Urban Parking App

UNIVERSITY of WASHINGTON

Jake Garrison, Kyle Lee, Daniel Ng, Jiayu Dong

6/1/16

Table of Contents

List of Figures

A - Overview

A.1 - Abstract

Parking in urban areas is a global issue, and it leads to drivers spending an average of 20 minutes just searching for a parking spot, as well as increased pollution, and increased city traffic. Popular navigation apps, such as Google Maps and Apple Maps, do not address parking. Nonetheless, there are numerous web and mobile applications that do address parking. These applications inform the user on available parking spaces, and some have the option to reserve and pay for the spots in advance. Urban parking is a similar system that stores parking information in the backend server, with the frontend application suggesting and navigating the user to parking spots based on their final destination, parking availability, and user preferences. This innovation has the potential to reduce the time drivers take to look for parking spots, and reduce traffic and pollution as downstream effects. It can also provide city planners with better information on drivers' parking behavior

Figure 1 below shows the parking problem to downtown Seattle. Drivers search for parking in the same place, which causes places in downtown to fill up faster. Drivers in the same area looking for parking will cause surface traffic.

FIGURE 1: TIME LAPSE OF DOWNTOWN PARKING, DARK RED IS MORE CONGESTION

A.2 - Project Description

The goal of the project is to assist drivers by delivering consumable information to drivers helping them to park better. We do this by aggregating all the data, process it and presenting it in a website. The overall data organization structure is the given in Figure 2. Data from SDOT are put into a server (currently utilizing Amazon Web Services) and a custom Python API is built for web page and smartphone access.

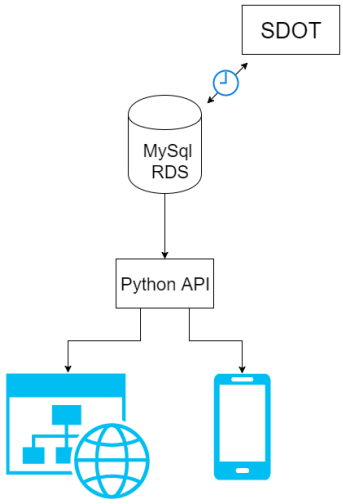


FIGURE 2: DATA ORGANIZATION STRUCTURE

The data is collected from SDOT and stored in a server (currently hosted by Amazon Web Services). Most of data cleaning and processing are done at this stage. A Python API is built to provide access to a webpage and an Android phone app. The database is updated daily with new transactions from SDOT. A Python script to query the database is used to obtain the following result

- Given a location (e.g. longitude and latitude) and a distance (e.g. 200 feet), the number of pay stations within that distance of the location.

- Given two sets of times, the number of transactions between these times at any particular pay station.
- Estimate relevant features such as the maximum occupancy associated with a pay station.
- Predicted parking densities, based on historical transactions and maximum occupancy.
- Routing instructions between two locations.

B - Data

The data provided by SDOT is retrieved from their public API at data.seattle.gov. Transaction data is defined as:

- Zone Desc = paid parking neighborhood by name (Text)
- Circuit Desc = paid parking subarea by name (Text)
- Payment Type = method of payment, should be Coins or Credit Card (Text); There may be other categories (chip one, etc)—those can be considered Credit Cards
- Server Time = time the transaction is uploaded by pay station (Time)
- Terminal Date = time of transaction in Pacific Time Zone (Time)
- Meter Code = unique meter identification code. The last two digits indicate the time limit of parking (02 = 2 hours). (Number)
- Amount = amount of money paid for the parking session (Number)
- Total duration in minutes = This field may include an extra 2-minute grace period or the prepayment hours before paid hours begin. Some older machines can only report time in 5 minute increments. (Number)
- Paid duration in minutes = Amount of parking time paid for. Some machines may report as rounded to the nearest 5 minutes. (Number)
- Address = location of the paid parking blockface (Text)

Transactions are communicated nighty, from pay stations to SDOT's internal database. This means that most recent data that can be retrieved will always be at least one day old. This represented a difficulty in trying to provide users with a real time view of parking availability. In addition, the times in which data would arrive was inconsistent, due to faulty pay stations, network outages, etc.

Blockface information was extracted from GIS shapefiles that SDOT uses for their own internal visualization. Blockfaces are represented as a pair of GPS coordinates indicating the beginning and end of the blockface. We chose to simplify this by using an average of these two coordinates. Maximum occupancy was also provided for most blockfaces, however as new stations go in and out of service, this data can become out of date quickly. These missing occupancy values can be determined by using a history of transactions at a blockface and taking the maximum number of open transactions at any given time as that blockfaces maximum occupancy. A combination of these two sources is used to create a more complete database of blockface information.

Potential inconsistencies in data may be due to events or construction. A list of such changes in blockface occupancy are documented by SDOT and updated semi regularly. Often times drivers leave their parking space before the end of the duration of their transaction. This can lead to over estimating the number of cars parked. Handicapped drivers or drivers parked illegally can lead to under estimation. These inconsistencies are difficult to measure without physically counting parked cars. In addition, a small number of transactions are attached to blockfaces that have no information or have no blockface at all. Transactions also sometimes arrive late or are missing. These are difficult to account for and hinder the ability of the machine learning model.

C - Back End

The back end encompasses all of the architecture needed to support the machine learning platform and user interfaces. This is built using a web framework, SQL database, and a number of recurring tasks. The core of this application lies in the database. The data stored within the database is used for all of the prediction and analytics. The web framework interfaces with the database to support the web front end and API. Finally, the real time nature of our solution warrants the use of automated tasks. Cron and Python enable this.

C.1 - Web Framework

The framework is built using Flask. Flask is an open source, BSD licensed "microframework" based on Werkzeug and Jinja 2. Werkzeug is a tool that provides many Web Server Gateway Interface (WSGI) utilities such as URL routing and HTTP header parsing. Jinja 2 is the templating language used for rendering front end web pages. This particular framework was chosen for its simplicity and extensibility. The back end runs on an Amazon EC2 instance using Amazon's Elastic Beanstalk, serving a RESTful API to front end clients. Ultimately, this serves as an intermediary between the database and the front end. Using REST API means that the back end can support any front end client with an HTTP connection. Currently, the API provides the following routes:

GET /blockfaces

Retrieves blockface information by key.

Parameters
element_keys: space separated list of blockface element keys

```
/blockfaces?element_keys=57349%2057350%2065545
```

Returns
json list where the keys are the requested blockfaces and the associated value is a list containing three gps coordinates representing the beginning, end and middle of a blockface and an integer value of the blockfaces maximum occupancy

```
{"57349": [-122.33055, 47.60182, -122.33083, 47.60202, -122.33069, 47.60192, 4], "57350":
[-122.33704, 47.60739, -122.33785, 47.60828, -122.33744, 47.60783, 5], "65545": [-
122.38249, 47.66685, -122.38324, 47.66645, -122.38287, 47.66665, 6]}
```

GET /blockfaces_in_radius

Retrieves blockface keys within a given radius of a point. The SQL query that backs this endpoint determines if a blockface lies within the circular area by using the spherical law of cosines.

Parameters
latitude: latitude value for the point
longitude: longitude value for the point
radius: radius in kilometers around the point

```
/blockfaces_in_radius?latitude=47.6097&longitude=-122.3331&radius=0.5
```

Returns
a list of element keys

GET /transactions
Retrieves a list of transactions within a time range

Parameters
start: start of time range
end: end of time range

Returns
a JSON list of transactions

GET /densities
Returns block density by time and key. "Density" is determined by the number of open transactions at an instance of time divided by the maximum occupancy of the blockface.

Parameters
at_time: time at which densities are requested
element_keys: space separated list of blockface keys

Returns:
JSON list where keys are element keys and value is density between 0 and 1

C.2 - Database

Data is stored using MySQL on Amazon RDS. This database stores all of the raw transaction data and pay station information as well as the analytics generated by the prediction algorithm. The schema for the database is shown in the following figure.

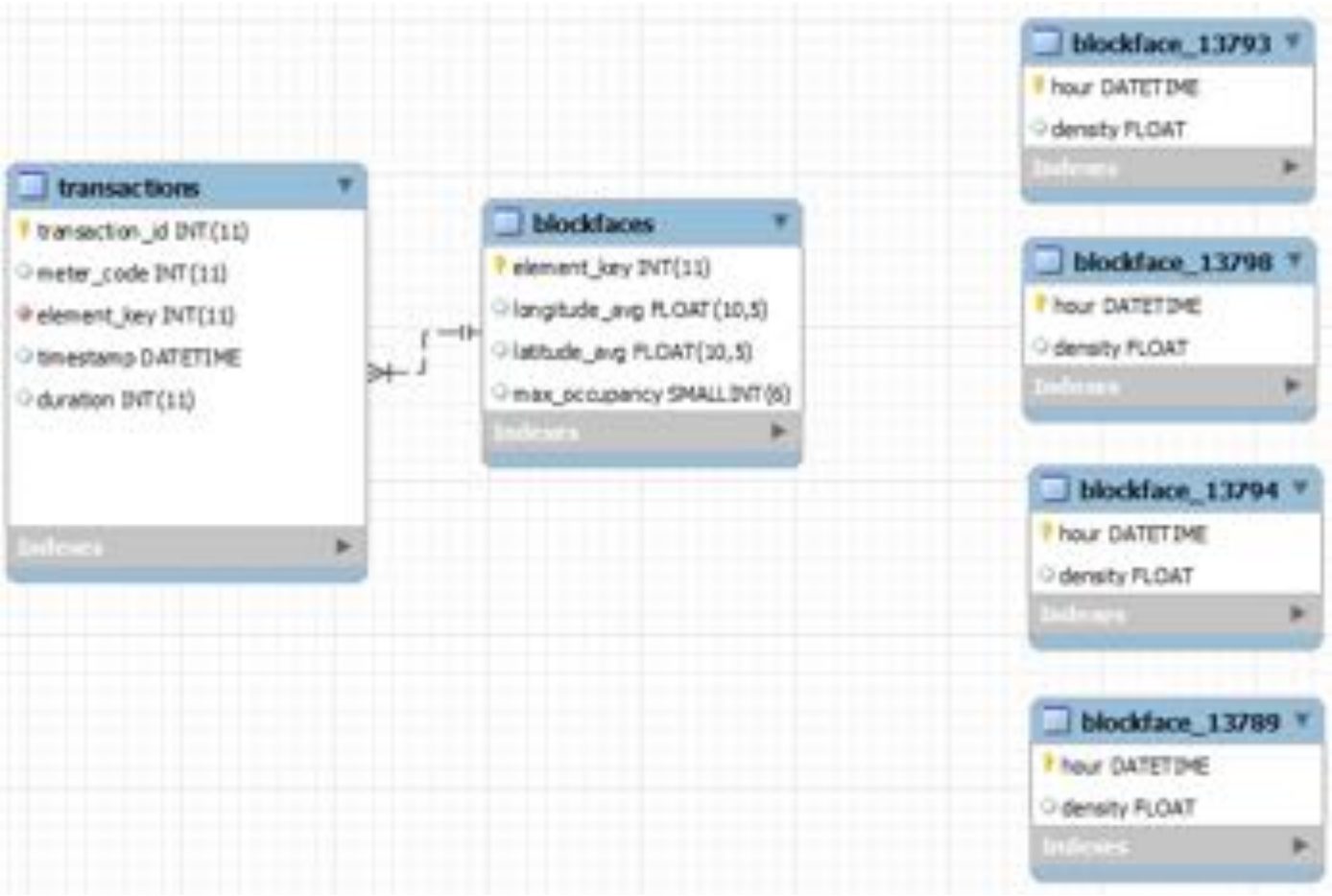


FIGURE 3: DATABASE DIAGRAM

Raw transactions are stored in the transactions table. Currently, this table holds a history of the past 3 years of transactions. Its columns hold information regarding the date and time of the transaction, the duration purchased, as well as the block that was parked on. The blockface information is stored as a foreign key to a table containing GPS coordinates for the block as well as its maximum occupancy. This table is indexed on element key as well as timestamp. Much of the queries that touch this database are

Additionally, data processed by the machine learning platform is stored in a series of tables, one for each blockface. These values are a normalized density for the block by hour of the day. This data is discussed in a later section.

C.3 - Cron Workers

An additional EC2 instance is used to host several recurring python jobs. Cron is used to schedule these jobs. The first is used to pull raw transaction from SDOT's public API. This worker runs daily and pulls all of the data from the previous day. The second worker is used to generate predictions on this new set of data.

D - Web Front End

The web front end was designed first because each member of the team has access to the browser and can use it as a development area. The web front end consists of 4 main areas, google maps, density chart, parking load and directions.

A Stable version of the website is regularly updated at http://parking-dev.us-west-2.elasticbeanstalk.com/ , we also bought the site urbanparking.xyz which links to our development site
The figure below shows what the website As of June 1,2016.

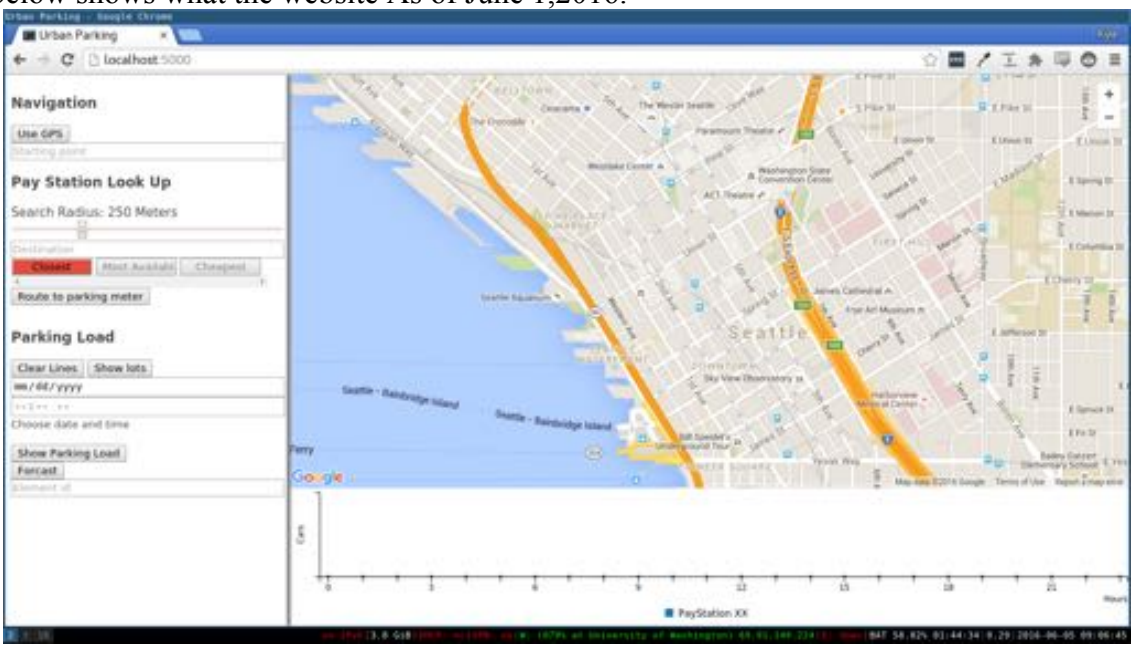


FIGURE 4: LANDING PAGE

D.1 - Navigation bar

The navigation bar allows users get routing directions to a parking spot within a set distance away from their destination. The user can input their start, destination and allowable walking distance, and we will display all pay stations within the range of the destination, both on the map and within a list. The list allows the user to quickly compare the top choices and choose which one to route to. After choosing which pay station, directions will be given to that particular pay station.

The destination and starting points both use google autocomplete forms and will find help find destinations in Seattle for you. Starting coordinates can also be input through the computers gps coordinates as long as the user permits it. Destination can also be picked by clicking on the map, a red marker and bubble surrounding it will be used to fill out the google forms.

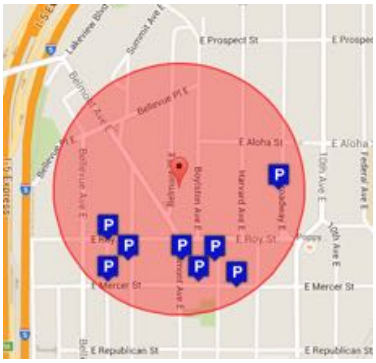


FIGURE 6: RADIUS BASED SEARCH

FIGURE 5: NAVIGATION PANE

Currently, there is 3 buttons set up for preferences, we would like users to be able to choose their pay station parking load, distance and cost. Right now only distance is set up, the rest are disabled placeholders. In order for availability to work, there needs to be a quicker way to grab all the densities of the paystations in area otherwise the UI hangs waiting for results. In order for cost to work, the KML cost maps from SDOT still has to be incorporated into the backend.

Directions

There is a built in JavaScript library for directions and display that allows user to get step by step directions with a specified transportation system and then display it on a google map. This method doesn't give us the flexibility to make multiple requests and choose alternative modes of transportation and compare results. We wanted to be able to see the difference between bussing and driving, and show that it could be faster, cheaper and even less walking than driving and persuade people to not drive. This would reduce surface traffic, pollution and give more parking spots to people. In order to get both bussing and driving directions and compare them, we had to make multiple response requests and parse the texts. The figure below, shows the resulting polyline when drawn from parsing the direction instead of using the display render.

FIGURE 7: ROUTE DISPLAY

D.2 - Displaying Density

Parking Meter chart

When a user clicks or hover on a parking meter spot either in the list or on the map, the parking chart will graph the predicted payload of the meter throughout the day. The User can select many parking meters to make an informed decision on which area to park. The map uses c3js a wrapper library for bar charts in d3.

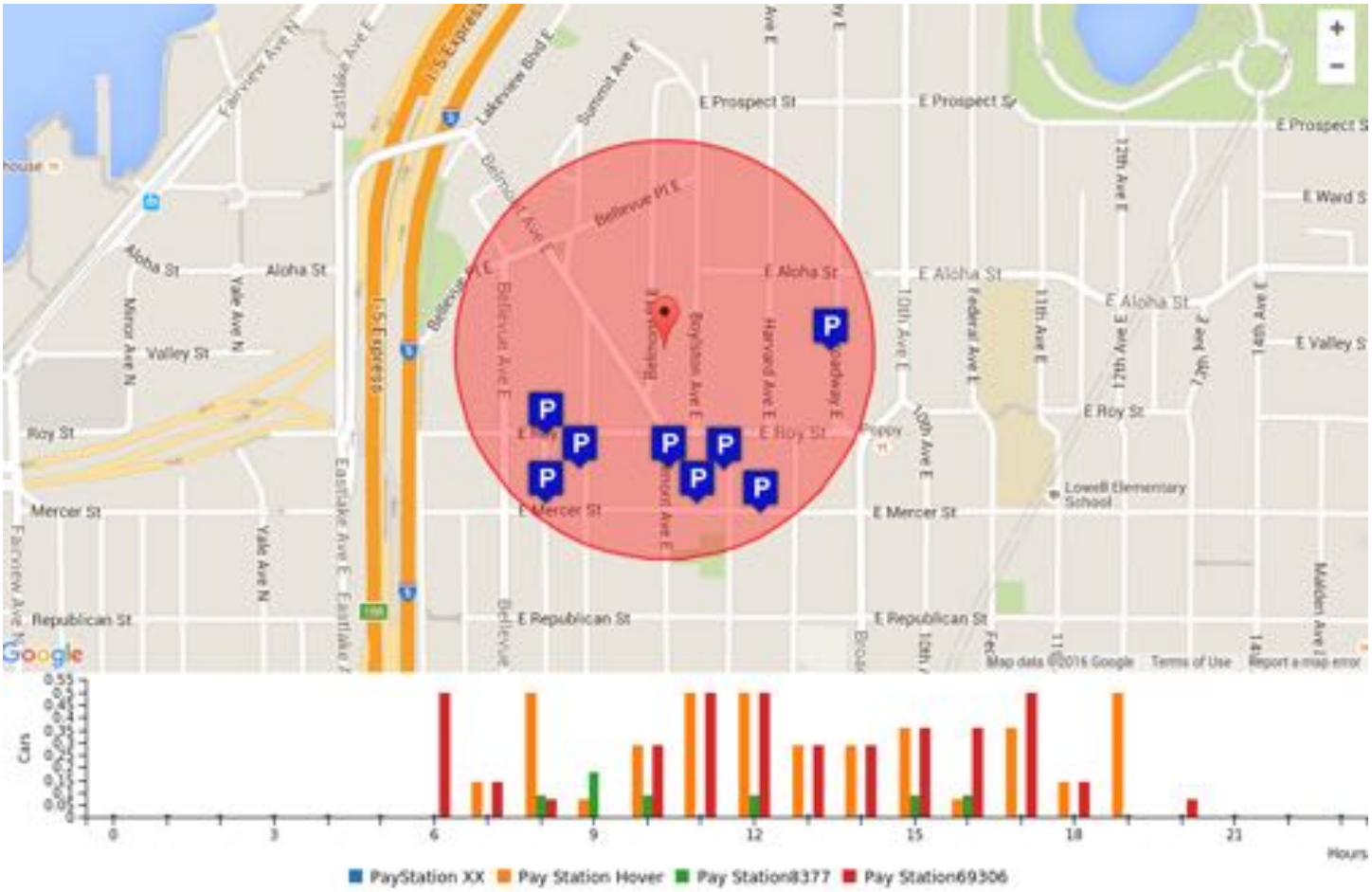


FIGURE 8: BAR CHART SHOWING EXPECTED PAYLOAD STATION OF CLICKED PAY STATIONS

Parking Load Heat Map

The transaction history can be explored in the Parking Load section of the web app. Historical and future dates can be viewed on a city level to provide insight on parking load distributed across the city. Currently our prediction algorithm is not part of the backend, so predictions are simple based off data from the previous year.

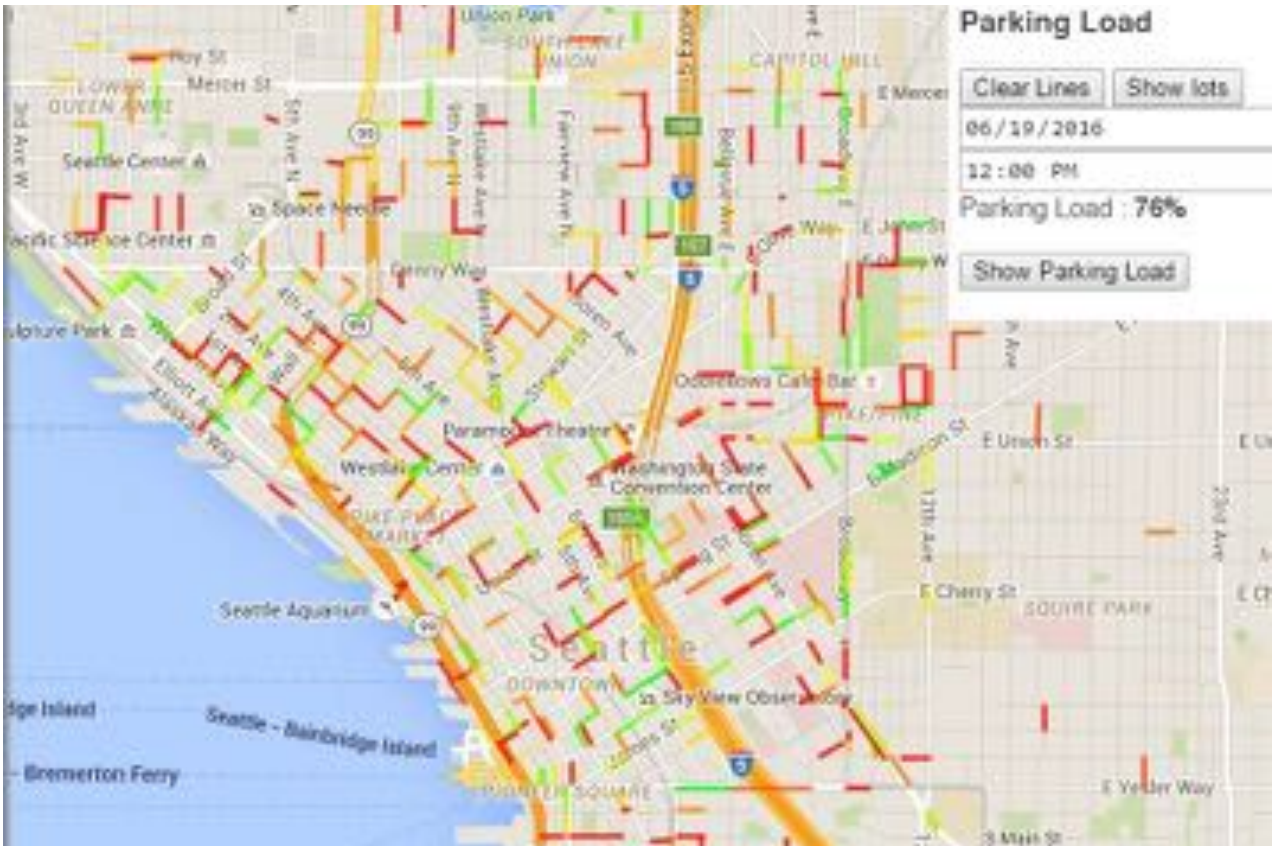


FIGURE 9: DENSITY PREDICTION FOR 6/19/2016 NOON. PREDICTION INDICATES A 67% LOAD

E - Mobile Front End

The mobile front end has the similar features as our website, which is designed on the Android platform using Java. The mobile app provides drivers a quick and convenient access to the current parking condition, so that they can make a more accurate decision. The mobile front end consists of 2 main activities, sign-in activity and map activity.

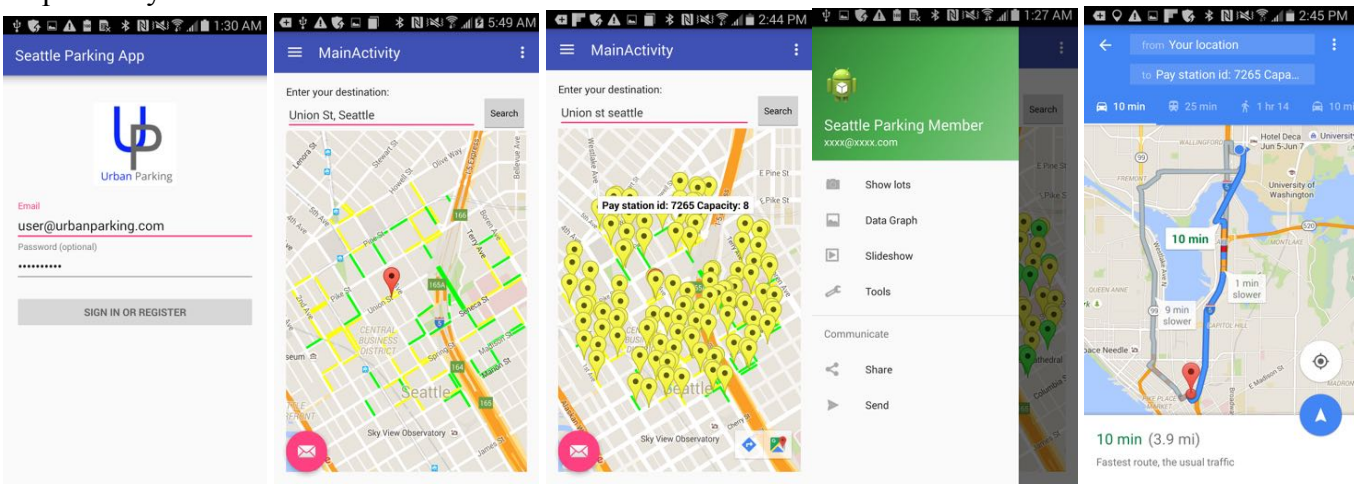


FIGURE 10 ANDROID APP SCREENSHOTS

1. Sign-in Activity

For potential commercial use in the future, a sign-in page is added to the mobile front end, which allows user to register or sign in to the app using email address and optional password. But, currently, there is no server to support this activity.

2. Map Activity

The map activity consists of a menu bar, a navigation menu, and a main console for google map. A text box is above the google map fragment allowing user to type in the destination. When user finish entering the address and click on the search button, the app will convert the street address to latitude and longitude using Geolocation. Then, zoom in the google map to the destination and plot the parking load heat map. The color will fade from red to green, which is from full to relatively empty.

The navigation menu provides some additional features. The function of show lots allows user to see the nearby pay stations and their max capacities. The icons in the bottom right corner are linked to Google Maps, which automatically set the destination to be the address of the selected pay station.

F - Prediction Algorithm

F.1 - Motivation

In order to provide an accurate estimation of parking load distribution across the city, a prediction algorithm was implemented. The motivation for this comes from the fact the SDOT pushes parking data somewhat randomly such that transaction data for the current day may not be available online for another week or two. It appears as though different pay stations push data at different frequencies, and for this reason city wide data is only complete up to about two weeks prior to the current day. Fortunately, SDOT stores data for the past few years, so past trends can be modeled.

The prediction algorithm analyzes historical data and correlates it with other features such as weather to forecast future trends for a given pay station. The algorithm relies on the Gradient Boosting regression technique [1] which utilizes decision trees that feed off a feature set to create a predication model. In addition to Gradient Boosting, other minor tweaks are applied to optimize the prediction for our application. One notable tweak is that days of free parking (as defined by SDOT [2]) are ignored since many holidays are hard to model as they may fall on different dates each year. The implementation relied on many of the open source Python libraries including Numpy, Pandas and Scikit-learn.

Though the algorithm is not yet integrated with our backend, the usage is roughly defined. A prediction window of size N is first specified, then for each pay station N days are forecasted based off the most up to date data at that moment. Once N days have passed, the prediction algorithm is invoked again to update the data for the next window. Predictions as well as historical data are stored in the database so the predictions can eventually be validated against the actual data. If the error is large enough, the algorithm can be re-parameterized. This could happen if construction disables a pay station, shifting the load elsewhere.

F.2 - Research

Several different algorithms were studied in order to create a predictive model. First, a simple nth order polynomial was fitted to the data and then extrapolated for forecasting. Another crude model involved just taking averages from historical data and repeating them in the future. After confirming these models were not complex enough, statistical models, specifically an autoregressive moving average model (ARMA) was proposed [3]. After significant tuning, this model was capable of decent results. Using the ARMA model as a benchmark, more advanced machine learning techniques were explored. The Python Scikit-learn libraries aided the benchmarking of several machine learning techniques from neural networks to simple regressive models [4]. The aim was to find something simple to parameterize and maintain, but powerful for solving the problem.

Boosting algorithms seemed good for this [5]. Gradient boosting in particular turned out to fit this demand and due to the ease of optimization and feature addition, it was selected for our predictive regression.

F.3 - Predication Example

In order to show the design process and effectiveness of the implementation, a working example for the pay station at 4727 44th Ave SW (chosen arbitrarily) will be explored.
To start, the data from 1/1/2014 to 1/1/2016 will be used for this historical data, and prediction will be focused on the month of 9/2015 (chosen so that prediction can be compared to actual data). The data for the defined window is shown below in Figure 11.

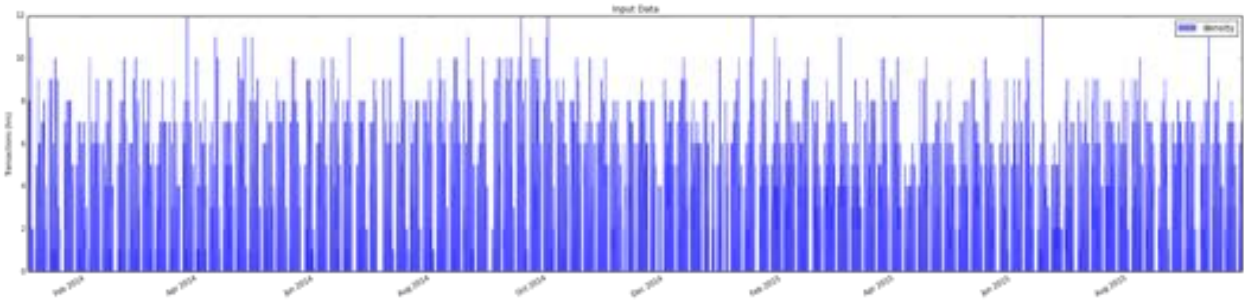


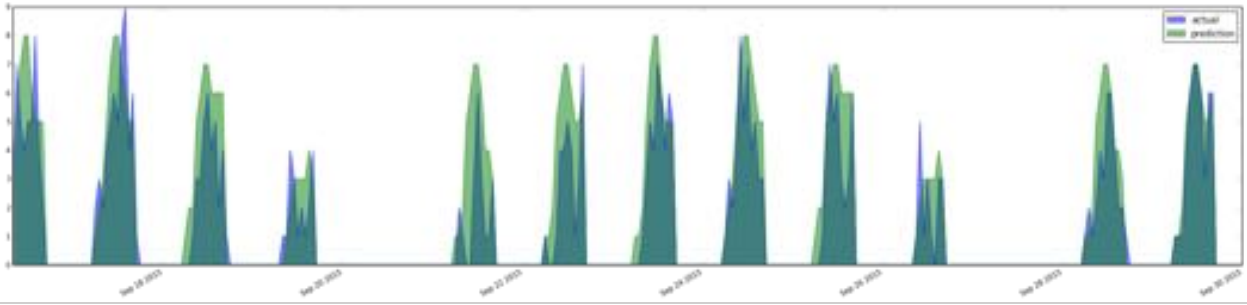
FIGURE 11: TRANSACTION HISTORY FOR WEST SEATTLE FROM 2015 TO 2016

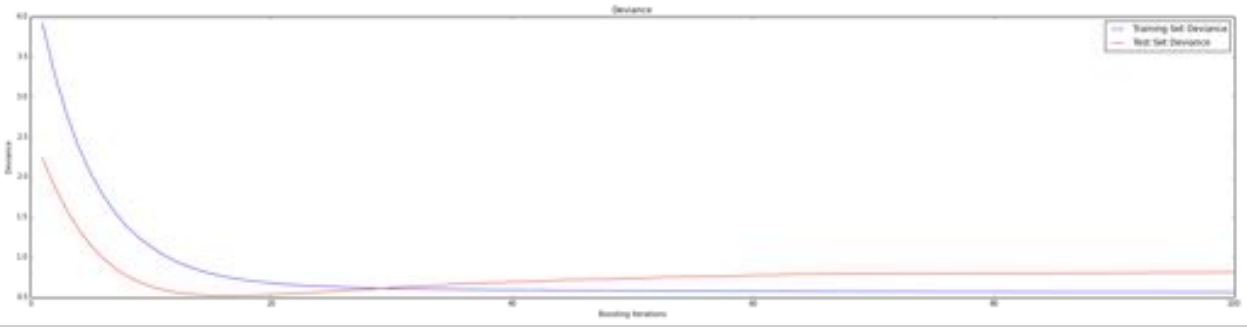
The historical data is shown as a histogram with hourly bins and indicates no more than 12 transactions occurred in a single hour. The peak time in terms of number of transaction is generally around noon. Note that his specific station has a max capacity of 8, so when 12 transactions occur in a single hour, multiple cars must occupy the same spot, meaning the duration for some cars was less than an hour.

F.4 - Results

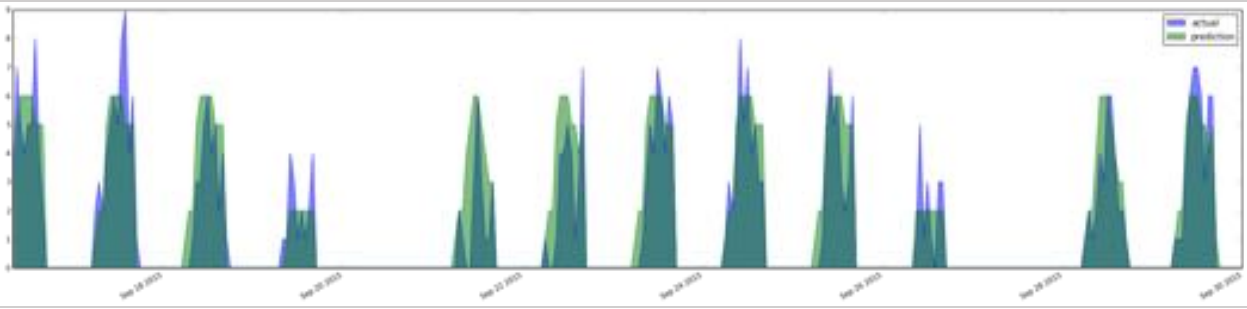
Before running the algorithm, model hyper parameters must be defined. The ones worthy of discussion are number of estimators, or the number of boosting stages to perform and learning rate, which how thorough each stage learns. There is a tradeoff between these two parameters that if set incorrectly can result in poor prediction. To show how these parameters affect the predication, two example are explored below. In Example 1, a learning rate of 0.1 and number of estimators of 100 is used, yielding a mean square error of 1.88 and R2 of 0.72. In Example 2, the leaning rate is lowered to 0.01 and the number of estimators is increased to 350, yielding a mean square error of 1.26 and R2 of 0.60. In general, a smaller learning rate requires more estimators to converge to a low error prediction and lowering the learning rate will increase the accuracy, but increase the processing time. The parameters used in Example 2 are found to be ideal for most pay stations.

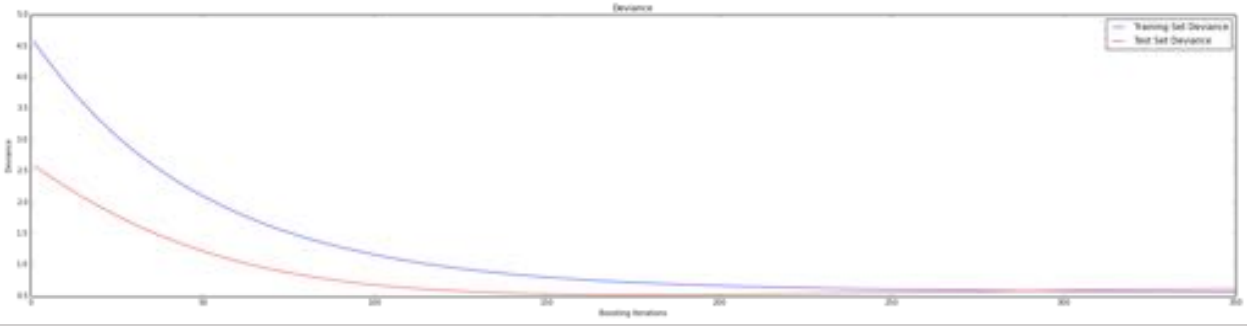
Example 1: MSE = 1.88, R2 = 0.72 (learning_rate = 0.1, n_estimators = 100)





Example 2: MSE = 1.26, R2 = 0.60 (learning_rate = 0.1, n_estimators = 100)





The pairs of plots above show the prediction overlaid with the actual data, as well as the deviance for each stage or iteration. Clearly as more stages are iterated, the error converges to a minimal value, but if too many stages are used (as in Example 1) the error actually will increase due to over fitting. Also, the learning rate is proportional to the decreasing slope of the deviance. A larger rate will have a steeper slope making the prediction faster, but more likely to miss the minimal error. The MSE and R2 errors are computed by comparing the actual with prediction result. To compare runtime, the average time for both example was computed to be 1.09 seconds for Example 1 and 3.28 for Example 2. Since this algorithm is only run once a week or so, the runtime is not a huge concern. A future feature could be to dynamically set the learning rate and number of estimators for each pay station. Below is a close-up of the last two days in Example 2. The MSE = .90 and R2 = 0.56. It can be seen that the general parking trend is modeled quite accurately, however this is not always the case do to random outliers.

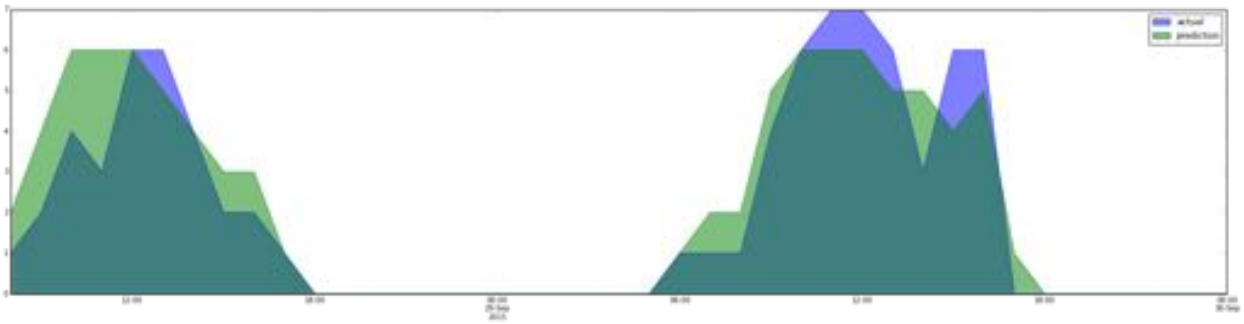


FIGURE 12: DENSITY OVER TWO DAYS

F.5 - Feature Analysis

In order to forecast, the historical data must be indexed into a feature set. Currently our features are outlined as such for each hour of historical data: Month (1 to 12), Weekday (1 to 7), Hour (0 to 24), Daily Rain (inches), Daily Mean Temp (F). Other features such as day of the month and year were considered, but increased the error or cause over fitting. Traffic data would also make an interesting feature. It was found that in parking spots near parks, the weather data was very correlated with parking load. These features define the decision trees used in the Gradient Boosting. The importance and dependence of the features can be analyzed to provide insight on their contributions. The plots in the figures below demonstrate this specifically for the case defined in Example 2 above.

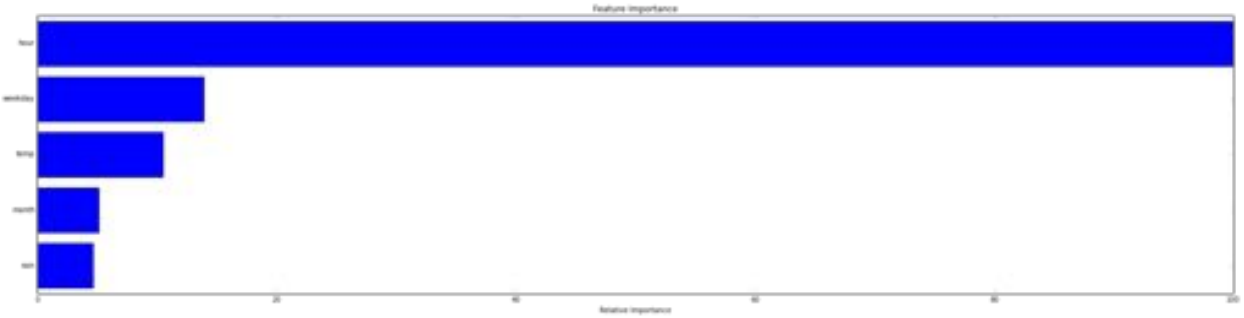


FIGURE 13: PREDICTION FEATURE IMPORTANCE

It is clear that for this pay station, rain doesn't not have much weight on the prediction, and as expected, the hour is the most important feature.

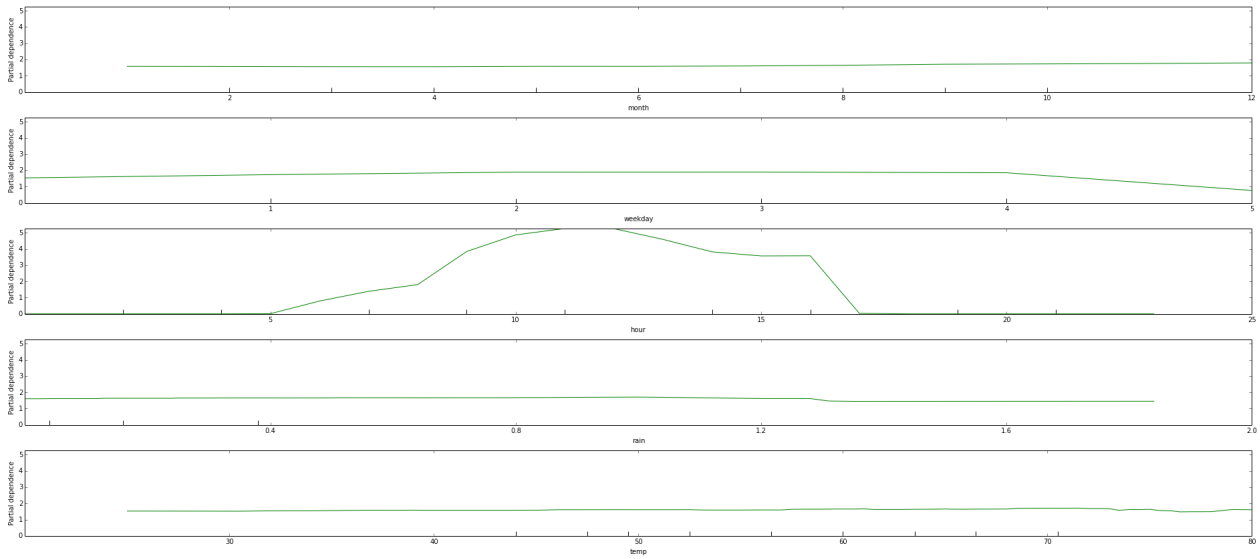


FIGURE 14: PREDICTION FEATURE DEPENDENCE

This plot indicates that there is no monthly trend, weekends generally have a lower density, mid day is the most popular, and more rain (> 1.2 inch) and higher temps (> 75) slightly decreases the density.

G - Team

G.1 - Expertise

Kyle expertise works in dealing with hardware with some interaction in software. He has taken a class in web design and interned as a web developer for a short time. Kyle has been on a robotics and the formula one race car team so he has experience in working with the unfamiliar and adapting to engineers of different disciplines.

Daniel comes from a software start-up background and had some experience with back end work. His strengths are in python and SQL. As a software engineer in the department of electrical engineering, most of his work is multidisciplinary in nature.

Jake comes from a diverse background of engineering. He was at one point an art student, then mechanical engineering, then finally settled with EE, specifically signal processing. He has used statistics and machine learning techniques in past data driven projects which makes him well prepared for drafting the prediction algorithm. Jake has also developed frontend interfaces for automotive vehicles, making him a useful aid in frontend tasks.

G.2 - Responsibilities

Kyle's responsibilities were to prototype a web interface primarily for the developers to use to test code and potential users to interact with.

Daniel's experience made it a clear choice to put him in the back end role. He would do any of the work needed for the supporting architecture.

Jake originally worked with Kyle on the frontend demo, then gravitated towards the prediction algorithm and related research.

Jiayu's responsibility was creating the mobile app based off the web app.

G.3 - Contributions

Kyle primarily added content with html and event interactions with JavaScript and jQuery. Classes and IDs have been placed on content allowing Some basic style to be applied to content through the website but this primarily allow a designer can come in and work on the aesthetics by adjusting the CSS. Kyle also helped on other affairs like editing presentation slides and status report.

Daniel took part in some of the initial R&D needed to understand managed all of the Amazon services that the application is hosted on. He set up the framework and database as well as the scripts used to populate it. Much of the API was also written by him.

Jake initially worked closely with Daniel on the usage of the density endpoints to create the frontend heatmap interface as well as functions to supply data to the histograms. Jake also aided Kyle in the UI tweaks and front end functionality. With the help of the rest of the team, Jake gave the pitch to CoMotion. Finally, Jake spent the bulk of his time doing R&D for the prediction algorithm.

Jiayu worked on the entire software development of the mobile end front. Based on the data extracted by the other team members, she developed the app on the Android platform using Java. Jiayu requested the Google API key and JSON lists provided by Daniel to accomplish her job.

H - Constraints

In our project, we are faced with making a product for anyone that drives. This means that these people come from vast backgrounds and have different levels of technological expertise. We had to design a user interface that others would understand intuitively by looking at it. There were a lot of iterations and adjustments that were slight improvements off each other. We tried to shape our maps to work similarly like google maps, because they are a huge company that a lot of people use.

Since we are building an app, many constraints due to physical complications do not apply. Our biggest internal constraint was not having as much experience developing web applications as other might. Another constraint was the data availability from SDOT. They are unable to provide real time, or even daily updates, which was the motivation for creating the prediction algorithm.

I - Timeline

Winter Quarter
Week 1 - 2: Project introduction, initial data analysis, meetings with SDOT and faculty
Week 3 - 4: Python experiments with data
Week 5 - 6: Backend and frontend started in python and javascript
Week 7 - 8: Early demos created with basic features, prediction research started
Week 9 - 11: Demos finalized, routing and density display added, CoMotion application
Week 11 - 13: Working frontend and backend, prediction algorithm started

Spring Quarter
Week 1 - 2: Bug fixing, prediction algorithm attempts
Week 3 - 4: CoMotion Pitch preparation
Week 5 - 6: Backend mostly finalized, Gradient boosting prediction demo

Week 7 - 8: AWS funding application, frontend UI enhancements
Week 9 - 11: Bug fixes, data analysis
Week 11 - 13: Poster presentation and paper

J - Future Work

- Front end:
  - Run UI tests and get a lot of people to use. Then hire a developer to do
  - use a library like bootstrap for columns and add styling to divs and text so that it works cross platform
  - Add routing options to compare bussing to give users more options.
  - Add pricing KML layers for parking area cost and do routing by price.
  - Use a more responsive library such as React.js for interaction elements
- Back end:
  - Integrate the prediction algorithm into database
  - Finalize database contents
  - Create error analysis metrics
- Prediction
  - Port to a more robust, saleable boosting library such as XGBoost [6]
  - Compare with neural network for predictions
  - Test on more pay stations, maybe tune model differently for each pay station
  - Integrate real time traffic as a feature for prediction
  - Social media based feature (hashtags or events near parking effect the density)

K - Citations

[1]J. Friedman, "Stochastic gradient boosting", *Computational Statistics & Data Analysis*, vol. 38, no. 4, pp. 367-378, 2002.
[2]"SDOT - Free Parking Days", *Seattle.gov*, 2016. [Online]. Available: http://www.seattle.gov/transportation/parking/freeparkingdays.htm. [Accessed: 30- May- 2016].
[3]2016. [Online]. Available: http://www-stat.wharton.upenn.edu/~stine/stat910/lectures/10_pred_arma.pdf. [Accessed: 12- Jun- 2016].
[4]"Documentation scikit-learn: machine learning in Python — scikit-learn 0.17.1 documentation", *Scikit-learn.org*, 2016. [Online]. Available: http://scikit-learn.org/stable/documentation.html. [Accessed: 12- Jun- 2016].
[5]A. Mayr, H. Binder, O. Gefeller and M. Schmid, "The Evolution of Boosting Algorithms", *Methods Inf Med*, vol. 53, no. 6, pp. 419-427, 2014.
[6]T. Chen, "XGBoost", *Dmlc.cs.washington.edu*, 2016. [Online]. Available: http://dmlc.cs.washington.edu/xgboost.html. [Accessed: 12- Jun- 2016].