Disappster

A Dublin Bikes Web Application

Project Report

Adam Ryan, Finnian Rogers and Jane Slevin

COMP30830: Software Engineering

UCD School of Computer Science

16/4/2021

Table of Contents

Project Overview

The goal for this project was to develop a web application to display occupancy information for Dublin Bikes with a prediction feature to predict bike availability at a given time using past occupancy and weather information. The application is built on top of the Google Maps API which displays each bike station as a clickable marker on the map which will display information about that particular station once clicked. Along with the map, we have a user interface which contains a drop-down selector of all the station names which will display the same information as clicking the marker on selection. The user interface also displays additional features such as an availability tab, which displays average past hourly and daily availability information for that station with visualisations using the Google Charts API and gives the option for the user to request a prediction for future availability at a particular station at a certain time. We also have a welcome tab that will display current weather information and a tab for our "nearest station" feature, which allows the user to enter their location and see the five nearest stations with them with an option to get directions from their inputted location to any of these stations. We also have a filtering feature that allows the user to filter stations on the map by current bike availability. The application is displayed entirely on one page with the exception of the about page which gives information about the application and is accessible through the navigation bar at the top of the page.

The backend of our application was built using python in the Flask framework. This handles a variety of things ranging from getting the latest availability information to be displayed for each station to handling user requests for future availability predictions. We built a variety of scrapers to gather and store data on bike availability and weather information in a MySQL database on the RDS (Relational Database Service) on AWS (Amazon Web Services). This information is accessed by our Flask backend to show the user the most up to date information and was also used to build our prediction model. The entire back-end of the application is hosted using Flask's built in server on an EC2 (Elastic Compute Cloud) instance on AWS and is accessible through port 5000 on the public IP address of the EC2 instance (34.229.255.100:5000/). More detailed information on the structure of the application can be seen in the architecture section of this report.

As briefly mentioned above, the machine learning component of our application which predicts future bike availability at a particular station at a given time is built using past occupancy and weather information that we gathered and stored. The actual machine learning model is made using the XGBoost python library which we found to return predictions with quite a high level of accuracy considering we had a limited amount of data to work with and this data is likely more unpredictable than normal due to the current pandemic and lockdown. Please see the Data Analytics section of this report for more information on how the machine learning component of the application works.

Our application is not currently built with mobile users in mind, although we would like to add this in future versions of the application (see the Retrospective/Future work section of the report). Our app was instead built with desktop users in mind and intended to function as more of a kind of journey planning app than an app you can use while out and about on a mobile device to quickly find a nearby station. This design choice for the application can also be seen in how our nearest station feature functions. It currently works by manually entering an address with the help of the Google Maps API and will show nearby stations to that manually entered address with the option for directions instead of automatically obtaining the user's geolocation using GPS or other technologies. This is because we did not build the current iteration of the application with mobile users in mind, although we would like to expand our application and this feature to add support for mobile users and automatically obtain a user's location in the future.

We believe the stand-out features in our application compared to similar applications to be the level of accuracy of our machine learning model and our "nearest stations" feature. We are also particularly happy with the layout of our codebase which is highly modularised, making it easier to scale and maintain if more features are added future versions of the application.

Although there are certain areas we would like to expand on in future versions and additional features we would like to add if given more time, overall we believe this to be quite a complete, highly functioning product that achieves what it is supposed to do very well with a strong foundation that would be relatively easy to build on and maintain in the future.

Architecture

Frontend Architecture

Google Maps API

The Google Maps API was used to embed a map within our Flask application and the Google Maps Distance Matrix API, Directions API, Places, Charts and Geometry libraries in the Maps Javascript API were used to extend the functionality of the map and add interactivity.

The Google Maps API is a powerful tool that provides extensive customisation ability. We used Google Maps Marker objects to display each of the 109 Dublin Bikes stations on the map. The appearance of these markers was customised by specifying icon image files which would be displayed instead of the default pushpin icon, with the specific image displayed dependent on current availability data retrieved from the database. An event listener was added to each marker to display an info-window on click of the marker or on click of various options in the side-panel, and the contents of the info-window was filled using data retrieved from our database. The appearance of the map was customised using the Google Maps Styling Wizard, a very easy-to-use styling tool which allows the web application developer to specify a range of parameters regarding the appearance of the map, such as the colour-scheme and density of various features and generates a JSON object which can be passed to the style property of the map. The Google Maps Distance Matrix API and Places library were used to add a Marker to the map when the user entered a location, allowing the user to view that location (discussed in greater detail below).

Google Maps Distance Matrix API

The nearest-stations-locator feature, which enables the user to input a location and retrieve a list of the five closest stations to that location and travel distance to each, was implemented using the Google Maps Distance Matrix API. This service is an extension of the Google Maps API and returns travel distance as calculated by the Maps API based on the recommended route between two points.

User input for this feature was collected via a HTML text box. The Autocomplete feature from the Places library in the Maps JavaScript API was used to add “type-ahead-search behaviour” to this input element, so that as the user enters a location, place suggestions will appear.

Place suggestions are biased towards locations in Ireland with the addition of the Autocomplete Component Restrictions property. When the user selects a location from those suggested, a request is passed to the Distance Matrix.

A geoJSON object containing station coordinates, name, number, current number of available bikes and current number of available bike stands was created and added to the Google Maps data layer. The data layer can be used to store geospatial data, which can be used to customise the map or in this case to pass pertinent data to the Distance Matrix API.

The Distance Matrix API can calculate travel distance for a matrix of origin and destination points but is limited to a maximum of 25 origins or destinations. The origin location in the case of our web application is the location entered by the user and the destination points are the Dublin Bikes stations, of which there are 109. In order to reduce the number of destination points passed per request to the Distance Matrix API, the Maps Javascript API Geometry Library is used to calculate the length of the shortest path between the origin location and each of the 109 stations. The ten stations closest to the origin location in terms of shortest path are then passed to the Distance Matrix API, which calculates the walking distance between the origin location and each of these stations. Ten station locations are passed when just five are displayed in order to reduce the likelihood of inaccurate output to the user, as shortest direct distance does not guarantee shortest travel distance. Finally, the five stations with the shortest calculated walking distance are added to a table and displayed on the web page, along with the walking distance for each. The travel mode for calculations is set to walking as we believe that a user interested in renting a bike will most likely be on foot. However, we envisage that in future generations of the application the functionality of this feature would be increased to enable the user to select a travel mode.

Google Maps Directions API

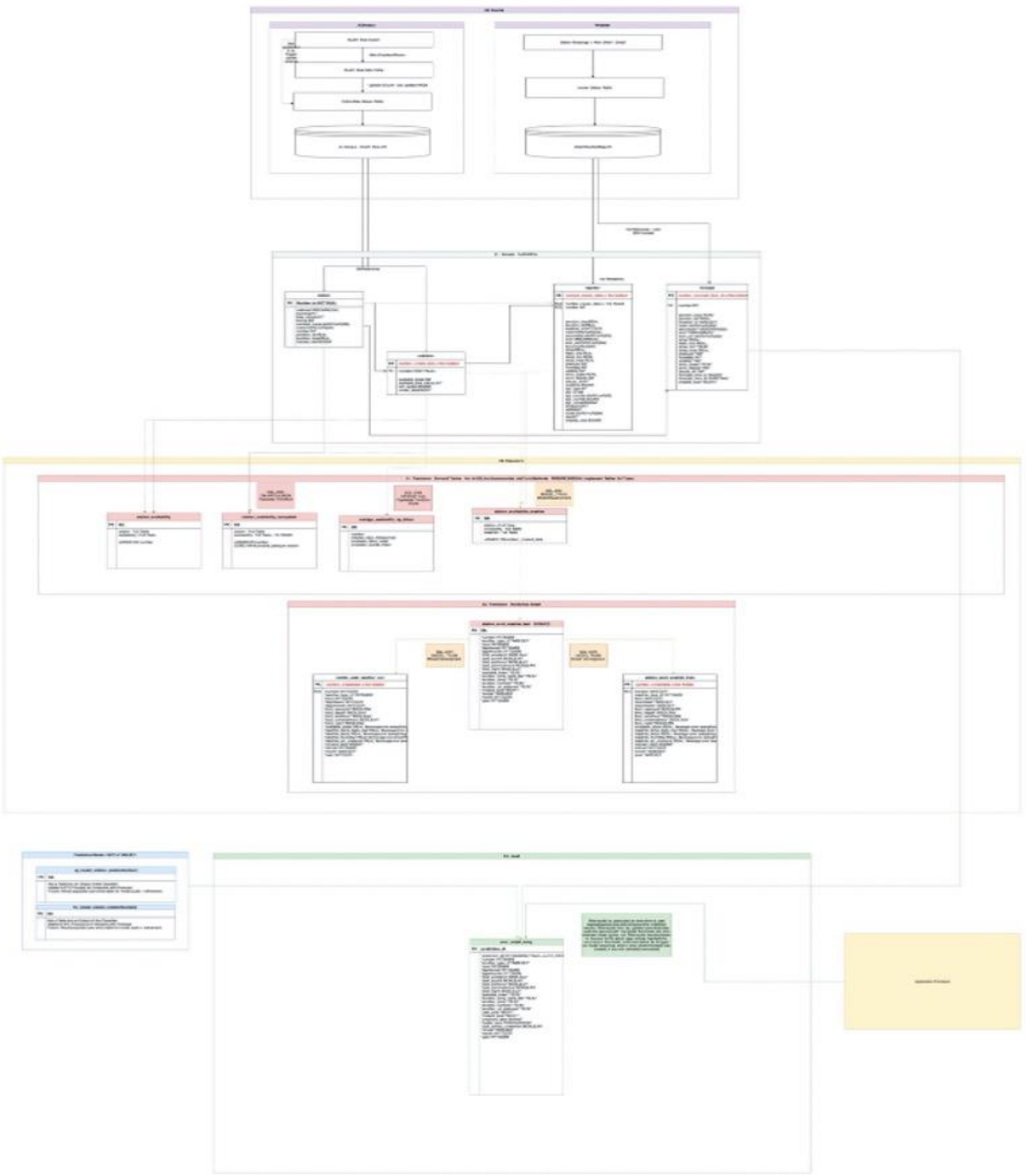
The Google Maps Directions API was used to extend the functionality of the nearest-stations-locator, enabling the user to select a station from the list of nearest stations, receive text directions to that station and view the route to the station on the map. To generate directions, a request is passed to the Directions Service containing the origin and destination locations and travel mode. The result of this request is rendered on the frontend in the form of a polyline marking the route on the map and text directions displayed in a HTML container.

Google Charts

The Google Charts API was used to provide the visualisations for our application. Although there are many excellent charting APIs in JavaScript, we chose to use the Google Charts API due to its ease of use and built-in interactivity. We were also provided with a tutorial on using the Google Charts API in this module which made it easier to get up and running than other libraries. We create visualisations in the form of bar charts for historical daily and hourly bike availability averages for each station which are rendered dynamically on the front end upon station selection. These visualisations are interactive, making it possible to hover over any bar in the chart to display information for that particular hour or day.

Backend Architecture

Overall Data Model



The figure above is the overall view of the Disappster data model. This figure captures the key integrations into our application, the tables within our internal database that is storing the dating we are retrieving from these external sources, the transformations which are

happening as part of the processing of our data, and the tables which are present for the purpose of model validation and auditing. These steps will be detailed more thoroughly in the sections below.

Source

Our application involves integrations into two external data providers by pulling data via API.

JCDecaux – Every 5 Minutes

JCDecaux is the key provider of the DublinBikes API, and stores details on the changes of bike station info and availability info. Based on an initial data exploration, we identified that the JSON output provided by JCDecaux contained data initially on 109 stations (increased to 110 later in the project as they began testing hybrid bikes) located within Dublin City Centre.

The data which is returned in the JSON response, and our initial determination as to what each field referred to and consisted of, is detailed in the table below:

Name	Type	All Instances	PK	FK	Field Values	Nullable	Description
number	int64	Yes	Yes	No	1, 2, … , 109	No	PK of Station
contract_nar	varchar(6)	No	No	Yes	dublin	No	Denotes JCDecaus's contract
name	varchar()	Yes	No	No	'MATER HOSPITAL'	No	Name of Station
address	varchar()	Yes	No	Maybe	Wolfe Tone Street	No	Address Name
position	Dictionary	Yes	No	Maybe	{'lat': 53.333653, 'lng': -6.248345}	No	Position - Latitude and Longitude as Dictionary
banking	bool	No	No	No	True/False	No	??? - Determine Meaning
bonus	bool	No	No	No	True/False	No	??? - Determine Meaning
bike_stands	int64	No	No	No	0,…,40	No	Maximum Number of Bikes at the Station
available_bik	int64	No	No	No	0,…,x-1	No	Number of Bikes available at the station
status	object	No	No	No	Open/Close	No	Open or Close - Identify meaning
last_update	float64	No	No	No	Date of Last Update	No	Timestamp updated

After exploring the data which was provided in the response, we quickly determined that stations update at irregular intervals. While some stations appear to update in real time in response to a change in availability, stations appear to also get flagged as updated if there has been no change in availability after some pre-determined time, likely configured on JCDecaux's system.

Based upon this information, we have identified one possible mechanism in which JCDecaux updates their station API and detailed what we suspect to be their internal process:

1. If a station has a change in available bikes or available stands, either a real-time update via API is introduced into their master table, or the station is pushed into a delta table.

2. If a station has not been updated within a certain amount of time, a check is done on the station to flag the station as having been updated and push it into the delta table to provide an assurance that the station is still being checked (the existence of a delta table versus an update flag is unclear to our group).
3. The update delta is used to update their master station data table which contains information on stations across countries (denoted by the contract_name).

The contract_name detail is flagged as a foreign key under the assumption that JCDecaux maintains a mapping of contracts and in the absence of a contract_id field the name likely serves as the link to their internal table. The address and position fields were flagged as potentials of being a foreign key in that we initially believed they might be useful in connecting to the weather data, but subsequent exploration revealed this not to be the case.

This data is captured in our application via the station_info harvesting module.

OpenWeatherMap – Every 5 Minutes for Real-Time + Every 24 Hours for Forecast

OpenWeatherMap is the service provider of our weather info data. Pulling information from OpenWeatherMap was decided based upon a real-time weather limit of 1M calls per month. Based on our projected usage of 940k-970k weather API calls based upon one call per station every five minutes for a month, and based upon OpenWeatherMap having different weather readings for various locations within Dublin County, we determined that this provider would be appropriate for our needs within the project.

The JSON response to the real-time weather API, and our initial determination as to what each column contained, is detailed below:

Name	Type	All Instances	PK	FK	Field Values	Nullable	Description
position_long	REAL	No	No	No	-6.2656	No	Longitude
position_lat	REAL	No	No	No	53.3581	No	Latitude
weather_id	INTEGER	No	No	No	803	No	Weather ID Type
main	VARCHAR(256)	No	No	No	Clouds	No	Short Description
description	VARCHAR(500)	No	No	No	broken clouds	No	Weather Description
icon	VARCHAR(20)	No	No	No	04d	No	Weather Icon
icon_url	VARCHAR(500)	No	No	No	http://openweather	No	Icon URL
base	varchar(256)	No	No	No	stations	No	Station
temp	REAL	No	No	No	286.21	No	Temp
feels_like	REAL	No	No	No	277.02	No	Feels Like
temp_min	REAL	No	No	No	284.82	No	Min Temp
temp_max	REAL	No	No	No	287.15	No	Max Temp
pressure	INT	No	No	No	1001	No	Air Pressure
humidity	INT	No	No	No	77	No	Humidity
visibility	INT	No	No	No	10000	No	Visibility
wind_speed	REAL	No	No	No	12.86	No	Wind Speed
wind_degree	INT	No	No	No	200	No	Wind Direction
clouds_all	INT	No	No	No	75	No	Clouds???
datetime	BIGINT	No	No	No	1614079641	No	Datetime
sys_type	INT	No	No	No	1	No	?
sys_id	INT	No	No	No	1565	No	System ID?
sys_country	VARCHAR(10)	No	No	No	IE	No	Country
sys_sunrise	BIGINT	No	No	No	1614065172	No	Sunrise Time
sys_sunset	BIGINT	No	No	No	1614102658	No	Sunset Time
timezone	INT	No	No	No	0	No	Timezone ID
id	BIGINT	No	No	No	6691027	No	???
name	VARCHAR(256)	No	No	No	Drumcondra	No	Location Name
cod	INT	No	No	No	200	No	???

The JSON response to the forecast data is largely identical, with one change in that the datetime field is instead a forecast_datetime field. Forecast data is provided in 3-hour blocks over the next 5 days with 4,400 rows of data provided per update across all stations.

One important element of the exploration of the weather data was the identification that the weather segment was not necessarily unique and could be returned as a nested response. Based upon the documentation, we determined that the primary weather type was the first item in this array.

In order to establish a one-to-one link between availability at a time for a station, and a weather update for that station at that time, we elected to take the primary weather type.

The real-time data is captured in our application via the station_info harvesting module, while the forecast info is captured in our application via the forecast_info harvesting module.

Extract Tables

This section details the tables used for the initial capture of data into our application. For the importing of data, the function engine.execute(SQL,Variables) was used to prevent data leaking and also to avoid potential SQL Injection via an API response (e.g. by using string formatting onto a raw SQL statement and the inserted variable escaping the query).

Station

The station table is used for the capture of the static station info from the JCDecaux API.

Name	Type	All Instances	PK	FK	Field Values	Nullable	Description
number	INT	Yes	Yes	No	1, 2, … , 109	No	PK of Station
contract_name	varchar(6)	No	No	Yes	dublin	No	Denotes JCDecaus's contract
name	varchar(256)	Yes	No	No	'MATER HOSPITAL'	No	Name of Station
address	varchar(256)	Yes	No	Maybe	Wolfe Tone Street	No	Address Name
position_lat	REAL	Yes	No	Maybe	53.333653	No	Position - Latitude
position_long	REAL	Yes	No	Maybe	-6.248345	No	Position - Longitude
banking	int64	No	No	No	True/False	No	??? - Determine Meaning
bonus	int64	No	No	No	True/False	No	??? - Determine Meaning
status	varchar(256)	No	No	No	Open/Close	No	Open or Close - Identify meaning
created_date	BIGINT	No	No	No	Date of Last Update	No	Timestamp updated

This is largely identical to the JSON response. We have split the response data into a station and availability table in following the lectures. The only alteration is the addition of the created_date column, and the splitting of the position dictionary into a latitude position and longitude position field for easy access.

Availability

The availability table is used for the capture of the static station info from the JCDecaux API.

Name	Type	All Instances	PK	FK	Field Values	Nullable	Description
number	int64	No	No	Yes	1, 2, … , 109	No	FK to Station
bike_stands	int64	No	No	No	0,…,40	No	Maximum Number of Bikes at the
available_bikes	int64	No	No	No	0,…,x-1	No	Number of Bikes available at the s
last_update	bigint	No	No	No	Timestamp of Upda	No	Timestamp updated
created_date	bigint	No	No	No - Not explicitly	Timestamp of Add	No	Timestamp Added to TB

This is largely identical to the JSON response. We have split the response data into a station and availability table in following the lectures. The only alteration is the addition of the created_date column. The composition of {number}_{created_date} became a link between the availability and weather table, allowing for a one-to-one mapping between weather updates and availability updates.

Weather

The weather table is used for the capture of the weather info from OpenWeatherMap:

Name	Type	All Instances	PK	FK	Field Values	Nullable	Description
number	INT	No	No	Yes - station	1...109	No	Station Number
position_long	REAL	No	No	No	-6.2656	No	Longitude
position_lat	REAL	No	No	No	53.3581	No	Latitude
weather_id	INTEGER	No	No	No	803	No	Weather ID Type
main	VARCHAR(256)	No	No	No	Clouds	No	Short Description
description	VARCHAR(500)	No	No	No	broken clouds	No	Weather Description
icon	VARCHAR(20)	No	No	No	04d	No	Weather Icon
icon_url	VARCHAR(500)	No	No	No	http://openweather	No	Icon URL
base	varchar(256)	No	No	No	stations	No	Station
temp	REAL	No	No	No	286.21	No	Temp
feels_like	REAL	No	No	No	277.02	No	Feels Like
temp_min	REAL	No	No	No	284.82	No	Min Temp
temp_max	REAL	No	No	No	287.15	No	Max Temp
pressure	INT	No	No	No	1001	No	Air Pressure
humidity	INT	No	No	No	77	No	Humidity
visibility	INT	No	No	No	10000	No	Visibility
wind_speed	REAL	No	No	No	12.86	No	Wind Speed
wind_degree	INT	No	No	No	200	No	Wind Direction
clouds_all	INT	No	No	No	75	No	Clouds???
datetime	BIGINT	No	No	No	1614079641	No	Datetime
sys_type	INT	No	No	No	1	No	?
sys_id	INT	No	No	No	1565	No	System ID?
sys_country	VARCHAR(10)	No	No	No	IE	No	Country
sys_sunrise	BIGINT	No	No	No	1614065172	No	Sunrise Time
sys_sunset	BIGINT	No	No	No	1614102658	No	Sunset Time
timezone	INT	No	No	No	0	No	Timezone ID
id	BIGINT	No	No	No	6691027	No	???
name	VARCHAR(256)	No	No	No	Drumcondra	No	Location Name
cod	INT	No	No	No	200	No	???
created_date	BIGINT	No	No	No - Not explicitly	1614101644	No	Date added to database

This data is largely identical to the JSON data. We added on the station number to allow for an easy link to station, and added the created_date column both for easy linkage to availability info and also for auditing purposes. Within this data, temperatures are provided in Kelvin. We elected not to convert these to Celsius on import as this conversion is solely a transformation (-273.15) and Kelvin is the standard unit.

Forecast

The forecast table is used for the capture of forecast info from OpenWeatherMap:

Name	Type	All Instances	PK	FK	Field Values	Nullable	Description
number	INT	No	No	Yes - station	1...109	No	Station Number
position_long	REAL	No	No	No	-6.2656	No	Longitude
position_lat	REAL	No	No	No	53.3581	No	Latitude
weather_id	INTEGER	No	No	No	803	No	Weather ID Type
main	VARCHAR(256)	No	No	No	Clouds	No	Short Description
description	VARCHAR(500)	No	No	No	broken clouds	No	Weather Description
icon	VARCHAR(20)	No	No	No	04d	No	Weather Icon
icon_url	VARCHAR(500)	No	No	No	http://openweather	No	Icon URL
base	varchar(256)	No	No	No	stations	No	Station
temp	REAL	No	No	No	286.21	No	Temp
feels_like	REAL	No	No	No	277.02	No	Feels Like
temp_min	REAL	No	No	No	284.82	No	Min Temp
temp_max	REAL	No	No	No	287.15	No	Max Temp
pressure	INT	No	No	No	1001	No	Air Pressure
humidity	INT	No	No	No	77	No	Humidity
visibility	INT	No	No	No	10000	No	Visibility
wind_speed	REAL	No	No	No	12.86	No	Wind Speed
wind_degree	INT	No	No	No	200	No	Wind Direction
clouds_all	INT	No	No	No	75	No	Clouds???
forecast_time_ts	BIGINT	No	No	No	1614079641	No	Datetime - TS format
forecast_time_dt	DATETIME	No	No	No	01/01/2021	No	Datetime - DT Format
created_date	BIGINT	No	No	No	1614101644	No	Datetime added to DB

This table is almost an exact replica of the weather info. As this was completed later in the project, we added on both a created_date and also a human-readable datetime column in the form of the forecast_time_dt colum.

Transformation Tables

This section details in broad terms the tables used for the transformation of data and the passage of data to the frontend. Many of these tables are constructed via SQL queries or via manipulation in pandas dataframes but not explicitly stored into our database. As a future action, these tables (highlighted in red in our ETL diagram, and joined via a broken link) should be embedded into our database and pre-calculated to increase the efficiency of our application while also allowing for analytics with less overhead.

As one of our team members has prior experience with development in SQL and Django, a significant number of query templates were constructed. These are contained in the sql.py file within our application, and these were established to allow interaction with the database via a limited number of queries.

Derived but Not Stored

Station_Availability

This table is constructed by joining the station table onto all of the availability data via number, and returning all columns in both tables.

Station_Availability_LastUpdate

This table is constructed by joining the station table onto an availability table, where the availability data has been pre-filtered to only display the max update date per station number. This is used to more efficiently pull the last update information for each station. This table is the primary table used to display information in the front end of the application as it contains all info on station and the latest availability update.

Station_Availability_Weather

This table is constructed by joining the station table onto all of the availability data via number, and then joining the weather data onto the availability table using the station number and the created_date, and returning all columns in both tables. This is essentially the

'master' table within our application, and this table is used as a base in training our machine learning models. Due to the size of the availability and weather table, and due to the composite key needed to join this data, this table has significant overhead and is a key target for embedding within an automated process.

Initially, this was going to be used also for the construction of our charts, to allow weather features to be added as dimensions to our charts and allow for easy filtering, however due to the overhead of calling this table, this was deprioritised and a leaner query was used.

Availability_groupedbyTime

In order to display average availability info for various bike stations, tables are constructed pulling from the availability and using pandas to group over various time features (e.g. day, hour, etc.). A table is constructed which contains the station number, the grouping time column (e.g. hour), and the mean available bikes and available stands.

Station_Availability_Weather_LastUpdate

This table is constructed by joining the station table onto all of the availability data via number, and then joining the weather data onto the availability table using the station number and the created_date, and returning the rows which correspond to the latest update for each station. This query is pre-filtered to the latest update information on the availability and weather table and as such is quicker than the primary station_availability_weather tables.

Other Variations

Numerous varieties of these tables were constructed to potentially enable other features or more efficient queries (such as all of the above but for a single station number) however in general the larger query performed such that the more specific queries would not be needed until the data volume grew significantly at which point swapping to a more filtered version may be optimal.

Machine Learning Table

As part of the construction of our models, a table is derived from the station_availability_weather table adding on various features which may be relevant to bike availability and omitting those which are not for the optimisation of an XGBoost Model. The table structure of what is derived is:

Name	Type	All Instances	PK	FK	Field Values	Nullable	Description
number	CATEGORY	No	No	Yes - station	1...109	No	Station Number
weather_type_id',	CATEGORY	No	No	No	400	No	Weather Type
'minute',	INTEGER	No	No	No	10	No	Minute of Update
'hour',	INTEGER	No	No	No	0,...,23	No	Hour of Update
'dayofweek',	INTEGER	No	No	No	0,...6	No	Day of Week of Update
'dayofmonth',	INTEGER	No	No	No	0,...,31	No	Day of Month of Update
'month',	INTEGER	No	No	No	0,...,11	No	Month of Update
'year',	INTEGER	No	No	No	1970,...	No	Year of Update
'bool_weekend',	BOOL	No	No	No	FALSE	No	Is the day a weekend?
'bool_dayoff',	BOOL	No	No	No	FALSE	No	Is the day a weekend or bank holiday?
'bool_workhour',	BOOL	No	No	No	TRUE	No	Is the hour during work hours?
'bool_commutehour',	BOOL	No	No	No	TRUE	No	Is the hour within an hour before or after work hours?
'bool_night',	BOOL	No	No	No	FALSE	No	Is the hour during the night?
'weather_temp_feels_li	REAL	No	No	No	287.15	No	Temperature Feels Like
'weather_temp',	REAL	No	No	No	287.15	No	Temperature Actual
'weather_humidity',	REAL	No	No	No	1000	No	Humidity
'weather_air_pressure'	REAL	No	No	No	1000	No	Air Pressure

Weekend, Day Off, Work Hour, Commute Hour, and Night were added as grouping mechanisms for the days and hours with the goal of capturing factors which might play an important role in the availability of bikes.

In the initial development, the minute, year, and day of month columns were not present and the table instead was based on averages over hours since the weather data only contained hourly blocks, however a decision was made to add in these more granular elements if possible since the time data is more granular and is a more significant factor in COVID times.

Derived and Stored

Test Data Set

The test data set is explicitly stored into our database. This is populated when the machine learning model is generated. It is identical to the Machine Learning Table but filtered to the test data. This can be used to inspect the results of testing and training outside of running the model development.

Training Data Set

The test data set is explicitly stored into our database. This is populated when the machine learning model is generated. It is identical to the Machine Learning Table but filtered to the training data. This can be used to inspect the results of testing and training outside of running the model development.

Loading Tables

In an industrial application development, load tables would be replicas of the transformed data but specifically designed to be loaded by the frontend and optimised for this purpose. Within this application, due to the comparative simplicity and lightweight nature of the app,

we primarily read data directly from the transformation tables and did not explicitly store the results. Only a single table is present in this area.

User_Model_Entry

This table was initially designed as a way to automate the training of the model, and identifying what sort of accuracy the model had. This table was going to capture runs of the prediction function triggered by a user. After the user triggered a run, the prediction date and entered details would be captured into our database. Once the user entry date was in the past, we would have captured info on the station availability at that time and could flag whether our prediction was true or false. We would then be able to set a cut off at which point the model would be retained, and used the prediction set to drive visualisations for our app and aid in the user acceptance of our application's ML element by highlighting this accuracy within the front-end of the application (hence it sitting in the Load section). This would also allow for easy analytics on the model's accuracy, and reporting on our accuracy, potentially triggering the retraining of our ML model once the accuracy falls below a specified cut-off (e.g. once 50% of predictions are out by a factor per station). Unfortunately, due to time constraints this element of the application could not be linked, so while the table sits in the database it is currently orphaned and not interacted upon by any processes.

Name	Type	All Instances	PK	FK	Field Values	Nullable	Description
prediction_id	INTEGER	Yes	Yes	No	1,...	No	Primary Key of Table Prediction number
'number'	INTEGER	No	No	Yes - station	1...109	No	Station Number
'weather_type_id'	INTEGER	No	No	No	400	No	Weather Type
'hour'	INTEGER	No	No	No	0,...,23	No	Minute of Update
'dayofweek'	INTEGER	No	No	No	0,...,6	No	Hour of Update
'dayofmonth'	INTEGER	No	No	No	0,...,31	No	Day of Week of Update
'bool_weekend'	INTEGER	No	No	No	FALSE	No	Is the day a weekend?
'bool_dayoff'	INTEGER	No	No	No	FALSE	No	Is the day a weekend or bank holiday?
'bool_workhour'	INTEGER	No	No	No	TRUE	No	Is the hour during work hours?
'bool_commutehour'	BOOL	No	No	No	TRUE	No	Is the hour within an hour before or after work hours?
'bool_night'	BOOL	No	No	No	FALSE	No	Is the hour during the night?
'available_bikes'	BOOL	No	No	No	1,...	No	Available Bikes Predicted
'weather_temp_feels_like'	BOOL	No	No	No	283	No	Temperature Feels Like
'weather_temp'	BOOL	No	No	No	283	No	Temperature Actual
'weather_humidity'	REAL	No	No	No	1000	No	Humidity
'weather_air_pressure'	REAL	No	No	No	1000	No	Air Pressure
'user_date'	BIGINT	No	No	No	1614101644	No	Timestamp Entered by User for Prediction
'created_date'	BIGINT	No	No	No	1614101644	No	Timestamp of Entry to DB
'predicted_date'	BIGINT	No	No	No	1614101644	No	Timestamp of Model Prediction
'model_type'	VARCHAR(100)	No	No	No	xgboost/sklinear	No	Type of Model Used
'bool_correct_prediction'	BOOLEAN	No	No	No	TRUE/FALSE/NUI	Yes	Was the prediction right - subsequent update once datetime.now()>user_date
'minute'	INTEGER	No	No	No	0,...	No	Minute of Update
'month'	INTEGER	No	No	No	0,...,11	No	Month of Update
'year'	INTEGER	No	No	No	1970,...	No	Year of Update

Design

As discussed in the Project Overview, the primary objective of this project was to create a web application that would enable the user to find a Dublin Bikes station at which there were bikes available to rent. Considering this focus on station location, we decided early in the development process that a map displaying the 109 Dublin Bikes stations would be at the centre of our application. This map was developed and integrated into our Flask application using the Google Maps API during Sprint 2. We intended that the user would be able to interact with this map in order to find a station and view availability information for that station. We identified four basic use-cases which summarise this functionality. These are outlined in the diagram below.

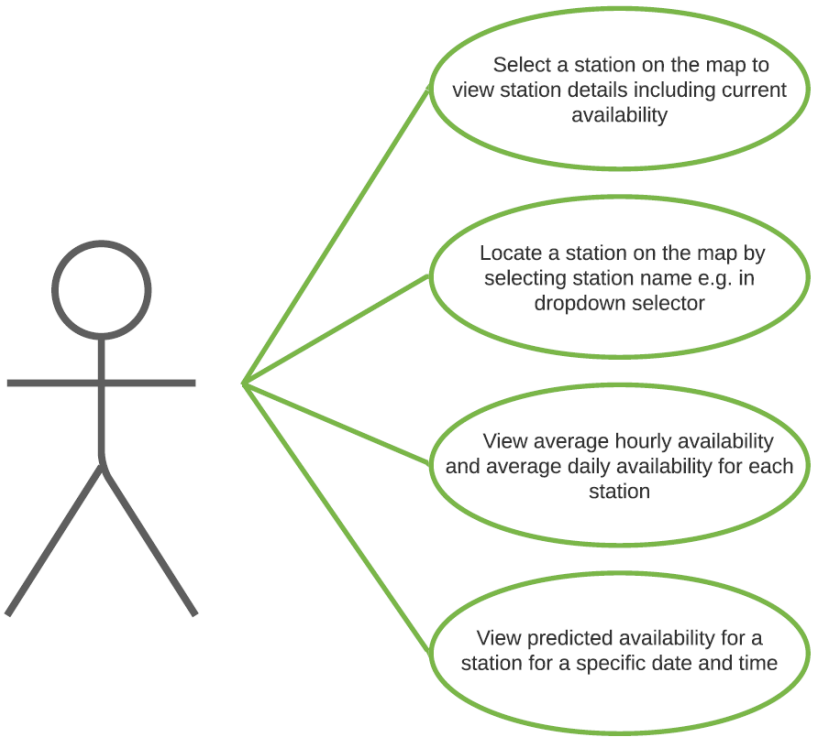


Fig. Use-case diagram

Initial Design

Our initial wireframe, which was created using Figma prototyping software, was designed to reflect this basic functionality (see below). At this point in the project, the application homepage was a full-page map with the stations displayed as clickable markers. Clicking on a

marker triggered an event to open an info-window displaying the station name and we planned to add availability information to the info-window. We also planned to add a dropdown selector to the web page, containing a list of stations. On selection of a station from the dropdown, the info-window for that station would open, therefore identifying the station's position on the map. We intended to add a second "Journey Planner" page to the web application, where the user could again select a station from a dropdown menu and view a summary of the average hourly and daily availability for that station, as well as predicted availability data. We had yet to determine the format in which this data would be displayed.

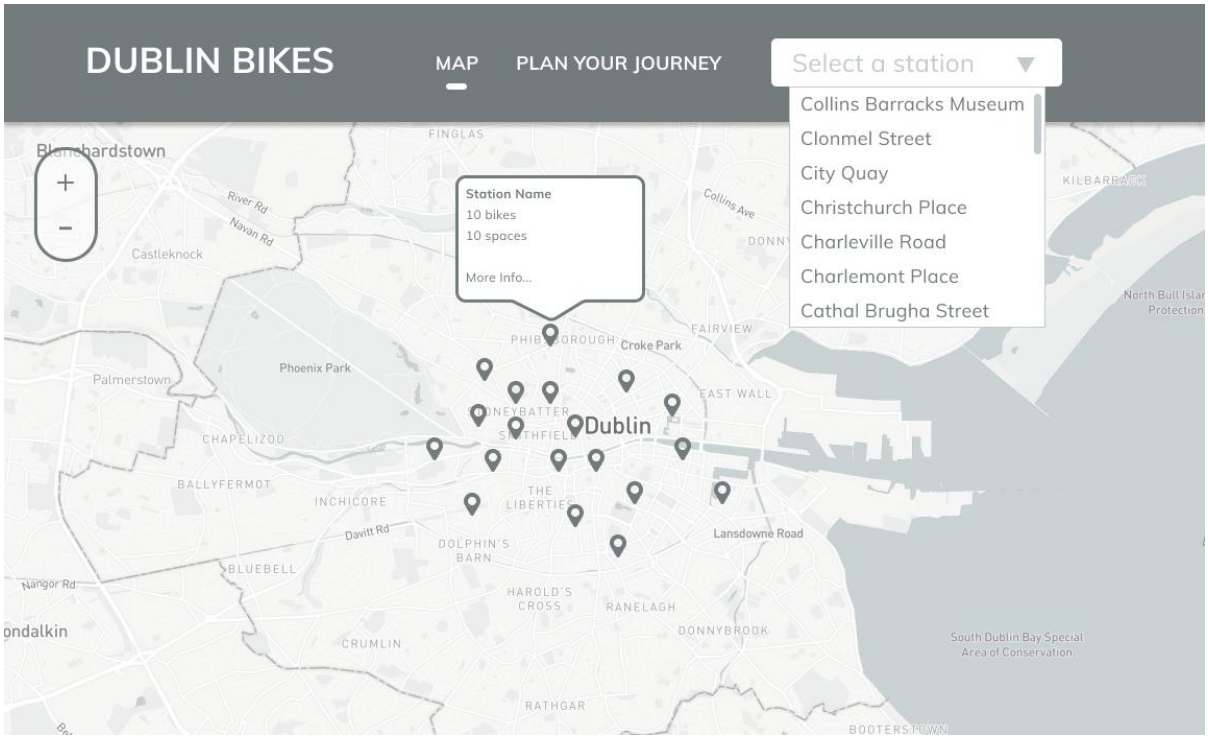


Fig. Initial wireframe

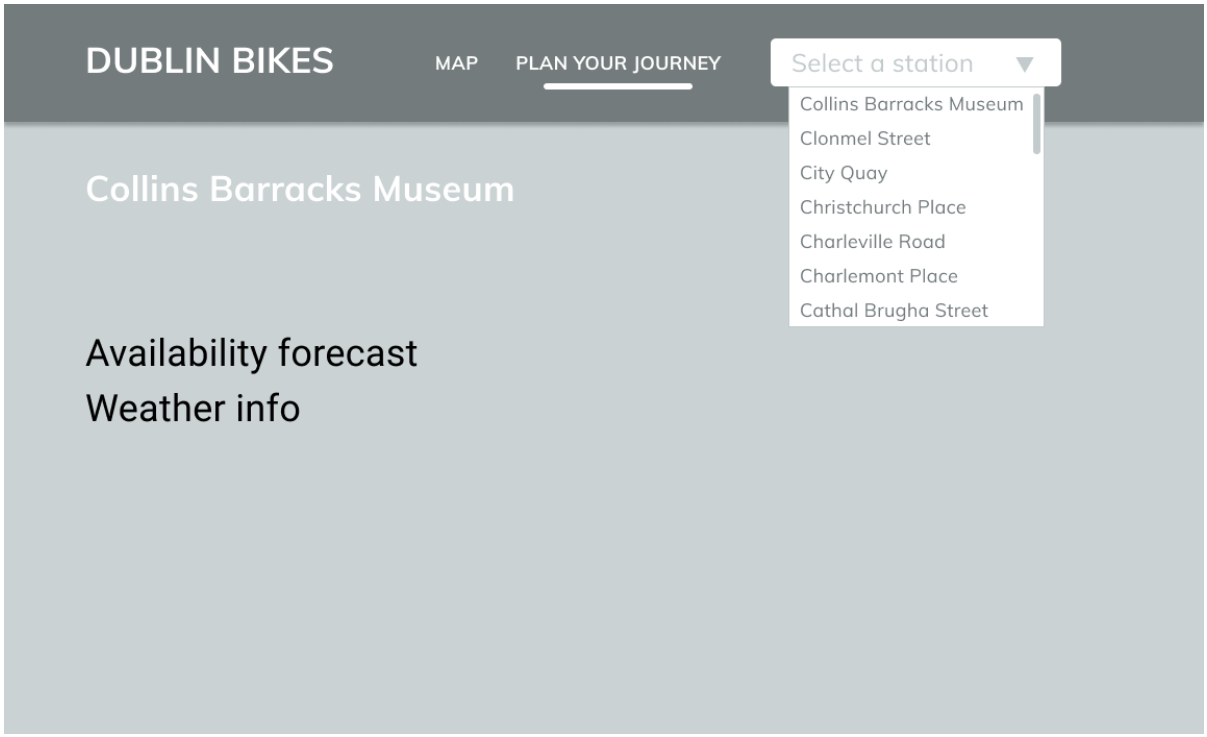
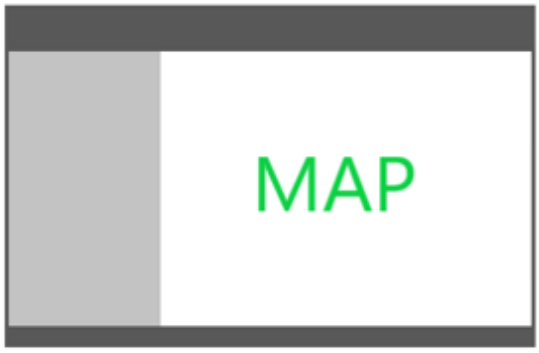


Fig. Proposed second page

Design Modifications

We presented this wireframe to our Product Owner early in Sprint 2 and while he was impressed with the overall design, he felt that the homepage was too map-centric and requested it be modified to include an area in which to display visualisations and other information.



We revised our design following this meeting adding a container to the left of the map, as shown here in light grey, for holding visualisations, average availability data and information pertaining to any additional features implemented. We also reconsidered our plans to add a second page to the web application. As most of the features we intended to add would provide the user with information that complemented the data displayed on the map, we believed it would improve usability and reduce overly complicated navigation if this information could be accessed and viewed alongside the map. We decided that as an alternative to creating multiple web pages to navigate through, we would add navigation functionality to the side-panel.

The HTML of the webpage was subsequently updated to include the proposed changes and any existing front-end bugs were fixed. During the latter half of Sprint 2 and Sprint 3, the following modifications to the design of the webpage were made:

- A container was added to the left-hand side of the webpage as suggested by our Product Owner. At this stage of the project, this was a simple HTML div element which we used to trial/dump the output of any features we were working on.
- The station markers were colour-coded to indicate bike availability levels, with red representing no availability, yellow representing moderate availability and green representing high availability.
- A station selector dropdown was added to the main navigation bar.
- The "Journey Planner" page plan was scrapped and instead we planned to add a brief project overview as a second page. At this point, the second page was also used as a test space for new features.

Finalising and Implementing the Design

Most of the work on the User Interface took place during Sprint 4. At this stage we had decided on any additional features we wished to add to our application and implementation of these features was either complete or underway. We had modified our wireframe to incorporate these features (see revised design and side-panel mock-ups below).

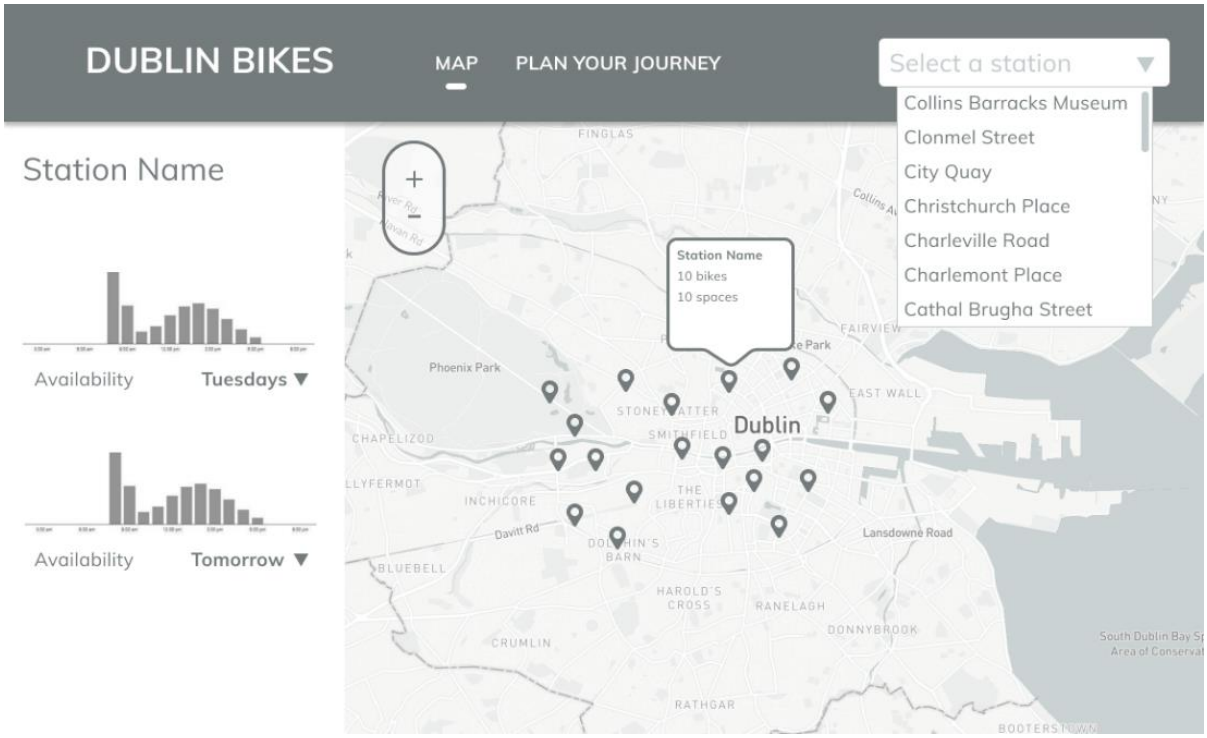


Fig. Updated wireframe following meeting with Product Owner on 4th March

We planned the four following views for the side-panel, mock-ups of which can be seen below.

- A default view (Fig. **X**) which would be visible on page load. We considered adding a summary of the average availability statistics for all stations to this view, but ultimately decided against this to avoid cluttering the homepage and bombarding the user with information. The utility of this information was also deemed dubious. As an alternative, we planned to display the current time and date, a brief textual description of the current weather conditions in Dublin and a visual depicting these weather conditions. We believed this would create a cleaner User Interface improving first impressions of the web application and would aid the user in planning their journey.
- A Charts tab (Fig. **X**), where we would display visualisations graphing the average availability for a selected station. The dropdown station selector would be moved to this section from the main navigation bar. We had initially intended to include three visualisations, graphing the average hourly, daily and weekly availability, as per the mock-up below, but we decided against including weekly averages, again doubting the use of this information to the user and aiming to minimise clutter. We planned to add

two additional input boxes to this section which would enable the user to input a date and time and on doing so they would be brought to the Predictions tab to view predicted availability for that station, date and time.

- A Predictions tab (Fig. **X**), where a brief textual description of the predicted availability for the selected station, time and date would be displayed. This tab would be accessed via the Charts tab. When planning the design, we had not yet decided if it would also be directly accessible.
- A Nearest Stations tab (Fig. **X**). This would include a text input box, into which the user could enter a location and receive information on the nearest Dublin Bikes stations to that location. This information would be returned as a simple list and would include the station name and the walking distance to that station.

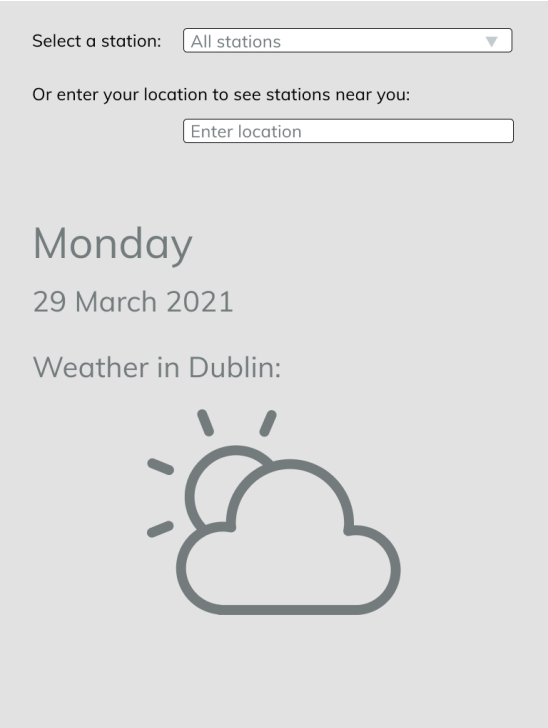


Fig. Default side-panel view

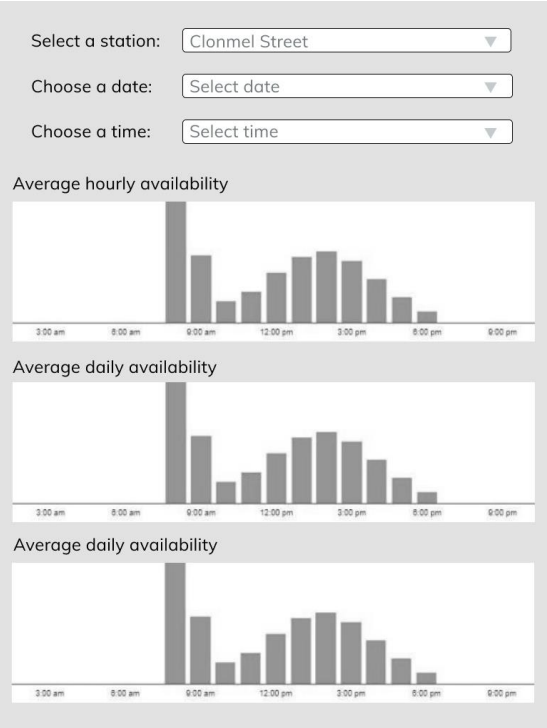


Fig. Charts tab

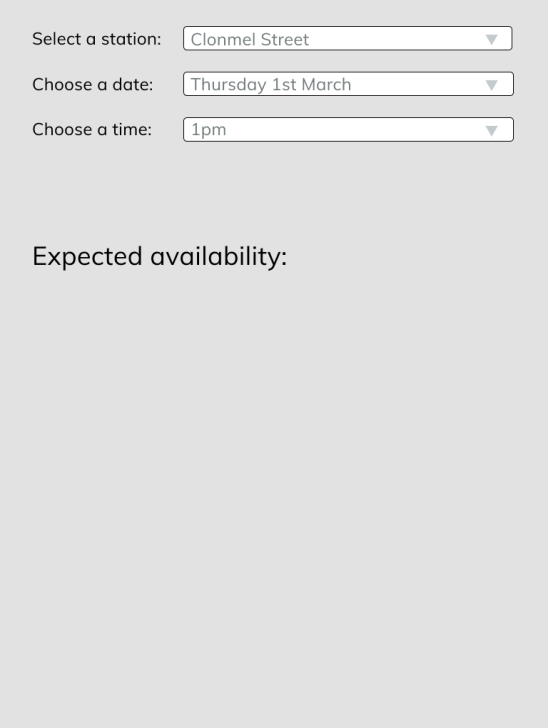


Figure X Predictions tab

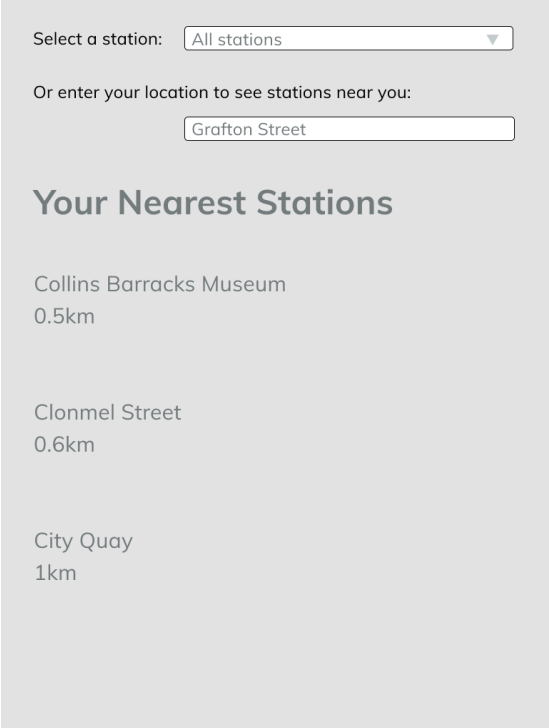


Figure X Nearest stations tab

Each of the features integrated in the side-panel would be linked to the map and on choosing a station from the dropdown list to view visualisations or choosing a station from the list of nearest stations, the info-window for that station would open on the map to indicate the location of the station of interest.

Design Deviations and Final Design

Our final design largely followed that outlined above with the following adjustments:

- Instead of four tabs we had three, with the visualisations from the Charts tab and prediction results from the Predictions tab grouped together into an Availability tab. We felt that transitioning to a separate tab on input of time and date was overly complicated and as the predicted availability was an extension of the average availability data it was appropriate to group this information.
- Availability data i.e. number of bikes currently available and number of bike stands currently available for each station was added to the nearest stations list.

The design was implemented and styled using CSS3. The application was designed for desktop use and while the information contained within each of the tab views of the side-panel was

mostly displayed using CSS Flexbox in an effort to increase responsiveness of design, improving compatibility with different user devices would be more thoroughly considered in future phases of Disappster. The web application was primarily designed for use with Google Chrome and was tested with Microsoft Edge and Firefox browsers. The Google Maps JavaScript API supports the web browsers listed here, as well as Safari and Internet Explorer 11, but cross-browser compatibility beyond this would also need further investigation and testing.

Our priorities in designing the User Interface of our web application were to create an appealing, uncluttered, accessible application and to maximise ease of navigation, to enable the user to find the information they required with minimum effort. In restricting the contents of the web application to one page with easily navigable tabs we believe we largely achieved our design aims.

Name	Type	All Instances	PK	FK	Field Values	Nullable	Description
prediction_id	INTEGER	Yes	Yes	No	1,...	No	Primary Key of Table Prediction number
,'number'	INTEGER	No	No	Yes - station	1....109	No	Station Number
, 'weather_type_id'	INTEGER	No	No	No	400	No	Weather Type
, 'hour'	INTEGER	No	No	No	0,...,23	No	Minute of Update
, 'dayofweek'	INTEGER	No	No	No	0,...6	No	Hour of Update
, 'dayofmonth'	INTEGER	No	No	No	0,...,31	No	Day of Week of Update
, 'bool_weekend'	INTEGER	No	No	No	FALSE	No	Is the day a weekend?
, 'bool_dayoff'	INTEGER	No	No	No	FALSE	No	Is the day a weekend or bank holiday?
, 'bool_workhour'	INTEGER	No	No	No	TRUE	No	Is the hour during work hours?
, 'bool_commutehour'	BOOL	No	No	No	TRUE	No	Is the hour within an hour before or after work hours?
, 'bool_night'	BOOL	No	No	No	FALSE	No	Is the hour during the night?
, 'available_bikes'	BOOL	No	No	No	1,...	No	Available Bikes Predicted
, 'weather_temp_feels_like'	BOOL	No	No	No	283	No	Temperature Feels Like
, 'weather_temp'	BOOL	No	No	No	283	No	Temperature Actual
, 'weather_humidity'	REAL	No	No	No	1000	No	Humidity
, 'weather_air_pressure'	REAL	No	No	No	1000	No	Air Pressure
, 'user_date'	BIGINT	No	No	No	1614101644	No	Timestamp Entered by User for Prediction
, 'created_date'	BIGINT	No	No	No	1614101644	No	Timestamp of Entry to DB
, 'predicted_date'	BIGINT	No	No	No	1614101644	No	Timestamp of Model Prediction
, 'model_type'	VARCHAR(100)	No	No	No	xgboost/sklinear	No	Type of Model Used
, 'bool_correct_prediction'	BOOLEAN	No	No	No	TRUE/FALSE/NU	Yes	Was the prediction right - subsequent update once datetime.now()>user_date
,'minute'	INTEGER	No	No	No	0,...	No	Minute of Update
,'month'	INTEGER	No	No	No	0,...,11	No	Month of Update
,'year'	INTEGER	No	No	No	1970,...	No	Year of Update

Data Analytics

Introduction

As part of the COMP30830 module, a machine learning component was required in order to predict station availability info given:

1. The Weather Forecast at that time.
2. The Time the user wished to make the journey.

To examine the initial approach which should have been taken towards the problem, research was initially conducted into similar problems documented in the literature to gauge an initial direction in terms of model development for bike sharing applications. Initial research pointed heacily in the direction of XGBoost as a model which has been a powerful contender within the space of predicting bike availability data in bike sharing systems (e.g. https://ieeexplore.ieee.org/document/9375958/ and https://core.ac.uk/download/pdf/36739381.pdf with the latter demonstrating a particularly strong case in the Kaggle 2014 winning solution).

XGBoost offers some strong advantages over other methods, particularly focusing on its speed, and the author's familiarity with XGBoost in the deployment of production models in the luxury retail and global tax compliance report sectors, where XGBoost provided a performant and insightful model. The implementation of a time series model was also considered, however based on the author's lack of exposure to these methods, and based on evidence that XGBoost was a strong performer in the area of bike sharing services, the decision was made to focus analysis on XGBRegressors compared to a simple Linear Regression Model.

A key unknown in the initial scoping of this problem was the effect of COVID-19 on bike availability data, and whether this would create unusual distributions in the data which would be challenging to account for; particularly as circumstances change and level 5 conditions loosen, it remains to be seen how applicable our data will be for the 'new' normal.

Data Exploration

Our data exploration in truth began when we initially extracted the API data to understand how our schema should be driven, and as such we had a strong understanding as to what sort of data we had access to, what unique values were present, and what sort of data cleanliness there was. This exploration is briefly touched on and captured within our Data Model section wherein we outline details on the data and identified what sort of values are present in each row, and what the data looks like. For this reason, this section will focus solely on Data Exploration in the context of model development.

Based on the project primarily occurring from February to April during Level 5 lockdown, my initial hypothesis was that time and location plays a much more significant role in determining what sort of distribution the availability data falls under than weather would within this time. The weather aspect in particular is complicated by the easing of restrictions in April as the weather improves likely resulting in a potential long-term biasing of our data. For this reason, I've paid particular attention to potential time features which may play a role in the availability. As a first step, I created a correlation heatmap of all the potential elements impacting the available bikes to identify if anything stood out. Temperature, pressure, and humidity appeared to be the most significant weather features on an initial examination, but, the key relations were observed between available bikes and station number.

Fig: A snippet of the correlation heatmap (full file available in Jupyter Notebook - Available Bikes line is present but too large to insert)

To investigate the relevance of time on the availability, I added on numerous time features (dayofweek, hour, month, year, minute, a weekend flag, a commute hour flag, a night flag, a day off flag) to help examine the relationship between time and availability. As station number was a significant factor in the availability, I decided upon an approach of an individual

model per station (however, if time and my laptop allowed, I suspect a weekend/weekday per station model would be highly significant in improving accuracy).

Feature Addition Exploration

We check if our hypothesis on impacting factors driven by knowledge of footfall distribution in South Dublin City Centre is well-founded.

Day of Week

While most of the weekdays seem to follow a similar distribution, there's a significant drop in the available bikes around mid-day particularly on weekends. This is highlighted clearly in the averaging of weekdays.

Day and Night

The day and night difference is, unsurprisingly, substantial as what I have classified as 'night' is largely dominated by the bikes being unable to be checked out.

Commute Hours

The commute hours are unsurprisingly highly indicative of the peak and rebound of bike availability during the mid-day decline. Given what we know about the days of the week, and the night hours, a more sophisticated would be to examine the rate at which bike availability decreases within this station and use the rate of change to predict where within the pre-work/lunchtime/post-lunchtime rush a time is, and use that as a feature.

Geofencing

Although this specific analysis only looks at the existing data and simple checks on the data, based on observing the availability data within certain time periods and how that is distributed throughout the city, I believe a more refined model would calculate the distance of a station from the city centre and classify this info groups of e.g. 1 to 5 based on the percentile difference between the closest station and furthest away.

A similarly more advanced method would be to observe that in stations surrounding the canal along the perimeter of the city, there are areas here which maintain a relatively high level of availability, while slightly inwards there are areas with a much lower average availability percentage. My immediate assumption is that within COVID Level 5 rstrictions, there are

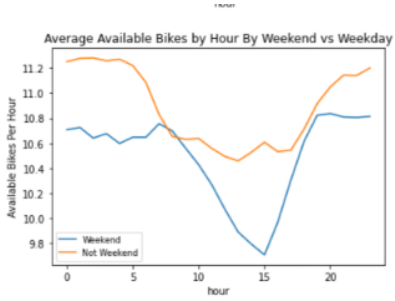
certain key areas just outside of the city centre where either people are more likely to still be in-office and this is driving this pattern (e.g. Smithfield/Heuston, Harcourt Street, and Ballybough/Phibsborough are more frequently under 25% capacity compared to some of the more inner-city areas, while the outer-city areas not following this apparent pattern). Unfortunately, while this would be an interesting aspect to explore the time commitment to add geofenced clusters to the model is likely not going to have sufficient gain to justify the time.

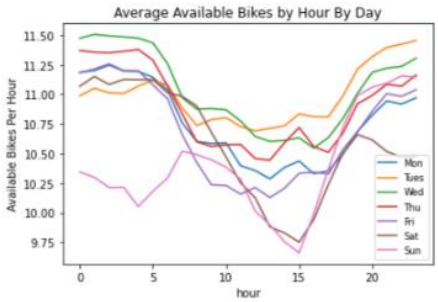
Conclusion

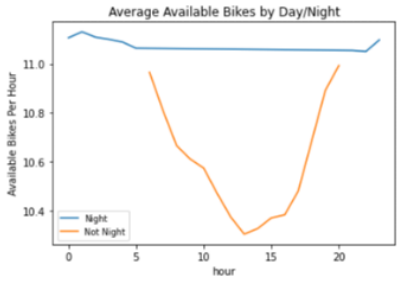
Based on these features, they should be included within the model.

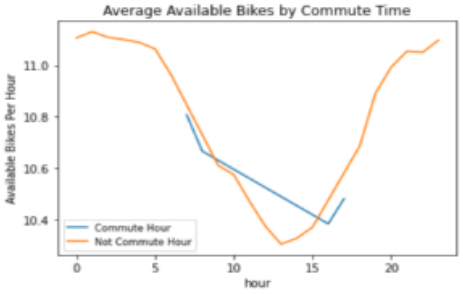
Aggregation vs Raw Data

One question which initially was unclear was whether it would be better to aggregate data into averages over hours rather than keep it in a more granular form. While initially I completed Git Pushes exploring this (largely driven to avoid overfitting and to align more closely with our weather forecast frequency, I ultimately decided to revert to a more granular data approach).









Test Data

Before investigating the implementation of a model, a key challenge was presented in the selection of testing data.

Due to the nature of this task, whereby we wish to forecast future availability based on present data, a careful selection of training data is required. If you simply choose a random split of training and test data, you will likely overfit your model and be left with a very poor understanding as to how it will perform. For example, if you create a random split on the entire dataset, your test data could be overly biased towards certain stations as there is no guarantee of an even split. Similarly, if you try to avoid this by first splitting by station, and then randomly splitting, you end up with the situation where the data could be more/less biased towards certain times of the day. Ultimately, the training process will not be a valid reflection of what the 'actual' requirement of the model is.

As we are forecasting, the test split was established by determining what date lies at the 80th percentile (this was adjustable, however to most properly train my model I elected to use 80% train rather than 70% train due to the highly in-flux nature of the data), and taking all data after this date as test data.

This is a more true-to-life reflection of the scenario, and while the Jupyter Notebook performance of the model would likely be poorer than just doing a random split and gauging the accuracy of that, I believe this result is more reflective of the real performance of the model than that would be.

To record what data was used as training and test data, this was populated into our training and test tables within the database.

Model Selection

As discussed, based on research and familiarity I had an initial leaning towards using an XGBRegressor (particularly based on the plot feature importance and the ability for XGBoost to deal well with 'raw' data), however in order to determine whether this choice was valid I initially tested this on a single station and compared the performance of a SKLearn Linear Regression model and an XGRegressor module. In order to optimise the performace of the

XGBRegressor module I've run GridSearchCV with various hyperparameters in an attempt to optimise the model's performance for each station. This results in a significant hit in the time required to run the models (between twelve hours and twenty-four hours depending on my laptop) however as this is running locally and is not automated in EC2 this is not too big of a problem. If an automated deployment was needed, the optimal hyperparameters per station should also be dumped in addition to the model itself (or added to a hyperparameter table in our RDS instance) and these should be imported as part of an automated model retraining procedure.

After using GridSearchCV, the predicted availability date was charted against the actual availability data, the model training score and accuracy was printed, the RMSE for the standard regression model and the XGBoost model was displayed, and a feature importance tree was charted for our models (please note these graphs are significantly large and appear squashed when posted in a report but are visible within the Jupyter Notebook and the figure size can be increased if too small):

Universal Data

```
Total dates: 14748
Test dates: 2949.6000000000004
Test index: 11798
```

XGBoost Data

```
Model Training Score: 99.77896165897533%
Model Accuracy: [86.24483597 81.53423646 97.66978279 96.70319413 92.88174445
97.85373959
 97.84905662 98.39905338 97.21219571 92.82513889]
```

```
The Original Vs Predicted Result Is:
```

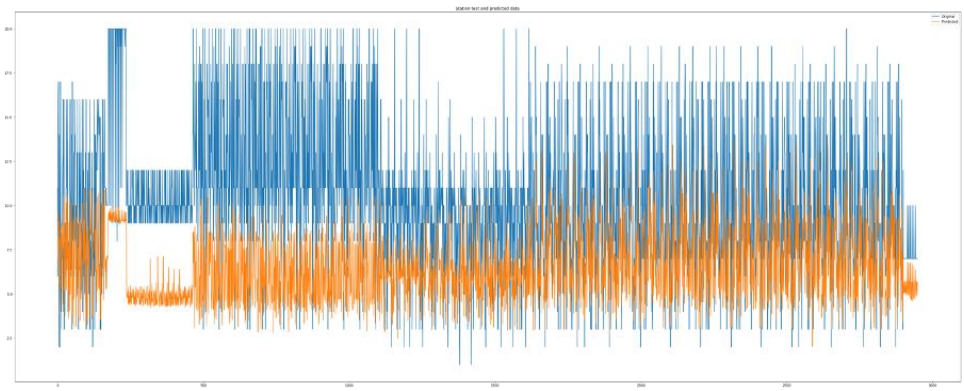
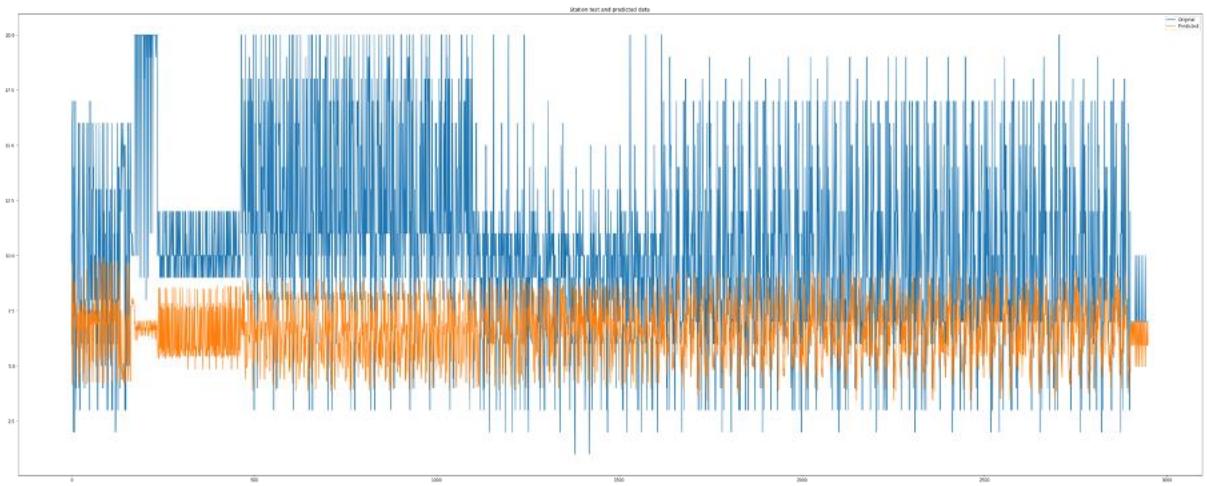


Fig. Prediction vs Actual for Station 4 for XGBoost (orange is predicted, blue is actual)

```
RMSE: 5.458065620460569
```

Fig. Feature importance tree for XGBoost

SKLearn Linear Regression



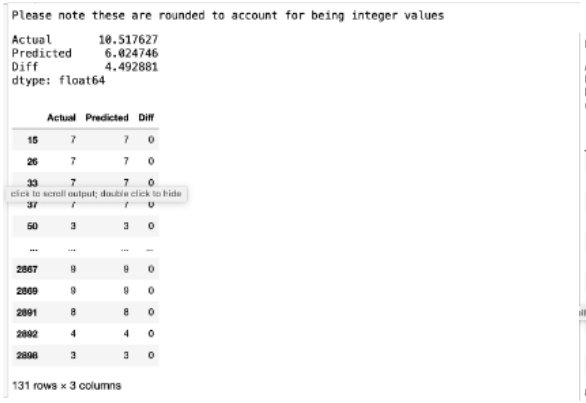
```
RMSE: 5.776783040250935
```

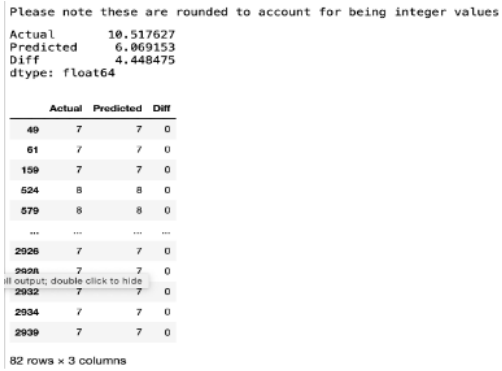
Output

As part of the model generation process, the actual vs prediction is outputted into a dataframe, as are various stats on the performance and hyperparameters used to receive the result above:

{4: {'model': XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1, colsample_bynode=1, colsample_bytree=0.7, gamma=0, gpu_id=-1, importance_type='gain', interaction_constraints='', learning_rate=0.07, max_delta_step=0, max_depth=8, min_child_weight=4, missing=nan, monotone_constraints='()', n_estimators=500, n_jobs=4, nthread=4, num_parallel_tree=1, random_state=0, reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=0.7, tree_method='exact', validate_parameters=1, verbosity=None), 'score': 0.9977896165897533, 'results': array([0.86244836, 0.81534236, 0.97669783, 0.96703194, 0.92881744, 0.9785374 , 0.97849057, 0.98399053, 0.97212196, 0.92825139]), 'predicted_vs_actual': Actual Predicted Diff 0 11 6 5 1 6 7 -1 2 16 9 7 3 17 8 9 4 7 9 -2 ... ... 2945 7 5 2 2946 7 5 2 2947 7 4 3 2948 7 5 2 2949 7 5 2 [2950 rows x 3 columns], 'rmse': 5.458065620460569, 'lin_model': LinearRegression(), 'lin_score': '', 'lin_results': '', 'lin_predicted_vs_actual': Actual Predicted Diff 0 11 7 4 1 6 9 -3 2 16 4 12 3 17 8 9 4 7 6 1 ... ... 2945 7 5 2 2946 7 5 2 2947 7 6 1 2948 7 5 2 2949 7 5 2 [2950 rows x 3 columns], 'lin_rmse': 5.776783040250935}}

By displaying information on these results, we observe:

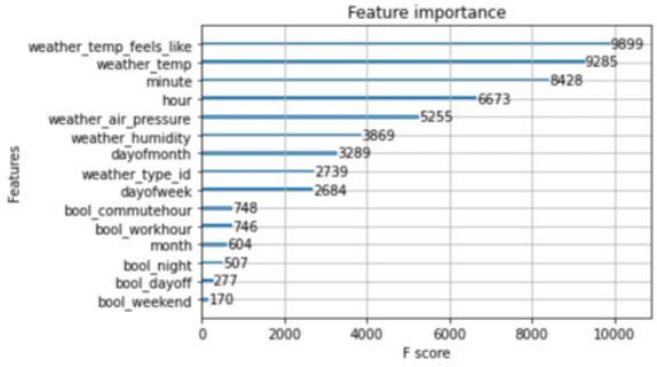




After rounding the actual and predicted availability to integer values an getting the difference between actual vs predicted, XGBoost is exactly right in 131 test vases compared to SKLearn in 82 test cases. The mean difference of these rounded results is slightly worse for XGBoost vs SKLearn (4.49 vs 4.45) however for XGBoost we can see that for Station 4 we the key features with importance to the model by weight are day of month, air pressure, day of week, humidity, the minute, the feels-like-temperature, nighttime, and workhour (by F-score the weather features are more prominent in importance). Arguable some of these features could be removed due to a confounding effect (e.g. feels-like is highly correlated with the actual temperature – potentially only one of these should be present).

```
Importance by Weight:

Out[85]: {'dayofmonth': 3289,
 'weather_air_pressure': 5255,
 'dayofweek': 2684,
 'weather_humidity': 3869,
 'minute': 8428,
 'weather_temp_feels_like': 9899,
 'bool_night': 507,
 'bool_workhour': 746,
 'month': 604,
 'hour': 6673,
 'weather_temp': 9285,
 'weather_type_id': 2739,
 'bool_commutehour': 748,
 'bool_weekend': 170,
 'bool_dayoff': 277}
```



This ability to identify what features are important for each station is ultimately what has resulted in XGBoost being chosen as the default model as it allows for more insight into how the model operates, particularly when both methods are so close in overall results.

As this process is incredibly RAM intensive, and time-consuming, station one is being left as a sample and the Jupyter Notebook of the analysis of this station in particular is being pushed to the Github for reference as is the full generation notebook – these cannot be pulled to the EC2 as they are very memory intensive.

Process

This section details our sprint process.

Sprint 1 (08/02/2021 – 28/02/21)

Context

We commenced Sprint 1 on 8th February and decided on an end-date of 28th February. Due to the completion of the SRS and other assignments at this time, key work on the project commenced on 16th February, with the prior week (8th – 15th) primarily involving the establishment of a management structure and gaining agreement on work processes which

we would follow throughout the project. Initially, this sprint was anticipated to end on the 21st February, however due to the SRS completion this end-date was moved to 28th February.

Within this sprint, we decided that each team member would establish their own EC2 and RDS instance, and develop a scraper so that each member of the team would have familiarity with both the backend of our to-be application, and to develop an initial familiarity with pulling data from an API, gain familiarity with SQL Alchemy, and gain experience with various aspects of the project.

Targets and Progress

Our overall goals within the sprint were:

1. We want to have the Project Management Documentation Setup and Complete.
2. We want to be set up on each platform (GIT, Jira, Confluence, and AWS).
3. We want to begin pulling and storing weather and station data.
    a. To achieve this, we want to understand the data which is present in both tables.
    b. We want to establish a schema to capture the info which will allow for the easy fulfilment of future requirements (namely the ability to link weather postings with the availability info).

Workshops

Within this sprint we had two meetings with our product owner, and an initial meeting in which we completed pre-planning and a broad project outline.

Our initial team meeting occurred within a workshop on 13th February where we reviewed the project requirements, discussed the broad timelines we would like to meet, settled on a management structure (e.g. the usage of Jira and Confluence), established a daily stand-up schedule, engaged in a pre-planning session for Sprint 1, and established Discord as a means of primary communication throughout the project. Within this workshop, we established who would be the scrum-master for this sprint.

Our first workshop with our product owner occurred on 18th February. In this workshop we provided the update that we were each working on a scraper individually as an initial action,

provided a demo of our Jira and Confluence platforms, discussed the requirements we had in mind for the project, and briefly outlined what we saw as the project roadmap:

1. Sprint 1 would focus on data capture for the scraper and project management structures
2. Sprint 2 would focus on a Skeleton Flask Application and basic front and backend functionality.
3. Sprint 3 would focus primarily on frontend features and model development.
4. Sprint 4 would focus on deployment, additional features, deployment of the application, and any fine-tuning of our model.

After this workshop, we created additional tasks for the JIRA board focusing primarily on the weather data components.

Our second workshop occurred on 25th February. In this workshop we detailed where each team member was with our scraper (most of the team either complete or adding in the weather data as the final step), detailed that we had all setup and established a database and were populating it with info, had updated the documentation on Confluence where appropriate (ETL process, Table Info, Integration details), and were on-track to meet Sprint One goals. We discussed the key plans for sprint two and that our end goals should involve:

1. Setting up a flask application.
2. The creation of a frontend which displays a map.
3. The continuation of our documentation and project approach,

Platform Setup

Following the lecture format, we each established an instance on Amazon AWS and an Amazon RDS instance. We each set up an individual Github instance and applied for API keys for JCDecaux for the purposes of pulling bike data. We elected to use OpenWeatherMap as our weather API provider, and each team member acquired an API key.

In addition to the platform setup, a team member created a Confluence Board and Jira Board and each member of the team signed up and was added to the project boards.

By the end of the sprint:

1. Each team member had an AWS Account, an EC2 instance, and an RDS instance.
2. Each team member had an individual Github.
3. Each team member had acquired an API key from JCDecaux for the purpose of pulling Bike Data.
4. Each team member had an Atlassian Account and access to our Jira and Confluence boards.
5. Each team member had an OpenWeatherMap API.

Project Management Setup

Our initial focus of the sprint was in the establishment of our project management system. As one of our team members had experience using JIRA and Confluence, we elected to use these tools to track the documentation of our project – JIRA would be used to track our tickets and progress, while Confluence would be used for the documentation of our project both in regards to detailing the technical implementation of application, and in documenting our daily stand-ups and workshops.

We began by establishing a Confluence structure detailed in the table below:

Page Group	Description
Product Roadmap	This section details the list of tickets which have been established within a project as per the JIRA Roadmap Macro, and a view of the timeline set for each ticket by Epic and Task ticket types.
Product Requirements	This section details the product requirements as detailed in the assignment document.
Team Member Contact List	This section details the list of each member in the group, and contact details for those members in addition to their JIRA and Confluence Tag.
Assignment Submissions	This section will contain a copy of the assignment submission which was

	completed following the end of the assignment.
Sprints	This section details information on sprint goals.
Meetings	This section contains records for meetings with our product owner, and tracking our Daily Standup logs.
Flask App	This section details the folder structure of our Flask Application.
DevOps	This section provides information on our DevOps process.
Integrations	This section details what integrations are present within the app, and the connection parameters which we are using (e.g. API keys).
Data Model	This section details the data model for our application.
ETL – Data Flows for Integrations	This section details a combination of our ETL processes and how data flows within the application. In particular, it contains an ER-diagram combined with an integration flow.
Design Decisions	This section documents decisions which we made in regards to the applications design.

Table: Broad structure of the Confluence Setup

Most of the setup of these pages, and the creation of templates for our Sprints, Daily Standups, Workshops, and Integrations commenced within Sprint One, and pages were updated where and when appropriate.

To document our tasks, we decided to create Epics which comprised of user stories, and subdivided each of these user stories into various Tasks which were then assigned to group

members. Where appropriate, Epics and Tasks were assigned to sprints, and the construction of these epics and user stories occurred during sprint pre-planning sessions.

For communication, we established a Discord server and decided that we would complete Daily Standups from 8:45am to 9am in which we would chart our work in progress, what was completed yesterday, and if we had any key blockers which required resolution. We decided that using Google Meetings would be an appropriate forum for daily standups so that we could add it to our calendars going forward. Similarly, we elected to use Google Meetings as the forum for our weekly meetings with the product owner.

By our first meeting with our project owner, both Jira and Confluence were established with Epics and Tasks created for Sprint One, and a broad documentation structure with templates for various page types which we used for the remainder of the project.

By our second meeting with our product owner, we gained agreement that we would continue to follow the practices on Jira and Confluence which we were already following.

By the end of this sprint:

1. We had successfully established a JIRA board with Epics and Tasks relating to Sprint 1
2. We had established Confluence board with the page structure described above for documentation of the project.
3. We had decided we would use the 'FRogers98' Github account as a unified Github account for the project for later sprints.
4. We had established a Discord server for open group communication.
5. We had created recurring daily and weekly standup invites in Google Meetings.

Station and Weather Scraping

The key development goal during Sprint One was the acquisition and accumulation of station and weather data.

We decided that to gain familiarity with the data and API scraping we would each develop an individual station and weather scraping process, and engage in a code-merge at the end of the sprint to decide upon a final scraping process to use. We also agreed that we would falling a standard ETL process of extracting and storing raw data, transforming this within our

application, and passing this back to the user, rather than manipulating the data from the API as it enters the database.

We began by creating a process to pull the station and availability data from JCDecaux's API. Before establishing a schema, we created functions to pull the data via the API, and store this as a dataframe in order to begin an initial data exploration. This involved identifying aspects such as the types of data, the cardinality of the columns, and a brief understanding of what data was being captured within each of the columns.

After identifying what data was contained within the JCDecaux data, we decided upon a split of the data into static station data, and availability data (in keeping with the lectures). As part of this data exploration, we quickly identified that apparent 'duplicates' were present within the availability data due to irregular updating of station availability info. Based upon the format of data, we identified that station data was updating in real-time if an update to the available bikes/stands had occurred, otherwise, if the availability info remained static, there was an interval following which station data would be flagged as having been updated. Due to this investigation, we decided that on all tables we would add a created_date column which would document the date that the row was posted into the database. This served a dual purpose in allowing for auditing of when info was posted into the database, but also allowed us to use an alternative to the last update date within the API (with the logic being that if the station availability info had changed, JCDecaux's system would have flagged an update; otherwise, the station info must not have changed since the last scraping occurred, and thus the availability info at that time is the same as the previous update). This initial investigation played a key importance, as it dictated that throughout the project our availability info would use the created_date column to identify the availability at each run of our scraper rather than JCDecaux's date_updated column.

As we decided that all tables should have this auditing column, and as we determined that the most sensible design of our application was to use this in-place of the last_update column, we identified that this served as a natural link between the availability and weather info so long as each update within a time-step contained the same update time.

We also identified that the position info returned by the API was a dictionary containing latitude and longitude. For ease of access going forward, we decided to split this into two fields.

Following this data exploration, we had identified a target schema for our station and availability tables, and documented these within the appropriate Confluence pages (with some high level details of our analysis for reference) per the example for Station below:

It captures bike stand numbers.

Station Table

Dublin Station Mapping

	Name	Type	All Instances Unique?	PK	FK	Field Values	Nullable	Description
1	number	INT	Yes	Yes	No	1, 2, ... , 109	No	PK of Station
2	contract_name	varchar(6)	No	No	Yes	dublin	No	Denotes JCDecaus's contract **dublin**
3	name	varchar(256)	Yes	No	No	'MATER HOSPITAL'	No	Name of Station
4	address	varchar(256)	Yes	No	Maybe	Wolfe Tone Street	No	Address Name
5	position_lat	REAL	Yes	No	Maybe	53.333653	No	Position - Latitude
6	position_long	REAL	Yes	No	Maybe	-6.248345	No	Position - Longitude
7	banking	int64	No	No	No	True/False	No	??? - Determine Meaning
8	bonus	int64	No	No	No	True/False	No	??? - Determine Meaning
9	status	varchar(256)	No	No	No	Open/Close	No	Open or Close - Identify meaning
10	created_date	BIGINT	No	No	No	Date of Last Update	No	Timestamp updated

Note

Number is a primary key.

Summary Data

	count	unique	top	freq
contract_name	109	1	dublin	109
name	109	109	MATER HOSPITAL	1
address	109	109	Wolfe Tone Street	1
position	109	109	{'lat': 53.333653, 'lng': -6.248345}	1
status	109	1	OPEN	109

	count	mean	std	min	25%	50%	75%	max
number	109.0	6.033028e+01	34.021250	2.000000e+00	3.100000e+01	6.100000e+01	9.000000e+01	1.170000e+02
bike_stands	109.0	3.211009e+01	7.683966	1.800000e+01	2.800000e+01	3.000000e+01	4.000000e+01	4.000000e+01
available_bike_stands	109.0	2.025688e+01	9.019041	0.000000e+00	1.300000e+01	2.000000e+01	2.800000e+01	3.800000e+01
available_bikes	109.0	1.111009e+01	5.276926	0.000000e+00	8.000000e+00	1.100000e+01	1.400000e+01	2.400000e+01
last_update	109.0	1.613474e+12	178619.620555	1.613474e+12	1.613474e+12	1.613474e+12	1.613474e+12	1.613475e+12

Header Summary:

	number	contract_name	name	address	position	banking	bonus	bike_stands	available_bike_stands	available_bikes	status	last_update
0	42	dublin	SMITHFIELD NORTH	Smithfield North	{'lat': 53.349562, 'lng': -6.278198}	True	False	30	8	22	OPEN	1.613474e+12

Quickstart

After agreeing on the schema for these tables, we established a function which would setup our database and we established a unified schema.

Our next step was to develop the functionality to save this info into our database which involved little difficulty for our team, but while working individually we tackled this via various approaches (e.g. SQL Alchemy, explicit SQL).

The final stage in the capturing of station and availability info was to deploy a module onto our EC2 instances which would allow for the scheduling of this process every five minutes. By our first meeting with our product owner on 18th February, we had each completed these elements within our individual scrapers, and had gained agreement that we were on-track.

After completing the capture of station and availability info, we subsequently investigated the capture of our weather info. We identified that OpenWeatherMap's free API tier would allow 1 million API calls of real time weather data over a month, and had weather info segmented across Dublin (i.e. different regions within Dublin would return different results). We calculated that as we had 109 stations, and as we were performing one call per station every 5 minutes (31,392 calls per day), assuming a month contains 30 days, we would be performing 941,760 calls per month if we called the weather data for each station every five minutes. Based upon this calculation, we decided OpenWeatherMap would be sufficient for our needs within this assignment.

We followed a similar process to the station and availability info (data exploration, schema documentation and agreement, function development) for the OpenWeatherMap data, and embedded the weather capture and storage function into our station and availability process to allow for a natural linking of weather info with availability info via the created date and station number.

By the end of the sprint:

1. We conducted a code review of each of our scraper processes and functions, and decided upon a scraper and schema (i.e. naming structure) which we would use for our development going forward.
2. We decided that we would archive the scrapers within our Github repositories to document this process, and we deployed the final module to our EC2 Instances
3. We determined that in order for us to have redundant databases in the event of a failure, we would each run this module going forward into our own databases and on our own EC2 Instances.
4. We had each established a database.
5. We had each developed a scraper.
6. We were each populating a scraper into the database every 5 minutes.

Retrospective

By the end of Sprint One we had made considerable progress. We had established a strong Confluence and Jira structure for the documentation of our project and had cleared off the key elements within our task backlog. We were each setup on the various platforms with the required API keys. We had the Weather and Bike Data being scraped and stored into our database.

Within our sprint retrospective, we decided to continue using Confluence and Jira after the product owner's comments on the documentation and structure which we had completed. This was a key callout for our first sprint and helped to reduce the overhead of project documentation for the remainder of the project, as establishing templates for integrations, workshops, daily standups, and other elements helped to record what we had done during the project and reduced the effort involved in maintaining this throughout.

The only tickets which remained open following the closure of the Sprint were those which concerned documenting the weather data and weather table.

Reflection

During Sprint One, strong progress was made which helped established some good principles for the project. In particular, the early establishment of Jira and Confluence helped significantly in the documentation of our project and in creating a clear direction for each sprint. The documentation of the daily stand-ups and workshops in particular were very helpful in directing our sprint pre-planning and and sprint retrospectives.

Similarly, our progress in establishing the weather, station, and availability date early and the initial exploration of data which we were receiving from the APIs paid off significantly in helping us to come to the design decision of using the availability and weather posted date as our key column to allow an easy and natural link between the availability data and weather data which made it very easy to subsequently extract these in a joined form.

In hindsight, while Sprint One went quite well, I would have preferred not to have developed the Weather/Station/Availability table outside of the context of the Flask application. Although the scope of the project was targeted towards a smaller application, I believe taking a more 'pythonic'/'django'-like approach of implementing a MVC-backend design structure

would have helped gain additional context in the construction of industrial web applications using a common design framework. By establishing these tables as models, and implementing all interactions with the database through model interactions, I believe this would have helped in the establishment of a schema which was naturally linked and potentially would have been easier in passing data to the front-end later on in the project rather than needing to develop more optimised SQL queries and create an SQL codebase. This was an element which as explored and had been added to the Github repository, however the overall learning curve for the team (in deployments via Alembic and the MVC structure) was ultimately deemed too high during the project to pursue further. However, I believe if we had followed this process from Sprint One this would have been more feasible to explore within the scope of the module while still keeping the applications relatively lightweight.

Working on the documentation structure early was beneficial in being able to reduce the upkeep cost later on, but it might have been helpful to establish an SRS-style document within the early stages of the project to direct development. This would have also been useful in being able to establish Epics which are valid across all sprints so that the pre-planning session for each sprint could have solely involved establishing the subtasks within that sprint and just assigning the Epics to sprints.

Burndown Reports

Our burnup, burndown, and cumulative flow diagrams as of the end of Sprint One are detailed above. The primary dates in which we established tickets on the Jira board were the 17th and 18th February before and after our first meeting with the product owner. Tickets were intermittently flagged as complete, and our burndown report by Jira task type is relatively strong with few areas of issue.

As we elected within this sprint to each work individually in building the scraper, the epic involving the scraper was duplicated for each team member (with the same tasks associated to each epic) to track our individual progress with the scraper.

The only tasks which were not flagged as done within the scope of the sprint were those involving documentation of the weather API data, and this was completed within Sprint Two.

Sprint 2 (01/03/21 – 19/03/21)

Context

This sprint was unique as it initially ran halfway into midterm break, which meant that we only had one workshop with the product owner instead of having them weekly as usual. We continued our daily stand-up meetings throughout the midterm break. Our workshop with the product owner prior to midterm break was more detailed than normal as they outlined what they expected to see from us by the end of the midterm. The agenda for the meeting was not only to discuss what was expected in sprint 2 but also what needed to be completed in the first half of sprint 3 before we would meet again. However, as we neared the originally set end date for the sprint, we took the decision to extend the sprint by an additional week until the end of midterm.

Before beginning sprint 2 we had all managed to get the same harvester running on ec2 to collect data on the bike stations and the weather and store this info in our databases. In regard to documentation, on our confluence page we had added our data model and recorded the previous sprint and we had finished setting up our shared Github repository but still needed to complete our backlog.

Goals

At the workshop we decided along with the product owner that sprint 2 should have two main focuses: flask and the front end. With flask our main aim was to become more familiar with the framework as we were still quite unfamiliar with it. We set the goal to have a skeleton flask app running that could pull relevant data from the database to respond to front end queries by the end of the sprint.

Our goals for the front end were quite a bit more specific and were the main focus of the sprint. The main goals we outlined in relation to the front end were to modify our wireframe to leave more space for visualisations which was requested by the product owner, to decide on a front-end framework (if any), to create a basic user interface involving the google maps api with clickable markers that would display information about that station on click and to have a dropdown selector of all the station names that are linked to the markers on the map. We also set ourselves the optional goal to request the latest data from our availability table

and display this information on the front end if we had enough time remaining in the sprint. We had never used the google maps api before beginning this sprint, so we had to familiarise ourselves with it in a similar way as with flask before building out our app.

Originally this was the last meeting we would have until the end of midterm and sprint 3 was set to begin halfway through midterm so our product owner also gave us some goals to complete for the first half of sprint three during this workshop. These goals consisted of building out our front end even more to make it more structured and interactive and to display some visualisations displaying weekly and hourly availability data for each station. However, as stated previously we ended up extending sprint 2 by an additional week so did not actually start sprint 3 until after midterm.

Retrospective

By the end of sprint 2 we all had a basic functioning flask app and front end using the google maps api. We all had each station displaying as a marker on the map which would display the station name on click or upon selection in a drop-down selector containing the name of every station. We also all had a flask app that was successfully pulling station data from our stations database and passing this data to the front end. We had started working on but not completed our optional goal of displaying the latest info from the availability table in the info window for each station. We also completed an updated wireframe in line with specifications of the product owner.

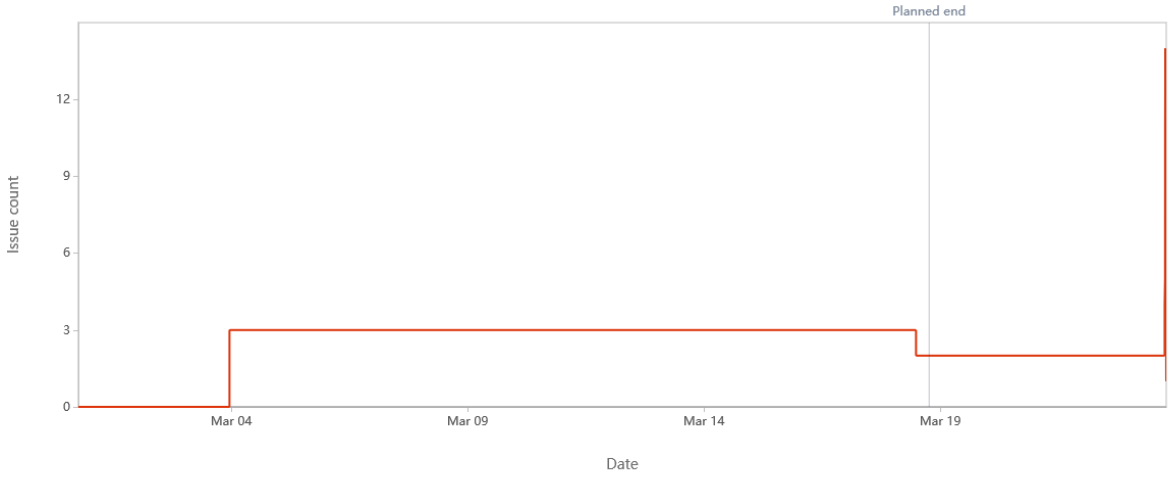
Reflection

This was one of the more challenging sprints with a rather steep learning curve as we had little to no prior experience with flask and the google maps api. This along with significant amounts of work being due for other modules during midterm resulted in us extending the sprint by an additional week. One major thing we learned from this sprint that helped us greatly in later sprints was the importance of collaborating by breaking the sprint into small manageable tasks that anyone can work on. In sprint 1 and sprint 2 we worked on the same things but individually so we would have a broad understanding of everything involved in these sprints. While this helped greatly with getting comfortable with previously unfamiliar technologies such as flask and the google maps api, it also resulted in a much greater workload

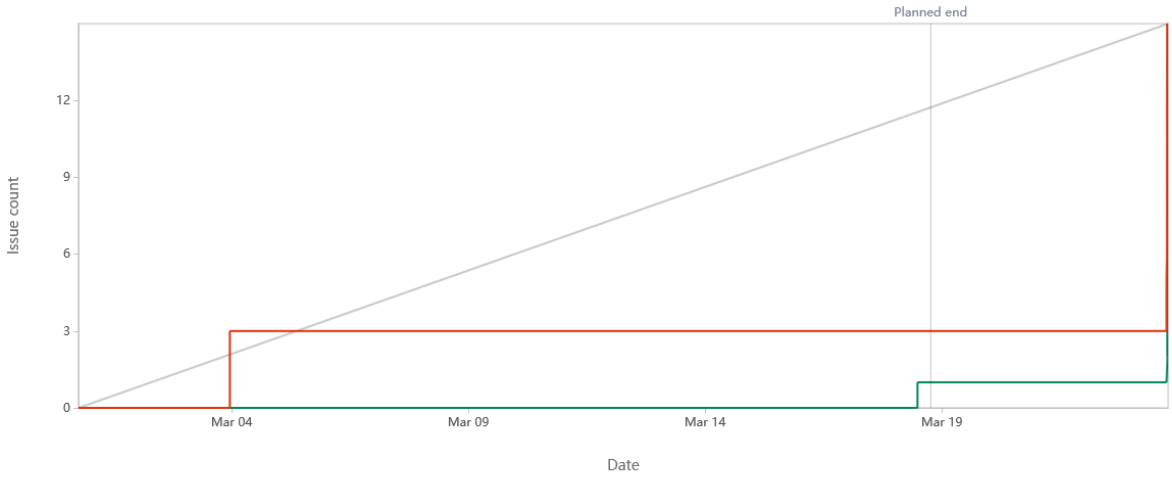
for each member of the team and made it a bit less clear what exactly what we should work on each day. From the end of this sprint on we unified our codebase and utilised our Jira board much more effectively, breaking each sprint into many specific tasks that would take a shorter amount of time than the type of tasks we had previously been using on the Jira board.

It's likely that if we had broken up our Jira board up into smaller, more specific tasks before beginning sprint 2 and were working on a unified codebase that the workload for this sprint would have been much more manageable. However we would likely not all have gained as much familiarity with the full stack and working in this manner did give us all a good foundation of knowledge of previously unfamiliar technologies that we could build off of in later sprints as we broke down the workload more specifically across the team. It was also more difficult to judge our progress throughout the sprint as we only had a weekly workshop with the product owner in one of the three weeks of the sprint because of the midterm break. Our relative neglect of the Jira board in this sprint can be seen clearly when comparing the burndown chart below with burndown charts from other sprints. This is likely largely due to the fact that we missed the structured weekly goals that our weekly meetings with the product owner gave us and the fact that we were working on individual codebases without breaking the Jira board into smaller tasks. You can see in our burndown chart that the vast majority of tasks were marked as complete at the very end of the sprint before unifying our codebase and changing our approach to Jira.

Burndown Reports



Burnup Report



Sprint 3 (19/03/21 – 31/03/21)

Context

We commenced Sprint 3 on Thursday 19th March, during the second week of the midterm break. While Sprint 3 was originally due to begin on Monday of that week, we had been given some flexibility regarding the lengths of Sprint 2 and 3 and as mentioned, we chose to extend Sprint 2 due to the steep learning-curve involved in familiarising ourselves with the Google Maps API and Flask web application framework and a heavy workload in other modules at that time. We held our Sprint 2 Retrospective and Sprint 3 Preplanning meeting on the Thursday in place of our Product Owner Workshop.

For a summary of our progress and the stage of development of the application on entering Sprint 3, see Sprint 2 Retrospective.

Sprint 3 Targets and Progress

At our Sprint 3 Preplanning meeting we outlined three overall aims. The first was to create visualisations using a JavaScript charting library to graph historical availability data and display these visualisations on our web application. In order to do so, we would need to write SQL queries to retrieve the relevant data from our RDS database and create routes in our Flask application to pass this data to the front end for graphing.

Secondly, we would continue to work on our User Interface, adding greater structure, detail and interactivity, including making the following additions:

- A side-panel on our home page, as depicted in our updated wireframe (see Section **X**), which we would use to display our visualisations and going forward, our availability prediction feature. A default view for this side-panel would also need to be designed and created.
- Current availability information for both bikes and stands would be added to the pop-up information-window for each station marker. This would involve querying the database for the most up-to-date availability data for each station and passing this data to the front end.
- Customisation of the basic station-markers we had added to our map to give the user an instant overview of current availability for each station using a traffic-light colour-coded system of green for high availability, yellow for moderate availability and red for low availability. We discussed the potential addition of radio buttons to allow the user to filter the markers displayed by level of availability and by available bikes versus available stands.

The third major task outlined for Sprint 3 was to begin the development of a machine-learning model that would predict future availability for each station, allowing the user to plan a journey in advance. We intended that the design and implementation of this model would dominate the second half of Sprint 3, but we would begin researching appropriate learning methods while working to clear the Sprint 2 backlog and adding the features outlined above.

We hoped to have made significant progress with the first two tasks by the following Thursday when we would meet our Product Owner again after a two-week break. Some of these tasks, such as the retrieval of availability and last-update data from the database and appropriate rendering of this data on the frontend had been included in our Sprint 2 backlog and we had already begun working on them.

For Sprints 1 and 2 which involved building a web harvester and a skeleton Flask application there had been little division of work, with each team member working to complete each of these tasks. We made this decision as we believed it was important to the successful development of the application that we were all familiar with the underlying structure, and as both tasks presented an important learning experience, involving web development tools and processes that we were largely inexperienced with. While this added to our individual

workloads for Sprints 1 and 2, it made for easier division of tasks in Sprint 3 and Sprint 4 as none of our team members were limited to working in any specific area. The workload also became more clearly divisible as we identified our targets during our Sprint 3 Preplanning meeting. We planned to carry out a code merge in the days following our Sprint 3 Preplanning meeting and from then on to work on separate smaller tasks. For sprints 1 and 2 we had under-utilised the Jira ticketing system as we worked in parallel and there was less need for coordination. However, when creating our backlog for Sprint 3 we broke each of the three major tasks we hoped to accomplish into smaller sub-tasks, each of which we believed would require approximately 1-3 hours work and then created one ticket per sub-task.

We completed our code merge the day following our Sprint 3 Preplanning meeting. We pushed our individual codebases to our GitHub features branch and met to review our code. For any elements that we had developed in parallel such as the database query and route for latest availability data, we chose one version to include in the final application based on the best estimated efficiency and succinctness and modularisation of code. We also integrated any additional features from each of our codebases, creating a single FlaskApp folder on the features branch which would hold the code for our merged application. Following this code merge, we worked by choosing tasks from the Jira board which we were interested in and believed we could commit to completing, and pulling from/pushing to the features branch as we completed each task.

We had made some progress when we met with our Product Owner the following Thursday but had not met all our targets as outlined above. We had colour-coded the station markers to represent availability, added the current number of available bikes as a marker label (later scrapped based on Product Owner preference) and added current bike and stand availability data to the information displayed in the info-window for each station. The info-windows were linked to the station dropdown menu and would open when a station was chosen from the dropdown. As well as the SQL query for most up-to-date availability, all queries required to retrieve historical availability data and any other queries that we envisaged may be useful had been written, and we were working on writing Pandas functions to summarise this data before passing to the front-end. We agreed with our product owner that our priority for the remainder of Sprint 3 would be to create a nice User Interface and create and display the visualisations. This would involve completing the Pandas functions to summarise historical

availability data, using a charting library to graph this data, adding a chart container div to the side-panel on our homepage, moving our station selector to this div and defining a JavaScript function to display and update the charts based on station selection from the dropdown menu or station marker selection on the map.

One issue which we had noticed at this point and which we discussed with our Product Owner was the lengthy loading time of any elements of our web application the rendering of which depended on the completion of a database query. Our application was structured so that database queries were running on page load, to display station markers on the map, and on events (on-click, on-change etc.) thereafter, and we wondered whether there was a better alternative. Our Product Owner assured us that some delay is to be expected as we are connecting to a distant remote server and that performance was likely to improve on deployment to EC2 as requests from EC2 would be communicating with the RDS database within the same infrastructure. Our Product Owner suggested that a simple deployment to EC2 would be worthwhile to investigate the speedup achieved. If still noticing delays we could consider using a static JSON file containing station information to reduce initial load time, simplifying our queries by having a table in our database with recent data only, or making queries to the database or directly to the API from our Flask application at regular predefined intervals, to maintain a regularly refreshed local JSON file with the most up-to-date availability data.

We met with our Product Owner again the following Thursday as Sprint 3 drew to a close. At this point we had created charts using Google Charts to display average hourly bike availability and average daily bike availability, added the charts to the side-panel on our application home page and created functions to update the charts on station selection. We had begun work on additional feature which would allow the user to enter their location and receive a list of their nearest stations using the Google Maps Distance Matrix API. We intended that this data would also be displayed in the side-panel and that the user would be able to navigate between side-panel views using a navigation bar or tabs, rather than adding additional pages to our web application. We had updated our wireframe to demonstrate the planned contents for each side-panel view (see Design section of this report). At the time we planned to have a default view, a tab for viewing charts, a tab for viewing nearest stations and a journey-planner tab for displaying availability predictions for an input station, time and date. We had also

ensured all database queries were optimised in an effort to reduce loading time. We were still noticing delays especially regarding chart loading time but had not yet attempted deployment to EC2. Following further discussion with our Product Owner we decided against making timed background queries to the database and planned to add a loading spinner in the event that loading times were still slow following deployment.

We discussed the addition of optional features with our Product Owner who made some suggestions such as filtering station markers by colour with the colour representing a level of availability (completed), customising the appearance of our map e.g. removing unnecessary landmark labels to reduce clutter, adding a feature to store the last station chosen by the user to increase usability, adding geolocation functionality (related feature allowing the user to enter their location and view nearest stations already in development), exploring responsive design, adding a feature to provide the user with directions to a chosen station, adding a feature to calculate the reduction in CO2 emissions per journey through travelling by bike versus travelling by car and adding weather warnings.

We had hoped to have made more progress into researching, choosing and developing a prediction model by the end of Sprint 3 but we were otherwise on track having completed most outstanding issues in our backlog by this point. We also seemed to be on par with most other groups in the class, which was encouraging. Our priorities following this meeting with our Product Owner and as we entered Sprint 4 were to begin work on a prediction model, continue our work on creating a nice User Interface, consider which additional features we wanted to implement and attempt an initial test deployment of the application to EC2.

Sprint Retrospective

By the end of Sprint 3 we had built out the User Interface of our web application significantly. We had coloured-coded our station markers by availability and added radio buttons to filter the markers displayed by colour. We had added availability information for both bikes and stands to the info-windows for each station. We had added a side panel to hold visualisations, availability predictions and nearest station information, and had added two charts displaying average historic bike availability data per hour and per day for each station, which we had programmed to update on selection of a station from the dropdown menu or on selection of a marker on the map. We had moved the dropdown station selector to the side-panel and we

had begun to add CSS to create a nice user interface. An additional feature which would allow the user to input their location and receive a list of the nearest stations was in development.

Regarding backend development, we had created a bank of SQL queries to pull all data required for displaying current availability data, generating charts of historic availability and some additional queries which we envisaged would be useful in the development of our prediction models. We had developed functions to connect to the database and execute each of these queries and we had created routes to format the data returned and pass this data to the frontend. We had also begun to explore data modelling techniques and libraries, in particular the XGBoost gradient boosting framework.

There were some small outstanding issues from Sprint 3 which would need to be added to the Sprint 4 backlog. We had not yet implemented a feature to allow the user to alter the colouring of station markers to reflect available bike stands rather than available bikes. There were some frontend bugs which would need to be addressed, including removing duplicates from the station dropdown and modifying the CSS to fix overlap of the station dropdown and the availability charts.

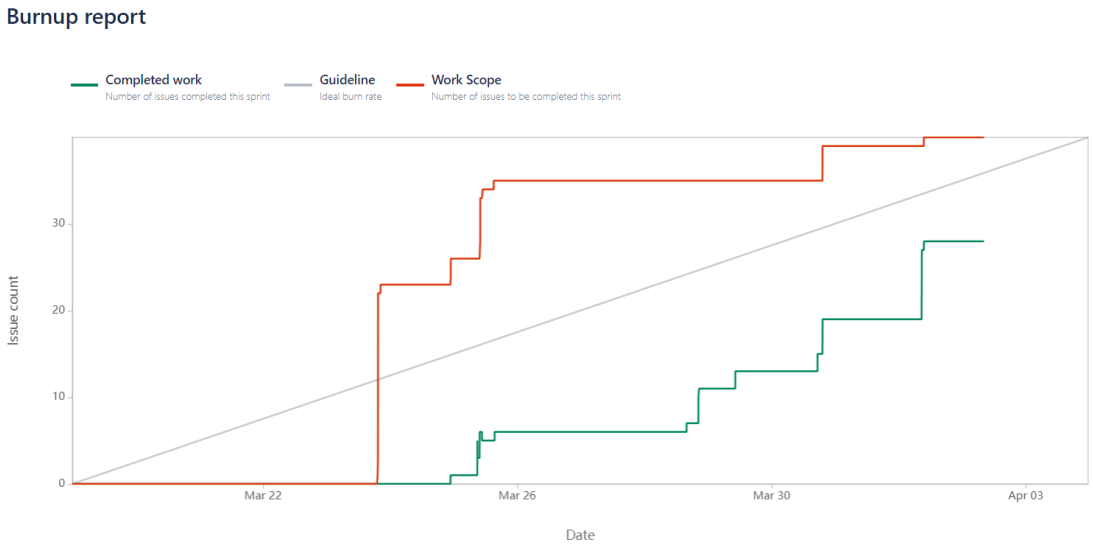
Reflection

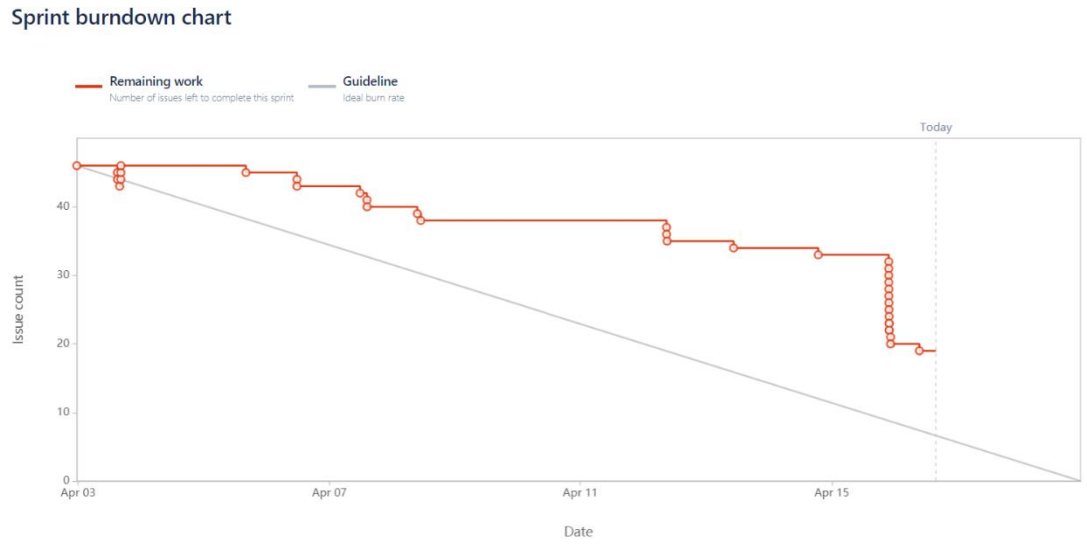
We had a slightly slow start to Sprint 3 as we tried to split our time between working on our web application and working on other modules with similarly heavy workloads. This can be seen by examining the burnup and burndown charts below, which shows few tasks being completed during the first few days of Sprint 3 but then a steady increase in productivity as the sprint progressed. By the end of the sprint, we had accomplished most of the targets we outlined at the beginning of the sprint. The main task outstanding was the research and development of a machine learning model, but as this was covered in lectures only towards the end of Sprint 3 we felt happy to add it to our Sprint 4 backlog. The other outstanding issues, listed above, were either minor issues or ongoing tasks.

As mentioned, our intention was to divide the workload for Sprint 3. This contrasted with previous sprints where we had all worked on completing the same tasks individually. We approached this by creating many tickets representing smaller “bitesize” tasks on Jira during our Sprint 3 Preplanning meeting. We discussed what areas we were each interested in working on at the preplanning meeting and briefly at our daily stand-up meetings and then

assigned the appropriate tickets to ourselves. There were some very minor teething issues at the start of the sprint as we got used to this method of coordinating our work, with some confusion as to who was working on what, but we adopted a regimented approach to updating our individual progress on Jira thereafter and the system worked extremely well. We as a team found that this was a more enjoyable way to work, as long as we met our targets, we could work at our own pace and were neither trying to keep up with others or waiting for others to catch up. It also gave us greater scope to delve deeper into the areas of web development that were of particular interest to us individually. However, the strong foundational knowledge we had acquired by each building the basic architecture of the application during sprints 1 and 2 stood to us and better prepared us to tackle tasks individually in Sprint 3.

Burndown Reports





Sprint 4 (01/04/21 – 16/03/21)

Context

We commenced Sprint 4 on Thursday 1st April, holding our Sprint 3 Retrospective and Sprint 4 Preplanning meeting on the Thursday evening. The meeting was therefore in part a follow-up to the Product Owner Workshop that had taken place that morning. As this was to be the final sprint, we were eager to get started. Our Product Owner had also given us much food for thought that morning regarding potential additional features for our web application and before we could begin working on these we needed to decide as a team what features we wanted to implement.

For a summary of our progress and the stage of development of the application on entering Sprint 4, see Sprint 2 Retrospective.

Sprint 4 Targets and Progress

At our Sprint 4 Preplanning meeting we outlined three overall aims. Firstly, and crucially, we needed to research, develop and implement a prediction model which would use the bike availability and weather data that we had been harvesting over the past month to predict future bike availability for a specific time, date and station location. As one of our team members had experience using the XGBoost gradient boosting framework which had been shown to perform well in similar bike-sharing demand prediction projects (e.g. https://ieeexplore.ieee.org/document/9375958/), we decided to use this software to develop the models, one for each station. We would also need to develop a web scraper to harvest weather forecast data, which we would need to make our predictions. The historical weather data we had collected thus far would be used to identify patterns e.g. availability of bikes is 20% higher at 8am on rainy days, and make appropriate predictions based on the weather forecast data. We would need to schedule the weather forecast scraper to run at predefined intervals and add tables to our database to hold the forecast data returned. We would also need to add tables to split our historical data into training and test sets, in order to evaluate the performance of our prediction models. We would need to create date and time selectors to accept user input, build a function to call the model when user input was

passed and create routes to pass user input to the prediction model and return the prediction data. Finally, we would need to build out how the prediction data would appear on the frontend.

The second priority identified for Sprint 4 was to continue the development of our User Interface. We were planning to use the side-panel on our web application homepage to display prediction data, charts and nearest station data, as outlined in the Design section of this report, but had not yet created a navigation system to allow the user to switch between views. As well as this, we needed to add date and time selectors to allow user input to the prediction models, a text input box to allow the user to input their location in order to retrieve nearest station information, a table to hold nearest station information data and a default view for the side panel. Our intention was that the current time and date and current weather conditions in Dublin would be displayed as default. A route would need to be created to pass up-to-date weather data to the frontend and a function to call this route on page load and display the data returned would also need to be define. Loading spinners needed to be added to the chart containers. The map had yet to be customised and the overall CSS needed to be updated to create a smooth, enticing, user-friendly interface.

The third major task identified for Sprint 4 was to implement some of the additional features suggested by our Product Owner. We had already customised our station markers to give the user an instant overview of current bike availability per station and we intended to add a filter to allow the user to alternate between bike availability and bike stand availability. We were in the process of developing a nearest station locator feature which would enable the user to enter their current location/a location and retrieve a list of the nearest stations. We planned on displaying the current weather data as a default view for the side-panel. We decided that we would also extend the functionality of our nearest station locator to provide the user with directions to each of the stations listed.

In addition to these three principal tasks, we identified several smaller issues to be addressed in Sprint 4 in preparation for deployment. We needed to implement unit tests in our code, remove redundant code and ensure our code was appropriately modularised. We also needed to fix the frontend bugs described in Sprint 3 Retrospective.

The workload for Sprint 4 was evidently already quite significant and in addition to this we needed to write this report. In preparation, we needed to ensure that all design decisions, methods, data flows, routes and tables were documented and that all our project management records were up-to-date. We planned to continue with the approach to task division we had taken during Sprint 3 and which we had found to work well i.e. dividing each major task into smaller sub-tasks each of which we expected to take no more than 2-3 hours work, ticketing these sub-tasks on Jira and working through them one-by-one. We hoped to have developed our machine learning models and have a polished User Interface to present to our Product Owner at our next meeting.

We met with our Product Owner the following Thursday. By this time we had developed and tested a prediction model for each station using XGBoost and linear regression models (see the Data Analytics and Backend Architecture sections of this report for more details) but had yet to integrate these models into our Flask application. We had built a web scraper to harvest five days-worth of weather forecast data, which we intended to run once daily due to the significant runtime associated with inserting forecast data for 109 stations for five days at three-hour intervals into the database. Again, we had yet to integrate this scraper into our Flask application. We had implemented our nearest-station-locator feature using the Google Maps Distance Matrix API and we had extended the functionality of this feature to provide the user with directions to these stations using the Google Maps Directions API. We had yet to add navigation functionality to the side-panel and therefore still had a significant amount of work to do in order to create a user-friendly, streamlined interface. We had customised the appearance of our map, creating our own user-friendly colour scheme and removing unnecessary landmark labels to reduce clutter.

We agreed with our Product Owner that our priorities for the remainder of Sprint 4 and the final week of the project were to integrate our prediction models, integrate our weather forecast scraper and schedule the scraper to run at regular intervals, organise, modularise and comment our code and deploy the application on EC2. Our Product Owner was content with the additional features we had implemented and advised us to prioritise finalising and polishing our User Interface over the addition of any further features. We discussed sourcing a domain name for our web application with our Product Owner, who advised us that this

was not a hard requirement for this project and that exposing the port so that our application was remotely accessible for grading would suffice.

Following this Product Owner Workshop, we set ourselves a soft coding deadline for the Sunday before the submission deadline as we had a heavy workload for the following week and wanted to address any deployment issues early. This gave us 3 days to implement the features mentioned above and build out our User Interface in preparation for a deployment to EC2 on Sunday. We hoped that moving deployment forward would relieve some pressure for the following week.

We successfully deployed our Flask application on the Sunday. As our AWS starter accounts were due to expire, we first had to create and run a new instance on one of our AWS classroom accounts and prepare this instance i.e. through downloading Miniconda, creating a new virtual environment and installing all modules. We initially attempted to set up Gunicorn to serve our Flask application but ultimately decided against this as it was proving somewhat less straightforward than we had expected and as using Gunicorn was not a hard requirement, we felt our time would be better spent finalising other areas of the project. Also, as our web application does not require location sharing it does not need to be served over HTTPS and therefore running the Flask app directly, adding --host=0.0.0.0 to make the server publicly available and allowing traffic from port 5000 on EC2 satisfied our needs.

Our priority for the remaining time until the deadline was writing our project report, as well as making any last adjustments to our Flask application. When we met with our Product Owner for the final time, our Flask application was ready for final deployment, pending Product Owner review and we had made significant progress with our report. Our Product Owner was pleased overall with the functionality and usability of the finalised Flask application and was happy for us proceed with deployment, suggesting only superficial adjustments that could be made if we found the time:

- The background colour of the side-panel should be updated as the weather icon was difficult to see. This would improve the accessibility of the application, as well as increasing the appeal of the application.
- Add spacing between individual station listings in nearest station list.

- Rearrange the nearest stations list to group the “Get Directions” and “View on Map” clickable options together. Less importantly, consider styling these options as buttons.
- Center the station selector and associated label in the “Availability” tab
- Vertically centre the “Stations” and “About Disappster” links in the header.

We agreed that the suggested adjustments would improve the User Interface of our application, particularly those that would enhance the accessibility of the application, and worked hard to find time to make these changes before the deadline the following day.

Sprint Retrospective

By the end of Sprint 4 we had a fully functional Flask application that allowed the user to:

- View all Dublin Bikes stations on a map.
- View current weather conditions in Dublin.
- Find a station by selecting a marker on the map or selecting a station from a dropdown menu.
- View current availability and graphed average availability for each station.
- Enter a time and date within the next five days to view predicted availability for each station.
- Enter their location and receive a list of the five stations closest to that location.
- Choose one of these five stations to view the recommended route from their location to that station and receive text directions describing the route.

Our prediction models and weather forecast scraper had been integrated and our User Interface finalised, bar the small adjustments mentioned above. Our test deployment had been a success and we were almost ready for a final deployment.

Sprint Reflection

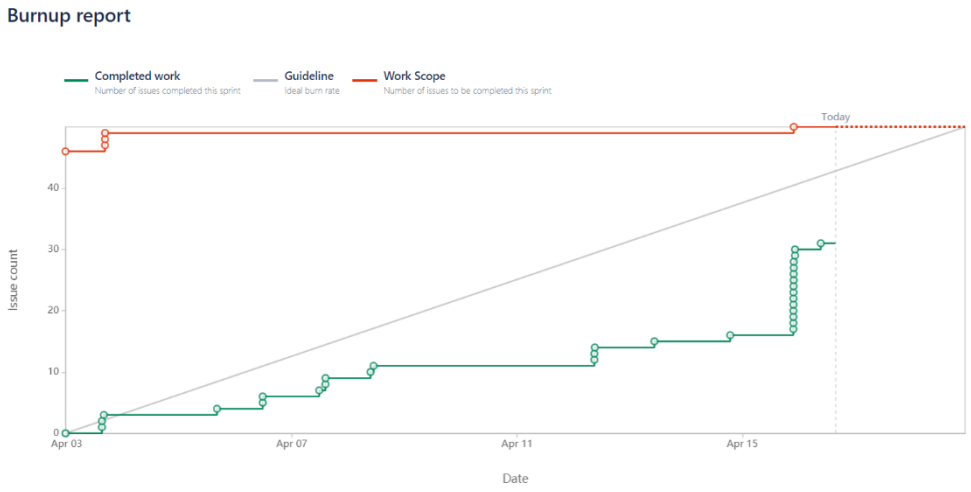
The workload for Sprint 4 was heavy and this was compounded by the additional task of writing the report. However, by the end of the sprint we had managed to meet most of the targets we had set ourselves, as can be seen by examining the burnup and burndown charts below. There were just a few issues outstanding, most of which were related to writing this report, which was still in progres. There were also some additional features that we had wished to implement, which are described in the Future Work section of this report, but we

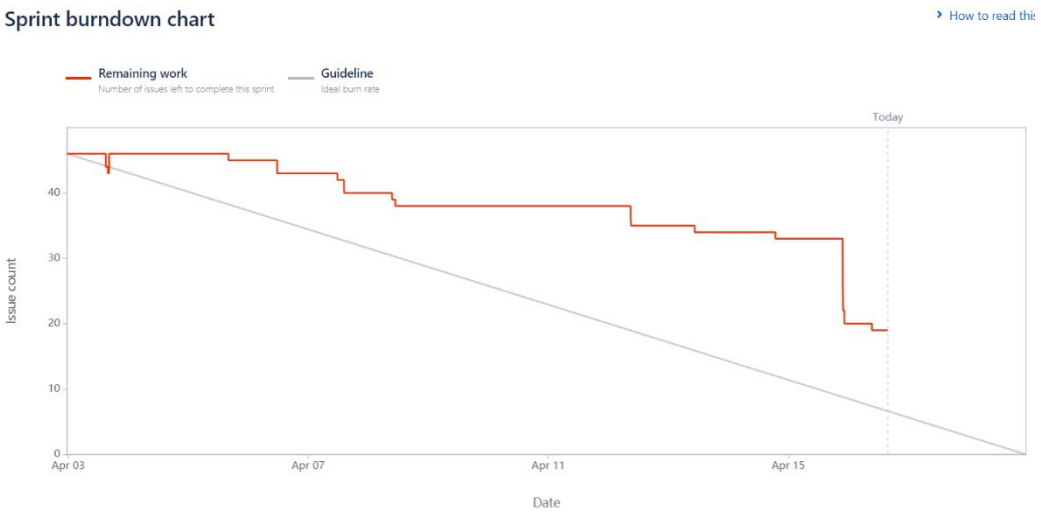
had managed to include all required functionality, incorporate some supplementary features and create a reasonably smooth User Interface and overall, we were satisfied with the application we had developed.

We worked well in Sprint 4, as is always the case with a deadline looming, but also as we had become very comfortable with the Agile methodology. We were making good use of the Jira board ticketing system to divide tasks and were each starting to learn what areas of web application development were of particular interest to us, which made it easier to decide who would do what. As well, our web development skills were improving and it was taking us less time to complete our individual tasks and less time to fix problems and bugs as they occurred.

Sprint 4 involved a lot of collaboration as we worked to integrate any features we had been developing separately and deploy our application on EC2. We held regular meetings in addition to our daily stand-up meetings to carry out these tasks and work through any related issues together. We were all motivated to do as much as we could to deliver the best product we could with the time remaining and worked well together despite being under additional pressure.

Burndown Reports





Retrospective/Future Work

In this section we won't discuss the retrospectives from each sprint as they have already been covered in each sprint's section in the process section of the report. Instead we aim to give a short general retrospective of the project and discuss areas for future work in the project. Looking back at the overall project we are quite pleased with our completed project and how we worked as a team. We are also quite satisfied with how we've progressed throughout the project in regard to both our technical skills and our soft skills in working as a team and project management.

One of the main learning points we took from the project in regard to project management was the importance of clear small tasks in an agile work environment. During the first two sprints our Jira board was mainly full of large over-arching tasks that we would each need to complete as we were not yet dividing up our work in a unified codebase. Although this helped with gaining a good level of understanding of the previously unfamiliar foundational aspects of our application with such as the flask framework and the google maps api, it was not the most efficient way of working.

Working in this manner during the first two sprints also meant that we didn't get as much out of our daily stand-up meetings as we could have. We found that these meetings were much more beneficial once we were working on a unified codebase whereas in the first two sprints it didn't matter quite as much if we knew exactly what everyone was working on that day as long as we all completed our work at the end of each sprint. Nonetheless, the meetings were still of great benefit as they kept us all accountable and on track and we could discuss issues

that arose in our own work but these meetings became more essential once we were working on a unified codebase and some work was dependant on another team member finishing their work first.

As mentioned previously, this method did help us a lot with obtaining a good foundation for the project but we are confident that changing our approach at the end of sprint 2 was the correct decision as the workload shifted more from becoming familiar with new technologies and building the foundation of our application towards building more advanced features. This allowed us to choose to work on areas that actively interested us or played to our individual strengths. We also found that having smaller tasks to pick from on the Jira board was also a much more enjoyable way to manage our work and less intimidating than starting a massive task.

One thing that we could have done that may have helped us keep track of our initial goals and broader plan for the assignment would have been to make an SRS. Although this would likely have been very time consuming and perhaps not possible to make a full SRS, a small document similar to an SRS simply outlining our initial vision and plan for the project may have helped us when making later design decisions and analysing how our plans changed throughout the project. This may have also cut down the amount of time we spent in pre-sprint meetings as we would have already known the larger tasks that needed to be completed from our SRS, allowing us to just split them up into smaller sub tasks for the sprint. Nonetheless, we are quite satisfied with our overall project management and feel that we documented our progress and structure of the project quite well throughout which made this less of an issue but more something we potentially could have done if we had time to aid us in writing this report.

Although we are overall very happy with our project and the progress we've made, unfortunately there are certain features that we had planned to complete but did not have enough time to integrate them into our application before the deadline. One of the most useful features that we originally planned to add but ran out of time was adding a radio button to allow the user to view the application in the context of available bikes or available stands, which would be much more useful when trying to return a bike. At the minute the application only works within the context of available bikes, our prediction model only returns predictions for bike availability currently, and our markers are colour coded by bike availability, not stand

availability. In theory, this shouldn't be massively time consuming as once the total capacity of the station and number of available bikes is known then the number of available stands can be easily calculated as total capacity – number of available bikes. Unfortunately, we simply ran out of time before we could actually integrate this feature, but it would likely be one of the first features we would add if we had additional time.

We are quite happy with our nearest-stations-locator feature which is our main additional feature outside of the project requirements, but we would extend out the functionality of this feature in future work if possible. Currently, the feature works by allowing the user to enter an address with the help of the google maps api and then our application will show the nearest stations to that address with the option to get directions to any of these stations. We would like to extend this feature to also automatically obtain the users geolocation and display the nearest stations to their live location instead of always having to manually enter an address. At the moment, when a user asks for directions to a nearby station the directions are always given with the assumption that the mode of transport is walking but we would like to change this to give the user the option to manually enter their mode of transport.

Finally, we would like to allow the user to choose if they want to see nearby stations with available bikes or available stands and automatically not show any stations with no currently available bikes/stands (depending on their selection). In a future version of the app we would also like to add a feature to save the last location entered by the user to automatically show them stations nearby with available bikes/stands. Ideally, we would accomplish this by storing this data on the client side without saving any information on our servers to circumvent any GDPR complications.

Along with automatically capturing the users live geolocation and offering them directions to nearby stations, we would also like to make the application more responsive and mobile friendly. Due to the nature of the app, it's likely that the majority of users would be using the app on mobile devices, so it would make sense to add greater support here to allow users to find nearby stations while out and about. Currently, our application works more like a journey planning app as it is designed to be used on a desktop web browser with emphasis on future predictions of bike availability as opposed to quickly determining the users location and finding nearby stations to either find or return a bike quickly while out.

On the backend there are several features we would like to integrate to streamline our application. Currently, we are deriving several tables using python in our back-end but in the future we would like to adapt this so that these tables are automatically generated in our database which would add visibility and reduce load on our EC2 instance by deriving them in python. This could also potentially give our application a performance boost as these tables would automatically be calculated or populated in advance instead of trying to calculate it on the fly in flask.

Our test and training data for our machine learning model is currently stored in a table in our database but this must be manually updated by running a python script at the moment. In the future we would like to integrate this model refresh into our application so our data would be automatically updated at predetermined intervals. In a later version of the app we would also like to add a table in our database which would keep track of predictions requested by the user and the predictions given back by our model. We could then also keep track of whether these predictions were correct or not which would allow us to see how accurate our model is and how the accuracy (hopefully) improves over time.

Although this is not technically a feature, we were not able to add as many unit tests to our application as we originally hoped and would like to add more in the future. Currently, our only unit test checks the connection to the database, but we had various other tests planned initially but were unable to integrate them in time. We were largely unfamiliar with writing unit tests before the project started and found that it was taking a disproportionate amount of time to develop and integrate them so we decided to push them back so we could still meet all our other more essential deadlines. Unfortunately, we were unable to come back to them before the end of the project but would like to integrate them in the future before further development if possible.

In the future we would also like to adapt our server to run using guinicorn and nginx instead of the built in Flask server. In a production setting we would also of course buy a domain name and host the app using this domain instead of simply opening port 5000 on our public ip address, but we did not deem this necessary for project submission and it would have cost money to purchase a domain name.

Meeting Logs

Please see the appendix which contains a full copy of our Confluence board which served as the main channel of documentation of work, meetings, and processes during the course of the project.

Appendix