training the previous values on the CPU took an extensive amount of time and we wanted to devote time to other models, so we moved on from there.

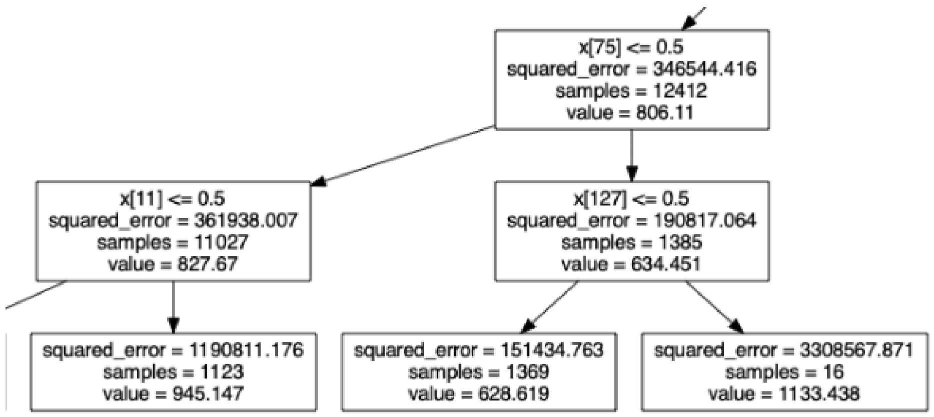


Figure 15: Close up of Decision Tree

When we briefly tried predicting whether a point would be an "outlier" before predicting specific trip lengths with separate models for the outlier vs non-outlier category, we also used Decision Trees with these parameters: one for the outlier prediction (which achieved about an 85% accuracy on the validation set, and one for each of the outlier categories - outlier vs non-outlier). This was pretty unsuccessful on our validation set's outlier class, so we did not end up uploading this version to the leaderboard.

3.2 Random Forest

A random forest is an ensemble model that consists of multiple decision trees. Each tree is trained on a random subset of the training data, and the final prediction is obtained by averaging or voting over the predictions of all individual trees. The parameters of a random forest include the number of trees in the forest, the maximum depth of each tree, the number of features considered for each split, and the criterion for measuring the quality of a split.

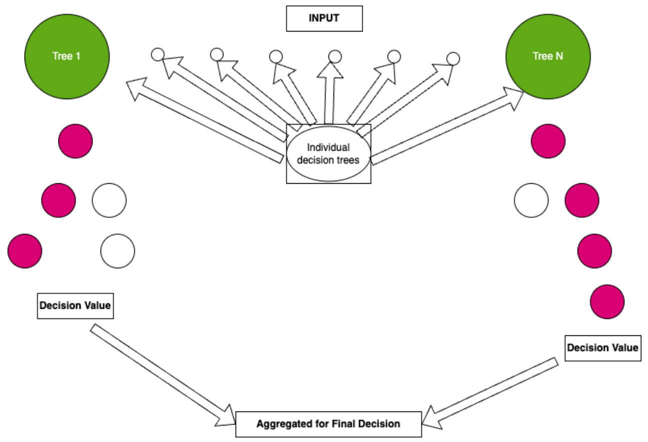


Figure 16: Generalized Random Forest Architecture

We also utilized Sci-Kit Learn's implementation for the Random Forest, and we utilized the same parameters from the Decision Tree with a max depth of 10, a minimum split number of 2, a minimum leaf size of 1, and a minimum weight fraction of 0. Along with the parameters we did not have time to train for the Decision Tree model, this model also added a parameter for the number of trees to use (we left it at the default 100) and the number of samples to draw from each tree. We again did not work to train these substantially because of time constraints - running 1 random forest regressor took the time of approximately 100 decision trees, so we moved away from this model and trying to modify the hyperparameters.

This model did end up doing better on the data than the decision tree even with similar parameters. We imagine this is because the increased depth from having multiple trees allowed it to handle some of the outlier cases that 1-3 decision trees alone struggled with.

3.3 Final Model - One Hot Encoding Multi-Layer Perceptron

An MLP is a feedforward neural network composed of multiple layers of interconnected nodes or neurons. It consists of an input layer, one or more hidden layers, and an output layer, all connected by activation functions after each layer. The standard activation function for an MLP is ReLU, where any values that are negative are replaced with a 0.

The parameters of an MLP include the number of hidden layers, the number of neurons in each layer, the activation function used, the learning rate, the optimization algorithm, and regularization techniques like dropout or L2 regularization.

We experimented with many parameters and regularization locations for this model, but in the end we settled on the parameters discussed earlier in 3A. With all of our neural networks, we used the Adam optimizer (which we discuss later in 4A).

3.4 Embedding-Based MLP

Our embedding-based MLP used different inputs and a slightly modified architecture. The inputs to this model were all categorical as well, namely: year, month, day, hour, week, origin stand, call type, taxi id, and the interaction terms created by concatenating origin stand with hour, origin stand with day, and day with hour. We also added another categorical variable called "TOD" indicating if the trip started in the morning, afternoon, evening, or night. From there, we passed these into an embedding layer, which created separate embeddings for each of the categorical variables, before passing this into an MLP network. The network had 3 hidden layers, all with ReLU applied. The first was a layer of size equal to the hidden size (128), the next had a size of hidden size (128) floor divided by 2, then the last one had a size of hidden size (128) floor divided by 4. We also applied a dropout layer after the last hidden layer, but before the output layer, before returning a single scalar as the predicted time. We tried this architecture because we believed that by making each consecutive layer smaller, we could encourage the model to learn more robust hierarchical relationships in the data. It did perform better than the 3-layer embedding-based model that we tried in which all layers were of the same hidden size, but ultimately didn't do as well as our one hot encoded model.

The best hyperparameters for this model in the end were a batch size of 256 (chosen to speed up training, and because the batch size didn't seem to have any effect on the validation or training loss), hidden size of 128, dropout probability of 0.3, and learning rate of 1e-2. We also determined these hyperparameters through grid search, trying the same sets of hyperparameters as before.

We trained the model for 5 epochs, as the validation loss appeared to gradually decrease until about this number of epochs had been run, at which point it began to increase again.

While we thought that this model should theoretically offer advantages over our one hot encoded model, as the embedding layer should already have learned a useful representation of the data, eliminating the need for more hidden layers within the network itself. Additionally, the embedding maps the data to a much lower dimension than our one hot encoding, which means there shouldn't be a need for as much width within the layers. However, this model did not perform as well as the one hot encoded model on the validation set, and even with extensive hyperparameter tuning and regularization, we weren't able to improve its generalizability. We do believe that this model could have been viable, but we suspect that it would take more extensive changes to the architecture of the model to get the performance that we would need, and we weren't able to accomplish this within the time we had for the project.

3.5 Long Short-Term Memory Network (LSTM)

For an LSTM, this model works best with time sequence data. This was one of the first models we tried (before the checkpoint for the project was due), which we tried because it was one of the most recent concepts we had learned about in class. The theory behind an LSTM is that we can use information stored from the previous time running the model to bring into the next iteration and so on, building up a system with memory. The architecture for a model such as this one is shown here in Image 17. This involves using a hidden set of values (h) to store the memory, along with managing a gate to choose what to forget (labeled f), a gate to take in new inputs (i), a gate to choose outputs (o), and a separate set of memory that acts as final input before deciding on the output (c).

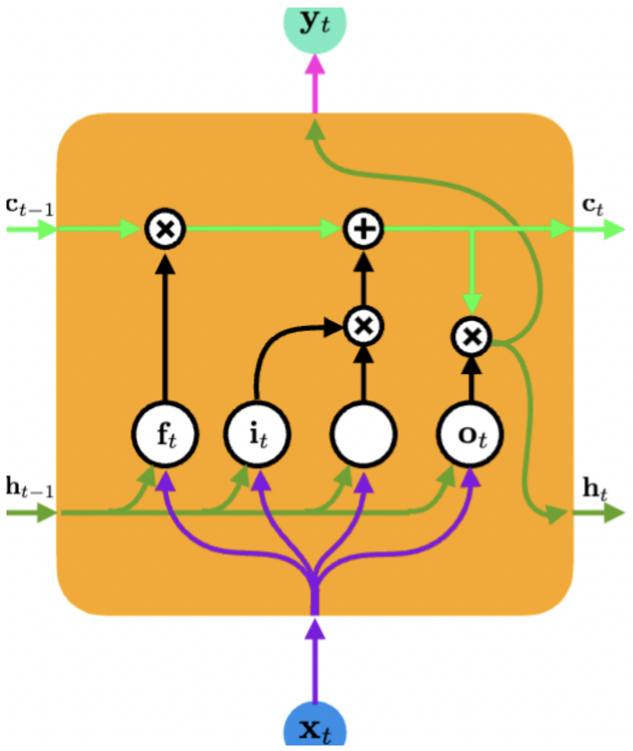


Figure 17: LSTM Architecture (Credit: Rose Yu, CSE151B Spring 2023)

In our implementation of an LSTM, we used six variables one hot encoded, ran it for 15 epochs with a learning rate of 1e-3, a hidden layer of size 100, a batch size of 32, one dropout layer of 0.7 right before the output.

We ended up moving away from this model, not because we believed it did not demonstrate potential, but that we could not make sense intuitively of how to effectively use the LSTM to its strongest use case with this data. The data in this problem is not inherently in sequence and with multiple taxis traveling at the same time, we thought that the history between each prediction did not make a lot of sense unless we were to train an LSTM for each individual taxi, or conduct some sort of sampling or aggregation. As a result, we abandoned this idea for the sake of time.

4 Experiment Design and Results

4.1 Problem A

4.2 Computational Platform

We exclusively moved to Datahub for our final project (instead of using the Colab/Datahub mix from previous). We mostly used the GPU (for any neural network models) and then used the CPU for training non-neural network models such as decision tree and random forest regressors.

4.3 Optimizers and Parameter Tuning

For our neural networks, we chose to use the Adam optimizer. This optimizer is known for computational efficiency and being good with sparse gradients, both of which being qualities we thought would be helpful for this problem since the dataset is large in size and the one hot encoded features result in a fairly sparse input tensor. We also tried using stochastic gradient descent, especially with our embedding-based model, as we believed that there was a more optimal solution which our model wasn't able to converge to just using Adam. But even after extensive hyperparameter tuning we couldn't get SGD to match Adam's performance.

Ultimately our final model used a learning rate of 1e-3. We did not utilize learning rate decay or momentum - these are other aspects that could've helped us improve our model, but we did not include because we were having trouble finding one model architecture that stood out above the rest, so we spent most of the time on this project trying different model architectures and features instead of using these parameters.

Other parameters we tried such as the number of layers and dropout and batchnorm, we had very inconclusive results with these. Our final iteration used 1 batch norm layer of the same hidden size between the second ReLU activation and the third linear layer. In many other models, we tried batch norms and they did not have the same lowering effect on the validation set so we are not sure why this model seemed to benefit from the batch norm more - our only potential idea behind this is that because the range of length values in the training set was very wide (see section 4C), this batch norm allowed the model to handle some of those outliers.

4.4 Epochs, Batch Size, and Training Time

In the end for our final model, we used 5 epochs, a batch size of 64, and our full model training time was approximately 5 and a half minutes.

As we stated in question 3, we ran a grid search to find hyperparameters for our model, testing batch sizes from the set [32, 64, 128, 256, 512], learning rates from the set [1e-2, 1e-3, 1e-4, 1e-5, 1e-6], and hidden sizes from the set [64, 128, 256, 512, 1024]. The differences in learning rates 1e-4 and 1e-3 were minimal but we found that on our model with one hot encoded features, the Adam optimizer performed best at 1e-3. We found that the model needed at least 128 nodes in the hidden layers to perform well, likely because our data was so high dimensional after the one hot encodings. But, the jump from 512 to 1024 didn't improve the RMSE on our validation set with any choice of other hyperparameters, so we decided it would be wiser to use 512 as our hidden size to speed up training.

For the epochs, in many of our models we ran into an issue where the loss would decrease fairly quickly on the first two epochs and then very slowly for every subsequent epoch (to the point of making almost no impact on any validation set, and even increasing the validation loss after only a few epochs). This meant that for many of our models, we ended up leaving the epoch numbers low, ie. 5-10 epochs which allowed us to try more models in the time we had. Once we chose our final model, we ended up training it for 5 epochs again because we had the same experience. This near-immediate convergence of the loss was the largest obstacle that we ran into, and seemed to be an issue with nearly all of the models that we experimented with.

Table 1: Model Comparison

Model	Parameters	Val. RMSE	Kaggle Test RMSE	Train Time Est. (1 epoch)
Decision Tree	See 3.1	636.36	777.93	45 sec.
Random Forest	See 3.2	$\sim$628	772.39	10 min.
Final (OHE) MLP	See 3.3	641.69	767	70 sec.
Embedding-based MLP	See 3.4	686.42	786	45 sec.
Early LSTM	See 3.5	739.40	790.26	106 sec.

(We had to cut the parameters from being in this table for the sake of space, but please view the reference sections to review what they were.)

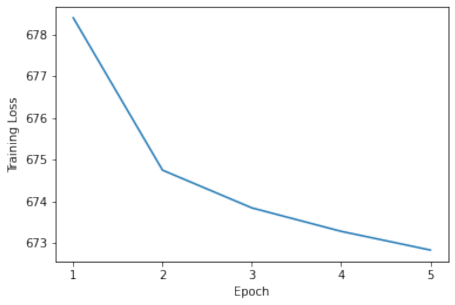


Figure 18: Training Loss over 5 Epochs

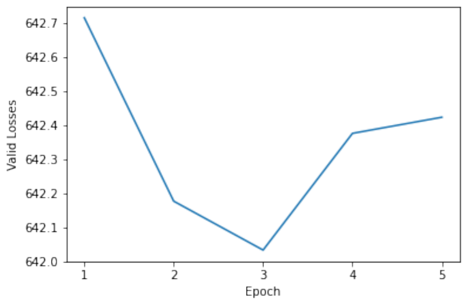


Figure 19: Validation Loss over 5 Epochs

From this table, we find that our more successful approaches were not always deep-learning based, and that for the minor variations in RMSE we got, often it was more worth it to focus on faster training models than more complex models, as that allowed us to experiment more and improve hyperparameters (which we found to be very important). If we had seen more significant increases in the scores for our more complex model, we might have chosen those instead.

Our best models also varied between one hot encoding over embeddings, which was a surprise to us: we thought that one hot encoding was a simplistic way of representing different variables and that embeddings could help us work on representing more relationships between. Our embeddings on random forest data seemed to do well, but one hot encoding for the perceptron was definitely better.

Once we fully moved over to one hot encoding however, we did run into issues with storage space that prevented us from trying more complex models (especially when we attempted to use more features). This made our training time slower along with filling up our RAM. To improve our training speed, we tried to evaluate the most important features (for a decision tree model, using Sci-Kit Learn's built-in feature_importances_ parameter as a method of evaluation in one case) and then dropped them, which is how we ended up with a small subset of features for our final model.

4.5 Visualization for Best Performing Model

See Figures 18 and 19.

4.6 Visualization for Ground Truth V. Samples

See Figures 20 and 21.

In Figure 21, some residual values were over 60,000 seconds ie. 16 hours, which made it impossible to see the rest of the distribution in one plot without cutting the larger residuals out.

With these two figures we see that this model really struggles with representing the full range of values that the real data represents, as there are large outlier trips that the model cannot account for.

4.7 Final Leaderboard and RMSE ranking.

Our best test RMSE on the final (public leaderboard) was 750.02345 - however, we submitted a Piazza post because this model was not the one we submitted (the weights for this one were not saved correctly, although it was produced by a model with the same architecture and hyperparameters as the one that did produce our submitted predictions).

For the model we selected for submission as our final model, we achieved a public test RMSE of 767.38 which placed us in approx. 68th position on the leaderboard at the time of writing this, Friday, June 9th at 2:05 PM.

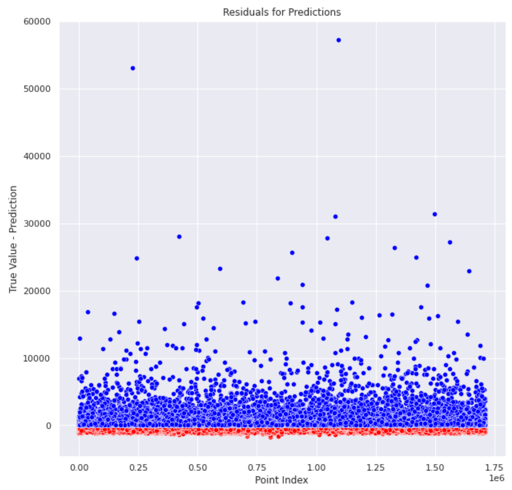


Figure 20: Residuals for Validation Set Actuals - Predictions

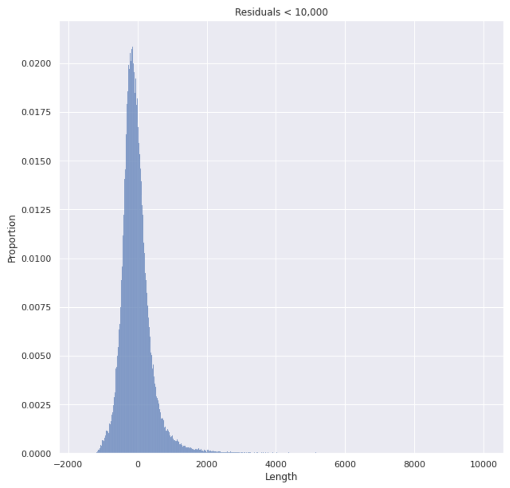


Figure 21: Residual Histogram (Residuals Capped at < 10,000)

model_notebook_test.csv
Complete · Oren Ciolli · 2d ago
767.37658

Figure 22: Final RMSE Value (Fri, Jun 9th at 2:05 PM)

5 Discussion and Future Work

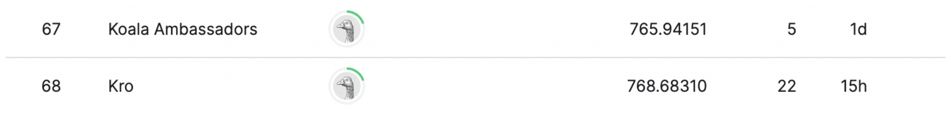
Our results in this project were very disappointing to us. For most of the time while working on this project, we hit a ceiling at a public test RMSE of 767, even though we had validation root mean squared errors that ranged from 224 through 680 (with a much larger validation dataset). We'll now take some time to reflect on the strategies we used and potential improvements which we didn't get the chance to implement before the competition ended.

5.1 Most Effective Feature Engineering Strategy

We found that the most effective feature engineering strategy was to focus on timeseries features and extracting the nuances of the timestamps to account for different trends across time of day, seasons, and days of a week since it heavily affects taxi traffic at the time and where riders are trying to go. There are lots of pieces of information in a single timestamp, so separating these and using them both individually and in interaction pairs is important. Adding interaction terms was probably the next most important feature engineering strategy, as certain variables may have very different effects on the output depending on the values of others. While our best model (on the public test) actually didn't use interaction terms, our next best model did, and we think that the theoretical advantage of these terms is worth mentioning even though it didn't improve our public loss by much.

5.2 Most Effective Technique for Improving Rankings

Our most effective strategy for improving our ranking (for the small improvements we achieved) was hyperparameter tuning. We tried multiple model designs/architectures and adding many features to add to our dataset, but ultimately, all of the models (with different features) seemed to converge to the same RMSE. So, turning to hyperparameter tuning for all of our neural network models and decision tree models allowed us to make small strides towards making the test error as small as we could.



67	Koala Ambassadors	765.94151	5	1d
68	Kro	768.68310	22	15h

Figure 23: Final Leaderboard Scores (Fri, Jun 9th at 2:05 PM)

5.3 Biggest Bottleneck

We had many bottlenecks in this project. The main one that we ran into regularly was the size and power of our computational resources through Datahub (and Colab/our local machines at times we used those before doing the final steps for this project). We had several models that looked like promising options, such as a larger neural network with a larger, sparser dataset, a KNN Regressor on a large one hot encoded dataset, or a set of models to predict "outliers" and then predict trip length based on the already-predicted outlier label. However, all of these models ran into issues with space (e.g. CUDA memory errors for tensor-based models or dying kernel (CPU memory issues) for Sci-Kit Learn based models). The other major challenge we encountered during our project was a bottleneck resulting from our lack of theoretical knowledge of deep learning model design. The lack of familiarity with deep learning best practices led to inefficiencies and suboptimal performance, as we had trouble thinking of new, creative ways to improve our models like we're used to doing with other machine learning models. This bottleneck issue significantly impacted our progress, as we had to spend considerable time researching and experimenting to overcome these hurdles. This experience really proved the importance of investing time in gaining expertise with the theory behind models used in a project to minimize such impediments and optimize our development process.

5.4 Advising a Beginner for a Similar Task

We have a few pieces of advice that we'd give to a beginner working on a similar task to this one. Firstly, deciding on a model architecture and understanding the logic behind it early is very important. This provides a solid foundation for any modifications to the model that you make later, as otherwise you may not understand why you are implementing your changes or why certain changes are more successful.

Another piece of advice we would give to a beginner is to plan to invest a lot of time into hyperparameter tuning. We tried multiple models throughout our process, but ultimately our most successful avenue of exploration ended up being a model that we had submitted a long time ago that we kept trying extensive parameter training on to eventually get a higher validation and test score. There can be a lot of potential hyperparameters to keep track of with a model like this, so make sure to keep a list of what you've tried with which features and why those values were changed (what the logic is behind the changes).

Finally, ensure that you are familiar with the capabilities of the tools you're using. We struggled with data storage and with Pytorch, and if we had more familiarity with Pytorch before the project began, we likely could've optimized for the storage we had access to by using more efficient tensors, batches, and dataloaders. Taking time to work on fundamental skills to troubleshoot issues you may run into while working on your model will be very valuable when something is not working as expected and you need to make a quick fix.

5.5 Other & Future Ideas

We believe that an avenue to explore would be to implement either a more advanced LSTM with datapoints arranged by time (similar to some of the top performing groups) or a CNN based on the fact that spatial data is present in the polyline grid. By employing a CNN architecture, the model could learn hierarchical representations of the taxi trajectories (which are inherently spatial), capturing both low-level details and high-level global patterns. If we were to be given more resources, we definitely would ask for a stipend to work with clusters. On multiple attempts, our runtime was timed out when using XGBoost and our GridSearchCV. We also considered using ensemble methods such as implementing many different networks, each one perhaps with different architectures and taking different features as inputs, to produce a set of predictions which would then be aggregated somehow to produce a single value. This is obviously more computationally intensive (and theoretically advanced) than what we were able to do in this project, but it's something that we find interesting and may be worth exploring. Another idea we were interested in is applying some sort of dimensionality reduction such as PCA in conjunction with our networks. We were limited in time, and when we realized we probably reached the limit of features we could encode using one hot encoding, we had the choice to focus on this dimensionality reduction with a high dimensional one hot encoded tensor (like the one we used in our final model) or switch to using an embedding layer. We opted for the embedding layer, but we think it would also be interesting to try dimensionality reduction with the very high dimensional onehot encoded tensor resulting from including more of our engineered features and interaction terms.

References

[1] Kingma, Diederik P. & Ba, Jimmy (2014) Adam: A Method for Stochastic Optimization, *3rd International Conference for Learning Representations, San Diego, 2015*.

[2] Yu, Rose (2023) RNNs 05-2 *CSE151B: Spring 2023 Version*.