NextBuzz

Using machine learning to predict Georgia Tech bus arrival times

Nicholas Petosa

Table of Contents

Motivation

It is a widely held belief in the Georgia Tech community that the bus system is unreliable. Georgia Tech students over reddit have asserted time and time again that time-to-arrival predictions made by NextBus are inconsistent with actual arrival time [1]. NextBus' unreliability is not unique to Georgia Tech – the San Francisco Examiner has published an article highlighting how unreliable NextBus predictions are for San Francisco streetcars, with predictions being inaccurate 40% of the time if a vehicle is 20 minutes away [2]. There are factors that NextBus does not seem to be accounting for in its predictions, like rush hour, weather, and other community-specific variables, like when class lets out. Many students have given up on buses altogether and walk from class to class [1]. They are paying $85 per semester for a service they are not even using because it is so unreliable [3]. Observant students that still use the buses have developed soft rules like "a blue bus at 9PM can do full loops in 10 minutes" in order to accurately forecast bus arrival [1]. The goal of this project is to perform data analysis on historical NextBus data to quantitatively measure its accuracy, and then to build a machine learner which can recognize patterns imperceptible to the casual observer and produce better time-to-arrival predictions. Once the model is built, I create a web app which displays improved arrival time predictions relative to the closest stop.

Related Work

NextBus is already in the business of predicting bus arrival times. NextBus makes predictions for buses across the country, not just for the buses at Georgia Tech. As a result, their predictions are lacking domain knowledge specific to local communities, like when rush hour occurs or when classes let out. NextBuzz will build on NextBus' time-to-arrival estimate to produce a more accurate prediction of bus arrival time.

There have been other attempts to deep-dive into the accuracy of NextBus predictions. Tommy Leung developed the NextBus Delay Tracker, a project which uses linear regression on two weeks of Cambridge Massachusetts NextBus data to generate better estimates [4]. The most obvious difference between our projects is that they concern different regions, but the most significant difference is that Tommy did not integrate external factors like weather and class schedule into his predictor. Further, he uses linear regression, a parametric model which will perform sub-optimally on this non-parametric multivariate problem. Another deep-dive into NextBus accuracy was performed in 2015 by the transportation tech company Swiftly. They published a post on medium breaking down how NextBus accuracy plummets as it tries to predict arrivals further out in time [5]. Swiftly does not attempt to use this data to solve prediction accuracy.

There are a number of major players which are currently providing Georgia Tech students and faculty with arrival predictions, and all of them are based on the NextBus API. The most rudimentary of these are the digital signs present above most bust stops around Tech, which display NextBus' predicted time-to-arrival. The problem with this medium is that students who are still in their dorm are unable to access arrival information. To solve this, NextBus provides an website which displays its predictions [6]. The UI for these predictions is clunky, particularly its map for tracking bus location. The best option students have for accessing bus arrival time predictions are the GT Buses apps, available on both the Google Play [7] and iPhone App Store [8]. These mobile apps provide a clean and convenient

interface for tracking buses and accessing arrival data. The only problem that remains is that their predictions are inaccurate, since they are sourced from NextBus. The goal of my web app is to augment existing solutions such as the GT Buses apps, and potentially work with its creator to integrate my NextBuzz predictions into the popular mobile application.

Project Components and Deliverables

The NextBuzz source code can be broken down into five major components. First is the data collection engine which pulls data from NextBus and Open Weather Map. Second is the data processing pipeline, which transforms raw data. Third is the code used for exploratory analysis experiments. Fourth is the code used for building machine learning models and evaluating their performance. Fifth is the web application which visualizes the learner's predictions.

- **README.md**
  - Technical documentation discussing environment setup and execution instructions for all experiments and code delivered in this project.
- **PowerPoint Slides**
- **20 days of raw NextBus/Weather historical data: https://gtvault-my.sharepoint.com/personal/npetosa3_gatech_edu/_layouts/15/guestaccess.aspx?docid=049935ca6c8164099b7b1593014a98fa1&authkey=AX34_rLI8XzhpNP5gXrj_UU**
- **Training/Testing data (raw data passed through data prep pipeline): https://gtvault-my.sharepoint.com/personal/npetosa3_gatech_edu/_layouts/15/guestaccess.aspx?docid=03a675787e44f4fb991d5058e05d33abf&authkey=AXsDjVSwl9saMAOz3RvMgVs&e=b59438058a9f4bf88fd011f485696391**
- **Data Collection**
  - `collect.py` – Script that queries NextBus' API and the Open Weather Map API every twenty seconds, and then stores that information in an sqlite database.
- **Data Processing**
  - `cleaner.py` – Data processing module. Deletes duplicates, turns categorical variables into one-hot encodings.
  - `feature_engineering.py` – Data processing module. Adds new features to a dataset derived from its existing features. Specifically, detects arrival events, generates time-related features, and generates Georgia Tech specific features.
  - `pipeline.py` – Controller which directs raw data through all processing modules, generating a processed CSV full of learnable data.
- **Exploratory Analysis**
  - `analysis.py` – Creates a scatter plot between any two features in the data set. Can also plot the arrival events detected by the heuristic.
  - `explore.py` – Controller which calls `analysis.py` with different examples.
- **Modeling and Evaluation**
  - `train.py` – Loads in learnable CSV, defines model algorithm and hyper-parameters. Calls `supervised.py`'s learning modules.

        - `supervised.py` – Conducts supervised learning. Exposes interfaces for rolling cross-validation learning, and train/test split learning. Returns a trained model and error metrics in a data frame.
- **Web application**
    - `webserver/server.py` – Exposes the website's REST endpoints, including functions for loading in the HTML and JavaScript, as well as exposing endpoints for functions in `backend.py`.
    - `webserver/backend.py` – Data layer. Interacts with sqlite to return the latest predictions to the web server.
    - Everything else in `webserver/` are assets for the front-end. This includes the semantic UI framework as well as Highcharts.

Technologies

- The entirety of the NextBuzz back-end was coded in python 2.7.
- Packages used:
    - scikit-learn: Used for all machine learning in the project.
    - Flask: The web application server.
    - Sqlite3: Client library for interacting with historical data in sqlite database.
    - Pandas and numpy for raw data manipulation.
- The NextBuzz front-end was created using HTML, the Semantic UI CSS framework, Google Maps API for geolocation, and the Highcharts Javascript library.
- Sqlite for database.

Data Analysis: Finding NextBus' Error

Data Collection

The first goal of my research was to quantify the inaccuracy of NextBus' predictions. To even begin describing NextBus' error, I first needed historical NextBus data. NextBus has a public API, but does not offer any endpoints for querying historical datasets. It was therefore necessary to construct such a dataset by scraping NextBus' API continuously and accruing bus information in a database. This is the exact purpose of the `collect.py` script, which queries NextBus every 20 seconds for bus data. Specifically, every 20 seconds, it loops through all available routes and stops, and for every possible combination adds a new row to the database containing the following fields:

- `timestamp` – The time that NextBus updated this prediction.
- `stop` – the ID of the stop this bus is approaching
- `route` – the route this bus belongs to (red, blue, trolley...)
- `busID` – the unique ID associated with this bus
- `numBuses` – the number of buses running on this route at this moment
- `busLat` – the latitude of the bus
- `busLong` – the longitude of the bus
- `layover` – whether the bus is running late
- `isDeparture` – bus is stopped for several minutes

- `predictedArrival` – the timestamp NextBus believes the bus will arrive
- `secondsToArrival` – NextBus' prediction of seconds to bus arrival
- `temperature` – current temperature in Kelvin
- `pressure` – current air pressure in hPa
- `humidity` – current percent air humidity
- `visibility` – current visibility in meters
- `weather` – a string describing the current weather condition
- `wind` – current wind speed, in m/s.
- `cloudCoverage` – current cloud coverage as a percent

Optionally, `collect.py` can also add a prediction field, an additional column which is the model's prediction given the scraped instance. This can only be used once a preliminary model has been built. A completed instance is then added as a row in a sqlite database, which is generated by the collection script if it does not already exist. In my experiments, I collected data over a 20 day period, giving me a total of 2 million unique rows to work with.

Arrival Detection

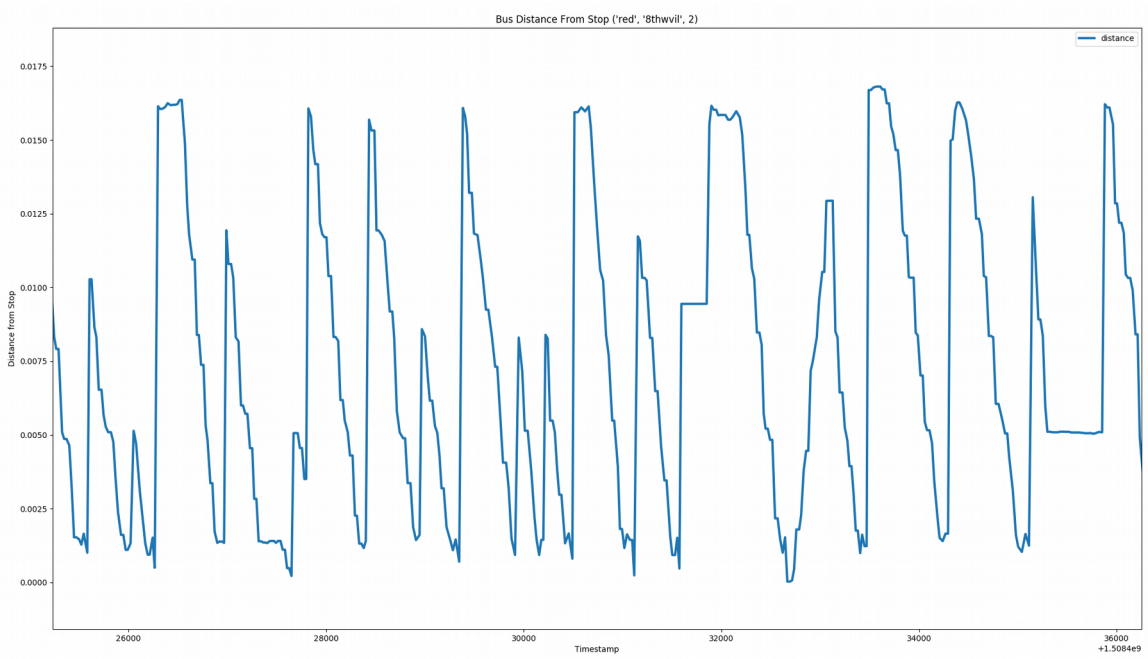
Once data is aggregated, it is necessary to prepare it using certain techniques so that it can be effectively analyzed. This step is typically called "data wrangling" or "data munging." All data preparation is done by passing it through `pipeline.py`, the most important transformation being from the `heuristic()` function of the `feature_engineering.py` file. This function detects when buses arrive at a stop, and sets the `newArrival` column to true for the next reading of a stop after an arrival occurs.

This is a very important step. In order to measure how inaccurate NextBus is, realize that we need to know the *actual time to arrival* for every row to compare the NextBus prediction against. NextBus' API does not provide any indication of an arrival event occurring, so we must infer that an arrival occurred based off of some signal in our time series data.

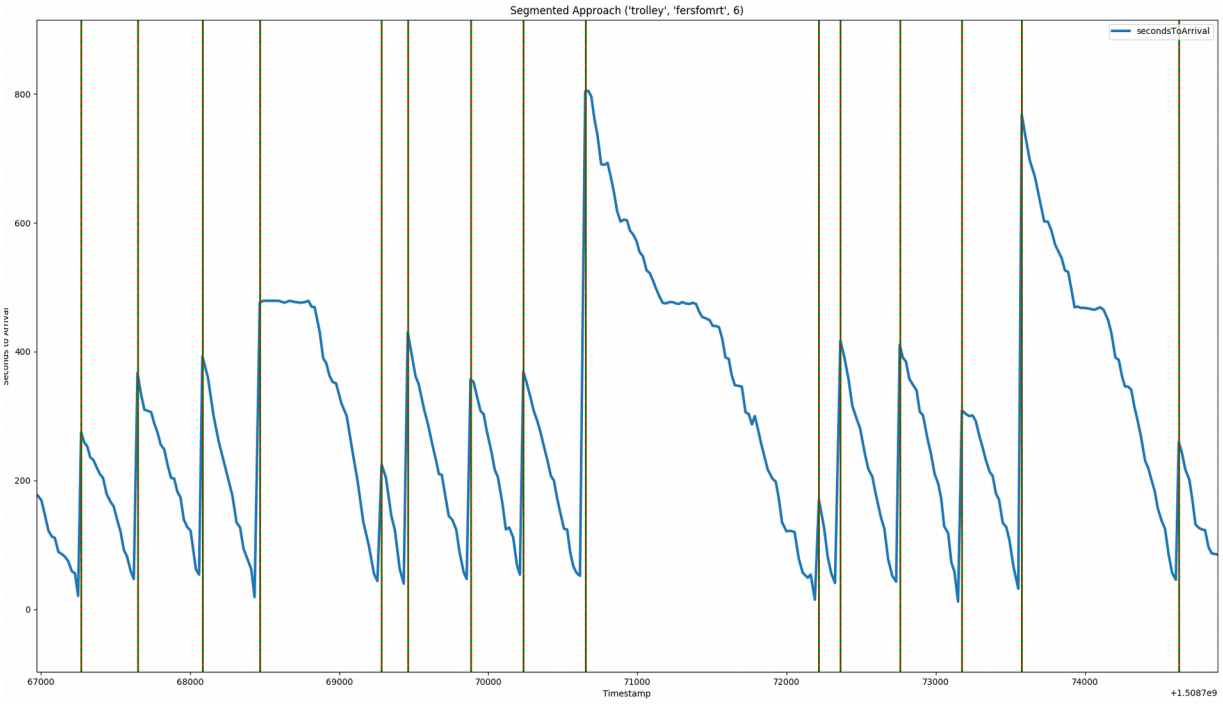
The most intuitive and sensible approach to detecting when an arrival event occurs is by looking for points where the distance of a bus from its next stop reaches zero. Distance from stop is not a metric that NextBus provides us, but it is something we can calculate using Georgia Tech domain knowledge, specifically the latitudes and longitudes of all stop. Yet despite the fact that distance is a direct measure of bus arrival, there are several reasons we cannot use it.

1. We are sampling once every twenty seconds, so it is very easy for us to miss an arrival event occurring.
2. Buses turn unexpectedly on their way to a stop. This leads to random spikes in the distance from the stop, making it harder to identify a spike caused by a bus leaving a stop.
3. There are at least 150 different route/stop combinations, so classifying certain spikes as arrivals is a tedious effort.
4. GPS data from buses is imperfect and delayed. NextBus mentions this shortcoming in its API docs: "It should be noted that sometimes a GPS report can be several minutes old." [9].

The following image is an example of a bus' distance from its next stop plotted over time. This figure was generated using the `plot_distance()` function of `analysis.py`. Note how there seem to be irregular spikes in the data, and how that might complicate arrival detection.



Instead of using a direct metric like distance to detect predictions, I looked for spikes in NextBus' predicted arrival time. The downside to this approach is that prediction time is an indirect measure of the real world, and it relies on NextBus' to produce sensible prediction. To counteract this assumption, I also created a test to validate the quality of my detected arrivals, which I will discuss in further detail later on. The benefit of using prediction data is that it is continuous, coherent, and simple to interpret. Using a simple rule, or *heuristic*, we can segment continuous NextBus data into distinct arrivals. The exact heuristic that I have used is as follows: *If the last prediction time we have seen was less than 80 seconds to arrival, and there is a spike in prediction time by over 40 seconds, assume that an arrival has occurred.* Below is a plot of NextBus arrival predictions over time in blue, with vertical bars indicating where my heuristic detected an arrival.



Since this is a derived metric, I needed some way to validate that my detected arrivals as correct. So to perform this verification, I used the following verification heuristic, which injects more domain knowledge into the task of segmentation: *If more than one bus is running on this route, then an arrival occurs when bus number changes. If only one bus is running, then an arrival occurs if there is a 300 second spike in arrival time.* I took a bitwise *and* of that secondary heuristic with my own, and found that my original heuristic has a 95% accuracy – that is, the secondary heuristic lines up with my heuristic roughly 95% of the time. The reason why the secondary verification heuristic is not used as the primary heuristic is that buses tend to disconnect and reconnect frequently, causing bus number to change spuriously. Creating a smarter and more accurate heuristic, perhaps using machine learning, is an open area of research for my project. It is critical that the arrival segmentation heuristic be as accurate and unbiased as possible, because an accurate data set begets an accurate model.

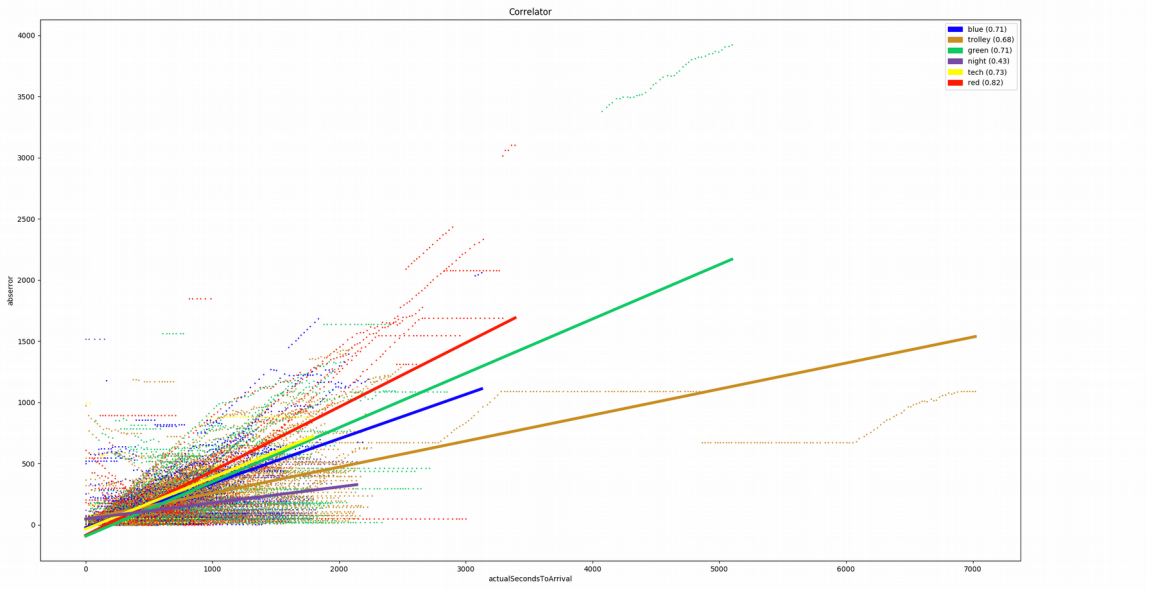
Another heuristic that was easier to design was the session heuristic, which detects when a bus route begins a new day of service. This was much easier to implement and justify – if there is a >10,000 second gap in data for a route/stop combination, assume that a day of service has just ended.

NextBus Error

Now that we have labeled all of our instances with their actual time to arrival, we can calculate NextBus' error for all predictions. Metrics of interest include the mean error, mean absolute error, root mean squared error, and median error.

- Mean error: 163 seconds under the actual time to arrival.
- Mean absolute error: 183 seconds.
- Root mean squared error: 370 seconds.
- Median error: 82 seconds.

Additionally, below is a scatter plot showing NextBus' absolute error vs. the bus' actual time to arrival. It is clear from this plot that the further a bus is from arrival, the more inaccurate NextBus' predictions are. The dots are color coded per route, as are the lines of best fit, all of which show a strong positive correlation. This was generated using the `plot()` function from `analysis.py`.

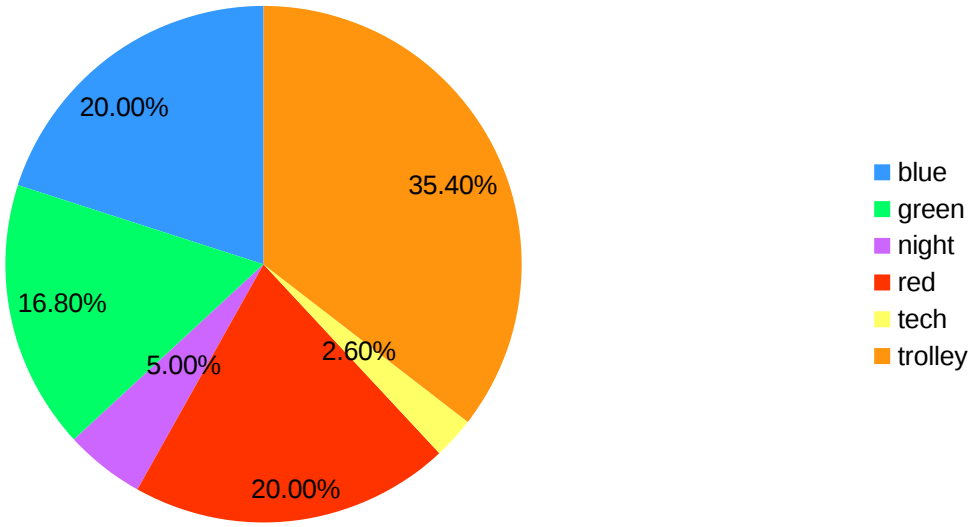


From these results, it is clear that NextBus' predictions are fairly inaccurate. One reason for this may be that NextBus intentionally predicts that buses will arrive sooner that expected, so that individuals get to bus stops early. Either way, there is a clear need for a more accurate model.

Other Insights

Before proceeding to the next section of this paper which discusses how a machine learning model was built, some tangential results of my in-depth exploratory analysis will be shown and discussed briefly.

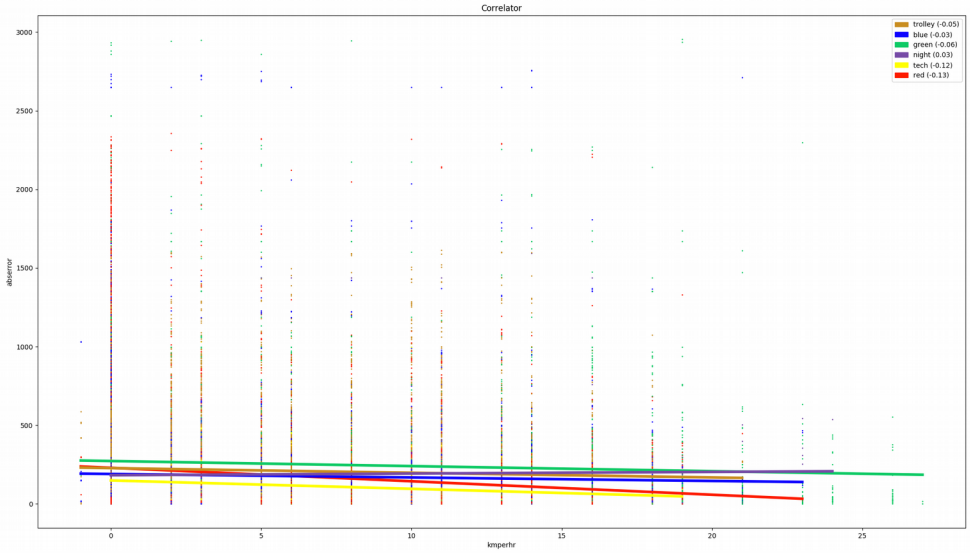
- **Frequency of each route**



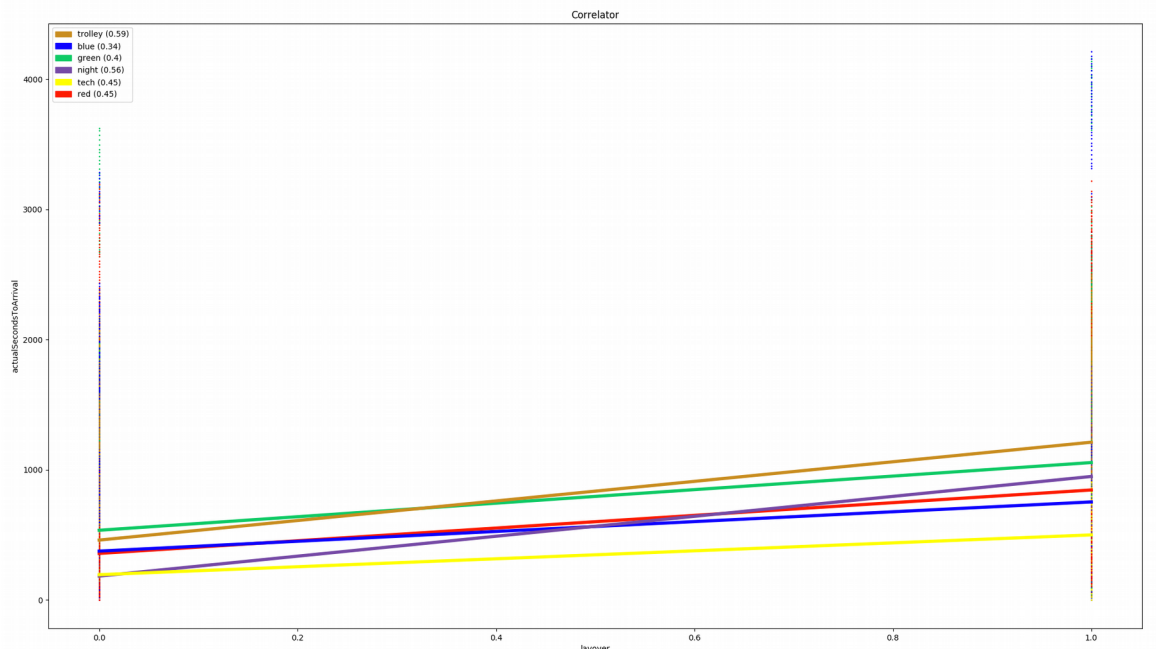
- **Loops per day**

  Buses make an average of 48.7 full loops in their route per day. In other words, if you stand at one stop for an entire day, you will see the same color bus pass you an average of 48.7 times.
  - Breakdown by route:
    - blue 59.252101 loops
    - green 40.522321 loops
    - night 13.121849 loops
    - red 72.267857 loops
    - tech 50.375000 loops
    - trolley 54.31428 loops
- **Actual time to arrival vs km/hr**

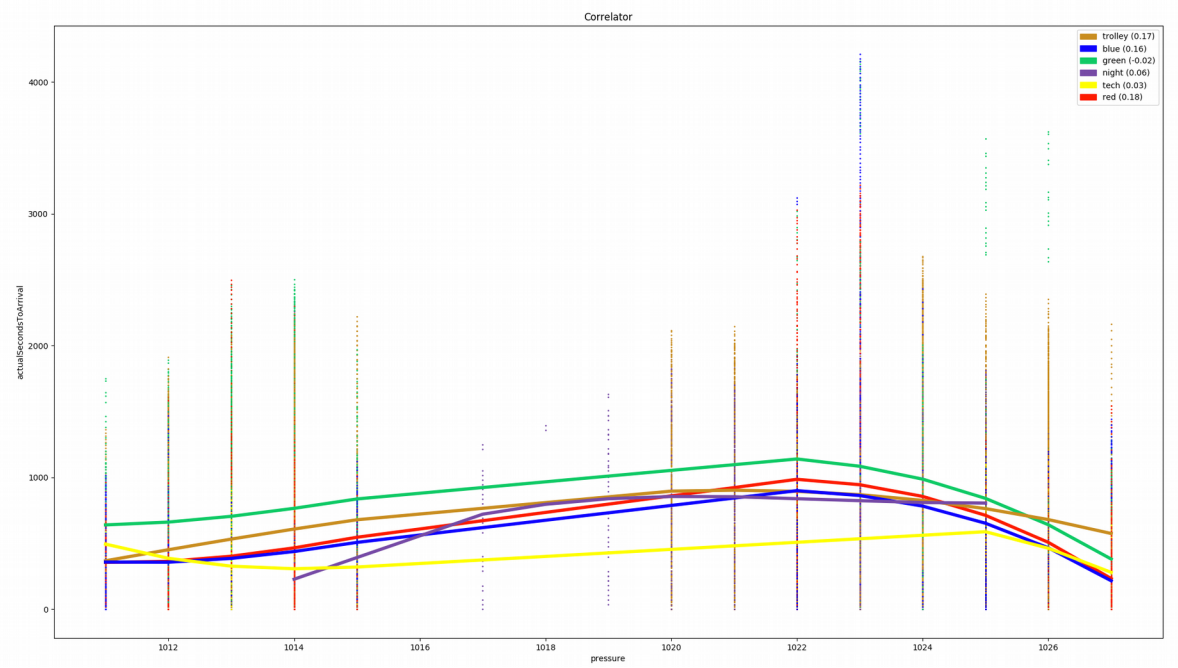
  As can be seen in this scatter plot, there seems to be a slight negative correlation; as speed increases, time to arrival tends to decrease.



- **Actual time to arrival vs. layover**
  As can be seen in this scatter plot, there seems to be a strong positive correlation; if a bus has layover, it takes longer to arrive.



- **Actual time to arrival vs. air pressure**
  As can be seen in this scatter plot, there seems to be some non-linear bimodal relationship between air pressure and actual time to arrival. These sorts of non-linear relationships were common for all weather-related features when plotted against actual seconds to arrival.



Modeling and Evaluating Predictions

Overview

Now that we have have identified that there is significant error in NextBus' predictions, the next step is to construct a machine learning model which can generate better predictions. Specifically, the goal is to build a regression learner which can predict bus arrival time as a float value in seconds. It does this by bootstrapping off of NextBus' mediocre predictions as a feature, as well as potentially using other features such as current bus location, bus speed, air pressure, and wind speed.

Model development was very much an iterative process. My first model had very poor performance, but over the course of iterative development in which I did model selection, hyper-parameter tuning, feature selection, feature engineering, and feature transformation, I was able to construct a model with significantly better performance than NextBus.

Data Preparation

At this point, we have accumulated 2 million or so rows in the sqlite database by running `collect.py` for twenty days. We must now clean this raw data and transform it into something that we can train a model on. To do this, we save our data as a CSV and load it from `pipeline.py`. This script in turn calls two other scripts, `cleaner.py` and `feature_engineering.py`, to manipulate the data further. The role of `pipeline.py`, specifically, is to do the following:

- Delete any duplicate readings.
  - We will have many duplicate entries in the database because we are sometimes querying for new instances faster than NextBus can update.
- Turn categorical variables into one-hot encodings.
  - A necessary step for learning off of categorical variable (non-numeric quantities, typically a string) is to transform them into a numerical encoding known as a one-hot encoding. Quite simply, for each distinct possible categorical value, create a new boolean column indicating existence. Consider this example from this project – the "route" column in the raw data can have one of the following values: "red", "blue", "green", "trolley", "night", or "tech". So for our one-hot encoding, we delete the original string-type column and create 6 new boolean columns of the form "is_red", "is_blue", "is_green", "is_trolley", "is_night", and "is_tech".
- Determine and mark arrival events.
  - The heuristic for arrival detection is discussed in the previous section.
- Engineer more features, injecting domain knowledge.
  - `feature_engineering.py` provides helper functions for synthesizing new features from existing features. The two classes of features that can be injected are time-related (`temporal(df)`) and Georgia Tech domain knowledge (`georgia_tech(df, gt_context)`)
  - `temporal(df)` is responsible for creating the following columns, based completely off the `timestamp` of that row:
    - `dayOfWeek` – Day of week where 0 = Monday and 6 = Sunday.
    - `Hour` – Hour in military time (0-24)
    - `month` – Month (1-12)

        - `minutesIntoDay` – Minutes since midnight
        - `isWeekend` – Boolean as to whether it is Saturday/Sunday or not.
        - `morningRush` – Boolean indicates whether the morning rush happens at this time.
        - `eveningRush` – Boolean indicates whether the evening rush happens at this time.
    - `georgia_tech(df, gt_context)` is responsible for injecting Georgia Tech domain knowledge as new features. It take an instance of `georgiatech.py`, which defines constants related to Georgia Tech that must be manually adjusted if Georgia Tech changes. Some examples of constants are the names of the routes, the colors associated with each route, and the latitudes and longitudes of each stop. It will create the following columns:
        - `distance` – the bus' distance from its next stop.
        - `classMode` – an integer indicating the current state of Georgia Tech classes at this time.
            - 0 = No classes at this time
            - 1 = Classes in session
            - 2 = In-between classes

Once all transformations are made, the data is saved to a CSV.

Model Definition

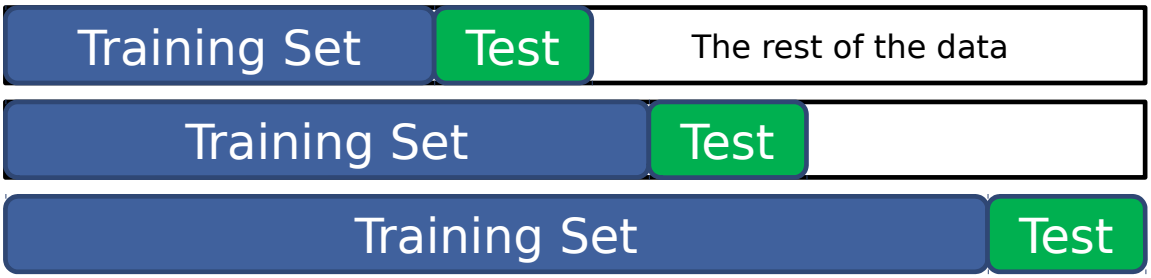
After passing through the data preparation stage, we can now use features and labels from our data for learning. This is initiated in the `train.py` script, which loads in a training data CSV, defines the parameters for a model (algorithm, hyper-parameters, features to use). It then makes a call to `supervised.py` with its data, and saves the returned model to disk. The details of the final model definition will be discussed shortly.

Model Evaluation

The most essential stage of the machine learning workflow is evaluation. `supervised.py` is responsible for carefully training and scoring models, providing metrics for in-sample (training) and out-sample (testing) error. `supervised.py` has the following roles:

- Splitting input data into training and testing sets.
- Normalizing the features.
    - Also known as feature scaling. This technique prevents outliers from overpowering learning and predictions by scaling them to the range of 0 to 1.
- Performing dimensionality reduction if requested.
    - Specifically, `supervised.py` has support for performing principle component analysis on the input data by calling `feature_transformation.py`. The transformed feature space can then be appended to the existing feature space or replace it.
- Training the defined model on the training set.
- Using the constructed model to predict values from the training and testing features.
- Calculating the error between predictions and actual labels.
    - The values for mean absolute error, root mean squared error, and median absolute error are calculated here.
- Averaging errors through rolling cross-validation.
    - We average the errors of each each training/testing iteration for our final cross-validated error metric.

Note how *rolling* cross-validation is used instead of just cross-validation. Rolling cross validation works by starting with some training data and some testing data for the first iteration, and then growing the training data and shifting the testing data for each additional iteration. The error metrics of each iteration are then averaged. The diagram below illustrates this approach.



We need to use rolling cross-validation as opposed to normal k-fold cross-validation because we have to worry about look-ahead bias. In time series data, look-ahead bias is a situation where some of your training data occurs after your testing data. This can never happen in the real world, and will artificially improve your model's performance. Rolling cross-validation enables some amount of cross-validation while not falling victim to look-ahead bias.

Final Model Performance

Here are the details of the model I converged on after many development iterations. To see my notes as as I tuned my hyper-parameters, check out `notes.txt` for a log of my accumulated inputs and outputs from my experiments.

- Model definition
  - Bootstrap Aggregation Regressor. This bagged learner consists of 150 Decision Tree Regressor base learners. Each base learner is trained off a randomly sampled 3% of the training set.
  - Features selected:
    - `route`
    - `stop`
    - `secondsToArrival`
    - `distance`
    - `layover`
    - `minutesIntoDay`
- Evaluation performed
  - Rolling cross-validation of partition size 10 and window size 5.
  - Error metrics:
    - Mean absolute error: 149 seconds (18.8% improved from NextBus).
    - Root mean squared error: 313 seconds (15.4% improved from NextBus).
    - Median error: 74 seconds (9.3% improved from NextBus).

Understanding the Final Model

There are several interesting phenomena I encountered while tuning my final model.

1. **Using weather data as features made my model worse.** This is because there was not enough in-sample variation in weather variables to successfully leverage weather variables out of sample. In order for weather variables to be useful, I have to collect more samples throughout the year during different seasons.
2. **Dimensionality reduction via PCA did not improve performance.** This suggests that the issue is not with the amount of data I have, but the variation and quality of it. I need more data, but not more of the same; rather, I need data that explores more of the possible state space.
3. I would frequently run out of memory when training complex models. The final model was over 800mb.

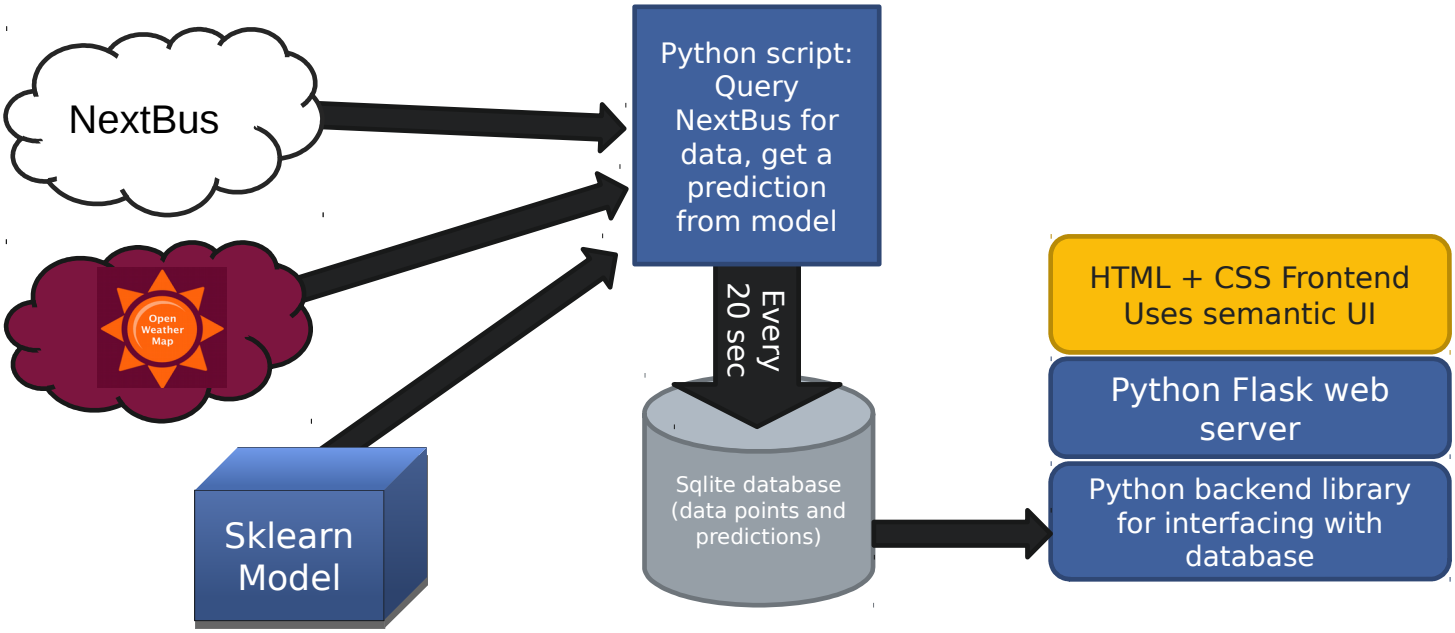
Web Application

Overview

Now that we have a model with modest predictive power, we can build a web app on top of it so that all Georgia Tech students can view the model's predictions and make use of them. Additionally, having a window into the model's real-time predictions can help identify its shortcomings, and we can then use those insights to design more resilient models. It is simple to swap out the model – just replace the model.pkl binary and restart the web server and collection service.

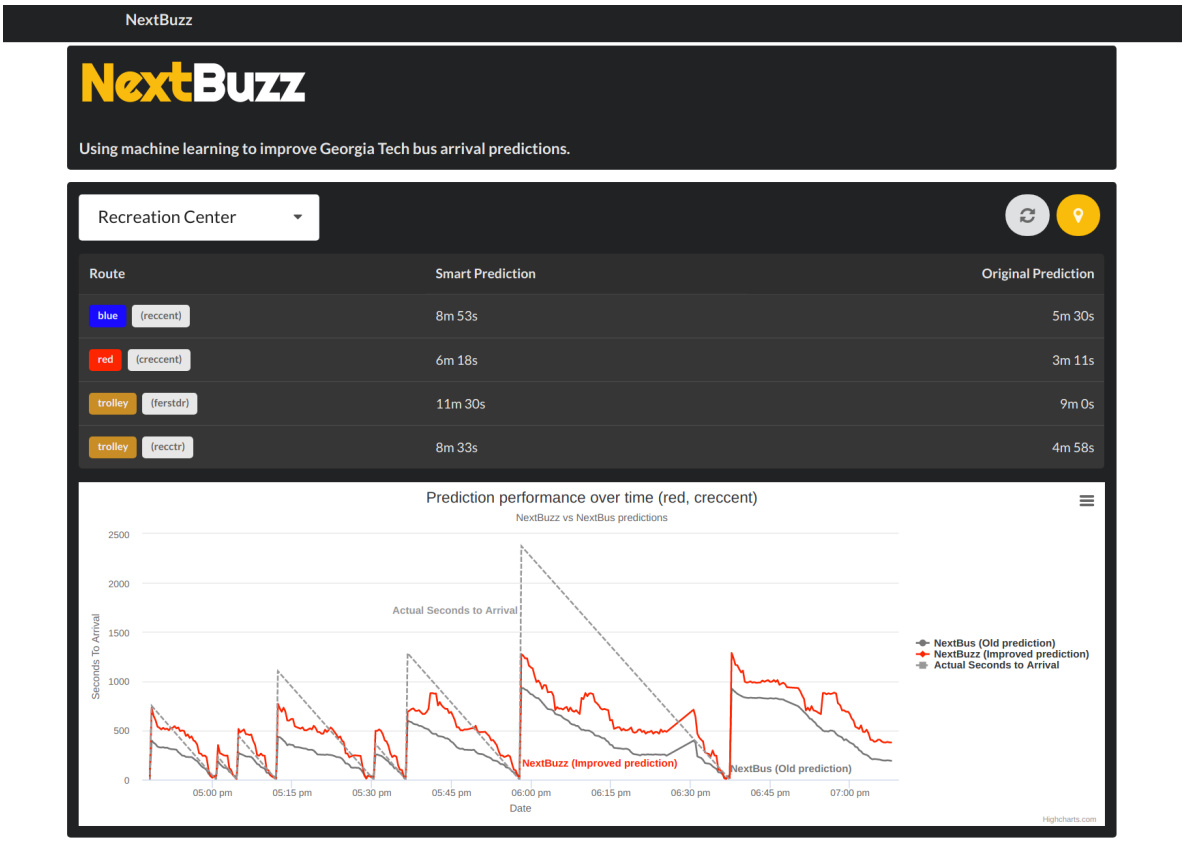
Application Architecture

The first step is querying NextBus and OpenWeatherMap for new data. Once acquired, this new data instance has a prediction assigned to it via model. The instance/prediction pair is stored as a row in the sqlite database. On the web app side, there is a Flask web server with a Semantic UI and Highcharts front-end. The web server has access to a back-end class, which can query the sqlite database for the most recent entries.



Application Interface

This application has a very clean and modern interface for locating bus predictions and analyzing historical model performance. Users can select the stop they want predictions for by selecting it in the drop-down, or alternatively can click on the pindrop button to select the nearest stop using Google Maps' geolocation API. Once a stop is selected, a list view populates with routes and prediction times, including both the NextBuzz and NextBus predictions. The user can click the refresh button to reload predictions for all routes approaching this stop. If a user clicks on a list item, a line chart is displayed of the past two hours of predictions with three lines: NextBuzz prediction, NextBus prediction, and actual time to arrival. Users can toggle these lines on and off by clicking on the legend. A live demo can be viewed at http://nextbuzz.ddns.net:8000



Future Work

- **Have the model automatically retrain itself.** In my proposal, I mentioned that the model should automatically retrain itself on the last month of data. From my experiments so far, it seems like models need to train on a much longer period than just one month, so automatic retraining is left unimplemented until more data can be collected.
- **Use my own predictions as features for bootstrapping.** Just as we leverage NextBus' weak predictions as a feature, we can leverage our own NextBuzz predictions as a feature for a dynamic programming style way of bootstrapping.

- **Add the Emory and NARA shuttle.** These are two lesser-known routes that operate from Georgia Tech that I should be analyzing.
- **Integration into GT Buses App.** I have been in contact with the owner of GT Buses, and he has expressed interest in integrating my model's predictions into the official Android/iOS GT Buses application.
- **Better way of detecting arrival events.** As mentioned earlier, the current method of detecting arrivals is through a rough heuristic. Ideally a smarter heuristic can be developed, perhaps through machine learning.
- **Deep Learning/Reinforcement Learning.** Other types of learning can be applied to this problem, and I would like to investigate that possibility in the future.

References

[1] https://redd.it/6veyy8
[2] http://www.sfexaminer.com/nextbus-muni-predictions-inaccurate-during-commute-hours-almost-half-the-time-study-says/
[3] http://www.bursar.gatech.edu/student/tuition/Fall_2017/Fall17-all_fees.pdf
[4] https://www.tommyleung.com/nextBus/nextBusBehindTheScenes.htm
[5] https://blog.goswift.ly/san-francisco-transit-prediction-accuracy-how-swyft-helps-you-commute-smarter-ab189cfedd71#.ldum05nhh
[6] https://www.nextbus.com/?a=georgia-tech#!/georgia-tech/
[7] https://play.google.com/store/apps/details?id=me.siddu.betternextbus&hl=en
[8] https://itunes.apple.com/us/app/gt-buses/id815448630?mt=8
[9] https://www.nextbus.com/xmlFeedDocs/NextBusXMLFeed.pdf