MIKEL BOBER-IRIZAR – MLND CAPSTONE PROJECT

PREDICTING NYC TAXI TRIP DURATION

1. DEFINITION

1.1 PROJECT OVERVIEW & INTRODUCTION

Taxis are a vital part of the ecosystem of many cities. In New York City alone, over 300,000 yellow cab trips are taken every day, in addition to almost 300,000 ridesharing trips. Due to the advent of ridesharing apps such as Uber, the total number of trips taken has almost doubled in the last two years.

To be able to operate a taxi service at such a large scale, companies use electronic dispatching systems to efficiently assign cabs to customers and spread out cabs across cities where they might be needed in order to maximize the number of rides each cab can take in a day.

One vital aspect of such systems is to predict how long a taxi trip will take once a cab is taken, so the system can understand how long a specific cab will be removed from the pool of free cabs and when it will be free to pick up more passengers. This allows dispatchers (and automated dispatch systems) to reduce passenger waiting times and increase revenue from each cab, creating a win-win situation.

Kaggle, the well-known competitive machine learning platform has hosted several competitions to predict the duration of taxi trips, such as the ECML/PKDD 2015 competition which used data collected from taxi trips in Porto, Portugal. In order to build an algorithm that can accurately predict the duration of a taxi cab trip, I will be using the dataset provided by the recently launched and ongoing New York City Taxi Trip Duration competition, which is a very large dataset released by the NYC Taxi and Limousine Commission.

1.2 PROBLEM STATEMENT

The Kaggle dataset consists of data collected from New York City over a period of 2009 to 2016 – in total, it contains information from over two million rides.

The goal of the problem is to predict the duration in seconds that a given taxi trip will take – making it a supervised regression problem. My capstone project will focus on trying to solve this problem and getting a maximal score on the Kaggle leaderboard based on the metrics described in the following section.

As inputs, we are given information such as the company which runs the taxi, the starting and ending locations, and date/time information – this is the information that we need to predict ride duration from. As the data given is tabular in nature, my approach will be to use standard supervised regression algorithms such as decision trees, support vector machines or linear regression.

Overall, I will approach the problem in multiple steps. Firstly, I will explore and visualize the data to gain an understanding of the features and where the 'signal' in the dataset lies. This will allow me to go onto feature preprocessing, where I will convert the features into formats more suited towards the regressors I will be using, as well as performing 'feature engineering', a term commonly used on Kaggle which refers to creating entirely new features which may be more predictive out of the existing features.

After I have created my final feature space that I am happy with, I will build a regressor upon the data to obtain a score. When I have obtained my classifier, I will perform parameter tuning to maximize the score obtained on a validation set, and then use the best model to create final predictions which I will then upload to the Kaggle leaderboard to obtain my final score.

I anticipate that my final solution will consist of a feature processing pipeline followed by a single trained supervised model which outputs predictions for the test set that can be uploaded to the Kaggle leaderboard.

1.3 METRICS

To measure the performance of the model, I will use the Root Mean Squared Logarithmic Error metric (referred to as RMSLE).

The RMSLE is defined on the Kaggle evaluation page as:

$$\epsilon = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\log(p_i + 1) - \log(a_i + 1))^2}$$

Where:
N is the number of samples,
p_i is the prediction of the trip duration,
a_i is the actual duration
$\log(x)$ is the natural logarithm of x.

This metric is **identical to Root Mean Squared Error (RMSE),** which is widely used for regression problems, **except that** the "log1p" of both the target and predicted values are taken

beforehand. `log1p(x)` is defined as `log(x + 1)` - the +1 is there to avoid taking the log of 0 (which is undefined).

This parallelism to RMSE is very useful, because it means that we can make a learner that optimizes RMSE (most out of the box regression algorithms) instead directly optimize RMSLE by taking the log1p of the target value beforehand and feeding that to the learner in place of the original target.

There are two reasons I have decided to use this metric. Firstly, this is the official metric that we need to optimize for the Kaggle competition and the metric upon which the submissions to the competition are ranked, so it makes sense to also try to optimize this locally.

In addition, I believe using the log-error also makes more sense than directly using the error for each trip. This is because we care more about the error in each sample **relative to the trip time** as opposed to simply the absolute error.

For example, one would say that a 5-minute trip incorrectly predicted as 2 minutes is much worse than a 50-minute trip incorrectly predicted as 47 minutes. If the RMSE metric was used, these two trips would have the same error. However, RMSLE would penalize the 5-minute trip more, even though both predictions were incorrect by three minutes. This seems more reasonable to me than penalizing them both equally, so for this reason I think RMSLE better represents what the model is actually trying to learn.

2. ANALYSIS

2.1. DATA EXPLORATION

All code for this section and section 2.2 is in `data_exploration.ipynb`

As part of the dataset from Kaggle, we are given two files, a `train.csv` and a `test.csv`. These two files represent the training and testing data given by Kaggle – the formats of the files are identical except for the fact that the testing data does not have target values included.

The training set contains 1,458,664 trip records, while the test set contains 625,134 records, making this a rather large dataset. Each trip record is represented by a row in the csv file, and has several features given for it. I have described each feature given briefly below:

Feature	**Description**
`id`	The ID of the trip. Not to be trained on
`vendor_id`	A categorical variable indicating the taxi provider associated with the record

`pickup_datetime`	The date and time that the taxi meter was engaged (and the passenger was picked up)
`dropoff_datetime`	The date and time that the taxi meter was disengaged. **This feature is only present in the training set.**
`passenger_count`	The number of passengers in the vehicle
`pickup_longitude`	The longitude of the passenger pickup location as a float
`pickup_latitude`	The latitude of the passenger pickup location as a float
`dropoff_longitude`	The longitude of the passenger's destination as a float
`dropoff_latitude`	The latitude of the passenger's destination as a float
`store_and_fwd_flag`	Whether the trip was "store and forward", meaning that the vehicle did not have any connection to the server during the trip and trip details were uploaded later. Denoted by "Y" or "N" values
`trip_duration`	The total duration of the trip in seconds. This feature only appears in the training set and is the **target value**.

This means we have a total of 8 features that we can train on, which includes one categorical, one Boolean and one timestamp feature, the rest being float-valued.

	passenger_count	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	trip_duration
count	1458644.000	1458644.000	1458644.000	1458644.000	1458644.000	1458644.000
mean	1.665	-73.973	40.751	-73.973	40.752	959.492
std	1.314	0.071	0.033	0.071	0.036	5237.432
min	0.000	-121.933	34.360	-121.933	32.181	1.000
25%	1.000	-73.992	40.737	-73.991	40.736	397.000
50%	1.000	-73.982	40.754	-73.980	40.755	662.000
75%	2.000	-73.967	40.768	-73.963	40.770	1075.000
max	9.000	-61.336	51.881	-61.336	43.921	3526282.000

From the statistics above we can see that the (latitude, longitude) pairs are clustered in a very small region around (-74, 40) – this is expected since that is the location of New York. However, there are some very large outliers thousands of miles away – these are most likely GPS errors. A similar case can be seen in the `trip_duration` target variable: The max of this variable is equal to just over 40 days, which is obviously an erroneous measurement. Such outliers are to be expected in such a large dataset. However, they are not very common – for example, there are only 82 samples out of 1.5 million which have abnormally low longitude values, so these anomalies are likely to be ignored by most machine learning algorithms.

The categorical `vendor_id` variable contains only two possible IDs, with 47% of the taxi trips containing the vendor ID '1' and the rest containing '2' – this variable should instead be treated as a simple boolean variable. The `store_and_fwd_flag` variable is also a boolean, but is only positive in 0.5% of taxi trips, showing a rare event. It's unclear how this variable could affect ride times.

2.2 DATA VISUALIZATION

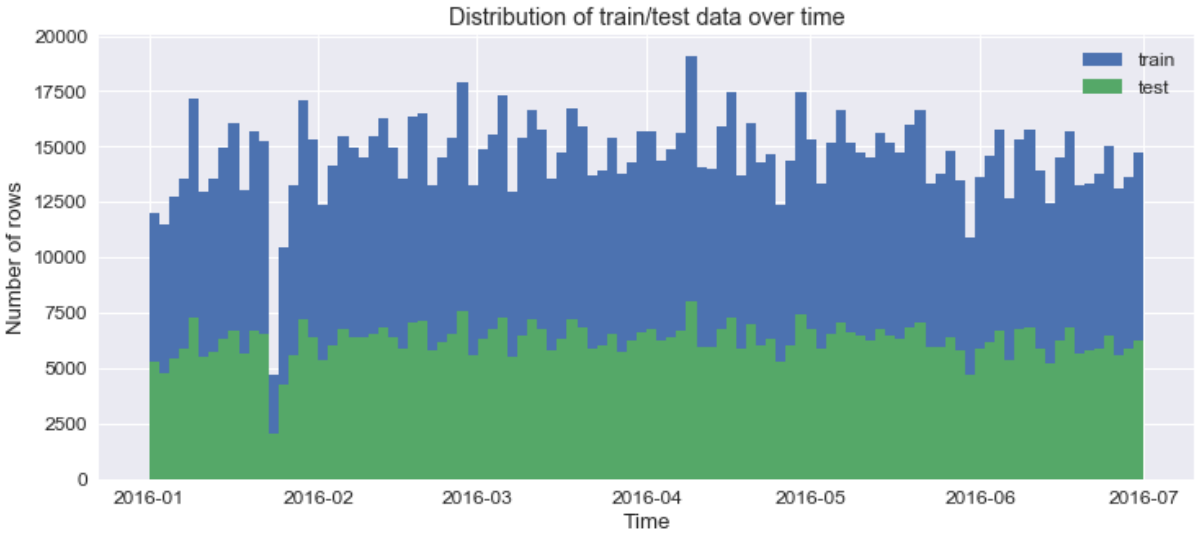
In this section, I will visualize and analyse some interesting aspects of the dataset given to gain a better understanding of different aspects of the data. The information gained from this is vital in order to figure out what feature engineering may help extract signal from the data, as well as what types of models may work well on the data.

I hypothesise that this problem has a substantial aspect of time-dependence, meaning that the distribution of the target changes a lot with respect to the time of day (traffic conditions may change, for example), as well what day of the week or year it is (special occasions may affect taxi trips, for example).

For this reason, I think it is important to investigate how the pickup datetime affects the data. One important aspect is to look at how the split between the training and testing sets are done. There are two different ways this could have been done:

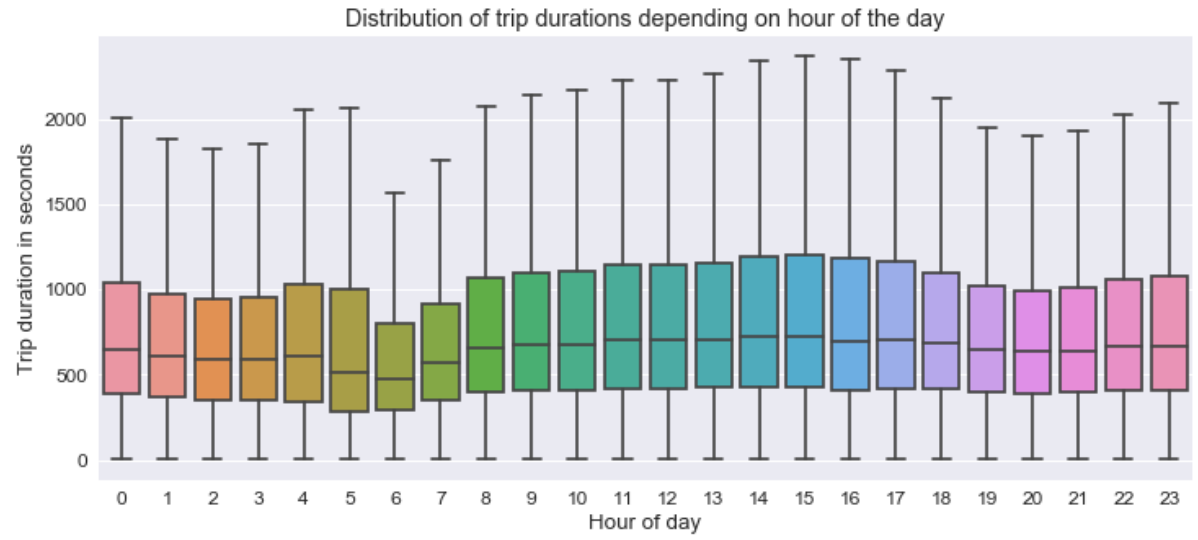
1) The testing data is sampled from the **same period of time** as the training data. This would allow us to get a lot more information out of the training set – for example, when predicting on a sample in the test set, we could look at trips made around the same period in the training set to understand the current traffic situation.
2) The testing data is sampled from a period of time **in the future** (relative to the training data). This creates a very different problem: instead of trying to predict a subset of the taxi rides around the same time, the challenge is instead about creating a model which could predict taxi trip time on a future day (where the outcome of recent taxi trips is not known). This option would make more sense for the data, as taxi companies are not

interested in predicting on past data but instead obtaining a model that can tell them how long future rides will take.



From the above plot, we can observe that the dataset is relatively uniformly (apart from an unusual dip in in January) distributed throughout the first half of 2016 (183 days in total). More interestingly however, both the training and test sets occupy the same distribution, which shows that the first approach described above was taken in preparing the data. This means that we are actually given target values for other taxi trips around the same time period of each test trip, and these can be used to better predict the trip time.

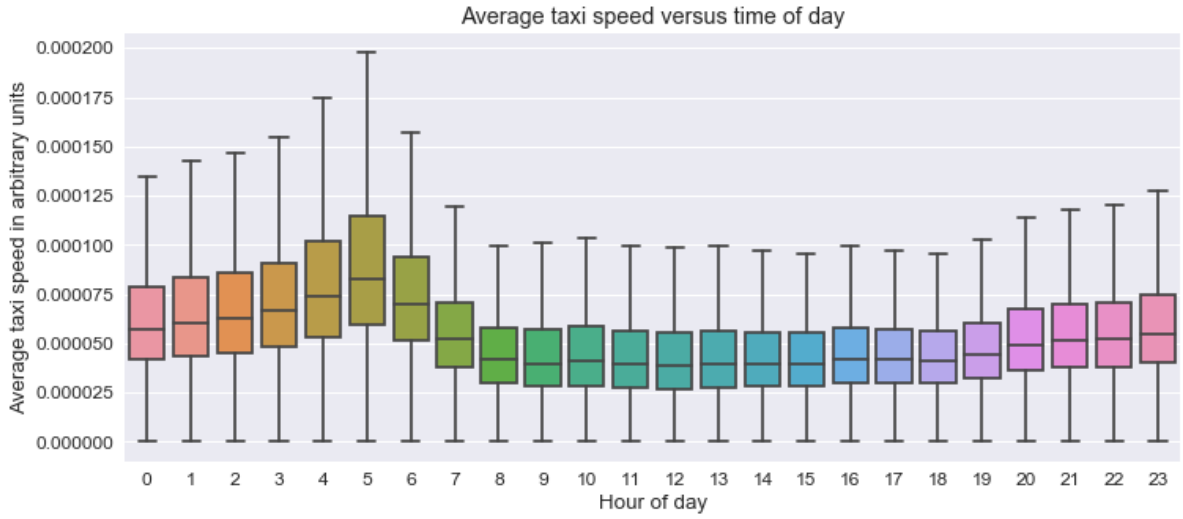
We can also look at how the trip duration changes based on the time of day using a boxplot:



This graph shows that the duration of taxi trips varies quite dramatically based on the time of day. In the early morning (5-8am) and to a lesser extent in the evening (19-21), customers tend

to take quicker trips than during other parts of the day. At 8am and onwards, the average trip duration tends to get longer. I believe this is due to increased traffic conditions at peak times meaning that taxis cannot travel as fast. However, to rule out the possibility of people simply travelling further during the day, I need to analyse the speed of the taxis directly.

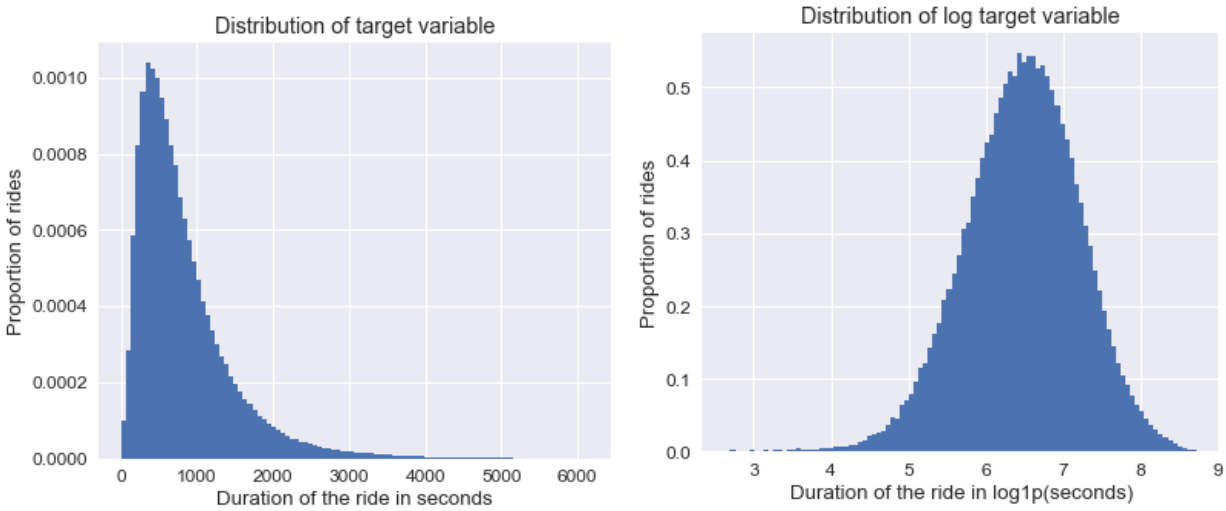
Since I am only given co-ordinates of the pickup and dropoff points, we are not given any information about the speed of the car. However, this can be approximated and inferred from other information: For this analysis, I will use the Manhattan (L1) distance between the two points and divide this by the trip duration to get a rough estimate of average speed. If we then plot the calculated speed using another boxplot, more interesting observations arise.



Here, the effect is even more pronounced. We can see that by my speed metric, the early morning is the time at which taxis travel fastest, and this slows down very substantially (average speed roughly halves) beginning at around 7am onwards. In fact, we can see that these observations almost perfectly match data released by the NYC Taxi Commission in 2013, which put the peak speed at 5:18AM and showed the same relationship for the rest of the day.

From the above analysis, I can see that time has a very large effect on taxi trip duration. For this reason, I will focus on building features that can capture this relationship (as well as the relationship with taxi speed) in the modelling phase, which should help predict trip duration at different times of day better.

In addition, we can look at the distribution of the target variable to understand what we are trying to predict:



From this, we can see that taking the log of the target value makes it much more normally-distributed. This should make it easier to be modelled by a regressor versus the long-tailed distribution of the original target value (where very large outliers could cause the model to overestimate other samples).

2.3 ALGORITHMS AND TECHNIQUES

XGBoost

For training models, I have decided to use the XGBoost algorithm. XGBoost is an implementation of a machine learning algorithm known as Gradient Boosted Trees (GBT), also referred to as gradient boosting machines (GBMs). The concept of this algorithm is as follows:

1) Fit a decision tree to the data
2) Evaluate the model against each data point
3) For each sample in the dataset, increase its weight based on how incorrect the model was in the last step
4) Fit another decision tree to the data using the reweighted data – this new tree will fit more to the areas where the previous trees did not perform well
5) Add this new tree to the ensemble, go to 2

The result of this is an algorithm which has the benefits of other tree ensemble models (such as random forests) while outperforming other similar techniques in performance.

I have chosen to use this algorithm for multiple reasons:

- Since it is a tree-based algorithm, it is completely scale-invariant. This means it can very easily handle features with bad scales, such as latitude and longitude, reducing the amount of preprocessing that needs to be done.
- Also due to its tree-based nature, it can learn non-linear relationships as well as relationships between features. This allows it to directly use the latitude and longitude features – it can learn different patterns for different areas of New York by splitting on these features where a linear model would not be able to. For example, using four splits in a branch the model could check if the car is located in a specific rectangle of the latitude and longitude.
- XGBoost is widely used (and often wins) in Kaggle competitions as it is known to tend to outperform similar supervised models extremely well on tabular data with a range of complex non-linear features, as we are given here.

The downside of using XGBoost is that it can potentially be prone to overfitting. However, this is much less of an issue when large datasets are present (as is the case here), and this can also be mitigated by using a validation set. The model can be evaluated on the validation set after every tree is added and 'early stopping' can be used to stop training the model when it stops improving on the validation set, lest it overfit.

When training, XGBoost takes in a matrix of (samples, features) and a vector of target values, returning a model. This model can then be used on another matrix of the same format (the test set) to return a vector of predicted target values. XGBoost only supports numerical input features, so features such as datetime and categorical variables will need to be transformed beforehand.

There are a few basic XGBoost parameters that need to be tuned to obtain optimal results:

Parameter	**Default**	**Explanation**
`max_depth`	6	The maximum depth (number of splits in a branch) of each individual tree in the ensemble.
`colsample_bylevel`	1.0	The proportion of input features which are available to the model to split on at each level of the tree. Decreasing this means trees are more likely to be different, which can improve ensemble performance.
`subsample`	1.0	The proportion of the data which each tree is trained on. Decreasing this has a 'bagging' effect which works for the same reason as for `colsample_bylevel`.
`min_child_weight`	1.0	The minimum amount of "weight" a tree node must have to be included. "weight" loosely means how much it contributes to the model. Increasing this can reduce overfitting.

`eta`	0.3	The learning rate (amount of reweighing between trees). Decreasing this always helps performance, but learning takes much longer for diminishing returns.

2.4 BENCHMARK

Arguably the simplest benchmark which can be used to compare between models is the performance obtained by always predicting the same value (this can be considered equivalent to a prior probability). In this case, I will find a single trip duration that minimizes RMSLE on the training set, and then use this to create a submission on the Kaggle leaderboard.

The optimal trip duration estimate can be found with:

$$optimal\ trip\ duration = expm1(mean(log1p(trip\ durations)))$$

This yields us a trip duration estimate of **642.54 seconds**, which gives us a **0.796 RMSLE score** on the training set and a **0.798** score on the Kaggle public leaderboard.

Thus, we can say that for a model to have learnt anything less trivial than the prior of the data, it must have a RMSLE error of less than our benchmark **0.798** on the leaderboard.

3. METHODOLOGY

3.1 DATA PREPROCESSING

To convert the data into a format which XGBoost can effectively use, as only numerical features are supported, some of our input features must first be transformed.

`store_and_fwd_flag`

The first feature that needs to transformed is the `store_and_fwd_flag`. This feature is a boolean, comprising of the possible values "Y" and "N". To transform this feature, I simply map "Y" to a 1 and "N" to a 0, which means it can be digested by XGBoost.

`pickup_datetime`

This feature is slightly more complicated to preprocess. It is given as a string in the form of "30/06/2016 23:59:58". To preprocess this feature, I do several transformations resulting a set of new features based on this time. First, I use the `pandas` package to parse the string into a Timestamp object from which I can extract useful features. From this data, I extract the following features:

- **The Unix epoch time** (number of seconds passed since 1970): This is useful for the model as it allows the model to learn patterns over longer periods of time in the dataset (for example, some seasonal effect, or a gradual decrease/increase in speed over the period of the dataset).

- **The minute of the day:** This is calculated using [`hour*60+minute_of_hour`] and is useful because it allows the regressor to model daily speed patterns (such as how the taxis move fastest at 5am).
- **The day of the week:** Represented by an integer, this feature allows the regressor to model different patterns during different days of the week – I expect traffic conditions are substantially different on weekends, so this allows the model to distinguish between conditions on weekdays and weekends.

`vendor_id`

Although this feature is technically a categorical feature, it does not require preprocessing, such as one-hotting to create a separate feature for each vendor. This is because there are only two vendors, so the feature can be treated as a boolean feature representing "vendor `1` vs not vendor `1`", as discovered in the Data Exploration section.

Dropped features

I have decided to drop a few features altogether from the dataset. The `id` variable simply identifies the row and does not provide any information relevant to the trip, so I have removed this feature from training. The `dropoff_datetime` variable has also been removed because it is only present in the training set, and does not provide any new information as it is simply equal to `pickup_time + trip_duration`.

FEATURE ENGINEERING

Another important part of my approach is "feature engineering" – the practice of making new features that allow the algorithm to model aspects of the data it could not easily model before. Tree-based models such as XGBoost are unable to model mathematical operations such as addition and subtraction between variables. This means some signal in the data which could provide a large boost to performance simply cannot be modelled: for this reason, it is important to compute these helpful features in the preprocessing phase.

One feature that should help regression but cannot easily be deduced by the model is the distance between the pickup and dropoff locations.

Distance

There are several ways to measure distance between (longitude, latitude) pairs. I have opted to use several methods and provide all the distance metrics to the model so it can use whichever one is most predictive (or a combination of several).

The first is the L1 distance, also known as the Manhattan distance – this is the distance while only being able to move north/south and west/east. This distance could be useful because it

resembles the road system of blocks in NYC, where cars can't travel diagonally between destinations. It is defined as:

$$d(a,b) = |a_{lon} - b_{lon}| + |a_{lat} - b_{lat}|$$

The next metric is the L2 or Euclidean distance. This is the direct distance between two locations, when travelling as the crow flies:

$$d(a,b) = \sqrt{(a_{lon} - b_{lon})^2 + (a_{lat} - b_{lat})^2}$$

Because the world is a sphere, this means that calculating the Euclidean distance between latitude and longitudes (which assumes the world is flat) is not always an accurate measure of true distance between two points.

Because of this, another more complicated distance formula known as great circle distance (implemented by the haversine formula) was created which calculates the distance between two points following the curvature of the earth's surface. Because of this, the Euclidean distance can be thought of as the 'map distance' while the haversine distance can be thought of as the real distance. At such small scale, the difference should be minimal but I have decided to include both into my calculations.

Using this, I create three new features: `dist_l1`, `dist_l2` and `dist_haversine`.

Direction of travel

I hypothesize that knowing the direction of travel can also be helpful to the model – for example, there may be less traffic in an area when going north versus when going south. Because we're not directly given the route that the taxi takes, we cannot know exactly which direction the taxi is taking, however, we can make some approximations.

Because of this, I created two new features, `delta_lon` and `delta_lat`, which are simply equal to the starting lat/lon minus the destination lat/lon. These features quantify how far west or east the taxi travelled, as well as how far north or south it travelled.

In addition to these features, I also calculate the bearing angle between the two co-ordinates. This process is a little more involved. Thankfully, `numpy` implements an `arctan2` function, which can easily calculate the bearing (in radians) of a single vector from the x-axis. This can be adapted to calculate the bearing between two points (our pickup and dropoff co-ordinates):

$$angle(a,b) = \frac{180}{\pi} \cdot arctan2(b_{lat} - a_{lat}, b_{lon} - a_{lon}) + 180$$

As an example of the output of this angle feature, if the destination is directly east of the pickup, the angle will be 90°, while if the destination is north east, the angle will be 45°.

This feature allows the model to better understand what direction the cab is going to travel in from its pickup.

Current traffic conditions

Because we are given information about past trips that have occurred in the dataset (and other trips around the same time), we can see how busy the taxi service is at the time of a given trip, and potentially use this to infer the traffic conditions at the time. The basic idea is that the number of taxis recently taken reflects the total number of cars currently on the road – at busier periods more taxis will be taken. By incorporating this info, the model can learn information about conditions beyond basic repeated trends (for example, holidays or special events may not fit the normal daily traffic and by looking for abnormal taxi usage the model can spot times when trip times may be unusual.)

To capture this information, I create two new features, `daily_count` and `hourly_count`. These features represent the total number of taxi trips taken during the current day and current hour respectively. To calculate this, I simply do a groupby operation over the datetime and count the size of each group.

Time estimates based on speed of other taxis

Since the training and testing sets cover the same time period, this means that for any given point in time in the dataset, we know the trip duration for ~75% of the taxi trips. Using this trip duration as well as the approximate trip distance calculated earlier, we can work out the average speed of each taxi. Then, by assuming the test set taxis travel at the same speed, we can use this to create time estimates based on current conditions at any given point in the dataset.

Using this principle, I create a feature called `haversine_speed_estim`, which is calculated with the following process:

1) For every sample in the training set, I compute the haversine distance, and divide this by the `trip_duration` (the time taken to move that distance) to obtain the average speed of trip.
2) For every hour in the dataset, I take the mean of the speed across all the training samples, calculating the average taxi speed for that hour.
3) For every sample, I calculate the haversine distance travelled, and multiply it by the average speed for that hour to obtain an estimate for `trip_duration`. This can be performed even on the test set, giving me time estimates for the entire dataset that can be trained on with XGBoost.

3.2 IMPLEMENTATION

My initial implementation of my solution is a set of python files consisting of my preprocessing and feature engineering pipelines to create a final dataset, followed by an XGBoost model with default parameters. It is split into multiple python. Note that this section will not go too far into the detail of generated features as these are well described in the previous section. What follows is a brief explanation of the implemented pipeline in each section.

Preprocessing (`preprocessing.py`)

1) The training and testing csvs are read into dataframes in memory using pandas.
2) The `store_and_fwd_flag` feature is mapped into a binary variable where '1' corresponds to 'Y', by applying a lambda function over the column.
3) The `pickup_datetime` is parsed into a Timestamp object with pandas – from this I extract the Unix time in seconds, minute of the day, and the day of the week, storing them as additional columns in the dataframe
4) The trip duration (y) and row IDs are extracted into separate variables, and new dataframes are constructed storing only the columns used for training (x).
5) `pickle` is used to serialise all the preprocessed data to disk.

Feature engineering (`engineering.py`)

1) The preprocessed data is read back into memory with `pickle`.
2) The three distance functions are applied over the data, creating three new features in the dataframe. Both the l1 and l2 distance are vectorized operations (meaning they can be applied across the whole dataset in one operation without loops) so they are fast. The haversine distances are calculated in a loop across the dataset using the haversine package – however, this is slow so multiprocessing is also used to do this on multiple cores in parallel.
3) The `delta_lat`, `delta_lon` and `angle` features are created as described in the previous section and added to the training dataframes. These features are all computed with vectorized operations.
4) Next, the traffic features are created. First, the `pickup_datetime` feature is split at the space or at the first colon (to create a string representing just the date or the date and hour respectively). Then, for these two features, they are grouped by value and the occurrences of each value is counted. This creates a lookup table which is then applied onto the training and testing data in a loop, creating the `daily_count` and `hourly_count` features.
5) Finally, the time estimate features are computed. This is done as described in the feature engineering section: the speed is calculated for each sample, then they are grouped by hour and a lookup table of hourly speed is constructed which is then applied

to the dataset. The mean speed of the entire dataset is also created: This is used to fill in the feature in any test samples where the estimated speed is not known.

6) After feature engineering is complete, the processed data is once again pickled to disk.

When creating this script, I ran across some complications in the process. Initially, when I wrote the code for calculating haversine distance I applied it across the dataset as a loop. However, since calculating haversine is resource intensive, this was very slow. Because of this, I decided to use the multiprocessing package to spawn multiple processes and use these processes to map the haversine function over the data using multiple cores. This made the feature engineering script much faster.

In addition, when I initially implemented the time estimate features, I hit an error where some hour that appeared in the testing set did not appear in the training set, so there was no speed estimate for that hour and the lookup failed. For this reason, I had to modify the code to use the average speed over the entire training set where no value was present in the lookup table.

Basic XGBoost model (`model.py`)

1) First, the features, target values and IDs are loaded from disk once again using pickle.
2) The np.log1p of the target values is taken and this is used as the new target – this is so that XGBoost directly optimizes RMSLE.
3) The training data is randomly split into a training and validation set, with 20% of data used for validation.
4) The datasets are all converted to XGBoost's internal DMatrix format, and the original arrays are deleted from memory
5) XGBoost is trained on the training set using early stopping on the validation set (stop training if the score hasn't improved in 5 rounds), returning a model which is used to predict on the test set.
6) A dataframe is constructed with the predicted trip durations (with `np.expm1` applied) and the trip IDs, and is written as a csv to disk. This csv can be submitted to Kaggle.

3.3 REFINEMENT

Running my initial implementation as described above converged after approximately 1,200 rounds of boosting, which when submitted to the Kaggle leaderboard, gave me a score of **0.39098 RMSLE** on the leaderboard, putting me in 167th position at the time of writing. After obtaining this result, I went on to refine my model.

To improve the model, I decided to opt for a parameter tuning approach to find the set of parameters that best optimizes the RMSLE on the validation set, and hopefully that which also optimizes the RMSLE on the Kaggle leaderboard. Usually, I would use a Grid Search approach over a space of predefined parameters for this. However, training the model is slow due to a

combination of the resource-intensive XGBoost algorithm, a very large training set, and many features (up to 30 minutes on a 16-thread Intel i7). This means that a grid search approach is unfeasible for this problem.

Instead I have decided to use my own approach to parameter optimization which has worked well for me in the past, and is much faster than grid search. The basic idea is the following:

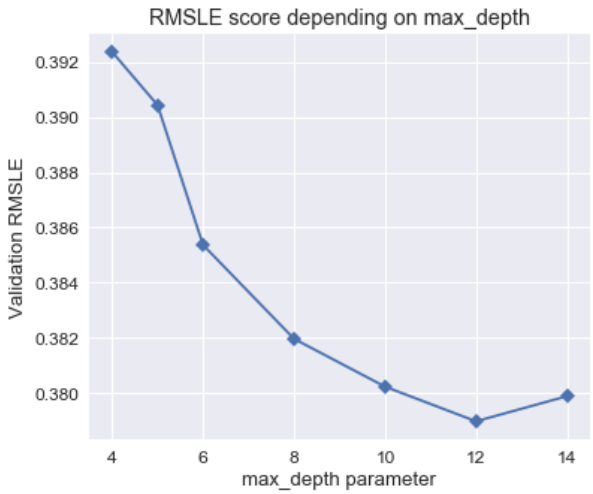
1) Some reasonable guessed default values are set
2) The parameters are tuned one-by-one. For every parameter:
3) Train a model for every value of this parameter to be tested, using the default values for all other parameters.
4) Find the value for this parameter that gave the best score
5) Update the default value for this parameter using the found best value
6) Move to optimizing the next parameter

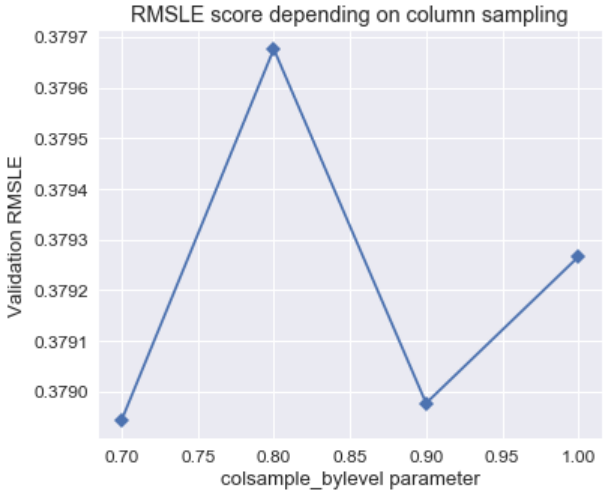
This means the optimization problem is linear with the number of parameters, while a grid search requires exponential computation with the number of parameters, making it much faster.

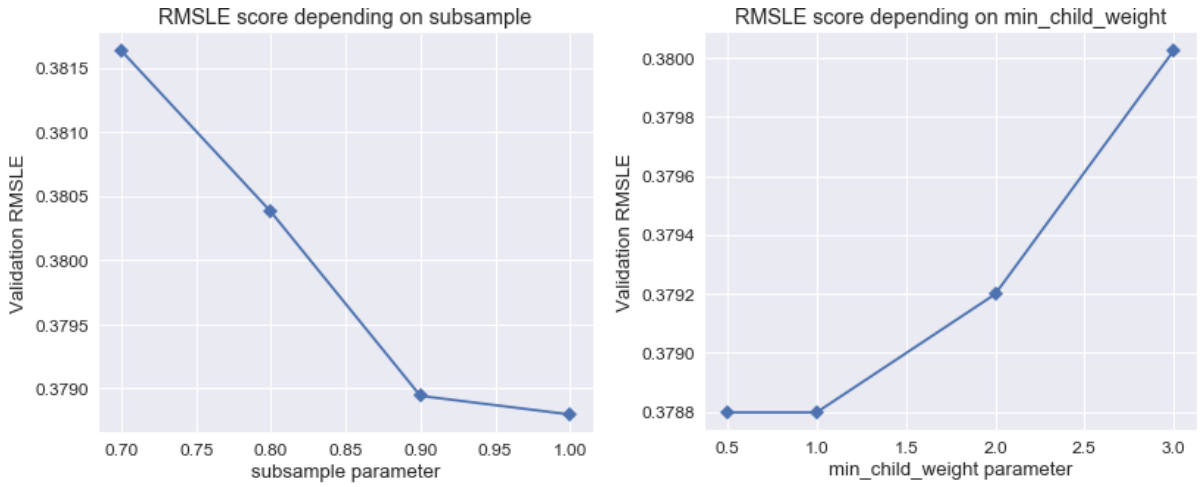
I have defined the following search space for my parameter optimization:

Feature	**Values**
`max_depth`	[4, 5, 6, 8, 10, 12, 14]
`colsample_bylevel`	[1, 0.9, 0.8, 0.7]
`subsample`	[1, 0.9, 0.8, 0.7]
`min_child_weight`	[0.5, 1, 2, 3]

This required 19 models to be built to find the optimal combination (versus 448 with grid search). The following results were obtained as a result of the optimization:







The full code for this optimization is in `model_search.py`. From this, an optimal set of parameters were obtained:

```
max_depth = 12, colsample_bylevel=0.7, subsample=0.99, min_child_weight=0.5
```

In creating my refined model, I trained another XGBoost model using these tuned parameters. In addition, I also halved the eta (learning rate) to 0.05, which leads to slower convergence but eventually nets a higher score. The code for this model is available in `model_optim.py`.

4. RESULTS

4.1 MODEL EVALUATION AND VALIDATION

The final model as implemented in the refinement section gave a big improvement on the leaderboard over my initial implementation, improving to **0.38092 RMSLE** from 0.39098 in the initial model. This improved my rank 99 places, leaving me at 68th out of 767 teams.

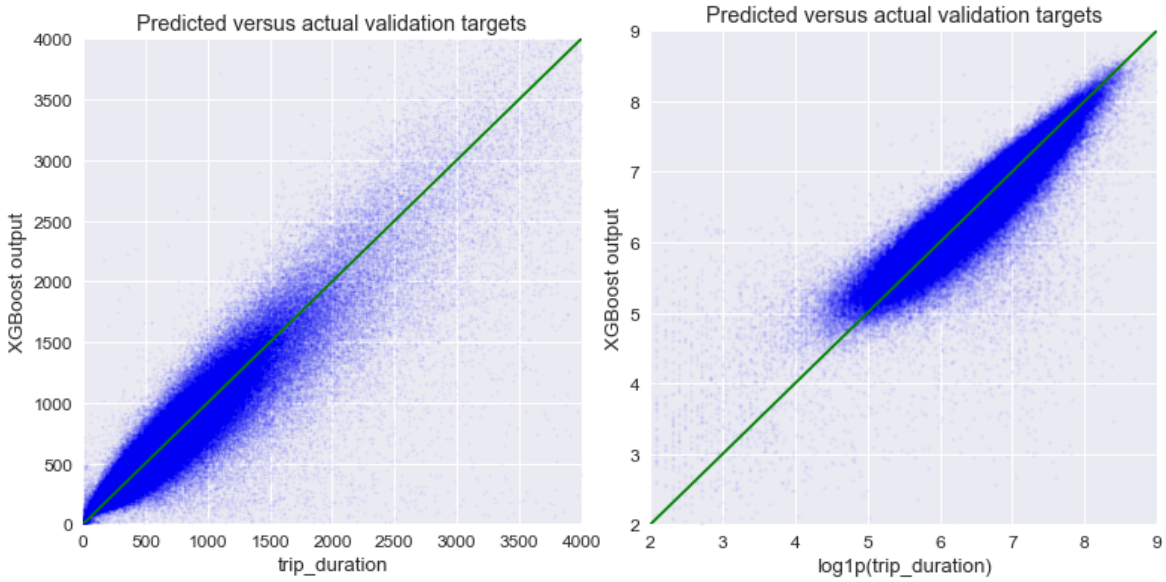
65	▼2	T Flana		0.38048	23	7d
66	▼2	NMSL.WSND		0.38059	12	13h
67	▲35	becarful		0.38091	21	3h
68	▲20	anokas		0.38092	7	~10s

Your Best Entry ↑

Your submission scored 0.38092, which is not an improvement of your best score. Keep trying!

69	▼4	Clement		0.38117	26	20d
70	▼4	Take Me Down to Homosceda...		0.38145	13	6d
71	▲140	Peter Guo		0.38161	6	3h

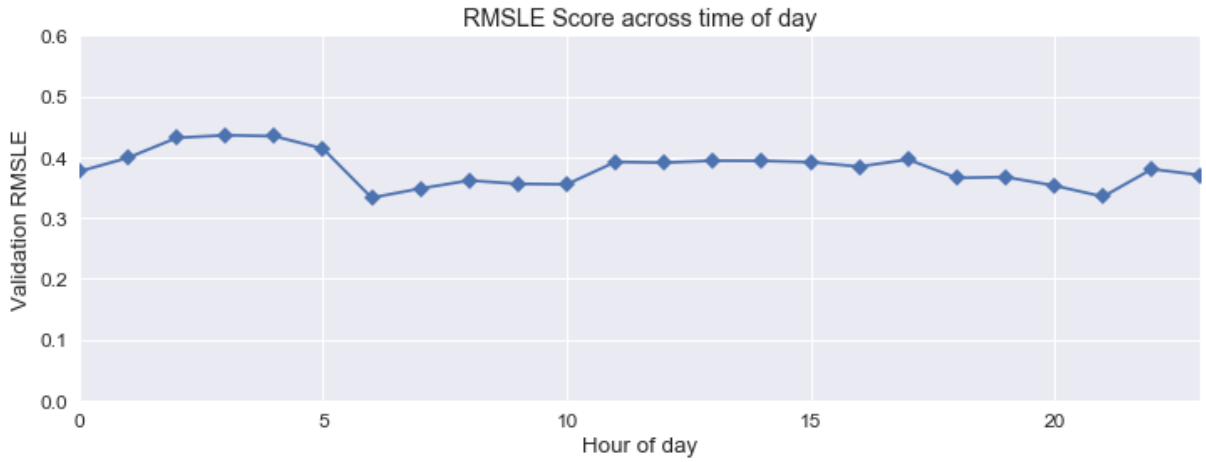
While the raw RMSLE result I obtained is satisfactory, I would like to verify whether the model is robust and can adapt across the whole training set or whether it only generalizes to a subset of data. To begin with, I will analyse how correct the model predictions are depending on what the target variable is, to check whether the model is able to predict well for both short and long trips:



From these plots, we can see a few interesting observations. First off all, we see that for most trips across most durations, the XGBoost model is reliable in generating good predictions. However, on the log plot on the right, we can also see that for samples where the log duration is less than 5, the model is not able to predict these very accurately on the log-scale. This corresponds to two-minute rides or lower, so it makes sense that these would be difficult to predict accurate to the second, as each trip has an inherent random time added to it (getting in the taxi, paying, etc).

However, looking at the graph of real time on the left, we can see that for the most part the model still predicts the trip time well – there are almost no cases where XGBoost overpredicts the trip time, but a few cases where it underpredicts. I hypothesise this is because of some effect on the taxi which is not captured in any of my features (e.g. passenger wants to take a route via some other location).

We can also look at how well the trip duration estimation works during different times of day.



The above plot shows the effect that differing times of day has on the performance of my model. The model appears to perform worse during the early hours of the morning when speed is generally faster and perform better when there is lots of traffic – this could have something to do with the model's worse performance (on a log-scale) predicting short trips. However, while I would argue this trend in the model performance definitely exists, the performance of the model is satisfactory at all times of day.

From this analysis, I am relatively confident that the model is robust across different rides and can be trusted.

4.2 JUSTIFICATION

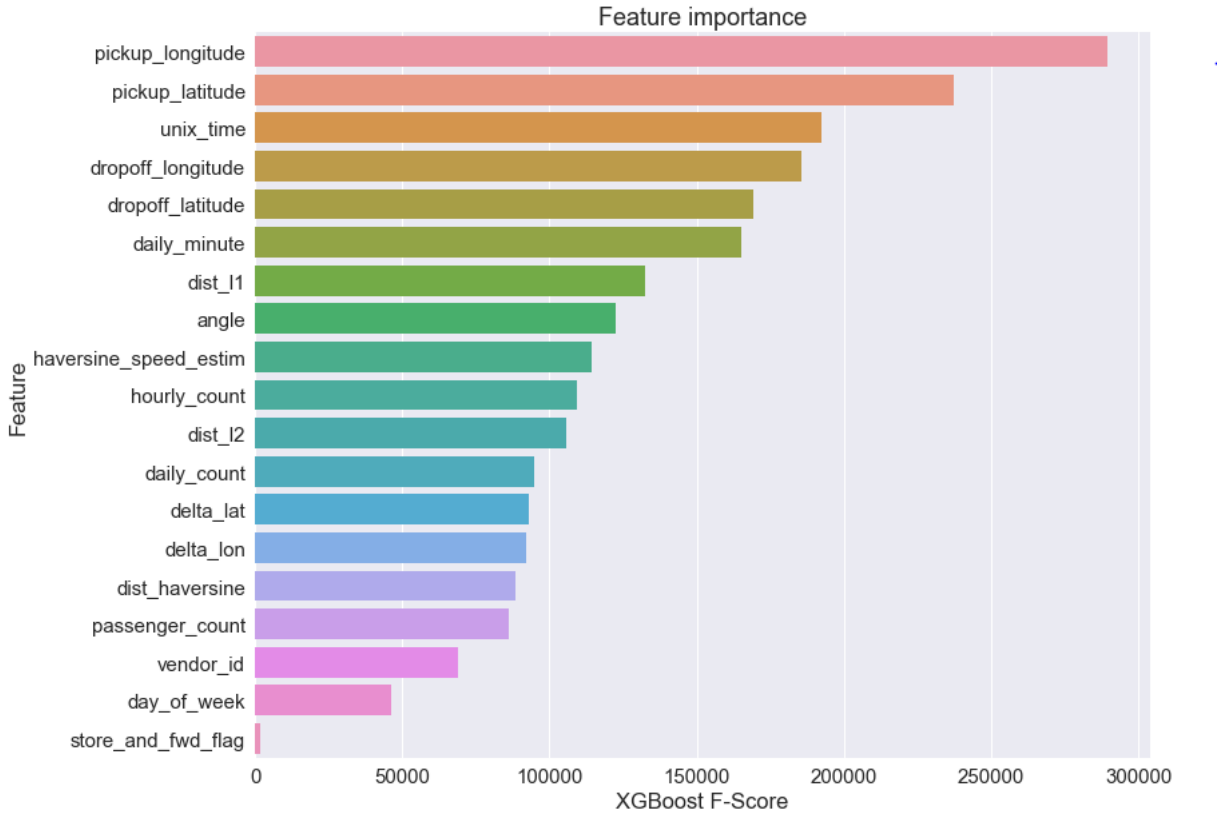
My final solution greatly outperforms the benchmark solution, more than halving the error, going from a **0.798 RMSLE** score down to **0.381**. Another metric which can be used to compare these two solutions is to look at how often they predict trip durations close to the actual durations.

If we define a good prediction as being less than 20% out from the actual (absolute, not log) duration, we can calculate that while the benchmark solution is correct just **21%** of the time, my refined model brings this value to **59%**, showing the extent of the improvement.

I feel that the final solution I have presented here has been able to well capture the signal from the data in a way significant enough to produce a useful model. Looking at the scatterplots of predictions in the last section, it can clearly be seen that the model has learned to well-predict the duration.

5. CONCLUSION

5.1 FREE-FORM VIZUALIZATION



This plot displays XGBoost's built-in feature importance metric, displaying how much each feature contributed to the final solution. Surprisingly, the pickup longitude and latitude are the most important singular features in the dataset – this may suggest that XGBoost actually learnt a very granular speed pattern over the pickup location.

From this plot, we can also see that L1 (Manhattan) distance is the most important distance metric for this problem. While all three distance metrics are lower in the feature importance than I would expect, it is worth noting that the features are lower in the importance as XGBoost has been given the similar information multiple times, so it does not rely too much on a single distance feature: the combined importance of these distance features would rank at the top, especially if you added features such as `delta_lat` and `delta_lon`.

Interestingly, `store_and_fwd_flag` provided basically no information to the model. This makes sense, as I did not see how this feature would impact trip duration during data exploration.

5.2 REFLECTION

My final end-to-end solution can be described in a few simple steps:

1) First, the input data is preprocessed to remove or transform non-numerical features.
2) Then, new features are created based on existing ones, in order to bring out additional signal from the dataset
3) A parameter-optimized XGBoost model is trained on the newly engineered data to build a model that can predict the duration of a taxi trip
4) This model can then be used on new input trip data to create predictions about trip duration

I think the most interesting part of this project was the feature engineering – this is where, instead of going through the same (almost muscle memory) technique and processes as in other problems, experimentation and new ideas are required to come up with novel features. I also found it the most challenging, as some of the features I built were complicated to implement in an efficient way, for example the time estimate features.

I think my final solution does fit what I expected to have for this problem. However, while this model performs well in the Kaggle leaderboard, I do not feel it would be an appropriate model for use in an actual taxi trip prediction situation. This is because in this dataset, the training and testing sets are aligned in time, meaning that you are given ground truth for data in the same time period as the test set. In a real-world situation, this data would not exist – you would only have data on *past* taxi rides, and not data on current or future rides. Because of this, the model would need to be retrained with a test set which is forward in time, and features such as the time estimate feature would not be possible in its current form as the ground truth data it is based on would simply not be available.

5.3 IMPROVEMENT

I feel that there are several aspects of my solution that could be improved, but where I did not have time to investigate these aspects for my project.

Firstly, I think that there is a lot more scope in the sort of feature engineering that could be done, especially incorporating outside data. For example, collecting and using weather data as a feature or a list of holidays and special occasions as features could have provided the model with more info that could let it pick up patterns in traffic conditions. In addition, I think that my approximate methods of measuring distance could be replaced, for example, with a much more accurate estimate based on a route plotted on mapping data. However, this is not something I could implement myself.

In addition, I feel the modelling side could also be improved. Very often, Kaggle solutions are made up of ensembles of large numbers of models whose predictions are combined (thus combining the strengths of each model), resulting in a meta-model with much higher

performance on the leaderboard than any single model. I think this would be the way forward if I wanted to improve my submission.

An improvement on the score I have set in this report is definitely possible. At the time of writing, the highest achieved score was **0.36531 RMSLE,** a large improvement. One of the top competitors shared details about his solution and said that it used an ensemble of models on top of 59 engineered features, showing that a large amount of work could be done to further improve my score.