Dublin Bikes App

COMP30830 - Software Engineering

Group - 18: Yingjie Niu, Haobo Liu, Sachin Soman

Dublin Bikes Deployment http://ec2-3-80-6-206.compute-1.amazonaws.com:5000

GitHub repository https://github.com/sachsom95/Dublin_bikes_official

---

1.Introduction **2**

1.1 Structure of Report 3

1.2 Group Information 3

1.2.1 Team Members 3

1.2.2 Individual Contributions 4

2.Overview **5**

2.1 Introduction 5

2.2 Objective 6

2.2 Target Audience 6

2.3 Audience Expectations 6

2.4 App Overview 8

2.5 Unique Selling Points 12

2.6 Features Implemented 15

3. Scrum Process And Project Management **16**

3.1 Git Workflow 16

3.2 Scrum Meetings and Stand-ups 17

3.3 Tools Used 18

3.4 Communication Method 19

3.5 Sprints 20

4.Architecture of app **20**

4.1 Technology used 21

4.1.1 Frontend: HTML, CSS, JavaScript, jQuery, Ajax, BootStrap 21

4.1.2 Backend: Python, Flask, Jinja2, MySQL 21

4.2 Issues faced and solutions 22

5. Analytics **23**

5.1 Machine learning model 23

5.2 Data preprocessing 24

5.3 Model training and evaluation 25

5.4 How to predict 27

5.5 Integrate model with frontend 27

5.6 Integrate model with backend (database) 30
5.7 Combine model with Flask 32
5.8 Issues and solutions 33
5.8.1 Investigate useful features 33
5.8.2 The response time is too long 33
4.8.3 The forecast information for the last day is not completely available 35
6.Future Work **36**
7 . Bugs **36**
8.Appendix **37**

1.Introduction

The purpose of the document is to provide a comprehensive report on the Comp30830 Software Engineering project 'Develop a web app for Dublin Bikes'. The report covers detailed information regarding Architecture, Design, Tech-stack, Front-end, back-end, Team organization, Management and scrum practices used.

The live deployment of 'Dublin Bikes App' can be accessed via the link provided below.
http://ec2-3-80-6-206.compute-1.amazonaws.com:5000

NOTE: Although we have provided full access to EC2 instances sometimes the link is not accessible via Eduroam or UCD wireless.

The full repository is hosted in Github as a private repository with Professor added as a collaborator. The reason for keeping the repository private is because of the API keys hosted in the repo. This will later be removed and configuration files will be added to store the sensitive information in the next stable release.

The Github repository of 'Dublin Bikes App' can be accessed via the link provided below.
https://github.com/sachsom95/Dublin_bikes_official

1.1 Structure of Report

The report will be divided into two parts. The first part of the report will cover the full technical workflow of the app. The next part is dedicated to describing how the development team worked towards the implementation of the project, especially the scrum practices and how it was modified and used for the task at hand.

1.2 Group Information

Group -18

The 'Dublin Bikes App' team consisted of 3 students of UCD CS Conversion. The group decided on a cyclic approach where each person would be a scrum master for two weeks. The team also decided that each member is familiar with the full tech-stack that will be used. After becoming familiar with the stack each member decided to work on different features of the project not based on personal preference but on the basis of efficiency. Each member will work on a feature and the other two members will review and approve it either via a pull request or by manually looking at the implemented code on the laptop.

1.2.1 Team Members

1. Sachin Soman - 19200494
2. Haobo Liu - 19201583
3. Yingjie Niu - 18209791

1.2.2 Individual Contributions

Each member of the team was fully aware of the full tech-stack and was involved in debugging and approval of each feature. With this regard inspecting the Github will also show a graph where each person at some point took responsibility for a sprint. Hence a peak for each contributor at different stages of the sprint. Some features like the UI needed more commits than features like the ML model hence the discrepancy in a number of commits between different contributors.

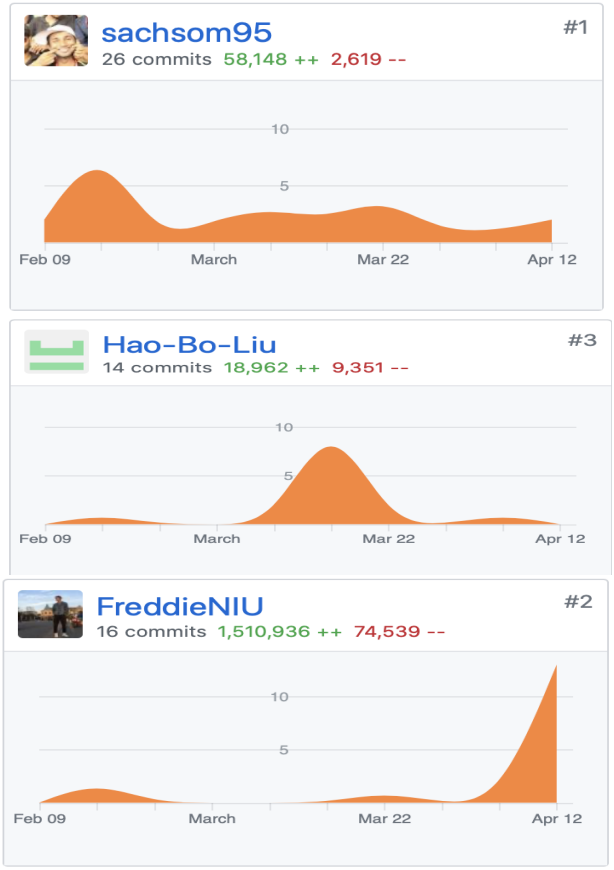


Fig 1

Hence based on unanimous voting we have decided that each member be given **equal contribution**

2.Overview

2.1 Introduction

The basic objective of the project is to create a web application that a user can use to find the closest available Dublin Bike and to find a station where the user can return the used bike. Apart from this, the app must be able to show the weather information on Dublin city so that the user can decide if it's suitable for the time to use a cycle.

2.2 Objective

The previous paragraph described the basic outline of the app. Other than that the App should fulfill the following objectives:

1. Display all available stations on Map
2. Display the station marker with different colors based on the number of available bikes
3. Clicking on station marker the website must generate information such:
    - Number of bikes available
    - Free slots to return bikes
    - Station address
    - Status of station
4. Display weather information for Dublin city
5. Find the closest station to get a bike based on user location
6. Find the closest station where the user can return the bike
7. Based on user inputs such as starting point, destination, date and time give a prediction of the number of bikes available in Starting station and number of stands available at the destination for the user to return the bike.
8. Generate visualization for a user to make easy interpretation of data allowing a user to make a decision fast without spending too much time on the app

2.2 Target Audience

Dublin in the last decade has started to emerge as a tech-hub. This has led to a huge influx of workers from around the world. Every day thousands of commuters flood the roads of Dublin early in the morning and late in the afternoon to use the public transport system. Lately, commuters have found out it's more beneficial to use cycles to commute through the busy rush hours of Dublin. And most people have started to take advantage of the facilities such as Dublin Bikes. Our target audience is the commuter population

that will be using the Dublin Bikes to commute on a day to day basis. These include students, workers going to offices near the city center and Tourists visiting the city. With around 102 stations and 1500 working bikes we can assume that at peak capacity we will have more than 2000 users using our app during the peak rush hours.

2.3 Audience Expectations

- The users must be able to locate a bike station with an available bike without spending too much time finding out details like station name and address. Looking at the map should give a fair understanding of all the bike stations with available bikes. This feature will be important for tourists or people who are not familiar with the city.
- The user must be able to find the shortest path to the nearest docking station so that they can return the bike before additional charges are incurred. A seasoned commuter must be able to look at the app to see how the weather is so that he may decide to use a bike instead of hopping on to a bus. Appropriate warnings such as rain must be shown to users from the app.
- People who are planning to go to the city may be in a few days for activities like picnic or shopping must be able to see if bikes will be available for use based on his/her expected travel plan, good visualization of the availability trend must be provided so users can modify his/her plans. These mentioned requirements are the basic expectations of the target audience

2.4 App Overview

The section covers the basic overview of the app. How it satisfies the objectives we set out to bring. How a user might use it and the thought process behind the design of the UI of the app based on our vision.
The first thing the user sees when he opens the website is a splash screen. The splash screen ensures that the website is fully loaded before the user enters the site. Another reason for the splash screen is that it makes the app look refined, smooth and gives an opportunity to show the brand name and greet the user.

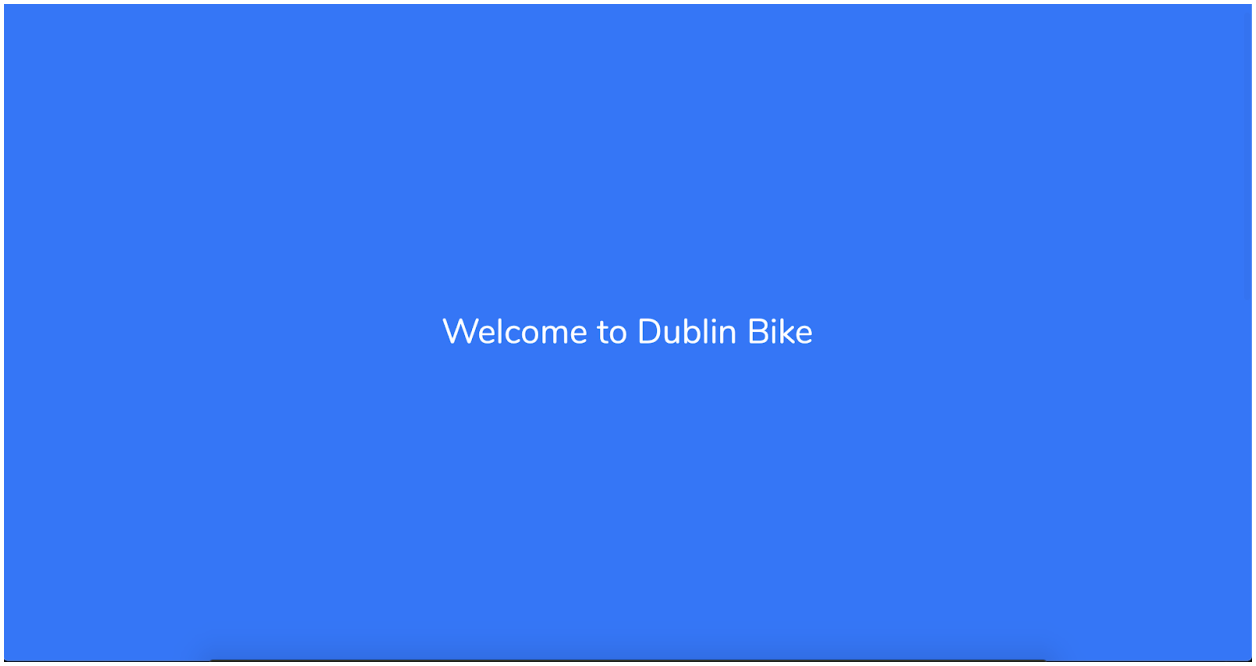


Fig 2 Splash screen

After all the DOM elements of the website are loaded. The screen transitions to the front page of the app; From the initial discussion of the project. We decided that the app must be viewed all on a single page. This is because the majority of users that use the app will be either walking or using public transport. Hence it will be difficult for them to go to different pages to view different data. Therefore all information and features of the app were made in a single page that doesn't need to be loaded more than once.

With the aim of providing maximum data to users on a single page, we looked for inspiration in various AngularJs and Bootstrap based websites. In one of the bootstraps documentation pages, we were able to find a category of web pages called admin and dashboard. This webpage was designed to provide maximum data for a user and we decided that we should also follow a similar approach to design our UI

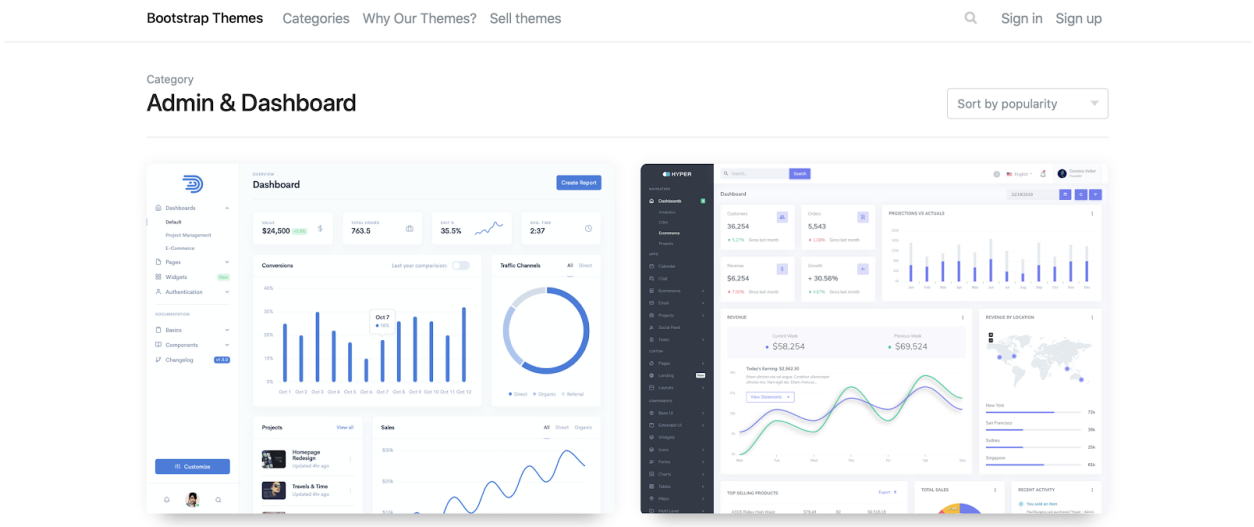


Fig 3 Dashboard based website

With all these observations taken into account we created a single page website with two parts.

The first part of the App as shown in Fig 4. will show the complete map with all Markers placed

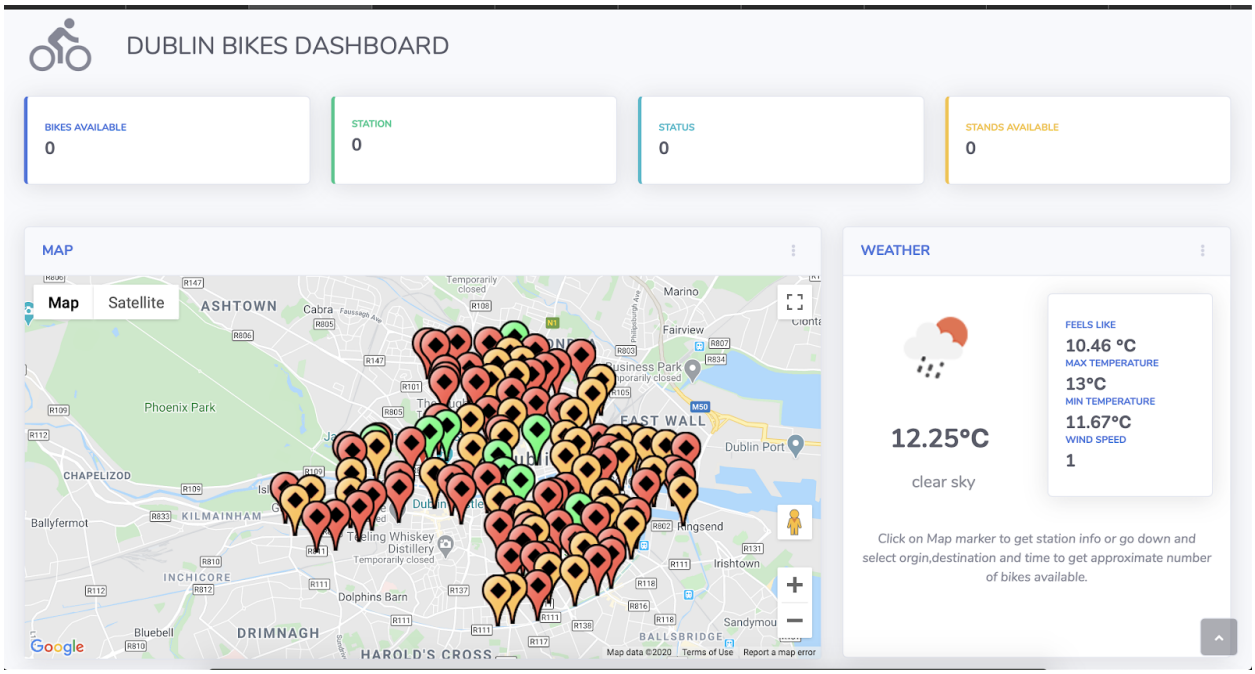


Fig 4 Section 1 of website

on a map indicating a station. The markers are color-coded based on the number of available bikes. A user can click on a bike station and get the most important features such as the number of bikes available on the top 4 cards. The right-hand side will show the most important weather information that a cyclist

needs. The weather app was designed from the bottom up. There was an option of using the widget directly from open weather, However, the widget UI was cluttered, not modifiable and not useful for a cyclist. We decided to make one ourselves specifically for the needs of a cyclist with dynamically changing weather icons and information in bold for a commuter to quickly get weather information. The implementation details will be explained in a later section.

This is the first section of the app and it's aimed at commuters who already know the city to quickly get a rough understanding of the bikes available in the city. Or for travelers who want to know how many stations are there and where they are situated.

The second part of the app is for people who want to find the closest station available to them with free bikes. Our initial plan was to use information from the HTML5 geolocation feature but later decided against it as it requires an SSL certificate in aws.

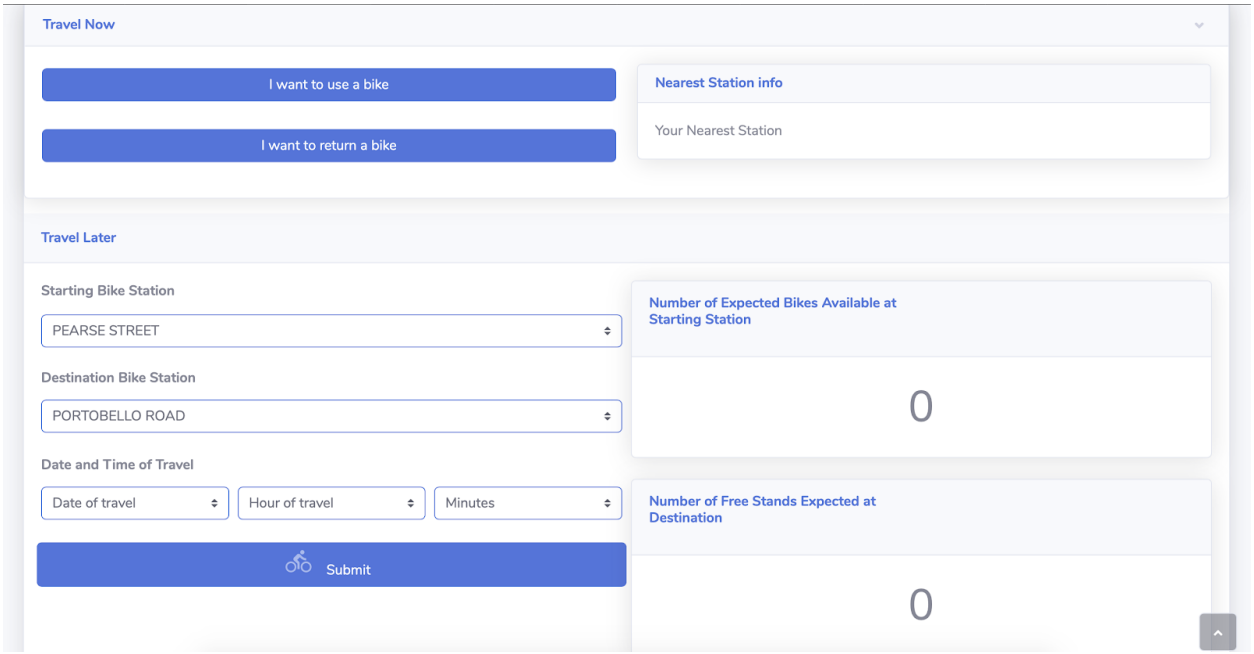


Fig 5 Section 2 of website

Instead, we let users press the ‘I want to use a bike’ or ‘I want to return a bike’ button which will create an icon on the map which we can drag. After dragging the icon to the desired position and if we press again the two options in the Travel Now card we get the closest station where we can get a bike or return a bike. This situation is not ideal however we will be able to implement a better version once we get a more reliable way of getting current location.

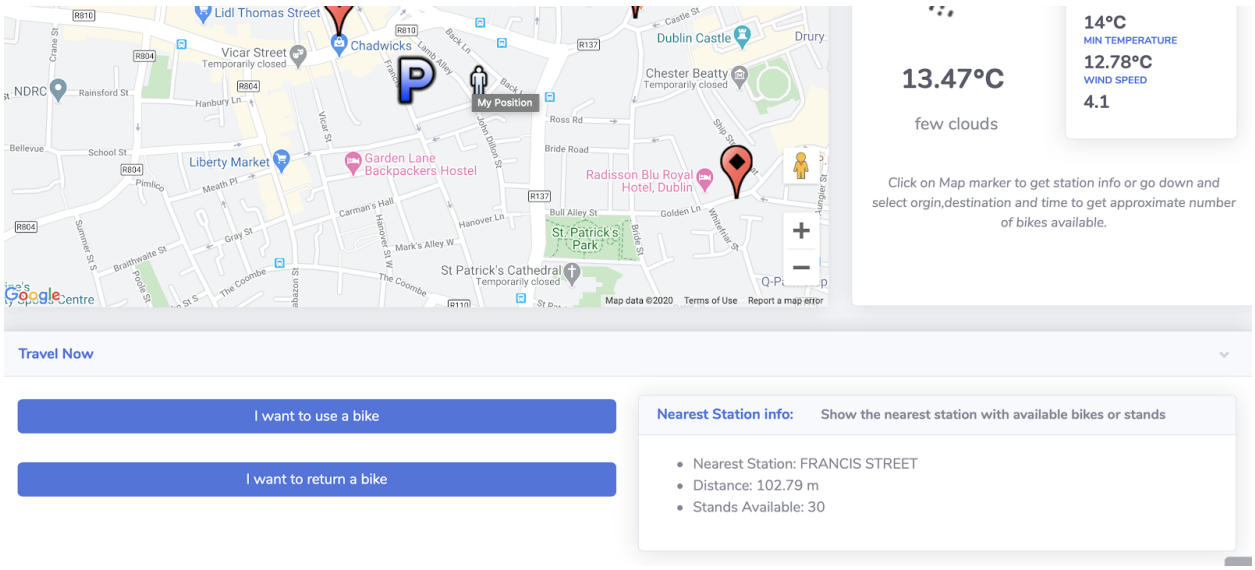


Fig 5 Travel Now card

The final part of the app is the most exciting feature. In this part, users can enter the start and end location along with the date and time of travel. Based on the input user provided. A machine learning model calculates and provides a prediction of available bikes at the start station and the number of free docks to return a bike at. Apart from these 4 charts are also generated which gives a clear picture of the bike availability over 24 hours and the next 7 days.

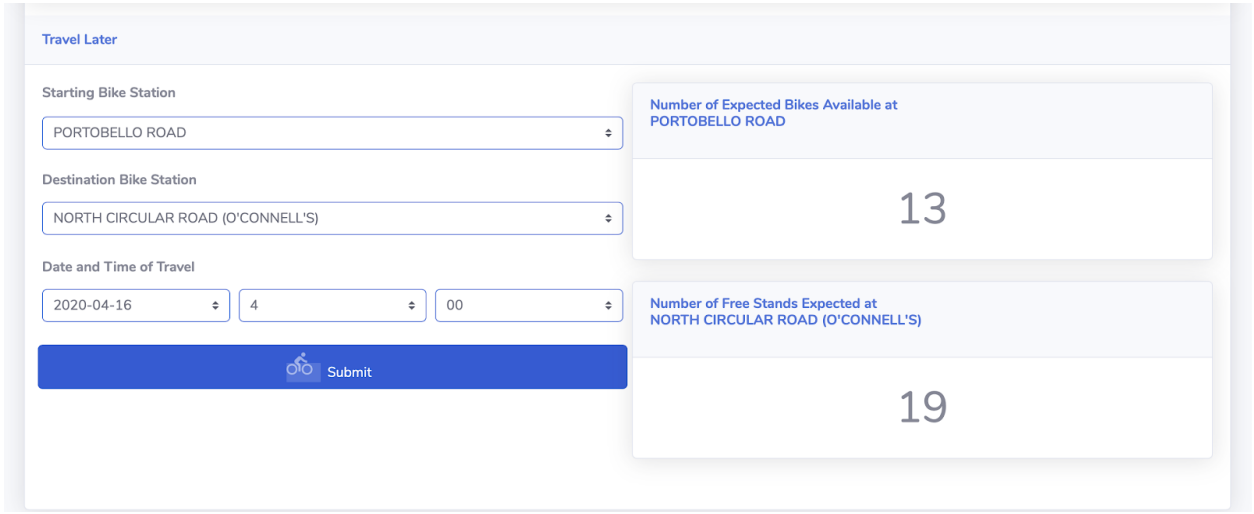


Fig 6 Predictions

The model churns out predictions in 3 to 4 seconds and loaders are implemented so that the user doesn't mistake the model calculation delays with unresponsiveness.

Fig 7 Data Visualizations

For the visualization, after multiple discussion we came to the conclusion we should have mainly 2 charts for both source and destination.

1. The first chart will show the distribution of bikes in a station for the next seven days in the morning.
2. The second chart will show the number of bikes available in the station for that full day.

These two data will give the user a general idea about the number of bikes they can expect in a station. The reason for employing both a bar graph and line plot is because we wanted to try them both and we couldn't decide which showed more info, therefore decided to keep both for now.

2.5 Unique Selling Points

This section will be based on a comparison with the official Dublin bikes website. The first main issue with official Dublin bikes is that it is a multi-page website. This requires the user to navigate multiple pages to get useful information. This is particularly tedious if the user is traveling on public transport or going through areas with poor network coverage to load pages constantly. The next major issue is the fact

that the website is not responsive. Given the nature of our app it's fair to assume most users will be viewing the website through a mobile device. Hence the official app is extremely cumbersome to use. The next major issue with the official app is that the data projected per unit area is extremely limited.

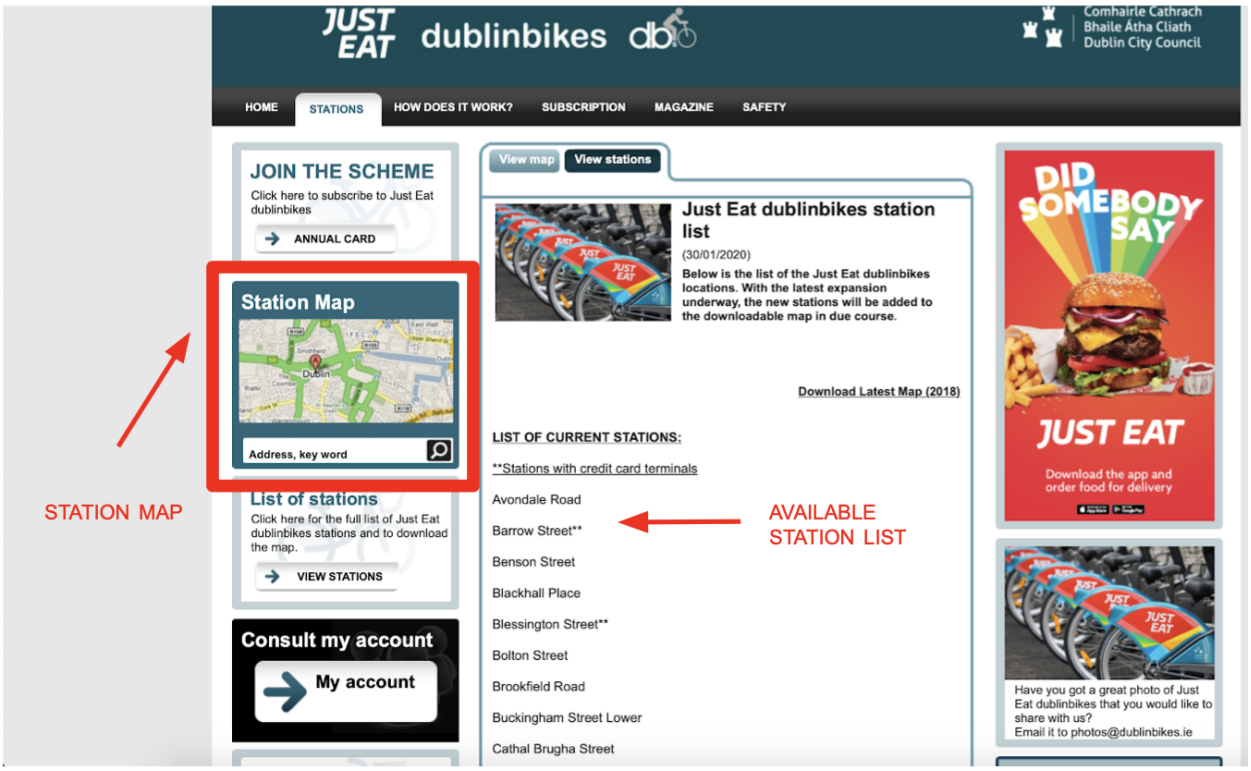


Fig 8 Official App

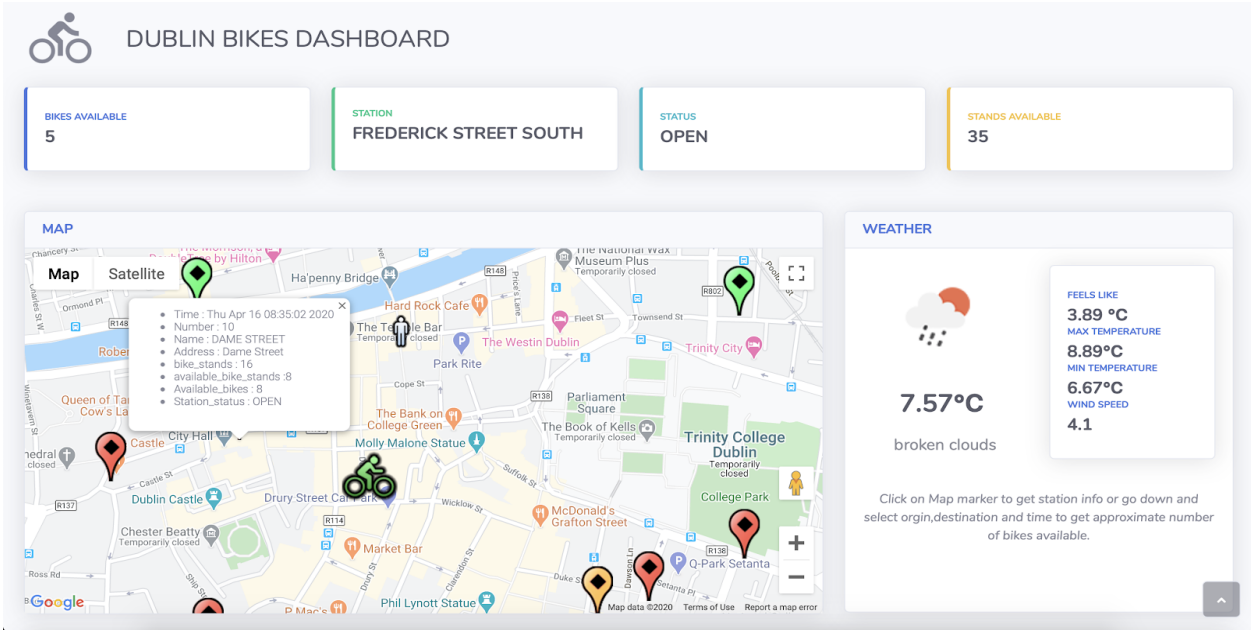


Fig 9 Our Product

Looking at the two figures(fig 8 and fig 9) side by side we can clearly see that the amount of information provided by the official bike app is extremely limited. The UI makes it difficult for users to navigate. To reinforce the fact, Looking at fig.8, available station list (marked in red) we see that the website just lists all available stations making it really difficult for users to use.

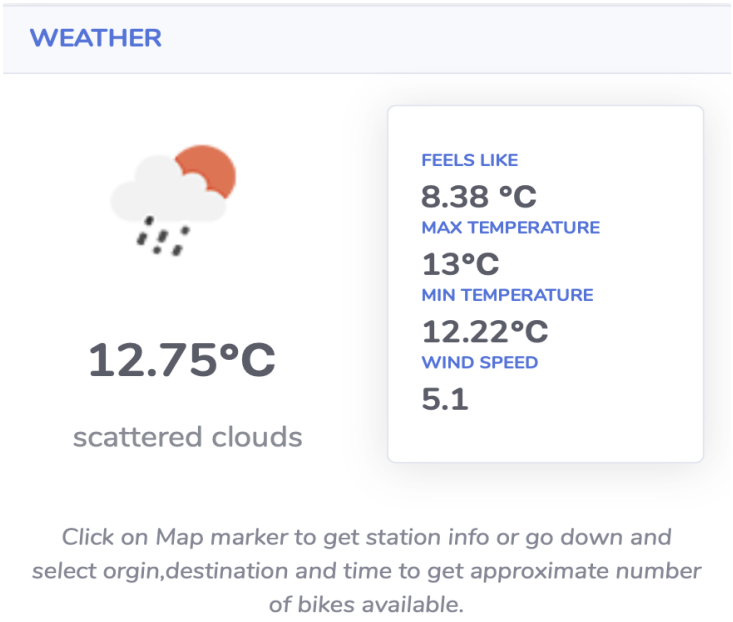


Fig 10 Weather Widget

Our App gives all essential details in a single page. The app is catered towards different types of users as explained in the target audience section with a high density of information essential for a cyclist. The user can look at stations through a map and click to get details or he can go through stations via dropdown arranged alphabetically to get a list of all stations. Apart from the basic functionality. The app has a weather widget catered towards the needs of cyclists which provides essential information in the shortest amount of time.

The next important feature is the ability to find the closest bike based on the location the user provides. The app can find the closest available bike or closest available dock where the user can return the bike. This is useful to return a bike to a station before the user is charged extra. Such features are not present in the official app.

The ML predictor is another feature implemented however it is not unique and is a requirement of the project therefore will be ignored in this section. To make things easier for recurring users local storage was implemented this means that next time the user loads the app the previous selections will be remembered and the user can simply click the submit button in the ‘Travel Later’ form.

In order for users to get some validation and positive reinforcement to use bikes we also added the carbon footprint message which will show the total distance traveled and the amount of Co2 saved by using the bikes when a user clicks the submit button in ‘Travel Later’ form.

2.6 Features Implemented

Feature	Status	Notes
Data collection via JCD API	Yes	
Data Management RDS	Yes	
Display Bike Data on Map	yes	Included colored marker for occupancy
Flask Application	Yes	
Weather Info	Yes	
Interactivity	Yes	
ML Model	Yes	
Deployed on EC2	Yes	
Responsive Design	Yes	Bootstrap saves the day
Dropdown of station	Yes	Has alphabetic ordering and remembers previous inputs
Data Visualization	Yes	Both line and Bar charts
Splash Screen	Yes	Finishes when full DOM is loaded
Loaders	Yes	Got from an open-source repository
Multi-color Markers	Yes	
Local Storage	Yes	
Geolocation feature	Yes	User needs to point on the map his location couldn't make the auto location work
Nearest station	Yes	User needs to point his the current location on the map
Cycling time	No	
Weather Warnings	Yes	User needs to submit a form for it to show.

Co2 Emission Info	Yes	User needs to press submit for it to show up.No separate form for it.

3. Scrum Process And Project Management

One of the main focuses of the project was to follow the scrum methodology in the development of features. The project was divided into 4 sprints where each sprint focused on some set of features and objectives. The sprints gave us a structure and enabled us to get a quantifiable measure of progress and based on it we were able to take corrective measures.

We decided that each member of the group should get the opportunity to do all roles in a team, with this in mind we had a new scrum master every 2 weeks. Another aspect we decided upon earlier was that everyone needs to learn all the tech-stack. The reason for this was because most tools and technologies were not familiar to any of us, therefore we didn't know who would be the best fit for implementing a feature. This meant in sprint 1 everyone worked on their own version of the app with each of us having our own version of RDS, EC2, scrapper and a rudimentary flask app. This brought in challenges as we didn't know which version of scrappers to choose or how to do version control. But we felt that this was necessary as we all became competent to work on the project.

3.1 Git Workflow

Once we were familiar with what we were supposed to do. We shifted next focused on making a git workflow. We realized that it was becoming difficult to keep track of changes. In Fact we lost the first version of our scrapper. Another issue was that each member at times used different libraries for the same purpose so we couldn't sync our work once it was done. Therefore, A git repository in GitHub was set up with two collaborators and a **branch git workflow** was soon adopted with each person working on a separate branch. If the feature is working other two teammates will review the pull request. If satisfied the feature branch is merged to the master branch. Though this was the plan it did not go as smoothly as expected because we were only getting familiar with version control. This will be apparent when the git graph is observed. However, most of the time we were able to stick to branch git workflow. There will be some branches where we tested the full app and we did not merge it to the master branch. By the end of sprint 1, the workflow was tested and employed.

3.2 Scrum Meetings and Stand-ups

After the first week of scrum daily stand-ups we found out that having a daily meeting was inefficient. Instead, we decided to have meetings when we had made progress or to discuss some bugs. Usually, we had an average of 3 to 4 meetings each sprint. However, we met every day in the library where we worked around 1 hour per day. Each meeting was logged in the Git wiki page (https://github.com/sachsom95/Dublin_bikes_official/wiki) with objectives, backlogs, scrum master details and also photos of whiteboard discussion. Git wiki was the perfect medium to log meetings as it was editable to everyone. An example screenshot of the wiki can be seen in fig 11 below.

Dublin Bikes Scrum meeting logs

Meeting :1

11 February 2020 Srum Master : Sachin , All present

Week 1 - Six weeks to go

1. Discussed an initial model for the project, We still dont know what is flask and what it can provide so that needs to be addressed
2.Decided to set up a common environment which will be pycharm
3.Made github pro for pvt repo
4.Learn Scrum
5.IN 2 DAYS WE PLAN ON MAKING A GITHUB PAGE WHICH OUTPUTS INFORMATION FROM JDCAUX
6.NEXT MEETUP ON THURSDAY

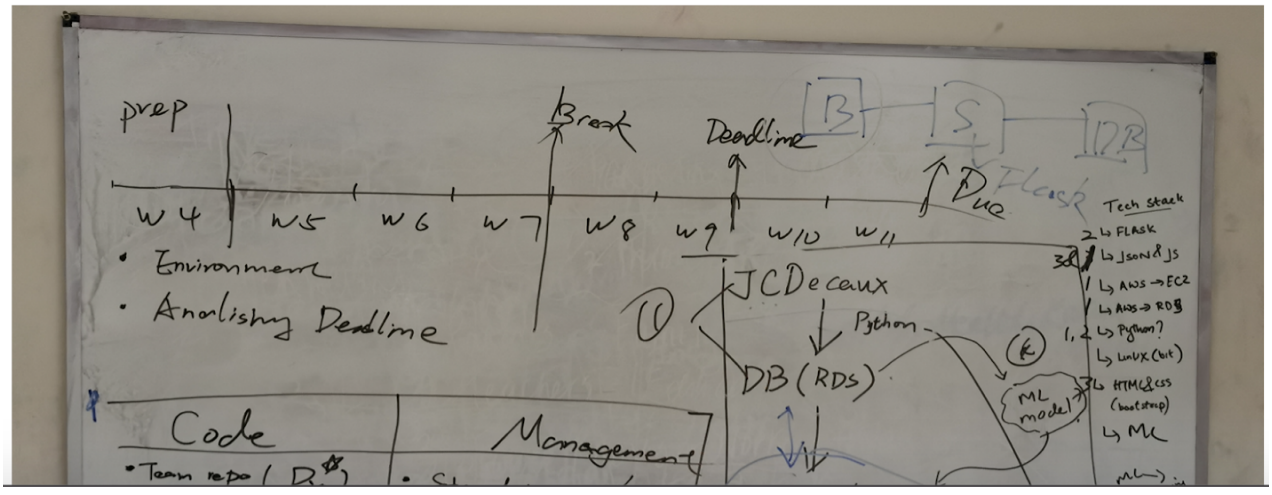


Fig. 11 A screenshot of git Wiki used for logs

A full log of the meetings cannot be provided in pdf format, therefore, to view the full-time line of meetings and retrospective do visit the wiki via the link.
The meeting system had to be changed after Ireland went into lockdown. The sprint meetings were now done during the practical hours of software engineering after the meet with the demonstrator.

3.3 Tools Used

A number of tools were suggested to document the scrum process. One such tool was Trello.
Trello is an online collaboration tool that can be used to plan and assign jobs to different team members to track progress and backlogs. We used Trello to keep a log of what others were doing and not to do the same job twice.

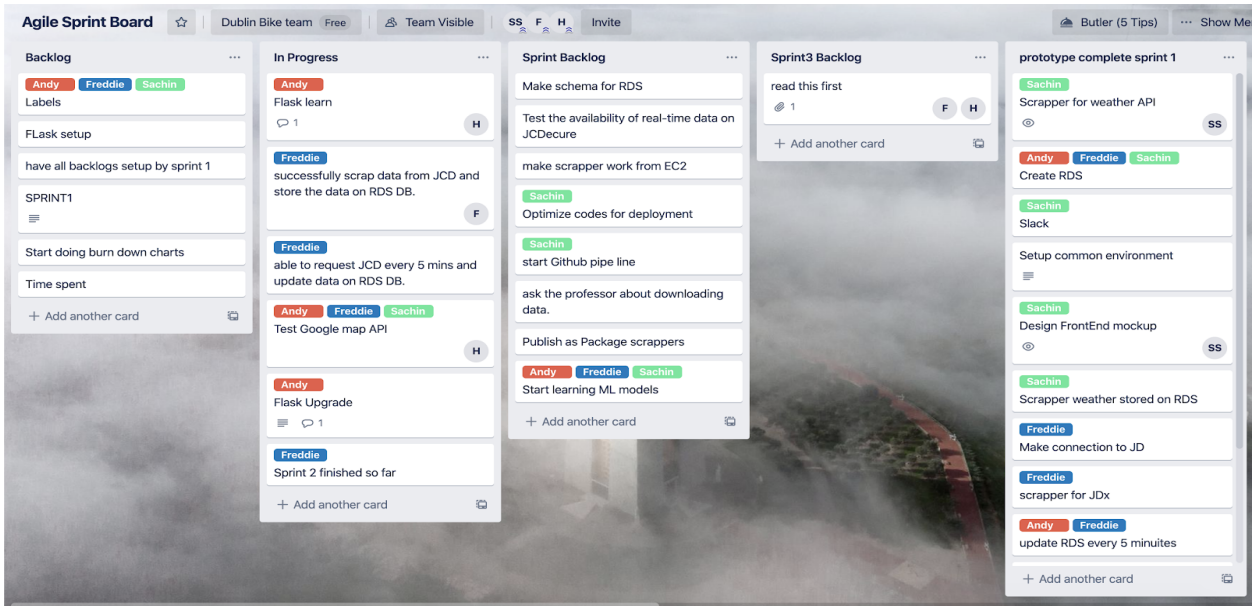
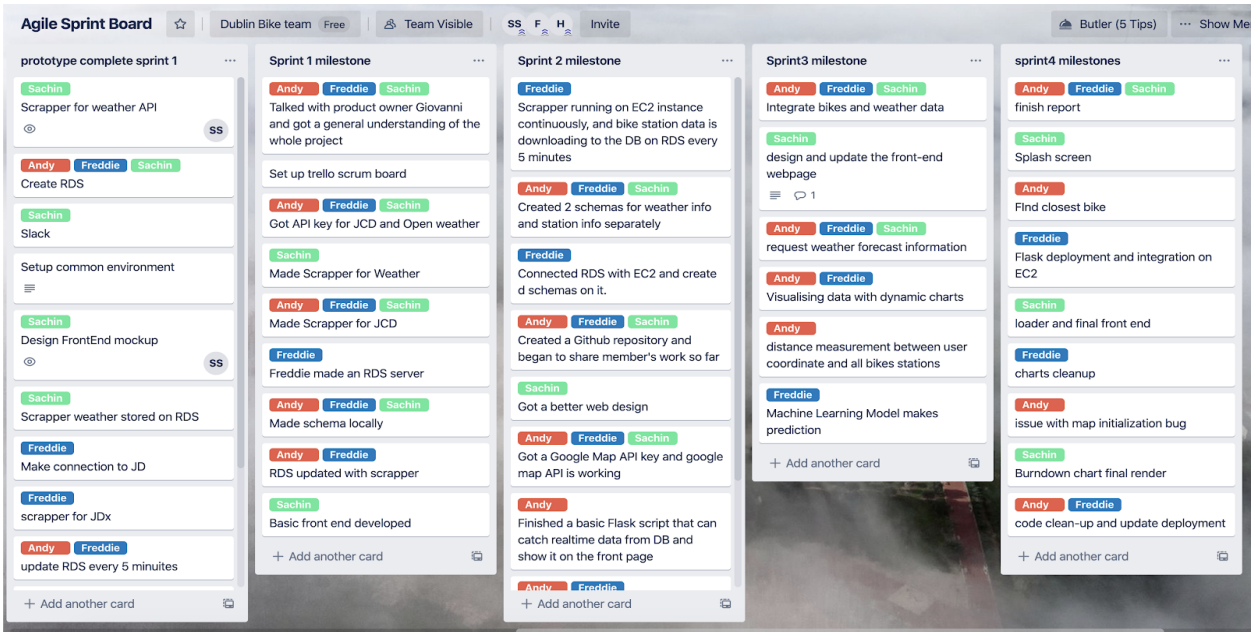

Agile Sprint Board
Dublin Bike team Free
Team Visible
Invite
Butler (5 Tips)
Backlog
Andy Freddie Sachin
Labels
FLask setup
have all backlogs setup by sprint 1
SPRINT1
Start doing burn down charts
Time spent
Add another card
In Progress
Andy
Flask learn
Freddie
successfully scrap data from JCD and store the data on RDS DB.
Freddie
able to request JCD every 5 mins and update data on RDS DB.
Andy Freddie Sachin
Test Google map API
Andy
Flask Upgrade
Freddie
Sprint 2 finished so far
Add another card
Sprint Backlog
Make schema for RDS
Test the availability of real-time data on JCDecure
make scrapper work from EC2
Sachin
Optimize codes for deployment
Sachin
start Github pipe line
ask the professor about downloading data.
Publish as Package scrappers
Andy Freddie Sachin
Start learning ML models
Add another card
Sprint3 Backlog
read this first
Add another card
prototype complete sprint 1
Sachin
Scrapper for weather API
Andy Freddie Sachin
Create RDS
Sachin
Slack
Setup common environment
Sachin
Design FrontEnd mockup
Sachin
Scrapper weather stored on RDS
Freddie
Make connection to JD
Freddie
scrapper for JDx
Andy Freddie
update RDS every 5 minuites
Add another card

Fig. 12


Agile Sprint Board
Dublin Bike team Free
Team Visible
Invite
Butler (5 Tips)
prototype complete sprint 1
Sachin
Scrapper for weather API
Andy Freddie Sachin
Create RDS
Sachin
Slack
Setup common environment
Sachin
Design FrontEnd mockup
Sachin
Scrapper weather stored on RDS
Freddie
Make connection to JD
Freddie
scrapper for JDx
Andy Freddie
update RDS every 5 minuites
Add another card
Sprint 1 milestone
Andy Freddie Sachin
Talked with product owner Giovanni and got a general understanding of the whole project
Set up trello scrum board
Andy Freddie Sachin
Got API key for JCD and Open weather
Sachin
Made Scrapper for Weather
Andy Freddie Sachin
Made Scrapper for JCD
Freddie
Freddie made an RDS server
Andy Freddie Sachin
Made schema locally
Andy Freddie
RDS updated with scrapper
Sachin
Basic front end developed
Add another card
Sprint 2 milestone
Freddie
Scrapper running on EC2 instance continuously, and bike station data is downloading to the DB on RDS every 5 minutes
Andy Freddie Sachin
Created 2 schemas for weather info and station info separately
Freddie
Connected RDS with EC2 and create d schemas on it.
Andy Freddie Sachin
Created a Github repository and began to share member's work so far
Sachin
Got a better web design
Andy Freddie Sachin
Got a Google Map API key and google map API is working
Andy
Finished a basic Flask script that can catch realtime data from DB and show it on the front page
Add another card
Sprint3 milestone
Andy Freddie Sachin
Integrate bikes and weather data
Sachin
design and update the front-end webpage
Andy Freddie Sachin
request weather forecast information
Andy Freddie
Visualising data with dynamic charts
Andy
distance measurement between user coordinate and all bikes stations
Freddie
Machine Learning Model makes prediction
Add another card
sprint4 milestones
Andy Freddie Sachin
finish report
Sachin
Splash screen
Andy
Find closest bike
Freddie
Flask deployment and integration on EC2
Sachin
loader and final front end
Freddie
charts cleanup
Andy
issue with map initialization bug
Sachin
Burndown chart final render
Andy Freddie
code clean-up and update deployment
Add another card

Toggle was another tool we tried. It measures the amount of time spent by each person on a particular task however we later decided against it as it didn't give any productive information.

3.4 Communication Method

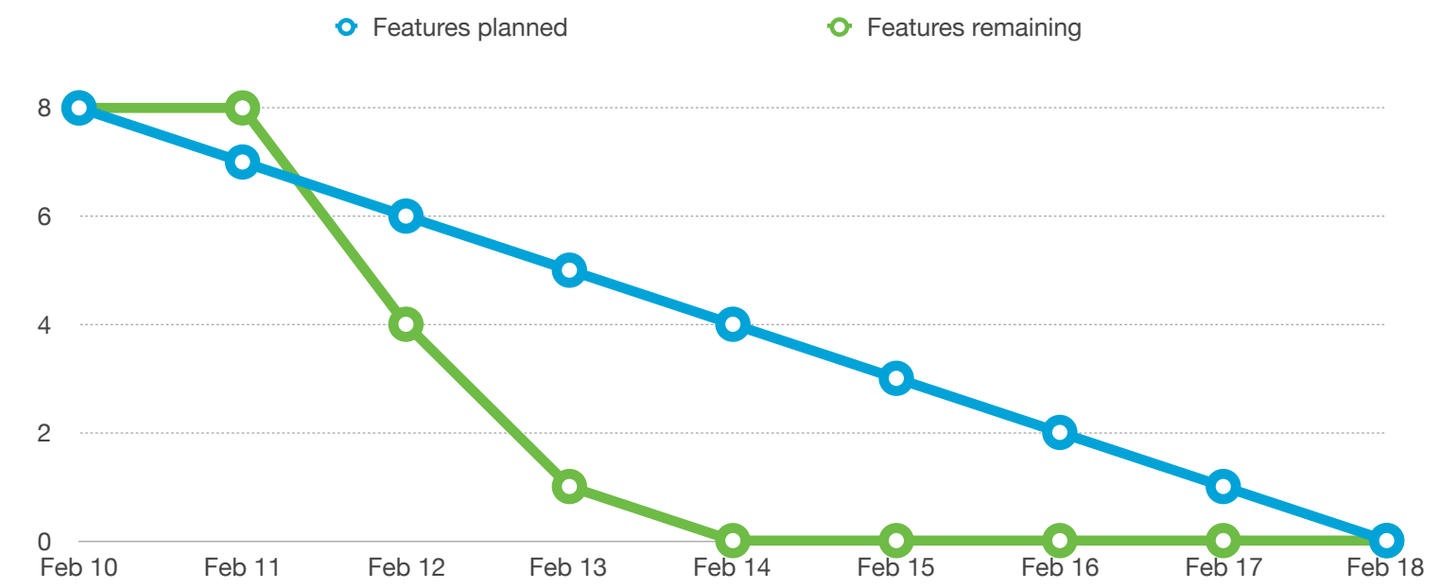
Initially, when the project started we had all discussions face to face. Although we set up a slack channel we rarely used it. After the lockdown started we needed to find a medium for communication. We decided to set up a WhatsApp group rather than a slack channel. The reason for this was Whatsapp was accessible through both PC and mobile devices easily. It enabled group call and sharing of media unlike slack which did not have features like group call and sharing of media was cumbersome in slack. A part of the communication in the group chat will be added in the appendix of the report.

3.5 Sprints

Sprints were used to keep track of the project. We used burndown charts, a way of quantifying the amount of progress we had and to know if we were on track. This section will explain each sprint with help of burndown charts.

SPRINT - 1

10 February to 24 February



NO OF FEATURES	EXPECTED TIME TO COMPLETE	ACTUAL TIME TO COMPLETE	MEETING DONE
8	14 days	7 days	3

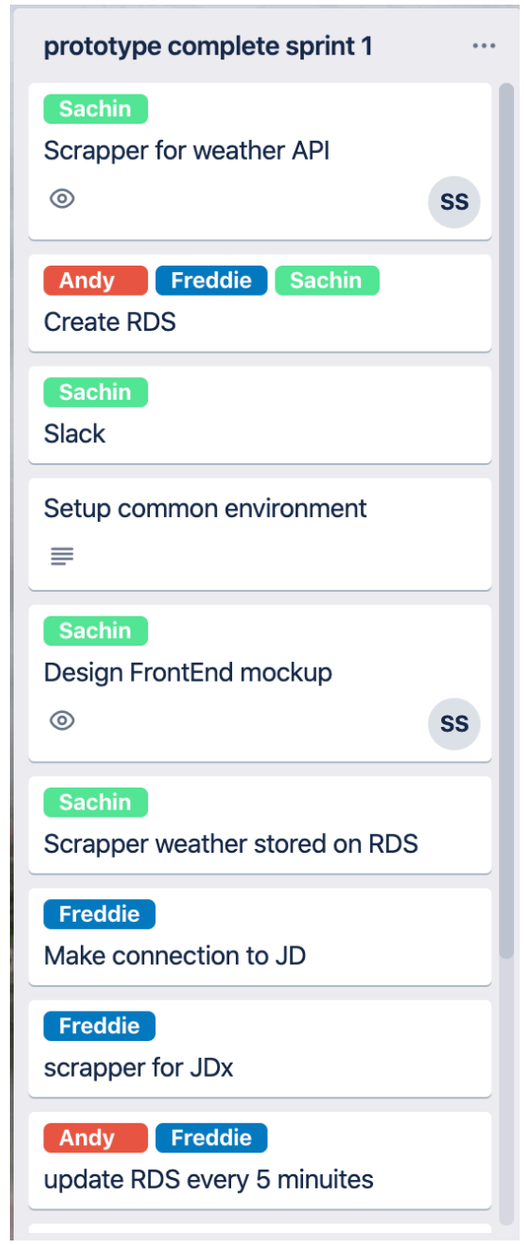
BACKLOGS AFTER SPRINT

0

DATE	FEATURES PLANNED	FEATURES REMAINING	NOTES
FEB 10	8	8	Planning and white board discussion
FEB 11	7	8	Setting up environment , common medium of communication, Git-workflow and test
FEB 12	6	4	Scrappers and many features done fast
FEB 13	5	1	API testing and queries done
FEB 14	4	0	RDS setup and working
FEB 15	3	0	N/A
FEB 16	2	0	N/A
FEB 17	1	0	N/A
FEB 18	0	0	N/A

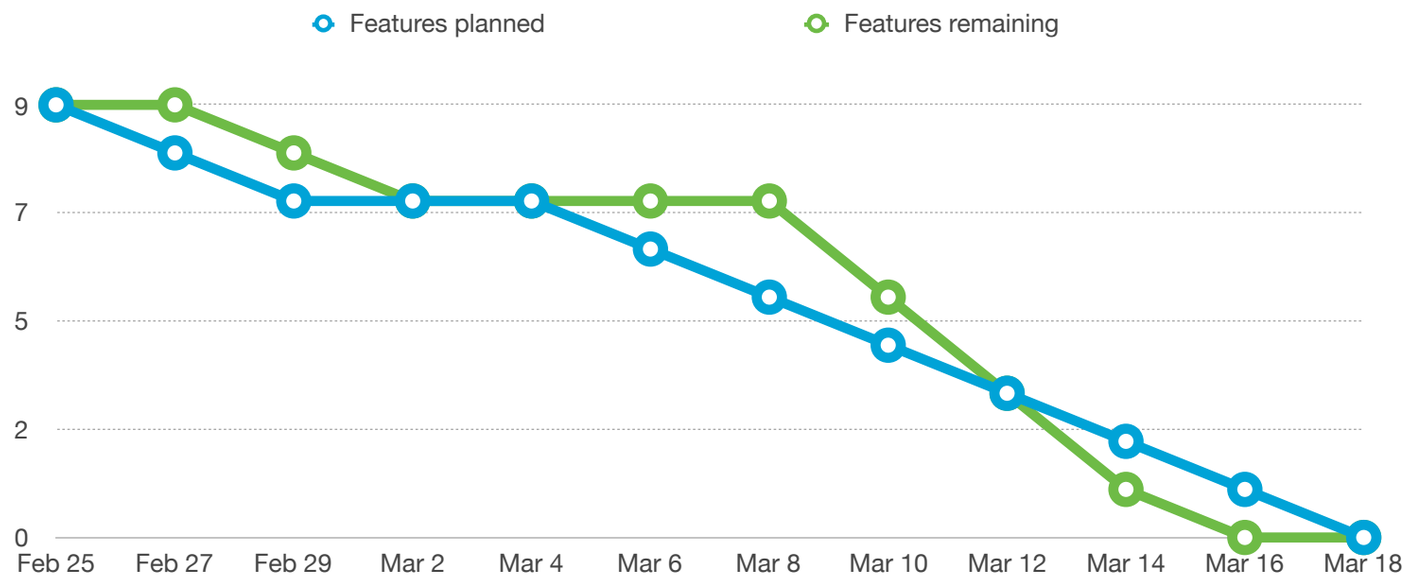
Notes

- The first sprint was done before the deadline.
- Initial planning and set up took two days after which development went smooth.
- Significant portion of stuff was completed in middle of week and was tested.
- RDS was set up.
- API keys and all queries was tested using POSTMAN
- Product owner verified work.
- Github was set up and running



SPRINT - 2

25 February to 18 March includes Spring Break



NO OF FEATURES	EXPECTED TIME TO COMPLETE	ACTUAL TIME TO COMPLETE	MEETING DONE
9	22 days	20 days	3

BACKLOGS AFTER SPRINT

0

DATE	FEATURES PLANNED	FEATURES REMAINING	NOTES
FEB 25	9	9	Unable to make scheduler work
FEB 27	8	9	Unable to make scheduler work tried infinite loop
FEB 29	7	8	Made better RDS schema
MAR 2	7	7	Mock-up website first version made
MAR 4	7	7	Looked at Crone tab to make scripts work
MAR 6	6	7	Cronetab scheduling worked.
MAR 8	5	7	No progress
MAR 10	4	5	MAP api works. Tried map on website it works. Learn Flask
MAR 12	3	3	Made first flask app with templating
MAR 14	2	1	Deployed app in EC2. Database and scrapper working
MAR 16	1	0	Markers working in app in EC2
MAR 18	0	0	Done with sprint 2

Notes

- Initially struggled with making scrapper run continuously, caused significant delay due to other subject work.
- Crone tab finally solved problem. Significant progress after that.
- Completed first UI based website based on mockup.
- Got the google map API.
- Put map in website.
- Flask app with marker made
- Tried first deployment with just map and markers.
- Databases and scrappers working well.

Sprint 2 milestone

Freddie
Scrapper running on EC2 instance continuously, and bike station data is downloading to the DB on RDS every 5 minutes

Andy Freddie Sachin
Created 2 schemas for weather info and station info separately

Freddie
Connected RDS with EC2 and create d schemas on it.

Andy Freddie Sachin
Created a Github repository and began to share member's work so far

Sachin
Got a better web design

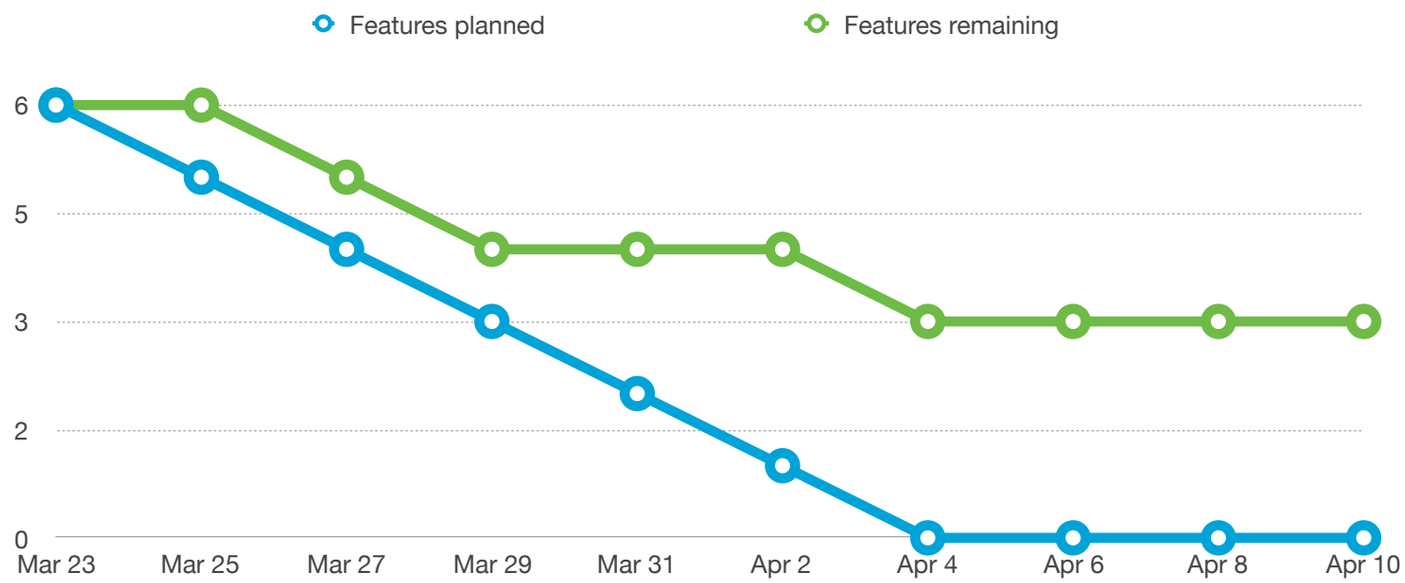
Andy Freddie Sachin
Got a Google Map API key and google map API is working

Andy
Finished a basic Flask script that can catch realtime data from DB and show it on the front page

Andy Freddie

SPRINT - 3

23 March to 5 April Post Covid -19 Sprint



NO OF FEATURES	EXPECTED TIME TO COMPLETE	ACTUAL TIME TO COMPLETE	MEETING DONE
6	12 days	16 days	2

BACKLOGS AFTER SPRINT

3

DATE	FEATURES PLANNED	FEATURES REMAINING	NOTES
MAR 23	6	6	Learn ML
MAR 25	5	6	Learn Visualization
MAR 27	4	5	Train a Random Forest
MAR 29	3	4	Cross-validation done
MAR 31	2	4	Improved UI fully responsive for all devices
APR 2	1	4	No progress shifted to online meetings and adjusting to changes.
APR 4	0	3	No progress
APR 6	0	3	Made predictions work
APR 8	0	3	No progress on visualization
APR 10	0	3	No progress on visualization

Notes

- Significant backlogs due to adjusting to new covid-19 lockdown.
- More time spent on learning best model to use
- A mistake was found in Database and important parameter not included. Luckily we had another db which was also running therefore got data from there.
- Random forest model trained with cross validation.
- Backlog of deploying the model and visualisation on EC2

Sprint3 milestone

Andy Freddie Sachin
Integrate bikes and weather data

Sachin
design and update the front-end webpage
1

Andy Freddie Sachin
request weather forecast information

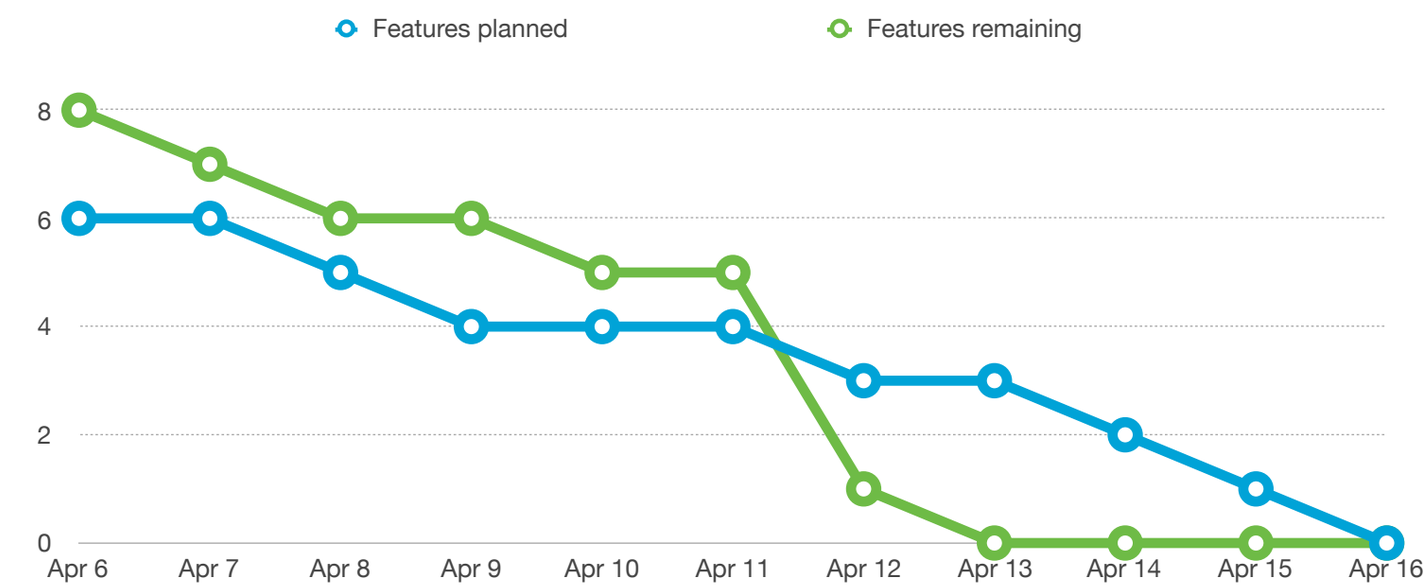
Andy Freddie
Visualising data with dynamic charts

Andy
distance measurement between user coordinate and all bikes stations

Freddie
Machine Learning Model makes prediction

SPRINT - 4

6 April to 18 April



NO OF FEATURES	EXPECTED TIME TO COMPLETE	ACTUAL TIME TO COMPLETE	MEETING DONE
6	12 days	12 days	5

BACKLOGS AFTER SPRINT

0

DATE	FEATURES PLANNED	FEATURES REMAINING	NOTES
APR 6	6	8	BACKLOG: Visualisation done on website
APR 7	6	7	BACKLOG: Web deployment of model done
APR 8	5	6	Final version of website completed
APR 9	4	6	Testing on EC2. Made it load faster
APR 10	4	5	Removed error where map wont load on refresh
APR 11	4	5	Loader done, splash done, visualisation done, major parts completed
APR 12	3	1	Find closest bike station left
APR 13	3	0	Testing
APR 14	2	0	Testing
APR 15	1	0	Cleaning code
APR 16	0	0	Github clean and add Aongus as collaborator

Notes

- Sprint 4 started with backlogs from sprint 3.
- Most of the required for sprint 4 was implemented without major issues
- Model works
- Made website respond better , added Splash and other stuff to make it look more refined.
- Some features suggested were ignored due to technical and time constraint.
- SSL certificate still and issue , may ignore and use user pointed location in map
- Cleaned code and GitHub
- Deployment is live

sprint4 milestones

Andy Freddie Sachin
finish report

Sachin
Splash screen

Andy
FInd closest bike

Freddie
Flask deployment and integration on EC2

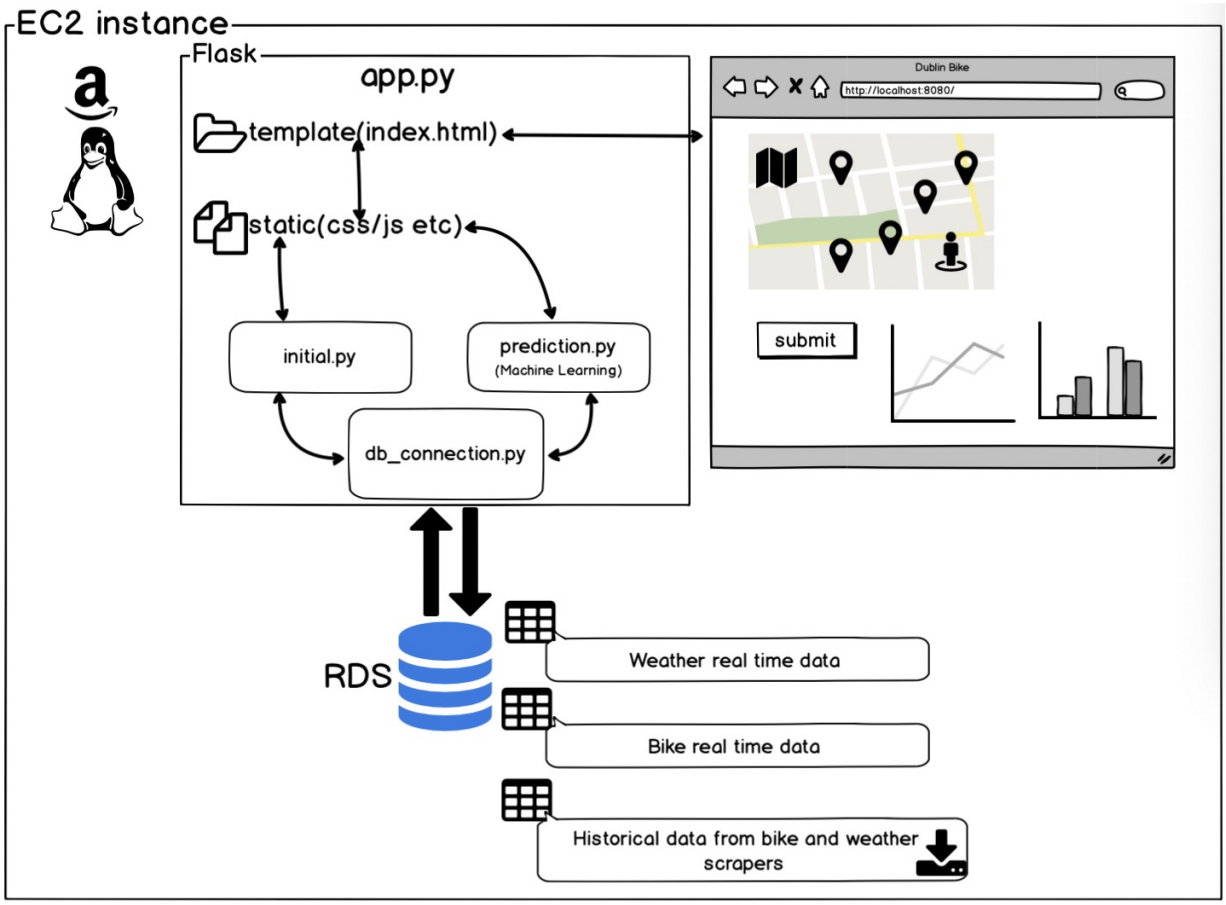
Sachin
loader and final front end

Freddie
charts cleanup

Andy
issue with map initialization bug

Sachin
Burndown chart final render

4.Architecture of app



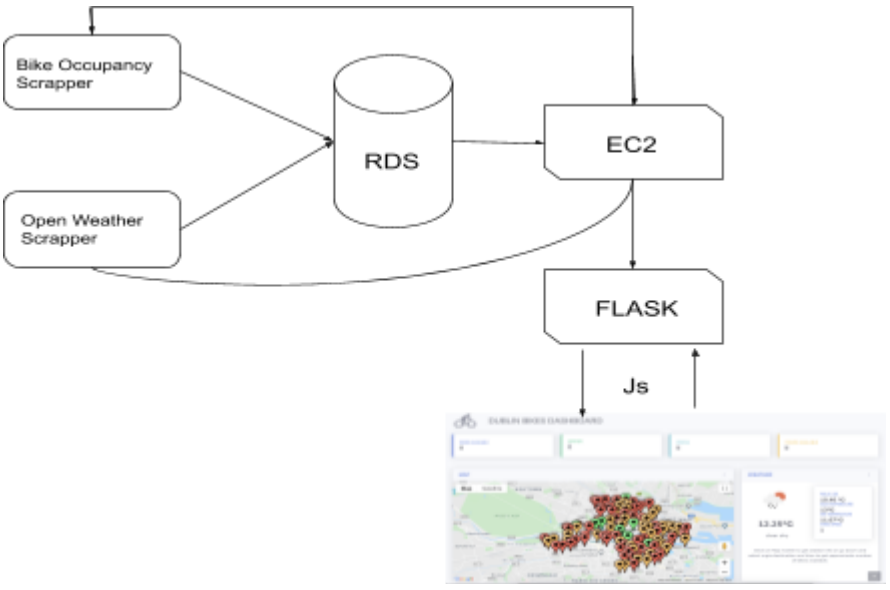


Fig. 13

The architecture of the Dublin Bikes app is as follows. Like any website, the most important part is the server. In this app, we created a flask based server. The server accepts the request and sends back the required data to the front end. The data is then made visible by the help of javascript. At some

places like the submit button, AJAX was used so that the front end need not refresh all the time. The server is hosted in a virtual Linux machine in an Amazon EC2 instance. The EC2 instance also has two python scripts running on it by cron-scheduler which takes data from two API and inserts it into an RDS database. The flask interacts with RDS and Javascript to make dynamic contents like weather widget on the web page.

4.1 Technology used

4.1.1 Frontend: HTML, CSS, JavaScript, jQuery, Ajax, BootStrap

- Grid system from BootStrap is introduced to build the layout of this web page and we also add more CSS code to make it complete and integrated. The bootstrap libraries are not loaded via CDN as sometimes not every library is loaded so we decided to keep it as part of flask assets.
- JavaScript and jQuery are used to send Ajax requests, process data passed to and from flask and fill required data in HTML tags.
- Ajax allows parts of the web page to be updated asynchronously by exchanging data with the backend without reloading the whole web page, which, to a great extent, enhances the user experience.

4.1.2 Backend: Python, Flask, Jinja2, MySQL

- The Flask application on the EC2 was designed modularly so that debugging and scaling would be much easier with different functional modules separated. This was achieved by putting different functions in blueprints that are registered in the app.py, the entry point of the Flask application.
- Jinja2 is the templating language for passing data from flask to HTML. The possible downside of using Jinja2 is to lead to low readability of the code because it is embedded into the front end code.
- Python for data analytics. See Part3 Analytics

4.2 Issues faced and solutions

- Scrappers could only run for 30 minutes. Solution: Using crone scheduler instead of an infinite loop
- The Dublin bike API of one member in the team got blocked due to frequent requests in a short amount of time. Solution: Register a new API key limit API calls; lesson learnt.
- Multiple versions of Flask application with each of us using different libraries and implementations. Solution: Pick better code and implementation and integrate them together.
- The google map wouldn’t load occasionally. Solution: We found out this is closed related to the load and execution order of script tags, so we switched the order and it functioned well.

- GPS location worked well on localhost but not on the EC2. It turned out the EC2 demands a security certificate which we didn't have using HTTP protocol. Solution: We decided to hardcode and mimicked a user location featuring a person icon.
- Splash screen not working in chrome. This was mainly due to caching of the previous version rectified by clearing the cache
- Slow prediction. The solution mentioned in 5.8.2
- Messy Github. We pushed lots of different files onto the Github. Solution: Clean it up, leaving the master branch only the stable version of the application.
- Dropdown minor bug. mentioned in section Bugs
- When we prepared a database for analytics, one of the databases did not have bike occupancy information hence could not be used for analytics. Luckily all three of us implemented our own version of the db so we used another database

5. Analytics

5.1 Machine learning model

The Ml model enabled us to carry out the predictions of available bikes and stations based on inputs provided. The model can be described in the following equation.

$$prediction = f(starting\ station, destination, time\ of\ travel, weather)$$

To train this model we needed to collect historical data. For this purpose we ran two scrappers one for weather and another for bike occupancy and stored in a database. This data was used for training the model.

A plethora of ML models were available to make the predictions. Our main aim was to develop the simplest model with this objective; we trained a linear regression that didn't provide a satisfactory result. Due to the poor results we looked in other models and found out a Random forest Regressor was the best fit.

The reasons are as follows :

- Our goal is to predict the number of available bikes and slots which is a regression task, and the random forest has both classification and regression algorithms.
- Random forest algorithm is an assemble algorithm, so the performance of random forest algorithms is always better than the single model algorithms.
- The random forest algorithm provides a reliable feature importance estimate to help me figure out which feature to use. In the model training progress, It found that the "Bonus" feature is useless based on the feature importance score. Therefore it was removed
- Since linear regression algorithms provided a very bad performance, we inferred that our dataset is non-linear. Random forest is robust to deal with outliers as well as non-linear data.
- Training and prediction are fast. There are more than 200,000 samples in the training dataset. We needed a model that can be trained fast. Also, it is better to have a fast prediction speed which affects the reaction time of our website. The random forest model can be stored in a file for future use, and this perfectly suits the requirement.

5.2 Data preprocessing

We downloaded over one month's historical data and stored it in the database. But for the training and testing process, we only used one week's historical data, from March 14 to March 20, for several reasons:

1. That period is a completed week so that the model would not miss any information on any day of a week.
2. The weather information begins on March 13, and March 14 is the first day with data for 24 hours.
3. During that period, Dublin was not locked down and people were not restricted to go out. So the data collected at that period is reliable.
4. There are more than 200,000 samples in one week which we think is enough for this task. Using a large dataset to train the model might lead to overfitting if there are many noises in the training dataset. Also, a large training dataset might lead to a long training time as well as a large model.

In our dataset, we have 13 features for bike station: "update time", "station number", "station name", "address", "latitude", "longitude", "banking", "bonus", "bike stands", "available bike stands", "available bikes", "station status" and "last update".

Update_time	Station_number	Station_name	Address	Latitude	Longitude	Banking	Bonus	Bike_stands	Available_bike_stands	Available_bikes	Station_status	Last_update
Thu Feb 20 17:08:39 2020	42	SMITHFIELD NORTH	Smithfield North	53.349562	-6.278198	True	False	30	29	1	OPEN	1582218222000
Thu Feb 20 17:08:39 2020	30	PARNELL SQUARE NORTH	Parnell Square North	53.353462	-6.265305	True	False	20	11	9	OPEN	1582218086000
Thu Feb 20 17:08:39 2020	54	CLONMEL STREET	Clonmel Street	53.336021	-6.262980	False	False	33	33	0	OPEN	1582217871000
Thu Feb 20 17:08:39 2020	108	AVONDALE ROAD	Avondale Road	53.359405	-6.276142	False	False	40	29	11	OPEN	1582217990000
Thu Feb 20 17:08:39 2020	56	MOUNT STREET LOWER	Mount Street Lower	53.337960	-6.241530	False	False	40	27	12	OPEN	1582218462000

Fig 10 Bike station dataset

Since the "station number", "station name", "address" and the combination of "latitude" and "longitude" are all identifiers of a station, we will use station number only, and discard the rest. We also used "banking", "bonus" and "bike stands" since we were not sure whether they would affect the result or not. The "available bikes" and the "available bike stands" features are our targets. We discarded the "station status" because it is a constant feature with all the values "OPEN". The "last update" feature was also discarded because it seems to be a string to record a station's update time in JCDecaux and we thought it is unhelpful to our prediction model.

	number	position_lat	position_lng	status	main	description	temp	feels_like	temp_min	temp_max	wind_speed	icon	pressure	humidity	date	time
0	42	53.349562	-6.278198	OPEN	Clouds	broken clouds	7.04	3.37	6.67	7.22	2.6	04d	998	65	2020-3-5	16:20:17
1	30	53.353462	-6.265305	OPEN	Clouds	broken clouds	7.04	3.37	6.67	7.22	2.6	04d	998	65	2020-3-5	16:20:17
2	54	53.336021	-6.262980	OPEN	Clouds	broken clouds	7.04	3.37	6.67	7.22	2.6	04d	998	65	2020-3-5	16:20:17
3	108	53.359405	-6.276142	OPEN	Clouds	broken clouds	7.04	3.37	6.67	7.22	2.6	04d	998	65	2020-3-5	16:20:17
4	56	53.337960	-6.241530	OPEN	Clouds	broken clouds	7.04	3.37	6.67	7.22	2.6	04d	998	65	2020-3-5	16:20:17

Fig 11 Weather dataset

About the weather dataset, we dropped the "position_lat", "position_lng", "number", "status", "date" and "time" features because they are duplicated in the bike station dataset. We also dropped the "description" column because we can leave the "main" column which represents similar information.

So, all the features that we used in the training process are: weekday, hour, station number, total bike stands, banking, bonus, main (the main description of today's weather), temperature, feels like, minimum temperature, maximum temperature, wind speed, pressure, and humidity.

5.3 Model training and evaluation

First, we split the whole dataset into a training dataset (70%) and a testing dataset (30%). Then, we trained the linear regression model with the training dataset and tested it with the testing dataset. The evaluation measures used in the testing process are

1. Mean absolute error (MAE)
2. Root mean squared error (RMSE)
3. R2

For the linear regression model, the MAE was 6.08 and the RMSE was 7.57. Both are too high and are way greater than 0 which is the value that the MAS and RMSE are expected to be. And the R2 of the linear regression model is 0.05, but it was expected to be or be close to 1. So, overall, the linear regression model did not meet our needs.

```
clf = LinearRegression()
clf = clf.fit(Xtrain,Ytrain)
# score = cross_val_score(clf,Xtest,Ytest,cv=10).mean()
score = clf.score(Xtest,Ytest)
score
```

```
0.05402934972241957
```

```
test_prediction = clf.predict(Xtest)
printMetrics(Ytest, test_prediction)
```

```
==============================================================================
MAE:  6.079886235069869
RMSE:  7.567007762066647
R2:  0.05402934972241957
```

Fig 12 Linear Regression

Then we fitted the random forest regression model and assessed it with the same evaluation measures. The MAE and RMSE were 0.40 and 0.94 respectively, and the R2 was 0.9856. All the measures showed that the random forest regression model did a good job, the accuracy was high.

```
rfc = RandomForestRegressor(
                            random_state=0
                            )
rfc = rfc.fit(Xtrain,Ytrain.Available_bikes)
score = rfc.score(Xtest,Ytest.Available_bikes)
# score = cross_val_score(rfc,Xtest,Ytest,cv=10)
score
```

```
/opt/anaconda3/envs/sklearn/lib/python3.7/site-packages/sklearn/ensemble/forest.
lue of n_estimators will change from 10 in version 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
```

```
0.9856741641378235
```

```
test_prediction = rfc.predict(Xtest)
printMetrics(Ytest, test_prediction)
```

```
==============================================================================
MAE:  0.40229667222920495
RMSE:  0.9364969523537686
R2:  0.9856741641378235
```

Fig 13 Random Forest Regression

5.4 How to predict

To predict the available bikes for the future, we need all the features in the training dataset. I divided all features into 2 classes :

1. Inputted by users: weekday, hour, station number
2. Backend data: total bike stands, banking, bonus, all features about weather

For the first class, we would encourage users to input the time when they want to use a bike and input the start station and destination station.

For the backend data, we directly got these data from our database because we have two real-time data scraping programs that would keep our database up to date.

With all the features we need, the prediction program would be invoked two times. One for the prediction of the number of available bikes of the starting station, another for the prediction of the number of available bike stands of the destination station.

5.5 Integrate model with frontend

As you can see in Fig 6, we created five dropdowns for input, in which the fifth one “Minute” is optional. The user can select the “Minute” or not before submitting, and it doesn't make any difference because the minute is not a feature that we need for prediction. But the “Start station”, “Destination station”, “Date” and “Hour” are compulsory, and a piece of prompt information would pop up to remind the user if they did not select any one or more of them. The following is the screenshot of the prompt information in Chrome.

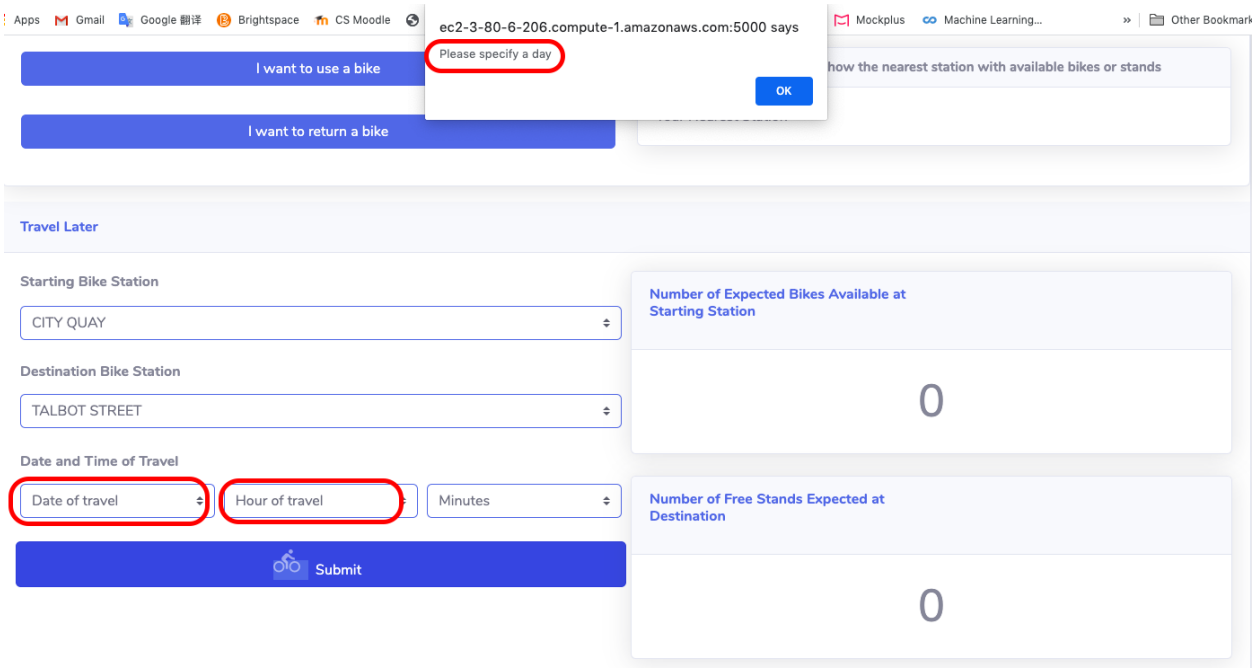


Fig 14 Input without date and hour

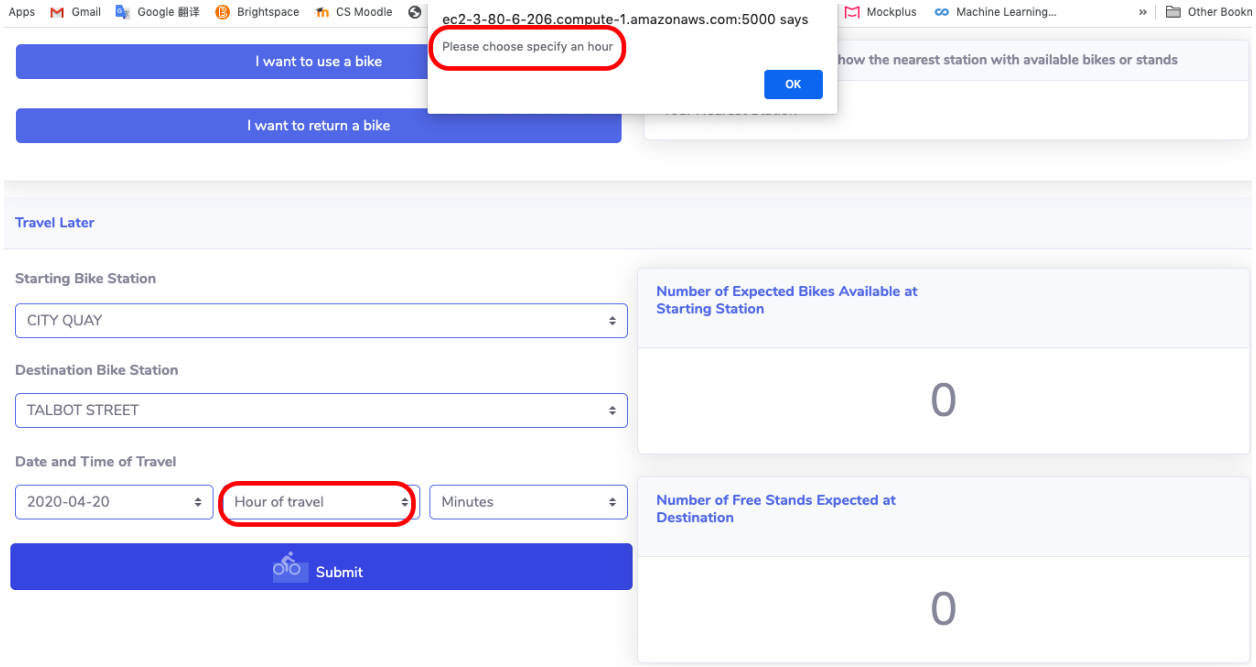


Fig 15 Input without the hour

Since our website would remember the start station and the destination station chosen by the user last time, these two values would not miss in most cases. But we do have prompt information if the start

station and/or the destination station are not given for some reason, like clear the cached data of the browser and forget to specify a station. After the user selected all the required information and clicked the "submit" button, the request would be sent to the backend and the loaders are loaded before the real prediction results are returned back.

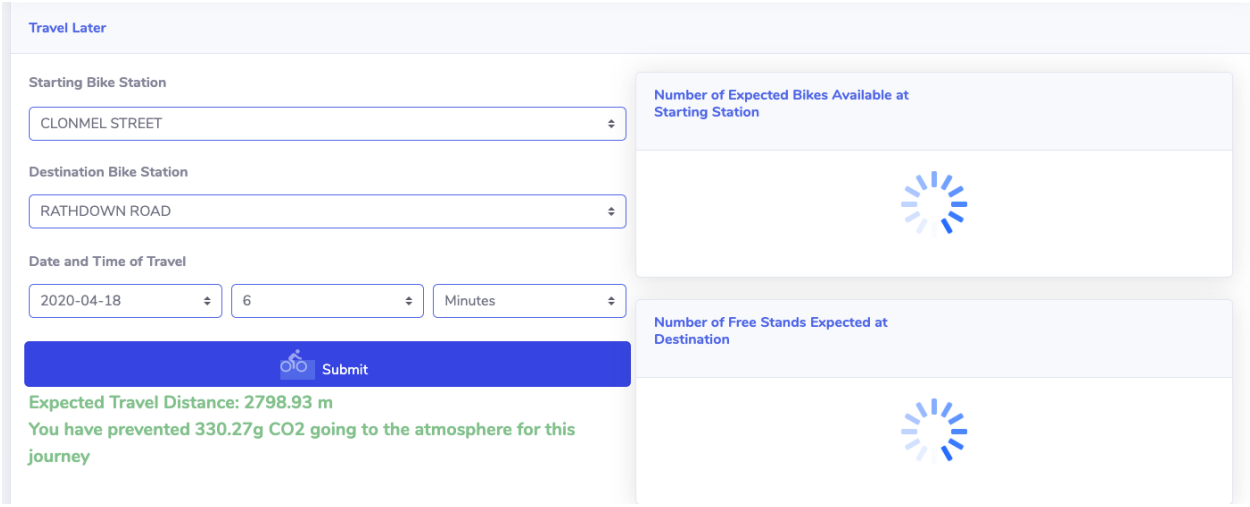


Fig 16 Waiting for prediction result

Except for the prediction of available bikes and stands of the given stations, we also have two charts showing the hourly prediction on the date chosen by the user to provide more useful information.

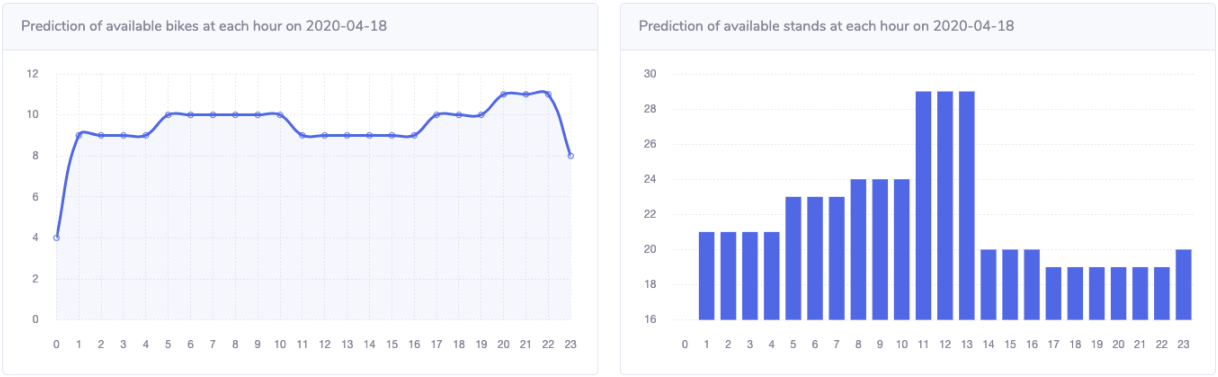


Fig 17 Prediction graph of a given date

These charts show the trend of how the number of bikes and stands might change during the whole day. I think this information may help users make a decision on when to use a bike.

To draw these charts, we need to invoke our prediction program 24 times for each submission. Also, we might use different weather information because the weather information would be refreshed every three hours according to the open weather API.

24 times prediction is a lot of work to do and it might need a long time. So, we separate these graphs and the prediction that the user-specified into two Flask requests so that the user would not wait too long just for these two graphs.

5.6 Integrate model with backend (database)

As for the backend, we have two tables in the database used to support the prediction program, one table for storing the real-time data of all bike stations and another table for storing the weather forecast data of the coming 5 days. Also, there are two scrapping programs running on the EC2 instance to make sure the information is up to date. The bike station scrapper would run every 5 minutes and the weather forecast scrapper would run every 3 hours.

forecast

Result Grid | Filter Rows: Search | Edit: | Export/Import:

datetime	hour	temp	feels_like	temp_min	temp_max	pressure	humidity	wind_speed	main
2020-04-17 03:00:00	03	15.47	13.47	15.47	15.47	1010	65	2.51	Clouds
2020-04-17 06:00:00	06	13.69	12.11	13.69	13.69	1011	71	1.78	Clouds
2020-04-17 09:00:00	09	12.82	11.31	12.82	12.82	1011	75	1.67	Clouds
2020-04-17 12:00:00	12	12.32	10.41	12.32	12.32	1012	75	2.07	Clouds
2020-04-17 15:00:00	15	12.85	10.91	12.85	12.85	1013	69	1.87	Clouds
2020-04-17 18:00:00	18	17.77	15.81	17.77	17.77	1013	50	1.87	Clouds
2020-04-17 21:00:00	21	20.83	18.03	20.83	20.83	1012	41	3.03	Clouds
2020-04-18 00:00:00	00	17.83	14.79	17.83	17.83	1011	51	3.52	Clouds
2020-04-18 03:00:00	03	14.1	11.36	14.1	14.1	1012	62	2.89	Clouds
2020-04-18 06:00:00	06	12.28	9.49	12.28	12.28	1014	67	2.77	Clouds
2020-04-18 09:00:00	09	11.73	8.9	11.73	11.73	1015	68	2.74	Clouds
2020-04-18 12:00:00	12	11.36	8.81	11.36	11.36	1015	68	2.23	Clouds
2020-04-18 15:00:00	15	12.65	10.36	12.65	12.65	1016	63	1.9	Clouds
2020-04-18 18:00:00	18	17.83	15.94	17.83	17.83	1017	46	1.41	Clouds
2020-04-18 21:00:00	21	18.92	15.95	18.92	18.92	1017	45	3.15	Clouds
2020-04-19 00:00:00	00	18.8	15.93	18.8	18.8	1016	46	3.08	Clouds
2020-04-19 03:00:00	03	14.28	11.6	14.28	14.28	1016	61	2.78	Clear
2020-04-19 06:00:00	06	12.29	10.15	12.29	12.29	1018	70	2.05	Clouds
2020-04-19 09:00:00	09	11.49	9.74	11.49	11.49	1018	73	1.45	Clouds
2020-04-19 12:00:00	12	10.83	9.27	10.83	10.83	1017	75	1.1	Clouds
2020-04-19 15:00:00	15	12.32	10.91	12.32	12.32	1018	70	1.02	Clouds
2020-04-19 18:00:00	18	17.53	15.55	17.53	17.53	1018	50	1.82	Clouds
2020-04-19 21:00:00	21	20.08	16.85	20.08	20.08	1017	41	3.43	Clouds
2020-04-20 00:00:00	00	18.38	15.02	18.38	18.38	1016	47	3.76	Clouds
2020-04-20 03:00:00	03	13.84	10.92	13.84	13.84	1016	63	3.15	Clouds
2020-04-20 06:00:00	06	11.62	9.32	11.62	11.62	1018	72	2.21	Clouds
2020-04-20 09:00:00	09	10.5	8.4	10.5	10.5	1017	73	1.65	Clouds
2020-04-20 12:00:00	12	9.85	7.76	9.85	9.85	1017	73	1.45	Clouds
2020-04-20 15:00:00	15	11.07	8.97	11.07	11.07	1018	67	1.44	Clouds
2020-04-20 18:00:00	18	14.2	11.43	14.2	14.2	1019	54	2.36	Clouds
2020-04-20 21:00:00	21	17.23	14.11	17.23	17.23	1019	45	2.91	Clouds
2020-04-21 00:00:00	00	16.39	12.53	16.39	16.39	1018	48	4.01	Clouds
2020-04-21 03:00:00	03	12.36	9.39	12.36	12.36	1019	64	2.85	Clouds
2020-04-21 06:00:00	06	10.94	8.38	10.94	10.94	1021	72	2.38	Clouds
2020-04-21 09:00:00	09	10.51	8.34	10.51	10.51	1021	74	1.81	Clouds
2020-04-21 12:00:00	12	9.92	8.03	9.92	9.92	1022	75	1.3	Clouds
2020-04-21 15:00:00	15	11.37	9.72	11.37	11.37	1023	68	0.95	Clear

Fig 18 Weather forecast table

realtime_data

Result Grid | Filter Rows: Search | Edit: | Export/Import:

Update_time	Station_number	Station_name	Address	Latitude	Longitude	Banking	Bonus	Bike_stands	Available_bike
Thu Apr 16 17:25:02 2020	2	BLESSINGTON STREET	Blessington Street	53.356769	-6.26814	True	False	20	17
Thu Apr 16 17:25:02 2020	3	BOLTON STREET	Bolton Street	53.351182	-6.269859	False	False	20	19
Thu Apr 16 17:25:02 2020	4	GREEK STREET	Greek Street	53.346874	-6.272976	False	False	20	17
Thu Apr 16 17:25:02 2020	5	CHARLEMONT PLACE	Charlemont Street	53.330662	-6.260177	False	False	40	26
Thu Apr 16 17:25:02 2020	6	CHRISTCHURCH PLA...	Christchurch Place	53.343368	-6.27012	False	False	20	15
Thu Apr 16 17:25:02 2020	7	HIGH STREET	High Street	53.343565	-6.275071	True	False	29	29
Thu Apr 16 17:25:02 2020	8	CUSTOM HOUSE QUAY	Custom House Quay	53.347884	-6.248048	False	False	30	28
Thu Apr 16 17:25:02 2020	9	EXCHEQUER STREET	Exchequer Street	53.343034	-6.263578	False	False	24	15
Thu Apr 16 17:25:02 2020	10	DAME STREET	Dame Street	53.344007	-6.266802	True	False	16	16
Thu Apr 16 17:25:02 2020	11	EARLSFORT TERRACE	Earlsfort Terrace	53.334019	-6.258371	False	False	30	20
Thu Apr 16 17:25:02 2020	12	ECCLES STREET	Eccles Street	53.359246	-6.269779	False	False	20	12
Thu Apr 16 17:25:02 2020	13	FITZWILLIAM SQUAR...	Fitzwilliam Square...	53.336074	-6.252825	False	False	30	20
Thu Apr 16 17:25:02 2020	15	HARDWICKE STREET	Hardwicke Street	53.355473	-6.264423	False	False	16	12
Thu Apr 16 17:25:02 2020	16	GEORGES QUAY	Georges Quay	53.347508	-6.252192	False	False	20	13
Thu Apr 16 17:25:02 2020	17	GOLDEN LANE	Golden Lane	53.340803	-6.267732	False	False	20	9
Thu Apr 16 17:25:02 2020	18	GRANTHAM STREET	Grantham Street	53.334123	-6.265436	True	False	30	21
Thu Apr 16 17:25:02 2020	19	HERBERT PLACE	Herbert Place	53.334432	-6.245575	False	False	30	24
Thu Apr 16 17:25:02 2020	21	LEINSTER STREET S...	Leinster Street South	53.34218	-6.254485	False	False	30	25
Thu Apr 16 17:25:02 2020	22	TOWNSEND STREET	Townsend Street	53.345922	-6.254614	False	False	20	11
Thu Apr 16 17:25:02 2020	23	CUSTOM HOUSE	Custom House	53.348279	-6.254662	True	False	30	11
Thu Apr 16 17:25:02 2020	24	CATHAL BRUGHA ST...	Cathal Brugha Street	53.352149	-6.260533	False	False	20	16
Thu Apr 16 17:25:02 2020	25	MERRION SQUARE E...	Merrion Square East	53.339434	-6.246548	False	False	30	19
Thu Apr 16 17:25:02 2020	26	MERRION SQUARE...	Merrion Square West	53.339764	-6.251988	True	False	20	7
Thu Apr 16 17:25:02 2020	27	MOLESWORTH STRE...	Molesworth Street	53.341288	-6.258117	False	False	20	11
Thu Apr 16 17:25:02 2020	28	MOUNTJOY SQUARE...	Mountjoy Square West	53.356299	-6.258586	False	False	30	21
Thu Apr 16 17:25:02 2020	29	ORMOND QUAY UPPER	Ormond Quay Upper	53.346057	-6.268001	True	False	29	16
Thu Apr 16 17:25:02 2020	30	PARNELL SQUARE N...	Parnell Square North	53.353462	-6.265305	True	False	20	17
Thu Apr 16 17:25:02 2020	31	PARNELL STREET	Parnell Street	53.350929	-6.265125	False	False	20	20
Thu Apr 16 17:25:02 2020	32	PEARSE STREET	Pearse Street	53.344304	-6.250427	True	False	30	12
Thu Apr 16 17:25:02 2020	33	PRINCES STREET / O...	Princes Street / O'C...	53.349013	-6.260311	True	False	23	8
Thu Apr 16 17:25:02 2020	34	PORTOBELLO HARB...	Portobello Harbour	53.330362	-6.265163	False	False	30	9
Thu Apr 16 17:25:02 2020	36	ST. STEPHEN'S GRE...	St. Stephen's Green...	53.337824	-6.256035	True	False	40	36
Thu Apr 16 17:25:02 2020	37	ST. STEPHEN'S GRE...	St. Stephen's Green...	53.337494	-6.26199	True	False	30	21
Thu Apr 16 17:25:02 2020	38	TALBOT STREET	Talbot Street	53.350974	-6.25294	False	False	40	1
Thu Apr 16 17:25:02 2020	39	WILTON TERRACE	Wilton Terrace	53.332383	-6.252717	True	False	20	14
Thu Apr 16 17:25:02 2020	40	JERVIS STREET	Jervis Street	53.3483	-6.266651	True	False	21	16
Thu Apr 16 17:25:02 2020	41	HARCOURT TERRACE	Harcourt Terrace	53.332763	-6.257942	False	False	20	11
Thu Apr 16 17:25:02 2020	42	SMITHFIELD NORTH	Smithfield North	53.349562	-6.278198	True	False	30	13

Fig 19 Bike station real-time info

All the backend data needed by the prediction comes from these two tables. Since we do not have the weather information for each hour, we used the closest time that is accessible in the table. For example, there is forecast information at 7, 10, 13 o'clock in the table, and the user wants to use a bike at 9, then we

would use the weather information at 10 o'clock to predict. If the user wants to use a bike at 8, then we would use the weather info at 7 instead.

5.7 Combine model with Flask

The trained models were saved in two files so that the model can be loaded quickly and the prediction results can be sent back in a short time when there is a request. The prediction program was invoked within the Flask program. Each time the "submit" button was clicked, the Flask program would receive a request and do some actions which include accessing the database to get all information needed by the prediction and invoking the prediction program with all required arguments.

After getting the prediction results, the Flask would return these results back to the frontend, and one request process was finished.

5.8 Issues and solutions

Here are some issues we faced with the machine learning model as well as the solutions we took for each issue. There might be some issues that we didn't deal with for some reason.

5.8.1 Investigate useful features

At first, we selected all the features that might be useful based on our experience. But that is not 100% reliable, and there are some features that we don't understand. So, we used the "feature importance" function of the random forest model to double-check all the features' effects to the final results.

```
feature_importances = pd.DataFrame(rfc.feature_importances_,
                                   index = train_feature,
                                    columns=['importance']).sort_values('importance', ascending=False)
feature_importances
```

	importance
Station_number	0.440465
pressure	0.156723
Bike_stands	0.075990
Day	0.071316
Hour	0.059454
temp	0.041802
feels_like	0.036021
Banking	0.032318
humidity	0.031465
temp_min	0.020390
wind_speed	0.018104
temp_max	0.014292
main	0.001660
Bonus	0.000000

Fig 20 Feature importance

Fig 20 shows the importance of all features in descending order, and we can see that the importance of "Bonus" is 0, which means the feature "Bonus" has no attribute to the prediction. So, we removed the "Bonus" from our useful feature list and trained the random forest model again.

5.8.2 The response time is too long

The first time we trained the random forest model and deployed it with our project, the reaction time (from clicking the "submit" button to actually see the prediction result) was too long (around 15 seconds). The reason was found to be the large model, the model file was around 300 MB. Then we optimized the model by specifying several parameters of the model, like n estimators, max depth, max features, etc. To find the optimized value of each parameter, we draw the learning curve for each parameter.

```
scorel = []
mi = 1
ma = 20
for i in range (mi,ma):
    rfc = RandomForestRegressor(n_estimators=i
                                ,n_jobs=-1
                                ,random_state = 0
                               )
    rfc = rfc.fit(Xtrain,Ytrain.Available_bikes)
    score =rfc.score(Xtest,Ytest.Available_bikes)
#     score = cross_val_score(rfc,Xtest,Ytest,cv=10).mean()
    scorel.append(score)
print(max(scorel),scorel.index(max(scorel))+mi)
plt.figure(figsize=[ma-mi,5])
plt.plot(range(mi,ma),scorel)
plt.show()
```

```
0.986753129038157 19
```

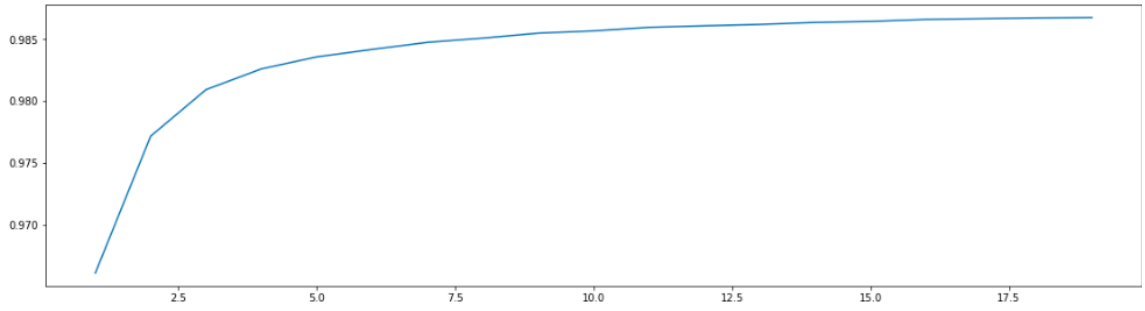


Fig 21 Learning curve of n estimators

```
scorel = []
mi = 1
ma = 40
for i in range (mi,ma):
    rfc = RandomForestRegressor(n_estimators=10
                                ,n_jobs=-1
                                ,random_state = 0
                               ,max_depth=i
                               )
    rfc = rfc.fit(Xtrain,Ytrain.Available_bikes)
    score =rfc.score(Xtest,Ytest.Available_bikes)
#     score = cross_val_score(rfc,Xtest,Ytest,cv=10).mean()
    scorel.append(score)
print(max(scorel),scorel.index(max(scorel))+mi)
plt.figure(figsize=[ma-mi,5])
plt.plot(range(mi,ma),scorel)
plt.show()
```

```
0.9857557547937671 27
```

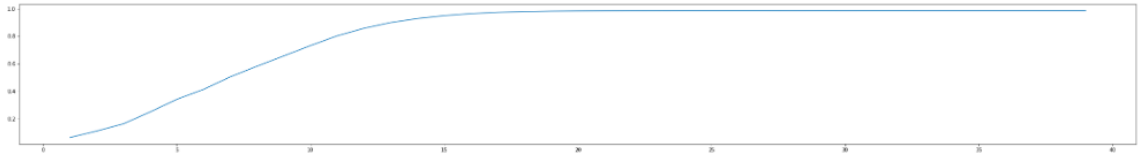


Fig 22 Learning curve of max depth

```
scorel = []
mi = 1
ma = 14
for i in range (mi,ma):
    rfc = RandomForestRegressor(n_estimators=10
                                ,n_jobs=-1
                                ,random_state = 0
                               ,max_depth=27
                               ,max_features=i
                               )
    rfc = rfc.fit(Xtrain,Ytrain.Available_bikes)
    score =rfc.score(Xtest,Ytest.Available_bikes)
#     score = cross_val_score(rfc,Xtest,Ytest,cv=10).mean()
    scorel.append(score)
print(max(scorel),scorel.index(max(scorel))+mi)
plt.figure(figsize=[ma-mi,5])
plt.plot(range(mi,ma),scorel)
plt.show()
```

```
0.9863489943653226 11
```

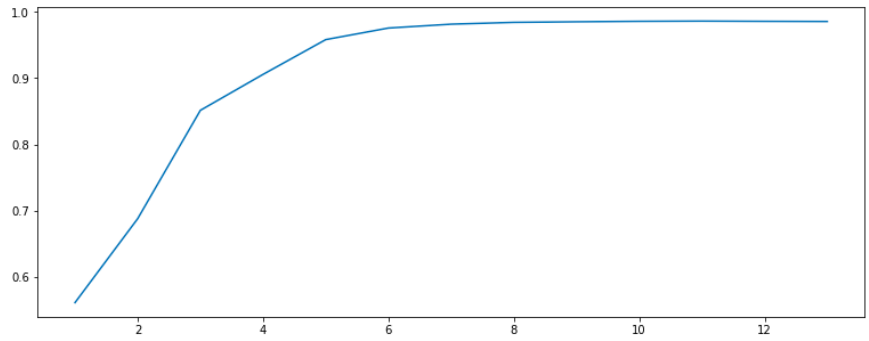


Fig 23 Learning curve of max features

Since the goal of selecting an optimized parameter is not only high accuracy but also to reduce the size of model files, we cannot select values with the highest precision. The finalized parameters of each model are as follow:

1. model for bikes : random_state = 0,n_estimator = 10, max_depth = 28, max_features = 11
2. model for stands : random_state = 0,n_estimator = 10, max_depth = 27, max_features = 11

After doing this, the model file was around 34MB and the reaction time is less than 4 seconds.

4.8.3 The forecast information for the last day is not completely available

The open weather API allows us to access the forecast data of up to 5 days. But the forecast information for the fifth day is not completely available. For example, the forecast data from April 17 to April 21 would be available on April 16, but the data of April 21 might only have the forecast at 0, 3, 6 o'clock if it is 6 am on April 16. In this case, if the user wants to use a bike, for example, at 5 pm on April 21, our prediction program would not find the corresponding weather forecast information in our database because the forecast around 5 pm on April 21 is not available for now. So, our program would use the last sample in the database, which, in the case of this example, should be the sample of 6 am on April 21 as the weather information and do the prediction. This might cause a problem if the weather at 5 pm was quite different from the weather at 6 am on April 21, and the prediction might not be accurate in that case.

6.Future Work

Additional features that can be implemented are real-time location handling. Right now the user has to manually point his location on map before finding the best bike station.
Another feature we can add is a login system where user can log in and have a history of travel and also features like the total amount of Co2 saved per user. Such initialtives can motivate people to use the app.

7 . Bugs

One possible bug which we found out was that the user can enter the date of travel which is one day into the past. The bug was found out late, therefore, couldn’t be rectified

8.Appendix

8.1 Scrum Stand-up Meetings

The full information on all the meetings and timeline is available as a wiki in the link provided below, The pdf format does not provide support for showing it here:

https://github.com/sachsom95/Dublin_bikes_official/wiki

8.2 WhatsApp Communications (Not full list as it is around 30 pages)

4/2/20, 20:47 - sachin: interesting indeed
4/2/20, 22:20 - ssachin: <Media omitted>
4/2/20, 22:20 - ssachin: but date time doesn't work in safari
4/2/20, 22:22 - Liu Andy ucd cs: how about chrome
4/2/20, 22:24 - ssachin: w3 school says crome supports everything
4/2/20, 22:30 - Liu Andy ucd cs: then chrome is the answer
4/2/20, 22:43 - ssachin: freddie hows the output is it a number?
4/2/20, 22:43 - Freddie niu ucd cs: For one sample, it is a number
4/2/20, 22:44 - ssachin: so number of bikes?
4/2/20, 22:44 - Freddie niu ucd cs: Yep
4/3/20, 15:39 - ssachin: <Media omitted>
4/3/20, 15:40 - ssachin: ?/
4/3/20, 15:40 - ssachin: any modifications on this
4/3/20, 15:40 - ssachin: if not I'll make placeholders for charts
4/3/20, 15:41 - Freddie niu ucd cs: I think its ok
4/16/20, 16:58 - Freddie niu ucd cs: done
4/16/20, 17:31 - ssachin: andy did u deploy co2 features ?
4/16/20, 17:32 - Freddie niu ucd cs: yeah
4/16/20, 17:32 - Freddie niu ucd cs: on the server now
4/16/20, 17:32 - Freddie niu ucd cs: u can check
4/16/20, 17:33 - Freddie niu ucd cs: still green font
4/16/20, 17:33 - sachin: no probs
4/16/20, 17:33 - sachin: just wanted to screenshot
4/16/20, 17:33 - Freddie niu ucd cs: i think it's fine and green shows environment friendly