

Research Practicum Project Report

Predicting Dublin Bus Journey Times

Eoghan Cullen, Shane Dunne, Finnian Rogers, Ross Kingston

A thesis submitted in part fulfilment of the degree of

MSc. in Computer Science (Conversion)

Group Number: 5

COMP 47360



UCD School of Computer Science

University College Dublin

August 27, 2021

Project Specification

Bus companies produce schedules which contain generic travel times. For example, in the Dublin Bus Schedule, the estimated travel time from Dun Laoghaire to the Phoenix Park is 61 minutes (<http://dublinbus.ie/Your-Journey1/Timetables/All-Timetables/46a-1/>). Of course, there are many variables which determine how long the actual journey will take. Traffic conditions which are affected by the time of day, the day of the week, the month of the year and the weather play an important role in determining how long the journey will take. These factors along with the dynamic nature of the events on the road network make it difficult to efficiently plan trips on public transport modes which interact with other traffic.

This project involves analysing historic Dublin Bus data and weather data in order to create dynamic travel time estimates. Based on data analysis of historic Dublin Bus data, a system which when presented with any bus route, departure time, the day of the week, current weather condition, produces an accurate estimate of travel time for the complete route and sections of the route.

Users should be able to interact with the system via a web-based interface which is optimised for mobile devices. When presented with any bus route, an origin stop and a destination stop, a time, a day of the week, current weather, the system should produce and display via the interface an accurate estimate of travel time for the selected journey.

Abstract

The work presented in this report encompasses the design, development and testing of a web application for Dublin Bus oriented travel information. The application aims to present the user with more accurate predicted arrival times than what can be found in static timetables. Dublin bus historic data from 2018 facilitated the training of K-Nearest Neighbour models for full routes, in each direction of travel. These models are shown to dynamically predict journey times with a mean absolute error of 7.5 minutes.

The application aims to facilitate a customised user experience by providing user accounts which personalise the application according to user preference. These user accounts are not required to use the application, but enhance the functionality of many of the application's innovative features. The project is presented to the user in the form of a mobile-optimised web application and was developed over four three-week sprints using Scrum Methodology. The application is accessible at thedeparted.games and the GitHub repository for the project is available at github.com/Eoghan-dev/comp47360_wotb.

Acknowledgments

We would like to thank the lectures first and foremost for their guidance and insights into our project. The mentor meetings provided us with stability and focus, which ensured we stayed on track with the project. We are grateful for the services UCD provided us during the project, which facilitated development.

We would also like to thank Laura Dunne for all help provided in the role TA. She provided insight into the more intricate parts of the project, built upon her own past experiences. This allowed us to better build our final application to a standard we were happy with.

Finally, we would like to thank all friends and family who completed our various surveys and tested the application. They found errors and provided feedback to improve on the usability of the application.

Contents

1	Introduction	1
1.1	Objective	1
1.2	Existing approaches and their limitations	1
1.3	Overview of the project	2
1.4	Report Structure	2
2	Description of Final Product	3
2.1	Application Overview	3
2.2	Project Specification	4
2.3	Innovations	5
3	Development Approach	8
3.1	Development Strategy	8
3.2	Sprints	8
3.3	Communication Method	9
3.4	Organizational Tools	9
4	Technical Approach	10
4.1	Overview of architecture	10
4.2	Backend	10
4.3	API's	11
4.4	DATA	12
4.5	Predictive Modelling	13
4.6	Frontend	13
4.7	Deployment	14
5	Testing and Evaluation	15
5.1	Application testing	15
5.2	Predictive Model - KNN	17
6	Major Contributions	19

6.1	Description of Role	19
6.2	Development Responsibilities	19
7	Background Research	23
8	Critical Evaluation and Future Work	24
8.1	Critical Evaluation	24
8.2	Future Work	25

Chapter 1: Introduction

1.1 Objective

Transportation is one of the leading causes of greenhouse gas emissions globally, with road transportation contributing 21 percent of the EU's total emissions of carbon dioxide, the main greenhouse gas [1]. Added to the environmental concerns surrounding road transportation, traffic congestion is a major issue facing major cities around the world. One way of tackling these issues is to encourage the use of public transport to reduce the number of vehicles on city roads.

Our bus networks can be a hugely valuable resource in this context. Bus services are the most commonly used form of public transport in the European Union and are a highly cost-efficient and flexible form of public transport. They form a vital part of a city's transport infrastructure for the following reasons:

- They improve social inclusion by providing access to education, employment, and healthcare
- They are a vital link in the overall transport infrastructure of a city
- They are important contributors to tourism
- One bus has the capability to replace 30 cars on the road which can help ease congestion[2]
- Buses are the most sustainable form of motorized transport with the lowest carbon footprint per passenger[2]
- They are the safest mode of transport accounting for only 2 percent of fatalities on EU roads[2]

It is therefore vitally important that citizens are encouraged to use bus services such as Dublin Bus. One way of achieving this is by providing an app that will allow bus users to better plan their journeys to improve their user experience. The objective of this project is to build upon existing approaches to develop a richer application that will enhance the Dublin Bus experience for its users.

1.2 Existing approaches and their limitations

In order to achieve the stated aim of the project, an analysis of existing solutions was carried out. The key findings of the analysis are outlined below and focus on the Dublin Bus web application itself and the approach taken by the Transport for Ireland application.

The following key limitations were identified:

The use of static generic travel times provided by many bus companies. The key example in this case being the static timetable displayed on the Dublin Bus website that provides estimated travel

time to its customers. This approach is clearly limited by the many dynamic factors present in a real-world scenario that affect bus travel times. These factors include traffic conditions affected by the time of day, day of the week and the month of the year. Weather conditions also have a significant impact on journey times.

The second major issue found with existing approaches centred on the ease of use or personalisation of the solution. This issue was evident on the Transport for Ireland application, which provides a much more dynamic solution than the static example cited previously. However, after analysing this, it was concluded that the application was limited by its lack of personalisation and therefore lacked the ease of use that is believed a busy modern user would benefit from.

1.3 Overview of the project

To address these limitations, real-time bus and weather data was analysed to create dynamic travel time estimates, which are made available to users through a web-based interface optimized for mobile devices. To further augment the usability of the application, the functionality for users to create their own personalized accounts will be provided. These accounts allow users to save their favourite routes and stops, making this information easily accessible for commuters on the go. The option for users to indicate their user payment status will be provided so that they can receive personalized fare calculation estimates, and the option to receive bus cancellation information for favourite routes will also be provided. A user features survey and a final usability survey will be conducted to gather feedback as the application is developed.

The project was carried out over a 12-week period in a team consisting of 4 people, with the time-period broken into four sprints of equal length. Four distinct roles were created within the team, a code lead, a coordination lead, a maintenance lead, and a customer lead. The success of the project relied on the quality of the research, the technical application, and the strong team dynamic within the group.

1.4 Report Structure

The purpose of this document is to outline in detail the solution to the problem outlined above. The report will include an in-depth description of the final product including the problem addressed, the broad functionality of the product and the innovations that elevate the product above the problem specification. The document will outline the development approach taken by the project team, how the process worked and what strategies were employed. The technical approach taken to outline the architecture of the system, the technical stack utilised and the justifications behind the design and technology decision made throughout the process will be described in detail. Finally, for the group portion of the report, the description of the testing and evaluation strategy for accuracy, efficiency, and usability of the final product.

Chapter 2: Description of Final Product

2.1 Application Overview

2.1.1 Directions Feature

The application has a range of features, which both address the minimum specification of the project and provide several innovative functionalities which go beyond the minimum requirements. The core feature of the application is a directions feature, which is what the user is greeted with upon loading the web page with the option to move between the Directions, Routes and Stops tabs to use each feature. This feature allows the user to enter an origin and destination location manually, or use their current geolocation as their origin location. Upon entering origin and destination locations along with a date and time, directions for each step of the journey are displayed to the user along with the estimated time and cost of each step. Finally, the user is also shown a predicted total journey time and cost prediction.

The path of the route is also shown on the map, which is handled using the Google Maps API (see figure 2.1a). Each step of the journey can either be a walking or bus step, the predicted time taken for the bus steps are generated using the predictive models trained on historic data, whereas the walking step time predictions are taken using the predictions returned by Google Maps. Journey directions and predictions can be generated for up to five days in advance.

By default, the fare prediction displayed to the user is for an adult fare with no leap card. If the user is logged in, they can change these settings to get more accurate fare predictions. Fare predictions are generated by the number of buses taken, and the distance travelled on each bus throughout the trip. This information paired with whether the user is an adult or child and has a leap card or not allows us to make a fare prediction.

The stops feature allows the user to select a route from an autocomplete text search or by choosing a favourite route saved to your account if logged in. Upon selecting a stop, it should be displayed on the map as a marker and an info window should be displayed to the user. This info window should contain the stop name and number, all routes that serve that stop and the next several buses that will arrive to the stop and how far away they are (see figure 2.1b). The time remaining for the next several buses to arrive is calculated using a combination of predictive models and real-time data from the General Transport Feed Specification (GTFS) API provided by Transport for Ireland (TFI).

The routes feature has a similar interface to the stops feature, where the user can select a route from an autocomplete text search or by choosing a favourite route saved to your account if logged in. Upon selecting a route, the user can see the path the route takes on the map along with all the stops on the route (see figure 2.1b). The user can select any stop on the map to activate the stops feature and see the information displayed in figure 2.1c.

2.1.2 User Accounts Feature

As mentioned previously, the application has user accounts which are not mandatory but allows the user to avail of additional features upon registration. If a user registers for an account, they then gain access to a dedicated My Account page, which consists of two tabs, favourites, and user settings. From the favourites tab, the user can add or remove favourite stops and routes for easier access in each feature and click a link to see the timetable for any favourite route (see figure 2.1d). From the user settings tab, the user can edit their fare status as adult or child and whether they have a leap card or not, which will be taken into account during fare calculation in the Directions feature. The user can also change their password from this page.

2.2 Project Specification

The minimum specification for this project was to create a mobile-optimised web application which, when presented with a bus route, origin and destination bus stops, a departure time, a day of the week and the current weather, should produce and display an accurate time prediction for the journey. The application achieves this specification, but in a more user-friendly way than manually entering all of these parameters when making a request to the system. This minimum specification can be interacted with through the Directions feature, which takes origin and destination locations, a date, and a time as inputs. These locations can be selected using a text search input, which automatically suggests places using Google Places API as you type.

Alternatively, the origin location be taken as the user's current geolocation. The date and time inputs are automatically set to the current date/time by default, but the user can request predictions for up to five days in advance, as that is as far in advance as weather predictions can be generated. Weather forecasts are generated using the Open Weather API, on an hourly basis for up to five days in advance, and stored in a database.

When the user enters two locations, a date and a time, a request is given to the Google Maps Directions API, which returns directions and information for each step of the journey (either walking or bus). For all bus steps, upon receiving this information from the API the application fetches weather data from the database for the appropriate time, matches the origin and destination locations to their nearest stops and gets the route number for the suggested route and feeds this information to the model, giving back an estimated time for that step.

Depending on the locations entered by the user, more than one bus may be necessary for the journey, in which case this step is repeated as many times as necessary. For walking steps, the journey time is taken automatically from the prediction generated by the API response by Google. Google's predictions for bus routes were used in lieu of the custom classifier predictions (described in 4), when it was unable to make predictions. An example of such a failure is when the bus routes were missing from the historic data used for training the models.

After this information for each step is gathered, directions, estimated time taken and a predicted fare for each step of the journey are displayed to the user, along with a total time and fare prediction for the entire journey (see figure 2.1a).

2.3 Innovations

2.3.1 Directions Feature

As mentioned in the product overview, there are a number of innovations that go beyond the minimum project specification in the application. However, before looking at these innovative features, it is worth noting that the Directions feature which encompasses the minimum specification also has a number of additional features built into it.

Firstly, it allows for a user to enter any two locations (or geolocation for origin location) and get predictions and directions based on this, rather than having to explicitly enter a start and end stop and route number. The minimum specification does not require directions either, which is an innovation built into this feature with the Google Maps API.

The fare predictor is also an innovative feature which is always displayed along with the time prediction. By default, this generates a prediction for an adult with no leap card, but the fare status is changeable from adult to child along with whether or not you own a leap card. The predictor is an estimator and is not 100% accurate due to the way Dublin Bus have set up their pricing tiers. They have separate prices for adults and children, with different prices within each depending on if they have a leap card or not. Children also have a different set price if they are travelling during hours they have defined as “school hours”.

All of these factors were relatively straightforward to account for in this feature, but what complicated things was that the price per journey changes depending on the amount of “stages” the bus has passed through. These stages appear to have no clear definition, not seeming to relate to a number of stops, but instead to be a predetermined distance in metres. As a definition for what a “stage” is according to Dublin Bus could not be found, it was decided to take a stage to be a bus stop, as this appeared to be relatively accurate from the research conducted. In short, for any given bus journey, the number of stages was directly mapped to the number of bus stops passed. Plans are in place to continue to do research on what consists of a “stage” but for the meantime it was felt that this is a reasonable compromise which could be easily updated in future work.

2.3.2 Routes Feature

The Routes feature is another one of the main innovations, which can be found on the main page of the application. The Directions, Routes and Stops features can all be accessed using tabs at the bottom of the screen from the home page. As mentioned previously, routes can be selected either through an autocomplete search field or by selecting a favoured route if logged in. Upon selection, the routes feature displays the path any bus route takes on the map, along with all the stops on the route as markers (see figure 2.1c). Any of the markers can be selected to see the same information for that stop that can be found using the stops feature. The line on the map for the route path is generated using the Google Maps Directions API.

This is accomplished by obtaining the start and end stops for the selected route and checking the timetable for that route to see when it should next leave the start stop of the route from the static JSON files. Once this information is obtained, a request can be sent to the Google Maps Directions API from between the co-ordinates of the start and end stop of the route and with a desired departure time of whenever the bus is next scheduled to leave. This worked in all tests except for a scenario where a route is requested at a time of day when there are no more buses running on that route, as the application is then unable to set the time of the next departing bus from the timetable as the desired departure time as there are no more departing buses for that

day. In this case, all the markers for the route are still shown, but the line generated from the API is not displayed on the map. It is planned to work on this in future work, but for the time being it seems to work well in most scenarios, except for a few select routes that only run early in the morning or stop running particularly early.

2.3.3 Stops Feature

The Stops feature can be interacted with in the same way as the Routes feature, but instead of displaying the route and all its stops on the map, it displays that stop and that stop only on the map along with an info window for that stop. The info window displays the stop name and number along with the routes that serve that stop, but the most innovative component of the stops feature is that it also displays the next several buses that will arrive to that stop and how far away they currently are. A view collects a list of all upcoming buses in the next hour from the static GTFS information and feeds this list to the predictive algorithm described in 4. The generated results are then ordered, and the view returns the next four buses past the current time. This feature can be considered an in-app equivalent to the electronic signs at many bus stops, which display the next arriving buses and how far away they are.

2.3.4 User Accounts Feature

Users have the option to create an account to avail of additional features in the application. The main features they can avail of are the ability to save favourite routes and stops for easier access from their respective features, and the ability to update fare and leap card status for more accurate fare predictions. All accounts are handled using an extended version of the built-in User model in Django and stored securely in the database. This allows us to use many of Django's built-in tools for authentication and account management, which facilitates secure and easy administration for this component of the application with full password hashing.

When registering for an account, users are asked for their first and last names, their email address and their password. Fare status (adult or child), leap card (yes or no) and favourite routes and favourite stops are additional fields which can be modified from the My Account page after registering. Fare status is set to adult and leap card set to no by default, while the favourite routes and stops are empty by default. Favourite routes and stops are stored in the database as comma separated strings, which can then be parsed as an array when handling this data in either Python or JavaScript. The application also contains a fully functioning forgot your password feature, where you can have a password-reset link sent to the email you signed up with. Alternatively, if you know your password and just want to change it, you can do so from within the My Account page in the application.

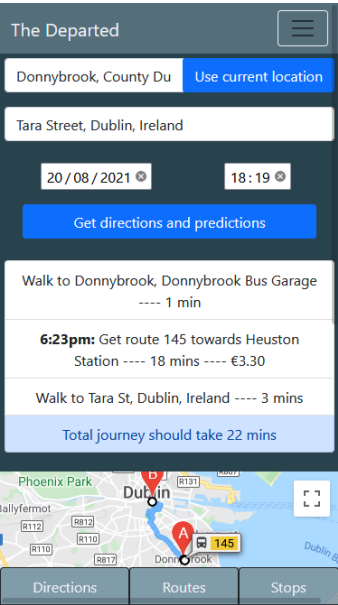
2.3.5 Cancelled Routes Feature

The application has a cancelled routes feature where currently cancelled bus routes can be found. This feature can be accessed from the routes tab on the home page of the application, whereupon clicking a button, an overlay screen with a table containing all cancelled routes is displayed to the user. This is accomplished by the server querying the GTFS Real-time API every five minutes and storing the results to the database. When the homepage of the application is opened, the database is queried to get all routes that are currently cancelled, and this is then parsed on the front-end and formatted into a table, so the user can view it once they click the button. The table displays the route name and head sign, along with the previously planned departure time

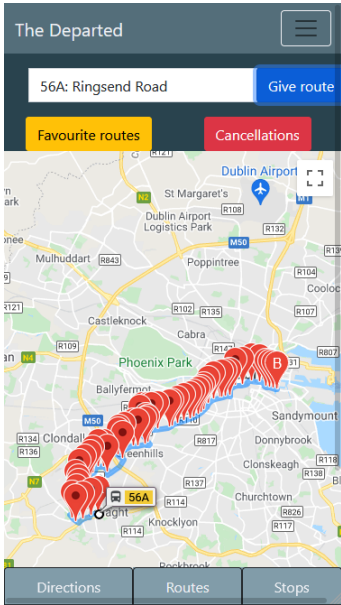
from the first stop on the route.

2.3.6 Additional Innovations

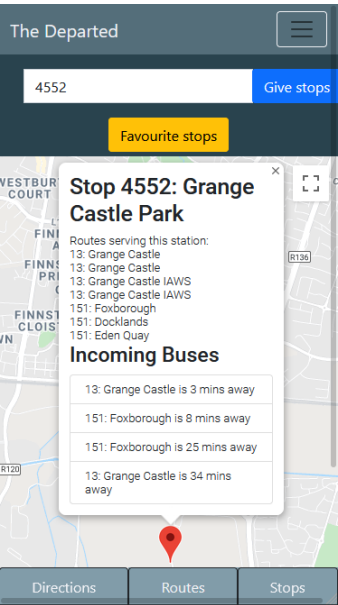
In addition to the aforementioned innovations, there are several smaller innovations in the application. The first of which is a timetable feature where you can see a full timetable for any particular route listing the departure times for each stop using static Dublin Bus timetables which were generated using the static GTFS data. There is also a dedicated Twitter page where live updates can be viewed from the Dublin Bus Twitter account.



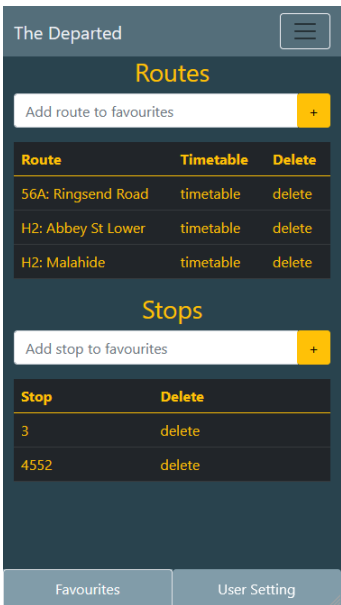
(a) Directions Feature



(b) Routes Feature



(c) Stops Feature



(d) Account favourites

Figure 2.1: Application pages

Chapter 3: **Development Approach**

This section of the document will cover in detail the work processes the group employed to deliver a final product. An outline of the development strategy used to manage the process and the operational tools employed to aid in the implementation of this strategy.

3.1 Development Strategy

The development strategy the group agreed on was the agile scrum methodology. All four members of the team had used this methodology before and were satisfied that it worked well to manage a software project to a successful completion. As mentioned in the introduction, the process was broken down into four sprints of equal length (3 weeks). Each sprint was top and tailed by pre- and post-sprint meetings. Each team member assumed the role of scrum master for one sprint, ensuring that the scrum framework was adhered to throughout.

3.2 Sprints

3.2.1 Sprint 1

During the first sprint, the group concerned itself with establishing a strong foundation for the rest of the project. This meant establishing good project management documentation which was done using Confluence, Jira, and Google Drive. An architecture for the project was agreed, and the technologies outlined in chapter 4 were chosen. Each member of the group got set up to use the relevant technologies. Communication and conflict resolution protocols were agreed. All relevant data was collected for the predictive model to be created.

3.2.2 Sprint 2

In the second sprint the project concentrated on establishing a solid front and backend communication, this was done successfully. A basic model was created with a pickle file to be used when building out the remainder of the application. Views were set up on Django that served predictions to the front-end. At this stage of the project, the group generated ideas for features and conducted a user features survey to ascertain feedback on the ideas generated.

3.2.3 Sprint 3

Sprint 3 proved to be a very productive period for the team. The frontend UI was brought close to completion with only some minor styling issue to complete. Fully functioning predictions were served to users at the frontend. During this sprint, the team carried out a number of long form meetings to intensify progress. All four members of the team participated in these meetings, and they proved very successful in progressing the project.

3.2.4 Sprint 4

Work in sprint 4 focused on testing the application's functionality and fixing bugs resulting from testing. A usability survey was carried out to test the usability of the application, and final polishing of the user interface was completed.

3.3 Communication Method

Robust communication protocols were established very early in the project. Daily stand-up meetings took place at 10am every weekday. A discord server was set up to conduct the group's regular meetings, the server was also used for communication outside formal meetings. Stand-up meetings were attended by all team members each day. These meetings ensured the team successfully tracked the progress of the project daily, while also fostering a very strong team ethic within the group. Code reviews were carried out every Friday to ensure all team members were kept a breast of the workings of the various components of the project. During sprint 1 a larger meeting was scheduled for Wednesday evenings however as the weeks progressed, these meetings were found to be redundant, and they were replaced with meetings called for specific issues as they arose. Conflict resolution was handled by discussing the pros and cons of an issue and coming to a consensus based decision.

3.4 Organizational Tools

Organizational tools were chosen to aid with communication as well as the management of the project. The tools used were Google Drive, GitHub, Confluence and Jira. Google Drive was used to store documents, so all team members could access and them as necessary. Confluence was used to record meetings including daily stand-ups, code reviews and any other meetings. Jira was used in tandem with confluence to plan and track sprints and to distribute task across the team. GitHub was employed to handle version control using a development branch, branches for each feature in progress and a final main branch.

Chapter 4: Technical Approach

4.1 Overview of architecture

For the project, the team used a range of technologies to aid in both the design and the functionality of our application. A brief overview can be seen in Figure 4.1, which shows the interactivity of these various technologies. At the centre of the application is Django, this joins and operates all the various technologies in unison. The web application the team have designed is aimed for mobile users, however, with the grid box methodology in bootstrap the app should also be clean and accessible for a user through a tablet or desktop.

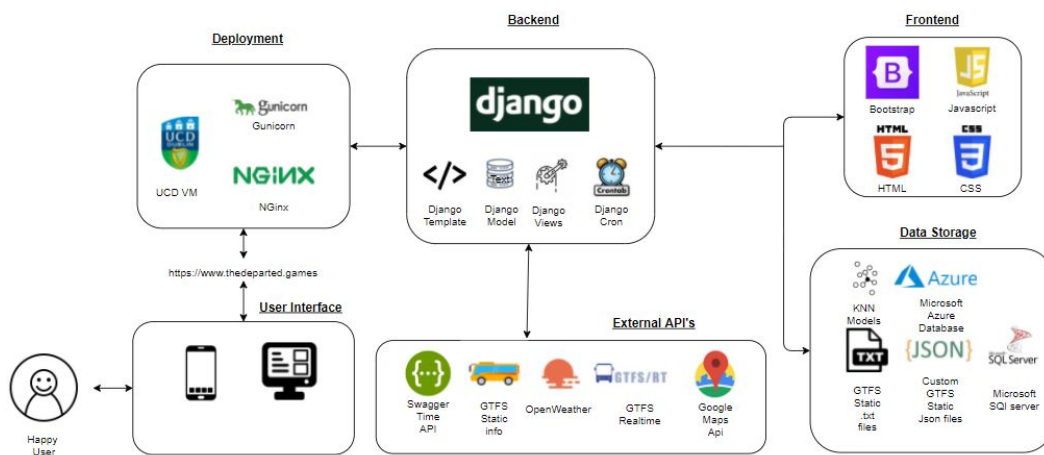


Figure 4.1: Application Stack

4.2 Backend

Django was used as the backend framework for this project. Django follows a models/templates/views architectural format for delivering a working application to the user. Django comes with a deep source of documentation pages available with it thanks to the application first being created in 2005 and being widespread in industry (Instagram, YouTube, NASA). It allowed for those working on the frontend to quickly come to speed with the base features available on the framework, as well as quickly troubleshoot many common problems that occurred over the course of the project.

The “model-view-template” is a software design template that Django uses. Models handle the database and provides customisation with how someone accesses the data and subsequently uses it. Template section deals with how information is presented to the user in the form of the UI. Views are used to complete tasks on the backend and collects data from the models; along with rendering templates with passed information. This provides a basic structure needed to create a

web-based application.

Within the models, data was scraped from various online API's. The data was then either; migrated to the application's relational database on Microsoft Azure or stored as JSON files locally. Data stored on the models could then be readily accessed through the Django ORM that can interact with a multitude of different relational databases. The ability to interact with a multitude of different relational databases allows the application to be highly adjustable and to quickly adapt to a new database. This was important within our project due to the use of Microsoft SQL Server, which is not a natively supported database with Django. This ensured that if any issues arose later, the database that Django was interacting with could be easily changed to one with a natively supported dialect of SQL.

Views provided a way to interact with this data saved within the models or locally and to access the required information. This could then be used to feed the data to the user. It also allowed us to account for users actions and to return the desired information to the customer from our templates. Views provided us with the ability to access our stored data whether this was from pickle files, models or JSON files and then customise the data to allow it to be correct rendered with the template.

4.3 API's

The main source of the Data we used for current information surrounding Dublin bus came from the General transit feed Specification or GTFS. GTFS allowed the specification of which transit mode was desired to call information for, for this project we contained collected data to on buses from Dublin bus. The information was received as text files on download. These files were converted to JSON files to facilitate simple integration with the frontend, along with an uncomplicated upload to the database on necessity. The information contained within provided an accurate source to find all bus stops and the routes served by each stop. It provided a way to find the official departure and arrival times for not only the start and end points but also for each stop along the route and save these in custom JSON files for quick access. This was essential for interaction with the predictive model, which required all this information to form a prediction of a route at any time. These JSON files can be scraped whenever major changes occur to the transportation route, but does not have to occur often as the current information is valid until 16/10/2021 per the calendar.txt file which gives the dates for bus trips and how long they are valid until.

For real time information relating to Dublin Bus, data was scraped from the GTFS-R API. This API did not contain an exhaustive list of buses. Rather, it contained information on cancelled buses or known delays on a route, instead of providing information on every bus currently running at a given time. This information contained less information that could be used reliably, but provided the application with a way to supplement the given information by informing the user if a bus was cancelled along a route.

OpenWeather API was used for scraping current, Forecast, and historical weather. It only gained information for one location across Dublin. This was due to the historical data being over a year-old, meaning that if more than one co-ordinates was wanted for accurate weather at each stop, then a fee of 10 dollars would need to be paid per location accessed. This meant that the information paired to work with the historical data would relate only to the general weather data from a single location in Dublin. As this is what the predictive model was based on, it was decided to use the same single location for the current and forecasted weather. Data from this location would then be fed into the predictive model to return an estimate.

The Google Maps API was used for the display of markers and routes to the user on the home page. This provided us with important information such as transit routes, along with information relating to the next bus along a route from a particular stop. This was used heavily when a user called the directions search feature, as this could direct a user which bus stop to walk to, and from there it would then feed the information to a view that would access our data to get the estimated journey time of this bus if possible. It also provided us with a fail-safe that we can simply default to google maps if for some unforeseen reason the view fails to return a value.

An API called swagger time API was also implemented late into the project. This was to deal with the one hour thirty-minute time difference in the UCD server compared to real-time. It was chosen over a simple time delta as it ensured that no matter the server the application is running on, the app will always be correct for parsing information to models for Dublin.

4.4 DATA

The Database used for this project was provided by Microsoft Azure, using Microsoft SQL Server as the relational database language. This was chosen as it was provided cheaply with the student version of Microsoft Azure and provided the most functionality at an affordable price to ensure that the credits given would not go over the limit throughout the project. This came with the issue of not being natively supported on Django as a language supported with ORM. This was overcome by using an earlier version of Django (3.0) that came with the ability to install mssql-django that allowed interaction with this database. This at an early stage still needed to be tested to ensure no major issues arose throughout the project but thanks to ORM provided with Django along with the model system it allowed us to use this database with the knowledge that if issues arose, we could adjust our chosen database to a natively supported one later in the project.

JSON files were also used for a lot of our static information, this facilitated simple management of the data needed for creation of bus stops across Dublin and timetables for given bus stops. It also provided a simple way of adding data to the database via Django fixtures(automated Django feature to allow certain information to be instantly transferred to the database) or custom models. From here we could easily see issues with the files scraped from the API from the GTFS API such as times over a 24-hour period and could easily provide fixes and catches, so these could be amended within the JSON files. These provided a convenient way to access this information, which could be read on load of a page, and which was needed for fluid interaction on certain pages, as can be seen with stop markers on the homepage or for various stop times in the timetable page or the loading of the data lists for routes and stops.

In total, the application contained 6 JSON files pertaining to general information about bus information, along with a JSON file representing bus times for each individual route. Meanwhile, the application had 2 consistently updating tables on the database: real time bus info, and weather forecast for the next 5 days in 3 hour intervals of accessible information. It also had a table that stayed static, which was the historical weather data that was paired with the historical bus information in 2018 to generate predictive models.

4.5 Predictive Modelling

36% of American smartphone users abandon a mobile transaction if it is too slow [3]. This concept of long wait times not being favourable is not shocking and was echoed during the user feature survey. It was very important to the group that the models make predictions fast but also maintain an acceptable degree of accuracy. Finally, due to the storage plan, deployed space on the hosting machine was limited and so the models were also required to be compact. In order to satisfy these requirements, the K-Nearest Neighbour (KNN) was used to classify the historical data. KNN was the model used in the final implementation, although many others were evaluated during the data preparation phase. The KNN method is a simple approach and so the total model memory usage was minimal, once compressed, amassing to 32 MB. The principle of KNN finds the closest k features to the target feature and classifies new cases based on the closest target features from the training data [4].

The structure of the preparation phase was centred around the model approach used. Again with the memory and efficiency constraints in mind, a full route model approach was chosen. With 130 bus routes, this would amount to 260 models due to the need to model in each direction as the traffic flow varies on either side of the road and the routes may differ in opposite directions. This number of models is far less than the 6000 needed for stop-2-stop. Due to the historic data being from 2018, without spending money there was no option to get a variety of locations around Dublin, therefore the weather is only taken from one location in central Dublin. This may not be a factor for most routes, however some busses come from the greater Dublin area where the weather may be different. The historic data was stored in the high performance server provided by UCD and was converted to a SQLite format to both reduce the load on the server and allow the data to be queried using simple SQL commands.

To build a full route model, the appropriate data had to be queried from the historical bus data. For example, for the route '56A', this route is queried using its ID and stored in a pandas' data frame. The weather for each trip is then added to the data frame based on the departure time and day of service. The day of service is then split to two features, day (represented numerically from 0-6) and month (represented numerically from 1-12). The final data frame used in modelling contains these features as well as the temperature, weatherId (which relates to the weather experienced at that time e.g. rain), and the actual/planned departure and arrival times.

After some cross validation of the k number in the classifier, it was set to 5. This provided an acceptable mean squared error when compared to $k=10$ (92 minutes² vs 140 minutes²). One thing to note that while this mean squared error was improved at a low k number (71 minutes²), this is due to oversampling, which could severely negatively affect future predictions [5]. Through a combination of python libraries, pickle and gzip, all classifiers were serialized and compressed to reduce their size as much as possible. Finally, a method made in views python file could be made to seamlessly integrate the predictions to the site.

4.6 Frontend

Ultimately, the user interacts with the application through the HTML pages outlined in Django templates. The user interface is determined by these various pages, with a large amount of functionality occurring through various JavaScript functions. These took user inputs and interacted with various views in Django to return the user requested information to be displayed on various pages through parsing through the returned JSON objects sent via views. This meant that the information a user needs to download locally was kept to a minimum to ensure that the user can

access the application with minimum internet access as might occur with the app as they will likely access it on the go where their connection may not be as strong as at home.

Most of the CSS was handled through bootstrap, the use of bootstrap allowed us to optimise the page to ensure that it was consistent design for not only mobile users as per the project specification but also for desktop users. This was achieved with bootstraps grid system that allows us to specify the number of columns we wish a piece of HTML code to occupy on a page via rows and columns for specific device size using predefined grid classes. This allowed us as a group to use the space most optimally for each device. Initially we used bootstrap studio to build the pages which provided us with a great base to build functions on the page, but we quickly found this cumbersome and restricting as we tried to settle on the final design of the page and started taking a much more hands-on approach with applying the bootstrap.

The web application contained 6 HTML pages with 5 of these relating to unique pages:

- Index html – Routes, directions, stops, map, Favourites, prediction displays and journey time
- MyAccount – favourite routes, fare status, login for additional features, register
- Timetable list - Full list of timetables available, with search feature to find desired route
- Timetables – Times for each departure time at every bus stop along route (based on static data)
- Twitter – Feed of Dublin bus twitter page with relevant info contained from there
- AboutUs – Page about us and the roles we took on the team + link to GitHub

4.7 Deployment

When the application began to take shape, it had to be deployed. Gunicorn and Nginx are used to serve and distribute the Django application. Gunicorn is a Web Server Gateway Interface (WSGI), which provides the instructions for how a web server can communicate with the application. Gunicorn strongly suggests using a proxy server to handle load balancing and serve static files and allow SSL handling, these last two are crucial to the functionality of the application. Django applications have a WSGI file which Gunicorn can bind directly to. Binding is the process in which a local address is associated with a socket. This project uses Unix domain socket, which is an interprocess communication mechanism that allows the bidirectional data exchange required to serve the static files [6]. Nginx is the medium through which the users of the application will communicate and is at the edge of the Internal network of the Linux server provided by UCD. In order to use the geolocation feature, the application must be using the https protocol and so a Secure Socket Layer (SSL) certificate is required. The domain name and corresponding SSL certificate was received from names.com. The Nginx configuration file then had to be designed as to redirect all traffic to the domain name, including converting http requests to the https protocol. After this configuration was completed, the application was then accessible from the domain name, thedeparted.games. Deploying the app allowed for user tests and load testing.

Chapter 5: Testing and Evaluation

From the beginning of development, testing and evaluation were recognised as integral parts of deploying a successful application. Firstly, a distinction should be made between testing and debugging. Debugging homes in on a specific aspect of code that is not functioning as intended and attempting to find the root cause and possibly employ preventative measures for future cases. In the case of this project, the root cause of many bugs resided in the python and JavaScript scripts, as they controlled the greatest number of moving parts of the application. Debugging relies on defects in the code which have been found, whereas testing is the process in which a system is subject to operations in order to find these bugs [7].

There are a multitude of different testing options available in order to derive the most meaningful aspects of the software component tested. Different stages of the application require more intense testing, for example if the application is to be realised to a large user base, appropriate load testing must take place. In this project, it was important that the tests were not intrusive, due to a short life cycle of application development. A mixture of manual and automatic testing was used during evaluation, each providing useful insights into the script function. This testing mainly fell under the categories of functional testing and non-functional testing. The former tests that the code is producing expected outputs, and the later testing the behaviour of the application under strain [7].


As mentioned in chapter 4, code accessibility was essential to code together and crucially opened up the team to collaborative testing. The peer review meetings held by the team highlighted issues that needed to be further tested and provided the team with a recommended approach to the topic. The testing and evaluation can be broadly broken down into two sections, Web application and predictive model testing. As described previously the web framework used in development was Django, this framework facilitated a variety of testing. The performance of the web application could then be testing using developer tools and load testers. Finally, the predictive model testing and evaluation was conducted using the sklearn metric library.

5.1 Application testing

Django received http POST requests from the front end of the application via views. These views would receive information from the URL based on the action in the application. Django has libraries which simply test this function. By importing the aptly named 'SimpleTestCase' to a test python file, a series of test cases can be made for the application. A Class was made which inherits the SimpleTestCase and based on the http response received from a known query the Django requests which govern the flow of the framework. The SimpleTestCase generates a coverage report based on the amount of responses that were expected. The current coverage of the application is 44%. This number is very low and below our standards however, the percentage is being drastically effected by the scraper files (2) which get current info and reformat into customised JSON files to suit the application's needs. Without these files the coverage increases to 75%, this is a much more acceptable figure. This test case should have been implemented sooner in development, to make retrospective testing more manageable. As developers wrote new functions appropriate test cases should have been made alongside them, this would have greatly increased efficiency in the later stages of development.

Like a newly released car, most people just want to know how fast it is. Like this newly released car, not only was it important for the application to be fast, but to be reliable too. A Google developer tool called Lighthouse was used to conduct a performance analysis of the site in production. These performance tests ensure that the application meets the teams' standards for efficiency. The categories of this test can be described as service-orientated, and efficiency-orientated. Where service would judge the availability of features and their response time and efficiency would judge the capability of an application to make use of its canvas [8].

8/15/2021

 <https://www.thedeparted.games/>

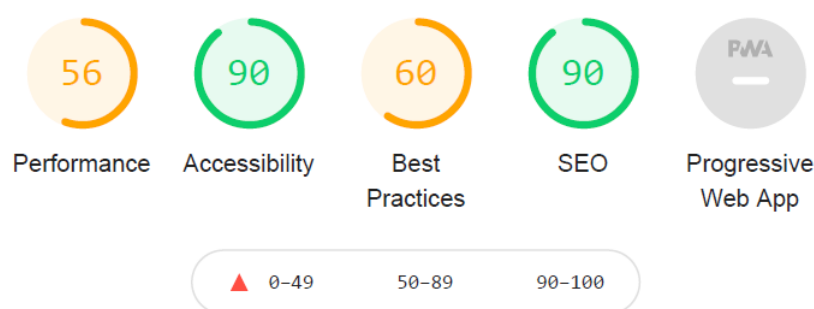


Figure 5.1: Lighthouse performance test results

As seen above in figure 5.1, the application performs very well in accessibility and search engine optimisation (SEO). The accessibility category can broadly be improved by going back through the code and adding labels to everything for screen reading, and improving the colour contrast to make elements pop. These improvements were noted and would have to be tackled before a public release. Again, the SEO can be improved by like additions as in a meta description to the HTML. Performance and best practices suffer in score. There is a large amount of style sheets being transferred onto the site, which blocks render times for more important features, as most of the CSS may not be used. The feature that is the most intensive for the application is the map. The large element size increases memory usage and hence load times, since this is a core feature this load time was deemed necessary by the team until further work can be done to improve load times. Lighthouse quotes possible savings of 1 second when those two issues can be addressed. However, the third-party blocking time from loading the map needs more work into fixing without sacrificing size. Finally, the best practices low score is less troublesome, with the application losing marks on geolocation request upon load and JavaScript libraries that are deemed untrustworthy. In summary, the score across the board will improve once the unnecessary files are removed from the site, which may take some time to weed out. Lighthouse was mainly used towards the end of the project, but if this tool was used earlier, there would be less work in cleaning the code at the end of production.

Throughout the peer reviews and live development of the JavaScript elements to the project, manual testing was employed to ensure the correct information was being passed through the site. This testing was most commonly a product of debugging in development. It is difficult to put a figure on performance of this type of testing due to the predominant factor being the person running the tests [9]. Using console logs through JavaScript was effective for small bugs, but were also used for larger problems and came at a high cost, specifically when attempting to join the front and back-end work for the first time.

Once deployed, the application was opened up to another type of testing. Load testing. This testing was mainly to see how the application and software used to deploy it could handle the strain of consistent usage. Using a stress testing application called "Webload" an investigation into the strain caused by multiple calls to the application was determined. Webload requires a script to be recorded before running this script end on end for a specified time. Webload also

specifies an amount of hosts to use during this time to simulate users. The script is simply recorded by operating the application, and the software records the calls made and adds them to the script.

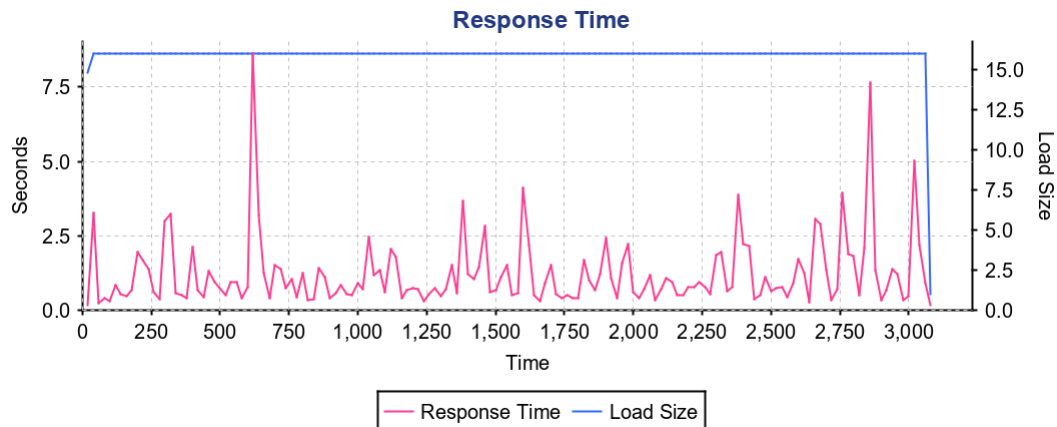


Figure 5.2: Webload: The response time of the application to repetitive calls over the course of an hour

Figure 5.2 shows the response times recorded over an hour-long testing with a 16 user load. The script in this case went through all the main features of the application including directions/predictions, stop/route search and user login and favourite saving. The large calls normally take around 2 seconds, which is shown in the trend. There are also occasional spikes where the response time exceeds 7 seconds. After some investigation this seems to be a database drawback as the data transfer units were set low due to cost. 16 calls to the database at the same time was causing the usage percentage to increase above 80%. In the future of the application, this data transfer rate should be increased to facilitate a higher number of calls.

5.2 Predictive Model - KNN

As mentioned in chapter 4, KNN was chosen for the predictive model due to its small file size and therefore greater speed of prediction. The models make predictions in less than half a second. This is an enormous advantage, and hence why they were chosen. The only question is their accuracy.

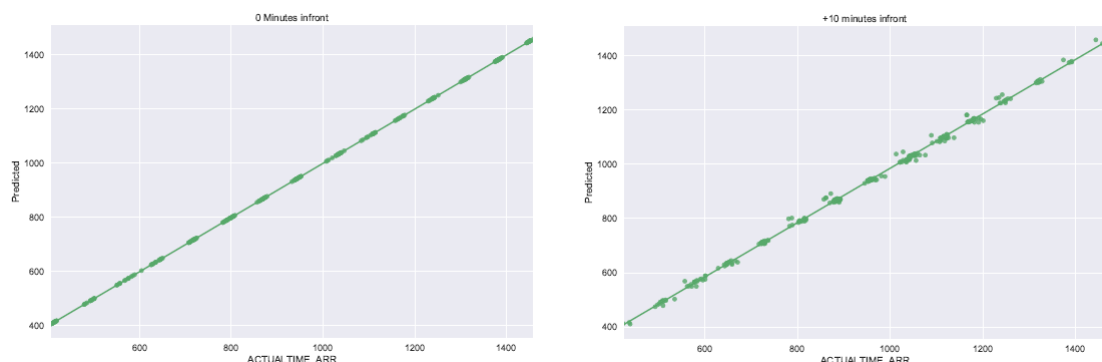


Figure 5.3: Route 56A: Actual vs Predicted arrival times

When testing the output of the KNN model for route 56A, the delay times can be summed from the actual and predicted values to give delay times. Grouping these delay times 5 minutes apart

and plotting them as seen in figure 5.3 shows an expected trend. As the delays increase, the spread between the points grows. This illustrates the drop in accuracy of the model when subject to unforeseen delays.

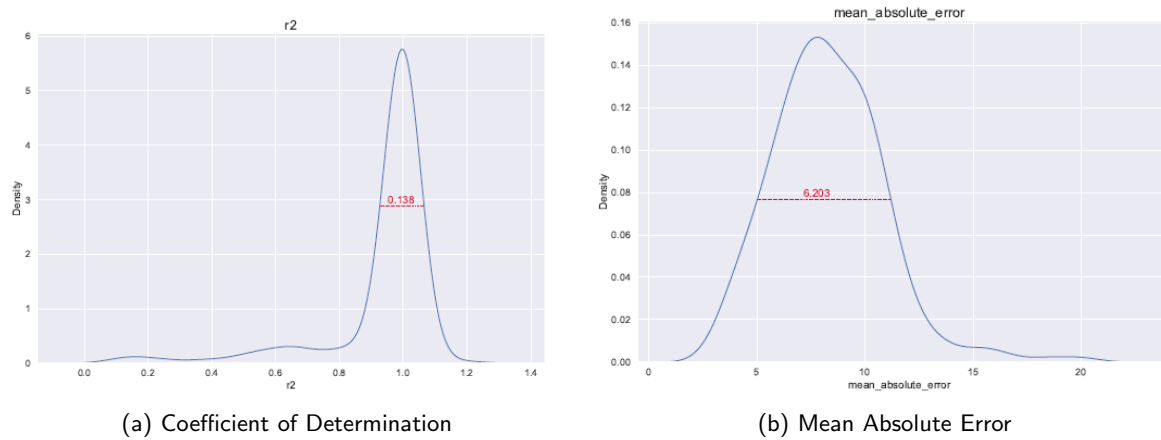


Figure 5.4: Cross Validation Metrics

The kernel density estimate, shown in figure 5.4, displays the cross-validation scores for the coefficient of determination (R^2) and the mean absolute error (MAE) for every model. While not the most telling of metrics, R^2 can quickly display how the predictions deviate from the line of regression [10]. Figure 5.4a represents a large portion of the models show some degree of linearity, which confirms the scatter plot findings of such a tight grouping in figure 5.3. The MAE, a common metric for evaluation and so is much more important to the results [11]. Figure 5.4b shows a large density of models with a MAE around 7.5 minutes. The full width at half max (shown in red) for this plot is relatively large at 6.203 minutes.

The size and speed of these models is exceptional and very favourable for the application in providing users with fast prediction times. The speed is also important for the next buses feature, which calls on the models. The models' mean absolute error could be reduced, but more importantly the full width at half max being reduced would greatly improve the reliability of the models. This improvement could be achieved by a more extensive feature analysis, and perhaps the most telling would be congestion data on the routes in question.

Chapter 6: Major Contributions

6.1 Description of Role

My role in the group for this project was coordination lead, which was my first choice. Some of the suggested responsibilities for this role in the role questionnaire at the beginning of the semester included: coordinating and solving group problems and leading development sessions, measuring group progress using measures defined by the team and managing the team diary and ensuring the group works according to the defined development process and methodology. In reality, we did not strictly follow these guidelines within each role and shared responsibilities across different roles, with each team member contributing to the responsibilities of each role. The exception to this was at the initial stages of the project, when we were still getting everything set up.

During the set up stage at the beginning of the first sprint I strongly advocated that we use Jira for bug and task tracking and Confluence for our project documentation. I had experience with these tools from our software engineering project in the previous semester and found them to have significant advantages over the tools suggested by other team members which were Trello for task tracking and a shared Google Document for documentation. Jira and Confluence are both developed by Atlassian so both tools were well integrated with each other, which helped link our documentation and issue/task tracking together quite well. Although these tools may have had a slightly harder learning curve than Trello and a shared Google Document, all team members agreed that it was worth the initial work to learn how to use it and were happy that we used these tools throughout the project.

The project was developed using Scrum Methodology with the workload being divided into four three-week sprints. As I did the initial set up with Jira and Confluence I was the Scrum Master for the first sprint, but as mentioned previously we shared the responsibilities of each role so we rotated Scrum Masters each sprint, with each person being Scrum Master for a sprint each. The role of Scrum Master entailed taking notes of the daily standups and any larger meetings we may have had such as pre-sprint meetings or post-sprint retrospectives.

6.2 Development Responsibilities

The team did not largely did not adhere to strict development roles where each member worked on an isolated component of the application. The exception to this is the data analytics/machine learning component of the application which was mainly managed by one team member, but the rest of the development was mainly shared between members, with some members working more on certain aspects than others. The main development aspects of the application that I was personally responsible for were the initial set up of Django on the back-end and back-end development in the early stages of the project and then writing the majority of the JavaScript for the application once we started building out the front-end. We also used Docker throughout the project, which I was responsible for researching and setting up the DockerFile for all team members to use but it had to be discarded at the end due to issues with deployment. I also wrote a bash script that all team members could use which ran all necessary commands to build

the correct docker image and container based on the DockerFile. All user interface design and implementation was shared evenly amongst team members. Bootstrap was used as a front-end CSS framework for the project and Bootstrap Studio was initially used to help design the UI, but this was later abandoned in favour of manually writing Bootstrap and CSS ourselves to allow us to better customise the application.

6.2.1 Django

Django was chosen as the backend framework for this project despite the fact that no team members had previously used it before. All team members had experience with Flask from the software engineering project in the previous semester but Django was chosen due to its additional built in functionality in comparison to Flask and its database ORM. As no team members were familiar with Django before beginning the project, there was an initial learning curve when first using the framework. It was my responsibility to do this initial set up and create the first views, models and scrapers that were developed. Django represents each table in the database as a “model”, which is a Python class which represents the table, with the attributes of the class representing the fields of the table and an instance of the class representing a row in the table. These classes then have a multitude of methods built in to modify the database. I set up the initial scrapers as class methods within the models so it could be called without needing to create an instance of the class, which was a convention we continued to use throughout the project.

Special care was taken to clearly comment everything in this initial stage and weekly code review sessions were held to ensure that other team members were brought up to speed with how Django works as a framework and how the various aspects being developed on the backend functioned. Although there was a learning curve when developing with Django, this was lessened due to the fact that there were certain similarities between Django and Flask and all team members had substantial experience coding in Python.

The other major component of the application that I was responsible for was the user accounts. These were implemented using an inherited custom class from the Django User class. This allowed the use of the in-built functionality provided by Django for user accounts which includes authentication and secure password hashing, while also allowing us to add additional attributes/fields to the user accounts. User accounts in Django are represented the same way as models, where an instance of the class represents a row in the database table for users. We added additional fields for users for fare status which could be either adult or child and was used to generate fare predictions, leapcard which was a Boolean for whether the user has a leapcard or not and was also used for fare predictions, along with favourite routes and favourite stops fields which were comma separated strings that could then be parsed as arrays on either the back-end or front-end. Favourite routes and stops are used in the routes and stops features to allow users to quickly access routes or stops they have saved.

6.2.2 JavaScript

It was decided to use Vanilla JavaScript for this project instead of using any JavaScript frameworks as no team members had any prior experience with such frameworks and it was felt that the time spent trying to learn a framework would not be worth any possible improvements it could bring to the application. Front-end development in JavaScript was by far the aspect of application development that I was most responsible for. As we began to build out the application after the initial set up of the back-end/database and planning stage of the project, JavaScript development became more and more important throughout the project as we added more features. All requests made to the back-end of the application are completed using the fetch API, which facilitates easy

use of AJAX requests. The await keyword is also used along with the fetch API in many functions where it is necessary to obtain the data being fetched before continuing to execute code in the function. In order for the await keyword to work correctly all functions that use it are defined as “async” functions which permits the use of this keyword which pauses the execution of code until a response is returned.

The entirety of the JavaScript for this project was contained in two files: map.js and scripts.js. map.js contains a singular function called initMap which is called via a callback function in index.html when the homepage and google maps api is loaded. This function sets up the map and loads all data necessary into the home page such as all bus stops and routes, along with adding the appropriate event listeners and initializing a class called AutoCompleteDirectionsHandler, which handles the directions feature of the application. Loading all this information upon loading the homepage increases initial loading times, but ensures that the app is then more responsive when later interacting with it as much of the relevant data has already been fetched from the back-end. Scripts.js holds all other JavaScript functions that are used in the application along with the AutoCompleteDirectionsHandler class which is initialised in maps.js.

The most complex component of JavaScript code for the project can be found in the Directions feature (see section 2.3.1), which makes use of the Google Directions API and the Google Places API. This feature is primarily driven by the AutoCompleteDirectionsHandler class and its route method in particular. This class takes an instance of a Google Maps map object, an array of all the markers loaded to the map which represent bus stops and instances of the directions handler and directions service objects from the Google Directions API, which handles the API request to Google Maps for directions and render the response on the map. All of these objects and the class are initialised in the aforementioned initMap function in Map.js.

The route method of the AutoCompleteDirectionsHandler class is called when a change in the input fields for origin or destination location is detected or upon clicking the “get directions and predictions” button. The input fields are text fields which automatically suggests locations as you type through an auto-complete feature implemented using the Google Places API. Alternatively, the user can opt to use their current geolocation as the origin location which is obtained using the Geolocation API which is supported by all modern web browsers.

Once the route method is called, the entered time and date by the user are taken from the input boxes and a request is made to the Google Directions API with the appropriate locations and date/time with a parameter indicating that only bus journeys should be returned from the API.. The API will then return a JSON response with walking and bus directions for each step of the journey. This response is then parsed to find the departure times, route names, departure stops and arrival stops for each bus step of the journey. This information is then passed to the back-end to obtain predicted travel times from the predictive model. Once all predictions are received from the back-end, all walking and bus steps are displayed to the user along with predicted times. All predicted walking times are taken from the API response as are bus travel times for routes that the model cannot generate a prediction for, which should only happen for a few select routes such as routes created after the historic data from 2018 that the models were trained on was generated. In addition to the predicted time for each step, a total predicted time for the total journey is also displayed to the user.

In addition to displaying directions and predicted travel times for each step, a fare estimation is also given to the user when possible. The application will show other bus routes than those ran by Dublin Bus and a predicted time from the API response if one of these routes is more appropriate. For these routes, a fare estimation is not displayed to the user as only fare estimations for Dublin Bus routes can be generated. Fare estimations are generated using a function which takes the user fare status, which can be either “adult” or “child” and a leapcard Boolean as parameters. The fare prediction is then calculated using hard-coded fare rates which were taken from the Dublin Bus website. These contain different fares for children and adults and different fares for each

depending on whether or not they own a leapcard. Children also have different fares depending on whether or not they are travelling between what Dublin Bus define as school hours. By default if a user is not signed in they are given a fare prediction for an adult with no leapcard but once a user signs in they can change their fare status and select whether or not they have a leapcard.

Along with these parameters, the number of “stages” a bus step takes is also used to generate a fare prediction. We could not find a definition for what a “stage” is either on the Dublin Bus website or from any other source but a stage seemed to be similar to a stop, meaning that if a bus journey passed five bus stops it would have five stages, so we took a “stage” to be a stop for the purposes of our application although we would like to generate more accurate fare predictions in future work if we can uncover a definition for a “stage” according to Dublin Bus. Along with the fare estimations for each bus step of the journey, a prediction for the total cost of the journey is given along with the total journey time at the bottom of the output displayed to the user.

The next most complex feature that was mainly implemented using JavaScript is the Routes feature (see 2.3.2). This feature displays all stops that the route passes and also displays the path the route takes on the map. Displaying the markers the routes follows was quite simple as all markers have attributes indicating which routes serve that stop. Displaying the path the route takes on the other hand was quite difficult. The route is drawn using the Google Directions API, but what complicated things is the fact that a specific route number cannot be sent as a request to the API and it can only accept locations from the Places API or co-ordinates. This meant that in order to draw the path a route takes a request had to be made for directions between the co-ordinates of the first stop on the route and the last stop on the route. This information alone was not sufficient as the Google Directions API would always just suggest the next route that was going between these co-ordinates instead of the route the user entered.

In order to get around this, the request includes a desired departure time which corresponds to the next timetabled departure time for that particular bus route, ensuring that the selected route number would always be returned by the Directions API as it would be leaving at the same time as the passed departure time. The start and end-co-ordinates along with the next timetabled departure time for the route are fetched from our back-end and then passed to the Directions API along with a parameter specifying that we would like multiple alternative routes to be returned instead of just the route Google determined to be the best.

The Google Directions API returns a JSON response in the same manner as in the directions feature but with multiple alternative routes. These routes are then parsed until the route number the user selected is found and it is then drawn on the map. In some scenarios the route selected by the user would not be returned in the alternative routes provided by API response, in which case the markers are still displayed but the route path cannot be drawn. However, it was found during testing that this was very rare and would usually only happen if a route was selected at a time in the day where there was no more scheduled departures for that route, as then the next timetabled departure time of the route could not be used as the desired departure time in the API request. This generally only happens if the feature is used very late at night or for certain routes that only run very early in the morning.

The stops feature is the final feature on the home-page which also involved JavaScript. This feature simply displays the marker representing a selected stop on the map and displays an info-window for the stop which displays the stop name/number, all routes serving the stop and the next several buses currently due to arrive at the stop and how far away they are in minutes. All of this information except for the incoming buses already exists as attributes for the marker from when the markers are created in the `initMap` function in `map.js` when the page is initially loaded. The incoming buses are obtained from the back-end, which gets the next timetabled buses to arrive at that stop for the next hour from the static GTFS data. These bus routes are then fed to the predictive model to generate a predicted arrival time for the next several buses which can be fed to the front-end on request and then displayed to the user.

Chapter 7: Background Research

Before beginning to write any code for this project, a certain degree of research had to be carried out in order to determine what tools would be used and the approach that would be taken for building the application. Personally, my role in the team for this project was very practical and coding-based so my background research was largely based on the tools we were using to build the application and learning how to use them initially. As mentioned previously in this document, Atlassian's tools Jira and Confluence were used for project management throughout the development of this application. These tools were chosen because of their widespread use by many highly successful companies, with 83% of fortune 500 companies using Atlassian products[12]. I personally also had prior experience using these tools in a previous project and found them to be highly effective so I advocated strongly to the group that we use them, which everyone was happy with after the initial learning curve and the power of these tools became clear. As a team we also did research on whether or not using a CSS framework would be worthwhile and it quickly became apparent that bootstrap was the tool best suited to our needs.

When deciding a back-end framework for this project we had the choice between Flask and Django. The selection was narrowed down to these two frameworks as all team members were most comfortable coding in Python and had prior experience in Flask. Django was chosen in the end due to its extensive documentation and tutorials which facilitated getting things running quickly [13, p. 31]. Django also has a number of built in features that helped set up aspects of the application such as user accounts quickly. Django is also widely used in the industry, being used by companies such as YouTube, Instagram and Spotify [14] and it was felt in the team that learning the framework would provide an excellent opportunity to both broaden our knowledge of back-end development and further our career prospects upon graduation. I personally did the bulk of the initial set up and coding in Django after following a number of online tutorials.

As mentioned previously, the team opted to use vanilla JavaScript for the front-end development for the project and not to learn a JavaScript framework. We briefly considered learning React due to its popularity in industry, being used by companies such as Facebook, Instagram, Twitter and Netflix [15], but opted against it as we felt any benefits that could be gained by learning the framework would be offset by the amount of time it would take to learn it.

Throughout the project Docker was used so the application could be run consistently on any of our computers and on the server when deployed in an attempt to avoid any potential "it works on my machine" issues [16]. I was personally responsible for following tutorials on Docker, learning how it works and then writing the DockerFile which sets up the image and container that would be used by the team. In simplified terms, a Docker image can be thought of as a set of instructions that are based off the DockerFile and can be used to build a Docker Container. A Docker Container is an instance of the image and works similarly to a customised virtual machine.

Generally speaking a project should only have one Docker image for any particular aspect of the application but any amount of containers can be built from this image. Docker worked well for the majority of the project for containerising the application, but when it came to deployment it became necessary to build a second docker image for nginx and to link the two containers together which proved quite time consuming so it was decided to discard Docker at this stage. Nonetheless, Docker was still a useful part of our development process as it allowed us to all work on the same development environment. All of the research on how Docker functions and all set up of Docker was conducted entirely by myself, and while it certainly had a steep learning curve, researching it and learning to use it was worthwhile and aided us greatly throughout the project.

Chapter 8: Critical Evaluation and Future Work

8.1 Critical Evaluation

Overall the project went relatively smoothly but of course there are some things that would be approached differently if the project could be done again. While it was felt that Django and Bootstrap were correctly chosen as frameworks for the project and it was the correct choice not to use a JavaScript framework, relying on Bootstrap Studio for much of the project to implement Bootstrap was likely a mistake. Bootstrap Studio is a drag and drop tool which is meant to make designing websites using Bootstrap quick and easy. Although using Bootstrap Studio in the initial stages did help to get an outline of the application quickly, it soon became clear that it was more of a hindrance than a help as the development process continued.

Once this stage was reached, it became very difficult to customise aspects of the UI beyond what this tool could do because no team member knew how to actually manually write bootstrap or could fully understand the code that Bootstrap Studio had produced. Once this stage was reached it became necessary to learn how to use Bootstrap without this tool and once the team switched to this approach it actually became much faster and easier to develop the UI and customise it to the team's liking. If this approach had been taken from the beginning of the project it is likely that the UI would be cleaner and more maintainable as all Bootstrap code would have been manually written and fully understood by the team.

Testing is another aspect of the application that did not go particularly well. Throughout the project it was planned to implement unit tests but this got pushed back continuously as the team chose to focus on developing features first and then write unit tests once things were functioning. While it was felt that this would quicken the development process, in hindsight the application may have actually been written faster if unit tests had been written first and the code would have been more robust. Many aspects of the code broke at different stages of development as new features were added or removed and having unit tests likely would have greatly sped up any debugging sessions and helped pinpoint what code was breaking. The application had a certain degree of code coverage from unit tests but these were almost entirely very simple tests that just checked if pages loaded, whereas it would have been far more useful if there had been the opportunity to test the more complex aspects of the application that were more likely to break.

While there were many meetings in the early stages of the project to discuss which innovative features we wanted to have in our application and how we would approach the project specification, perhaps this could have been greater defined within the group. Throughout the development process more features got added to the application as they were thought of but perhaps if all features had been planned from the beginning the various features would have interacted better with each-other, making for a more cohesive application. Although the directions, stops and routes features on the home page all function well, they do not interact with each-other at all with the exception of being able to select a stop to call the stops feature from within the routes feature.

The decision to use custom-made JSON files over a database-driven approach for the static GTFS data of the application was also likely a mistake. This approach was initially chosen when we were in the early stages of development and the only data being used was the full list of bus stops and routes from the GTFS data. As this data never changed and the entirety of the data was

loaded each time instead of selectively querying anything. At this stage of development, it made more sense to convert the data from txt files to locally stored JSON files and send them directly from the server to the front-end instead of querying the database, but as development continued and more and more complex features were added this stopped being the case. As more complex queries to the GTFS data had to be made, the JSON files became more and more complex as more detail was added to them which greatly increased their size. Along with this, many of the features we added only queried certain parts of the JSON files which had to be done by looping through the files which is likely much less efficient than querying a database. The code to scrape the GTFS txt files and make custom JSON files by combining the different original files was also quite time consuming for the team member in charge of this task and perhaps if a database based approach had been taken instead they would have had more time to help with other aspects of the application and the loading-speeds would also likely have been increased.

There are also a number of bugs that still remain in the application, particularly in the directions feature. Currently the application makes one singular call to the Directions API for an entire journey and the JSON response is parsed through and passed to the back-end so predictions can be made for each bus journey in the trip. The problem with this is that if some of the trips can't be generated with our predictive models and the predictions from the API response are used instead. This means that sometimes the predicted arrival time generated from our model for one bus may be for after when the following bus using the API's prediction is due to depart. The only way this could be fixed would be if instead of querying the API for the entire journey and passing it to the back-end, the API was instead queried for each bus trip in the journey and passed them to the API one by one. Unfortunately by the time this was realised the project was in the later stages of development and significant portions of the application would have had to be completely re-designed.

8.2 Future Work

Although the team was happy that our product met the minimum specification and a number of useful innovative features were added to the application, there is only so much that can be completed in a twelve week project and there are several features that would have been added if there was had more time. Firstly, the team would have liked to add far more unit tests and undertake more forms of testing on the application such as automated use of the website using a framework such as Selenium. If the opportunity for future work presented itself, this is likely the first thing that would be implemented before adding any additional features.

The second thing that should be attempted in future work would be to migrate our custom JSON file setup to a database-oriented setup. The performance of both approaches could then be compared and unless the JSON files were considerably faster than querying the database it may be better to migrate the JSON files to the database so the application would be more robust, as the scraper that generates the JSON files from the txt files is quite complex and has caused issues previously when trying to change aspects of it.

The application currently uses predictions from the Google Directions API for certain scenarios where a prediction cannot be generated using the predictive models (such as if a route only began running after 2018 when the data the models were trained on was generated). It would be nice in future work if this reliance on Google Maps could be reduced and the predictive models could instead be utilised for all Dublin Bus routes. As mentioned above in the critical evaluation, a key component of future work would be to re-design the system so it sends one API request at a time for each bus trip of the journey. This would be the only way to fix the bug where the predicted arrival time from the model is after the next bus's departure time according to API response.

Bibliography

- [1] EU Commission Expert Group. *Road transport: Reducing CO2 emissions from vehicles*. URL: https://ec.europa.eu/clima/policies/transport/vehicles_en. (accessed: 21.08.2021).
- [2] The European Automobile Manufacturers' Association. *Buses: what they are and why they are so important*. URL: <https://www.acea.auto/fact/buses-what-they-are-and-why-they-are-so-important/>. (accessed: 21.08.2021).
- [3] Jess Hohenstein et al. "Shorter Wait Times: The Effects of Various Loading Screens on Perceived Performance". In: May 2016, pp. 3084–3090. DOI: 10.1145/2851581.2892308.
- [4] Zhongheng Zhang. "Introduction to machine learning: k-nearest neighbors". In: *Annals of Translational Medicine* 4.11 (2016). ISSN: 2305-5847. URL: <https://atm.amegroups.com/article/view/10170>.
- [5] Alexandre Miguel de Carvalho and Ronaldo Cristiano Prati. "Improving kNN classification under Unbalanced Data. A New Geometric Oversampling Approach". In: *2018 International Joint Conference on Neural Networks (IJCNN)*. 2018, pp. 1–6. DOI: 10.1109/IJCNN.2018.8489411.
- [6] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks*. 5th. USA: Prentice Hall Press, 2010. ISBN: 0132126958.
- [7] Thompson Morgan Samaroo. *Software Testing : An ISTQB-BCS Certified Tester Foundation Guide*. Ed. by Brian Hambling. BCS Learning and Development Limited, 2015.
- [8] Tanuska Pavol, O. Vlkovic, and Lukas Spendla. "The Usage of Performance Testing for Information Systems". In: *International Journal of Computer Theory and Engineering* (Jan. 2012), pp. 144–147. DOI: 10.7763/IJCTE.2012.V4.439.
- [9] Juha Itkonen, Mika V. Mantyla, and Casper Lassenius. "Defect Detection Efficiency: Test Case Based vs. Exploratory Testing". In: *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*. 2007, pp. 61–70. DOI: 10.1109/ESEM.2007.56.
- [10] Dalson Figueiredo, Silva Júnior, and Enivaldo Rocha. "What is R2 all about?" In: *Leviathan-Cadernos de Pesquisa Política* 3 (Nov. 2011), pp. 60–68. DOI: 10.11606/issn.2237-4485.1ev.2011.132282.
- [11] Weijie Wang and Yanmin Lu. "Analysis of the Mean Absolute Error (MAE) and the Root Mean Square Error (RMSE) in Assessing Rounding Model". In: *IOP Conference Series: Materials Science and Engineering* (Mar. 2018). DOI: 10.1088/1757-899x/324/1/012049.
- [12] Atlassian. *Customers*. URL: <https://www.atlassian.com/customers>.
- [13] Devndra Ghimire. "Comparative study on Python web frameworks: Flask and Django". In: (2020).
- [14] *Top 10 Django apps and why companies are using it?* Apr. 2020. URL: <https://www.geeksforgeeks.org/top-10-django-apps-and-why-companies-are-using-it/>.
- [15] Spec India. *Top 25 Companies/Brands using Reactjs Development*. Jan. 2020. URL: <https://medium.com/front-end-weekly/top-25-companies-brands-using-reactjs-development-8be87b32cec2>.
- [16] *Why Docker?* URL: <https://www.docker.com/why-docker>.