UiO : Department of Informatics
University of Oslo

Predicting bike-share usage patterns with machine learning

Arnab Kumar Datta

Master's Thesis Autumn 2014

Predicting bike-share usage patterns with machine learning

Arnab Kumar Datta

31st October 2014

Abstract

This thesis looks at how machine learning algorithms might be used to predict bike-share traffic. We determine the accuracies of estimators such as decision trees, random forests and boosted decision trees. The effect of factors such as weather, geographic location, time of day, day of week etc on the number of bikes at a bike-share station are also investigated. Finally, we outline how a web-based prediction system that uses the estimators mentioned in this thesis could look like.

Contents

List of Figures

List of Tables

Preface

I would firstly thank my thesis supervisor Volker Stolz, for valuable technical insights and writing advice. In addition, I also thank my co-supervisors Olaf Owe and Cristian Priscariu from the PMA group for their advice and words of encouragement.

I would also like to thank Mathias Holte, Sigmund Hansen, Seline Tomt, and Kristoffer Waløen for providing valuable feedback on my thesis.

Most of all, I thank my parents for providing moral support through a daunting writing process and for numerous proof-reading sessions.

Part I

Introduction

Chapter 1

Introduction

1.1 History of bike-share systems

Public bike-share systems were first conceptualized in 1965 in Amsterdam under the Witte Fietsen (translated to "White bikes") initiative [1, p.19]. The entrepreneur behind the plan, Luud Schimmelpennink came up with the idea of leaving 2000 free white bikes in Amsterdam that would be free for everyone to use. Users of the system could pick up any bike, ride it to their destination and leave it there for the next user. There were no locks or bike stations in this system. This resulted in an unreliable system, as there was no way to predict where users could find free bikes. The program was also compromised by theft and vandalism, as there was no user accountability.

Nearly 26 years later, the first large-scale 2nd generation of bike-share systems was introduced in Copenhagen, Denmark in 1991 [1, p.20]. In this system, bikes were designed to be picked up and returned at specific locations which resulted in a more reliable system. In addition, a coin deposit system akin to the ones found in supermarket trolleys was put into place. However, the system still suffered from theft due to the lack of user accountability.

1.2 Bike-share systems today

Modern 3rd generation bike-share systems require customers to authenticate themselves through identification in order to increase user accountability. Some bike-share systems today now require users to pay with a credit card so that the user is charged the price of the bike in case of theft. In addition, most bike-share systems use proprietary parts in their bikes to discourage disassembly and resale of parts.

Bike-sharing today has gone through technological improvements such as real-time status maps of bike-share stations (see figure 1.1), smartcards and electronic-locking racks and on-board communication systems for location tracking [1, p.20].

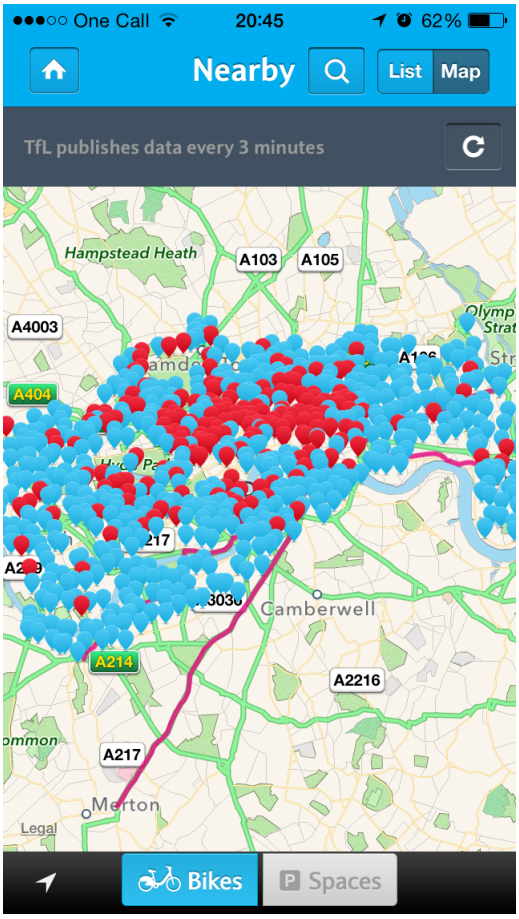


Figure 1.1: London bike-share system in real-time (Barclays Bikes iphone app). Each of the bike-share stations are shown using a pin icon. The pins are colored blue if the station is full, and red if the station is empty.

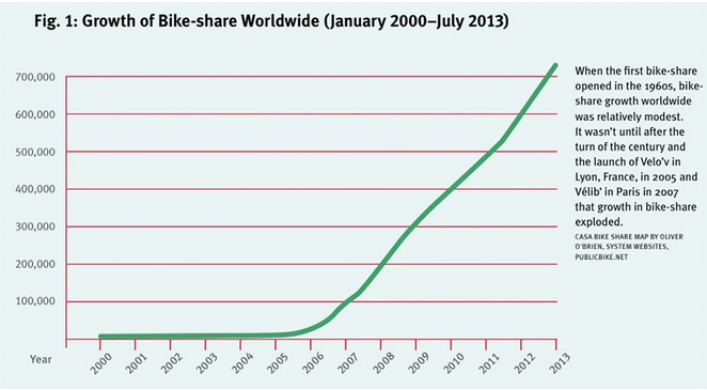


Figure 1.2: Bike-share growth worldwide (picture courtesy of the ITDP bike-share planning guide [1])

Bike-share systems have experienced significant growth worldwide [1, p.13], [2, p.5] and are quickly gaining popularity as a green and healthy way to travel. This growth has created challenges that

are presented in the next section.

The following bike-share systems were studied in this thesis:

- Barclays Cycle Hire (London)
- Capital bikeshare (Washington D.C.)

1.3 Challenges of modern bike-share systems

1.3.1 Overflows in the bike-share system

Let us define the term *overflow* as a situation where a bike-share station is in danger of being too full (i.e. customers can't park their bikes there). This leads to customers being forced to use another bike-share station or park the bike privately overnight (see figure 1.3).

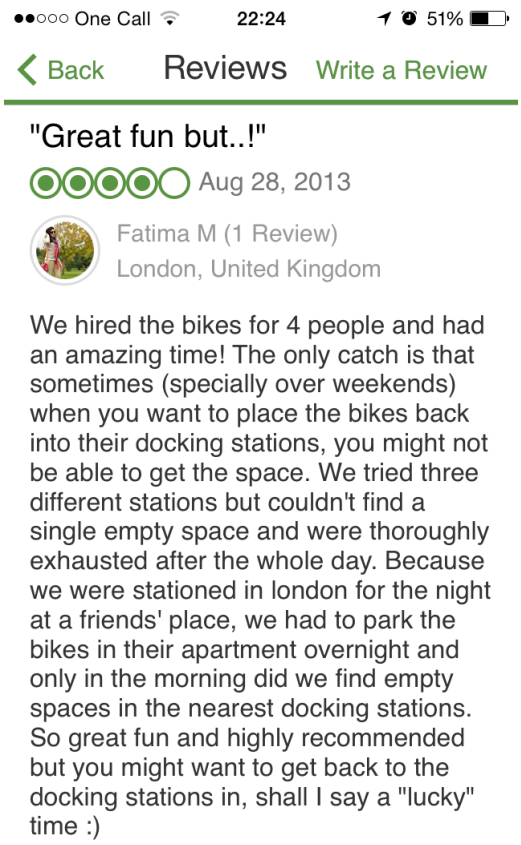


Figure 1.3: A customer reviews London's bike-share system on the tripadvisor website [3]

It would therefore be useful for customers to have a prediction system that tells them if a bike-share station will be full when they arrive.

In addition, when commuters have a wide range of choices in regards to where they park their bike, it would be helpful to know which station would be the least likely to be full.

1.3.2 Imbalances due to commute patterns

Commute patterns will place imbalances in bike-share systems. In addition to the overflows mentioned above, let us define *shortage* as a condition where a bike-share station is in danger of running out of available bikes. Shortages and overflows occur as part of the daily commute pattern, as shown in figures 1.4 and 1.5.

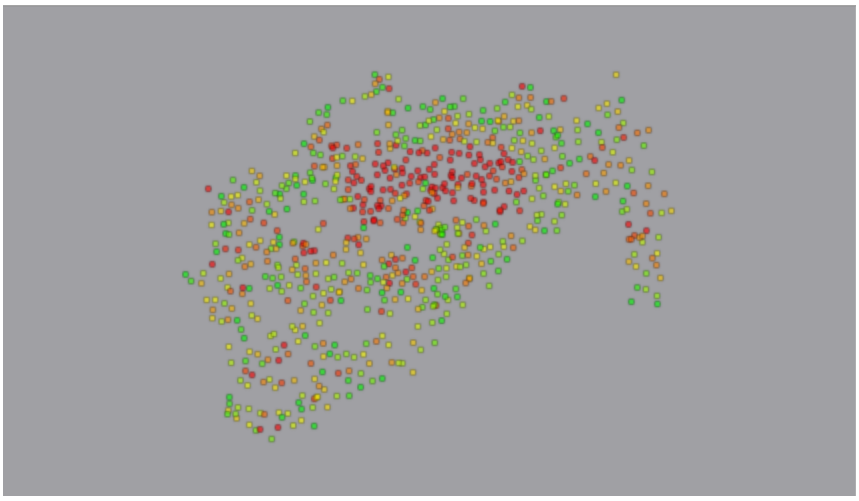


Figure 1.4: London bike-share status before the morning rush (approx 7:20 AM) on a Tuesday. Image generated by the visualization module mentioned in 3.2).

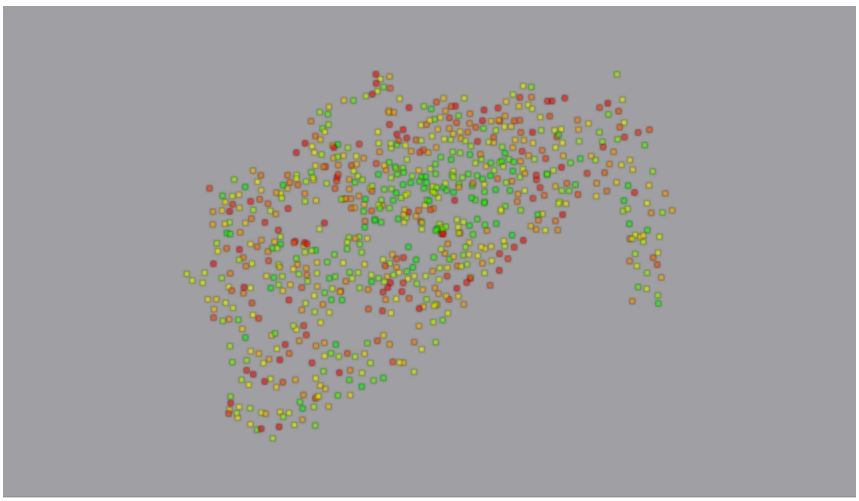


Figure 1.5: London bike-share status after the morning rush (approx 10:35 AM) on a Tuesday. Image generated by the visualization module mentioned in 3.2).

The images in figures 1.4 and 1.5 are generated from traffic data recorded in the London bike-share system run by Barclays. Each bike-share station is represented by a circle, and the color of the station represents how full the station is: the redder the station, the more empty it is and the greener the station, the closer it is to being completely full.

When shortages arise, it is important that redistribution trucks restore balance to the bike-share distribution. It is therefore just as important to understand when bike-share stations run empty, and this is in fact just the inverse problem (predicting overflows and the likelyhood of overflows).

1.4 Mathematical description of the problem

Let us consider bss to be the status of a bike-share station, $0 <= X <= Y <= 100$ (X and Y being arbitrary classification boundaries), where the possible values of bss (for classification) might be:

- shortage (0 - X% free bikes in the bike-share station)
- balanced (X - Y% free bikes in the bike-share station)
- overflow (Y - 100% free bikes in the bike-share station)

For regression purposes, bss is defined as the number of bikes available at the given bike-share station.

Let w be the weather recorded at time t, where air temperature T is measured in °C, cloud cover CC is measured in okta and precipitation (last 24h) RR is measured in mm. Let st be the station described by the latitude lat, longitude lng, altitude alt, and the station_id id. And finally, let t be the time described by the variables: time of the day h, day of the week $wday$, day of the month d, month m.

The concepts are formalized below:

$$w = (T, CC, RR) \tag{1.1}$$

$$t = (h, wday, d, m) \tag{1.2}$$

$$st = (lat, lng, alt, id) \tag{1.3}$$

The contribution of this thesis is to answer the following questions:

1. Given a station st, the weather conditions w and the time t, can we predict the bike-share station status bss using a machine learning algorithm?
2. What are the best performing estimators for this particular task?

3. How do factors such as time of day, weather etc affect bike-share traffic?

The software is designed to predict future observations based on past data. Example: Assume that the current time is 8:00 AM on July 15th 2014. The estimator of choice is trained on data from July 1st-14th, and asked to predict the status of the bike-share station at Cambridge St (Washington DC) at 9:00 AM July 15th 2014. Classifier-type estimators will be able to predict whether the station can be expected to overflow, experience a shortage or be balanced, while regressors will just output the expected number of bikes at the given station.

1.5 Research goals

The high-level goals of this thesis, is as follows:

- Reduce frustration amongst customers by letting them know beforehand that a station is going to be empty or full. Accurate predictions will allow them to choose an alternative path.
- Enable the bike-share operators to be proactive. If they can receive a prediction about the net loss in number of bikes X at a given station in a timeframe T, they can then place $X + b$ number of bikes at that station, where b is a small buffer that ensures that the station will not be empty. Similarly, if they are able to predict an influx, they can do the opposite.

In order to achieve these goals, the system must be able to predict the future based on historical data. It must also be flexible enough to take into account factors that cause variance like wind, rain, temperature etc, while generalizing enough to understand concepts like weekdays and weekends, morning and afternoon rush hours etc. that produce periodic patterns in the data.

Chapter 2

Related work

2.1 Data Science for Social Good: Divvy Bikes

The Eric and Wendy Schmidt Data Science for Social Good (DSSG) fellowship is a summer program at the University of Chicago that invites data scientists from all over the USA to use data mining, machine learning and big data analysis techniques for solving projects that have social impact.

The public bike-share system in Chicago is run by a company called Divvy. The divvy system, like most other bike-share systems share a weakness that occurs due to commute patterns. There is an influx of bikes in the city center during the morning, and in the suburbs during the afternoon. This leads to imbalance in the system. In Divvy, the rebalancing of bikes was done with trucks that drove around with bikes. However, the problem was that this was done on a reactionary basis rather than on a prophylactic basis [4]. Therefore, although the system would eventually be rebalanced, there was a scope of improvement if the operators could predict overflows and shortages.

In July 2013, one of the projects was to predict when Divvy bike share stations would be empty or full. This would help the Divvy operators to see an estimate of how the system would look in the future.

2.1.1 Data and Analysis

The data collected as part of the preliminary analysis included both bike-share data and weather data. The sources were:

- historical bike-share data from the O'Brian bikeshare datacollection project [5]
- Weather data from forecast.io

The DSSG team looked at the Boston bike share system, and visualized daily usage patterns. Each station was investiaged individually, and the team looked at the number of bikes that

were available at every minute of the day (00 : 01, 00 : 02, 00 : 03 etc). This number was collected for the whole span of a large dataset (spanning a year), and then averaged. Figures 2.1 and 2.2 show the commute pattern from two bike-share stations: one downtown and another suburban.

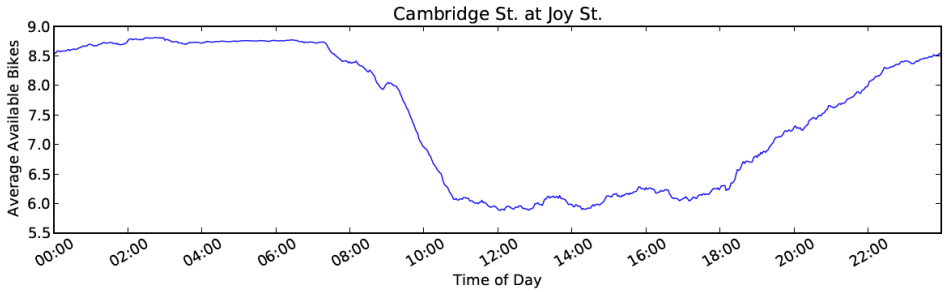


Figure 2.1: Traffic data for Cambridge St. Image courtesy of the DSSG team [6]

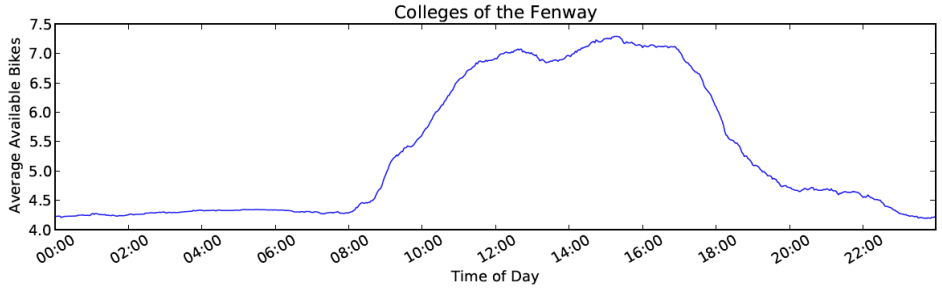


Figure 2.2: Traffic data for the Colleges of the Fenway station. Image courtesy of the DSSG team [6]

The commute pattern from figure 2.1 and 2.2 is very similar to the one found later in figure 4.7 and it shows that stations in downtown and suburban areas seem to complement each other for most of the day.

They also found that the standard deviation of the number of bikes available at each station during the day was high. This meant that they could not simply average the recordings for any given time during the day and use that to predict the number of bikes. In order to explain the high variance in the data, they took into account factors like:

- Is the observation recorded on a weekday or weekend?
- Was there rain in close proximity of the observation?
- Is there a big event happening close to the observation that might explain a huge influx of riders?

2.1.2 Data models

The two models that were most successful in predicting the data were:

- Autoregressive binomial logistic model
- Poisson point process model

Note that the models were fit to each individual station in the system, rather than the system as a whole. This is different from the approach chosen in this thesis, as the attempt of this thesis is to create a more general prediction system that would also be capable of predicting how a new bike-share station (one without any historical data) would behave.

2.1.2.1 Data features

The feature values chosen for the data were:

- current number of bikes at a station S
- available spaces at a station S
- the hour of the day
- the current temperature
- the current precipitation

Additionally, the logistic model stored the number of bikes at a station S fifteen and thirty minutes ago as part of an autoregressive structure [7].

The DSSG team also had access to rebalancing data i.e. recorded times for when the bike share trucks added or removed bikes at a station. This helped them distinguish changes caused by riders and the rebalancing team. The reason this was important is that the point of both models was to predict what would happen to a station if it were to be left untouched by the rebalancing team.

The rebalancing data was then adjusted for, by manipulating the number of bikes recorded in historical data. Say for instance station S received five bikes from a truck at 08:00 on Tuesday 7th July 2013, and this increased the number of bikes from 10 to 15. The adjustment would be to instead assume that the rebalancing did not take place and simulate what would happen from there on.

2.1.2.2 Poisson Point Process

A Poisson process is a stochastic process that counts the number of times a type of event E occurs in a given time interval T. The occurrences are considered independent of each other. In this case, there are two types of events: arrivals $E_{arrival}$ and departures

$E_{departure}$. The assumption is that these are also independent, and therefore the rates for arrivals and departures are modelled separately from each other.

The number of bikes arriving or departing are calculated using a maximum likelyhood estimator [8] that determines how weather elements, the time of the day and day of the week will play a role and outputs a coefficient that is then applied to the poisson point process to determine when the next event will be. This provides the model with the time of the next event, but not what type of event it will be. This is decided by a coinflip that takes into account the likelyhood of departures happening at the bike-share station at that particular point in time.

The result is then simulated by updating the number of bikes at the station, and calculating the time until the next arrival or departure event.

Note: In case the poisson process predicts an arrival of a bike at a station that is already full, the model treats it as if the rider parked at an alternative location. The same is true if the model predicts a departure from an already empty station. This is merely to avoid simulation results that would be invalid i.e. a station with more bikes than it has the capacity to hold, or a station with a negative number of bikes.

2.1.2.3 Autoregressive Binomial Logistic Regression

Binomial Logistic Regression models provide a prediction on how many bikes a station S will have at the end of a time interval T. This is done by taking the probability that a bike will be present at any given dock $P(D_{bike})$, $D \subset S$, and multiplying it by the number of parking spaces in S. In other words: $E(S) = \sum P(D_{bike}), D \subset S$

In addition, the logarithm of the odds of a dock being full currently $log(P_{currently\_full})$, fifteen minutes ago $log(P_{full\_15\_mins\_ago})$ and thirty minutes ago $log(P_{full\_30\_mins\_ago})$ are incorporated into the estimator along with temperature and precipitation values at the current time. The estimator will then adjust its probability output for each dock holding a bike, and the expected number of bikes can then be found using the equation given above.

2.1.3 Results

The Poisson model predicts a certain chain of events in a given timeframe T when the simulation is started with a certain number of bikes n. This model is run thousands of times, and the results accumulated as a distribution as shown in figure 2.3. Note that starting the simulation with fewer bikes increases the likelyhood that the station will be empty in two hours (as expected).

The simulations shown in figure 2.3 can be used to model the probabilities that a station will be empty or full at any point during

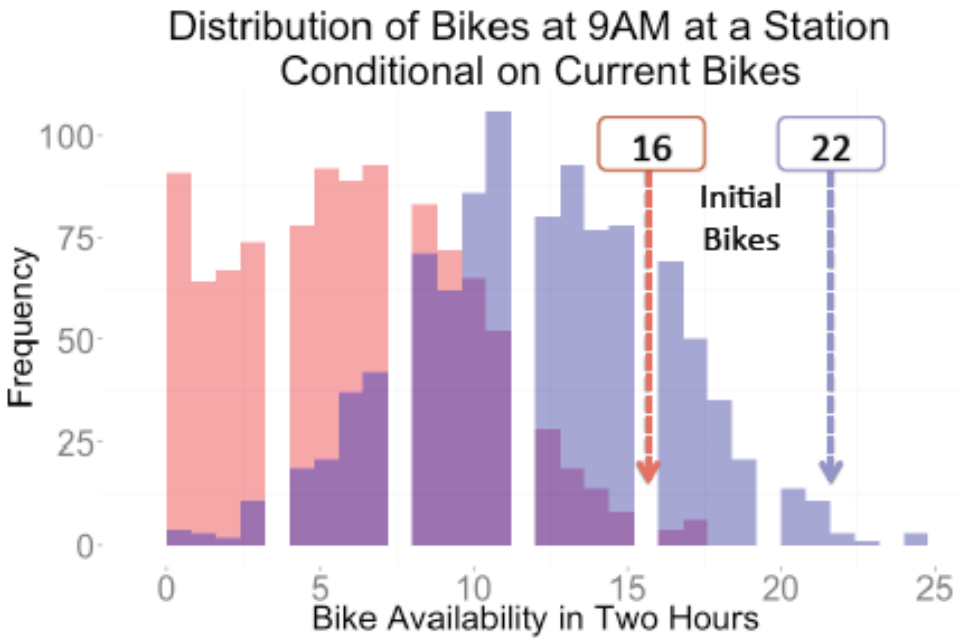


Figure 2.3: The pink distribution is when the simulation is started with $n = 16$, and blue is for $n = 22$. Image courtesy of the DSSG team [6]

a time interval given a starting point with a certain number of bikes. In other words, the results can be used to compute $P(k = 0)$ (empty station) and $P(k = N)$ (full station) for any station at any given point in time.

Due to the distributions being dependant on starting conditions, it is possible to predict how many bikes the station must start with to be completely empty by the end of the time interval. Figure 2.4 demonstrates this, and also shows how much of a tolerance this prediction has. It is for instance possible to be 95% sure that if the simulation starts with 19 bikes, it will not be empty during the time interval 7 AM - 9 AM.

The models and errors were evaluated using MSE (Mean Squared Error) which is defined as follows: The mean $\overline{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$ $MSE(\overline{X}) = E((\overline{X} - \mu)^2)$

The evaluation method was to train the model on one year of data, and then try to predict incrementally larger time intervals. The largest interval trained was one week. The RMSE (Root Mean Squared Error) in terms of number of bikes increased (as expected) as the size of the test set was expanded (see figure 2.5). Note: the reason for using RMSE instead of MSE in this plot is simple; it has the same units as the estimator's predicted number of bikes and can therefore be plotted in the same graph.

Lastly, the DSSG team created 90 test sets that the models predicted and visualized. The visualizations were histograms consisting of 90 error terms that corresponded to each of the test sets. The predictions were then visualized as in figures 2.6, 2.7 and 2.8. It

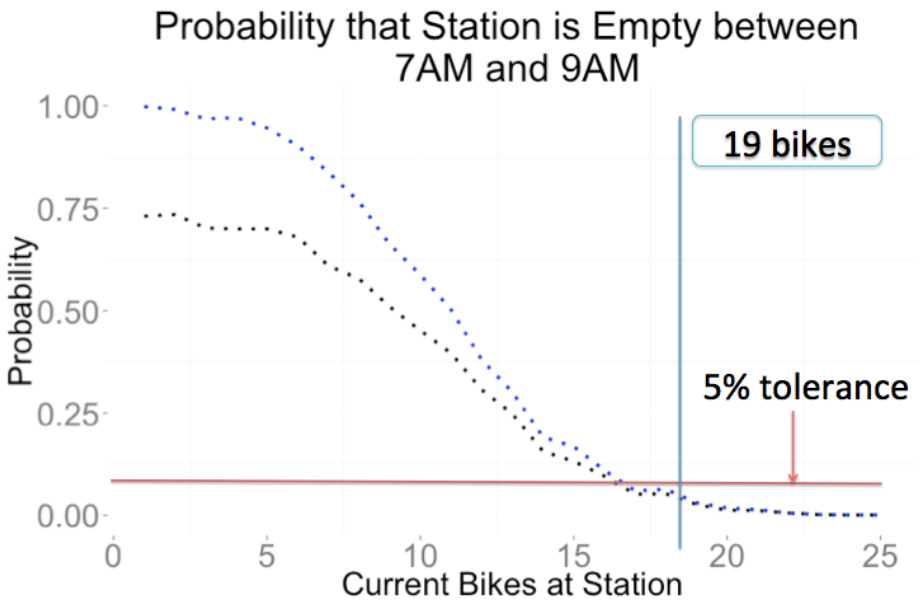


Figure 2.4: Sample probability graph that a station will be empty between 7 AM and 9 AM on a given day. The dotted blue line represents the likelyhood of this occuring anytime between 7 and 9 AM. The dotted black line is for estimating the likelyhood that the station will be empty exactly at 9 AM. Image courtesy of the DSSG team [6]

can be seen that the uncertainity of the predictions increases as the prediction time interval gets larger.

Despite the fall in accuracy when predicting an hour ahead, the predictions are still pretty accurate (77% of predictions are correct within an RMSE of 5 bikes). The DSSG mentions on their websites that they would like to explore whether higher accuracy can be achieved by using ensemble techniques such as bagging [9] or boosting [10]. Boosting and bagging methods are covered deeper in chapter 6 (ensemble methods).

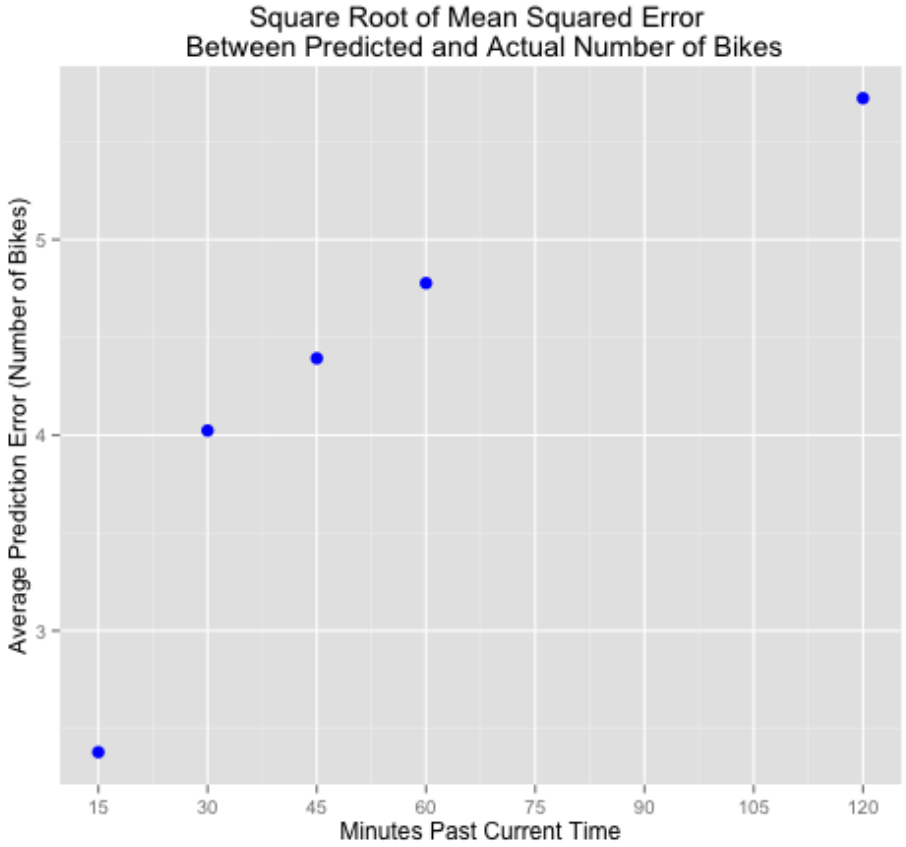


Figure 2.5: RMSE in number of bikes predicted vs actual number of bikes. Image courtesy of the DSSG team [6]

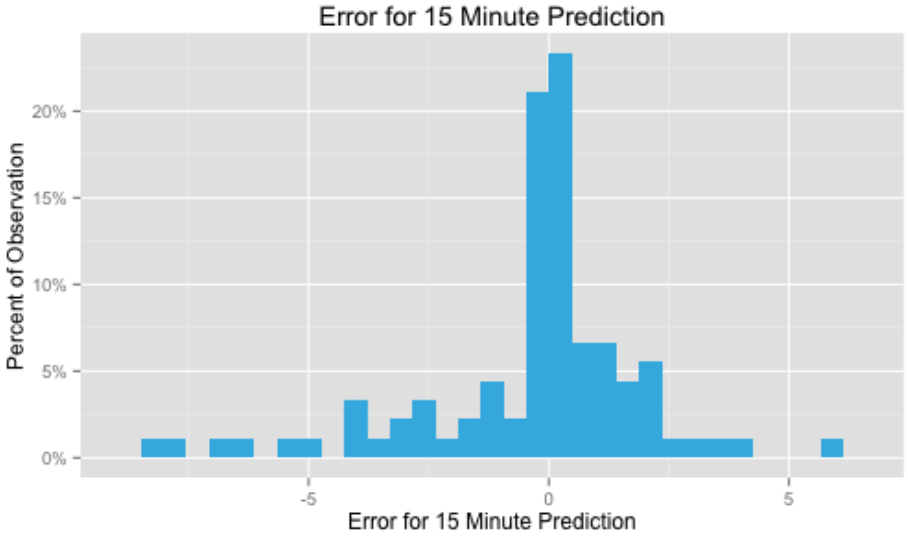


Figure 2.6: Error histogram for 15 minutes forward in time. Image courtesy of the DSSG team [6]

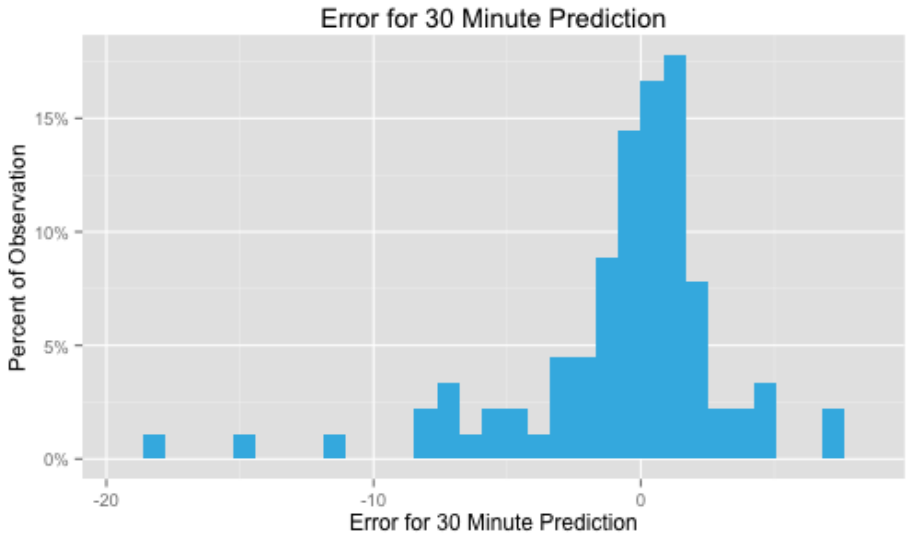


Figure 2.7: Error histogram for 30 minutes forward in time. Image courtesy of the DSSG team [6]

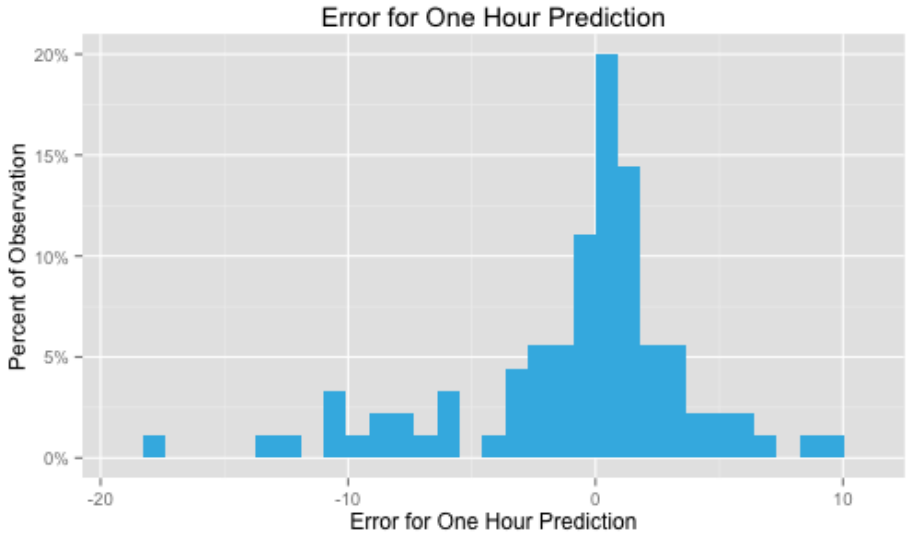


Figure 2.8: Error histogram for one hour forward in time. Image courtesy of the DSSG team [6]

2.2 VanderPlas: "Is Seattle really seeing an uptick in cycling?"

In October 2012, the city of Seattle (Washington, USA) installed a bike counter device, known as a "eco-totem" on the Fremont bridge (see fig 2.9). The device was funded by the Mark and Susan Torrance Foundation, and then acquired by the Cascade Bicycle club which gave it to the city of Seattle. The Seattle Department of Transport (SDOT) publishes the data collected from the eco-totem on their website [11] through an API. The intent behind the installation was not only to act as a motivator to encourage people to use bicycles as a means of commuting but also to aid analysis of bicycle traffic.

Figure 2.9: Bike counter on the Fremont Bridge, Seattle

The data from the eco-totem was analyzed by Jake VanderPlas, who works as a data scientist at University of Washington eScience Institute [12]. He investigated [13] the validity of claims from Seattle Bike-Blog that bicycle usage was on the rise [14]. In the introduction to the article he states his research goals for the data collected:

> Bicycle advocates have been pointing out the upward trend of the counter, and I must admit I've been excited as anyone else to see this surge in popularity of cycling (Most days, I bicycle 22 miles round trip, crossing both the Spokane St. and Fremont bridge each way).
>
> But anyone who looks closely at the data must admit: there is a large weekly and monthly swing in the bicycle counts, and people seem most willing to ride on dry, sunny summer days. Given the warm streak we've had in Seattle this spring, I wondered: are we really seeing an increase in cycling, or can it just be attributed to good weather?

> Here I've set-out to try and answer this question. Along the way, we'll try to deduce just how much the weather conditions affect Seattleites' transportation choices.

2.2.1 The data

Hourly bicycle counts were downloaded from the SDOT website [11] as CSV files. In addition, weather data was acquired from the National Climatic Data Center website [15]. Both the hourly bicycle data and weather data were parsed using Pandas [16], a free open-source library for data analysis in python.

The fremont bridge records the following every hour:

- time in the format DD:MM:YYYY HH:MM:SS
- number of northbound bicycles
- number of southbound bicycles

2.2.2 Overviews of the data

First VanderPlas creates a brief overview of the data, shown in figure 2.10. Following this, the data is modelled using a Linear Regressor and de-trended for the following factors:

- Hours of daylight per day
- Day of Week
- Temperature
- Precipitation

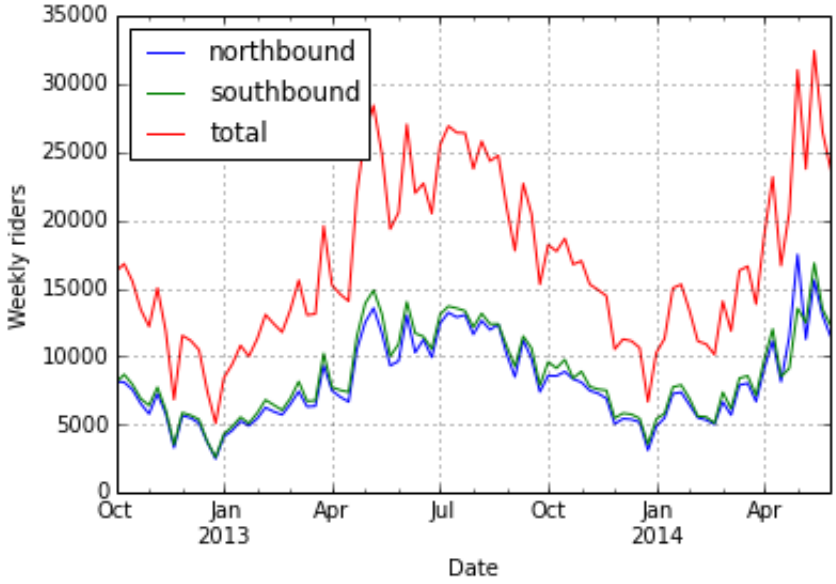


Figure 2.10: Weekly traffic data for the Fremont bridge. The green and blue lines signify soutbound and northbound traffic respectively, while the red line is the total traffic on the bridge. Image courtesy of Jake VanderPlas [13].

2.2.3 De-trending the data

The de-trending is done in an identical fashion for all the factors mentioned above and therefore only one of them, hours of daylight per day will be described below as an example. VanderPlas says the following about the de-trended data we see in our example:

> This is what I mean by "de-trended" data. We've basically removed the component of the data which correlates with the number of hours in a day, so that what is left is in some way agnostic to this quantity. The "adjusted weekly count" plotted here can be thought of as the number of cyclists we'd expect to see if the hours of daylight were not a factor.

First, the average hours of daylight d in Seattle is computed for every month in the year (figure 2.11). Then, the weekly bicycle traffic is plotted against the average daylight hours (figure 2.12).

This data is then fitted to a Linear Regressor [17], and the results visualized (figure 2.13) using the snippet shown in Listing 2.1. Thereafter, a quick look at the model coefficients reveal how much one hour of daylight affects the number of weekly crossings on the bridge: 2000 riders per extra hour of daylight in this case (see Listing 2.2).

The data is adjusted by subtracting off the trend that follows as a natural consequence of the extra daylight hours and replacing that chunk of the y-component with the mean instead (see Listing 2.3). A visualization of the adjusted data is shown in figure 2.14. After the data has been de-trended for all the factors mentioned above, the error covariance for each of the factors is calculated, and used to compute the error bars for each of them (see Listing 2.4).

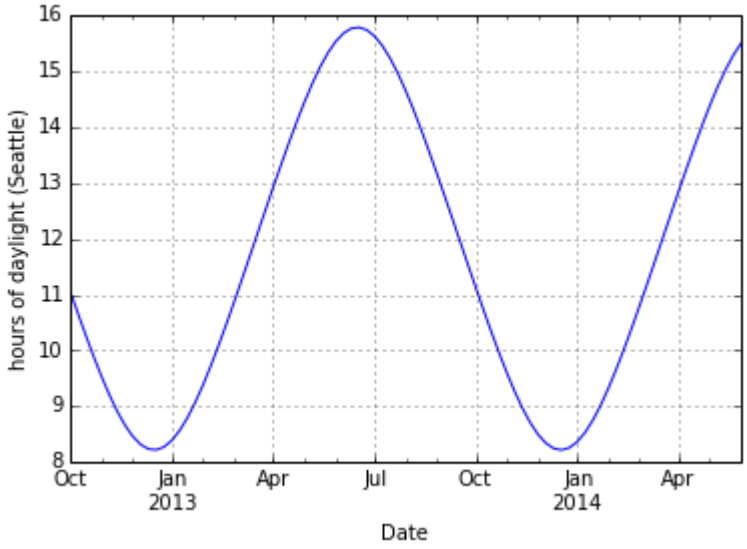


Figure 2.11: Hours of daylight (Seattle) given as a function of the time of year.

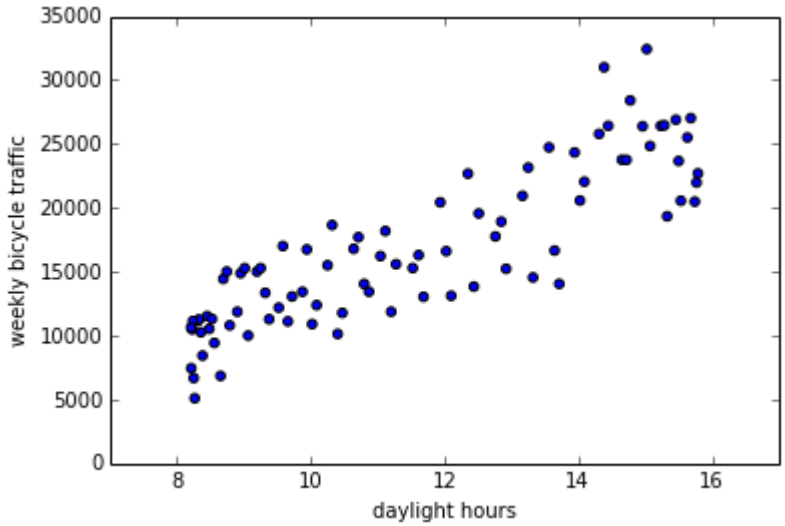


Figure 2.12: Weekly bicycle traffic given as a function of the hours of daylight (Seattle). Image courtesy of Jake VanderPlas [13].

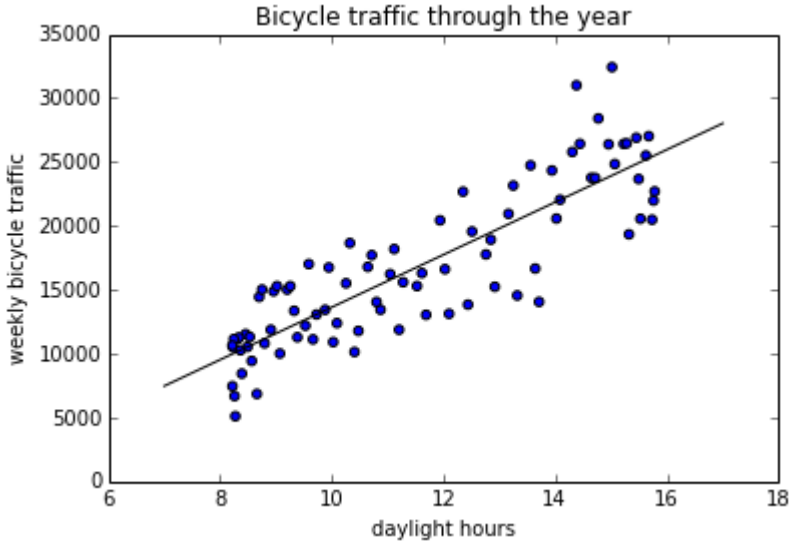


Figure 2.13: Fitting a linear regressor to the data. Image courtesy of Jake VanderPlas [13].

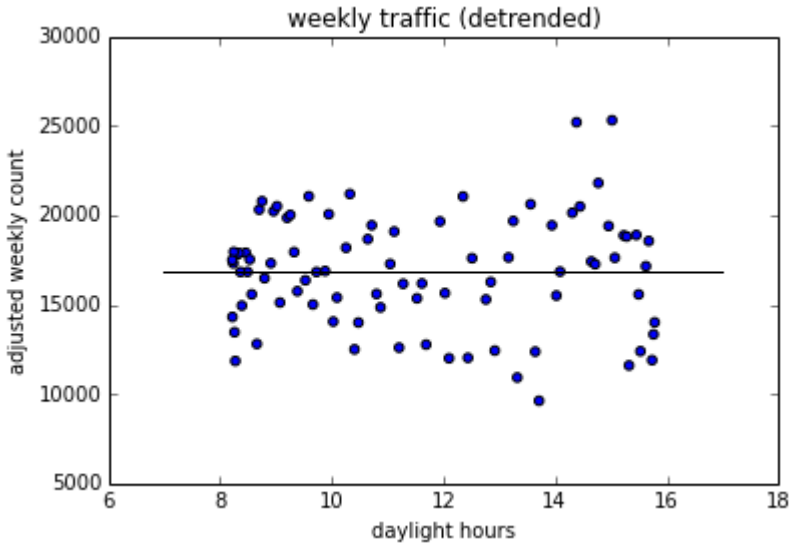


Figure 2.14: Weekly traffic data de-trended for hours of daylight. Image courtesy of Jake VanderPlas [13].

Listing 2.1: Fitting a Linear Regressor to bike crossing data

```
from sklearn.linear_model import LinearRegression

X = weekly[['daylight']].to_dense()
y = weekly['total']
clf = LinearRegression(fit_intercept=True).fit(X, y)

weekly['daylight_trend'] = clf.predict(X)
weekly['daylight_corrected_total']
 = weekly['total'] - weekly['daylight_trend']
 + weekly['daylight_trend'].mean()

xfit = np.linspace(7, 17)
yfit = clf.predict(xfit[:, None])
plt.scatter(weekly['daylight'], weekly['total'])
plt.plot(xfit, yfit, '-k')
plt.title("Bicycle␣traffic␣through␣the␣year")
plt.xlabel('daylight␣hours')
plt.ylabel('weekly␣bicycle␣traffic');
```

Listing 2.2: "Calculating the coefficient for daylight hours; the coefficient is the increase in the number of crossings for every extra hour of daylight"

```
In [10]:print(clf.coef_[0])
Out [10]:2056.44964989
```

Listing 2.3: Replacing the trend with the mean

```
trend = clf.predict(weekly[['daylight']].as_matrix())
plt.scatter(weekly['daylight'], weekly['total']
  - trend + np.mean(trend))
plt.plot(xfit, np.mean(trend) + 0 * yfit, '-k')
plt.title("weekly␣traffic␣(detrended)")
plt.xlabel('daylight␣hours')
plt.ylabel('adjusted␣weekly␣count');
```

Listing 2.4: "Computing error covariance"

```
#Calculating error bars
#X is an array of feature vectors for the observations
#y is the de-trended target variable for X
vy = np.sum((y - daily['final_trend']) ** 2) / len(y)
X2 = np.hstack([X, np.ones((X.shape[0], 1))])
C = vy * np.linalg.inv(np.dot(X2.T, X2))
var = C.diagonal()
```

2.2.4 Results

The results of the analysis yield some interesting statistics:

- Rain: “Every inch of rain translates, on average, to about 800 cyclists staying home.”
- Day of week: “As you might expect in a city of bicycle commuters, there is roughly 2.5 times the amount of traffic on weekdays as there is on weekends. Bicycles are not just for entertainment! In Seattle, at least, they are a real means of commuting for thousands of people per day, and the data show this clearly.”
- Temperature (°F): “We see that for every increase of ten degrees, we add around 250 crossings on the Fremont bridge!”
- Daylight: “We see that, once the effects of rain and temperature are removed, each hour of daylight results in about 125 more crossings at the Fremont Bridge. This is fewer than the 2000/week (300/day) that we saw above: this is because our first model did not include precipitation and temperature: apparently the weather is far more important than the darkness in affecting ridership!”

Finally, VanderPlas looks at the question "Is ridership increasing?" when the data is de-trended for annual and daily trends, weather trends etc. The analysis suggests that there are 4.4 ± 0.8 new riders per day, which translates to a 10% growth in bicycle traffic from 2012.

In this thesis, weather factors and their effects on bike-share traffic were also investigated, albeit in a different fashion; a feature importance table (table 7.1). It is worth noting that the concept of a feature being encapsulated within another is encountered in this thesis as well, where monthly trends encapsulate weather phenomena such as temperature, precipitation etc. VanderPlas also admits that Linear Regression is not a complex enough model to capture the complexities of bike patterns (as evidenced by an RMSE of 500 bikes every week in the final model), and mentions that ideally he would like to fit RandomForest estimators to his data.

Chapter 3

Software

Figure 3.3 on page 32 shows a brief overview of the software modules that were developed as part of this thesis. Note that both the PyBikes API and Weather Underground website were developed externally.

3.1 Data collection

3.1.1 PyBikes

PyBikes is a software module written by Lluis Eskerda. The API can be found at https://github.com/eskerda/PyBikes . The API provides a common interface to different bike-share systems around the world. It is intended mainly for data analysis projects.

3.1.1.1 Usage

The API can be used as follows:

```
>>>import pybikes
>>>#Washington DC uses a bixi system
>>> DC_bikeshare = pybikes.getBikeShareSystem
... ('bixi', 'capital-bikeshare')
>>> print(DC_bikeshare.meta)
{
    'name': 'Capital BikeShare',
    'city': 'Washington, DC - Arlington, VA',
    'longitude': -77.0363658,
    'system': 'Bixi',
    'company': 'PBSC',
    'country': 'USA',
    'latitude': 38.8951118
}
>>>DC_bikeshare.update()
>>> print(len(capital_bikeshare.stations))
191
>>> print(capital_bikeshare.stations[0])
--- 31000 - 20th & Bell St ---
bikes: 7
free: 4
```

```
latlng: 38.8561,-77.0512
```

3.1.1.2 Data format

Bike-share data is stored as JSON (JavaScript Object Notation) files. JSON is an open-standard format with human-readable text which is used to store bike station statuses in the chosen cities.

Bike station information is stored as follows:

```
{

 "0": {
  "capacity": 10,
  "description": "31000 - 20th & Bell St",
  "id": 0,
  "latitude": 38.8561,
  "longitude": -77.0512
    },
   ...
}
```

Bike station observations are logged in the following format:

```
{
    "city": "Washington, DC",
    "time": 1396928285 ,
    "stations": [
  { "id": 0, "bikes": 5, "free": 6},
  { "id": 1, "bikes": 6, "free": 5},
  { "id": 2, "bikes": 9, "free": 6},
  ...
    ]
}
```

3.1.2 Elevations API for obtaining altitudes

The ITDP bike-share planning guide [1, p.116] states:

> For example, most systems have found that stations at the tops of hills are often empty, as people will check out a bike and ride down the hill, but will rarely ride up the hill to park at that station.

In order to investigage how much of an impact altitude has, altitude data was required. PyBikes does not collect data on bike-share station altitudes, but it does contain data on the geolocation of bike-share stations. A third-party library, Google Elevations API, was used to resolve the altitudes of bike-share stations.

An example of the Elevation API is given below

```
  //API request
  http://maps.googleapis.com/maps/api/elevation/json?
  locations=59.6,10.72&sensor=true

  //This is the json result generated by the request
  {
    "results" : [
```

```
    {
  "elevation" : 107.5778503417969,
  "location" : {
    "lat" : 59.6,
    "lng" : 10.72
  },
  "resolution" : 610.8129272460938
    }
  ],
  "status" : "OK"
}
```

3.1.3 Weather data

Additionally, in order to determine the importance of weather on bike-share stations, weather data collection was required as well. This was done using the weather API at http://www.wunderground.com/

The API gives a response in the following form:

```
Time,TemperatureC,DewpointC,PressurehPa,WindDirection
,WindDirectionDegrees,WindSpeedKMH,WindSpeedGustKMH,
Humidity,HourlyPrecipMM,Conditions,Clouds,dailyrainMM,
SoftwareType,DateUTC

2014-07-02 00:00:00,47.6,26.2,1016.8,East,
100,0.0,0.0,31,0.0,CLR,BKN, 0.0,
WeatherDisplay:10.37,2014-07-01 23:00:00,

2014-07-02 00:10:00,47.6,26.2,1016.8,East,
100,0.0,0.0,31,0.0,CLR,BKN,0.0,
WeatherDisplay:10.37,2014-07-01 23:10:00,
```

The timestamp is then converted into UNIX epoch time [18], and the weather for that timestamp is stored as follows:

```
    "weather": {
        "Clouds": "BKN",
        "Conditions": "-RA",
        "DewpointC": "8.5",
        "HourlyPrecipMM": "1.0",
        "Humidity": "85",
        "PressurehPa": "1011.1",
        "SoftwareType": "WeatherDisplay:10.37",
        "TemperatureC": "10.9",
        "Time": "2014-04-01 01:51:00",
        "WindDirection": "North",
        "WindDirectionDegrees": "0",
        "WindSpeedGustKMH": "0.0",
        "WindSpeedKMH": "0.0",
        "dailyrainMM": "1.0"
    }
```

Finally, the weather values are added to, and stored in the bike-share log files mentioned in section 3.1.1.2.

3.1.4 Data loader script

The machine learning algorithms that are used, cannot deal with JSON data. Instead they require numerical data. A data loader was developed for this, where the JSON files were converted using numpy and python.

A typical log file containing bike-share data and weather data looks like this

```
    {
    "city": "Washington, DC",
    "stations": [
        {
            "bikes": 6,
            "free": 5,
            "id": 0
        } ...
    "time": 1403388000,
    "weather": {
        "Clouds": "",
        "Conditions": "",
        "DewpointC": "15.3",
        "HourlyPrecipMM": "0.0",
        "Humidity": "65",
        "PressurehPa": "1011.4",
        "SoftwareType": "WeatherLink 5.9.2",
        "TemperatureC": "22.2",
        "Time": "2014-06-21 18:08:00",
        "WindDirection": "North",
        "WindDirectionDegrees": "0",
        "WindSpeedGustKMH": "11.3",
        "WindSpeedKMH": "6.4",
        "dailyrainMM": "-2539.7"
    }
}
```

This is then converted into the following format:

```
  #Observations are numpy arrays of type
  # ['epoch', 'time_of_day_hours',
  #'day_of_week',  #'station_id', 'latitude',
  #'longitude', 'altitude', 'TemperatureC',
  #'HourlyPrecipMM', 'number_of_bikes']

  data = array (array([1.403388000e+09,
          0.00000000e+00, 7.00000000e+00,
          0.00000000e+00, 3.88561000e+01,
          -7.70512000e+01,  1.55606689e+01,
          2.23000000e+01, 0.00000000e+00,
          6.00000000e+00])
  , ...)
```

After this, we are ready to feed the data into our chosen machine learning algorithms, and analyze accordingly. These arrays are also used in conjunction with pyplot to produce graph plots that illustrate bike-share traffic (see figure 3.1).

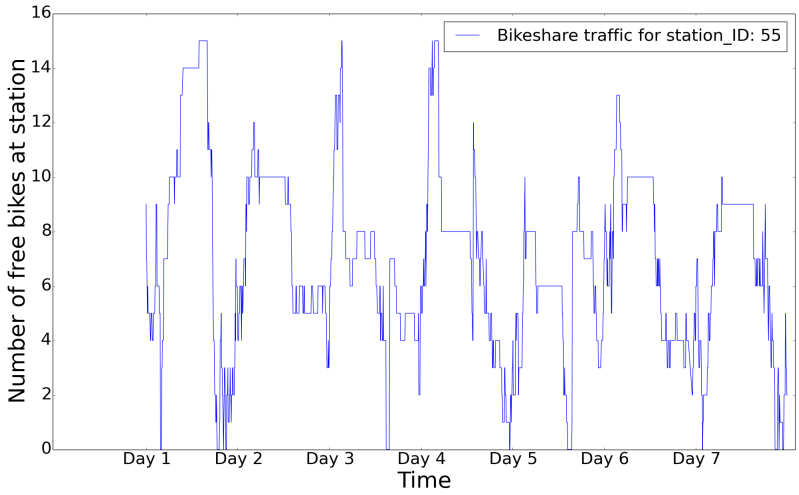


Figure 3.1: Bike-share traffic recorded from June 21st to 28th in Washington D.C. at station 55

3.2 Visualization module

When dealing with large amounts of data, it is always useful to have some means of visualizing the data quickly. In the case of bike-share systems, the visualization module included in the software enables us to view bike-share history of a month in about twelve minutes. Additionally, it makes it easier to notice behavioral differences in downtown stations compared to suburban stations (see figure 1.4 and 1.5 for instance).

The module itself is written in C++, using the Qt Framework. Qt is a framework that was developed in 1991 by Trolltech, and is now maintained by Digia. The framework focuses on creating cross-platform applications and providing easy-to-use GUI toolkits to developers.

A class diagram of the module is shown in figure 3.2.

3.3 Analysis module

The analysis module can be further broken down into three main scripts:

- Preprocessing scripts; these scripts are used to append additional feature information to stations and datapoints, in order to improve the performance of regressor and classification scripts.
  - Spectral clustering based on similarity matrices
  - Individual listing of similar stations for each station in a network

- Estimator loader scripts (classifiers and regressors)
- Genetic Algorithms for adjusting hyperparameters for the estimators

3.4 Module for emailing reports

When large datasets with machine learning algorithms, experiments could sometimes take upwards of three to four hours. It was therefore practical to develop a module that could automate the testing and email the results when the experiments were finished.

3.5 Github repository for source code

The source code is organized into three repositories:

- Visualization module: https://github.com/arnabkd/bikes-timeline-qt
- Data analysis module: https://github.com/arnabkd/bikeshare-analysis
- Data collection module: https://github.com/arnabkd/pybikes-datacollection

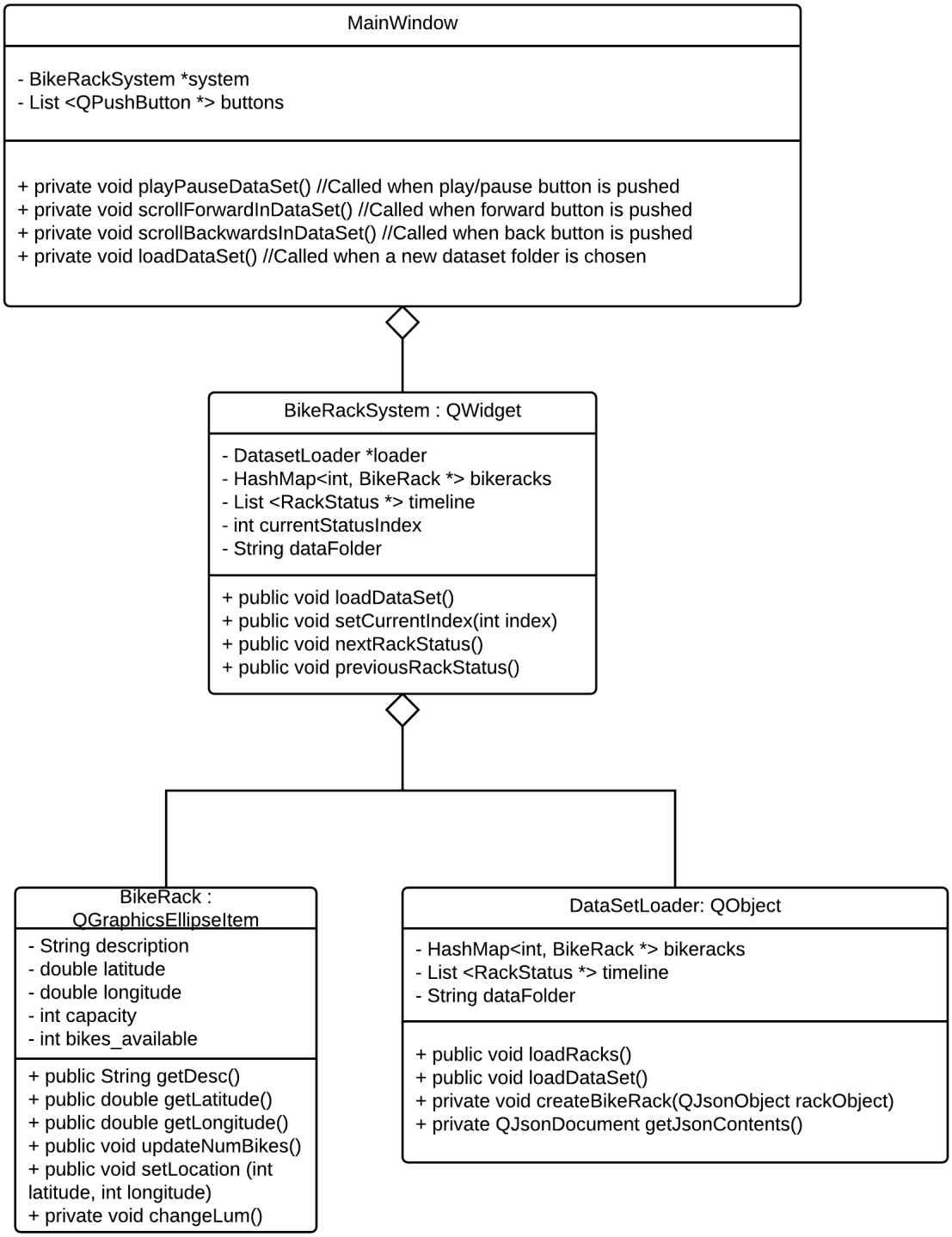


Figure 3.2: Class diagram for the visualization module written in C++

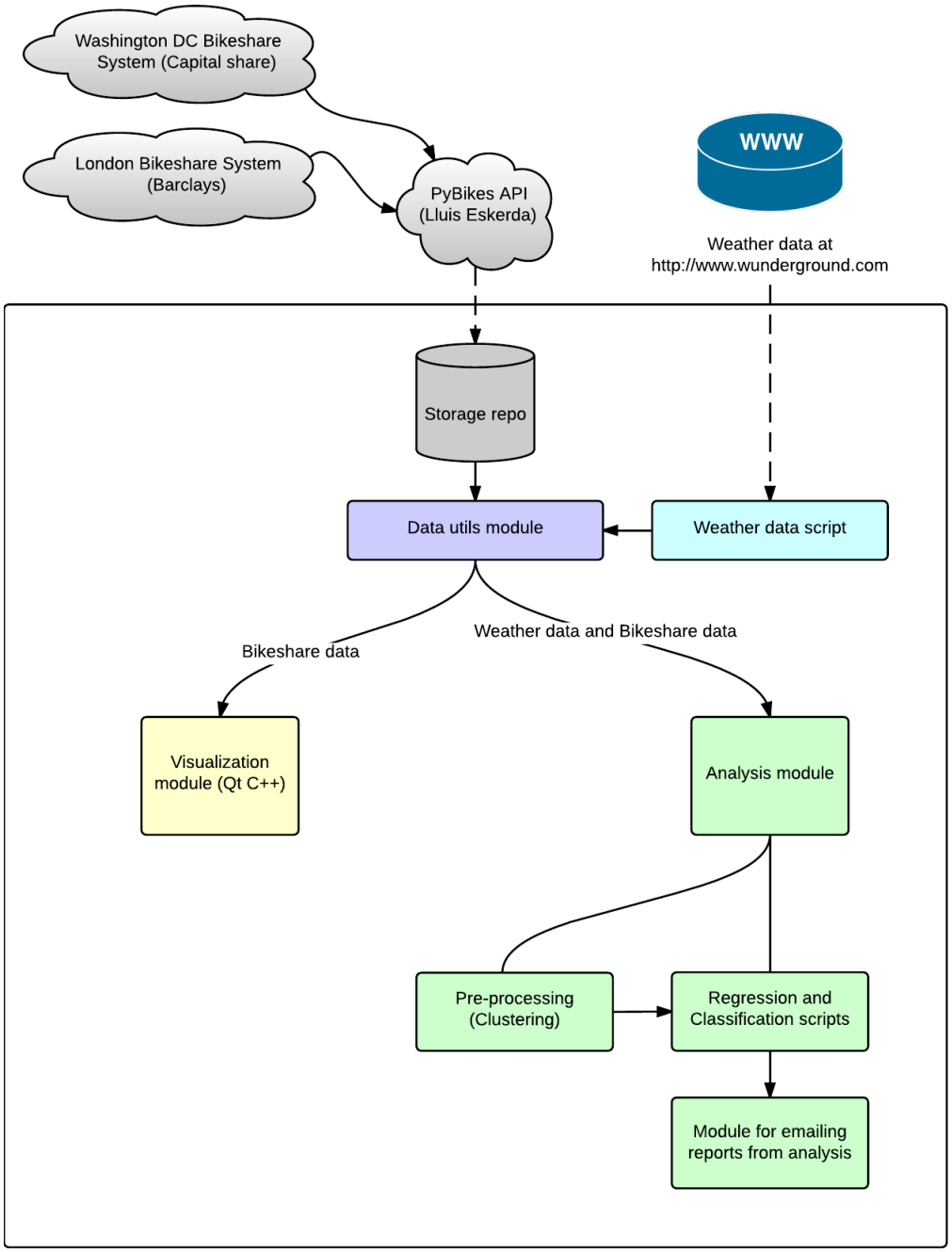


Figure 3.3: Software overview for this thesis

Part II

Analysis

Chapter 4

Machine Learning

4.1 Introduction

Machine learning is a subfield of computer science that deals with construction of models that can generalize datasets in a way that closely resembles the way human beings collect data, generate hypotheses and test them. In other words, the machine must learn to generalize and summarize the dataset rather than simply recall it.

There are several approaches to machine learning, based on the type of input given to the machine during learning:

- Supervised learning
- Unsupervised learning
- Reinforcement learning

4.2 Supervised learning

4.2.1 What is supervised learning?

Every fall at the University of Oslo, there is a programming course: Introduction to object-oriented programming INF1000. At the end of the course, all students are evaluated. In order to do so, the course instructor might devise an exam. An appropriate exam must fulfill the following criteria:

- The exam must be relevant. If students were only taught how to program in Java, it is unfair to ask students to program the final exam in Python.
- The exam must be challenging. If students are given the exact same problems that they trained on, they would simply be required to recall the answers and write them down.

The desired outcome of this situation is that students learn how to generalize and learn concepts, and then apply those concepts to problems that are related but not identical to the ones they have encountered before.

If one extends the student-exam concept to supervised learning, the machine would be given lots of training questions, with answers. The machine learning terminology for training questions with answers would be "training set with target variables". At the end of the learning period, the machine would be given test questions without answers, known as the "test set without target variables". The goal is that the machine will have generalized its knowledge from the training set to such a degree that it will be able to predict the target values for the test set fairly accurately (see figure 4.1). The results from such a prediction might then be compared to the actual target values in order to evaluate the performance of the estimator (see chapter 9).

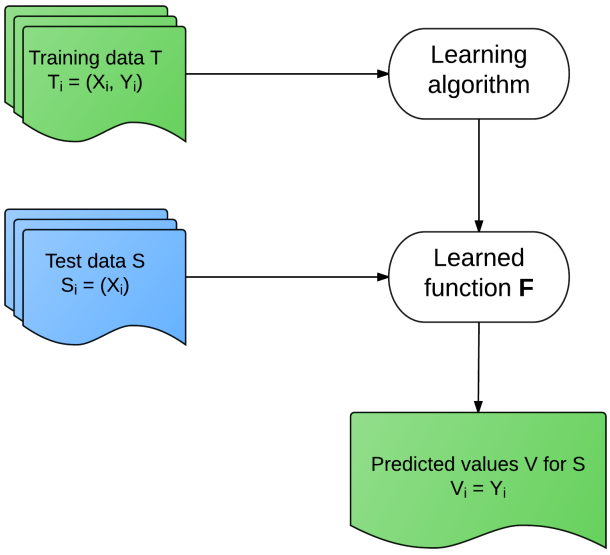


Figure 4.1: Basic overview for all supervised learning algorithms

When using machine learning with bike-share data, there were two major factors to consider:

- Bias-variance tradeoff
- Presence of interactions between data features i.e. the complexity of the ground truth function

4.2.1.1 Bias-variance tradeoff

The goal in supervised machine learning is to create a statistical model of the underlying process that created the training data. The bias-variance tradeoff is the issue of minimizing two different

sources of error that supervised machine learning algorithms suffer from:

- bias errors that occur from the models not being complex enough to reflect the complexity of the ground truth that created the training data.

- variance errors resulting from overcomplicated models that capture and model the noise in the training data.

In order to understand why these errors occur, it is important to note that the bias-variance tradeoff is the issue of determining the complexity of the models we choose.

Moore and McCabe (2002) [19] uses the analogy of a dartboard and a dart thrower to illustrate the bias-variance tradeoff (see figure 4.2). When the dart-thrower has low bias and has low variance, they tend to hit the bulls-eye very accurately. This is the ideal situation. However, in the case of high variance and low bias, the dart-thrower will miss uniformly in all directions. In the case of high bias, and low variance, the dart-thrower will miss in one direction, but the spread of the darts will be small.

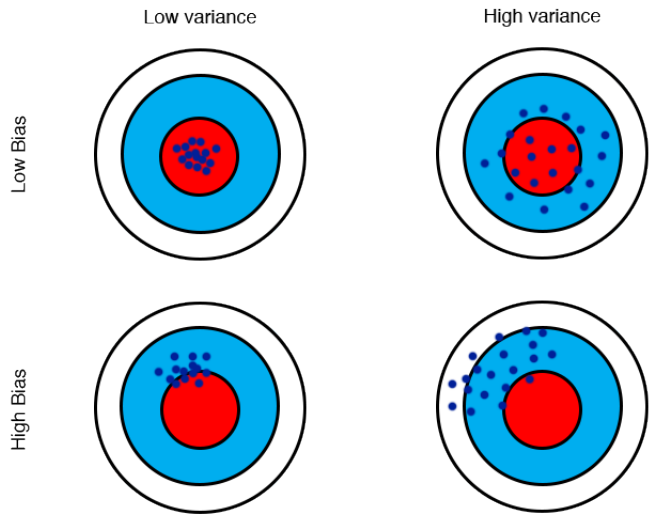


Figure 4.2: Bias-variance tradeoff illustration

It is possible to think of high-variance estimators as algorithms that are very flexible. Examples include deep decision trees (discussed in chapter 5), support vector machines etc. Similarly, low-variance estimators can be thought of as inflexible. Examples of low-variance estimators are: simple linear regressors, shallow decision trees etc.

The typical method of reducing variance is to increase bias, and vice-versa. It is generally (but not entirely) impossible to reduce bias and variance at the same time.

Below is an example of resolving the bias-variance tradeoff simply by choosing the complexity of a model. Figure 4.3 shows a dataset that was generated by adding some noise to a sine function (see Listing 4.1).

Listing 4.1: Generate a noisy but simple dataset using a sine function

```
#Imports
import numpy as np
import pylab as plt

#Introduce some noise to make the dataset slightly realistic
rng = np.random.RandomState(1)
X = np.sort(5.0* rng.rand(80,1), axis=0)
y = np.sin(X).ravel()
y[::5] += 3 * (0.5 - rng.rand(16))

#Create a test set
X_test = np.arange(0.0, 5.0, 0.01)[:, np.newaxis]

#Visualize the dataset
plt.scatter(X, y, marker="o", color="black", label="Data")
plt.legend()
plt.title("A noisy sine function")
plt.show()
```

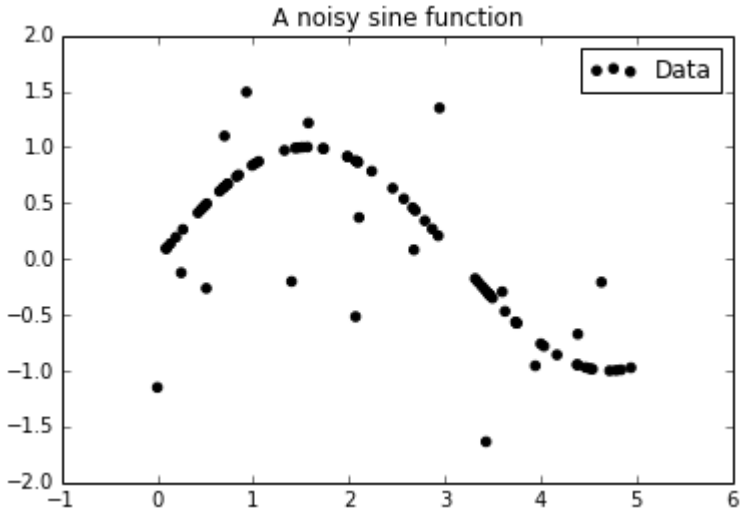


Figure 4.3: Noisy sine dataset

The first model, a Linear Regressor was fitted to the data (Listing 4.2) and the results are shown in Figure 4.4. It is clear from the prediction plot that the linear regressor was too simple (high-bias) to predict the dataset properly.

Listing 4.2: Predicting a noisy sine function using Linear Regression

```
from sklearn.linear_model import LinearRegression

#Create a Linear Regressor
estimator = LinearRegression()

#Fit the estimator
estimator.fit(X,y)

#Prediction
y_pred = estimator.predict(X_test)

plt.scatter(X,y, color="black", marker="o", label="Data")
plt.plot(X_test, y_pred, color="red", label="Prediction")
plt.title("Predicting a sine function with a linear regression
        model")
plt.legend()
plt.show()
```

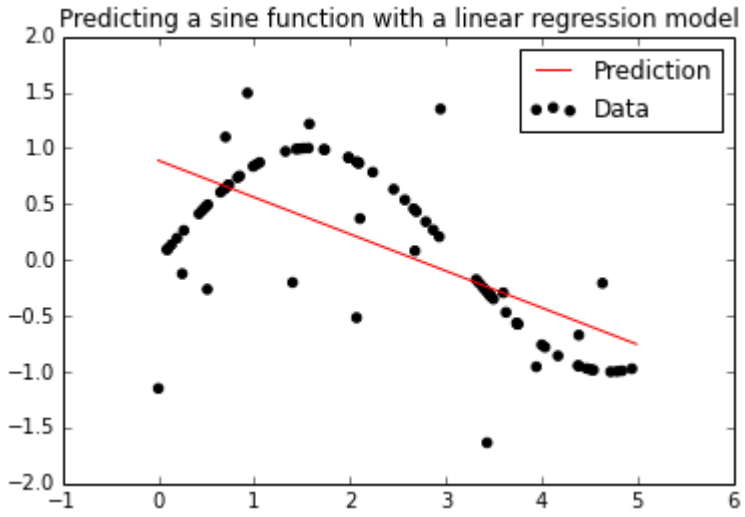


Figure 4.4: Fitting with a high-bias, low-variance model

Next, a Decision Tree Regressor was fitted to the data: Listing 4.3 and 4.5. This illustrates the opposite problem: the model has now learnt the training data by heart and considers the noise as part of the ground truth. This is a high-variance, low-bias model. Note that the low bias comes from the fact that decision trees only become biased when a certain class or value range dominates.

Listing 4.3: Predicting a noisy sine function using a Decision Tree

```
from sklearn.tree import DecisionTreeRegressor

#Create a deeper DecisionTreeRegressor
estimator2 = DecisionTreeRegressor(max_depth=None)

#Fit the estimator
estimator2.fit(X,y)
```

```
#Prediction
y_pred2 = estimator2.predict(X_test)


plt.scatter(X,y, color="black", marker="o", label="Data")
plt.plot(X_test, y_pred2, color="blue", label="max_depth=None")
plt.title("Predicting a sine function with a DecisionTree with
        maxdepth=None")
plt.legend()
plt.show()
```

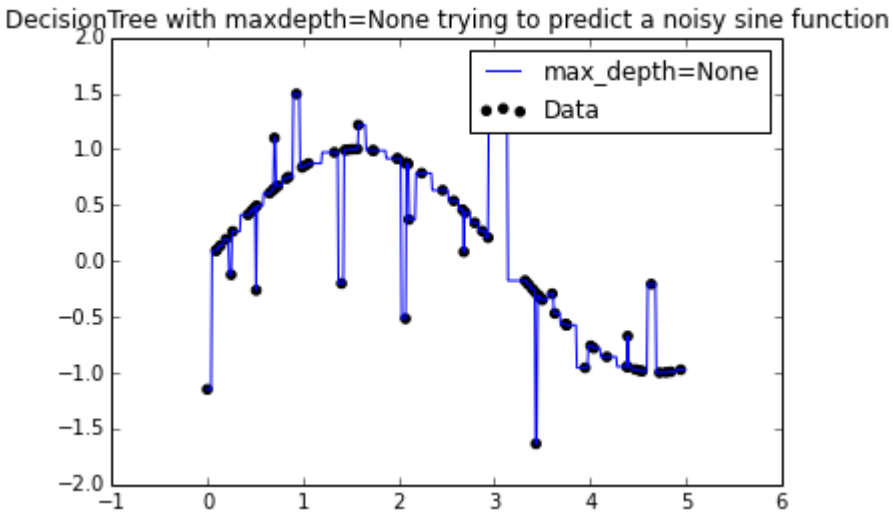


Figure 4.5: Fitting with a high-variance, low-bias model

Finally, the bias-variance tradeoff is resolved by making the decision tree shallower (and thus reducing the variance) as can be seen in Figure 4.6. It is quite clear that this model does not consider most of the noise as part of the ground truth. This is not the model that achieves the best error score on the training data so far, but it is the one that visually seems to be generalizing the best.

It should be noted that the reason why the tradeoff was so simple to resolve in this case was merely that the dataset itself was simple. Real-world data deals with a higher signal-to-noise ratio (SNR), and that is the domain of ensemble learning, namely Boosting and Bagging. Bagging works by averaging large amounts of estimators to reduce the variance (bias stays the same). Boosting, and AdaBoost in particular builds estimators iteratively by using reweighting techniques to teach each successive estimator more about the mistakes of its predecessor, thereby reducing variance and bias errors. Both of these techniques are disscussed in greater detail in chapter 6.

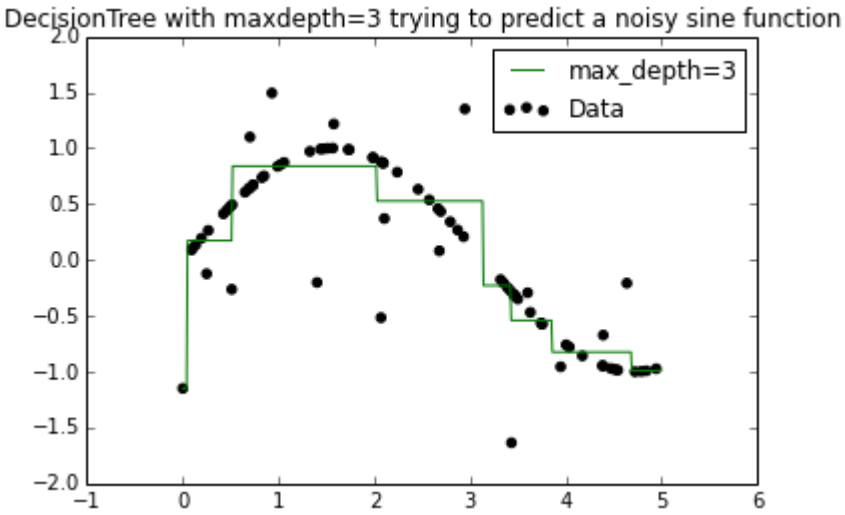


Figure 4.6: Bias-variance tradeoff resolved

4.2.1.2 Size of training set and complexity of the ground truth function

The function complexity of the number of bikes at a bike-share station includes (but is not limited to) the following features:

- Weather conditions (temperature, wind, rain etc)
- Time (day of week, time of day, month)
- Station variables such as the location of the station, bike capacity, altitude

It is not clear how important the different features are to the ground truth function, but it is clear that it might be quite complex. If this is indeed the case i.e. lots of non-linear interactions between features or feature interactions being different in different input spaces, then a low-bias, high variance estimator (i.e. a Decision Tree) is required to accurately predict the test set, along with a large training set.

In the bike-sharing context, traffic might behave completely differently in the month of June compared to the month of February, even on days when the weather conditions are comparable. A user might see that it is a sunny day outside but decide not to rent a bike purely based on the fact that it is the middle of February and they have a mental block against it. These kinds of cases will result in noise, where the algorithm will suspect that sunny days will bring up traffic but the reality will be different. It is therefore important to have a big enough dataset where such noise can be easily ignored.

4.2.2 Approaches to supervised learning

Supervised learning can be further split up into two approaches (see examples below):

- Classification: In classification, the target variables are class labels, i.e. all observations are of class A, but the problem is to determine which sub-class of class A they fall into.

  Example: All emails fall into the class *Email*, but spam classifiers strive to determine which ones fall into the *spam* and *relevant_email* subclasses respectively.

- Regression: In regression, the target variables are real/continuous values, i.e. the problem is to determine how big or small the target variable will be.

  Example: While browsing Netflix, Alice gave a 5/5 rating to some comedy movies, 2/5 to a few thriller movies and 3/5 to some mixed-genre movies. She then comes across a new movie which is in the thriller/horror category. Based on her previous experiences, Netflix can try to use a regressor to predict how well she will like the movie on a scale from 1-5.

4.3 Unsupervised learning

In unsupervised learning, the datasets are similar to the ones in supervised learning, except for one difference: they do not have a target variable. The problem in unsupervised learning is to uncover a hidden structure or feature interaction in data that has not been labeled. In this thesis, unsupervised learning, and more specifically clustering is attempted as a pre-processing method for identifying hidden structures in the bike-share systems.

4.3.1 Clustering

Clustering is the task of grouping a set of data points in such a way that they are more similar (in one or more feature spaces). A quick look at the datasets reveals that downtown and suburban stations behave differently (see figure 4.7). It is therefore logical to try and cluster stations into two groups:

- suburban stations
- downtown stations

In this thesis, each station was pre-processed with a clustering script. The stations were clustered by their behaviour; stations that emptied and filled up at similar times, were put in the same clusters. However, this did not lead to any improvement in the predictions.

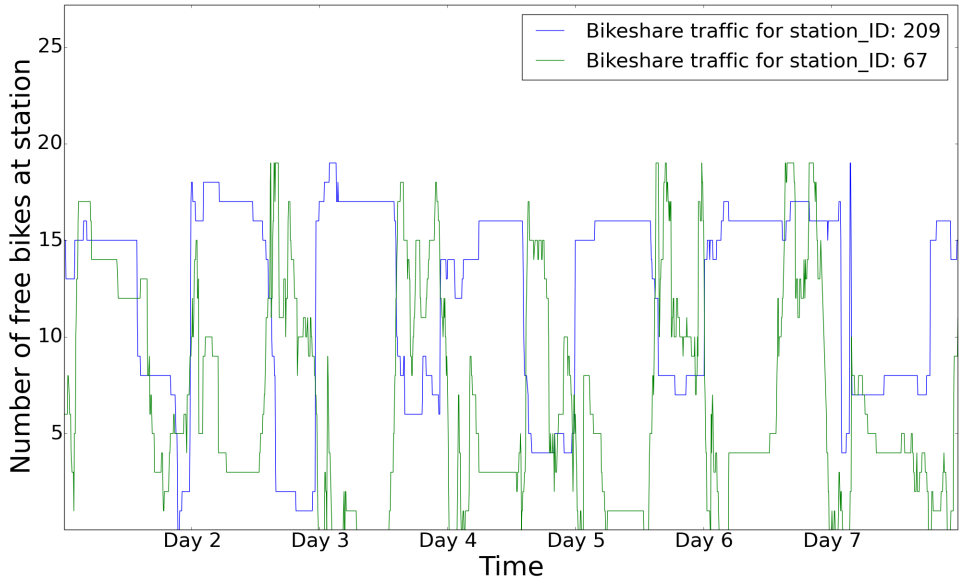


Figure 4.7: Number of bikes at a suburban (green line) and downtown station (blue line) in Washington D.C. over the course of a week

4.4 Classifiers

Classification is the problem of learning about a labeled set of datapoints and then using that knowledge to apply labels to unlabeled set of previously unseen datapoints.

A classifier is an algorithm that takes an observation and decides which sub-category the observation belongs to. An example of classification would be determining whether a given email should be tagged as "spam" or "non-spam".

In machine-learning, a classifier has access to a set of observations that have been accurately tagged. This set is known as a training set. A simplified example training set for the bikeshare domain is shown in table 4.1. This trains the classifier, which is then given a test set similar to table 4.1 except that the test set does not have a "Class" column. The classifier produces the "Predicted class" column for a test set (shown in table 4.2).

Note that classification is only used as a helpful concept to explain decision trees in this thesis, and not to actually predict shortages and overflows.

Hour of day	Day of week	Month	Lat	Lng	MAMSL	CC	T	wind	Class
00:00	5	5	38.89	-76.96	4.92	4	31	4.8	Balanced
12:00	7	4	38.90	-77.03	25.43	4	14.3	1.6	Balanced
17:00	7	7	38.89	-77.08	69.89	4	26.4	9.7	Shortage
02:00	3	5	38.89	-77.08	71.20	4	20.7	3.2	Shortage
01:00	5	4	38.88	-77.04	6.70	4	21.6	11.3	Overflow

Table 4.1: Example training set for a classifier

Note: Cloud cover (CC) is given in okta, temperature (T) in degrees Celsius, wind in km/h and altitude (MAMSL: Meters Above Mean Sea Level) in meters.

Hour of day	Day of week	Month	Lat	Lng	MAMSL	CC	T	wind	Predicted Class
2	2	6	38.92	-77.07	79.38	4	31.1	4.8	Balanced
20	1	5	38.90	-76.98	12.62	4	22	12.9	Balanced
17	7	6	38.89	-77.01	8.54	4	19.7	4.8	Shortage
22	6	6	38.85	-77.05	16.15	4	24.1	14.5	Balanced
0	6	3	38.90	-77.02	15.24	4	11	3.2	Overflow

Table 4.2: Possible classification output for the example test set

Note: Cloud cover (CC) is given in okta, temperature (T) in degrees Celsius, wind in km/h and altitude (MAMSL: Meters Above Mean Sea Level) in meters.

4.5 Regression

Regression in machine-learning is similar to classification, with the exception of the target variable, which is a real value as opposed to a class label. In the bike-share domain, this would mean that while classification algorithms might output whether or not a station will be in a overflow, balanced or shortage state, regression algorithms will output the expected number of bikes at the station.

The training set will be slightly different for regression algorithms (table 4.3), as will the output for the test set (table 4.4).

4.6 Evaluation metrics for models

Creating a prediction model is only the first part of creating a prediction system. Models need to be evaluated against each other in an objective manner to determine which one such a system should use. It is therefore common in machine learning to use the predictions from the test set and compare it to the actual data using error metrics. Error metrics are functions that take prediction values, compare them to actual values and output a score for how well the predictions fit the truth.

Hour of day	Day of week	Month	Lat	Lng	MAMSL	CC	T	wind	Free bikes
0	5	5	38.89	-76.96	4.92	4	31	4.8	5
12	7	4	38.90	-77.03	25.43	4	14.3	1.6	5
17	7	7	38.89	-77.08	69.89	4	26.4	9.7	0
2	3	5	38.89	-77.08	71.20	4	20.7	3.2	1
1	5	4	38.88	-77.04	6.70	4	21.6	11.3	13

Table 4.3: Example training set for a classifier

Note: Cloud cover (CC) is given in okta, temperature (T) in degrees Celsius, wind in km/h and altitude (MAMSL: Meters Above Mean Sea Level) in meters.

Hour of day	Day of week	Month	Lat	Lng	MAMSL	CC	T	wind	Predicted number of free bikes
2	2	6	38.92	-77.07	79.38	4	31.1	4.8	5
20	1	5	38.90	-76.98	12.62	4	22	12.9	5
17	7	6	38.89	-77.01	8.54	4	19.7	4.8	1
22	6	6	38.85	-77.05	16.15	4	24.1	14.5	5
0	6	3	38.90	-77.02	15.24	4	11	3.2	13

Table 4.4: Possible regression output for the example test set

Note: Cloud cover (CC) is given in okta, temperature (T) in degrees Celsius, wind in km/h and altitude (MAMSL: Meters Above Mean Sea Level) in meters.

4.6.1 Error metrics for Classification

In classification, the error metrics that are available in scikit-learn are:

- Information gain (entropy)
- Gini impurity

Given a region R in a dataset D, we can define the misclassification rate for R as $E_R = \sum_{c \in C} P(y \neq \hat{y})$ where C gives the number of subclasses present in D and $P(y \neq \hat{y})$ represents the likelyhood that the prediction of a sample will be wrong. This is known as the "Impurities metric".

The Information gain (or entropy) metric H_R is based on the misclassification probability mentioned above:

$$H_R = -\sum_{i=1}^{n} P_R(y) * log(P_R(y))$$

For CART type decision trees, the equation is:

$$H_R = -\sum_{i=1}^{n} P_R(y) * log_2(P_R(y))$$

Note the difference in logarithmic base, the CART type decision trees have log_2 simply because all the decision trees implemented in scikit-learn are binary trees.

The Gini impurity G_R is given as:

$$G_R = \sum_{y \in Y} P_R(y)(1 - P_R(y))$$

The classification trees in scikit-learn support either the gini impurity or entropy as their error metrics.

4.6.2 Error metrics for Regression

RMSE (Root Mean Squared Error) and R^2 (coefficient of determination) are two popular choices when evaluating regression models. The DSSG divvy project [7] uses RMSE, while scikit-learn uses R^2. In order to understand the choice of error metric in this thesis, the error metrics themselves must be understood first.

4.6.2.1 Residual sum of squares, Explained sum of squares, Total sum of squares

Regression is basically the process of applying a best-fit model to a series of real values. Given a regression model $\hat{Y}$ that approximates the data $Y = f(x)$, with the mean $\bar{Y}$ (see figure 4.8), for each datapoint y_i it is possible to calculate:

- How far the datapoint is from the model: $y_i - \hat{y}_i$
- How far the model is from the mean: $\hat{y}_i - \bar{Y}$

In statistical data analysis [20] Residual sum of squares is given as $RSS = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$, Explained sum of squares as $ESS = \sum_{i=1}^{n} (\hat{y}_i - \bar{Y})^2$ and Total sum of squares as $TSS = RSS + ESS$. The metric R^2 (coefficient of determination) is $R^2 = ESS/TSS$.

Since $TSS = RSS + ESS$, it also follows that:

$$R^2 = \lim_{RSS \to 0} \frac{ESS}{RSS + ESS} = 1.$$

In other words, the "best" fit possible for any regression model will give us an R^2 value of 1 (see figure 4.9) and similarly worse models will give us lower values of R^2 (see figure 4.10).

However, it would be too simplistic to simply use R^2 as the only metric. R^2 tends to be inflated [21] as more parameters are added to the regression model $\hat{Y}$. So for instance, in the domain of bike-share traffic, as the regression models account for more and more variables (such as temperature, precipitation, wind, location of the station, time of day, day of week etc), the R^2 score will increase without any guarantee that the newer variables decrease the residuals RSS. In this thesis, R^2 was therefore only used by the genetic algorithm to compare models that had the exact same number of variables since all estimators in scikit-learn implement a `score()` method which outputs the R^2 on a dataset with a trained

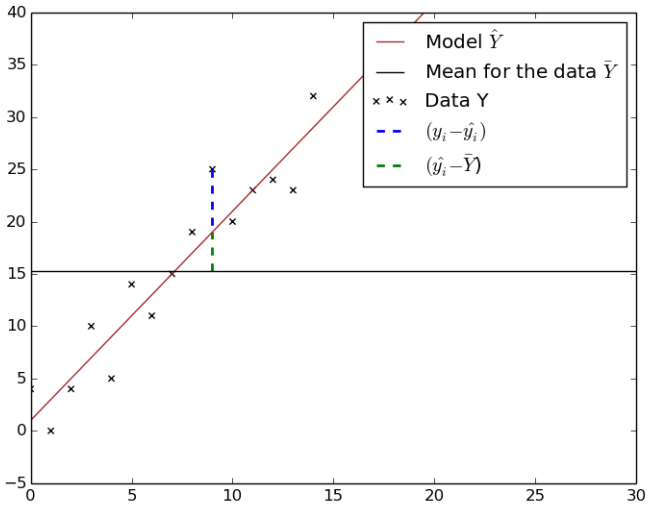


Figure 4.8: Estimating distance from the mean, distance from the model. The distance from the model is given by the blue dotted line, and the distance between the model and the mean by the green dotted line.

model. In contrast, the RMSE requires an extra step to compute which requires the training and test data to be stored at all times, which requires a large amount of memory. This is only practical for analysis on the final models, but not when comparing several estimators generated by a genetic algorithm.

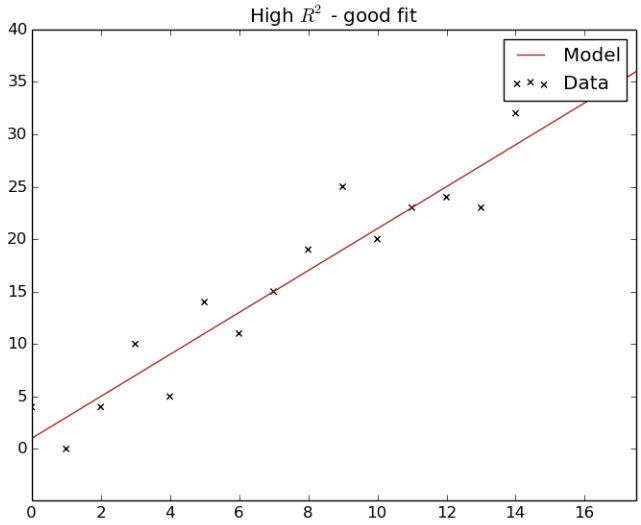


Figure 4.9: High R^2 - good fit

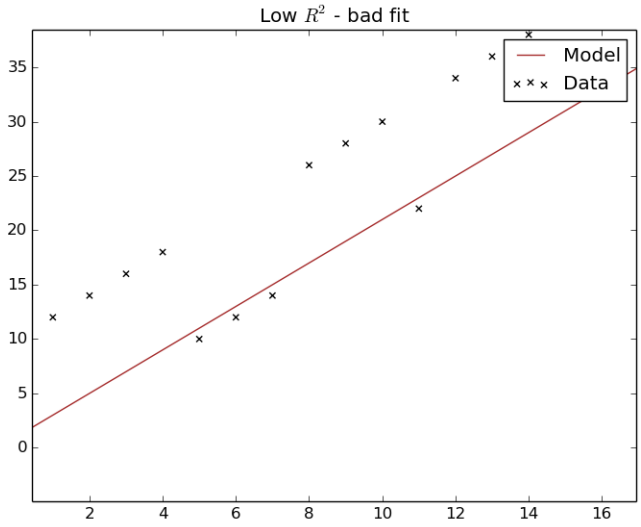


Figure 4.10: Low R^2 - bad fit

4.6.3 Root mean squared error (RMSE)

RMSE is another general purpose error metric for regressions. It is defined as : $RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$

Given a test set where:

$X = [1,2,3,4,5]$

$Y = [2,4,6,8,10]$

$\hat{Y} = [4,4,6,8,10]$

$\bar{Y} = \frac{1}{n}\sum_{i=1}^{n} y_i = \frac{10+8+6+4+2}{5}$

First, the Standard Error is calculated: $se_i = (\hat{y}_i - \bar{Y})^2$ (see table 4.5), then the Mean Square Error $MSE = \frac{1}{n}\sum_{i=1}^{n} se_i = 5.6$, and lastly the Root Mean Square Error: $RMSE = \sqrt{MSE} \approx 2.36$.

x_i	y_i	$\hat{y}_i$	$\bar{Y}$	$(\hat{y}_i - \bar{Y})$	$se_i = (\hat{y}_i - \bar{Y})^2$
1	2	4	6	-2	4
2	4	4	6	-2	4
3	6	6	6	0	0
4	8	8	6	2	4
5	10	10	6	4	16

Table 4.5: Calculating the standard error

The advantage of using $RMSE$ is that it provides an error metric that has the same unit as the target variable. In other words, if a model for bike-share traffic has $RMSE = 2.4$, it denotes that the model is off by approximately 2.4 bikes at all times. This is the reason the DSSG team chose this error metric. In this thesis, RMSE is used as the metric on the test sets so that the model error results could be compared to the ones achieved by the DSSG team.

Chapter 5

Decision Trees (CART)

A decision tree (also called CART for Classifcation and Regression Tree) is a statistical prediction model, that can be used for classification or regression. The structure of the model is a tree structure, where each node represents a question and the child edges from the node represent the possible answers to that question.

Consider a machine that asks the following questions about an email that's received:
Machine: Does the word "free" appear more than three times?
Answer: Yes
Machine: Does this email contain the words "lottery" and "won"?
Answer: Yes
Machine: This email is predicted to be spam.

The above is the basic concept behind decision trees i.e. it asks questions to determine the predicted class/value of the input data. Decision trees are chosen in this thesis for their transparency, as well as the fact that scikit-learn provides a `feature_importance()` function for their decision tree class. This helps identify the factors that affect bike-share traffic in a much more intuitive way than estimators like SVMs, neural networks etc. This chapter focuses on the representation of the decision tree, the learning algorithm that creates the structure, adjustable hyperparameters along with pros and cons of using decision trees.

5.1 Representation

We begin by visualizing an example of a simple learned decision tree (figure 5.3) that has been trained on a very small training set (leaf nodes usually have a wide range of values) with a depth of two. Each question can be answered as a yes/no question (because this is a binary tree). The "yes" child nodes are marked in green, and "no" child nodes in red. The leaf nodes contain target values (in this case number of bikes) recorded in the training set.

Note that we can ask different questions in the left and right subtrees. For instance, if it is not true that the time of day $06:00 \leq T \leq 10:00$, we can ask a different question in the left subtree (ex: $altitude \leq 70m$?). It should also be mentioned that the authors of scikit-learn decided to implement binary decision trees to avoid code complexity. In general, decision trees need not be binary [9].

5.2 Learning algorithm

Decision trees ask questions when learning a training set, but the important part is to ask them in the right order. Let's assume that a set of questions Q looks like this:

- Is it raining?
- How much rain has been recorded in the last hour?
- Is it sunny?
- What is the time of day?
- What day of week is it?
- What month is it?
- Where was the observation recorded? (downtown or suburban)

It is important to note while that the answer to the questions might be categorical (ex: downtown or suburban), real values (time of day), boolean (raining/not raining) etc, these answers must be represented numerically. This is because scikit-learn estimators only accept numpy arrays, which may only contain numerical values. An overview of the learning algorithm is provided below:

Algorithm 1 Decision tree learning

```
1: function DECISION TREE LEARNING(Training_set, Q)
2:     tree ← NULL
3:     while not STOP-CRITERION(tree, Training_set) do
4:         q ← FIND-BEST-SPLIT(Q, Training_set)
5:         d_L, d_R ← SPLIT(Training_set, q)
6:
7:         Left ← DECISION TREE LEARNING(d_L, Q)
8:         Right ← DECISION TREE LEARNING(d_R, Q)
9:
10:        tree ← (q, (Left, Right))
11:    end while
12:    return tree
13: end function
```

The `Split` and `Find-best-split` functions are described in further detail in section 5.2.2. Section 5.2.1 describes how the decision tree decides when to stop, i.e. the `Stop-criterion` function.

5.2.1 Stopping criterion

The `Is-stop-criteria-met` function is implementation-specific, and it is a boolean function that that returns True/False for a subset of the training set $s \in S$. The stopping criteria can be split into two:

- The hyperparameters $min\_samples\_split$ (least number of samples required to split a node) and $min\_samples\_leaf$ (minimum number of samples for a leaf node) are checked to see if the node should be split further or not.
- The hyperparameter max_depth of the tree is checked against depth of the current internal node to determine whether $depth_{current\_node} == max\_depth$.

5.2.2 Finding the best split

The `Find-best-split` function boils down to "If you could ask only one question at this time, what would that question be?". Scikit-learn uses a metric to determine which question splits the data in the "best" possible manner. The metrics supported are Gini impurity and entropy for classification trees (see section 4.6.1), and MSE for regression trees (mentioned in section 4.6.3).

Consider the following 2D projection of how an example dataset is split by three questions asked by a decision tree (figure 5.1). Each of the three lines drawn in red, green and blue represent a question.

The red line isolates five instances of the class x on one side. It is possible to define the red line as a binary question (hereby known as the "red question"), and all datapoints plotted to the left of the red line as answering "yes" to the red question. Therefore all datapoints d_i answering "yes" to the red question, are surely $d_i \subset x$. In other words, the red line provides a 100% accuracy rate on one side of its splits.

Since neither the blue nor the green line provides such a high level of accuracy on either side of their respective splits, the red line is chosen as the question with the best split, and therefore is the first question to be asked. The green line is chosen second, as it has the second best split, and lastly the blue line is chosen as it has the worst split of the three. The metric used in this example is known as entropy (see section 4.6.1). Note that it is not necessary to ensure that one side of a split assures a 100% accuracy rate.

When analyzing regression datasets, the questions separate real values instead (figure 5.2).

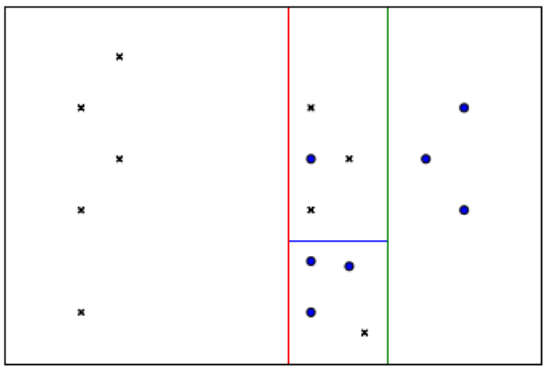


Figure 5.1: Binary class dataset divided into three partitions by a decision tree.

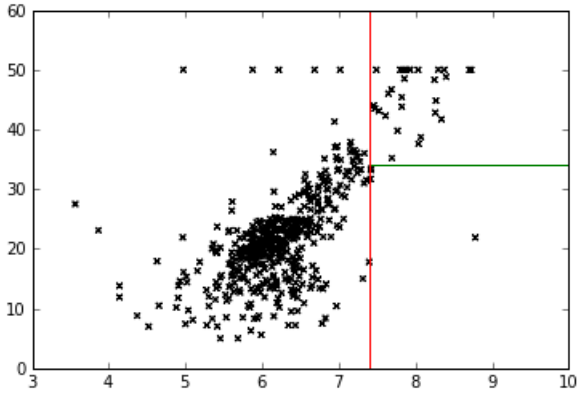


Figure 5.2: Regression dataset divided into two partitions by a decision tree

In the scikit-learn implementation of decision trees, there are two different strategies for choosing the "best" split. The first option is to do an exhaustive search on all possible splits and find the split that scores the best on the chosen error metric (MSE for regression, gini/entropy for classification). Another option is to pick a set of random splits and choose the best scoring split in the set.

5.2.3 Miscellaneous remarks

Any question that is asked once in a subtree will not be asked again, since asking that question again would not split the data any further; the reason being that the information gain will simply be zero. Looking back at section 5.2.2, it would be equivalent to drawing the red line again and again after we are done drawing both the red, green and blue lines.

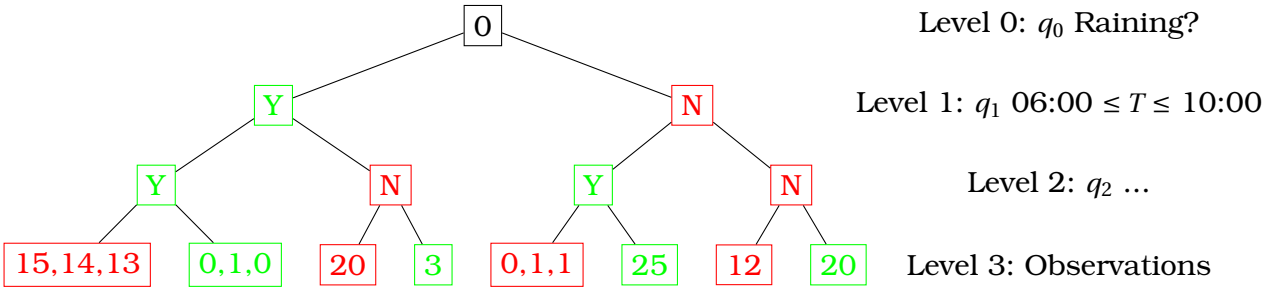


Figure 5.3: Example of a learned decision tree. Terminal nodes store the observed number of bikes at a bike station. Note: This is a Decision Tree Regressor

5.3 Strengths and weaknesses

The strengths [22] of decision trees are:

- Easy to understand when visualized as boolean logic (see figure 5.3). Models like neural networks, SVMs are not as easy to interpret.
- High variance: Decision trees have high enough variance to successfully predict complex functions. This is a double-edged sword (see below under "disadvantages of decision trees").
- Fast predictions on complex models: The cost of performing a prediction is $O(log(h))$ where h is the depth of the decision tree. This is very low compared to the potential number of decision boundaries (there are at least $2h-1$ leaf nodes, and at most $2^{h+1}-1$).

The disadvantages of decision trees include:

- The learning algorithm is greedy by definition, and there is no guarantee that it will find a globally optimal split at each internal node. The task of finding the optimal decision tree for a training set is NP-complete.
- Bias: Decision trees can become biased when the dataset is not balanced (applies to both classification and regression problems). This can be solved by balancing the dataset (as was done in this thesis when dealing with classifiers).
- High variance might lead to overfitting. A decision tree might create an overly complex model of the data that fits the noise as well as the ground truth.

In summary, decision trees seem like a good initial choice for the complex nature of the data (see figures 3.1 and 4.7). Chapter 7 investigates whether they are flexible enough to capture the complexity of the data.

Chapter 6

Ensemble learning

Ensemble learning algorithms are meta-estimators that use multiple learning estimators to produce better performance than any of the individual estimators could. Theory [23] [24] suggests that both boosting and bagging should bring down the variance of the decision tree estimators mentioned in the chapter 5 and they are therefore relevant to look at in this thesis.

6.1 Bagging

Bagging was introduced in 1994 by Leo Breiman [23], and it stands for Bootstrap Aggregation. He showed through a series of tests on regression and classification cases that bagging gave significant gains in accuracy. The bagging algorithm can be divided into three steps:

- Bootstrapping the training set randomly
- Training a set of models M on bootstrapped data
- Averaging all models $m \in M$ to create a final model m_{final} such that $m_{final}(x) = \frac{1}{n} \sum_{i=0}^{n} m_i(x)$

6.1.1 Bootstrapping

Bootstrapping is the act of taking an observation $(x_i, y_i) \in D$ and randomly resampling it to create sets like $b = ((x_i, y_i), (x_m, y_m), (x_j, y_j))$ where i, j, m are all randomly picked. Since all the datapoints in D are randomly chosen to be resampled (sampling by replacement, see "unordered samples with replacement" in [25]) into bootstrap sets like B, it is very improbable that $B = D$. It is important to note that a bootstrapped dataset B_i is built independently from B_j, and is therefore likely to be different. As an example, assume $D = 0, 1, 2, 3, 4, 5$. This may be resampled into one bootstrap dataset $B_1 = 0, 0, 1, 2, 4, 1$.

Scikit-learn has a bootstrapping mechanism that can be used as shown in Listing 6.1 to create bootstrapped datasets. The dataset used in the example is a regression dataset that is included in scikit-learn [22]. The bootstrapped datasets can be seen in figures 6.1, 6.2 and 6.3. Note that the datasets shown are not identical, even though they are similar.

Listing 6.1: Bootstrapping

```
from sklearn import cross_validation
from sklearn import datasets

dataset = datasets.load_boston()
data, target = dataset.data, dataset.target

#Plot original dataset
plt.scatter(data[:,5], target, marker="x", color="black")
plt.title("Complete training set")
plt.show()

bs = cross_validation.Bootstrap(len(target), n_iter=3)
n = 0

#Plot bootstrapped datasets
for train_index, test_index in bs:
  n += 1

  bs_xtrain = [data[:,5][i] for i in train_index]
  bs_ytrain = [target[i] for i in train_index]

  bs_xtest = [data[:,5][i] for i in test_index]
  bs_ytest = [target[i] for i in test_index]

  plt.scatter(bs_xtrain, bs_ytrain, marker="x",
   label="Training set", color="blue")
  plt.scatter(bs_xtest, bs_ytest, marker="x",
   label="Validation set", color="red")
  plt.title("Random bootstrap sample #%d
        of %d (n_samples = %d)" %(n,len(bs),bs.n_iter))
  plt.legend()
  plt.show()
```

6.1.2 Fitting bootstrapped datasets and aggregation

Once the bootstrapped partitions have been created, the next step is to create a bunch of models on each of the partitions. In the example shown in Listing 6.2, the models are created by hand, but this is just for illustration purposes. Finally, the models are aggregated by averaging the output $m_i(x)$for all $x \in X$. The final model can be seen in figure 6.4 on page 61.

Listing 6.2: Model fitting

```
m_x = range(4,10)
m1 = [(10*x - 40) for x in m_x]
m2 = [(11*x - 45) for x in m_x]
```

```
m3 = [(11.5*x - 45) for x in m_x]
models = [m1,m2,m3]

#Aggregate bootstrap models to form a final model
m_final =[(sum(model[i] for model in models))/len(models)
 for i in range(len(m1))]


for i in range(len(bs_data)):
  x_train, y_train, x_test, y_test = bs_data[i]

  plt.scatter(x_train, y_train, marker="x", label="Training set",
   color="0.25")
  plt.scatter(x_test, y_test, marker="x", label="Validation set",
   color="0.75")

  plt.plot(m_x, models[i], label = ("Model for bootstrap #%d"
        % (i+1)), color="red", linestyle="--")

  plt.title("Model#%d fitted to bootstrap partition #%d"
   %((i+1),(i+1)))
  plt.legend()
  plt.show()

plt.scatter(data[:,5], target, color="black", marker="x",
 label="original training set")
for i in range(len(models)):
  plt.plot(m_x, models[i], label = ("Model for bootstrap #%d"%
  (i+1)), color="red", linestyle="--")

plt.plot(m_x, m_final,color="green", linestyle="-",linewidth=1,
 label="Mean of bootstrap models")
plt.legend()
plt.title("Aggregated model")
plt.show()
```

Figure 6.1: Model #1 on bootstrapped dataset #1

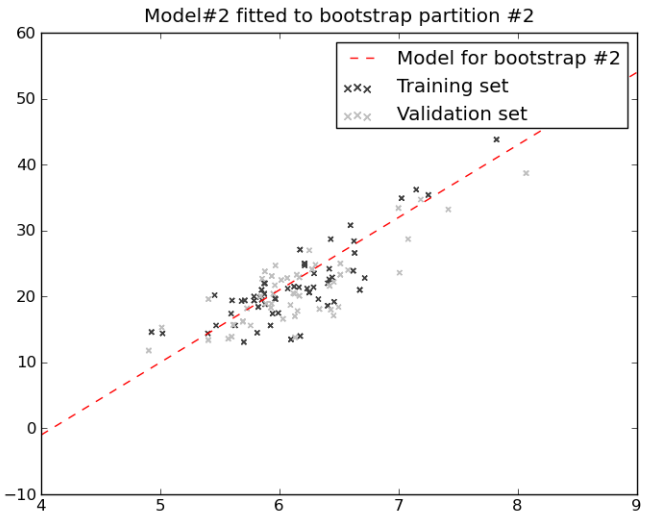


Figure 6.2: Model #2 on bootstrapped dataset #2

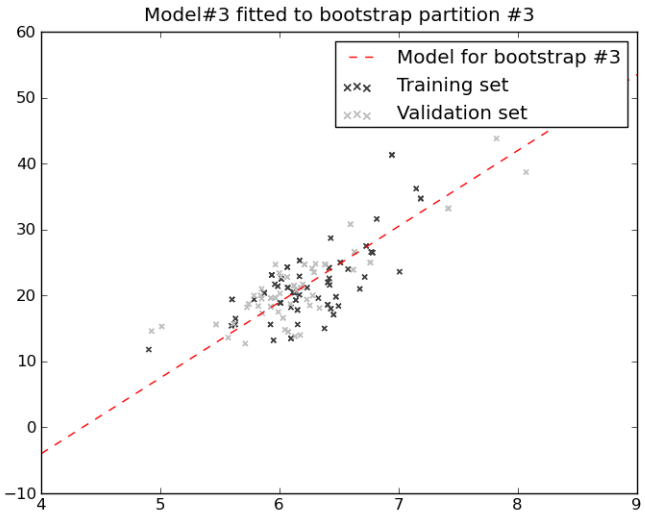


Figure 6.3: Model #3 on bootstrapped dataset #3

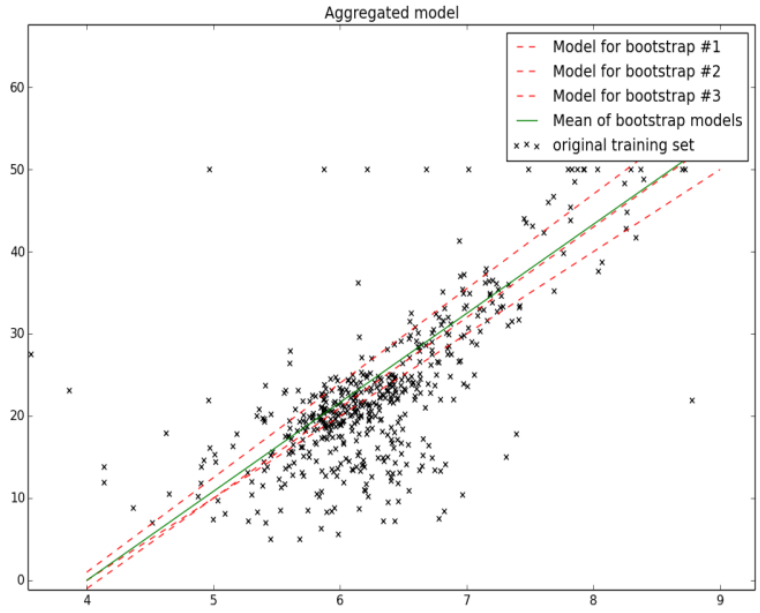


Figure 6.4: Aggregated model

6.2 Random Forest

The Random Forest algorithm was introduced by Leo Breiman in 2001 [26]. It has been shown to have excellent performance as shown in Caruana & N-M [27], and came in a close second only to boosted decision trees, beating relatively complex algorithms such as neural networks and SVMs.

Random Forests combine bagging with the concept of random feature selection introduced by Ho [28] and Amit and Geman [29]. The learning algorithm for random forests is given below:

Algorithm 2 Random Forest Learning

function RANDOM FOREST(dataset D, n_estimators = n)
    $((x_1, y_1), ..., (x_m, y_m)) = D$
    $B \leftarrow$ CREATE-BOOTSTRAPS(D,n)
    $RF \leftarrow \{\}$
    for all $d_i \in B$ **do**
        $S \leftarrow$ Total feature space for d_i
        $s \leftarrow$ SELECT-K-RANDOM-FEATURES(S, k)
        $Q \leftarrow$ GENERATE-QUESTIONS(s)
        $t_i \leftarrow$DECISION-TREE-LEARNING(d_i, Q)
        $RF \leftarrow$APPEND(RF, t_i)
    end for
    return RF
end function

In scikit-learn, the number k for the size of the random feature subspace is by default set to $\sqrt{p}$ for a dataset with p features.

Generally, one can grow an infinite number of trees without danger of overfitting [9], due to the law of large numbers; a theorem stating that when an experiment is performed enough times, the mean of the results will be close to the expected value. Increasing the number of trees in a random forest decreases the variance of the estimator, while maintaining the bias of the forest. The number of trees in scikit-learn is controlled by the `n_estimators` hyperparameter.

Random forests do not require cross-validation, due to the so-called OOB (out-of-bag error estimate). During the learning process of the random forest, bootstrap datasets $D_i \in B$ are created by taking examples with repetition. The datapoints left out of every dataset $D_i \in B$ are called out-of-bag samples for that dataset (there are N such samples in total, where N is the number of samples in the original dataset). After the estimators $tree_i \in RF$ are created, they are all tested on the out-of-bag samples, and the error is the generalized error metric for random forests.

Other than the hyperparameters mentioned above, random forests also include the hyperparameters mentioned for decision trees and these are applied to all decision trees in the forest.

6.3 Boosting

A boosting algorithm is a meta-estimator for machine learning that combines multiple weak (or simple) estimators to form a single strong estimator. The prediction techniques make it a type of ensemble method, and was first introduced by Robert Schapire [30].

The core concept of boosting can be explained by comparing it to students preparing for exams. Assume that a student is attending the course INF1000 (an introduction course at the University of Oslo), and wants to prepare for the final exam. The INF1000 course focuses on two primary subtopics:

- Programming
- UML drawings

When preparing for the exam, students must learn to solve problem sets that involve both subtopics. A student may therefore look at earlier exams to get an idea of what kind of problems might be expected on the final exam. This would be the training dataset D in a machine learning context. The learning is completed in three steps:

- The first objective of the learning period before the exam could be to find out more about the weaknesses/strengths in the

topics covered in INF1000. This is accomplished by just taking an exam from the previous years and solving it to the best of the student's abilities.

- After solving the exam, the student looks at the error they made, and analyses the problems they made an error on. Upon noticing the mistakes, the student decides to spend more time learning more about them. This ensures that they are less likely repeat the same mistake again.
- Lastly, the student decides to repeat the process again and again, until time has run out and it is time to sit for the final exam.

6.3.1 AdaBoost

In this thesis, the boosting algorithm called AdaBoost (short for Adaptive Boosting) was used. The pseudocode is shown below:

Algorithm 3 AdaBoost Learning for classification

1: **function** ADABOOST(dataset D, n_estimators = n, estimator_type=T)
2: $E \leftarrow$ CREATE-ENSEMBLE(T, n) ▷ Create n estimators of type T
3: **for all** Estimator $e_i \in E$ **do**
4: TRAIN(e_i, D, wts)
5: $Y \leftarrow D$.target
6: $\hat{Y}$ = PREDICT(e_i, D)
7: $e \leftarrow wts * (Y \neq \hat{Y})$ ▷ Calculate weighted error
8: $\alpha_i = 0.5 * \frac{log(1-e)}{e}$ ▷ Calculate coefficient alpha
9: $wts \leftarrow wts * \exp(-\alpha_i \cdot Y \cdot \hat{Y})$ ▷ Update weight vector
10: $E \leftarrow$APPEND(RF, t_i)
11: **end for**
12: **return** E
13: **end function**

Some remarks:

- The algorithm outlined above is only for classification purposes. For regression, the pseudocode must be changed slightly; instead of looking at misclassifications, the algorithm looks at loss functions [31].
- wts in the algorithm is a weight vector for all datapoints in D, such that wts_i for (x_i, y_i) is some real value $v \in \mathbb{R}$.
- The calculation for α is related to a property of AdaBoost. Without going into too much detail, the formula is derived from the surrogate loss function $C_{ada} = \sum \exp(-y^i f(x^i))$ that AdaBoost corresponds to. In essence, the weight of a given point (x_i, y_i)

increases if $y_i \neq \hat{y}_i$ and decreases otherwise (i.e. increase the weight of samples that the predictor erred on, and decrease otherwise).

- e signifies the error metric (mean accuracy for classifiers). For regression purposes, the formula $\alpha_i \leftarrow \frac{barL}{1-barL}$, is used as the error metric (L signifies either a linear or exponential loss function) (see Drucker 1997 [32]). e is calculated by multiplying the weight vector wts with the mislabeling rate: $Y \neq \hat{Y}$

After the AdaBoost algorithm has finished learning the training data, it links the decision boundaries of all the weak estimators in some way to form one strong estimator. In scikit-learn, all the estimators in the ensemble perform a majority vote to determine the output of the ensemble as a whole.

6.3.1.1 Criticism

It should be noted that Long & Servedio (2007) [33] concludes that "convex potential boosters cannot withstand random classification noise". The consequence of this result is that if a non-zero fraction of a training dataset is mislabeled, the boosting algorithm will try to fit the noise and will produce a final model with an accuracy no better than 0.5 (i.e. random). This finding does apply to AdaBoost, but as it can be seen in chapter 7, AdaBoost still provides predictions that are on-par with Random Forests.

6.4 Strengths of ensembles over individual estimators

Ensembles by their very nature, reduce the variance of any single estimator and can therefore be expected to deliver better results. In addition, when the ground truth function is complex, ensemble estimators tend to capture this better as they are more flexible. In chapter 7, the improvements that can be achieved by employing either boosting or bagging techniques are investigated.

Part III

Results and conclusion

Chapter 7

Results

This chapter looks at how decision trees, random forests and Adaboost compare to the Poisson models used by the DSSG tree. It is worth mentioning that the time-window for the predictions in this thesis was much larger, as the intention of the software is to help customers plan their commute in advance.

7.1 Preliminary results

All regressors tested in this section, were run with the following conditions:

- Total length of dataset: 3 months (May 2014 - July 2014)
- Dataset taken from city: Washington DC
- Ratio of training to test samples: 80-20
- Feature space:
  - Time of day (hours)
  - Day of week
  - Day of month
  - Month
  - Station latitude
  - Station longitude
  - Station altitude
  - Temperature in degrees °C
  - Precipitation in mm
  - Cloud cover in okta

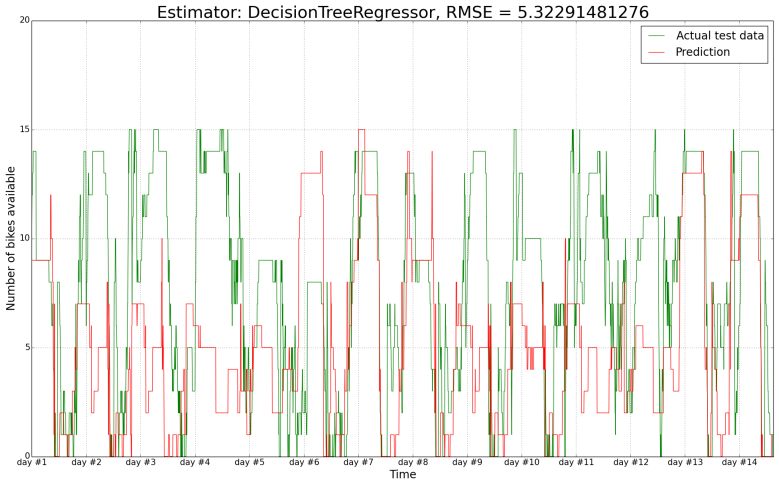


Figure 7.1: Predictions from a single decision tree

7.1.1 Decision Trees

The results in 7.2 show that using a decision tree as an estimator will deliver comparable results to the ones achieved by the DSSG team. The DSSG team achieved an RMSE of approximately 6.5 when predicting more than two hours ahead (see section 2.1.3). In comparison, the decision tree model above in figure 7.2 predicted over a period of 14 days and maintained an RMSE of 5.3.

7.1.2 Random Forests

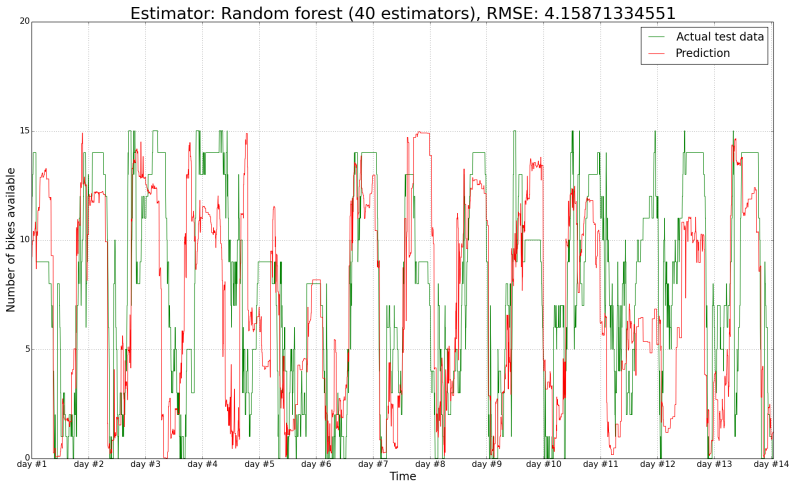


Figure 7.2: Predictions from a random forest containing 40 decision trees

A random forest regressor fitted to the same test set achieves

an RMSE of 4.15, which is better than the results achieved by the decision tree in section 7.1.1. The improvement over the decision tree can be calculated as follows:
Improvement $((RMSE_{dt} - RMSE_{rf})/RMSE_{dt}) * 100)\% = ((5.3 - 4.15)/5.3) * 100)\% = 21.7\%$

7.1.3 AdaBoost

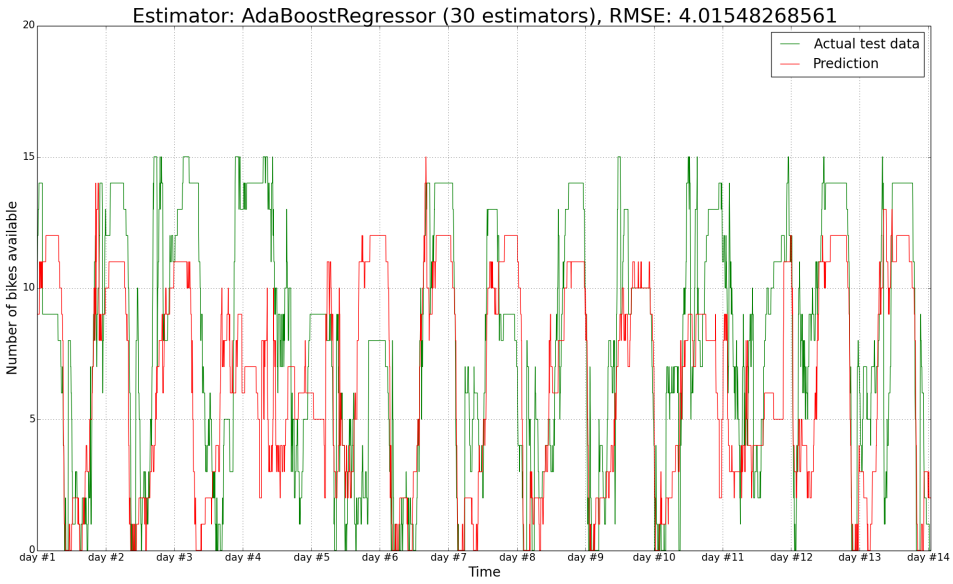


Figure 7.3: Predictions from an AdaBoostRegressor containing 30 decision trees

The AdaBoost regressor achieves even better results than the random forest, with a RMSE of 4.01. Again the improvement over a single decision tree is calculated as follows:
Improvement $= ((RMSE_{dt} - RMSE_{ada})/RMSE_{dt}) * 100)\% = ((5.3 - 4.01)/5.3) * 100)\% = 24.3\%$

7.2 Effect of training-test ratio on estimator accuracy

The training-test split ratio is usually determined by the Pareto principle ("80% of the effects come from 20% of the data"). However, there does not exist any general consensus that any one split ratio is better than the other (in terms of the accuracy of the trained model). Generally speaking, if the estimators are complex (i.e. high variance), and are given lots of data, they might eventually overfit. In contrast, when estimators are not given enough data, they might underfit and not perform well enough.

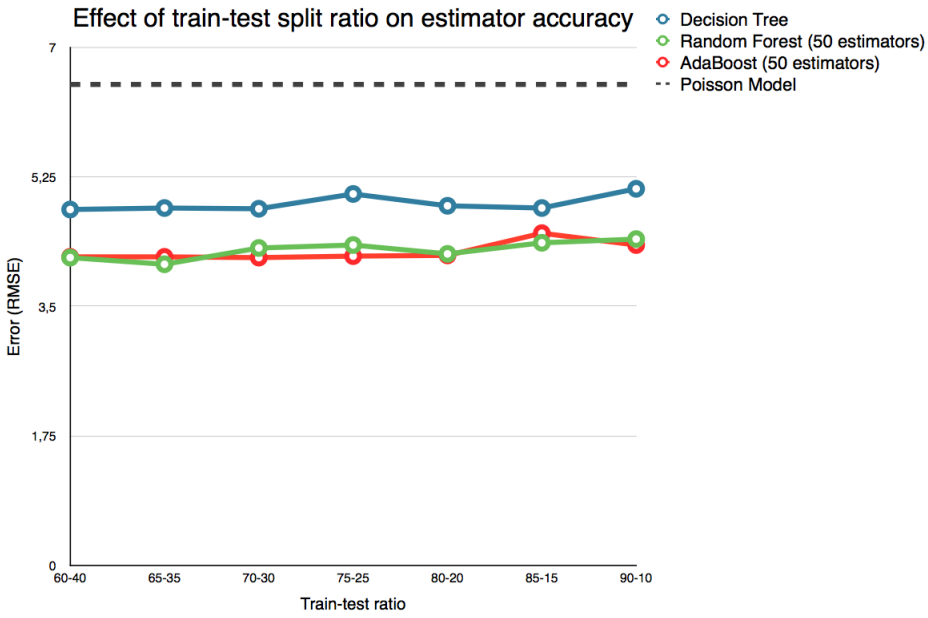


Figure 7.4: Effects of train-test ratio

Figure 7.4 shows the effect of different training-test ratios when training the different estimator types. The bike-share datasets are slightly special (in terms of machine learning) as the target variable (number of bikes) follows a periodic pattern. This would explain why changing the ratio of training to test data does not affect the error rate significantly; even at a 60-40 ratio, the estimators know enough to predict accurately. The general 80-20 ratio is therefore hold here; a 60-40 split is enough (for this dataset).

7.3 Effect of ensemble size

In the case of random forests, Breiman 2006 [26] states that they cannot overfit just because one increases the size of trees in the forest. They can however underfit if the ensemble size is not big enough. It appears from figure 7.5 that the number of estimators $n = 10$ is enough, and extending the number of estimators beyond that point does not appear to give any benefits.

In contrast, AdaBoost can overfit if it has too many estimators [24]. This is due to its additive nature, that adds a lot of complexity to the models (increasing the variance). There does not appear to be any significant benefit from fitting a large number of models, but repeated experiments (that form the basis of figure 7.5) show that there might be a "sweet spot" at $n = 35$. However, the potential gain (0.2 RMSE) is not big enough to be significant.

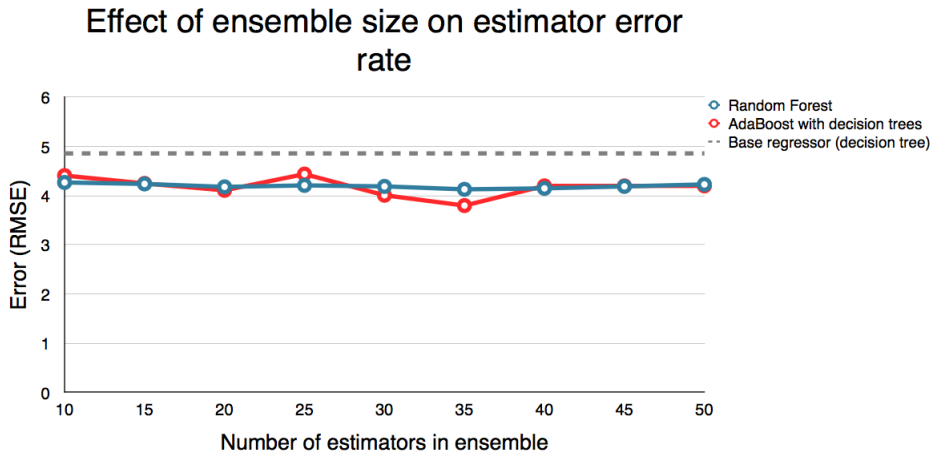


Figure 7.5: Effects of ensemble size on error rates. The dotted gray line reflects the error rate of the decision tree, which is the base estimator used in the AdaBoost and Random Forest algorithms

7.4 Feature importances

All three estimator types (decision trees, random forests, boosted decision trees) have the option to rank the features in the order of importance. The order of importance simply means how important the estimator considers them to be in terms of determining the target variable (number of bikes). Figures 7.6, 7.7 and 7.8 show how they are ranked by the three algorithms mentioned above.

It is evident from table 7.1 that the feature importances are fairly similar amongst the three models. The top ranking feature amongst all three estimators is the "epoch" feature. The epoch at any given time is defined as the number of seconds since January 1st 1970. This is a strong indication that there is (at least) one time-based feature hidden in the data that is causing a lot of the variance. It is not clear what that feature could be as the time of day (in hours), day of week, day of month and month have already been considered as features. It is probable that the estimator accuracy might be increased if the aforementioned time factor is discovered.

It is also interesting to note that although rain was cited as an important factor by Vanderplas [13], rain is not considered important by the estimators used in this thesis.

One theory that was investigated, is that the dataset could be skewed in favor of observations without rain. This could cause decision trees (and by extension the ensembles) to be biased. When looking at the dataset for May-July 2014, it is clear that the dataset is definitely skewed: 911 observations with rain were recorded, while 17964 observations had no rain. Unfortunately, the rebalancing trick that is normally used to correct bias in decision trees cannot be used here as the training set would then become too

Feature name	Decision tree	Random forest	AdaBoost
Epoch	19.19	20.78	26.88
Time of day (hours)	14.06	13.2	13.05
Day of week	10.4	11.7	8.6
Day of month	6.35	2.65	3.4
Month	0.29	0.0	0.0
Station ID	10.4	11.4	6.9
Station latitude (deg)	12.7	13.5	9.5
Station longitude (deg)	7.7	9.9	7.4
Altitude	11.20	11.3	12.3
Temperature (C)	6.78	5.3	9.5
Hourly Precipitation (mm)	0.27	0.12	1.3
Cloud cover (okta)	0.0	0.0	0.0

Table 7.1: Feature importances for a trained decision tree, random forest and adaboost estimator. Each column lists the importance of the different features in terms of percentage.

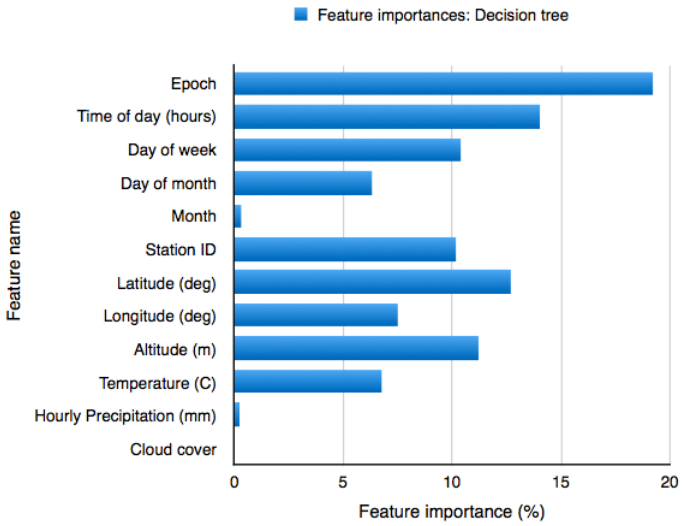


Figure 7.6: Feature importances considered by a individual decision tree

small for the estimators to train accurately. To put it into context, every single day of data contains 288 observations per station; taking only 911 rainy observations and 911 non-rainy observations is effectively the same thing as looking at six days of data (instead of 92 days).

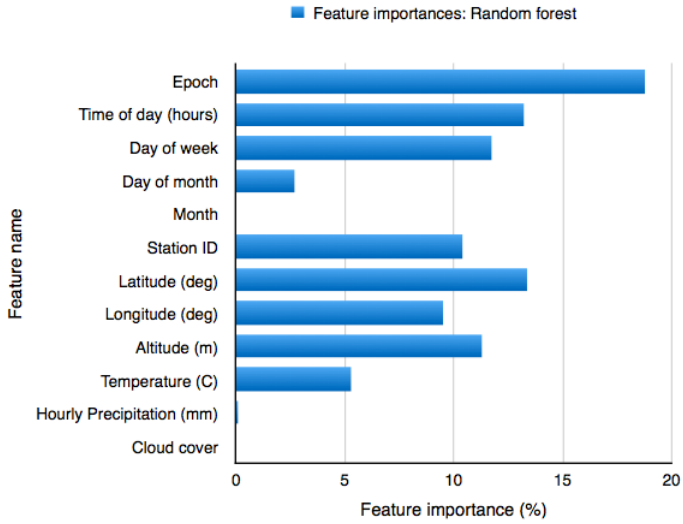


Figure 7.7: Feature importances considered by a random forest

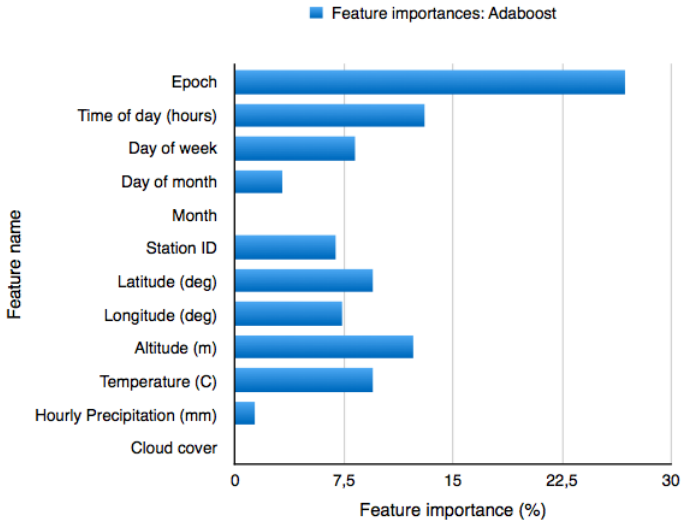


Figure 7.8: Feature importances considered by an AdaBoost algorithm

Chapter 8

Conclusion

This thesis was set out to explore the concept that bike-share traffic could be successfully analysed and predicted with machine learning patterns. The study also sought to compare some popular estimators to determine which was best suited for a prediction system. As a final step, the study determined the effect of climatological, geographical and time-based variables on the traffic flow.

8.1 Research process

The research for this thesis started by looking at the work of the DSSG team (section 2.1) and Jake VanderPlas (section 2.2). Vanderplas [13] ends the article by mentioning that ensemble learning might provide better results when predicting bike-share traffic.

To begin with, a data collection system (for both bike-share data and weather data from Washington D.C.) was put in place to ensure that the prediction system had enough data to work with. This was followed up by implementing a conversion system to provide a numerical representation that machine learning algorithms could work with.

After looking at Caruana [27], decision trees and ensemble methods using decision trees were investigated as possible candidates for an estimator. The error rates of these estimators were compared to each other and to the work of the DSSG team to see if there was any significant improvement (chapter 7).

Lastly, this section looks at how the prediction system developed in this thesis can be used as the back-end for a web based prediction system.

8.2 Findings

8.2.1 Model choices

Chapter 5 and 6 introduce Decision trees, Random Forests and AdaBoost as the estimators used in this thesis. It is clear from sections 7.1.2 and 7.1.3 that ensemble learners outperform individual decision trees, which in turn outperform logistic regressors (at least in the long term).

It is also apparent that the training-test ratio does not matter as much here, since 60% of three months (a little over a month) is enough for the models to stabilize their error rates.

It is also shown that using more than 10 estimators does not have a significant effect on the error rate for Random Forests, while AdaBoost has a "sweet spot" of sorts around 30-35 estimators. The recommendation would therefore be to use either a random forest or AdaBoost with decision trees as bike-share estimators.

Lastly, it is worth mentioning that the models in this thesis have not considered when, where and how many bikes were added/removed by bike-share operators as part of efforts to rebalance the system. If a feature that describes rebalancing effects is added to the data, it is possible that the error rate could decrease. However, this has not been done in the thesis as it was not possible to access the details of when/where the rebalancing was done by the bike-share operators.

8.2.2 Factors affecting bike-share traffic

Section 7.4 shows the different features that are considered important by the three estimator types used in this thesis.

The time of day being important (13-14% importance on average across all three models) is expected, as people in Washington do consider bike-share to be a valid commute option. The existance of a commute pattern is further confirmed by the Qt-based visualization module and how important the "day of week" was considered by the estimators (10.4 % importance).

It is also interesting to see that the station (location) parameters matter almost as much as the time-based ones (table 7.1). The claim put forth by the ITDP in section 3.1.2 that users often ride bikes from the top of a hill but seldom back again, was shown to have merit as all models conclude that the altitude of a bike-station has about 12% importance.

Unfortunately, no explanation was found as to why the epoch tag of every observation was considered so significant by all three models. It is however clear that there is some time-based factor at play here, which if found, would probably increase the accuracy of the models.

8.3 Future work

The software developed in this thesis provides part of the necessary back-end for a (web-based) prediction system for bike-share systems globally. The vision for such a product is outlined in the sections below.

8.3.1 Web prediction system

As the DSSG team states in their blog [34], bike-share operators are currently reacting to shortages and overflows that have already happened in their system. The software developed in this thesis provides long-term predictions, that can help bike-share operators be proactive in their operations. Figures 8.1 and 8.2 outline a wireframe design for such a system. Note the alerts shown in figure 8.2; these alerts are meant to help operators avoid overflows/shortages in the short-term.

Figure 8.1: The web prediction app showing the predicted traffic flow of bike-share station #59 in the Washington DC bike-share system. The time window of the prediction is set to two days but can be adjusted by using the spinbox next to the "Prediction dates" label.

For end users, the mobile interface proposed is farily simplistic (see figure 8.3). The user enters the time and chooses a station for which they want a prediction. This sends an API request to the web server, and returns a prediction as shown.

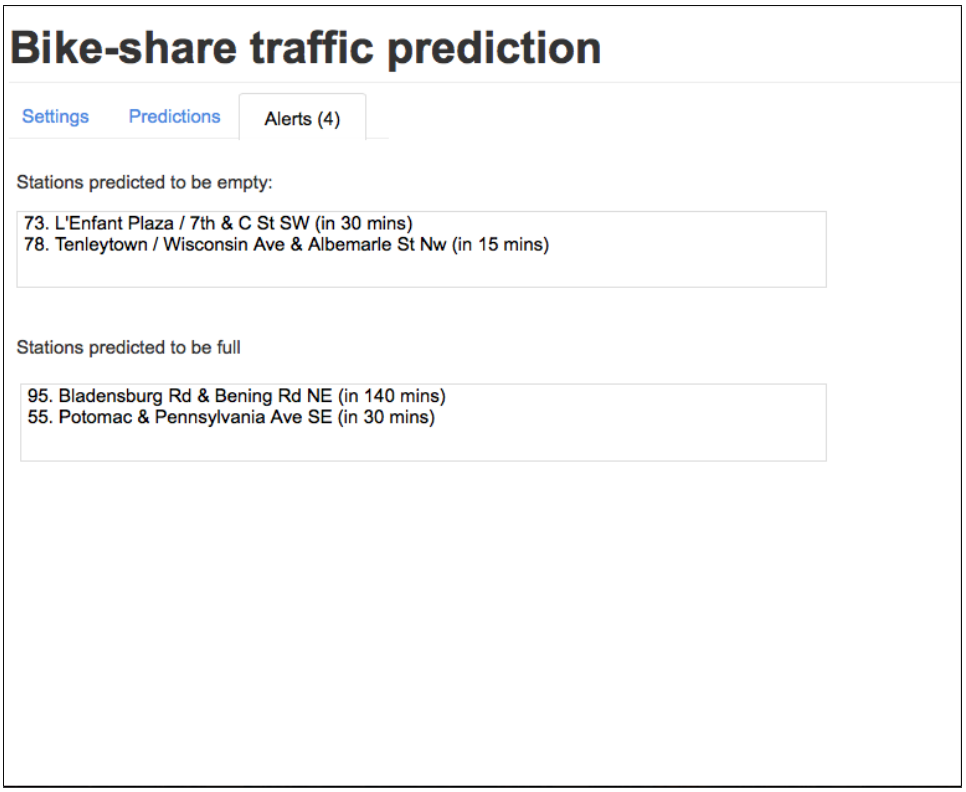
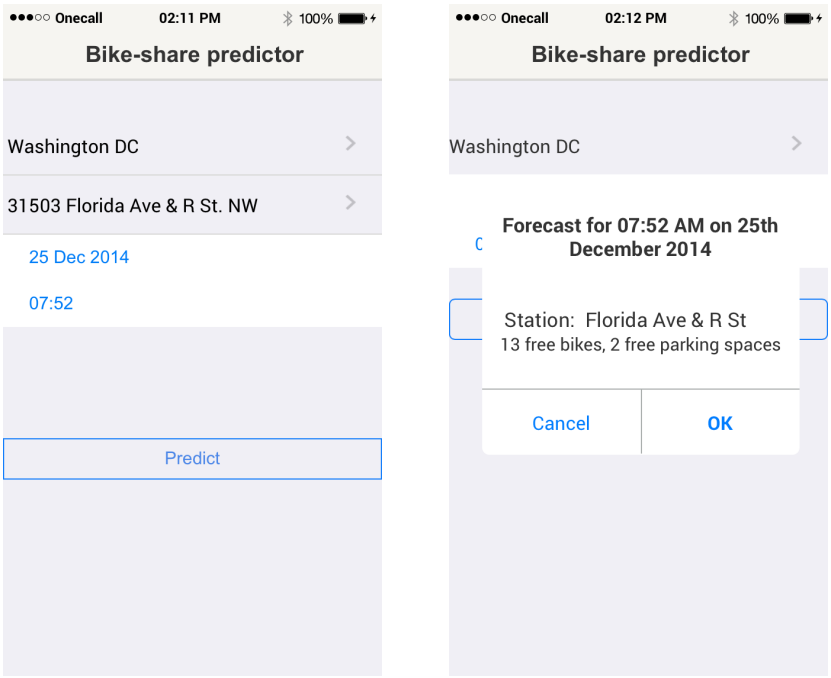


Figure 8.2: The system would generate alerts to warn the operators of shortages and overflows that occur in the next x hours (x can be adjusted in the "settings" pane)

8.3.2 Hadoop / Map reduce for bigger datasets

The data analysis software developed in this thesis is time-consuming to run; up to an hour of computation time required for three months of data. However, this is only on a desktop computer. This computation time can be lowered drastically by using parallel programming.

Random forests in scikit-learn are designed for use in clusters. To ensure parallel processing, one only needs to set the `n_jobs` variable to N where N is the number of processors available in a cluster. The desktops that were used in the analysis part of the thesis were only able to handle about 50 estimators before the memory constraints were breached. Using parallel programming would allow for a significantly faster process.



(a) App homescreen (b) Prediction

Figure 8.3: Mobile app for end users

Bibliography

[1] Institute for Transportation and Development Policy (ITDP). *ITDP Bikeshare Planning Guide*. URL: https://go.itdp.org/display/live/The+Bike-Share+Planning+Guide (visited on 19/06/2014).

[2] Olivier O'Brien and UCL Centre for advanced spacial analysis. *Bicycle sharing systems - Global trends in size*. URL: http://www.bartlett.ucl.ac.uk/casa/pdf/paper196.pdf (visited on 19/06/2014).

[3] *Barclays review by customer*. URL: http://www.tripadvisor.com/ShowUserReviews-g186338-d2151262-r174553920-Barclays_Cycle_Hire-London_England.html#REVIEWS.

[4] *Divvy: Helping Chicago's New Bike Share Find Its Balance*. URL: http://dssg.io/2013/08/09/divvy-helping-chicagos-new-bike-share.html (visited on 20/09/2014).

[5] Olivier O'Brien. *Bike Share Map*. URL: http://bikes.oobrien.com/global.php.

[6] *DSSG Analysis*. URL: https://github.com/dssg/bikeshare/wiki/Analysis (visited on 29/10/2014).

[7] 'Data Science for Social Good: Divvy project Methodology'. In: (). URL: https://github.com/dssg/bikeshare/wiki/Methodology (visited on 20/09/2014).

[8] John Aldrich. 'R.A. Fisher and the making of maximum likelihood 1912-1922'. In: *Statistical Science* 12.3 (Sept. 1997), pp. 162–176. DOI: 10.1214/ss/1030037906. URL: http://dx.doi.org/10.1214/ss/1030037906.

[9] Leo Breiman. 'Random Forests'. English. In: *Machine Learning* 45.1 (2001), pp. 5–32. ISSN: 0885-6125. DOI: 10.1023/A:1010933404324. URL: http://dx.doi.org/10.1023/A%3A1010933404324.

[10] Yoav Freund and Robert E. Schapire. *A Decision-Theoretic Generalization of on-Line Learning and an Application to Boosting*. 1997.

[11] Seattle Department of Transport. *Fremont Bridge Hourly Bicycle Counts by Month, October 2012 to present*. https://data.seattle.gov/Transportation/Fremont-Bridge-Hourly-Bicycle-Counts-by-Month-Octo/65db-xm6k. (Visited on 29/09/2014).

[12] Jake VanderPlas. *Personal website*. URL: http://www.astro.washington.edu/users/vanderplas/ (visited on 29/09/2014).

[13] JHake VanderPlas. *Is Seattle really seeing an Uptick in Cycling?* URL: https://jakevdp.github.io/blog/2014/06/10/is-seattle-really-seeing-an-uptick-in-cycling/ (visited on 29/09/2014).

[14] Seattle Bike Blog. *Fremont Bridge smashes bike count record + Bike use rises all over town*. URL: http://www.seattlebikeblog.com/2014/05/14/fremont-bridge-smashes-bike-count-record-for-real-this-time-bike-use-rises-all-over-town/ (visited on 29/09/2014).

[15] National Climatic Data Center. *Climate Data Online*. URL: http://www.ncdc.noaa.gov/cdo-web/search?datasetid=GHCND (visited on 29/09/2014).

[16] Pandas. *Pandas*. URL: http://pandas.pydata.org/ (visited on 29/09/2014).

[17] Scikit-learn. *Scikit-learn Linear Model: Linear Regression*. URL: http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html#sklearn.linear_model.LinearRegression (visited on 29/09/2014).

[18] *General concepts: epoch*. URL: http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_15.

[19] G.P Moore D.S. McCabe. *Introduction to the Practice of Statistics*. 6th edition. W. H. Freeman, 2007.

[20] S.E. Maxwell and H.D. Delaney. *Designing Experiments and Analyzing Data: A Model Comparison Perspective*. Avec CD v. 1. Lawrence Erlbaum Associates, 2004. ISBN: 9780805837186. URL: http://books.google.no/books?id=h-bMhmQMifsC.

[21] Alvin C. Rencher and Fu Ceayong Pun. 'Inflation of R2 in Best Subset Regression'. In: *Technometrics* 22.1 (1980), pp. 49–53. DOI: 10.1080/00401706.1980.10486100. eprint: http://www.tandfonline.com/doi/pdf/10.1080/00401706.1980.10486100. URL: http://www.tandfonline.com/doi/abs/10.1080/00401706.1980.10486100.

[22] F. Pedregosa et al. 'Scikit-learn: Machine Learning in Python'. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[23] Leo Breiman. 'Bagging Predictors Technical Report No. 421'. In: (1994). URL: http://statistics.berkeley.edu/sites/default/files/tech-reports/421.pdf (visited on 13/10/2014).

[24] Robert E. Schapire Yoav Freund. 'A short introduction to boosting'. In: *Journal of Japanese Society for Artificial Intelligence* (1999). URL: http://www.yorku.ca/gisweb/eats4400/boost.pdf (visited on 13/10/2014).

[25] Eric W. Weisstein. *Ball Picking*. URL: http://mathworld.wolfram.com/BallPicking.html.

[26] Leo Breiman. 'Random Forests'. English. In: *Machine Learning* 45.1 (2001), pp. 5–32. ISSN: 0885-6125. DOI: 10.1023/A:1010933404324. URL: http://dx.doi.org/10.1023/A%3A1010933404324.

[27] Rich Caruana and Alexandru Niculescu-Mizil. 'An Empirical Comparison of Supervised Learning Algorithms'. In: (). URL: http://www.cs.cornell.edu/~caruana/ctp/ct.papers/caruana.icml06.pdf (visited on 13/10/2014).

[28] Tin Kam Ho. 'Random decision forests'. In: (1995). URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=598994 (visited on 13/10/2014).

[29] 'Shape Quantization and Recognition with Randomized Trees'. In: (). URL: http://www.cis.jhu.edu/publications/papers_in_database/GEMAN/shape.pdf (visited on 13/10/2014).

[30] Robert E. Schapire. 'The Strength of Weak Learnability'. In: (). URL: http://www.cs.princeton.edu/~schapire/papers/strengthofweak.pdf (visited on 13/10/2014).

[31] L. Weiss. *Introduction to Wald (1949) Statistical Decision Functions*. English. Ed. by Samuel Kotz and NormanL. Johnson. Springer Series in Statistics. Springer New York, 1992, pp. 335–341. ISBN: 978-0-387-94037-3. DOI: 10.1007/978-1-4612-0919-5_21. URL: http://dx.doi.org/10.1007/978-1-4612-0919-5_21.

[32] Harris Drucker. 'Improving Regressors Using Boosting Techniques'. In: *Proceedings of the Fourteenth International Conference on Machine Learning*. ICML '97. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, pp. 107–115. ISBN: 1-55860-486-3. URL: http://dl.acm.org/citation.cfm?id=645526.657132.

[33] 'Random classification noise defeats all convex potential boosters'. In: *Machine Learning Journal 78* (2010). URL: http://www.phillong.info/publications/LS10_potential.pdf (visited on 13/10/2014).

[34] DSSG. *Helping Chicago's new bike-share*. URL: http://dssg.io/2013/08/09/divvy-helping-chicagos-new-bike-share.html (visited on 26/10/2014).