Predicting Station States in Bike-Sharing Systems

Linsen Chong

Abstract

In this project, we develop and utilize several machine learning techniques to predict station state (inventory, full or not full, empty or not empty) for bike-sharing systems using public available datasets. Computational experiments suggest promising results.

Keywords: regression, classification, graphical model

1. Introduction and Problem Statement

This work is motivated by the efficient use of bike sharing programs in big cities, such as Washington D.C. and Boston. We are interested in inferring the time dependent demand patterns at all bike sharing stations. An interesting question arises: if a station is currently empty (i.e., no bikes are available) at this moment, will it still be empty in the next 15 minutes?

This question is of interest to the bike users as well as bike sharing companies. The users are interested when they want to rent a bike. If an user at the empty station is able to know that a bike (comes from a return) will come in a short while (e.g., 15 minutes), the user might be willing to wait for 15 minutes to get the bike. From the perspective of the bike sharing company, such as Hubway, based on our knowledge, the company needs to send trucks to reshuffle bikes due to the imbalance in supply-demand patterns at different stations. A reshuffle is to pick bikes at some full stations and send them to the empty stations. Therefore, to know exactly at what time, which station is full is helpful for the bike sharing company to determine the optimal strategy in order to minimize reshuffling cost.

In this project, we utilize the existing bike sharing data to make prediction for the demand in the near future. We use the number of bikes available at a station as the metric to represent demand. The fewer number of available bikes indicates higher demand. In the first part of the project section 2, we briefly introduce the datasets we use. In section 3, we use regression analysis of predict demand at different stations. Subsequently, we want to predict if a station becomes full or if a station becomes empty in the next time slot. Therefore, in section 4, we use classification method to group stations. We try different classification methods and compare their results. In addition, in section 5, we propose a graphic model to analyse the hidden connections between different stations. We summarize our findings, comment on the limitations and future work in section 6

2. Data Processing

2.1. Hubway Data

In this project, we focus on the Hubway bike sharing system of Boston. This is because their provided database include detailed time-dependent station-based data, i.e., the number of vehicles represent at a station at a specific time.

Hubway provides several datasets: trip history data, rebalancing operation data, station snapshot data and station location data. In this report, we use station snapshot data. The dataset includes station id, data collection time, number of bikes, number of empty docks and the station capacity. The data are collected every one minute at all stations. Note that the sum of bikes and the empty docks gives the capacity.

For the interest of the readers, we briefly introduce the other datasets here as well. The trip history data include the detailed information of every single trip (e.g., from the departure until the arrival). The most important information includes the departure station, the arrival station, trip duration and the time of departure. Rebalancing data consist of the number of bikes that were picked up and dropped off on a given day. This dataset does not include time or station component. Thus, it is impossible to know at what time does the truck pick how many bikes at which station. Therefore, we cannot tell the *actual* demand (i.e., the number of bikes in the snapshot plus rebalanced bikes) at a station from the rebalancing data.

Table 1: Features of the bike demand prediction problem

Feature	Description
bikesX t	Number of bikes at current time t at station X
bikesX $t-1$	Number of bikes at the previous time slot $t-1$ at station X
bikesZ t	Number of bikes at current time t at other stations Z
capacity	the capacity of bike stations
weekday	binary variable that determines the day was a weekday
hour	the hour of the day
quarter	the quarter of an hour (we collect data every fifteen minutes (every quarter))
rain	binary variable that determines if there was rain during t
drizzle	binary variable that determines drizzling during t
mist	binary variable that determines mist during t
fog	binary variable that determines if there was fog during t
temperature	air temperature in Fahrenheit at time t
humidity	Relative humidity at time t
visibility	visibility (in miles)
wind speed	average wind speed
holiday	binary variable to determine if it was a US holiday
daytime	binary variable to determine if it was daylight

2.2. Weather data

We believe weather is also critical to be taken into account for demand prediction. This is suggested by Regue and Recker [1]. Also, weather data are provided by the data challenge of Washington D.C. bike share system. Therefore, we use hourly weather data for weather station at Logan International Airport from the National Climatic Data Center [2]. The dataset includes weather type (e.g., drain, drizzle, mist, fog), temperature, humidity, visibility, wind speed, etc.

2.3. Data preprocessing

We preprocess the Hubway station snapshot data and weather data before we merge them together. Data are collected every one minute in the station snapshot dataset. In this project, since we are interested in predicting the number of bikes for the next 15 minutes, we take the average of the 15 data points to obtain the mean number of bikes of a quarter. In addition, as suggested by Regue and Recker [1], we pick data from Sunday May 6th to Sunday July 29th, 2012 during which the number of working station was constant. Moreover, we believe the supply-demand imbalance scenario is more likely to occur during rush hour. A clear sign can be founded in the bike station at North station where the probability of it being empty is high. This is probably because a lot of people who use bike to commute start to come back home from work. In addition, in the weather type record of the weather dataset, there are several different weather types (e.g., rain, fog, drizzle). We use a binary coding scheme to process each weather type.

For the weather data, since it is recorded only every one hour and we have four data points in one hour (we group the station snapshot data into 15-minute intervals), we assume the weather conditions to be the same in the four data points of the hour. This of course will result in inaccuracy. For instance, if a light rain (drizzle) occurs from 4:12 pm to 4:28 pm and the weather information is collected at 4:05 pm, the weather data will not be able to know the rain has occurred. This is a clear drawback since the majority of trip durations are less than one hour. Since weather related features are very important, this low resolution data may cause inaccuracy in our prediction.

We summarize the features in Table 1. We select features are based on Washington D.C. bike sharing data challenge and Regue and Recker [1]. Note that we assume the number of bikes at different stations are correlated in time. Thus, we put bikes at other stations into our feature space.

As a result, we use 51850 data points, that is 85 days. We choose 70 percent of data for training and validation and 30 percent for testing. We use the first 60 days data for training and validation (we use 20 percent of data in validation). The rest 25 days are for testing. We normalize the data before running our regression and classification algorithms.

In addition, we notice that the magnitude of the features are very different. Some of the features are binary (e.g., rain, fog), but some features are not. Therefore, if these raw data are used, numerical problems might arise, especially when computing kernels. Therefore, we normalize numeric features and use binary coding to transform categorical features (i.e., hour, quarter). We did not process binary features.

3. Regression

We apply three different regression techniques: Ridge Regression (with regularization parameter λ), Ridge Regression with a Gaussian Kernel (with regularization parameter λ and bandwidth β) and Lasso (with regularization parameter λ). We build regression model for each station. We first use a 5-fold cross-validation on the training data set to select the best parameters. Regularization parameter λ and bandwidth β are chosen from $[10^{-4}, 10^{2}]$ evenly based on a log scale. We then apply these three models on the testing data set to evaluate the prediction performance. Figure 1 depicts the minimal MSE getting from cross-validation on training data set. Ridge regression with Gaussian kernel has the least in-sample error (station-wise average MSE=1.712) which does make sense because it has the most expressibility. Lasso is the next to the least one (station-wise average MSE=1.831), and pure Ridge has the largest in-sample error (station-wise average MSE=1.926). However when we generalize the model on testing data (Figure 2), it turns out that pure Ridge regression has the most accurate prediction power (station-wise average MSE=3.290). Lasso is the next one with station-wise average MSE, 8.621. Ridge regression with a Gaussian kernel has the worst out-of-sample performance (station-wise average MSE=9.655). This discrepancy between in-sample and out-of-sample error might be caused by the fact Ridge regression with Gaussian kernel overfits the training data too much.

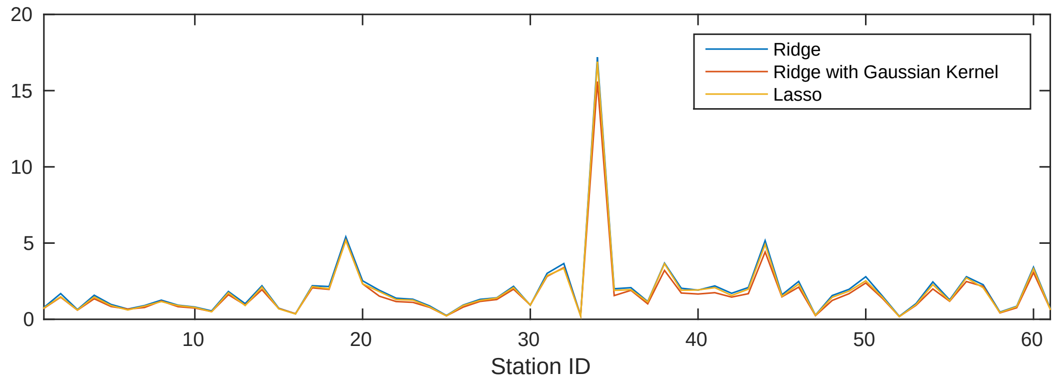


Figure 1: Minimal Cross-validation MSE of Training Data Set for All 61 Stations

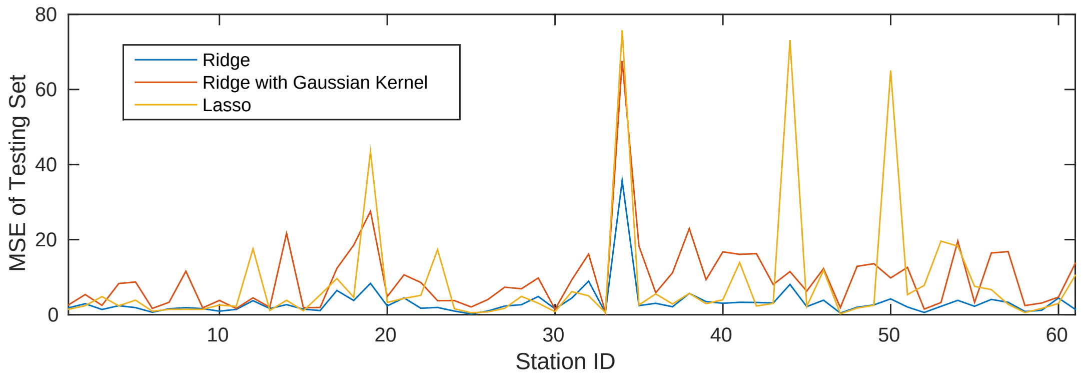


Figure 2: MSE of Testing Data Set for All 61 Stations

Implementation notes We use Python scikit-learn package to implement these three models.

4. Classification

We are interested in two types of binary classification problems :1) classify stations to be full and not full; 2) classify stations to be empty and not empty. This is because some stations are more likely to be "arriving" stations(e.g., Boylston / Mass Ave). They are more likely to be full than the others. Thus, our interest is to know whether the station will be full or not. On the other hand, some stations (e.g. Tremont St / West St) are used as "departing" stations. Our interest is to know if the station will be empty or not. Again, in this section, our feature space is at current time t and our prediction is at time $t+1$.

Since it is very rare that a station can be full (i.e., the number of bikes equals the station capacity), We use the ratio of the number of bike to the station capacity to determine an empty or a full station. If this number is greater than 0.8, we consider the station is full. If the ratio is less than 0.2, we consider the station is empty.

4.1. Basic SVM classifier

We first run experiments on the whole dataset to predict the class of all stations. Thus is, we use one single classifier to predict the class for all stations. We choose SVM classifier (with Gaussian kernel) to determine whether a station will be empty. As a result, the misclassification error on the validation data is quite high (>0.3). (We omit the details due to space limit). We think this high error may due to the fact that different stations treat features differently. For instance, the bikes of a station at time $t+1$ may be heavily depend on the number of bikes at its neighbouring station at time t, but not depend on the number of bikes at stations that are far away. Thus, we think the SVM classifiers should be station based, i.e., we should train an SVM classifier for a station.

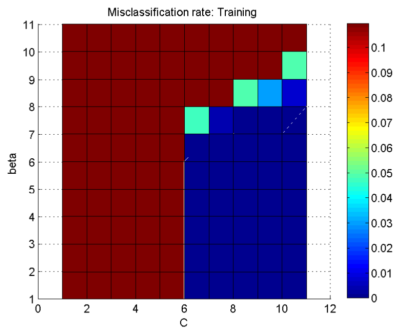
For the rest of this section, all the classifiers we trained are station based.

4.1.1. Station-based SVM classifier

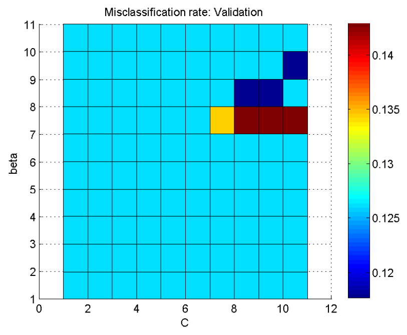
In this part, for the purpose of illustration, we pick two specific stations, namely station at Boylston / Mass Ave and station at Tremont St / West St. As we mentioned briefly earlier on, we consider station Boylston / Mass Ave as an arriving station, where our interest is to know whether it will be full and not full. On the other hand, station Tremont St / West St is a departing station. Our goal is to predict whether it will be empty.

We use 80 percent of training/validation data for training, 20 percent for validation. We use Gaussian Kernel with soft margin SVM method. We tune the two parameters, namely cost C and bandwidth β according to the minimum cross validation results. We then test the performance of the best SVM classifier (the one with minimum cross validation error) using the test data. We use heatmap where x and y axis present the range of C and β, the color represents misclassification rates.

Figure 3 shows the training error and validation error of the SVM Gaussian Kernel classifier on the arriving station where the problem is to classify if a station is full or not. We choose β from $\beta = [10^{-5}, 10^{-3}, ..., 1, ..., 10^5]$ and $C = [10^{-5}, 10^{-4}, ..., 1, ..., 10^5]$. (Note that the numbers on the x axis and y axis are labels, not actual C and β values. i.e. $C/\beta = 1$ in the axis is equivalent to $C/\beta = 10^{-5}$.). The minimum validation error is 0.1176, when $\beta = 0.001$ and $C = 100$. We use test data for prediction and we get the misclassification rate to be 0.1606.



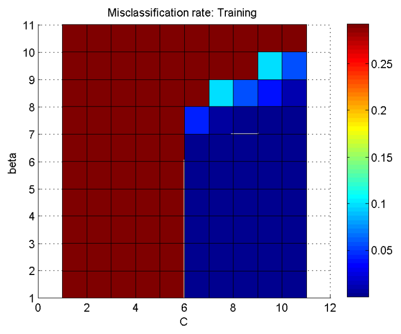
(a)



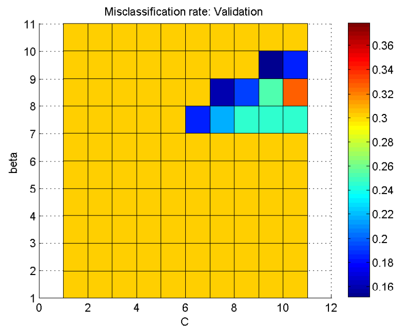
(b)

Figure 3: Misclassification rate of LR Gaussian Kernel classifier, station Bolyston/ Mass Ave

Figure 4 shows the training error and validation error of the SVM Gaussian Kernel classifier. This time, we predict if the station will be empty. We choose β and C from a logarithm scale, same as Figure 3. The minimum validation error is 0.1513, when $\beta = 0.001$ and $C = 1000$. We use test data for prediction and we get the misclassification rate to be 0.2118.



(a)



(b)

Figure 4: Misclassification rate of LR Gaussian Kernel classifier, station Trement St/ West St

As a result, we show that the station-based classifiers outperforms the basic SVM classifiers, which shows the need to train station based SVM classifiers. This make sense, since the bike of a station probably depends heavily on its neighbouring stations.

However, tuning the "best" C and β is a computationally expensive task. This is done through grid search in this project, i.e., we simply enumerate over all possible C, β combinations. This could be time consuming when data become large, although in this project we only look into a subset of data and do not encounter computational issues.

4.2. AdaboostSVM

The motivation of this method is in two folds: 1) we are interested in using an additive model to hopefully a better classifier (hopefully majority votes will be better); 2) we hope the additive model is not computationally expensive to be found. This is because by design, Adaboost needs only a weak classifier; and the parameters of the weak classifier needs not to be fine tuned as long as its training error is less than 50 percent. Therefore, we can arbitrarily assign C and β to build a weak classifier. Since parameter enumeration is no longer needed, we hope Adaboost algorithm is more computationally efficient.

We use SVM classifier with soft margin as the weak learner. Then the task of each weak learner is to classify the weighted data. To do this, in the primal form of the soft margin SVM problem, we use a weighted penalty $C\sum_i w^{(i)}\xi_i$ where $w^{(i)}$ is the weight of sample i. When transformed to the dual problem, these weights will occur on the upper bound of dual variables α_i.

The formulation of the SVM dual problem is given as follows:

$$\begin{aligned} \max_{\alpha} \quad & \sum_{i=1}^{n}\alpha_i - \frac{1}{2}\sum_{i,j=1}^{n} y^{(i)}y^{(j)}\alpha_i\alpha_j(K(x^{(i)},x^{(j)})) \\ \text{s.t.} \quad & \sum_{i=1}^{n}\alpha_i y^{(i)} = 0 \\ & 0 \le \alpha_i \le Cw^{(i)} \end{aligned}$$

where w_i is the weight of sample i. They are computed based on the traditional Adaboost algorithm (we omit the detailed equations here for the sake of simplicity), $K(x^{(i)},x^{(j)}))$ is the kernel function (K could be linear or Gaussian). The final classifier has the form:

$$G(x) = \text{sign}(\sum_{m=1}^{M}\alpha_M G_m(x)) \tag{1}$$

where α_M is the weight of the prediction of weak classifier m.

Since SVM is used as the weak classifier, there are some important issues to be addressed. If at some iteration m, the weighted samples become separable, then classifier vote α_m will be infinity. Then, the Adaboost algorithm will become too attached to the training data and not generalize well. Therefore, we 1) choose the initial β and

C such that weak learner do not separate data in the first iteration 2) we check the α_m for classifier m to see if infinity occurs and 3) we control the termination criteria.

Adaboost can be terminated on three occasions: 1) when an α_m becomes infinity 2) when an α_m goes to zero (this means the weighted error of classifier m goes to 0.5 because the weighted data are more difficult to be classified correctly, and thus, the weights are not updating any more) 3) when the cross validation error stops reducing. In this project, we control the termination by cross validation error, yet we check 1) and 2) after we terminate the Adaboost SVM algorithm.

We use two weak classifiers in this problem: SVM with linear kernel and SVM with Gaussian kernel. We choose cost $C = 1$ for both Adaboost models and we use $\beta = 0.002$ for the Gaussian kernel. These values are selected through trial and error.

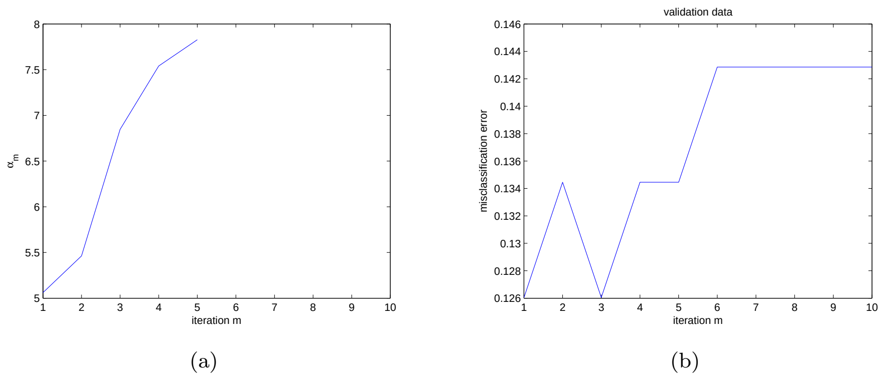


Figure 5: α_m and misclassification rate of Adaboost SVM linear kernel classifier, station Bolyston / Mass Ave

Figure 5(a) shows the α_m of the weak classifier in the Adaboost SVM (linear kernel) method for station Bolyston/ Mass Ave. α_m goes to infinity after the fifth iteration. Figure 5(a) shows the misclassification rate on the validation data using the Adaboost classifier obtained at iteration m. It shows clearly the optimal Adaboost classifier is obtained at iteration 3 (misclassification rate is 0.1261). Although the misclassification error on the training data can go to 0 at iteration 5, we stops Adaboost at iteration 3. We test the Adaboost classifier we obtained from iteration 3 on test data. We get a misclassification rate 0.2549.

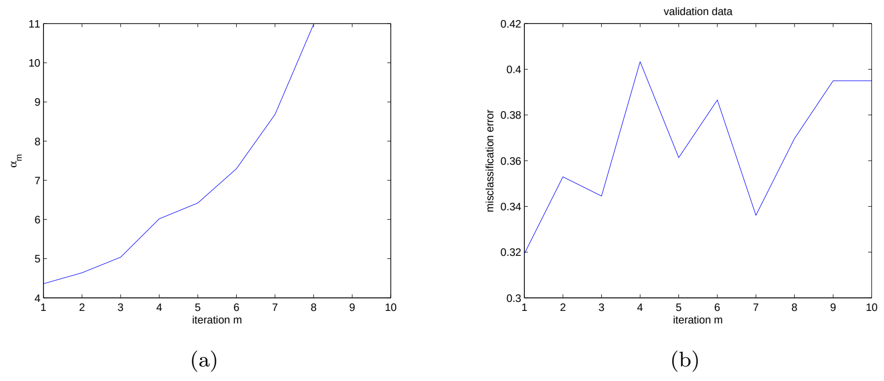
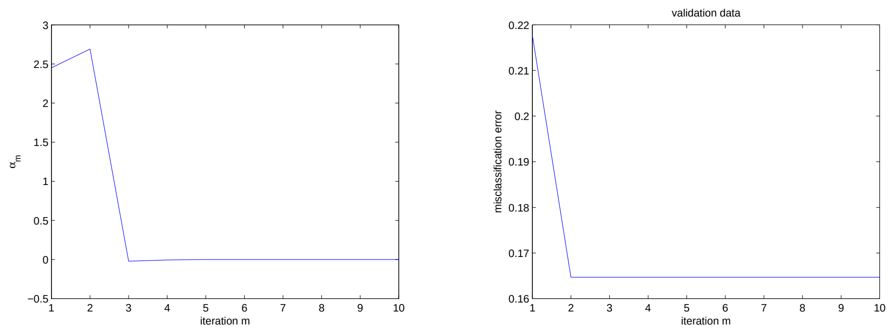


Figure 6: α_m and misclassification rate of Adaboost SVM linear kernel classifier, station Tremont St / West

In Figure 6, α_m is constantly growing yet the misclassification error on the validation data set is not. α_m reaches infinity at iteration 9. The minimum misclassification rate is 0.319 and is obtained at the first iteration. We put this classifier into test by using the test data. The resulting misclassification error 0.3765.

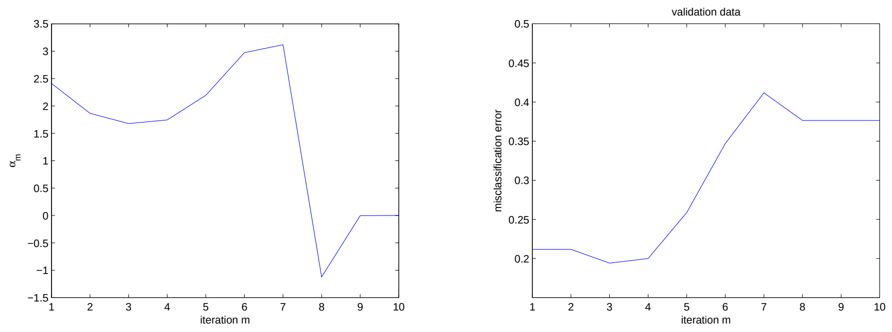
As a result, the weak classifier linear kernel SVM does not always work. Thus, we continue our experiment using Gaussian kernel SVM weak classifier.

Figure 7 and Figure 8 show respectively the performances of the Adaboost method with SVM Gaussian kernel for the two stations. In Figure 7(a), the α_m goes to 0 at iteration 3, which means the classifier stops improving. The respective cross validation misclassification error is 0.1647. When use this Adaboost classifier to test on the



(a) (b)

Figure 7: α_m and misclassification rate of Adaboost SVM Gaussian kernel classifier, station Bolyston / Mass Ave



(a) (b)

Figure 8: α_m and misclassification rate of Adaboost SVM Gaussian kernel classifier, station Tremont St / West St

test data, for station Bolyston / Mass Ave, the misclassification error on test data is 0.09, which is the best we can get so far. In Figure 8 (station Tremont St / West), the minimum validation error is obtained at iteration 3, with misclassification error 0.1941. The corresponding misclassification error on the test set is 0.1490.

In sum, in terms of misclassification error on test set, the AdaboostSVM method using Gaussian kernel as weak classifier outperforms Adaboost SVM using linear kernel method. It also outperforms basic SVM classifier with the best parameters. Thus, this method shows the added value of boosting. Also, since it needs only three iterations to find the optimal Adaboost classifier, it is very computational efficient. On the other hand, as a benchmark, to tune the optimal SVM classifier (without boosting), we try 121 iterations.

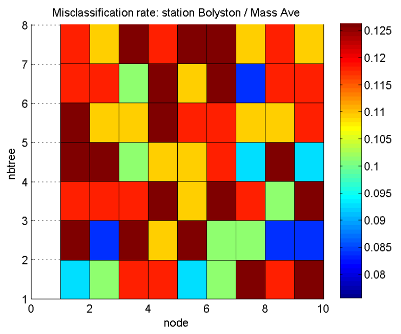
Implementation notes We would like to fine-tune each step of the adaboost SVM algorithm, therefore, we hand-code the weighted SVM dual problem using the formulation described in this section. This is essentially a constrained quadratic programming problem. We use Python package scipy.minimize to solve this optimization problem. Note the python scikit-learn package can automate this process. In this work, we hand-code the algorithm because we would like to tune the whole steps of the algorithm. For the station-based SVM basic classifier, we use the built-in SVM classifier in scikit-learn.

4.3. Random Forest

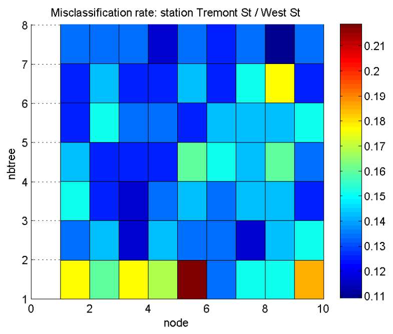
The feature space we use include the number of bikes at all stations at the current time t. We suspect that for one station, the bikes at its neighbouring stations should contribute more. This inspires us to use random forest algorithm for classification since it randomly selects features to grow a tree, and thus has the potential to select dominant features.

In random forest algorithm, we tune two parameters through validation, namely the number of trees and the minimum size of terminal nodes. The number of trees ranges from 50 to 400 (i.e., 50,100,150,...,350, 400). The number of terminal nodes ranges from 2 to 20 (i.e., 2,4,6,...,18,20). Again, we use heatmap to show the misclassification error of validation data. We use heatmap where x axis represents the number of trees, y axis represents the minimum size of terminal nodes. Note that x axis and y axis are labels, not actual numbers.

We show the misclassification error with respect to the validation data in Figure 9.



(a)



(b)

Figure 9: Misclassification rate of two stations

We summarize the performance of the random forest classification method as follows:

Bolyston / Mass Ave: the minimum misclassification error of the validation dataset is obtained when the number of trees being 150 and the minimum terminal nodes being 20. The corresponding misclassification error is 0.0756. When we use this random forest classifier on the test data, the misclassification error is 0.1254.

Tremont St / West St: the minimum misclassification error of the validation dataset is obtained when the number of trees being 350 and the minimum terminal nodes being 16. The corresponding misclassification error is 0.1092. When we use this random forest classifier on the test data, the misclassification error is 0.1294.

We compare the performance of random forest algorithm with the Adaboost SVM Gaussian Kernel method. In general, both algorithms perform very well. For station Bolyston / Mass Ave, Adaboost SVM Gaussian kernel is better than random forest by 2 percent. For Tremont St / West St, random forest performs better by 2 percent. Thus, in practice, we suggest readers use both methods to address classification problems.

Implementation notes We use the build-in package random forest in scikit-learn to train the classifier. The best parameters are found through grid search.

5. Graphical Model

We build a Bayesian network to do inference on station states. We include $61 \cdot 3 = 183$ variables where the first 61 variables represent station state (empty or not empty, full or not full, following the definition in Section 4) at time $t-1$, the intermediate 61 variables represent station state at time t, and the final 61 variables represent state at time $t+1$. We denote $n_{s,t}$ as the state variable of station s at time t. We collect 850 data points, where the first 70% are used in building and estimating our Bayesian network, and the remaining 30% are used for testing purpose.

Chow-and-Liu is used in learning the best tree structure. In the complete graph with 183 nodes, we first calculate edge weight $w_{n_{s_1,t_1}-n_{s_2,t_2}}$ using training data based on the following formula:

$$w_{n_{s_1,t_1}-n_{s_2,t_2}} = \sum_{j=0,1}\sum_{k=0,1} \hat{P}(n_{s_1,t_1}=j, n_{s_2,t_2}=k)\log\frac{\hat{P}(n_{s_1,t_1}=j, n_{s_2,t_2}=k)}{\hat{P}(n_{s_1,t_1}=j)\cdot\hat{P}(n_{s_2,t_2}=k)},$$

where $\hat{P}(n_{s_1,t_1}=k) = \frac{N_{s_1,t_1,k}}{N}$, $\hat{P}(n_{s_1,t_1}=k, n_{s_2,t_2}=j) = \frac{N_{s_1,t_1,k,s_2,t_2,j}}{N}$, which are estimated from data.

We then calculate the minimum spanning tree of this complete graph using the negative value of the weights. Arbitrarily picking one station state node in time $t-1$ as a root node, we construct a direct acyclic graph (DAG) based on this spanning to conduct inference.

The conditional probability table (CPT) is also estimated from training data, for a given node n_{s_1,t_1}, $\hat{P}(n_{s_1,t_1}=j \mid \text{parents}(n_{s_1,t_1})=k) = \hat{P}(n_{s_1,t_1}=j, \text{parents}(n_{s_1,t_1})=k)/\hat{P}(\text{parents}(n_{s_1,t_1})=k)$.

For inference, we use station states in time $t-1$ and t to infer $t+1$. We use a 0-1 loss function, thus for a particular state node $n_{s,t+1}$ at time $t+1$, we perform the following prediction rule, we predict 1 if

$$\hat{P}(n_{s,t+1}=1 \mid n_{1,t-1},\cdots,n_{61,t-1},n_{1,t},\cdots,n_{61,t}) > \hat{P}(n_{s,t+1}=0 \mid n_{1,t-1},\cdots,n_{61,t-1},n_{1,t},\cdots,n_{61,t}),$$

0, otherwise. We use the testing data sets to evaluate our prediction performance. The classification results are incredible. For empty/non-empty classification, all but one stations enjoy 100% accuracy, the one left has accuracy 252/255, i.e., only mis-classifies three instances. For full/non-full classification, all but one stations enjoy 100% accuracy, the one left has accuracy 254/255, i.e., only mis-classifies one instance.

Implementation notes Chow-and-Liu is implemented in Python with NetworkX package. Bayesian network inference is implemented using bnet toolbox in Matlab.

6. Concluding Remarks and Future Directions

In this work, we use public available data to predict station state of bike sharing systems. We first develop several regression models to predict bike inventory at each station. We then design several binary classification methods to predict if an "arriving" station will be full in the next fifteen minutes, and to predict if a "departuring" station will be empty in the next fifteen minutes. Our designed Adaboost algorithm and random forest yield desirable performances. Finally, we construct and estimate a Bayesian network to infer station state only using historical station inventory data, which gives us the best prediction performance.

We want to comment on two key points regarding the performance of our methods. 1)Regarding data: As we mentioned before, although the resolution of Hubway snapshot data is 15 minutes per data point, the weather is collected only one per hour. Thus, if a rain can start and end in one hour, it may not be known in our dataset. Therefore, some regression/classification error may due to this inaccuracy. Also, this work does not take account of the existing reshuffling effects of Hubway due to the limitation in dataset. Our results would be more reliable if station-based reshuffling data are provided.

2) Regarding regression/classification method: In this project, when we do regression and classification we consider bike from all the stations for predictions and let the regression/classification algorithm to determine which features are the most important ones. Yet, it is possible to consider only the snapshots of neighbouring stations as features to predict the demand of a station. Then how to define the neighbouring stations is critical. One way is to look at the trip log data provided by the Hubway and calculate the number of trips from one station to the other. Therefore, the neighbouring stations of a station could be the stations where a high number of trips take place between them. We do not look into this dataset in this project, but it could be interesting for our feature work.

References

1. Regue, Robert, and Will Recker. Using Gradient Boosting Machines to Predict Bikesharing Station States. Transportation Research Board 93rd Annual Meeting. No. 14-1261. 2014.

2. National Climatic Data Center. Quality Controlled Local Climatological Data (QCLCD) — National Climatic Data Center (NCDC).