CPSC 381 Final Project Report: Yuttle Tracker

Koray Akduman, Justin Dominic, Addison Goolsbee, Charlie Stephenson
Code: https://github.com/kakduman/yuttle-tracker

INTRODUCTION

Planning the journey to and from class is one of the most frequent time management decisions Yalies makes daily. Students use many transit options to optimize their trek from one lecture to the next. These options include bikes, skateboards, scooters, Naruto running, and the Yale Shuttle ("Yuttle"). The latter is notorious among the student body for its unpredictable arrival times. The most commonly used route is the Blue route from lower campus to the central and upper campus. Notable stops include the Med School, the intersection of York St. and Elm St., College St. at WLH, the CEID, and the Watson Center. The existing bus schedule app, DowntownerApp, provides estimated arrival times. Unfortunately, these times have proven to be extremely inaccurate. These inaccuracies can result in tardiness or absence from classes, exams, meetings, and other significant events. As a result, most students either don't bother to track the Yuttle, or outright avoid any contact with the disgraced transit system.

Our solution was to create a model that is comparable to, if not better than, DowntownerApp, which, as we found out, has an average error of 7.60 minutes for any given stop of the Blue route. We have delivered a model that can more accurately predict the Blue route's arrival times given the latitude, longitude, time of day, and day of the week (and sometimes DowntownerApp's ETAs for each stop). Internally, we also calculated three more inputs: percent route completion, percent time of the 11-hour schedule, and last stop (last stop is a given input of the API but is wildly inaccurate, so we calculate it ourselves based on the longitude/latitude and previous locations). The output will be the expected arrival time for each subsequent stop.

To accomplish this, we took the following steps

1) Set up a scraper that sends GET requests to the DowntownerApp API every 10 seconds, and have it run for 3 weeks, collecting raw data.
2) After we're satisfied with the amount of raw data collected, post-process the raw data by removing useless data and converting existing raw data into more useful metrics.
3) Create a linear regression model and a neural network that will use the processed data
4) Train the models on 80% of the processed data and test on the remaining 20%
5) Using the DowntownerApp ETAs in the collected data, compare our results using each model to Downtowner App's predictions to see which is better (has the lowest average time off for each stop)

DATA COLLECTION

Before we could start training a model, we first had to collect massive amounts of data on the buses. Luckily for us, DowntownerApp has a (somewhat) public API which we can access via GET requests. We used the following URLs for data collection:

- https://yale.downtownerapp.com/text/routes shows the stop IDs for each route. Since we were making a model for the blue route (route 1), we used all 32 stops of the blue route, in order, and wrote them down as their ID numbers. You can find this array in the file utils/route_stops.txt
- https://yale.downtownerapp.com/routes_buses.php lists all active buses and their information such as latitude. We use this endpoint to get a list of all buses on the blue route
- https://yale.downtownerapp.com/routes_eta.php?stop={stop} lists DowntownerApp's estimate time of arrivals for each bus that is on a route that will visit the stop. We used these values so we could compare our model to DowntownerApp's
- https://yale.downtownerapp.com/routes_stops.php converts stop IDs to latitude-longitude pairings, helping us better calculate where the bus is. This is particularly used during the post-processing step

Using these three endpoints we set up a scraper that collects data and also generates data, like the percentage of the route that the bus is at (based on latitude and longitude). The following files are used for the scraper:

- scraper.py: the main driver file for the scraper
- utils/route.py: class definition for a route. Contains functions for collecting data from the endpoints and writes data to a file
- data/raw_data: folder where all the data goes to. There is a separate file for each bus ID. Note that a single bus can change IDs from one day to the next, even if it's on the same route.
- utils/path_conversion.py: calculates the percentage of the route that is currently completed, allowing us to much more easily determine what stop the bus is at/where the bus is. Used in route.py
- utils/time_conversion.py: used in route.py, gives us the fraction of the 11-hour day (buses ride from 7 AM - 6 PM) and the weekday
- utils/get_estimated_times.py: used in route.py, gets all the ETAs from the DowntownerApp and puts them into a dictionary for each entry

In order to collect data, we set up scraper.py to run for about 3 weeks, once every 10 seconds, using a Birdflop server. We threw out all data that was on Saturday/Sunday (non-standard bus route), as well as all data that was not within the standard 7 AM - 6 PM shuttle schedule. Because we had to read so many GET requests every 10 seconds, we parallelized our scraper. This caused the scraper time per scrape to reduce from over 10 seconds to about 1.6 seconds, which is more than enough time to record the data. In total, here is all the 'raw' data we collected for each bus on the blue route:

- Lat: latitude, from DowntownerApp
- Lon: longitude, from DowntownerApp
- pathPercent: latitude and longitude converted into a point on the route as a percentage of the full route
- sinProgress: pathPercent within a sine function: sin(2π*pathPercent). To better describe the data by highlighting the circularity of the data. The path percent field goes gradually from 0 to 1, and then jumps immediately from 1 to 0, however, this should be treated the same as 0 to 0.001. Both sinProgress and cosProgress help to signify this in the data.

We use sine and cosine because if we just used one, there would be two points along the path where the sine/cosine values are the same, but having both helps to differentiate them.

- cosProgress: pathPercent within a cosine function: cos(2π*pathPercent) (see above)
- lastStop: the last stop the shuttle was at, from DowntownerApp. This turned out to be inaccurate some of the time so we had to recalculate it in the post-processing step
- lastUpdate: the UNIX timestamp of the last time the DowntownerApp checked the bus' data. Usually updates every 10 seconds (which is why we run the scraper every 10 seconds. From DowntownerApp
- dayPercent: lastUpdate converted to a percentage of the 7 AM - 6 PM time range. If not in that time range, the data is thrown out
- Weekday: lastUpdate converted to a day of the week. If Saturday or Sunday, the data is thrown out
- estimatedTimes: a dictionary of DowntownerApp's estimated arrival times for each stop, for the specific bus. In total there are 32 stops on the blue route, so this became a LOT of data.

POST-PROCESSING

Creating the post-processing section took the majority of the time on this project, as our script was continuously updated to reflect edge cases that we had not considered. We first go through our data and filter out data that we cannot use. Using the lastUpdate field (in UNIX time), we remove all the data on Monday, April 22. Protests at Yale meant that an intersection used by the Blue Line was blocked, and the line was rerouted. Data for this day was therefore irrelevant to our model. We also remove any data for arrivals after 6 PM, as the Blue Route officially ends at 6 PM, and the stops after it may be for the bus after it switches routes. We further remove all data that don't have a complete arrivals dictionary (i.e. if the bus never arrives at the other stops, we can't train our model off of this data).

The core of this file goes through the data in reverse order, determining at which point the shuttle arrives at a given stop and updating a dictionary that keeps track of the bus arrival times. When the bus arrives at a stop that already has a value, this dictionary is updated to replace that stop's value, ensuring that it only has the most recent value for the bus' arrival at that stop. At each line, the dictionary is appended in its current state to the data, giving us a value for when the shuttle will arrive most recently that dynamically updates for every stop. We additionally reset the data when the day changes or when the bus is otherwise idle for multiple hours.

In order to know the bus' arrival at each stop, we use a function that finds the most recent stop of a bus given a path_progress and a de_facto_heading. The path_progress is the progress the bus has made along the route, and the de_facto_heading is roughly the direction that the bus is heading. Using these two variables, we are able to map the bus to the most recent stop it has visited and therefore have a continuously updating lastStop variable that we assign to the bus and can reference when updating our arrivals dictionary. The de_facto_heading indicates

whether the bus is headed northbound or southbound based on the previous stops it has visited. Since the bus overlaps its own path for one block, it is hard to determine the path_progress given the latitude and longitude alone. We use the de_facto_heading variable to determine the direction of the bus and therefore update the path_progress and lastStop variables accordingly, removing this source of confusion.

MODEL & TRAINING

Beginning this project, we aimed to test the ability of two different models to predict Yuttle arrival times. First, we wanted to observe whether the data had enough linearly separable information to train a reasonably accurate model. Our input data takes the form of 6 features (pathPercent, sinProgress, cosProgress, lastStop, dayPercent, weekday), with sinProgress and cosProgress being derived from pathProgress in an attempt to make the cyclical route avoid jumping from 1 to 0 and therefore potentially increase the linear separability of the data. Our output takes the form of a 32-dimensional vector, with each dimension corresponding to the predicted arrival time at each of the 32 stops. Since linear regression natively provides one output for a given input, we trained 32 linear regression models using scikit-learn's MultiOutputRegressor(), which fits one given regression model per target dimension. Therefore, our code below created a multi-output model composed of 32 linear regression models which we used to predict 32 outputs given 6 input features:

```
# Model definition and training
model = MultiOutputRegressor(LinearRegression())
model.fit(X_train, y_train)
```

We then trained one model using an 80/20 train/test split of our collected data, provided as 126,478 6-dimensional input vectors and their corresponding 32-dimensional output vectors. With this linear regression approach, we achieved a 10.11 minute mean absolute error. While 10.11 minutes is comparable with the DowntownerApp's 7.60 minutes, it still fails against the DowntownerApp's predictions. In an effort to improve our accuracy, we wanted to see the effects of integrating our predictions with the DowntownerApp's predictions. We trained a second multi-output linear regression model by appending the DowntownerApp's predictions to the original input vectors, resulting in an input of 126,478 38-dimensional input vectors and their corresponding 32-dimensional output vectors. This model, as predicted, performed significantly better than our original input, achieving a mean absolute error of 7.83 minutes. Still, it fails to beat the DowntownerApp's estimates. We hypothesize that this is because linear regression optimizes for mean squared error, but mean absolute error is a more meaningful metric in our situation. While the mean squared error is 0.0012 in our model versus the DowntownerApp's native 0.0015, the mean absolute error is still higher in our model. These data show that linear regression is likely not the best model to choose for our situation because it (1) fails to optimize for the most important metric and (2) does not capture the complexity in the data.

We next created a neural network with layer sizes of (input_dimensions, 64, 256, 1024, 256, 32 (output_dimensions)). These layer sizes were deemed decently optimal after much hyperparameter tuning. We combined this with a ReLU activation function between the layers, the Adam optimization algorithm that performs stochastic gradient descent, and a learning rate

scheduler that halved the initial learning rate of 0.001 every 10 epochs. The model was trained for 75 epochs using a loss criterion of L1 loss, which corresponds to the mean absolute error—the exact metric we wanted to optimize. Since we were planning on training only one model in each case without comparing between models and choosing the most optimal one, we elected to again not include a validation split, opting instead for a 80/20 train/test split. We trained two models with this setup: one on the initial 6-dimensional inputs and one on the modified 38-dimensional inputs. We achieved a 4.98 minute mean absolute error on the initial 6-dimensional inputs (without the DowntownerApp estimates), and a 4.67 minute error on the modified 38-dimensional inputs (with the DowntownerApp estimates). These results demonstrate that our model successfully outperformed the DowntownerApp, and further outperformed it after integrating it with the DowntownerApp's internal estimates.

RESULTS

We wanted to test our model's findings relative to the Yale Shuttle's DowntownerApp provided estimates. Using the data we collected over two weeks, we found that DowntownerApp had a mean average error of 0.0115 and a mean squared error of 0.0015—in other words, the DowntownerApp's estimates were off, on average, by 7.6 minutes throughout its route.

Our linear regression yielded the following results:

Including Downtowner Estimates In The Model:

- Mean Squared Error: 0.001
- Mean Absolute Error: 0.012
- Mean Distance: 7.825 minutes

Without the Downtowner Estimates:

- Mean Squared Error: 0.001
- Mean Absolute Error: 0.015
- Mean Distance: 10.108 minutes

Downtowner Estimates:

- Average MSE: 0.0015
- Average MAE: 0.0115
- Average Distance: 7.601 minutes

Overall, our linear regression estimates were *worse* than the base app's predictions—even when we included the base predictions as a variable in our model. We hypothesize this is because we optimized for MSE with linear regression instead of MAE. This gives us a lower MSE but not a lower MAE. But we care about MAE because that's the number, in minutes, of how far off the estimates are to actual arrivals. So, linear regression isn't the best option.

We then chose to use a neural network because it seemed most natural, given that we desired an output of an estimated time rather than a classification system. Our neural network yielded *significantly better results*:

75 epochs, halving learning rate every 10 epochs, **not using DowntownerApp estimates:**

- Test Loss: 0.008
- Average distance: **4.982 minutes**

75 epochs, halving learning rate every 10 epochs, **using DowntownerApp estimates:**

Test Loss: 0.007

Average distance: **4.670 minutes**

Our neural network *significantly outperformed* the base predictions, regardless of whether or not we incorporated the base predictions. Our best model was 38.56% closer to the true arrival time than the base predictions!

CONCLUSION

Throughout our process, we faced numerous challenges that impacted our results. First, the inherent variability in the bus schedules introduced significant noise into our data. Factors such as non-standard routes on weekends/evenings and route disruptions from events like campus protests further complicated our dataset. For example, on Monday, April 22, campus protests moved to a critical intersection on the Blue Line, forcing a detour that caused significant delays. Yale shuttles were also used that morning to transport arrested students to a police processing facility, and this was counted as movement in the API data. As a result, we had to throw out all the data from that day, and even after sanitizing the data, the buses remained unpredictable, making creating a consistently accurate model difficult.

Another major issue was the reliability of the DowntownerApp data. The app often provided blatantly incorrect information regarding the last stop of a bus and included stops that didn't exist (e.g. stop 0), leading to a scenario often referred to as “garbage in, garbage out.” This unreliable input made it difficult to train our models effectively. Furthermore, the public API used to collect the data lacked any documentation, which meant much of our initial effort was spent understanding and manually correcting the data fetched via undocumented GET requests observed through inspect element. When we were combing through which data to collect from these GET requests, we also mistakenly decided to not include the heading (the direction the bus was oriented) because we thought we would have no use for it. However, when we were post-processing and training, we realized that some buses pass the same road/points on different parts of the route (i.e. going to the end vs coming from), which was necessary for

accurate predictions. We ended up having to manually calculate the heading, and it would've saved us a significant headache had we just collected the data initially.

Despite these challenges, our neural network model demonstrated a notable improvement over the baseline provided by DowntownerApp. However, the limitations posed by the short duration of data collection (only two weeks) and the manual labor required to clean and preprocess the data suggest areas for future improvement and scaling:

To enhance the reliability and scalability of our predictive model in the future, we recommend:

- Extended Data Collection: Longer periods of data collection across different seasons and conditions would provide a more robust dataset, leading to better model training and generalization.
- Automated Data Cleaning Tools: Developing tools that can automatically detect and correct anomalies or incorrect data from the source will reduce manual intervention and improve the quality of the training dataset. However, this is challenging because our most effective data cleaning came from a thorough understanding of when routes were disrupted; as this project scales up, it may become harder to identify when to throw out bad data.
- Documentation and Open APIs: Advocating for better documentation and open APIs from apps like DowntownerApp would facilitate easier access to data and more transparent methods for data collection and usage. Furthermore, any improvements to the API such as removing instances of junk data would make it significantly easier to scale and train new models.

In conclusion, while we faced numerous challenges in creating a more accurate model for predicting Yale Shuttle arrival times, our neural network approach demonstrated significant improvements over the existing DowntownerApp estimates. By addressing the limitations encountered during this project and exploring further enhancements, we believe that future iterations of this model could provide even more reliable and valuable information to the Yale community. Going beyond the scope of our project, these results demonstrate that machine learning has great real-world applications, and implementing similar models can help transit systems across the world become more efficient, robust, and accurate.