SOFTWARE ENGINEERING
Code Review

for

Disappster

Version 1.0

Adam Ryan (14395076)

COMP30830

April 22, 2021

Contents

1 Introduction

1.1 Purpose

As part of the COMP30830 module, we have to conduct a Code Review on Chicago Bike-Share. This document is to chart the results. This component comprises one question split into eight components.

2 Question 1

This chapter is designed to list the questions and answers for question one.

2.1 Question A

Give a high level overview of the application. Describe these components and how they work and how they are connected to each other. (10)

2.2 Answer A

At a high level, the codebase is split into the following folders:

- Analysis - This section contains various graphs which were produced in analysing the average available bikes over time in a variety of areas in PDF format. These results are left in a raw form without particular commentary or insight. It also contains the code which was used to produce the results. Broadly, it works by connecting to the database, producing a PDF, pulling the results into a dataframe indexed by date, grouping the dates into blocks of two minutes, assigning the minute of day (although it does this with a lambda and apply map which is not an efficient operation in pandas), taking descriptive stats, and for each station the plotting the time graphs followed by closing the PDF.

- API - This is used for allowing access to their data via API (e.g. by creating an API for the app). The utils.py file is used for validating certain aspects returned by app.py. App.py is the key driver. It allows acquisition of data by entering the route with city, year, month day. It validates the city that was entered and fetches the results (if invalid it prints a message and returns an empty JSON). It protects against SQL injection by typecasting the inputs into the SQL statement and passing them in using cur.execute(statement, variables). It zips the response then jsonifies it in passing it as a response.

- Data - Data is historical data which was acquired. Similar to our app, data from the scraper was only realtime so they got a data dump of historic data which we know from the analysis section was in two minute chunks. They clean_data which are shell scripts to convert TSV into CSV (although why this is needed is not clear as TSVs are valud CSV formats) The create_db file is then a series of

SQL commands which need to be run on the database to create the schema and populate the tables. They also have a models.py file which it seems is when they were experimenting with driving the app through Models (but don't seem to have an appropriate format in their codebase for deploying this).

  - There appears to be a weakness in this folder in particular. They've expanded the codebase from Python and SQL commands called in Python, to Python, shell scripts that need to be run externally, and SQL scripts that need to be run directly on the database (why? This should be possible in Python).

- For Wiki - This contains a series of results which pertain to analysis that should be displayed in the Wiki. This folder is unnecessary and should be pushed into Analysis. They also have R code here which, again, expands the codebase and maintainability requirement to produce these results. This should be in Python and ideally a module in the Analysis section.
  - The R code itself is not particularly strong. They have hardcoded values put in as results which were outputs from the poisson_web model. It looks like they were using R to leverage ggplot2, however a ggplot style can be used in matplotlib making this redundant.
  - Actions: This folder should be integrated into Analysis, the R code should be removed and replaced. The values should not be hardcoded and the result should be produced by calling the model function for the output (or functionising a method to get the output).

- Model - The folder contains one file relating to extracting data. It contains methods for pulling data into a dataframe (station and availability), and a method to rebalance data for a given station and timeframe. Oddly these files read data from a CSV but this should be in a database table. The final method is about turning setting up the model into a poisson process by looking at the departure and arrival times of bikes to identify how long their journey took. It adds on some time components which are going to be used in the model. One odd aspect is for the appending of the weekday flag they first copy the day of week data and then do conditions on that copy, whereas this could be completed in one step using loc. They have a fit file which imports the extractor, and their fit poisson function calls these methods. They set up data matrices for arrivals and departures and fit a poisson model based on the predicted number of departures and arrivals in a time period. Interestingly, this approach is different to our model but likely an XGBoost Poisson model could be an interesting way. Essentially their model works by identifying how many numbers are in a station now. They calculate what the net movement will be within an hour, and add the result to the current station (in the interest of space I'm not going to detail all the workings of this process). They then run the poisson model over each station id and pickle the resultant model and develop a function to load the model. They simulate a series of tests for certain starting conditions, and build in boundary conditions into the model (if over max then max, elseif under 0 then 0). Finally, they run the simulation for

model assessment and capture hits of the boundary condition. Finally, they have a validation method which produces descriptive stats on the model's performance (particularly looking at station 17) over a number of different conditions. Based on their commit history I can see this was not the original approach for the model and previously it had incorporated weather info and used GLM. A binomial model had also been tested in the app development.

- Scrapers - They contain two versions of scrapers for working with different versions of the API. Looking at their most recent V2 version, the database_scraper file retrieves from a URL and City passed as an argument, they have a working file presumably used to test (should be removed), a shell script for running the scraper with command line arguments, and a crontab example to demo how it runs continuously.

- Web - This contains the bulk of the web app. Static is split into CSS/static geography JSON data, images, Javascript, and their model file. Templates is split into a base template dictating the layout of the overall web page, a table template, and a time slider template. They use template inheritance to avoid repeating code which is a good practice. They have a copy of their poisson_web file for generating models continuously building on the static model generation, and two files relating to caching the results and rerunning on regular intervals. Finally, they have app.py which runs the app and holds the routes.

- Other - They have extensively created Read Mes and a Github Wiki which is quite nice for documentation via Github for a public project.

2.3 Question B

Comment on the structure and organisation of the github repository. For example, is it well organised? Are the instructions clear for getting the app up and running? Are the dependencies clearly described?. (10)

2.4 Answer B

I've largely touched on this in 2.2 so please refer to that for much of how the application works. In general, the application is pretty well structured. The instructions and documentation is largely speaking, well written, however because the application is split into a variety of components, across multiple codebases, in separate folders which are required for app.py to run, I think there should be an overall series of steps on how to install the application and set it up. While this becomes clear in reading through each of the components as described above, a singular location detailing these steps would make it more accessible for people overall. While there is a requirements file which covers some of the Python modules, it does not touch on aspects like the shell scripts which need to be run, the database setup which needs to be completed in PostgreSQL directly on the database, and the model validation component. I think a series of clear steps and ordered running steps would make the instructions on how to run the application with all of the dependencies clearer for the user, because at present it's not really clear what the order should be without delving into the various folders and reading through the code in my opinion.

The segmentation of the folder structure is quite nice, but there is room for an Archive folder and archiving some of the components of the app which are not used (e.g. V1 and V2 scrapers, the For Wiki folder and Analysis folder). Some of this could be consolidated in a clearer way.

There is some redundancy in some of the code which is created (e.g. the get station function, and repeated use of connecting to DB without methodising this) and there is some room for splitting the modelling section into some additional methods as the model creation function captures non-singular processes and it would be good to make these into methods to compartmentalise and tidy the code.

The methodology for producing the model is well outlined in the Wiki in the methodology an analytics section. Using a Poisson model is a sensible approach for the model.

2.5 Question C

What you can infer about the development from the Github information (commits, stats, etc) (10)

2.6 Answer C

Seen branches were used in the development of the model; master, circle-markers, demo, initial-ui-work, mapbox-new-api-key,temp-demp, and vidhur-getting-ready-for-final. Based on the branch structure, it seems they followed a feature-branch process and merged into main.

The general timeline is:

- Hunter Owens creates the repository in July 2013.
- Over the next few days scrapers, database setup, historic import etc. is added.
- Around the 11th of July, a branch was created an it looks like this posed significant challenges from the team. A series of branches were created from this new branch and merged back in to it primarily relating to scraping data, cron scripts, and plotting data for initial exploration. This branch and main ran in parallel with Adam Fishman primarily committing to main while the rest of the team developed on the branch. A number of the commits at this stage are not clear ("too embarassing dont ask" and "waaah"). By 16th of July the model was deployed and a rough flask app created. Significant problems and a rather unwieldy DevOps process is present as revealed in the GitTree at this time. 2.1
- At the 30th of June the overall git tree is largely cleaner as the branches are pulled from master following a feature merge, however some poor DevOps processes are still apparent as branches were pulled from feature branches and some circular feature branches are evident. In this time period, updates primarily focus on testing and changing various aspects of the model (e.g. initial exploration of GLM model), front-end features, and the weather scraping.
- By the end of August, a series of bug fixes and features had been added, and the switch to Poisson Models was under way. iPython notebooks were removed and the folder structure had been slightly tidied. A series of documentation (readme files) had been completed and were undergoing revision.
- At the beginning of September, the Binomial models were removed and the project and underwent final cleanup and documentation over the remainder of December.

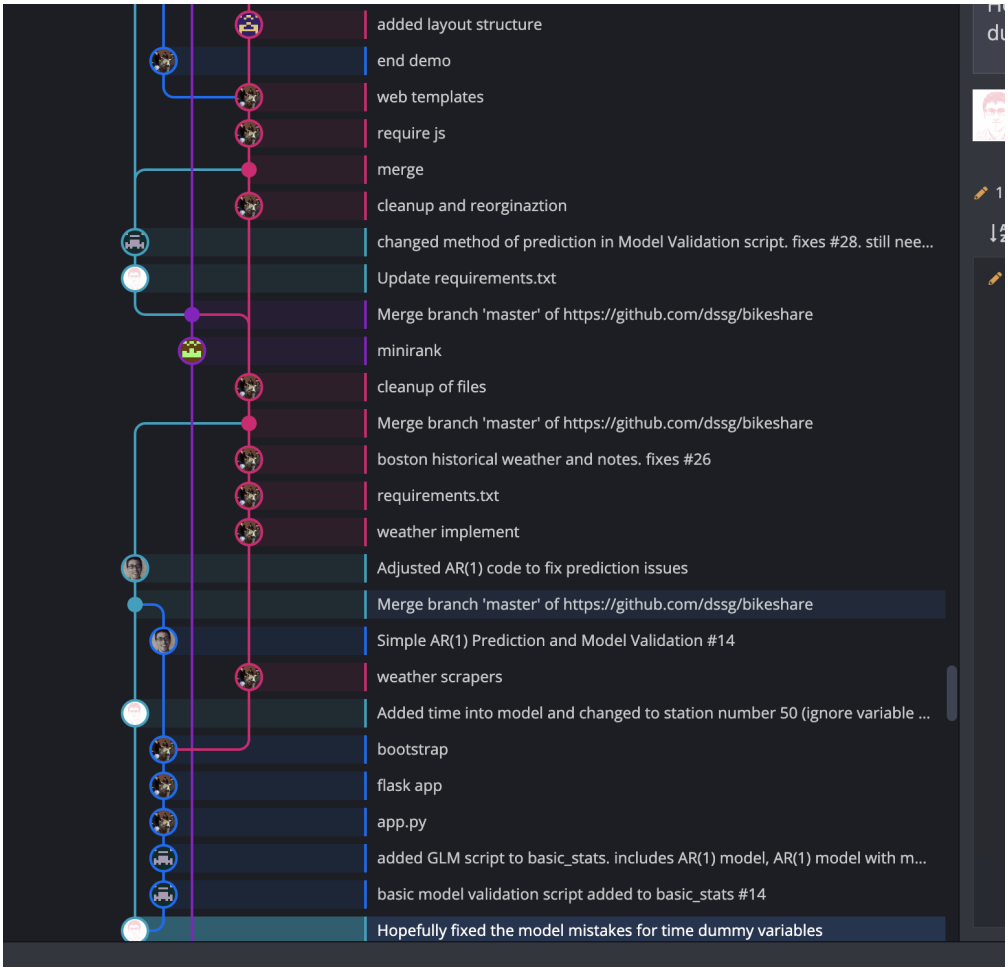

Figure 2.1: A rather unwieldy git branch structure

Based on the 6 open issues and 58 closed issues, it seems the project was primarily documented through Github (rather than via e.g. Jira and Confluence).

Based on the commit history, Hunter Owens and Juan Valez conducted a large proportion of the project. There are six key team members and they account for the bulk of commit activity. Breanam and Walter appear to have handled aspects of the Wiki construction and cleanup of the git folder structure Vidhur looks to have conducted quite a bit of work on some of the frontend features. Adam Fishman looked to have primarily been involed in elements of the model construction and analytics.

The overall key leads on the project appear to have been Hunter and Juan.

2.7 Question D

Comment on the scrapers (bikeshare/scrapers), mentioning their design, flexibility, robustness, etc. (10)

2.8 Answer D

The functioning is as outlined in 2.2. The bike scraper is primarily driven via commandine arguments where a city and URL are passed.

It's nice how the connection to the database is variable and can be set, and commandline calls are a nice addition, however I do not like how the scraper is not functionised. Similarly, the variable names within it are not very clear (j, J2) and the the main function cannot really be adapted quickly to other JSON requests as it would require updating aspects such as the INSERT INTO statements, and there is hardcoded validation of the cities with separate insert statements for each which means it overall is not very adaptable if a larger variety of cities need to be moved within scope.

The ForecastIO section is very nicely built, constructed using a class mechanism that's used for interacting with Forecast data. The one criticism I would have is that the rest of the app does not really conform to using an OOP style structure, and as such although this is likely a better approach I think it should be unified with the other scraper. I would personally recommend that instead of just having a Forecast class, they have a Scraper class which is then extended by Forecast and Weather. This would help in the abstraction and consistency of scraping data and stick with an OOP design for scrapers for consistency. I would likely change the tuples with city coordinates to a dictionary with keys of the city name and a sub-dictionary with the coordinate tuples and returned data (initially empty) as this would remove the need to manually list each city (which isn't maintable long-term) and the dictionary then looped over to access each forecast results, however for a fixed and limited number of cities this appraoch would suffice albeit there is room to enhance how automated and scalable it is. This is definitely a much better approach than what is taken with the bike data scraper and with some small changes could be made to scale.

2.9 Question E

Comment on the flask web application. Pay attention to the structure of the code, the routes, the templates etc. How does the flask web app handle database connections? How does it handle serving model predictions? How does it handle providing an API to the web frontend? You can compare and contrast with your own project, which implemented a similar design. (10)

2.10 Answer E

For the front end of the application, the app primarily uses template extension to build a base template in which the core of the application fits. Layout.html is used for most of the front end of the application, while tables extends layout to add in the data tables and template is used to add in features like the circles, mapbox and pie-charts. In customer_mapbox it looks like the API is listed for Juan's mapbox in "jpvelez.map-zuhdyp2h" however I am not entirely certain if this is the only API instance. The user-accessible API is configured in the API file and works broadly by providing a route with city and time as outlined in 2.2. The frontend looks to be largely written in Bootstrap and a lot of the variables are not particularly insightful with single letter variables common resulting in this aspect of the code being slightly challenging to read. A minimal number of routes are present primarily relating to displaying the key frontend pages and calling and loading the predictive models, loading them in using the cache.p file. The database parameters are generally loaded via environment variables, and within the flask app these are configured in the poisson_web file.

2.11 Question F

Comment on the predictive model. The python code is in Model. You should pay attention to the libraries used for the modeling, how is the model trained and evaluated, how is it stored for later use in the webapp?

2.12 Answer F

Following on from the operation of the model outlined in 2.2 where I've detailed the analysis steps and functioning of the model, the model is generated at regular intervals and stored into cache which is then loaded into the frontend presumably to allow for quicker predictions. The key component of the model which made it to production is that it is a Poisson Process which looks at the bike availability transactions to get a net movement rate for certain times and establishes boundary conditions. One aspect which I am not clear on is how the weather updates factor into the model. I can see via the Git commits that there was an original exploration of Binomial models and GLM models where significant analysis on weather was done and weather was incorporated into the model (see commit: 2bbdd5d93a0f3f2a2d1ca0a034cbee908e2ce5e9 for an example of their initial models which incorporated weather updates) however it looks like this aspect was removed and they instead chose to focus on the net movement based on time. It's not clear to me what drove this change in methodology, but the usage of a Poisson process seems rather sensible. The R scripts were updated to remove the weather analysis component and instead evaluate the Poisson model without factoring weather. Based on the removal, my assumption is that either the Binomial and GLM models were poorly performant in comparison to treating it as a Poisson Process, or weather was identified as not being a key factor in station availability. The model was evaluated by simulating various instances (simulation) and by plotting the outcomes using gpplot in the R script. The methodology is further outlined in their Github. This seems to be slightly weak unless these simulated points are known results. It looks like they've trained their model on the entire dataset (or at least I can't seem to identify a test/train split) to ascertain the accuracy using real data, but perhaps this is featured elsewhere which leaves a little bit of doubt to me as to how accurate the model actually is.

One element I think is probably better designed than our own models is the continuous refreshing and storing into cache.

2.13 Question G

Is all the code easily understood? Comment on the readability of the code. How well can you understand the function? Can you easily see the program/data flows? (10)

2.14 Answer G

I believe I can read some of the code well (particularly the backend functionality) however I think a lot of the data flow is quite obscure. While I know at a high level:
To briefly outline the high level data flow I can identify:

- Historic data is populated into specific tables.
- Metadata on stations is populated.
- Forecast and Bike data is regularly populated and inserted into relevant tables.
- Models run on recurring schedule to update the cache file which contains info on prodictions.
- Cache file is used to access the model data to increase optimisability(?)/responsiveness in web application.
- Poisson Model output is used to feed some of the analysis data (e.g. the R script). Documentation is prepared on exploratory and model analysis.

One aspect that I'm uncertain of is the usage of the weather scraper given the move to the Poisson model. I'm not sure if this is a historic feature or if it has a practical purpose in the application any more. Similarly, it would be useful to have a high level ETL diagram to help clearly outline the processing of data.

The frontend code is very tricky for me to read and the filenames are not always particularly clear. A lot of the code features one letter variables which is a very poor practice and makes it very difficult for me to parse. It being written using Bootstrap and jQuery also makes it tricky for me to fully identify how it functions at a more granular level as I'm not familiar with either.

One particular issue I have with the application is that the codebase is unnecessarily spread out across R scripts, SQL (to be run on the database), Python, and Perl (excluding the frontend functionality). Overall, this creates significant overhead in maintaining the code.

2.15 Question H

Examine this Git Commit: https://github.com/dssg/bikeshare/commit/9564d3b3a500196c6e787dd3bd21a0f8312e79f866933c00a0c0dd69912ae4

Prepare a short code review of this commit.

2.16 Answer H

- App.py - In the predict route, they've removed the commented code which was used for placeholder data, and changed the method from predicting for a single station to predicting for the complete array of stations. in predict_all, they've changed the default minutes value of 10 with 0, and instead of calling all of stations are calling the first 74 stations (although I'm not sure why this is hardcoded). They loop through each station for the prediction. Previously they only used three stations presumably for testing purposes.

- poisson_web - To accomodate the changes in what's used to call, first they've removed the try catch likely having identified the result should always be accurate (although I don't see why this was necessary and would suggest this should not be removed). They've also in the round change added in the metadata (station name, current bikes, and stands) to return this information as well.

- In customer_mapbox a change has been made to the API key. They've added error handling for a prediction error being returned and given behaviour. They've explicitly changed the circle colour which should be populated. They've made the call to Circle be a variable array rather than defining an array within the call for the coordinates, and what I believe to be the size to 100% instead of 200*percentage full (I am not certain as to why this change was necessary or why they wouldn't just state 100 instead of half of 200). They've also changed how the popup gets populated, removing some of the redundant string casts, set up a table like structure in how it gets called, and tidied the call to the feature array rather than using window properties.