Final Year Project

dBikes Planner: Smart Routing Support for Bike Rental Schemes

Oisín Quinn

Student ID: 16314071

A thesis submitted in part fulfilment of the degree of

BSc. (Hons.) in Computer Science

Supervisor: Dr Gavin McArdle

UCD School of Computer Science

University College Dublin

May 11, 2020

Table of Contents

Abstract

A major issue in many modern cities is disruptive traffic congestion, one of the leading contributors of greenhouse gases and emissions in urban centres. In recent years, bike-sharing schemes have grown in popularity throughout to world as an accessible and cheap way of providing sustainable and efficient transport to the residents of a city.

This project tackles the issue of reliable route planning for the Dublinbikes bike-sharing scheme. It develops a RESTful API for planning routes and predicting bike availability in a bike-sharing scheme. It also develops an Android application that uses this back-end API to allow users to plan routes to a given destination, using station availability prediction to select the most appropriate Dublinbikes station for the route.

Outline of Report Structure

Interim Report Chapters

Note: These chapters have been modified slightly to reflect the completed status of the project. New additions to these chapters have been formatted in red text.

Chapter 1: Introduction

Chapter 2: Related Works and Ideas

Chapter 3: Data and Context (adapted from Data Considerations from the Interim Report)

Additional Final Report Chapters

Chapter 4: Detailed Design and Implementation

Chapter 5: Evaluation

Chapter 6: Conclusions and Future Work

Acknowledgements

I would like to thank my supervisor Dr. Gavin McArdle for his invaluable support, guidance and advice throughout this project.

Thank you to everyone who provided feedback and suggestions for this project, from its incarnation to its completion. I would like to especially thank Darragh Clarke, Charles Kelly and Thomas Creavin for your help in testing this app and helping solve one particularly tough bug late in the development process.

Thank you to Carlos Amaral for adding an open license to your Dublinbikes dataset on my request, giving others the freedom to extend your previous work.

Finally, thank you to all of my family for your unconditional support throughout my education.

Project Specification

Core requirements

The student will identify suitable features for bike and parking space availability prediction for a given city.

The student will develop an algorithm to predict bike and parking space availability.

The student will evaluate the effectiveness of the approach.

The student will develop an Android application for real-time and future route-finding which integrates the algorithm and incorporates user preferences for route type.

Advanced requirements

The student will include a VGI component.

The student will examine the possibility of multi-modal trips.

Chapter 1: **Introduction**

Traffic congestion is a critical issue faced by all major cities in the 21st century. It has been identified as a potential cause of slowing economic growth [1], and can cause respiratory health problems in children [2]. Transportation is also one of the leading contributors of greenhouse gas emissions globally, with road transport accounting for 10% of global greenhouse gas emissions [3]. To help tackle the issue of traffic congestion, cycling has been identified as an appropriate replacement for short urban car trips that could reduce annual CO_2 emissions by tens of thousands of tonnes [4].

One of the most successful methods of promoting cycling in urban areas is the introduction of bike-sharing schemes (BSS) across the globe. As of June 2014, there were 712 cities implementing BSS internationally, with over 800,000 bicycles across approximately 37,500 stations [5]. The first bike-sharing scheme was Amsterdam's *White Bike Plan*, introduced in 1965 [6]. This scheme involved fifty white, permanently-unlocked bicycles being placed throughout the city. The scheme was a failure, but inspired other schemes in Cambridge, UK and La Rochelle, France. These schemes were known as First-Generation Bike-sharing Schemes. These schemes featured permanently-unlocked bikes that were free to use, and often stolen. In 1995, Second-Generation Bike-sharing was launched in Copenhagen with the *Bycyken* scheme [7]. 1,100 bikes were distributed throughout the city at designated bike racks and were unlockable after a 20DKK (€2.50) coin deposit.

Today, most BSS follow the principles of Third-Generation Bike-sharing [6], pioneered by *Vélo à la Carte* in Rennes, France. These systems use computer systems to track bike usage, electronic locks and docking stations [8]. Dublin's bike-sharing scheme (known as *Dublinbikes*) has become increasingly popular, with 66,203 subscribers and 2,972,857 journeys year-to-date as of 30th September 2019 [9]. Due to the usage of computer systems in Third-Generation Bike-Sharing Schemes, there is an abundance of publicly-available data that can be used to analyse, improve and optimise the BSS using real-world usage.

One notable issue with BSS is that each docking station has a limited number of available docking stands for bikes. If a user arrives at a docking station with no available stands, they must either wait for a bike at the station to be checked-out or travel to a different docking station with an available stand [10]. Conversely, if a docking station has no available bikes, users must wait for a bike to be checked-in, travel to a different docking station with at least one bike or use an alternative mode of transport. This adds a level of uncertainty to journeys, as it can be difficult for a user to predict whether bikes or stands will be available at a station before starting your journey.

In addition to this, it can be difficult to discover the most suitable route to take when using bike-sharing schemes. Users cannot cycle directly to their destination, since bikes must be returned to a docking station before a journey is completed. One must take the availability of parking spaces into account when planning this journey, as well as a station's proximity to the final destination. There are some major differences between route planning for cycling trips and route planning for car or pedestrian trips. Cyclists often prefer longer trips if they are perceived to be safer, with dedicated cycle lanes and less right turns into oncoming traffic. Cyclists often also take into the account the incline of a route, as uphill routes are much more physically draining. However more experienced cyclists may prefer to take a tougher route if it was considered to be shorter, and may not care as much about dedicated cycle lanes. This difference of preferred route type is important to take into consideration when generating routes.

This project tackles the issue of transport uncertainty in relation to bike-sharing schemes by de-

signing a route-planning API that predicts bike and docking space availability for the Dublinbikes, third-generation BSS. An Android application has been developed that utilises this API to let users plan journeys to a given destination. By developing a mobile app to solve this problem, this project creates a portable way for users to generate the optimal route at any time, by simply using the smartphone in their pocket. Mobile apps also have the added benefit of GPS technology, so the user's location can be accessed, making for a more seamless user interface.

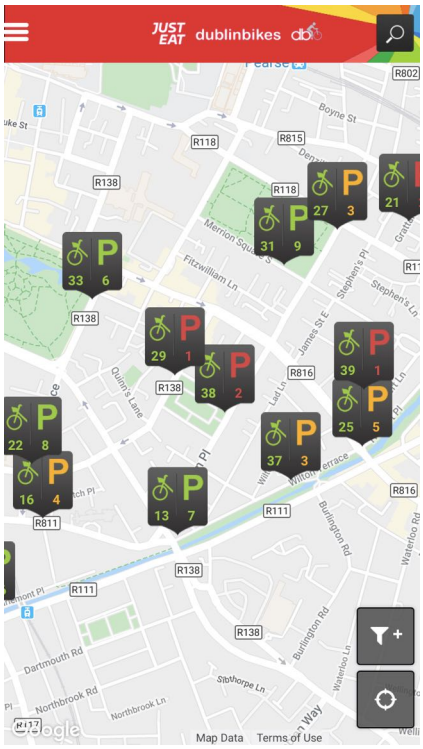
In this paper, Chapter 2 will discuss the existing applications of bike-sharing schemes, route planning, bike availability prediction and Android app development. Chapter 3 will discuss the data sources that will be used by this project, and the associated considerations. Chapter 4 will provide an in-depth description of the project's design and implementation, discussing the completed implementation for the back-end web server, machine learning model and front-end Android application. Chapter 5 evaluates the performance of the model and describes the results of a user test of the application. Chapter 6 discusses the conclusions of the project, and explores possible ideas for future expansions of the project.

Chapter 2: Related Work and Ideas

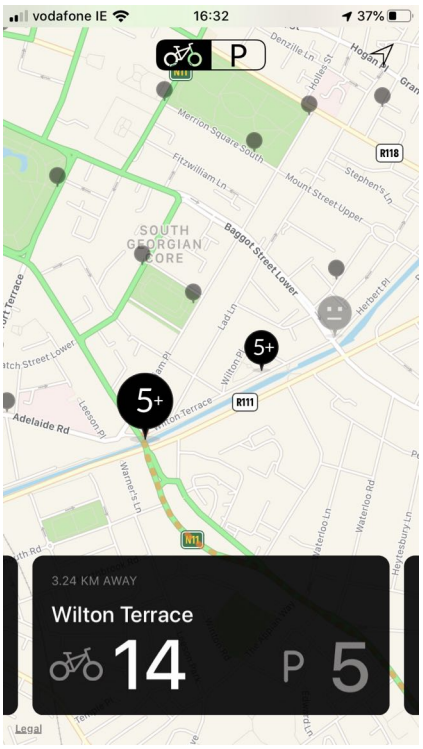
In this chapter, we will examine the bigger picture of bike-sharing schemes and discuss existing technical solutions for solving the pitfalls associated with them. We will discuss currently available mobile applications for users of bike-sharing schemes (Section 2.1), existing attempts of bike availability prediction (Section 2.2), route planning for cycle journeys (Section 2.3) and the current state of Android app development and the associated technologies (Section 2.4).

2.1 Current applications for bike-sharing schemes

Currently in Dublin, there are multiple apps available on and the Google Play Store and Apple App Store for tracking Dublinbikes availability in the city. The official app is titled *Just Eat Dublinbikes*[1]. It provides information about the availability of bikes and bike parking spaces in a map view (Fig. 2.1a). The app has not been updated in the past two years and has an average rating of 1.3 stars on the iTunes App Store.



(a) A screenshot from the official Just Eat dublinbikes app



(b) A screenshot from the NearBikes app

Figure 2.1: Screenshots from various mobile applications designed for the Dublinbikes BSS

Other available apps include *Simple DublinBikes* by Julien Drecq[2] and *NearBikes for Dublin Bikes* by Pablo Silva[3] (Fig. 2.1b). Another app called *Joyride* by UXtemple Limited[4] provides similar

[1] Just Eat dublinbikes: https://apps.apple.com/ie/app/just-eat-dublinbikes/id897509153
[2] Simple Dublinbikes: https://apps.apple.com/ie/app/simple-dublinbikes/id1308222409
[3] NearBikes for Dublin Bikes: https://apps.apple.com/ie/app/id1178781947
[4] joyride: https://apps.apple.com/ie/app/joyride/id1387335728

functionality but works with all of the worldwide BSS run by advertising firm *JCDecaux* (which includes cities such as Dublin, Brisbane, Vienna and Lyon). However this particular app failed to work when Dublin was selected as the city. Google Maps also recently added an integration with Dublinbikes, displaying the location of all stations and their bike availability levels in their app.

All of the mentioned apps provide similar levels of functionality using the same JCDecaux API: they all show the location of bike docking stations and the availability of bikes and bike parking spaces across Dublin. None of these apps provide any sort of prediction of future bike availability, and I could not find any apps on either the Apple App Store or Google Play Store that provide availability prediction or route planning for Dublinbikes. Other cities have similar apps that provide an all-in-one public transport solution (such as *Transit*[5] and *Citymapper*[6]) and route-planning for BSS, but do not provide any way of predicting bike availability.

2.2 Bike availability prediction algorithms

When predicting the availability of bikes in bike-sharing schemes, various machine learning algorithms have been used to attempt to solve this problem with varying levels of success. Some of the less accurate attempts include predictions based solely off the previous amount of bikes at a station (Last Value prediction or LV), predictions based on the average amount of available bikes at a station at this time in previous days (Historic Mean or HM) or a Bayesian Network model that categorized a prediction into imprecise bins based off the ratio of available bikes to total number of spaces [11]. These methods were relatively accurate when the prediction window was small, with accuracy within 5% of the actual amount of available bikes. However as the prediction window increases, the prediction error also increases: with a prediction window of 120 minutes, even the best of these models had a prediction error of 13%, corresponding to an error of over 5 bikes on average [11].

Two of the most successful methods of bike availability prediction discovered are the use of an Auto-Regressive Moving Average (ARMA) [12] model and a Two-Stage General Additive Model (TGAM) [13].

An Auto-Regressive Moving Average is a combination of two well-known machine learning models: an Auto-Regressive (AR) model and a Moving Average (MA) model [14]. The AR model predicts future bike availability based on the historical trends of bike availability for a station. The Moving Average (MA) model takes into account the error in predictions of availability in previous time periods. By combining these two models, A.Kaltenbrunner, et al. [12] created a model with a much higher accuracy than previous solutions. In their example, they use twenty minutes worth of historical data to influence the model. The mean absolute prediction error when using this method was a difference of 1.3908 bicycles, which is a stark improvement over more naïve approaches. With a prediction time of an hour, the ARMA model has a mean prediction error of just 1.39 bicycles (with a maximum error of 6 bicycles). The paper notes how "The use of more sophisticated time series analysis techniques (ARMA) and in particular the incorporation of information from surrounding stations allows us to improve these predictions further" [12]. However, this algorithm does not take into account the effect of seasonality on the model (for example, how weekday and weekend usage varies, or how summer and winter usage varies). This system was tested on Barcelona's *Bicing* system, so these techniques may behave differently in a different city (such as Dublin).

A Generalized Additive Model (GAM) [15] is a generalized linear model with a linear predictor involving a sum of smooth functions of covariates [16]. Chen et al. [13] used a two-stage generalized additive model (TGAM) to implement a function that predicts the availability of bikes at stations,

[5] Transit: https://apps.apple.com/us/app/transit-bus-subway-times/id498151501
[6] Citymapper: https://apps.apple.com/us/app/citymapper-transit-navigation/id469463298

and if no bike will be available, it also predicts the waiting time at the station for a bike to arrive.

The GAM detailed in the paper takes in four inputs: the type of day it is (i.e. a weekday or weekend), the time of day, the time of the year and the current weather (including the temperature and humidity). The GAM is a linear combination of multiple smaller functions dependent on the specified inputs, as well as two functions that represent the autoregressive behaviour of the model, taking into account the capacity of the stations over time. For short-term predictions, they used the full model. For medium-term predictions, the category of the weather is ignored (while still considering the temperature and humidity). For long-term predictions, all weather-related data and autoregressive terms are ignored and a new value relating to the historical average for that time of day is used instead. If no bikes are available at a station, this GAM will generate an estimated waiting time until a bike will become available at the station. The waiting time is an exponential function, with a corresponding intensity depending on the current time (for example, you must wait much longer for a bike later at night than at busier times of the day).

Chen et al. compared the performance of the ARMA and TGAM algorithms on the Dublinbikes BSS in their 2013 paper "Uncertainty in urban mobility: Predicting waiting times for shared bicycles and parking lots" [13]. In their study, they found that "TGAM yields 40% lower average and weighted root mean squared errors than LV, HM and ARMA, which all perform similarly" [13]. For medium-term predictions, TGAM yields 50% lower waiting times than ARMA, which both vastly outperform LV and HM. For long-term predictions, TGAM has 40% shorter wait times on average than ARMA. This shows that TGAM out-performs all previously discussed prediction models by taking extra variables such as weather and humidity into consideration.

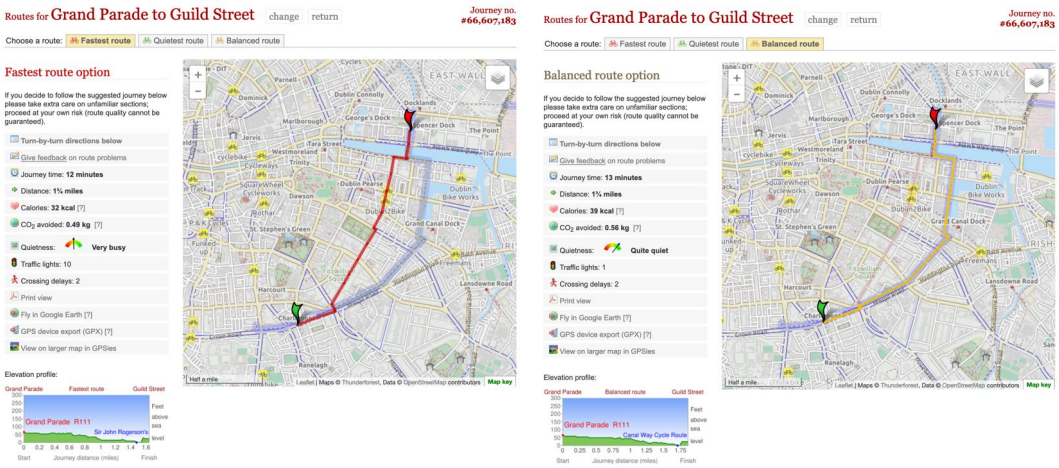
2.3 Route Planning

Route planning is a complicated task. This is especially true for bicycle route planning, as cyclists may prefer a longer route that would be safer due to the presence of cycling lanes and shared pedestrian & cycling areas. The shortest route may not always be the preferred route, as a longer route may in fact be preferred, due to a lack of traffic lights and traffic congestion, more suitable terrain or a more accessible route slope [17]. Therefore, designing a route planning algorithm is not a trivial task.

Various public APIs exist with different implementations of bicycle route planning, as discussed by Hrncir, Jan, et al. [18]. Google Maps[7] provides a cycle route planning API, but does not let the user configure their preferences for the route and also does not have a free tier on offer currently. Other alternatives with free options include Openrouteservice[8] and OpenTripPlanner[9].

One of the most versatile APIs is *CycleStreets*[10]. CycleStreets is an application developed by the Cambridge Cycling Campaign to help cyclists discover suitable routes for cycling across major cities and towns in the UK and Ireland. This API is unique because it takes into account the route attributes that are most important to cyclists: inclines, cycle lanes, lighting and surface quality. It also lets the user enter up to 30 waypoints to include in their route. The most unique feature is that users can decide exactly what kind of route they would like; they can choose between the fastest route (Fig. 2.2a), the quietest route with less traffic, the shortest route where you travel the shortest distance or a balanced route that finds a compromise between all three of those descriptions (Fig. 2.2b). The API also returns the CO_2 emissions saved by cycling instead of driving, the number of traffic lights encountered and calories burned. These features make this API the most useful API to use for generating cycling routes in the UK and Ireland.

[7] Google Maps: https://cloud.google.com/maps-platform/routes/
[8] Openrouteservice: https://openrouteservice.org
[9] OpenTripPlanner: http://www.opentripplanner.org/
[10] CycleStreets: https://www.cyclestreets.net/api/



(a) The "fastest" route

(b) The "balanced" route

Figure 2.2: Comparison of routes generated by CycleStreets from Charlemount Luas stop to Guild Street

2.4 Android app development

Android is a mobile operating system developed by Google and the Open Handset Alliance. The most common way to develop Android applications is the native approach. This involves using Android Studio[11], Google's official Integrated Development Environment, defining the app's logic in Java or Kotlin and defining layouts in XML. This is considered the de-facto method of developing an Android application. In recent years, alternatives to native Android development have become increasingly popular, with many alternatives allowing you to write code once and create both Android and iOS apps from the same source code. Cordova[12] is a framework that lets programmers write applications in HTML, CSS and JavaScript. However, its biggest issue is that it renders the application inside a WebView which causes a noticeable decrease in performance. Ionic[13] is a similar framework that allows developers to use the popular JavaScript frameworks Angular[14] and React[15]. However, it prevents developers from mixing Ionic code with native code, and has performance downsides since it also renders the application inside a WebView instead of using native components.

In recent years, a new subset of app development frameworks allow you to write cross-platform code that compiles into native code, which helps to reduce the performance issues associated with the previous generations of frameworks. Xamarin is Microsoft's open-source solution for native cross-platform development. One issue with Xamarin however is that it depends on the .NET framework and the C# programming language, which while growing in popularity, aren't as widely used as some JavaScript-based alternatives. Flutter is Google's cross-platform framework, but it requires developers to learn a new programming language called Dart. Progressive Web Apps (PWAs) are also surging in popularity, where the app is simply a website optimized for mobile devices while storing necessary HTML files on the user's device and adding an icon to the user's home screen. PWAs do not support all of the hardware features of a device and they have limited support on iOS and older versions of Android. PWAs can also not be added to the Google Play Store or Apple App Store.

[11] Android Studio: https://developer.android.com/studio
[12] Cordova: https://cordova.apache.org/
[13] Ionic: https://ionicframework.com/
[14] Angular: https://angular.io/
[15] React: https://reactjs.org/

One of the most popular forms of native mobile development is React Native[16]: a framework based off React, the popular JavaScript framework developed by Facebook. React Native has become increasingly popular, with Facebook, Instagram, Skype, Bloomberg, UberEATS and Salesforce using React Native in their mobile apps [19]. React Native compiles to native Android and iOS code, while also allowing developers to write native code in Java, Kotlin or Swift for the appropriate operating system. Tools such as Expo[17] make it even easier to develop React Native apps, taking away the need to use Android Studio or XCode to develop your app and adding useful APIs that make app development even easier. React Native also has a strong community, with over 30,000 answered questions on Stack Overflow [20]. React Native combined with Expo seems to be the most complete solution to cross-platform app development. One issue with React Native, however, is that it does not have the same level of third-party library support that native Android development has. This can be a major setback when a project requires components that are unique to the Android development environment.

2.5 Summary

There are many mobile apps for bike-sharing schemes using real-time bike data currently available, but none of them go beyond providing basic data about the current availability of bikes at a station. No apps currently allow users to plan routes or predict availability of bikes in the future. The area of bike availability prediction has been researched thoroughly in the past, with multiple effective models developed in recent years, with some using the Dublinbikes BSS itself. Route planning is also a well-researched topic in computer science, with many tried-and-tested algorithms available for public use. There are also many free public APIs available for route planning for all modes of transport, including cycling. Finally, Android app development has advanced in recent years, with new cross-platform frameworks such as React Native and Flutter growing in popularity. Using this research, this project will create a route-planning app for the Dublinbikes BSS that will consider both bike availability and the optimal route for the user.

[16] React Native: https://facebook.github.io/react-native/
[17] Expo: https://expo.io/

Chapter 3: Data and Context

Data plays a crucial part in this project. Data is needed to discover the current number of bikes at a station, and historical bike availability data is used to build the predictive model. We also need weather data as an input in our model to greater predict the availability of bikes and spaces at stations. Since we are using datasets with many features, it is important to consider exactly which features of the data are the most relevant to the project. Routing data and APIs are also used to generate optimal routes between two locations.

This section discusses in detail what data sources will be used by this project and which features of these data sources will be used.

3.1 Bike usage data

This project uses Dublinbikes usage data obtained from the official JCDecaux Dublinbikes API[1]. This JSON API provides real-time information about the state of every bike station in Dublin. The endpoint returns a JSON array containing a series of JSON objects, with each object representing the state of a bike station. Each object contains the station ID, station name, position, update time, total number of bike stands, number of bikes currently available and the number of bike stands currently available. This API was chosen because it is the official API of the Dublinbikes BSS. It contains all of the data we need to predict bike availability, and since it is the official API, it is a reliable API that rarely goes offline.

Two sources of historic bike usage data are used for this project. The first source is a dataset obtained from GitHub[2] containing real-time information obtained from the JCDecaux Dublinbikes API every ten minutes, over a period of eight months from September 2016 to April 2017. While this dataset is not the most recent, it is one of the most detailed public datasets available at the time of publishing that included data collected over a substantial period of time. The dataset is also very complete, containing information about every bike station over a period of eight months without any major omissions. The dataset has 6.32 million rows and has a disk size of 362MB. It consists of a separate file for each station. Each file contains six columns: the ID of the bike station, the name, the address, the total number of bike stands, the number of available bike stands and an ISO timestamp of when this data was obtained. This data is licensed under a GNU General Public License v3.0 license, allowing private and commercial usage of the data.

The second part of this dataset was collected directly from the JCDecaux Dublinbikes API. A Python script (shown in Listing 3.1) was written to access the API and store the returned data in a .csv file on a UCD-provided server. This Python script runs every five minutes on a cron job and stores all of the returned data. This dataset contains approximately five months of bike data.

[1] JCDecaux Dublinbikes API: https://developer.jcdecaux.com

[2] Carlos Amaral's Dublinbikes dataset: https://github.com/amaralcs/dublin_bikes/tree/master/data_dump

```
import pandas as pd

result = pd.read_json("https://api.jcdecaux.com/vls/v1/stations?
contract=dublin&apiKey=API_KEY")

with open('bike_times.csv', 'a') as f:
    result.to_csv(f, header=False)
```

Listing 3.1: Python script for accessing JCDeceaux API and storing retrieved data

This data is licensed under the Open Licence from Etalab, a license compatible with the Open Data licenses of ODC-BY and CC-BY 2.0 [21]. This license "promotes wider reuse by allowing copying, redistribution, adaptation and commercial data". Since this data is already public and anonymous, there are no privacy or ethical issues associated with the usage of this data. All obtained data will be released publicly on GitHub after this project has been evaluated.

```
{
    "number":42,
    "contract_name": "dublin",
    "name": "SMITHFIELD NORTH",
    "address": "Smithfield North",
    "position": {
        "lat": 53.349562,
        "lng": -6.278198
    },
    "banking": true,
    "bonus": false,
    "bike_stands": 30,
    "available_bike_stands": 25,
    "available_bikes": 5,
    "status": "OPEN",
    "last_update": 1573049161000
}
```

Listing 3.2: JSON snippet for an individual bike station's state

The most important attributes of this data (shown in Listing 3.2) for the project are the total number of bike stands, the number of available bikes and the update time. These are used in the model to predict bike availability. The address and position attributes are used by the Android app to display the location of each station.

3.2 Weather data

Data relating to the current weather is obtained in real-time from OpenWeatherMap[3]. OpenWeatherMap's "Current weather data" service provides real-time weather information for a given location, including the temperature, pressure, humidity and wind speeds. This API has a free usage limit of sixty requests per minute, but since this application will only be used in Dublin, the back-end server will only need to retrieve weather data every few minutes. This API was selected because it provides real-time information, while alternatives such as Met Éireann only provide data every hour. OpenWeatherMap also has a generous free tier, providing all the required information.

This data will be supplemented with historical data from Met Éireann, detailing weather conditions

[3]OpenWeatherMap: https://openweathermap.org/

Date	Precipitation Amount (mm)	Air Temperature (C)	Vapour Pressure (hPa)	Relative Humidity	Mean Wind Speed (knots)	Sunshine Duration (hours)	Visibility (m)
30/09/19 16:00	4.5	11.8	13.7	99	13	0.0	7000
30/09/19 17:00	0.3	11.9	13.6	98	14	0.0	4500
30/09/19 18:00	0.6	12.2	13.9	98	18	0.0	3500
30/09/19 19:00	2.6	12.7	14.3	98	19	0.0	5000
30/09/19 20:00	1.2	13.0	14.6	98	17	0.0	6000

Table 3.1: Edited snippet of data from Met Éireann's historical weather dataset for their Dublin Airport weather station

from the Dublin Airport weather station every hour since January 1989[4]. This dataset contains the precipitation amount, air temperature, vapour pressure, relative humidity, mean wind speed, sunshine duration, cloud height and visibility at Dublin Airport for every hour (as shown in Table 3.1). This dataset was picked over alternatives such as the Phoenix Park weather station dataset, because the Dublin Airport dataset contains useful weather features that most other stations do not track, such as mean wind speed, sunshine duration and visibility. While Phoenix Park is closer to Dublin city centre and would provide slightly more relevant weather information for this project, the extra features detailed above will help us design a more accurate prediction service, taking into account features that previous algorithms have ignored.

The predictor used in this project will use many weather features to help predict bike availability. In particular, the temperature, precipitation amount and relative humidity will be used and compared to historical bike availability. These were the features used by Chen et al. in their studies [13]. In addition, additional weather features such as mean wind speed and visibility are experimented with in this project in an attempt to improve on prediction accuracy.

3.3 Routing data

To generate the best physical routes for a user between two points, this project uses two different free APIs. The CycleStreets API[5] is used to generate the cycling portion of the route, while the Open Route Service API is used to generate the walking portion between the user's location to the appropriate bike station, and from the destination bike station to the user's final destination.

The JSON payload returned from the CycleStreets API contains a list of "marker" objects, each signifying a different individual segment of the overall cycling route. Each "marker" object contains a description of the segment, including the location of each point in the segment, a verbal description of the direction, distance and elevation information, the estimated time to cycle this route and a value signifying the busyness of the segment.

Similarly, the OpenRouteService API is used to generate the best walking route from a user's

[4] Met Éireann dataset: https://data.gov.ie/dataset/dublin-airport-hourly-weather-station-data
[5] CycleStreets API: https://www.cyclestreets.net/api/

location to the best station (as well as from the drop-off station to the user's destination). Similarly, the JSON payload for this API returns a list of "segments", each detailing the location of the segment, the distance and time spent to walk this segment, the names of each street in the route and also the basic descriptions of the directions required to reach the destination.

When a route is generated, the route is stored for future reference and to associate a route with volunteered geographic information (as discussed in Section 3.4). This data is not associated with any personal data, and simply stores the UUID of the route as an identifier.

3.4 Volunteered geographic information

This project also avails of volunteered geographic information to help improve the quality of future route calculations. After a user has physically completed a journey generated by the app, they receive a notification on their phone. By clicking on this notification, the user is given a short questionnaire that they can use to provide feedback on their last completed route. This questionnaire asks the user if either of the two stations in their journey had queues for bikes or space. This is a measure of how busy a station is that cannot be deduced via the Dublinbikes API.

If the user provides this data, it is sent to the web server alongside the unique ID representing this route. This information is stored in a Google Cloud Datastore, where it can be used by future iterations of this project. This volunteered geographic information has no personal information associated with it, as all routes are generated anonymously.

Chapter 4: Detailed Design and Implementation

The implementation of this project consists of a native Android application that allows users to plan routes between their location and their destination. It calculates the most appropriate bike stations in the Dublinbikes BSS to suit their journey by communicating to a back-end web server, taking predicted bike or space availability into account. Users can also browse the location of stations in the Dublinbikes BSS and view the real-time status and historical usage of any station.

The back-end architecture is designed to follow the microservices approach of software development. With microservices, we split our application "into smaller cooperating components that are running out of process and talking to one another, which can be maintained separately, scaled separately, or thrown away if needed" [22]. Google AppEngine is used to provide high-speed, reliable infrastructure for our back-end web server. The *dBikes Planner* Android application accesses these endpoints to calculate and retrieve the best journey for the user. A high-level diagram of this system is described in Fig. 4.1.

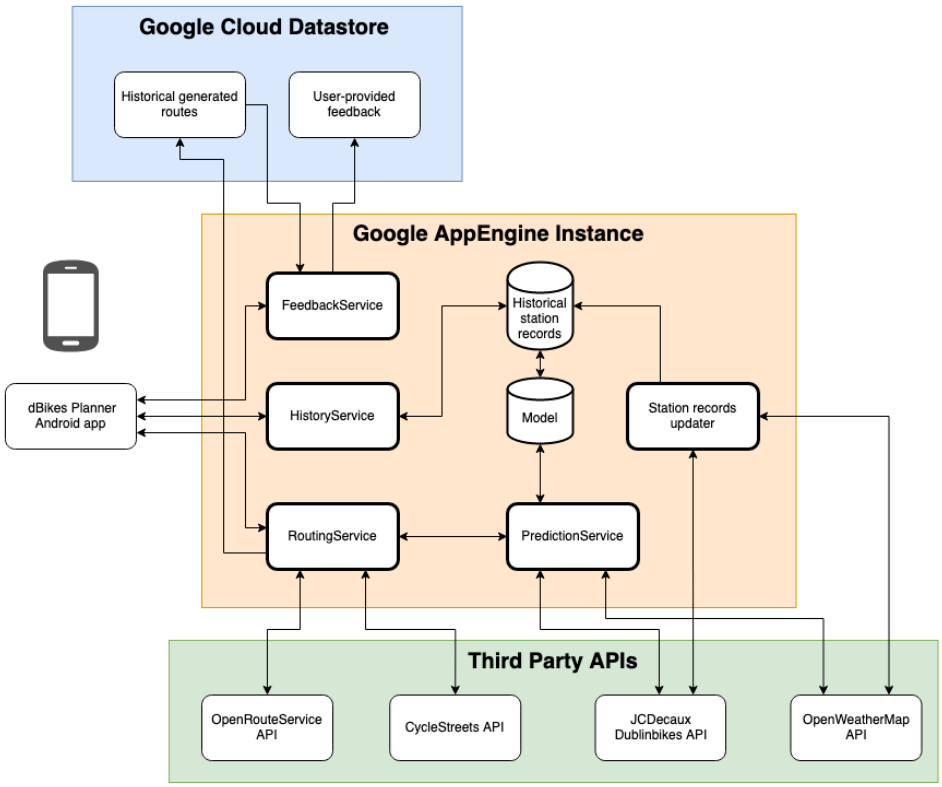


Figure 4.1: High-level diagram of the implemented dBikes Planner system

The dBikes Planner Android app contacts the Python back-end using REST HTTP requests, sent to a Google AppEngine instance hosting the back-end services. To calculate a route, the Android app sends a request to the RoutingService which in turn calls the PredictionService to calculate the most suitable stations for the user's trip, and in turn calculates the best route for the user (described in detail in Section 4.2). The AppEngine Instance consumes multiple third-party APIs in order to calculate this route. Once the route has been finalised, the route information is stored

in a Google Cloud Datastore and the result is returned to the Android app. This chapter describes the design and implementation of these features and services.

4.1 Bike Availability Predictor

As discussed in the previous chapter, this project uses both bike and weather data to predict the availability of bikes and spaces at BSS stations. For ease of use when loading this data into the prediction model, these datasets need to be combined and separated for each station. To achieve this, a series of Python scripts were created. Since all collected Dublinbikes data were stored as .csv files, they can be loaded using the pandas library and manipulated as DataFrames. Since the time periods for the bike data and the weather data were different, a function was defined to find the closest corresponding weather datapoint to the bike station datapoint (shown in Listing 4.1).

```
def nearest_time(weather_dates, entry_date):
    return min(weather_dates, key=lambda x: abs(x - entry_date))
```

Listing 4.1: Excerpt from Python script matching weather data to bike data

After preparing the data, it was decided to select a single bike station to use to evaluate the model during the early stages of development. The main benefit of using a single station to test is that model fitting times would be greatly decreased, which would help increase the speed of development. The Charlemont Street station was selected as the test station, as it follows a common pattern of bike availability (as shown in Fig 4.2): high levels of availability in the early morning followed by very low levels during morning commuting hours. The availability fluctuates during the day, before increasing rapidly in the evening commuting hours. This station was also selected because the student was familiar with its usage patterns based off personal usage.

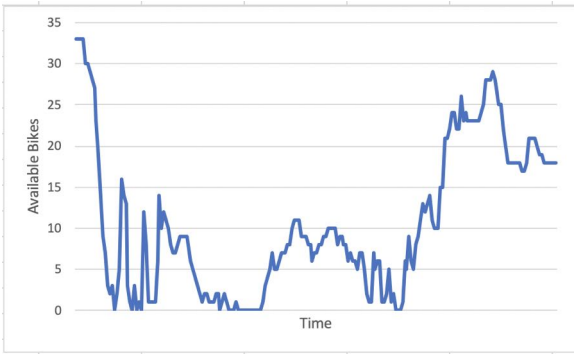


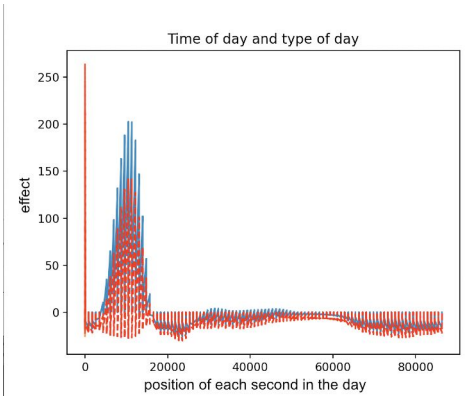
Figure 4.2: A graph showing the average availability of bikes at Charlemont Street station on a weekday

Using this dataset, work began on building a Generalised Additive Model using Python's pyGAM library [23]. Using the pyGAM documentation[1] and a conference presentation by the library creator at PyData Berlin 2018[2], a basic implementation of the pyGAM model was created. A tensor product was used to combine the time-of-day and type-of-day (e.g. if it was a weekend or weekday) attributes, and spline terms [24] were used to represent the day of the year, the temperature, relative humidity, wind speed, precipitation and visibility. LinearGAM, GammaGAM and PoissonGAM were experimented with at this stage to see which value would give the best results. Early testing resulted in inaccurate results, with predictions of bike availability frequently being incorrect by over 20 bikes. Accuracy was improved upon by increasing the number of splines used for each feature. Generating

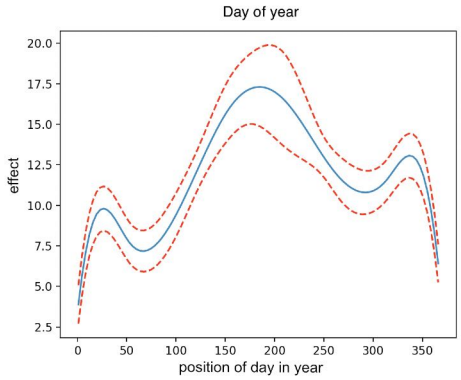
[1] PyGAM documentation: https://pygam.readthedocs.io/en/latest/

[2] pyGAM: balancing predictive power and interpretability using generalized additive models at PyData Berlin 2018: https://www.youtube.com/watch?v=XQ1vk7wEI7c

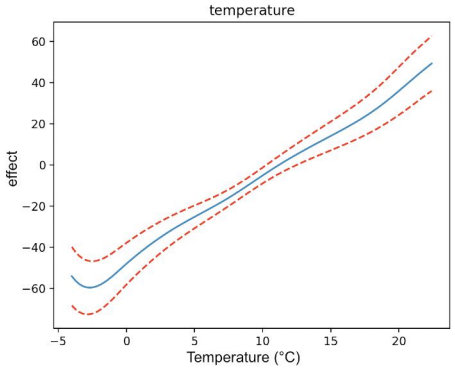
the partial dependence graphs for each spline shows the effect each spline has on the overall model, as shown in Fig. 4.3.



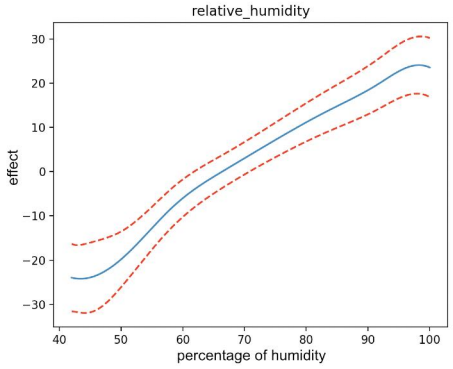
(a) Partial dependence graph for the tensor product of time_of_day and type_of_day



(b) Partial dependence graph for the spline term of day_of_year



(c) Partial dependence graph for the spline term of temperature



(d) Partial dependence graph for the spline term of relative_humidity

Figure 4.3: Plotted partial dependence functions for some example terms in the LinearGAM

At this point, we started to encounter some issues with our approach. While accuracy improved by increasing the number of splines used for each feature, this dramatically increased the computational time needed to fit the model. On average it would take half an hour to fit the model on a high-performance desktop computer with multiple cores and ample RAM. Since each individual bike station has different availability data associated with it, as well as individual daily patterns, a different model object would be required for each individual station. Since there are over 100 Dublinbikes stations, it is likely that it would take 50 hours to fit the model for all 100 stations. Unlike the methods of prediction described in Section 2.2, the predictor in this project is being used in a real-time application. For this reason, long fitting times that may be acceptable in other studies that are concerned solely with bike availability prediction, are not acceptable for this project, where prediction is just one part of a larger application.

In addition, pyGAM does not provide the functionality to add data to a model after it has been fitted. For example, after fitting the model, it is impossible to add the latest real-time bike information to the model without completely re-fitting the model. This issue, combined with the issue of long fitting times, indicated that this approach is not appropriate for this model.

Noting these shortcomings of the Generalised Additive Model, the feasibility of alternative machine learning techniques was examined, notably the Auto-Regressive Moving Average (ARMA). While experimenting with ARMA, I used the "statsmodels" library in Python, which has the most widely

used implementations of ARMA, including ARIMA (Autoregressive Integrated Moving Average) and SARIMAX (Seasonal Autoregressive Integrated Moving Average - the "X" addition to the name indicates it supports exogenous variables).

After some research, SARIMAX was chosen as the most suitable ARMA implementation as it is designed around seasonal data, like what is seen in our station dataset. After discovering the appropriate parameters using autocorrelation and partial autocorrelation plots, the model was constructed (with the Python code described in Listing 4.2).

During testing, it was noted that fitting the model for seasonal data was also very time-consuming and not appropriate for a real-time system. The results were also quite inaccurate, potentially due to the fact that this model is not taking into account the impact weather has on the real-time system.

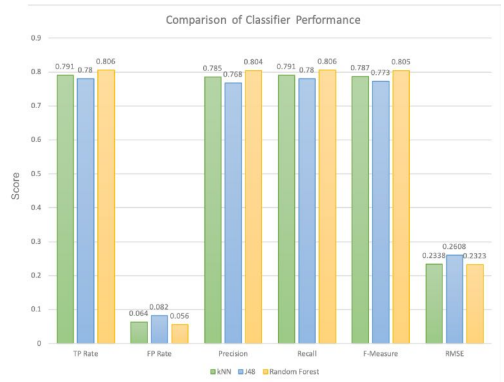
```
import statsmodels.api as sm

# data_train is the training data
# loaded from "Charlemont Street.csv"
model = sm.tsa.statespace.SARIMAX(
    data_train[['available_bike_stands']],
    trend='n', order=(2,1,5), seasonal_order=(1,1,1,235))
```

Listing 4.2: Excerpt from Python script constructing the SARIMAX model

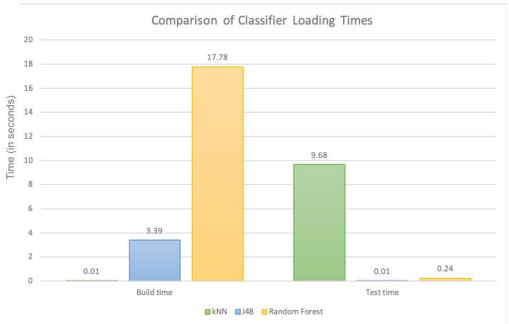
After experimenting with the previously discussed techniques, it was clear that neither were suitable for a real-time prediction system. Instead, it could be more appropriate to experiment with more widely used lazy learning machine learning algorithms. As well as this, it could be sufficient to convert this problem from a regression problem into a classification problem. When predicting availability for a cycling route, the user does not care if there are precisely 34 or 36 bikes available at a station. Instead, the user cares about the general level of availability at a station, i.e. if there are many, few or no bikes available.

To experiment with classification, five categories were selected, corresponding to different levels of availability; "empty" corresponds to zero available bikes, "very low" corresponds to one or two available bikes, "low" corresponds to three or four available bikes, "moderate" corresponds to between five and nine available bikes and "high" availability corresponds to ten or more available bikes. This same criteria was also extended to bike stand availability.

Weka Explorer was used to test the viability of different algorithms. The algorithms chosen were k-Nearest Neighbours, J48 Deicision Trees, Näive Bayes and Random Forest. These four algorithms were selected as they are some of the most widely used classification algorithms, and each function in their own unique way. The Charlemont Street station was used for testing, with a 95% split of training and test data. 95% was chosen as the split size, as it provides a balance between having a large training set and a sufficiently large test set of 2,293 cases. It is important to note that a percentage split is not the perfect method of splitting data for time-series problems such as this. Using a percentage split, the training set is likely to contain occurrences from after the test set, which can distort results. However, I believe that a percentage split can still be a sufficient method of estimating the high-level performance of different classifiers, while still being aware that the results obtained are only estimations. Taking this into consideration, the estimated accuracy and predictive performance of each classifier was compared, alongside the loading times for fitting and predicting, as shown in Fig. 4.4.



(a) Comparison of classifier performance



(b) Comparison of classifier loading times

Figure 4.4: The main aspects of each classifier were compared to select the optimal classifier

All three classifiers performed relatively well. Random Forest performed slightly better than kNN, with J48 trailing slightly behind in all categories. However, the difference between each classifier's performance is relatively subtle overall. In relation to loading times, Random Forest has a large up-front build time of 17.78 seconds. kNN in comparison has a build time of 0.01 seconds. While it had a testing time of 9.68 seconds, this test contained 2,293 test cases. In practice in our system, this would not be an issue, since only one prediction will take place at a time, which would take less than 0.01 seconds to complete.

After considering this comparison, kNN was selected as the classifier of choice, since it performed well when predicting and also had very manageable loading times for fitting and predicting. Since building takes so little time, the model can be fitted on-the-fly when requests are submitted, which will save memory in the back-end. Random Forest may give slightly more accurate results, but its long fitting time may cause issues in a back-end implementation and would certainly require the model for each station to be stored in memory.

In the finished project, the "sklearn" Python library is used for its KNeighborsClassifier implementation. The model uses 10 neighbours, weighted by the inverse of their distance, to make its predictions. The value of 10 neighbours was selected as this provided the highest f1-score in preliminary testing, as it is large enough to take into account both the most recent and the general trend of the dataset, without overfitting to the largest category. The code snippet in Listing 4.3 shows the implemented Python code used to build and fit the model.

```
def get_fitted_bikes_model(station_name):
    data = read_file_for_station(station_name)

    model = KNeighborsClassifier(n_neighbors=10,
        weights='distance')

    X = data.iloc[:, [2, 3, 4, 6, 7, 9, 10, 12, 15]]
    y = data.iloc[:, [13]]

    minmax = MinMaxScaler()
    X = pd.DataFrame(minmax.fit_transform(X), columns=X.columns)

    model.fit(X, y)

    return model
```

Listing 4.3: Excerpt from the webapp normalising the data and creating and fitting the kNN model

Before data is loaded into the model, the data is normalised using a MinMaxScaler from the "sklearn" library. This scaler transforms the data so that all values are in the range between 0 and 1. This is a necessary step for kNN classifiers, improving the accuracy of the predictor and reducing computational time for predictions.

One issue encountered with this implementation is that since kNN is a lazy learning classifier, it stores all of its data in memory after fitting. This caused an issue with an early deployment, as this exceeded the Heroku memory limit of 500MB. This issue was subsequently avoided by fitting the model dynamically whenever a request for that station is sent to the web server. Since kNN takes less than 0.01 seconds to fit its model, this is perfectly suited for our application.

4.2 Back-end API

This application uses the Python library *Flask*[3] to create a REST microservices web-server for this project. Python was selected as the language for the web server so that the web server and machine learning logic could be written in the same language and combined in the same codebase. Python also has a variety of useful libraries for dealing with large amounts of data, notably *pandas*[4], *geopy*[5] and *scipy*[6]. Flask was selected as the framework for this project because it is a lightweight, easy-to-use, microservices-focused web framework that provides all of the features this project needs.

The API is split up into four independent services with corresponding endpoints.

When the user wishes to calculate a route, a request is sent to the RoutingService, detailing the start location, the end location and when the user plans to leave. The RoutingService then finds the three closest stations, and passes each station off individually to the PredictionService.

The PredictionService uses the model to predict availability at the station under one of five categories: "empty", "very low", "low", "moderate" and "high". As detailed in Section 4.1, these labels corresponding to different levels of bike availability; "empty" corresponds to zero available bikes, "very low" corresponds to one or two available bikes, "low" corresponds to three or four available bikes, "moderate" corresponds to between five and nine available bikes and "high" availability corresponds to ten or more available bikes. This same criteria was also extended to bike stand availability. The model is populated with historical station records, stored as .csv files, which are updated every five minutes by a separate background process. The JCDecaux Dublinbikes API and OpenWeatherMap API are used to retrieve bike usage and weather data for predictions. Each prediction is sent back to the RoutingService.

The RoutingService uses these predictions to select the best stations for the journey, and uses the CycleStreets API to calculate different cycling routes for different user route preferences. The OpenRouteService API is used to calculate the walking segment from the user's current location to the starting bike station, and from the ending bike station to the user's selected destination. This information, alongside a unique ID, is finally returned to the Android app, where it is displayed in a user-friendly manner. The route is finally saved in a Google Cloud Datastore of historical generated routes.

In parallel to this, the HistoryService generates graphs showing average station usage based on historical station records. This creates a more inviting user interface in the Android app.

Finally, the FeedbackService is used for the Volunteered Geographic Information aspect of the app.

[3] Flask: https://flask.palletsprojects.com/en/1.1.x/
[4] pandas: https://pandas.pydata.org/docs/
[5] geopy: https://geopy.readthedocs.io
[6] scipy: https://docs.scipy.org/doc/scipy/reference/

After a user has completed their journey, a notification is sent to the user asking them to provide some feedback on their journey. If the user clicks on the notification, a short questionnaire appears, asking the user whether the stations they visited had a queue of people waiting for a bike or not. This is useful information that cannot be accessed through the Dublinbikes API and could be used to enhance the performance of station availability prediction. The FeedbackService receives this information, matches it up with the user's route ID and stores it in a Google Cloud Datastore.

Each of those endpoints can be deployed as separate instances, due to the microservices approach taken by this project. This makes the system easier to deploy and maintain, as changing one service in the application requires only that service to be re-built and re-deployed.

4.2.1 Infrastructure and Deployment

This application is currently deployed on a single Google AppEngine[7] Flexible instance. AppEngine provides us with an easy-to-use, automated infrastructure which requires a simple git commit to push changes to production. One major advantage of using Google AppEngine is that we can deploy code to a production environment without needing to consider the workload of managing a web server. With AppEngine, we can define the required runtime, entrypoint and infrastructure required in a google.yaml file (as shown in Listing 4.4). AppEngine takes this file and builds a Docker container in a Linux VM that we have complete control over.

When changes have been made to the back-end code, deploying the codebase is as simple as typing the command `gcloud app deploy`. Logs can be accessed from the terminal using the command `gcloud app logs tail`.

Currently, all services are hosted on a single AppEngine instance. This is due to the fact that this system has a limited number of services, and very low traffic. Using multiple instances would be more expensive, so currently a single AppEngine instance is sufficient. However, it is useful to be able to quickly scale up this system if demand increased after a general release.

Another benefit of using AppEngine is that Google Cloud has a plethora of useful tools for building webapps. Google Cloud Datastore is a simple, schema-less way of storing query-able data. Google Cloud Storage is a cheap, fast way of storing files. Both of these services are used in this webapp to store data in a fast, reliable and distributed way. By using these services instead of local file storage or a local database, it is easier to scale the system for busy usage periods. Since both of these services and AppEngine are hosted in the same Google datacenter, transaction speeds are comparable to using local I/O or a local server.

```
runtime: python
env: flex
entrypoint: gunicorn -b :PORT main:app

runtime_config:
  python_version: 3

manual_scaling:
  instances: 1
resources:
  cpu: 1
  memory_gb: 0.5
  disk_size_gb: 10
```

Listing 4.4: 'google.yaml' file used to define the AppEngine deployment

[7] Google Cloud AppEngine https://cloud.google.com/appengine

Originally, Heroku[8] was chosen as the webapp host. However, issues were encountered when large amounts of data were loaded into the model and Heroku's 512MB memory limit was reached. Heroku also encountered issues with scaling, as it does not have the in-house data storage support that Google AppEngine has. Heroku also seemed to struggle with route-generating requests, often taking over 10 seconds to calculate the route. Waiting 10 seconds to generate a route is not acceptable from a usability point-of-view, so Google AppEngine was chosen as an alternative (where route generation always takes less than 5 seconds).

4.2.2 PredictionService

The PredictionService correspondes to the "/predict" endpoint on the server. "/predict/bikes" is used to predict the availability of bikes and "/predict/bikestands" is used to predict the availability of bike stands. Both endpoints take two query parameters: "station" is the name of the station to be predicted, and "minutes" is used to make predictions of availability in the future. This service follows the prediction methodology described in Section 4.1.

In its current implementation, the PredictionService also uses the Advanced Python Scheduler[9] to update bike and weather data every five minutes. The updated weather data is stored in memory, and the updated bike data calls the "station records updater", which updates the station .csv files stored in the AppEngine Instance. This updater accesses the JCDecaux and OpenWeatherMap APIs, and writes the updated data to the corresponding station .csv file.

When the "/predict/bikes" or "/predict/bikestands" endpoints are contacted, they call the predict_availability() method. This method loads the corresponding .csv file for the selected station from the project's Google Cloud Storage bucket, normalises this data with a MinMaxScaler and fits the KNeighborsClassifier model with this DataFrame. It then calls the `predict()` and `predict_proba()` methods on the classifier, and combines this information into a JSON file. `predict_proba()` returns the predicted probability of each label being correct, signifying the classifier's confidence in the prediction. This is useful information to have when selecting the best station in the RoutingService.

4.2.3 RoutingService

The RoutingService (which is the "/route" endpoint) is used to generate routes between a user's location and their chosen destination. It takes three request parameters: a "start" and "end" location, formatted as coordinates in the format "longitude,latitude" (e.g. "54.075648,-7.118114"), and a "minutes" parameter, which is in how many minutes the journey will begin. This allows the user to plan journeys that take future availability patterns into considering, for example planning a route at 2pm for your commute home from work later in the day.

When the RoutingService starts up, a GET request is made to the JCDecaux API to retrieve a list of all bike stations and their co-ordinates. This ensures that the system is compatible with any new stations added to Dublinbikes scheme in the future. This also means that the RoutingService can easily be extended to work with any JCDecaux BSS, such as Vienna, Brussels and Lyon. This bike data is stored as a global variable that can be accessed by all future requests. We also add the co-ordinates of the stations to a KDTree from the "spatial" module of the "scipy" library. The KDTree is used to rapidly find the k nearest neighbours to a given co-ordinate. It is used by the system to find the 3 nearest bike stations to a user's location. While it does not take into account the exact walking distance between two locations, and does not take into account geographical obstructions, the rough approximation it calculates is accurate enough for this use case, and is

[8] Heroku: https://heroku.com
[9] Advanced Python Scheduler documentation: https://apscheduler.readthedocs.io/en/stable/

also calculated very quickly.

When a request is received by the service, it calls `find_best_station()` to find the best starting and ending station. It uses the previously-mentioned KDTree to find the three closest stations to the specified location. We choose three stations because it's a compromise between selecting enough stations so that we can plan the best route, while also not adding too much extra walking time (or computational time) to the user's route.

By default, we set the closest station to be the "best station". Next we iterate through the three stations to retrieve their predicted availability. For each station, we calculate their distance from the starting location using geopy's distance function. This calculates the "how the crow flies" distances between two points. We then estimate the time taken to walk this distance, using the widely accepted speed of 1.4m/s in this calculation [25]. We then pass this information to the PredictionService, which returns the predicted availability and corresponding prediction probabilities.

During deployment, it was discovered that calling an internal endpoint like this can cause issues when deploying multiple services to the same instance. To overcome this issue, we currently call the function directly from the PredictionService file instead of making a HTTP request. This decreases loading times and avoids the errors encountered with calling internal APIs from within the same instance.

After predictions have been made for each station, a selection process begins by evaluating and selecting the best station. Firstly, if any station is predicted to have "high" availability, this station is instantly returned. The reasoning behind this is that it is incredibly likely for a station with "high" availability to have at least one bike or bike stand available when the user arrives. Due to how stations are iterated, if more than one station has "high" availability, the closest station is selected. Using similar logic, the algorithm will also select a station if it has "moderate" availability, as long as the current time is not within the busiest periods for the Dublinbikes BSS (considered to be between 8am and 11am, as well as between 4pm and 7pm).

If no station has "high" availability, the algorithm iterates through the remaining availability levels in order of availability: "moderate", "low" and "very low". If just one station has the highest label, it selects this label. If multiple stations have this label, it looks at the predicted probabilities and selects the station that is most likely to have had its previous estimate understated. This selection method is not perfect, but it is designed to ensure that it is very unlikely for a user to arrive at a station with no bikes available.

When the best stations have been selected, the service uses the CycleStreets API to generate the fastest, quietest, shortest and balanced routes between the two stations. Due to the design of the CycleStreets API, this requires three seperate API requests. While this is slower than fetching just one API request, it makes the user experience in the Android app, as users can quickly switch between different route types. Next, the walking route from the user's location to the starting station and from the ending station to the user's destination is calculated using the OpenStreetMap API.

Finally, these routes, alongside the start and end station names and a unique identifier, are stored in a JSON object and returned to the user. This JSON object is also stored in a Google Cloud Datastore, where it can be accessed for future reference by the FeedbackService.

4.2.4 HistoryService

The HistoryService ("/history") is used to generate graphs detailing the average historical usage of a given station, as well as usage for the current day so far. This graph is displayed in the station

view of the Android application, as described in Section 4.3.

This service works by reading the 5000 latest recorded records for a given station. These records correspond to the count of bikes availability for the last month. The function then uses the Python library numpy's `polyfit()` and `poly1d()` functions to generate average values for this station, as shown in Listing 4.5.

```
import numpy as np

x, y = (filtered_data['time_of_day'],
        filtered_data['available_bikes'])
    coefficients = np.polyfit(x, y, 50)
    poly_func = np.poly1d(coefficients)

    new_x = np.linspace(18000, 86400)
    new_y = poly_func(new_x)

    result = {"graph": []}

    for x_value, y_value in zip(new_x, new_y):
        result['graph'].append((x_value, y_value))
```

Listing 4.5: Excerpt from the webapp generating average bike availability data for a given station

This information is then returned to the Android app in JSON format (as described in Listing 4.6), in an array detailing the number of bikes available at each moment in time.

```
{
  "graph": [
    [
      18000.0,
      23.56069639778138
    ],
    [
      19395.918367346938,
      27.72044546715228
    ],
    [
      20791.836734693876,
      29.248691958123036
    ],
    [
      22187.755102040817,
      27.96651313789668
    ],
    ...
  ]
}
```

Listing 4.6: JSON snippet of the payload returned by the "/history" endpoint

4.2.5 FeedbackService

The FeedbackService ("/feedback") collects Volunteered Geographic Information (VGI) from the user after a journey has been completed. It receives a JSON payload from the Android app

detailing how long the user had to wait at each station before getting a bike. The structure of this data is not important, as it is currently not manipulated by the FeedbackService. In its current incarnation, the FeedbackService simply saves the request's payload as a Feedback object in the Google Cloud Datastore. In the future, it is envisioned that this VGI could be used to tune the model, adding a new aspect to the prediction that cannot be accessed from the JCDecaux API. However due to a lack of data and limited timeframe for this project, this feature is not currently implemented.

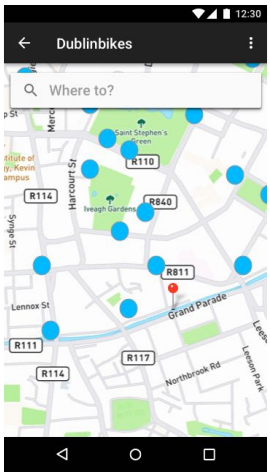
4.3 Android app

The user-facing portion of this project is an Android app that lets users generate smart routes to a given destination. It communicates directly with the back-end endpoints described in Section 4.2.

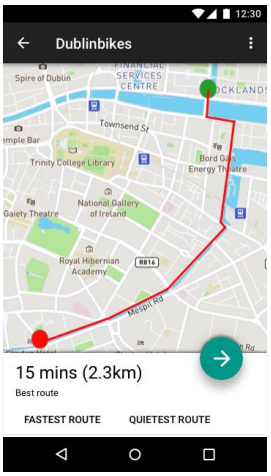
In the early stages of development, it was decided that it was important that the user interface was as simple and minimal as possible and follow the Material Design guidelines[10] as closely as possible. The main screen of the app shows an interactive map of all of the bike stations near the user's current location. This screen also contains a search bar where users can type in their destination. When the user enters a destination, a request is sent to the RoutingService, requesting a route to this destination.

When the RoutingService returns a response, the route screen shows the details of the route. This includes a mapping of the route, the details about the route length and alternative routes (such as the fastest and quietest). After selecting a route, the app displays a list of directions for the user's generated route.

At the start of development, Sketch was used to create mock-ups of the desired user interface shown in Fig. 4.5. The app would be centered around a single MapView, showing all of the stations in the Dublinbikes BSS. When a route is entered, the MapView is altered to show the generated route. By using a single MapView, the application can transition between views easily without completely reloading the map.



(a) The main screen shows you a map of nearby bike stations with a search bar to enter your destination



(b) The route screen shows you the generated route with a card showing extra details

Figure 4.5: Early-stage mockups of the dBikes Planner Android app

[10] Material Design guidelines: https://material.io/design/

4.3.1 App development platform

Originally, React Native was selected as the framework on which to develop this application. This was chosen because React Native is easy to develop for and is a cross-platform framework that allows you to develop iOS and Android apps simultaneously. Expo was also to be used, allowing apps to be created without having to build the app in an IDE such as Android Studio or XCode. However, early in the development process using React Native, it was discovered that the platform's support for the Google Maps API was relatively poor. While React has support for some Google Maps libraries, including *react-native-maps*[11] and *react-native-maps-directions*[12], these libraries have very limited support in Expo itself and require Android Studio to be used to their full capabilities. Some of these libraries are also poorly supported, with many issues that are not actively being fixed. As well as this, some Google Maps API services are not implemented at all, such as the Google Places Autocomplete API[13]. To implement this elegantly in our application, we would be required to inject native Android code into the codebase, which could be a time-consuming process.

For these reasons, we decided to move development to a native Android app using Java. While this means that we are no longer concurrently developing an iOS app, a native Android app means we can use the Google Maps API natively in our app and use native Android components in our application in contrast with React Native's hybrid components. This leads to a more consistent design, more in line with the wider Android design language.

4.3.2 App overview

When the user opens the application for the first time, they are greeted with a prompt requesting the user's location (Fig. 4.7a). This is used to show the user's location on the map, to show nearby stations and to act as the starting point of a user's generated route. After accepting this prompt, the main "map view" is displayed to the user 4.7b. This view shows the user's location on a map, alongside markers representing each station in the Dublinbikes BSS. The colour and label of each marker is dependent on the number of bikes or spaces at a given station (as shown in Fig. 4.6). All station details (including their location) are fetched on app launch from the JCDecaux Dublinbikes API. This ensures that the app always contains the most up to date information about the stations in the Dublinbikes BSS.

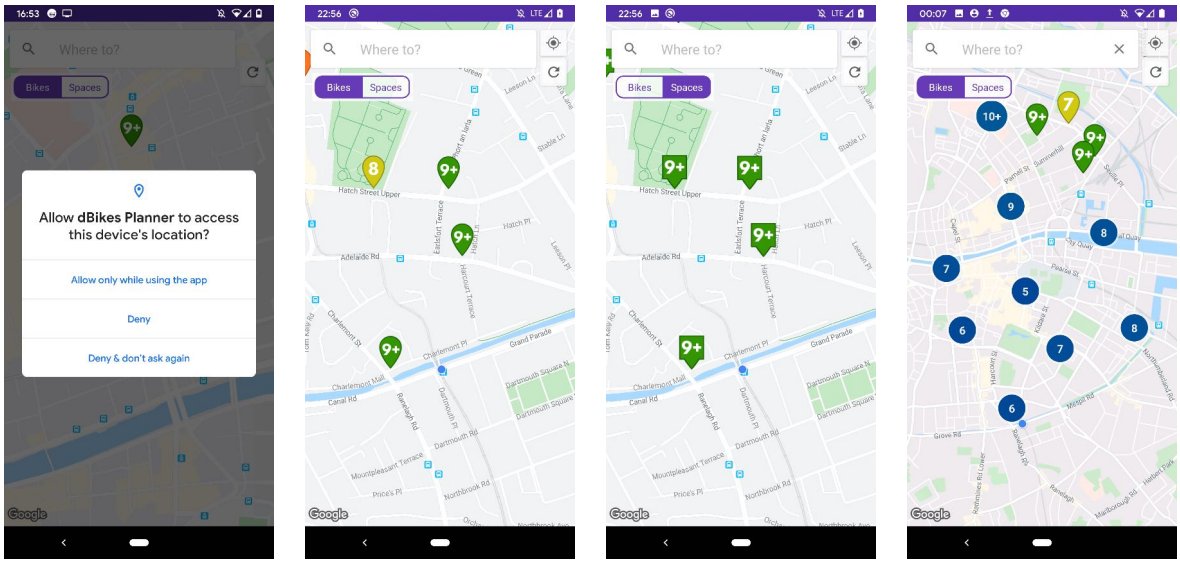
Figure 4.6: All possible station markers, describing the availability of bikes at a station.

For this view, the Google Maps Android SDK[14] is used extensively to plot stations, routes and the user's location on an easy-to-use map interface. A single Activity is used throughout the app, using a single SupportMapFragment to display both stations and routes. This approach was chosen over using multiple activities to make the app more fluid. By using one activity to display both stations and routes, the same map is used, which is much less jarring and efficient than loading a new activity when a user switches between these two views. While this makes the source code slightly more difficult to comprehend, this is an acceptable trade off for a superior user experience.

[11] react-native-maps: https://github.com/react-native-community/react-native-maps
[12] react-native-maps-directions: https://github.com/bramus/react-native-maps-directions
[13] Google Places Autocomplete API: https://developers.google.com/places/android-sdk/autocomplete
[14] Google Maps Android SDK: https://developers.google.com/maps/documentation/android-sdk/

In the top right of the screen, the GPS button can be used to zoom to the user's current location. Below this, the refresh button fetches the latest information from the JCDecaux API to display the latest bike and space availability information to the user. In the top left, a SegmentedGroup from the Android library 'android-segmented-control'[15] is used to toggle between displaying the availability of bikes and spaces at each station. This changes the marker for each station from a circular shape to a square shape (as shown in Fig. 4.7c), signifying that the label on each marker represents the availability of spaces at the station.

When the user zooms out in the map, nearby stations start to cluster together (as shown in Fig. 4.7d). Clustering is used to allow the user to see the general location of bike stations while not obstructing the map itself. This cluster grouping is generated using a ClusterItemManager and a customised DefaultClusterRenderer.



(a) When the app is launched for the first time, a request is made to access the user's location.

(b) The initial map view, showing the user's location and the nearest stations, with icons describing the number of bikes available at a station.

(c) The map view, showing the number of spaces at each bike station. A different icon is used to represent the availability of bikes and spaces.

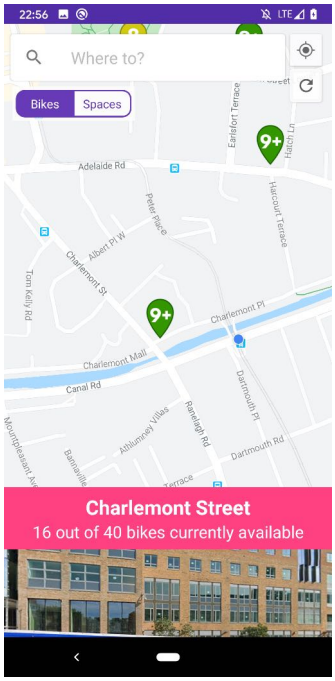
(d) Clustering is used to group nearby stations together, to show the relative positions of stations while not obstructing the map itself.

Figure 4.7: The main map view of the Android app

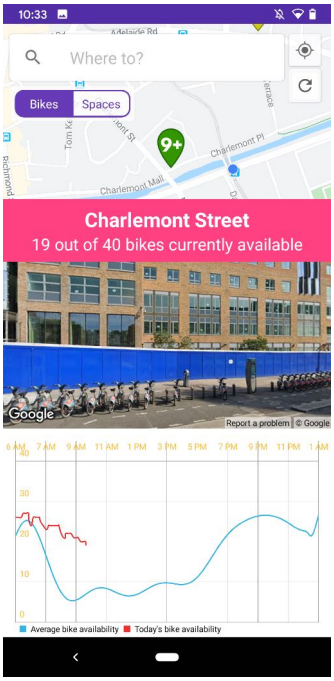
When a user clicks on a station, a bottom sheet appears from the bottom of the screen (shown in 4.8a). This is implemented by using the BottomSheet component from Google's Material Design library[16]. The title of this bottom sheet displays the name of the station and the current level of bike or space availability. Dragging up this sheet displays a Google Street View embed of the station's location, which can be moved around to view the general area around the station (as shown in 4.8b). Underneath this, a graph is displayed, showing the average of availability of bikes (or spaces) alongside today's availability of bikes (or spaces). This graph is created using data fetched from the "/history" endpoint, as detailed in Section 4.2.4. This graph allows users to see how busy a station is today compared to its average availability. This bottom sheet can be dismissed by swiping down or by clicking the "Back" button on your phone.

At the top of the map view, there is a search bar with the hint "Where to?". This is an *AutocompleteSupportFragment* from Google's Places SDK for Android[17]. When the user clicks on this

[15] android-segmented-control: https://github.com/Kaopiz/android-segmented-control
[16] BottomSheet Material Design Guidelines: https://material.io/components/sheets-bottom
[17] Google Places SDK: https://developers.google.com/places/android-sdk/



(a) When a user clicks on a station, the bottom sheet below peeks, showing the current status of the station.



(b) By scrolling up this bottom sheet, a Google Street View of the station location is shown, alongside a graph detailing the average availability and today's availability.

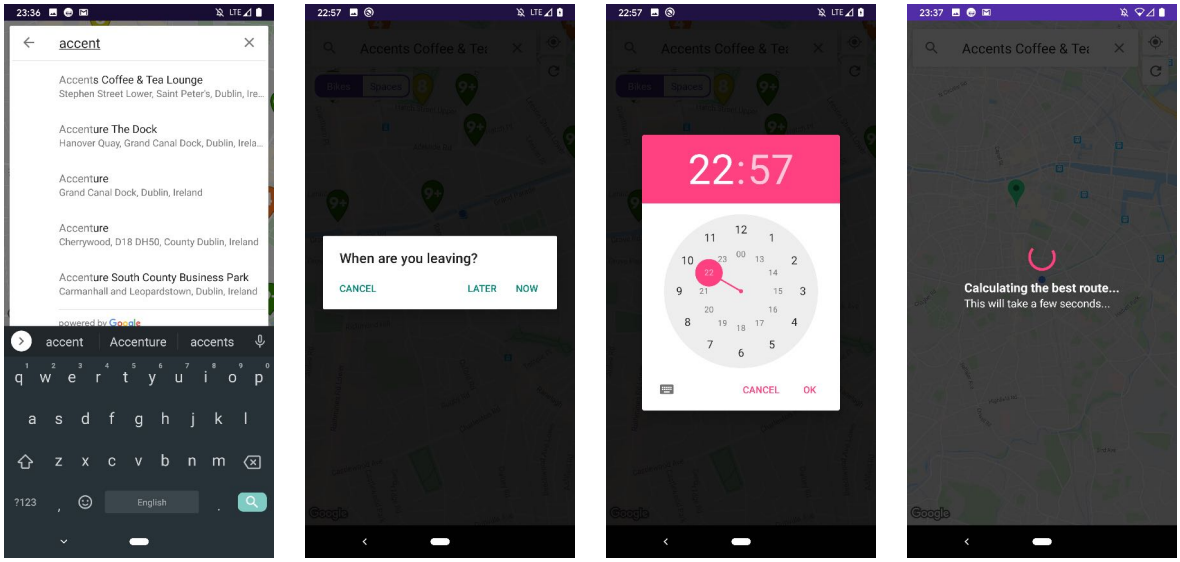
Figure 4.8: The station view of the Android app

search bar, the search bar expands to fill the screen and the keyboard appears (as shown in 4.9a). As the user types in their destination, suggestions appear on the screen. These suggestions are limited to the Dublin city region, and include streets, neighbourhoods, landmarks and businesses.

When a user selects a destination, a *modal* pops up, asking the user if they are starting their journey now or in the future (shown in Fig. 4.9b). If the user selects "Later", a TimePickerDialog is displayed, letting the user select exactly what time they want to leave at 4.9c. After a time has been selected, or the user specifies that they are leaving "now", a HTTP GET request is made to the "/route" endpoint of the dBikes Planner back-end, as detailed in Section 4.2.3. While the app waits on a response, it displays a loading wheel alongside a black shim that darkens the background 4.9d. This request typically takes around 4 seconds to receive a response.

After the response has been received, the map zooms out to display the generated route (Fig. 4.10a). This consists of the user's location, a dotted line representing the walking route to the starting station, a solid line representing the cycling route to the ending station, and a dotted line representing the walking route from the ending station to the user's destination (as shown in Fig. 4.10c). The bottom sheet describes the duration and distance of the route, including a breakdown into walking and cycling segments.

The bottom of screen shows a row of chips, letting the user change between four types of cycling route: the quietest route, the fastest route, the shortest route and the balanced route. Clicking on these chips changes the type of the displayed route (as shown in Fig. 4.10b), as well as the route statistics. The "balanced" route is selected by default, as it is the most suitable route for most cyclists.



(a) The AutocompleteSupportFragment, used to autocomplete places using the Google Maps API. Here the user enters their destination.

(b) The modal shown to the user, asking if they will start their journey now or in the future.

(c) A TimePickerDialog is used to select at what time the user will be leaving at.

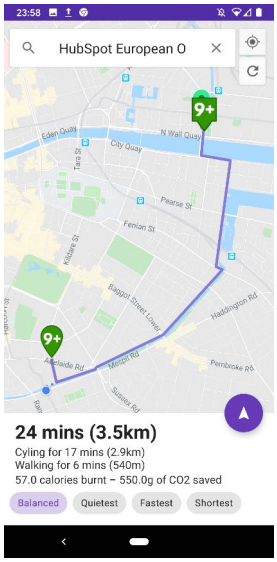
(d) While the route is being calculated, this loading screen is shown to the user.

Figure 4.9: The process of setting a destination for your route in the Android app

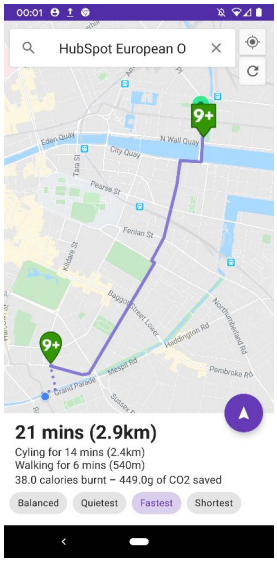
By clicking the purple floating action button, the bottom sheet scrolls down to display a list of cycling directions for the journey (shown in Fig. 4.10d). This is achieved using a RecyclerView and a custom Adapter, allowing us to display a dynamic number of directions without encountering degraded performance.

When a user clicks the purple floating action button, the app interprets that as the start of a journey. When the app estimates that the journey has been completed, a notification is sent to the user (shown in Fig. 4.11a). This notification requests Volunteered Geographic Information from the user about how busy a station was, and if the station had a long waiting time for a bike or a space. Clicking on the notification launches the FeedbackActivity, displaying two questions about if there was a queue at either station in the user's journey (shown in Fig. 4.11b). When the user submits this feedback, a JSON payload is sent in a POST request to "/feedback/<route_id>" (as described in Section 4.2.5). The user is returned to the map view, and a toast pop-up thanks the user for their feedback (shown in Fig. 4.11c).

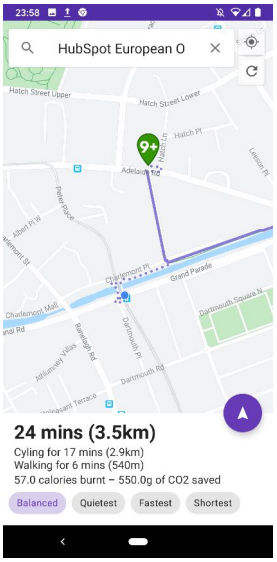
If the user chooses not to provide feedback, they can simply dismiss the notification in their notification drawer, and the application will continue to work as described.



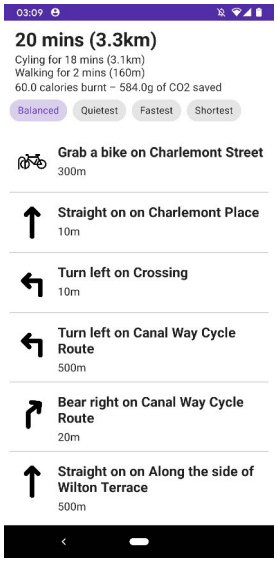
(a) The route view shown to the user, showing the starting station, ending station and the walking and cycling route.



(b) The fastest route generated by the app. This can be changed by clicking on the chip buttons at the bottom of the screen.

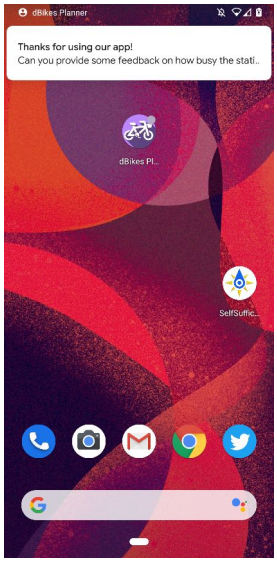


(c) In the route view, the walking segment is visualised with a dotted polyline, while the cycling segment is visualised with a solid polyline.

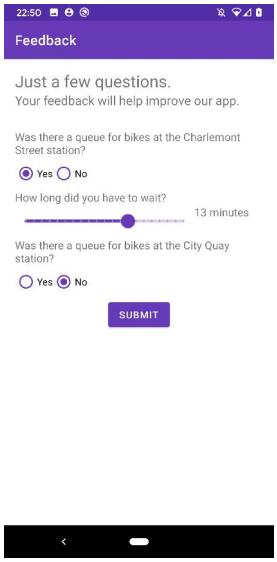


(d) By clicking the purple floating action button, the app displays a RecyclerView showing the directions to take.

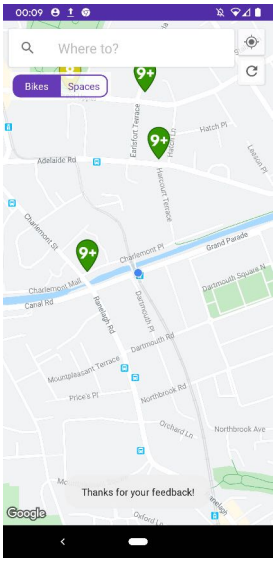
Figure 4.10: The route view, as shown to a user after entering their desired destination.



(a) After the app estimates that the user has completed a journey, a notification is displayed, requesting feedback on the generated journey.



(b) The FeedbackActivity asks if either station in the journey featured queues, and if so, how long the user was waiting.



(c) After the feedback is submitted, the user is returned to the main map view and shown a toast pop-up, thanking them for their feedback.

Figure 4.11: The feedback section of the Android app

Chapter 5: Evaluation

In this project, the two main aspects to evaluate are the model that predicts bike and space availability, and the usability of the Android client application. This chapter details how each of these components are evaluated, and the results of this evaluation.

5.1 Model Evaluation

To evaluate the effectiveness of the model, the precision, recall and f1-score of the KNeighborsClassifier approach are analysed. Precision is defined as the proportion of retrieved results that are relevant. In this case, precision is the percentage of correct predictions for a category as a proportion of the total number of predictions for that category. Recall is defined as the proportion of relevant results that are retrieved. In this case, precision is the percentage of correct predictions for a category as a proportion of the total number of actual cases for that category in the test set. The f1-score is the harmonic mean of the precision and recall, trading off precision against recall to calculate an overall score for the performance of the classifier. The higher the f1-score of a classifier, the better the level of performance.

To evaluate the model, the "metrics" package in the "sklearn" library is used. To split the dataset into training and test data, TimeSeriesSplit is used. This ensures that the training data only contains data from before the test data in the time series, ensuring that the evaluation is as realistic as possible. The *n_splits* parameter is used to control the size of the test data to differentiate between short, medium and long-term predictions. For this section, short-term predictions correspond to predictions in 10 minutes time, medium-term predictions correspond to predictions in 30 minutes time and long-term predictions correspond to predictions in 60 minutes time.

Google Colaboratory was used to run these evaluations. The Colaboratory script selects each station record from Google Cloud Storage. For each station, it splits the data into 8,000 splits. This number is selected to ensure that each training split contains at least six entries. For each split, the predictions after 10, 30 and 60 minutes are recorded, corresponding to short-term, medium-term and long-term predictions. This process is completed for each split, and repeated for each individual station. Using the classification_report() method from *sklearn*'s *metrics* package, metrics were calculated for short-term predictions (as shown in Table 5.1a), medium-term predictions (as shown in Table 5.1b) and long-term predictions (as shown in Table 5.1c). These metrics are compared in Fig. 5.1a.

Comparing these results, we notice that short-term performance is good, with a weighted average f1-score of 0.77. This performance decreases as predictions are made further into the future, to an f1-score of 0.666 for medium-term predictions and of 0.603 for long-term predictions. It is also noted that the metrics for the "empty" and "very low" categories are quite poor in general. This can be partially accounted for by the fact that these categories are very small (corresponding to 7% and 5% of entries respectively). These categories also represent a very small change in bike availability: "empty" stations have zero bikes available, while "very low" stations have one or two bikes available. Another reason for poor performance is that the K Nearest Neighbours approach does not take into account the ordinal aspect of categories when making predictions. These scores do not differentiate between an "empty" station being classified as "very low", which in a real-world scenario is a very small error, in comparison to "high", which is a much bigger

	precision	recall	f1-score	support
empty	0.409	0.391	0.4	555
very low	0.232	0.202	0.216	406
low	0.554	0.561	0.558	1098
moderate	0.663	0.682	0.672	1298
high	0.938	0.943	0.94	4643
macro average	0.559	0.556	0.557	8000
weighted average	0.768	0.772	0.77	8000
accuracy			0.772	8000
root mean squared error			0.71125	8000

(a) Evaluation results for short-term predictions (10 minutes)

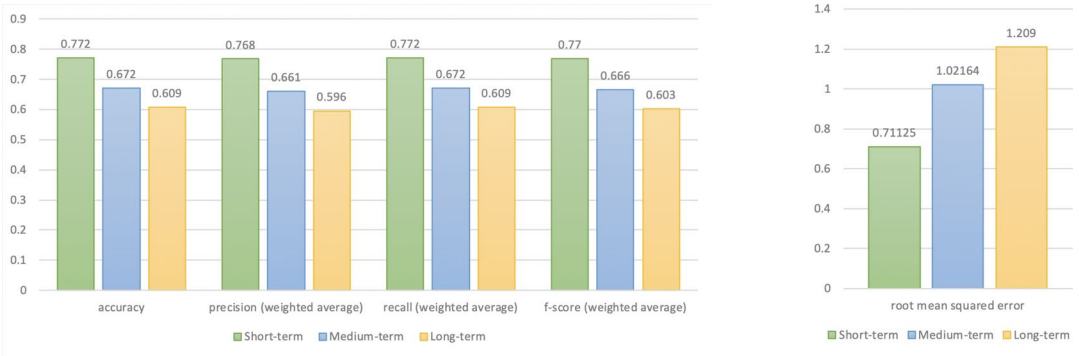
	precision	recall	f1-score	support
empty	0.303	0.291	0.297	573
very low	0.182	0.131	0.152	375
low	0.402	0.409	0.406	1125
moderate	0.479	0.464	0.471	1327
high	0.86	0.888	0.874	4600
macro avg	0.445	0.437	0.44	8000
weighted avg	0.661	0.672	0.666	8000
accuracy			0.672	8000
root mean squared error			1.02164	8000

(b) Evaluation results for medium-term predictions (30 minutes)

	precision	recall	f1-score	support
empty	0.225	0.22	0.223	577
very low	0.163	0.098	0.122	388
low	0.32	0.344	0.332	1072
moderate	0.376	0.358	0.367	1349
high	0.809	0.836	0.822	4614
macro avg	0.379	0.371	0.373	8000
weighted avg	0.597	0.609	0.603	8000
accuracy			0.609	8000
root mean squared error			1.209	8000

(c) Evaluation results for long-term predictions (60 minutes)

Table 5.1: The metrics associated with short-term, medium-term and long-term predictions.



(a) A bar chart comparing the general performance metrics of the model.

(b) A bar chart comparing the root mean squared error of the model.

Figure 5.1: Bar charts comparing the performance of the model for each type of prediction. Short-term, medium-term and long-term predictions correspond to predictions in ten minutes time, thirty minutes time and sixty minutes time respectively.

error. In order to better evaluate the true error in predictions, the root mean square error of the results was calculated (following the findings by L. Gaudette, et al. [26]). This was calculated by substituting each station category with an integer value between 0 and 4, based off its position in the scale from "empty" to "high". Using the "mean_squared_error()" function from "sklearn", the root mean squared error for each prediction type was calculated (as displayed in 5.1b).

These calculations show an average root mean square error of 0.71125 for short-term predictions. This is a relatively low score, meaning that on average, the predictions calculated are assigned the correct category, and any incorrect predictions are generally to a adjacent category. Similarly, an average root mean square error of 1.209 for long-term predictions, which is marginally less accurate. While this method of evaluation is not entirely precise, it gives us a better idea of the quality of the model by taking into consideration the fact that this is an ordinal classification problem. These results show that the model generally makes the correct prediction, and in the cases where it is incorrect, it generally assigns the prediction the category adjacent to the correct category.

5.1.1 Comparison to previous predictive solutions

This report initially hoped to directly compare the performance of its model to the achievements of other researchers, such as and Chen, et al. [13] and Kaitenbrunner, et al. [12]. However, by transitioning this project's model from a regression problem to a classification problem, it makes it very difficult to directly compare the results obtained in this paper to previous findings. Since the data used by these papers is not publicly available, it is impossible to use binning to convert their solutions into classification problems to directly compare their results.

Since the Average Root Mean Square Error (ARMSE) calculated by these papers is based off the error in the number of predicted bikes, to make a rough comparison, this project's RMSE must be converted from a basis of category to a basis of bike availability. Since the number of bikes represented by each station category is different (an "empty" station always contains zero bikes, while a "moderate" station could contain between five and nine bikes), an average of 3 bikes is chosen as the rough difference between categories. By multiplying the RMSE obtained in Fig. 5.1b by 3, we can obtain a rough estimate of what the ARMSE for the k Nearest Neighbours (kNN) model would be, in relation to bike availability. These results are visualised in Figure 5.2).

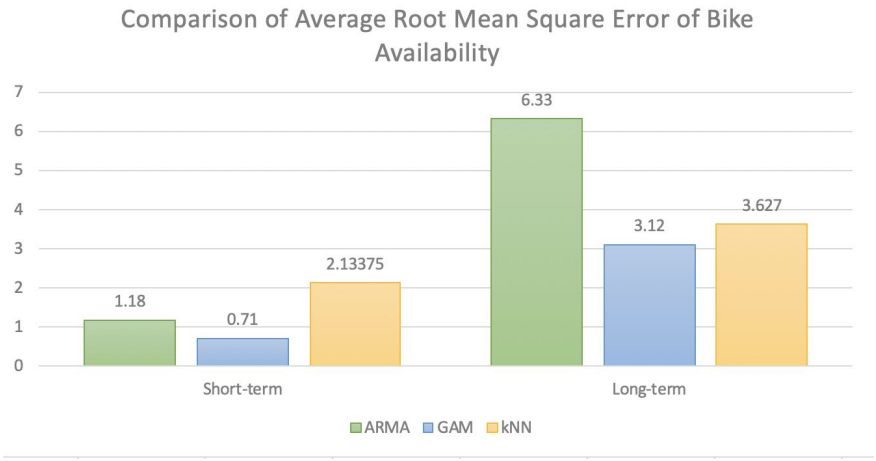


Figure 5.2: Comparison of the Average Root Mean Square Error developed in this project to the GAM predictor developed by Chen, et al. [13] and the ARMA predictor developed by Kaltenbrunner et al. [12]

While this process is simply a rough estimate of the effectiveness of the model in comparison to previous developments, it provides interesting insights. It shows that the kNN (k Nearest Neighbours) model performs slightly worse than the GAM (Generalised Additive Model) and ARMA (Auto-Regressive Moving Average) implementations, but the error itself is quite low at 2.13375 bikes. For long-term predictions, the kNN model performs very well, performing better than the ARMA model with an error that is 42.7% smaller. The performance by the kNN model is much closer to that of the GAM. This is likely explained by the fact that the GAM and kNN predictors both take weather data into account when making predictions, while the ARMA model only takes historical data into account.

Overall, the kNN model performs sufficiently for the requirements of this project. The performance of the kNN model is between that achieved by Kaltenbrunner et al.'s ARMA model [12] and Chen, et al. [13] Generalized Additive Model. Combined with the fast fitting and predicting speeds of the kNN model in comparison to previously mentioned models, the kNN model is an effective solution for this project's prediction task.

5.2 App evaluation

To evaluate the usability of the Android app, a user test was designed. Originally, it was planned to use popular cycling Facebook groups to find willing test participants, such as the Dublin Cycling Campaign Group. However due to the current restrictions on travel due to the COIVD-19 lockdown, this idea was decided against. Instead, this user test was promoted on social media to find participants who would be interested in testing a Dublinbikes route planner application. 23 applications were received in total.

A document was shared to the test participants, introducing the application to them and describing the basic functionality of the app. The document also provided a link to a signed .apk file that they could install on their phone. Initially, it was planned to use Google Play's internal beta testing system to distribute the app in a friendlier way to non-technical people, but due to coronavirus-related delays, the app approval process was too slow for our needs. This document did not provide a strict guide on what exactly to test; instead, the participant was instructed to explore the application, "play around with the features" and test any functionality they discover.

Due to the current national lockdown, some modifications were made to the application to make

the app easier to test from home. Firstly, the location was set to Charlemont Luas Station by default. This location was chosen because it is close to three different stations of various average levels of availability. This provided a way for the user to test the route-planning process without being located in Dublin city centre itself. Secondly, the feedback feature was modified so that a notification would be sent to a user ten seconds after accepting a journey. Apart from these minor changes, the application functioned as described in Section 4.3.2.

After the user tested the app itself, they were sent a short questionnaire. This questionnaire is based on the System Usability Scale (SUS) [27]. It contained all ten questions from the SUS, but the phrase "this system" was replaced with "this app", to make the questionnaire more accessible and relevant to this project. The questionnaire also contained a free-form text question where participants were invited to "provide any other feedback on (their) experience using the app". This questionnaire was GDPR compliant, providing a GDPR notice outlining the use of data obtained from this questionnaire, as well as the participant's rights to access, view and edit this data. (This questionnaire is provided as Appendix 6.2)

In total, 21 out of 23 interested parties completed the user test. Overall, the results (as shown in Table 5.2) were overwhelmingly positive (as visualised in Fig. 5.3).

By following the System Usability Scale scoring system [27], the application has a usability score of 88.81. Following the "Curved Grading Scale for the SUS" defined by J. Lewis, et al. [28], our score corresponds to an A+ grade in usability, corresponding to the 96-100 Percentile range. Based off the findings by A. Bangor, et al. [29], this score also corresponds to an "excellent" level of user-friendliness. While these results are not necessarily from active Dublinbikes users (due to the COVID-19 lockdown), they do show an accurate evaluation of the usability of the application itself and the level of consistency and intuitiveness of the user interface.

The questionnaire also contained a general feedback section, unrelated to the System Usability Scale, where testers were invited to provide any general feedback on their experiences using the app. Many participants complimented the user interface; the app was described as "a breeze to use" with a "very clean UI". Another user stated that they "knew instantly what the icons on the map were and what they were for without really thinking they were just intuitively where (they) thought they would be". The simplicity of the app was also commended, with multiple users noting that the app was "very intuitive and easy to use". One user stated that "it looks like wizardry to me how well it works". I think this shows that the app is very usable, which is confirmed by the positive results from the System Usability Scale questionnaire.

A surprisingly popular feature was the description of calories burned and CO2 saved when displaying a route. Five individual users mentioned how this was a useful feature to have, which could indicate that this would be a useful aspect of the application to expand. One user also praised the general accessibility of the app's UI, praising the contrast of colours throughout the app and the "easily visible" text and other user interface components.

Some users complimented the accessibility of the application to new cyclists. One user mentioned how they "loved the option for a quiet route and street view of a station" and that they "can imagine it being popular with beginner cyclists". Other users really appreciated the ability to choose between multiple cycling routes, and praised the ability to quickly switch between the quietest and fastest routes.

The feedback also contained suggestions to improve the application. One user suggested that tapping the title bar of a station should expand the station view, which has since been implemented in an updated version of the app. Other suggested enhancements for the station view included colour-coding the title bar to the availability of bikes, and providing a full-screen view for a station's graph showing historical availability.

Other users suggested enhancing the CO2 and calories burned aspects of the app by taking into

	Strongly disagree	Disagree	Undecided	Agree	Strongly Agree
I think that I would like to use this app frequently.	1	0	5	11	4
I found the app unnecessarily complex.	17	4	0	0	0
I thought the app was easy to use.	0	0	1	7	13
I think that I would need the support of a technical person to be able to use this app.	16	4	0	0	1
I found the various features in this app were well integrated.	0	0	0	10	11
I thought there was too much inconsistency in this app.	15	6	0	0	0
I would imagine that most people would learn to use this app very quickly.	0	0	0	6	15
I found the system very cumbersome to use.	12	8	1	0	0
I felt very confident using the app.	0	0	1	6	14
I needed to learn a lot of things before I could get going with this app.	15	4	2	0	0

Table 5.2: Questionnaire results from user test, showing how many people selected each option for each question in the questionnaire

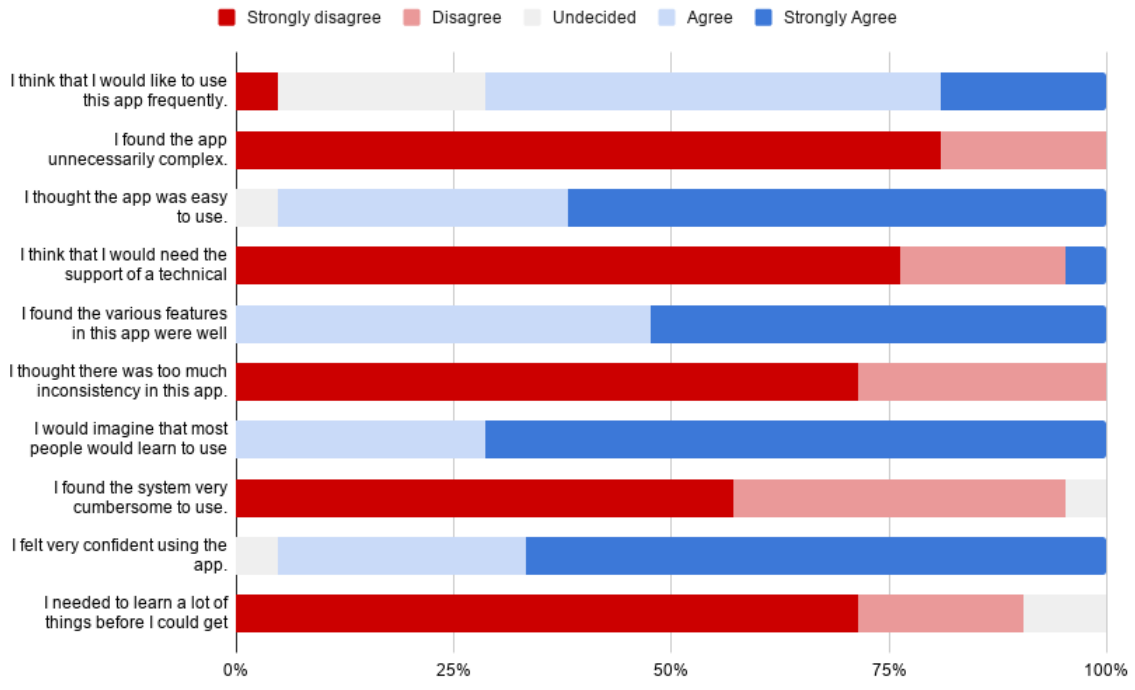


Figure 5.3: Visualisation of the distribution of the System Usability Scale questionnaire results.

account the walking part of the route when calculating these features. While this is a non-trivial feature to implement since this data is not provided by the route-planning API, this could be a potential feature for future versions of the app. Another larger suggested feature was to display the closest station to an area without generating a route. This would not be difficult to implement, but would require changes to the UI of the app to elegantly implement, which is not a core part of this project. Other smaller suggestions that have been implemented in the updated app include a renaming of the hint for the Auto-complete search bar from "Search" to "Where to?" and a reduction of the priority of the app's notifications.

Overall, the user test was a success and provided the project with a positive evaluation, as well as useful feedback and suggestions for future development. Two users even suggested that they would be willing to pay for this app on the Google Play Store, which is a positive affirmation of the quality of the app itself.

Chapter 6: Conclusions and Future Work

This concluding chapter discusses in-depth the potential future directions to bring this project, looking at both smaller and large-scale enhancements. Finally, this chapter concludes with a discussion of the accomplishments of this project as a whole.

6.1 Future Work

While this project achieves its goals, there are still many improvements and enhancements that could be made.

As discussed in Section 5.2, the individual feedback from the user test indicated potentially useful new features, such as the ability to find the nearest station to a destination without generating the route, which would be a useful feature for established cyclists who already know their preferred route. Another small-scale enhancement would be for the application to recognise when it would be quicker for a user to simply walk between two destinations that are in close proximity, instead of taking a longer, slower route on a bike. These would be useful features to implement and are supported by the feedback provided in the questionnaire.

One obvious extension of the application would be to incorporate the Volunteered Geographic Information (VGI) obtained from users into future route calculations and predictions. Currently, after a user completes a route, they receive a notification requesting feedback on wait times at stations. This information could be incorporated in route generation, for example if a station always has a long queue during rush hour. This enhancement can also be combined with the research by Chen, et al [13] in the area of prediction of waiting times at stations. While this would add a new element of complexity to predictions, it could be a worthwhile endeavor if it improved the usefulness of generated routes. This feedback system can also be extended to retrieve other information from the user that cannot be obtained from the Dublinbikes API.

There is also space for improvement in the area of availability prediction in this project. While the k Nearest Neighbours approach works sufficiently for this project, it struggles to classify for the "empty" and "very low" categories. This is partially due to the small size of these categories, with "very low" only applying when a station has 1 or 2 bikes. I propose that the "very low" and "low" labels could be combined to try and increase the precision and recall of these categories.

While k Nearest Neighbours works sufficiently, improved results may be achieved by returning to regression-based solutions. While the regression-based models researched in this paper suffered from slow fitting times, there may exist other regression-based solutions that would be suitable for a real-time web API implementation such as this project.

In order to release this project for use by the public, changes would be required to the infrastructure to accommodate this. Google AppEngine Flex instances are expensive to run, and would need to be scaled up to handle heavy traffic during peak usage hours. A migration to a Google AppEngine Standard instance would be required to reduce costs, but this would require a re-design of how this application handles background processes and threads. The Google StreetView API is also quite expensive, costing $7 per 1,000 dynamic StreetView loads (outside of the basic free provisions of 28,000 loads). This cost could prohibit the public release of this application, so this feature

should be substituted with an open-source alternative, such as Mapillary[1]. While these changes to infrastructure are not difficult, they would require time to implement to ensure cost-efficiency and scalability of the application.

Due to the project's deadline, this project does not implement a robust test suite. If this project is developed further, unit and integration tests for the front-end and the back-end would ensure that the system functions as required and isn't hindered by future feature implementations.

6.1.1 Large-scale Enhancements

While the previously mentioned enhancements are small in scope, there are some much larger changes that could be made that would increase the scope of the application.

One major feature would be to integrate this app with other modes of public transport, introducing multi-modal trips. This would allow users to combine cycling trips with bus, train or tram segments. This would be especially useful for users located outside of Dublin city centre, where Dublinbikes is currently not available. In these locations, Dublinbikes can only be used in conjunction with other modes of public transport.

To achieve this, we could use Transport for Ireland's Real Time Passenger Information API. This API provides an endpoint where for each bus, train and tram station in Ireland, the API provides information about when the bus, train or tram will arrive at the station. This information is based of GPS data retrieved from vehicles, leading to more accurate predictions of arrival times in comparison to using static bus or train timetables. While applications such as Google Maps[2] and Transit[3] are currently available in the App Store for multi-modal trips, neither of these apps take the predictive element of multi-modal journey planning into account or integrate tightly with bike-sharing schemes.

Multi-modal journey planning is a difficult problem to solve. Previous studies have found that minimising the maximal travel time of a multi-modal journey planner that takes uncertainty into account is an NP-Hard problem [30]. This problem can be simplified by using published timetables, such as the machine readable static General Transit Feed Specification (GTFS)[4] which most transport companies use to provide schedule information. However this can be unreliable and cause missed transfers between different modes of transport. While the introduction of support for multi-modal trips would be a useful enhancement to implement, it would greatly increase the scope of the project and add additional time complexity to journey planning.

In regards to how this approach could integrate with the current project, the existing RoutingService could be extended to support multi-modal trips. This service would require some major refactoring, as it currently assumes that all routes start and end with a bike station. Instead, a route could consist entirely of public transport options, or consist of a combination of public transport and cycling. A graph could be constructed with each public transport or bike station as a node in the graph, and an adapted version of the A* search algorithm could be used to find the quickest paths. This algorithm would also have to use the existing PredictionService to ensure that bike stations used in a route will have bikes available at that given time. Similarly, changes would have to be made to the Android application to support the generation of multi-modal trips, and care must be taken to ensure that the simplicity of the app's user interface is preserved.

While this is achievable, adding support for multi-modal trips would greatly complicate the RoutingService, as it now has to use route-finding algorithms to find the best route, in contrast to

[1] Mapillary: https://www.mapillary.com/
[2] Google Maps: https://maps.google.com/
[3] Transit: https://transitapp.com/
[4] GTFS Static Overview: https://developers.google.com/transit/gtfs

the current implementation where routes are generated by the CycleStreets API. I think adding multi-modal trips would take away from the main appeal of this application, which in my view is the ability to quickly generate cycle routes around a city. For that reason, I think multi-modal routefinding would work best in a standalone, public transport-focused application that incorporates multi-modal planning with the predictive system for bike-sharing schemes developed in this project.

Outside of multi-modal trips, other possible enhancements exist that would add useful features to the app for cyclists. One such feature would be to include bike racks from Dublin City Council in the app, allowing journeys to be calculated for users who would like to use their own bike instead of the Dublinbikes BSS. Dublin City Council provides approximately 8,000 bike racks across Dublin [31], which would greatly expand the appeal of the app to users based outside Dublin city centre. While these bike racks do not have an API that describes if they are currently being used or not, the nearest cluster of racks could be displayed, and the user could use the map view of the app to find other nearby bike racks. This would be a useful feature to implement, because while a website displaying the location of bike racks in Dublin currently exists[5], no mobile apps currently exist that display public bike racks in Dublin or incorporate route-planning. A potential additional feature would be to allow users to add new bike racks to the app, incorporating another aspect of volunteered geographic information into the application.

Another enhancement would be to integrate support with the Bleeperbike[6] bike-sharing scheme in Dublin. Bleeperbike is part of the latest generation of dockless bike-sharing schemes that are increasing in popularity around the world [32]. These schemes differ from third-generation BSS such as Dublinbikes, as they do not require you to travel to a designated docking station to retrieve a bike. Instead, these bikes can be parked at any official Dublin City Council bike rack and can be unlocked by scanning a QR code on a user's mobile phone. These schemes are much cheaper to operate (as expensive docking stations do not need to be constructed), and provide users with more flexibility in their routes. By incorporating this BSS into the application, this would provide users the flexibility to choose between the Dublinbikes or Bleeperbike bike-sharing schemes, depending on their preferences. This feature could be easily implemented and integrated with the current system, adding a new station for each cluster of Bleeperbike bike racks. Due to the low levels of usage of Bleeperbikes, the predictive element would be unnecessary, however if necessary, historical Bleeperbikes usage data could be used to track bike availability over time. However, the biggest challenge facing the implementation of this feature is the lack of an official Bleeperbike API. After reaching out to Bleeperbike, the CEO replied that they would happily let this project access their private API, but they did not detail exactly what data could be accessed.

Finally, one issue with the current implementation is that it is currently only available on Android. iPhones are used by many people in Ireland, with iOS having a 45.57% market share in Ireland in April 2020[7]. While React Native was considered infeasible for an app of this sort, it is worth studying the feasibility of developing a standalone native iOS app for this project. Since this project uses a microservices web API approach for its back-end, this means that an iOS app could be easily developed, using the same route-planning and prediction logic as the Android app. This could also be extended to an online web-app for the project, allowing users to create routes from their web browser. While these new apps would require some effort to implement, they would introduce the application to a wider audience.

[5] Dublin Bike Parking: https://dublinbikeparking.com
[6] Bleeperbike: https://bleeperbike.com/
[7] StatCounter: https://gs.statcounter.com/os-market-share/mobile/ireland

6.2 Final Conclusion

This project has succeeded in its goal of developing an algorithm for predicting availability of bikes and spaces at bike stations in the Dublinbikes bike sharing scheme. The effectiveness of the chosen k Nearest Neighbour approach has been proven, with a short-term weighted average f1-score of 0.77, a medium-term f1-score of 0.666 and a long-term of f1-score of 0.603. While this approach differs from that detailed in the literature by approaching from a classification perspective, it performs sufficiently for both short-term and long-term predictions. A microservices back-end was built around this algorithm, allowing end users to predict the availability of bikes and spaces at stations, as well as the ability to plan routes between locations using the Dublinbikes bike-sharing scheme. This back-end is designed to scale, with individual services and a flexible supporting infrastructure. An Android app was developed to consume this API, letting users generate the optimal route for their journey from their Android smartphone. This application was evaluated in a user test, concluding with a very high usability rating of 88.81, corresponding to an A+ grade in usability using J. Lewis, et al's curved grading scheme [28].

A VGI component was added to the application, allowing users to provide feedback on their route after reaching their destination. This VGI component allows the project to retrieve useful information that cannot be obtained from the Dublinbikes API, such as a user's waiting time at a station. This paper also explored potential future work for this project, including the possibility of integrating the app with multi-modal trips, consisting of multiple modes of public transport.

Overall, this project has successfully completed its goal of developing smart routing support for bike rental schemes. The process defined in this application is location-agnostic, and could easily be replicated to any city across the world with a bike-sharing scheme with an open, free API. Since this project is specifically designed for JCDecaux bike-sharing schemes, it can instantly be integrated with any other JCDecaux bike-sharing scheme, including cities such as Brussels, Lyon, Luxembourg, Brisbane, Stockholm and Paris. Due to the flexible design of this project, these cities can be supported by making very minor changes to the back-end code and changing a single line of code in the Android app. As well as this, the project can easily be extended to any other bike-sharing scheme from any other company with an accessible API.

With the minor changes detailed in Section 7.1, this application will be released to the public via the Google Play Store, where users from across the Dublin will be able to quickly and efficiently travel across their city using the efficient and environmentally friendly mode of transport that is cycling. If successful, this launch will be expanded internationally, introducing smart bike route planning to bike-sharing schemes to cyclists all across the world.

Bibliography

1. Sweet, M. Does traffic congestion slow the economy? *Journal of Planning Literature* **26,** 391–404 (2011).
2. Wjst, M. *et al.* Road traffic and adverse effects on respiratory health in children. *Bmj* **307,** 596–600 (1993).
3. Baumert, K. Navigating the numbers: Greenhouse gas data and international climate policy. *http://www. wri. org* (2005).
4. Lindsay, G., Macmillan, A. & Woodward, A. Moving urban trips from cars to bicycles: impact on health and emissions. *Australian and New Zealand journal of public health* **35,** 54–60 (2011).
5. Shaheen, S. A., Martin, E. W., Cohen, A. P., Chan, N. D. & Pogodzinski, M. Public Bikesharing in North America During a Period of Rapid Expansion: Understanding Business Models, Industry Trends & User Impacts, MTI Report 12-29 (2014).
6. Shaheen, S. A., Guzman, S. & Zhang, H. Bikesharing in Europe, the Americas, and Asia: past, present, and future. *Transportation Research Record* **2143,** 159–167 (2010).
7. Briton, E. World City Bike Implementation Strategies: A New Mobility Advisory Brief. *The New Mobility Agenda, Jun* (2008).
8. Shaheen, S. & Guzman, S. Worldwide bikesharing (2011).
9. JCDecaux. *Just Eat dublinbikes - latest figures! / Reports / Magazine / Dublin* Oct. 2019. http://www.dublinbikes.ie/Magazine/Reports/Just-Eat-dublinbikes-latest-figures.
10. Gast, N., Massonnet, G., Reijsbergen, D. & Tribastone, M. *Probabilistic forecasts of bikesharing systems for journey planning* in *Proceedings of the 24th ACM international on conference on information and knowledge management* (2015), 703–712.
11. Froehlich, J. E., Neumann, J. & Oliver, N. *Sensing and predicting the pulse of the city through shared bicycling* in *Twenty-First International Joint Conference on Artificial Intelligence* (2009).
12. Kaltenbrunner, A., Meza, R., Grivolla, J., Codina, J. & Banchs, R. Urban cycles and mobility patterns: Exploring and predicting trends in a bicycle-based public transport system. *Pervasive and Mobile Computing* **6,** 455–466 (2010).
13. Chen, B., Pinelli, F., Sinn, M., Botea, A. & Calabrese, F. *Uncertainty in urban mobility: Predicting waiting times for shared bicycles and parking lots* in *16th International IEEE Conference on Intelligent Transportation Systems (ITSC 2013)* (2013), 53–58.
14. Box, G. E., Jenkins, G. M., Reinsel, G. C. & Ljung, G. M. *Time series analysis: forecasting and control* (John Wiley & Sons, 2015).
15. Hastie, T. J. & Tibshirani, R. *Generalized additive models* English. ISBN: 0412343908;9780412343902; (Chapman and Hall, London, 1990).
16. Wood, S. N. *Generalized additive models: an introduction with R* (Chapman and Hall/CRC, 2017).
17. Broach, J., Dill, J. & Gliebe, J. Where do cyclists ride? A route choice model developed with revealed preference GPS data. *Transportation Research Part A: Policy and Practice* **46,** 1730–1740 (2012).
18. Hrncir, J., Song, Q., Zilecky, P., Nemet, M. & Jakob, M. *Bicycle Route Planning with Route Choice Preferences.* in *ECAI* (2014), 1149–1154.

19. *React Native - A framework for building native apps using React* https://facebook.github.io/react-native/showcase.
20. *Newest 'react-native' Questions* https://stackoverflow.com/questions/tagged/react-native.
21. *JCDecaux Open Data License* https://developer.jcdecaux.com/files/Open-Licence-en.pdf.
22. Thönes, J. Microservices. *IEEE software* **32,** 116–116 (2015).
23. Servén, D. & Brummitt, C. *pyGAM: Generalized Additive Models in Python* Mar. 2018. https://doi.org/10.5281/zenodo.1208723.
24. Schumaker, L. *Spline functions: basic theory* (Cambridge University Press, 2007).
25. Mohler, B. J., Thompson, W. B., Creem-Regehr, S. H., Pick, H. L. & Warren, W. H. Visual flow influences gait transition speed and preferred walking speed. *Experimental brain research* **181,** 221–228 (2007).
26. Gaudette, L. & Japkowicz, N. *Evaluation methods for ordinal classification* in *Canadian conference on artificial intelligence* (2009), 207–210.
27. Brooke, J. *et al.* SUS-A quick and dirty usability scale. *Usability evaluation in industry* **189,** 4–7 (1996).
28. Lewis, J. R. The system usability scale: past, present, and future. *International Journal of Human–Computer Interaction* **34,** 577–590 (2018).
29. Bangor, A., Kortum, P. & Miller, J. Determining what individual SUS scores mean: Adding an adjective rating scale. *Journal of usability studies* **4,** 114–123 (2009).
30. Botea, A., Nikolova, E. & Berlingerio, M. *Multi-modal journey planning in the presence of uncertainty* in *Twenty-third international conference on automated planning and scheduling* (2013).
31. *Where to park your bicycle* http://www.dublincity.ie/main-menu-services-roads-and-traffic-parking-dublin/where-park-your-bicycle.
32. Sun, Y. Sharing and riding: How the dockless bike sharing scheme in China shapes the city. *Urban Science* **2,** 68 (2018).

User Test Questionnaire

This questionnaire will collect your answers in relation to the usability of the dBikes Planner application. No other personal information is collected. This information is collected to evaluate the usability of the dBikes Planner app. This data will not be shared with third parties and will be used to measure the usability of the app. This data will be retained for 90 days and will be stored securely using Google Forms.

You have the right to access, view, and edit their own information in a timely manner. You have the right to be forgotten, which means being deleted from your survey results. You also have the right to be able to opt-out form your future messages.

If you have any questions, contact the Data Protection Officer Oisín Quinn at oisin.quinn@ucdconnect.ie.

	Strongly Disagree				**Strongly Agree**
1. I think that I would like to use this app frequently	1	2	3	4	5
2. I found the app unnecessarily complex	1	2	3	4	5
3. I thought the app was easy to use	1	2	3	4	5
4. I think that I would need the support of a technical person to be able to use this app	1	2	3	4	5
5. I found the various functions in this app were well integrated	1	2	3	4	5
6. I thought there was too much inconsistency in this app	1	2	3	4	5
7. I would imagine that most people would learn to use this app very quickly	1	2	3	4	5
8. I found the app very cumbersome to use	1	2	3	4	5
9. I felt very confident using the app	1	2	3	4	5
10. I needed to learn a lot of things before I could get going with this app	1	2	3	4	5

11. Please provide any other feedback on your experience using the app.

Additional User Test Feedback

The following feedback was collected from question 11 of the user test questionnaire: "Please provide any other feedback on your experience using the app.

1. Very clean UI, Liked the c02 saving note attached to journeys, liked the multiple journey options
2. Take my money
3. Very simple app to use
4. When tapping on the pink location bar down the bottom it would be nice to have it pop up just like when the bar is dragged up
5. Really nice app. I liked showing how many calories you've used and how much cO2 you've saved. One thing on that note, is I would like to get more information on the CO2 figure, is it compared to a single car or public transport etc, but that would just be a nice extra.
6. Really easy to use
7. really good my dude! it was very quick, I liked the CO2 and calories burnt numbers, choosing between quietest and fastest/shortest is great. Dark mode also looks great. My only complaint was the I wish you could make the graph full screen when you tapped or swiped up on it- be cool to see it more clearly. I felt like I could keep swiping up when the graph showed up and it felt weird when I couldn't. That's all tho, and it's super small. I would 100% use this if I had Dublin bikes. And tbh it looks like wizardry to me how well it works. Well Done!
8. Search bar could display "Destination" rather than "Search". Search feature to find bikes / drop off closest to an area rather than always finding a route. Doesn't always find the optimum route, searching for "Ranelagh" would expect "no bike route found" rather than walking further to collect a bike than it would to walk to the drop-off. Maybe colour the information of the bottom bar to match the number of bikes instead of the pink/magenta constant colour when clicking on a location
9. App - runs smooth and fast. no glitches or glaring weird stuff over the weekend's use. UX - Minimal/Material design, very cool. Accessibility - I am not qualified to talk about this, but I like the contrast-y colours lol. All elements on the screen are easily visible (in terms of size and colours). Features - Love the option for a quiet route and street view of a spot, can imagine it being popular with beginner-ish cyclists. Overall - Very cool, much needed app to commute.
10. Very intuitive and easy to use
11. Hey! I found the app a breeze to use and it provides really useful information. I'd definitely pay a once-off fee to use it as it has the standard of similar paid utility apps.
12. The calories burnt and CO2 saved counters are unique features. They are not part of the Dublin Bikes App (I think!)
13. Nice app! If I was a Dublin bikes user I'd say it would be very handy. Notifications asking for feedback might be better with a lower priority.
14. I felt the number of calories being burnt might not have been correct in a number of cases, maybe include the number the walk burns too. Otherwise great job :)

15. Overall a very well developed app which was clear and easy to use. The only issue was with some routes were if they were too far away the app would make the majority of the journey a walk i.e. from the start location to Sandyford. This may just be an issue with Dublin Bikes though.

16. One of the best betas I have ever tested. Well put together and integration of features (especially street view) was brilliant. I knew instantly what the icons on the map were and what they were for without really thinking they were just intuitively where I thought they would be. I am sorry though, reviews are meant to be constructive and criticize to make the app better but I am struggling to find flaws with this application

17. Extremely intuitive and easy to use!!

18. Loved the calories burned feature. Should sell on play store!