

Activity Monitoring App

Module Code: CM3203

Student Number: C1519696

Author: Nathan Melly

Supervisor: Dr A I Abdelmoty

Moderator: Dr C Perera

Word Count: 24758

Abstract

Millions of people use wearable activity tracking devices to monitor their activity and maintain a healthy lifestyle. As a result, a wealth of data is gathered in these devices that can be used to learn about the user that recorded it. This project explores the creation of a mobile application to be used in conjunction with a Fitbit activity tracker, in order to build a user profile based on data gleaned from the device. Furthermore, the application will alert the user when they deviate from this profile.

Acknowledgements

I would like to express my gratitude to Dr Alia Abdelmoty for her guidance and support throughout this project.

Table of Contents

Abstract	2
Acknowledgements.....	3
Table of Contents	4
Table of Figures.....	6
Introduction	8
Project Aims	8
Personal Aims.....	8
Project Scope.....	8
Approach.....	9
Assumptions.....	9
Summary of Outcomes	10
Background	11
The Problem	11
Fitbit Fitness Trackers	11
Related Existing Software	11
Methods and Tools	14
Personas and Requirements.....	16
Personas.....	16
Requirements	18
Specification and Design	30
Use Cases	30
Data Mining and the Dynamic User Profile	34
User Interface Design	39
Heuristic Evaluation.....	44
Implementation	59
Software Architecture Diagram	59
Data Flow Diagram	60
Class Diagram	61
UI Code.....	62
Profile Building Code	64
Data Mining Code	71
Fitbit Communication	84

Database	85
Other Notable Code.....	85
Packages Used	86
User Interface.....	87
Results and Evaluation	92
Testing and Evaluation of Data Mining Approaches	92
Evaluation Against Requirements	94
Future Work.....	99
Network Errors	99
Changing Classifier.....	99
Adjusting Decision Tree Classifier Grouping Method.....	99
Remove Sleep Datasets	99
Classifying Days	99
Authentication Bug.....	99
Further Testing	100
Manual User Input.....	100
Conclusions	101
Reflection on Learning	102
Appendices	103
Appendix A	103
References	109

Table of Figures

Figure 1 Use Case Diagram	34
Figure 2 Colour scheme	39
Figure 3 Profile Page Design	40
Figure 4 Activity Page Design.....	41
Figure 5 Exercises Page Design	42
Figure 6 Exercise Page Design.....	43
Figure 7 Alerts Page Design	44
Figure 8 Software Architecture Diagram.....	59
Figure 9 Diagram of data flow within system.....	60
Figure 10 Class Diagram	61
Figure 11 Navigation	64
Figure 12 Getting a Profile ready to update	65
Figure 13 Requesting data	65
Figure 14 Creating profile activity values	66
Figure 15 Building profile activity values	67
Figure 16 Decision Tree creation	68
Figure 17 Creation of deviations after classification	68
Figure 18 Creation of deviations (detail).....	69
Figure 19 Creation and use of graph-based classifiers	70
Figure 20 Retrieval of intraday data	72
Figure 21 Processing of minute datasets into hourly datasets for decision tree	74
Figure 22 Decision tree hourly datasets stored in database.....	75
Figure 23 Decision tree classification.....	76
Figure 24 Example of a decision tree classifier.....	77
Figure 25 Initial retrieval and processing of data for graph-based classifier	78
Figure 26 Creation, formatting and storage of hourly datasets.....	79
Figure 27 Removing datasets from when the user was not wearing their device	81
Figure 28 Simulating a graph reading	82
Figure 29 Graph-based classifier boundaries	83
Figure 30 Profile Page Design	87
Figure 31 Profile Page Screenshot	87
Figure 32 Profile Page Screenshot (no profile)	87

Figure 33 Activity Page Design.....	88
Figure 34 Activity Page Screenshot.....	88
Figure 35 Exercises Page Design	89
Figure 36 Exercises Page Screenshot	89
Figure 37 Exercise Page Design.....	90
Figure 38 Exercise Page Screenshot.....	90
Figure 39 Alerts Page Design	91
Figure 40 Alerts Page Screenshot	91
Figure 41 Test results for both classifiers.....	93

Introduction

Devices such as smartphones, smart watches, and activity trackers have become ubiquitous in recent years. Users willingly give their personal information, such as location, health, activity, and sleep data, to these devices, and many people use them to track their fitness and daily activity. This data allows people to quantify how active they are, and users are likely to be motivated to do more exercise and track their progress as a result.

The aim of this project is to see how a user can be profiled based on data extracted from a Fitbit activity tracker, and what insights can be gained about a user regarding their lifestyle. I will create a mobile application that stores and analyses the extracted data. The data will be mined in order to build a profile for a user which will be displayed within the application. This profile will provide the user with information about their daily activity.

The application will be able to tell the user when they are deviating from their “normal” behaviour (i.e. if they are less active than usual). This will be useful for users as they will be able to see clearly how they are deviating from their regular activity, allowing them to rectify the deviation by being more active, thus maintaining (or improving) their profile.

This project will see two approaches to mining the user data used to build a model of the user, which can subsequently be used to classify new datasets. These approaches are a decision tree-based approach and a cumulative graph-based approach. Both classifiers will be outlined in this report before being compared to each other in the Results and Evaluation section of the report.

Project Aims

I aim to design and develop an android application that analyses data gathered from Fitbit devices and generates a user profile based on this data. The application should be able to alert the user when they are deviating from their profile. I aim to develop functionality within the application that visualizes how the values in the user’s profile have changed over time and allows the user to see how their daily statistics compare to these historic values.

Personal Aims

Before this project, I had some experience with JavaScript and no mobile application development experience, so I wanted to use this project as an opportunity to learn a new JavaScript framework and gain some experience in developing mobile applications.

Project Scope

The project scope evolved as the project progressed. The main aim of the project was originally to design and implement a mobile application that could build a user profile from a user’s data and alert the user when they deviated from this profile. This core aspect of the project did not change and remained the main focus of the throughout, but other aspects of the application planned at the start of the project eventually lost precedence to new ideas that were developed when creating the application, due to understanding different ways that a user’s Fitbit data could be utilised. Throughout this report, I will endeavour to highlight areas where plans were changed and provide insights as to why these decisions were taken.

The final scope of the project was the creation of an application that could build a user profile from a user's data that contained 'simple' information about the user, such as the number of steps, calories, floors etc. the user completes/burns/climbs in a day, whilst also using more advanced data mining techniques to build, test and update various models that could be used to find links between the user's activity data and their activity level, as defined by Fitbit (this could be one of four levels: 'sedentary'/'not active', 'lightly active', 'fairly active', and 'very active'). This project assesses the capabilities of two data mining approaches in the context of this activity tracking application.

Approach

At the start of the project, some research was carried out to identify how well existing mobile applications solve the problem. Following this, user personas and initial requirements were defined in order to gain an insight into how the application should ideally behave once the project is complete. This was followed by the development of use cases and user interface designs to gain a deeper understanding about what the application should be like in its final state.

Once an overall design for the application was created, the implementation began with the user interface and navigation, which are core aspects of the application. When these were working, the Fitbit authentication and request functionality was added. Next, the simple profile building was done, which involved requesting and processing the data in order to create the cards for steps, calories, etc. on the profile page. Once complete, the functionality to display the data in the activity and exercise pages was added, before the most challenging aspect was tackled: the model building and data classification.

Finally, the application was evaluated based on how well it met the requirements and the data classification aspects of the project were tested on real data in order to gain an understanding of how effective they are at modelling user activity.

Assumptions

The main assumption for this project is that users synchronise their Fitbit device with Fitbit before refreshing their profile. Based on this assumption, the application creates the user profile based on data from the date that the user got their Fitbit device to the day before they build the profile for the first time. Any subsequent profile builds on following days combine the original data from the first build with new data from between (and including) the last build date, and the day before the current day.

Another assumption is that the mobile device has a working, stable connection to either Wi-Fi or mobile data, and that the Fitbit API [1] is fully functional. Accounting for network errors is something that could be done in the future if the project were to be continued, however given the short timeframe of this module, this was deemed unnecessary.

This project assumes that the user has a specific Fitbit device, namely the Charge 3. This is because not all of Fitbit's trackers measure the same types of activity, so there is a question around whether this application would work with other devices in the Fitbit range. Future work could include ensuring that the application adapts to the user's Fitbit device, requesting only data that is available on that specific device.

Summary of Outcomes

This project finds that data mining techniques can be used to create classifiers based on the user's data. These classifiers can classify new user data using existing data. This is one way that the application can tell when the user has deviated from their profile. Other simple data analysis is done on user activity data to create average values for the user activities that can be compared to new data, in order to see how the user deviates from their usual activity. All the user's data and deviations are stored and displayed within an Android application.

The project evaluates the effectiveness of two data mining techniques within the context of the application, to decide which is more appropriate for use within the application in future. It is found that a graph-based method is more accurate than a decision tree method.

Background

The Problem

Activity trackers are becoming increasingly popular and users can record more data and learn more about their health and fitness as a result. Vast amounts of data have become readily available to users of activity trackers, which leads to the intriguing question of what can be done with this data. The overall problem that this project explores concerns how users can be profiled based on their activity, and ways in which they deviate from this profile. This data mining will be built in to a mobile application that allows users to view their data, and any deviations that the system finds. Within this problem however, some smaller, subproblems exist, including the following research questions:

- How can a user's data be used to model the user?
- What information can be derived about the user's activity?
- In what ways can a user deviate from their profile?
- How can deviations be detected?

Fitbit Fitness Trackers

Fitbit [2] produce a range of activity trackers, from small wristbands that measure 'simple' user activity such as steps, heart rate etc., to fully fledged smart watches that have the same core functionality of the activity trackers, whilst also having the capability to run more complex applications. The device used throughout this project was my personal Fitbit Charge 3, which can measure the 'simple' user activity as described above, as well as being able to automatically detect exercise sessions and track sleep. The specific data used throughout this project consists of the following:

- Steps
- Calories
- Distance
- Heartrate
- Floors
- Active Minutes
- Activities (exercise)

Throughout this project, steps, calories, distance, heartrate, and floors data is referred to as "activity data", and "activities" are referred to as "exercise data", unless specified otherwise.

The Fitbit data is accessible via Fitbit's Web API [1]. This allows applications to access users' data, enabling them to read, create, and modify it.

Related Existing Software

At the start of the project, other applications related to tracking user activity and exercise data were investigated. Given that the aim of this project was to create an Android application, research was focused on applications that are available on the Google Play Store, however this was not exclusive. Notes on the main features of each application were made, before evaluating how well the applications perform in the context of this project's research questions.

Application 1: Fitbit

Naturally, the first application explored was Fitbit's own offering to the mobile application market [2]. This application is used to setup Fitbit devices and synchronize data from the device with the Fitbit servers, and would likely be present on most users' mobile phones as a result, regardless of how usable the application is. Therefore, this application cannot be classed as a competitor to this project, however it is still a good reference point for what existing applications can do.

By default, the main screen of the Fitbit application shows the activity data listed above (steps, calories, distance, heartrate, and floors) at the top of the application screen. Below this are other sections concerning exercises, sleep, live heart rate, hourly activity, and food/drink tracking. Pressing on any of these buttons takes the user to a screen containing more detail about the chosen statistic.

Application 2: Strava

Strava [3] is a freemium application that allows users to record themselves running, swimming, or cycling. It places a focus on the social aspect of fitness and allows users to add friends, view each other's activity, and compete with one another, providing motivation to work out more.

Application 3: MyFitnessPal

MyFitnessPal [4] is a freemium application that allows users to manually enter food, drink, and exercise data in order to achieve weight, nutrition, and exercise goals. It uses user food and exercise data to calculate how many calories the user should consume to achieve their daily goal, and ultimately their overall weight goal.

Application 4: Rbitfit/Fitcoach

Fitcoach [5] is an R package that loads a user's Fitbit data and allows users to analyse the data with the aim of helping the user reach their fitness goals. It analyses which activity variables (steps, calories, distance etc.) have the greatest effect on the user reaching their goals and plots the performance of the user relative to these variables.

User Profiles and Modelling User Data

Fitcoach is the only application of the four that creates a model using the user's data. This model is created to find links between variables in the user data and a specific goal variable, set by the user. A user selects a goal variable and Fitcoach analyses data taken from Fitbit to tell the user which of the other variables are having the greatest impact in optimising their goal variable. This is a contrast to the other three applications, which do not seem to create models using the user's data. Fitcoach is also the only application that does not maintain a user profile; it simply analyses user data when provided some.

Fitbit's main screen allows users to view the data recorded by their device. Fitbit also creates 7-day summaries for the user that allow them to see how well they are achieving their goals. The Fitbit profile does not focus on analysing usual behaviour and is instead more focused on simply displaying the raw data from the device instead and telling the user whether they reached target values on a given day.

Strava's profile is one area within the application where user exercise sessions are found. The profile keeps track of how much exercise the user has recorded in each week and month and compares this monthly data to previous months. For running, swimming, and cycling data, Strava presents the user with 'Average Weekly Activity', 'Year-To-Date', and 'All Time' summaries of their data within the profile area of the application.

MyFitnessPal is focused on nutrition more than activity. Users still log exercise within this application however, and this information is used in conjunction with their food logs to tell the user how close they are to reaching a weight goal. This data is not used to create any information that one may expect to find in a profile though. The profile section of this application simply displays the user's goal and their progress towards reaching it.

Information Derived from User Activity

Fitcoach uses user data to analyse which variables are most useful in helping the user achieve a given goal. Furthermore, this model can be used to predict a value for the goal variable, given a sample of the user's data. This is the most interesting information gained from any of the four applications. The other three applications only provide the user with averages of values that the user inputs. Averages are a very useful way to give users an indication of their usual behaviour, so these applications are all good in this respect, however it is surprising more analysis is not done given the amount of data the applications gather.

Profile Deviations

Fitbit provides users with a 7-day summary of their data. This compares the user's data to their goals and tells the user which days they did not meet their goals in the last 7 days. This is good, however it tells the user how they deviate from goal values, not their usual behaviour. Strava, however, compares the amount of activity done by the user in the current month to their values from the previous month, highlighting whether the user has maintained the same amount of activity as in the previous month. MyFitnessPal does not tell the user when they deviate from their profile, as no values or model seem to be created within the profile to compare to. As mentioned previously, Fitcoach does not maintain a profile, and so is not capable of finding deviations within the user data.

Conclusions

A common theme throughout the applications that have user profiles is that the profile is primarily a place for users to view their data. Given the vast amount of data gathered in these applications however, it is surprising that more is not done with the data to learn about the user's behaviour. It was decided that this project would therefore focus on not only displaying data to users, but also using this data to provide the user with information about the way they behave. The user should be presented with a summary of values that describe how active they are on average in a day and tell the user ways in which their data is inconsistent with their usual behaviour when this is the case. The way that Fitcoach explores links between different variables within user data is interesting, however more could be done than simply finding links between types of data. This project will focus on using user data to classify user activity. This will allow the application to compare classifiers for different data in order to find deviations within the user's behaviour.

Methods and Tools

Methods

Potential Methods

Waterfall Method

The waterfall method is a linear approach that tackles each aspect of the software development lifecycle in order. This method would be good for this project as it would provide a clear structure to the project. This method may be unsuitable however as I have no experience with React Native, and designs and implementation are likely to change throughout the project as a result, and this method is too rigid to allow for these changes.

Agile Method

The agile method involves adding to the application one piece at a time. This approach involves planning and designing parts of the application as they are required. This method is very flexible and would be good for the project as it would allow for changing requirements and design regularly.

This approach may not work for this project as it would involve little planning at the start of the project. This means that the goal state of the application would be unclear throughout the project. This does not seem like a good idea considering the lack of experience with React Native. Ideally, as much as possible should be planned at the start of the project, however this should be flexible for changes to designs and requirements, should this be necessary.

Incremental Method

The incremental method of software development involves incrementally designing and developing parts of the application, adding more each time. This method would be good for this project as it does not require a full understanding of how each part will be implemented early in the project. This means that there is some flexibility in terms of changing requirements, which is likely for this project.

Using this method means that the most important aspects of the application can be implemented first, such as the user interface, before adding more functionality in stages until the application is complete. Furthermore, after each increment, there is a working application. This is ideal for this project as there are potentially many challenges that will not be planned. Using this method ensures that there will always be a working application with some core functionality, so there is some fallback if more challenging functionality causes problems.

Chosen Method

For this project, the incremental development method was adopted. This is largely because this method enables the application to be implemented in stages, allowing for flexibility with designs. I will endeavour to implement the simplest aspects of the application first, such as the user interface, before incrementally adding parts of the application until the application is finished. As the implementation progresses and becomes more challenging throughout the project, there will always be a working application from the previous increment if something goes wrong.

Tools

React Native

React Native [6] is a framework created by Facebook that allows for simultaneous development of Android and iOS applications using a single codebase. The framework uses JavaScript and React (also created by Facebook), and despite being a relatively young framework, is used in a range of popular mobile applications including Facebook, Skype, and Uber. I wanted to work with a new tool during this project, so I chose React Native for the reasons mentioned above.

Third-Party Packages

Third-party React Native packages and code will be discussed in the implementation section of this report.

Fitbit Charge 3 HR Tracker

Data for this project was gathered throughout the duration of the project from my personal Fitbit [2] activity tracking device. This data used in the project from this device is the steps, calories, heartrate, distance, floors, and active minute data. Data was accessed through the Fitbit Web API [1].

MongoDB

The Fitbit API [1] serves data in a JSON (JavaScript Object Notation) format and MongoDB [7] is a document database that stores data in “flexible, JSON-like documents” [8]. For this reason, and the fact that the application was built using a JavaScript-based framework, I decided that MongoDB would be a natural choice of database. The actual implementation used was a third-party React Native package called ‘react-native-local-mongodb’ [9]. MongoDB was preferred over an SQL-based database as data would have to be processed more in order to be stored in an SQL database, however it could be stored in a format close to its original format in a JSON-based database such as MongoDB.

GitHub (Git)

GitHub [10] is a website that provides version control using Git [11]. I decided to use Git to track changes to the application and revert any changes that I did not want to keep. Furthermore, the commit messages used throughout the implementation stage of the project provide some commentary on how the application changed over time.

Trello

Trello [12] is a project management tool that allows users to create ‘boards’ in order to manage tasks. Each board contains lists, which contain cards. Each card relates to a task. Lists used in this project include ‘Backlog’, ‘Open’, ‘In Progress’, and ‘Done’.

Balsamiq

Balsamiq [13] is the tool used for creating the design mock-ups in the Specification and Design section of this project.

Personas and Requirements

Personas

User Classes

The application was designed with a variety of users in mind. The users that the application was designed for can be broadly divided into four groups:

1. Novice fitness enthusiast with a good level of technical experience
2. Experienced fitness enthusiast with a good level of technological experience
3. Novice fitness enthusiast with minimal technological experience
4. Experienced fitness enthusiast with minimal technological experience

The application is aimed at fitness enthusiasts of varying experience levels. Novice fitness enthusiasts may only be interested in viewing statistics such as the number of steps they do or the number of calories they burn in a day, whereas more experienced users may be interested in actively tracking statistics and comparing how they have performed. Some users will also be more versed in technology and the use of mobile applications than others. Core aspects of the application should be intuitive and easily accessible by all users.

User Personas

Persona 1 – Primary Persona

Name: Clive Brown

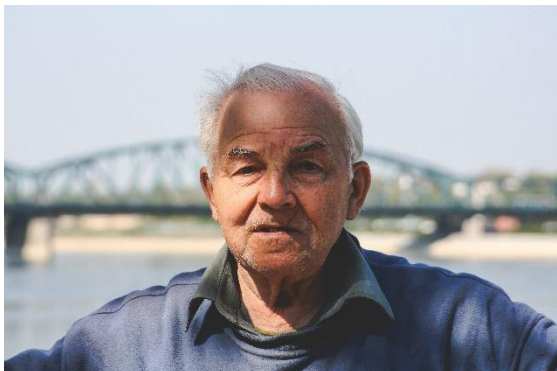


Image source: <https://www.pexels.com/photo/old-man-6110/>

Based on: Group 1

Quote: “I need to make sure I keep active so I can stay healthy throughout my retirement”

Description: Clive Brown is a 65-year-old semi-retired driving instructor. Until recently, Clive lived a sedentary lifestyle and his main hobby was fishing with his friends. When he gets home from a stressful day in the car, Clive regularly enjoys a bottle of wine after dinner with his wife. His wife suggested that he started being more active to ensure that he maintains good health when he retires; Clive has started going on morning walks with his wife before his first driving lesson each day as a result.

Goals:

- View how many steps he has done each day
- View his total distance covered each day
- View how many calories he burns each day

- Be alerted when he is not on track to do his usual number of steps
- Be alerted when he is not on track to move his usual distance
- Be alerted when he is not on track to burn his usual number of calories

Persona 2 – Secondary Persona

Name: Jane Bolton



Image source: <https://www.pexels.com/photo/three-women-s-doing-exercises-863977/>

Based on: Group 2

Quote: “I love using tech to track my daily activity because I can see how much fitter I am getting”

Description: Jane Bolton is a 25-year-old accountant. When she is not working or spending time with her friends, Jane enjoys working out at her local gym. She loves participating in fitness classes and has met many of her friends this way. Jane is aiming to run the London marathon soon and trains at least four times per week in preparation for it. This intense training means that Jane wants the ability to track how her performance improves over time to ensure that her training regime is working effectively.

Goals:

- Monitor how far she can run and the time it takes each day
- Train regularly and be notified if her performance levels drop

Conclusions

Based on the personas, it was decided that the application should provide simple information to the user, such as how many steps and calories they do on average each day. This information should be easy for the user to locate within the application as it is functionality that is likely to be used by most users.

Based on persona 2, the application should also have a dedicated section for the tracking of exercise sessions. This is something that is not as likely to be used by as many people, and so could be placed deeper within the application. This exercise section should display statistics from different exercise sessions to the user, such as the distance they travelled during a run.

Furthermore, the application was already intended to create a model of the user and let them know how well the model classified their data, however based on the personas, it would also be nice to alert the user as to how they deviated from the simple statistics such as average

steps and calories from day to day, as well as how their average values change after they update their profile with new data.

Requirements

Functional Requirements

Must

Requirement 1:

The system must build a profile from user activity data.

Acceptance Criteria

When data is uploaded to the application for the first time, the system begins building the profile and informs the user that the profile is being built. Once complete, the system tells the user that the profile building has finished, and the profile is displayed.

Justification

This will allow users to view a summary of their activity.

Requirement 2:

The system must alert the user when they are less active than their profile.

Acceptance Criteria

When the user refreshes their profile, deviations are displayed on the 'Alerts' screen, telling the user if/how they were less active than their usual profile.

Preconditions

- A user profile already exists.

Justification

This will encourage the user to be more active.

Requirement 3:

The system must allow the user to refresh their profile once per day

Acceptance Criteria

The user pulls the profile page down to initiate a refresh. The system provides feedback that a profile build has started. Once complete, the values in the profile change as appropriate and the user is alerted by the system that the profile build has finished.

Preconditions

- A user profile already exists.

Justification

This will allow users to compare new data to their profile to find deviations and rebuild their profile to include the new data.

Requirement 4:

The system must provide feedback to the user when a profile refresh has started and finished.

Acceptance Criteria

After the user has initiated a refresh, the system provides feedback that a profile build has started. Once complete, the user is alerted by the system that the profile build has finished.

Preconditions

- A profile refresh has been initiated by the user.

Justification

This is so that the state of the system is clear to the user.

Requirement 5:

The application must remain responsive whilst the profile is being built.

Acceptance Criteria

When a profile build or refresh is in progress, the user can still navigate around the application and view their data.

Preconditions

- A profile build or refresh has been initiated.

Justification

This is so that the application always remains usable.

Requirement 6:

The system must build a model of the user's activity level based on the activity values in their profile.

Acceptance Criteria

When data is imported into the application, a model is built based on the data. When more data is imported later, the model is used to classify the new data. Any deviation between the predictions and the classifications are highlighted in the Alerts tab.

Justification

This will allow the system to classify new data, in order to find areas of the user's activity that deviate from the norm.

Requirement 7:

The system must allow the user to view historical activity data in a graphical format.

Acceptance Criteria

When the user presses on any of the activity statistics on their profile page, they are directed to a page containing a list of daily values for the chosen activity and a graph containing the values.

Justification

This is so that the user can easily compare how their activity data from one day compares to others.

Requirement 8:

The system must allow the user to view historical activity data in a list format.

Acceptance Criteria

When the user presses on any of the activity statistics on their profile page, they are directed to a page containing a list of daily values for the chosen activity and a graph containing the values.

Justification

This is so that the user can efficiently read their data, meaning that they can learn about their activity.

Requirement 9:

The system must allow the user to view the durations of historical exercise sessions in a graphical format.

Acceptance Criteria

When the user presses on any of the cards on the exercise page, they are directed to a page containing a list of values for each session of the chosen exercise, as well as a graph of the values and a picker that allows the user to choose which statistics they would like to view. The user selects 'Time' from this picker and the duration of all exercise sessions of the chosen type are displayed in the graph and list.

Justification

This is so that the user can easily compare how their exercise time from one session compares to others.

Requirement 10:

The system must allow the user to view the durations of historical exercise sessions in a list format.

Acceptance Criteria

Same as requirement 9.

Justification

This is so that the user can efficiently read their data in a structured and organised way.

Requirement 11:

The system must allow the user to view the calories burned during historical exercise sessions in a graphical format.

Acceptance Criteria

When the user presses on any of the exercise types on the exercise page, they are directed to a page containing a list of values for each session of the chosen exercise, as well as a graph of the values and a picker that allows the user to choose which statistics they would like to view. The user selects 'Calories' from this picker and the calories burned during all exercise sessions of the chosen type are displayed in the graph and list.

Justification

This is so that the user can easily compare how the number of calories burned in one session to others.

Requirement 12:

The system must allow the user to view the calories burned during historical exercise sessions in a list format.

Acceptance Criteria

Same as requirement 11.

Justification

This is so that the user can efficiently read their data in a structured and organised way.

Requirement 13:

The system must allow the user to view the distance covered during historical exercise sessions in a graphical format.

Acceptance Criteria

When the user presses on any of the cards on the exercise page, they are directed to a page containing a list of values for each session of the chosen exercise, as well as a graph of the values and a picker that allows the user to choose which statistics they would like to view. The user selects 'Distance' from this picker and the distance covered during all exercise sessions of the chosen type are displayed in the graph and list.

Justification

This is so that the user can easily compare the distance covered in one session to others.

Requirement 14:

The system must allow the user to view the distance covered during historical exercise sessions in a list format.

Acceptance Criteria

Same as requirement 13.

Justification

This is so that the user can efficiently read their data in a structured and organised way.

Requirement 15:

The system must display the number of steps the user does on average in a day on the profile page.

Acceptance Criteria

When the user opens the profile page, a card containing the number of steps the user does on average is displayed.

Preconditions

- A user profile has been built.

Justification

This is so that the user can gain an understanding of how many steps they usually do.

Requirement 16:

The system must display the number of calories the user burns on average in a day on the profile page.

Acceptance Criteria

When the user opens the profile page, a card containing the number of calories the user burns on average is displayed.

Preconditions

- A user profile has been built.

Justification

This is so that the user can gain an understanding of how many calories they usually burn.

Requirement 17:

The system must display the distance the user covers on average in a day on the profile page.

Acceptance Criteria

When the user opens the profile page, a card containing the distance the user covers on average is displayed.

Preconditions

- A user profile has been built.

Justification

This is so that the user can gain an understanding of how much distance they usually cover.

Requirement 18:

The system must display the number of floors the user does on average in a day on the profile page.

Acceptance Criteria

When the user opens the profile page, a card containing the number of floors the user does on average is displayed.

Preconditions

- A user profile has been built.

Justification

This is so that the user can gain an understanding of how many floors they usually climb.

Requirement 19:

The system must display average resting heart rate of the user on the profile page.

Acceptance Criteria

When the user opens the profile page, a card containing the user's average resting heart rate is displayed.

Preconditions

- A user profile has been built.

Justification

This is so that the user can learn how fast or slow their usual resting heart rate is.

Requirement 20:

The system must display the number of active minutes the user completes on average in a day on the profile page.

Acceptance Criteria

When the user opens the profile page, a card containing the number of active minutes the user does on average is displayed.

Preconditions

- A user profile has been built.

Justification

This is so that the user can gain an understanding of how many minutes they are usually active for in a day.

Requirement 21:

The system must store the user's data.

Acceptance Criteria

The user closes the application and then opens it again to find the same data present that was there before the application was closed.

Preconditions

- Data has been imported from Fitbit.

Justification

This is so that the user does not need to keep importing their data whenever they wish to use the application.

Requirement 22:

The system must rebuild a profile from user datasets when new data is entered or imported.

Acceptance Criteria

Once data has finished importing, a refresh is initiated. The system provides feedback stating that a build is in progress. Once complete, the values in the profile change where appropriate and the user is alerted that the build has finished by the system.

Justification

This is so that the profile is as up-to-date as possible. This also helps maintain consistency, as the user can safely assume that the profile that they are presented is up-to-date and uses all available data.

Should

Requirement 23:

The system should allow the user to manually log data for an activity.

Acceptance Criteria

On the page for a given activity, the user presses a '+' button. They then input and submit a value for the given activity before being returned to the screen for the activity. The newly added dataset is displayed in the list and on the graph.

Justification

This is so that the user could use aspects of the application without the need for a Fitbit device. This is also beneficial for users who own a Fitbit but were not wearing it for some time.

Requirement 24:

The system should allow the user to edit basic profile information.

Acceptance Criteria

The user can press a button on the profile page that navigates them to an 'Edit Profile' page where the user can edit basic information such as name and age.

Justification

This is so that the user can ensure that their details are up-to-date within the application.

Requirement 25:

The system should allow the user to manually log the date, time, duration, distance, and type of an exercise session.

Acceptance Criteria

On the exercises page, the user clicks a '+' icon that opens a 'Log Exercise' form. The user must input the relevant values before submitting the form. When opening the exercise page for the given exercise type that they created, the newly added values appear in the list and graph on the relevant view on the page (for example, duration would appear on the 'Time' view).

Justification

This is so that the user could use aspects of the application without the need for a Fitbit device. This is also beneficial for users who own a Fitbit but were not wearing it for some time.

Requirement 26:

The system should highlight the average value of an activity on the graph for that activity.

Acceptance Criteria

On the graph for any activity, a line is displayed across the graph that shows the average value for the activity.

Justification

This is so that the user can easily compare their data with their profile value (the average).

Requirement 27:

The system should derive the user's most common exercise type and display this in the profile page.

Acceptance Criteria

On the profile page, there should be a "Most common exercise" card that tells the user what their most common exercise is.

Justification

This is so that the user is aware of what exercise they spend the most time doing.

Requirement 28:

The system should order the exercise types on the exercise page based upon the user's data, with their most frequently done exercise types at the top of the list.

Acceptance Criteria

On the exercise page, the exercises are listed in order of how much time the user spends doing each one. This order can be verified by checking the total amount of time spent doing each exercise type within the Fitbit application.

Preconditions

- The user has logged at least two exercise sessions of different types.

Justification

This is because the amount of time spent doing an exercise is an indication of how interested the user is in that exercise. The user is likely to want to view data for exercises that they are most interested in, and so these should be easy to locate.

Requirement 29:

The system should alert the user when they have not recorded as many sessions of their most common exercise as usual.

Acceptance Criteria

On a weekly basis, the system should send the user a notification if they have not recorded as many sessions as usual during the previous week.

Preconditions

- The user has logged exercise sessions.

Justification

This is so that the user is aware that they are not on track to keep up with their usual behaviour.

Requirement 30:

The system should only display exercise types on the exercise page if the user's data contains at least one session of that exercise type.

Acceptance Criteria

All types of exercises present within the Fitbit application should be checked. All types logged within the Fitbit application should be present within this application.

Justification

This is so that the user does not need to waste time looking through a list of exercise types that they have never done to find one that they have and want to view the data for.

Could

Requirement 31:

The system could base deviation alert times on the user's usual wake up and sleep times where this data is available

Acceptance Criteria

If the user's wake up time is 9am and sleep time is 11pm, they are awake for 14 hours. If they set the application to give them notifications after every 25% of the day (excluding sleep time), then the application should generate alerts at 12:30pm, 4pm, and 7:30pm.

Justification

This is so that the user is reminded about exercise at times that are suitable for them.

Requirement 32:

The system could create challenges for the user based on data in their profile.

Acceptance Criteria

When the user opens the application or the first time on a given day, challenges appear in the 'Alerts' page.

Justification

This is so that the user has targets to meet so that they stay active. The fact that they are based off the user's data makes the challenges more achievable. The application could set a target for all users of some random value such as 10,000 steps per day, however this may seem like a lot to some people. If someone walks 5,000 steps in a day, then a challenge of 5,500 or 6,000 steps would much more achievable. If the user maintained this level of activity, eventually their profile average would increase, and so the challenges would get harder.

Requirement 33:

The system could provide insights based on the user's sleep data (such as the amount of time the user usually spends in each stage of sleep).

Acceptance Criteria

The user opens a 'Sleep' page within the application where they are presented with cards containing information derived from their Fitbit sleep data, or a message stating that there is no sleep data in the database if that is the case.

Justification

It would be beneficial for users to learn how much sleep they are getting, as well as the quality of the sleep that they are getting.

Requirement 34:

The system could allow the user to import data directly from their Fitbit profile.

Assessment Criteria

When building the profile, the user is redirected by the application to a Fitbit authentication page where they must enter their Fitbit credentials in order to allow the application to access their Fitbit data. Once the build has finished, data visible in the application matches relevant data within the Fitbit application.

Justification

This is so that the user does not need to manually input data.

Requirement 35:

The system could allow the user to import new data in a file.

Acceptance Criteria

The user presses a button that opens a folder containing data files. The user selects the file they wish to import into the application and the system parses the file and updates the user's profile as appropriate.

Preconditions

- The file must be present on the device.

Justification

This is so that the user does not need to manually input data.

Requirement 36:

The system could allow the user to manually set exercise reminders.

Acceptance Criteria

On the exercises page, the user presses an icon that opens a 'Set Reminder' dialog. The user must input a name for the reminder and a time. Once the reminder has been confirmed, the user must be notified about the reminder by the system at the correct time.

Justification

This is so that the user can ensure that they are keeping active and healthy.

Requirement 37:

The default start time for exercise reminders could be derived by the system from the start times of previous exercises.

Acceptance Criteria

On the exercises page, the user presses an icon that opens a 'Set Reminder' dialog. The user must input a name for the reminder. The time for the reminder has been automatically populated by the system with a value derived from start times of previous exercise sessions.

Preconditions

- The user has logged previous exercise sessions.

Justification

This is so that the user can be reminded to exercise at around the same time of day each time.

Will Not

Requirement 38:

The system will not allow users to view other users' profiles.

Acceptance Criteria

There is no sequence of events within the system that allows one user to view the profile of another user.

Justification

Due to the personal and private nature of the data, it would be inappropriate to allow other users to read a user's data.

Non-Functional Requirements

Requirement 39:

The profile should take no longer than $n * 20$ seconds to rebuild after an import of n days of data.

Acceptance Criteria

A timer should begin at the start of the build process of n days and stopped once the build has completed. If the timer reads a value less than $n * 20$ seconds, then the requirement is met.

Justification

This is so that the user does not have to wait for a long time when a profile is being built.

Requirement 40:

The application should be intuitive and easy to use.

Acceptance Criteria

The application should be reviewed against Nielsen's Usability Heuristics.

Justification

This is so that the user has a good experience when using the application.

Specification and Design

Use Cases

This section of the report outlines the main use cases for the application. For other use cases, see Appendix A.

Actors

Fitbit Owner

A person that owns a Fitbit and wishes to use their device in conjunction with the application.

Database

The database within the application.

Fitbit Web API

The API [1] that provides data, gathered by the Fitbit device, that will be stored in the database for future use.

Original Use Cases

Create Profile (and import data)

Preconditions

- The application is already running.

Main Flow

1. **Press the Profile icon**

This use case starts when the user presses the profile icon on the main tab bar at the bottom of the application screen. The profile page is displayed. A message is displayed that tells the user to pull the screen down to build a profile.

2. **Pull the screen down**

The user presses on the screen and drags their finger down the screen to initiate a refresh.

3. **Sign in to Fitbit**

The user is presented with a Fitbit login screen. The user must sign in using their Fitbit credentials and then confirm that they allow the application to access data from their Fitbit profile.

4. **Contact Fitbit Web API**

The system starts requesting the data from the Fitbit Web API. The data returned in the response is stored in the local database.

5. **Build Profile**

The system uses the newly stored data to build a profile for the user. Once complete, the user receives an alert stating that profile building is complete. **Note:** Deviations are created at this stage of the use case.

Alternative Flows

3A. **Sign in to Fitbit (system)**

The user has already authenticated the application with Fitbit recently, so the application can use refresh tokens created from the initial authentication to reauthorize the user without them having to input their Fitbit credentials again.

View Profile

This use case allows the user to view their profile within the application. The profile is the main page within the application and is opened by default when the application is opened. The profile page is accessible within the application from the profile icon on the main application tab bar.

Preconditions

- The application is already running.

Main Flow

1. **Press the Profile icon**

This use case starts when the user presses the profile icon on the main tab bar at the bottom of the application screen. The profile page is displayed.

Refresh Profile (and import new data)

Preconditions

- The application is already running.
- The user has created a profile.

Main Flow

1. **Press the Profile icon**

This use case starts when the user presses the profile icon on the main tab bar at the bottom of the application screen. The profile page is displayed.

2. **Pull the screen down**

The user presses on the screen and drags their finger down the screen to initiate a refresh.

3. **Sign in to Fitbit**

The user is presented with a Fitbit login screen. The user must sign in using their Fitbit credentials and then confirm that they allow the application to access data from their Fitbit profile.

4. **Contact Fitbit Web API**

The system starts requesting the data from the Fitbit Web API. The data returned in the response is stored in the local database.

5. **Build Profile**

The system uses the newly stored data combined with the data that was already in the system to build a new version of the profile for the user. Once complete, the user receives an alert stating that profile building is complete. **Note:** Deviations are created at this stage of the use case.

Alternative Flows

- 3A. **Sign in to Fitbit (system)**

The user has already authenticated the application with Fitbit recently, so the application can use refresh tokens created from the initial authentication to reauthorize the user without them having to input their Fitbit credentials again.

View Activity Data

This use case allows the user to view their activity data for any given activity. The activity page can be reached by pressing on any of the activities on the profile page.

Preconditions

- The application is already running.
- The user has synchronised the application with Fitbit.
- The user has tracked activity with their Fitbit device.

Main Flow

1. Press the Profile icon

This use case starts when the user presses the profile icon on the main tab bar at the bottom of the application screen. The profile page is displayed.

2. Press an activity card

The user presses a card on the profile page and is taken to a page that contains more detailed data for that activity.

View Profile Deviations

This use case allows the user to see the ways that they have deviated from their usual profile.

Preconditions

- The application is already running.
- The user has created a profile prior to the current day.
- The user has refreshed their profile on a day after the initial profile creation date, thus creating deviations.

Main Flow

1. Press the Alerts icon

This use case starts when the user presses the alerts icon on the main tab bar at the bottom of the application screen. The alerts page is displayed, containing the profile activity deviations, activity level deviations, and profile updates.

Updated Use Cases

As the project progressed, the use cases evolved with it. Some use cases were added, whilst some original use cases were deemed unnecessary. This section highlights these changes.

New Use Case: Edit Settings

This use case allows users to edit any settings within the application, such as the name of the user, or perform tasks such as clearing the database. Originally, the application was going to have separate settings pages for the Profile, Exercises, and Alerts areas of the application, however this was deemed unnecessary within the scope of the project and a single 'Settings' page was created. This new 'Settings' page also replaces the 'Edit Profile' page.

Preconditions

- The application is already running.

Main Flow

1. **Press the Profile icon**
This use case starts when the user presses the profile icon on the main tab bar at the bottom of the application screen. The profile page is displayed.
2. **Press the Settings icon**
The user presses the gear-shaped icon to open the 'Settings' screen.
3. **Edit settings**
The user makes whatever changes they wish to their settings.

Alternative Flows

- 3A. **Cancel the edit**
The user presses the 'Back' button and if the user made changes, they are asked to confirm that they do not want to submit the changes before cancelling. The user is returned to the profile page.

New Use Case: Clear database

This use case allows users to delete all data within the application. This means that users can then create a fresh profile if there are any issues in their current profile.

Preconditions

- The application is already running.

Main Flow

1. **Press the Profile icon**
This use case starts when the user presses the profile icon on the main tab bar at the bottom of the application screen. The profile page is displayed.
2. **Press the 'Settings' icon**
The user presses the gear-shaped icon to open the 'Settings' screen.
3. **Press 'Clear All Data'**
The user presses the 'Clear All Data' button. Once the data is cleared, an alert pops up to notify the user that the database has been cleared.

Removed: Edit Profile Information

Replaced by new 'Edit Settings' use case defined earlier in this report section.

Removed: Edit Profile Settings

Replaced by new 'Edit Settings' use case defined earlier in this report section.

Removed: Edit Exercise Settings

Replaced by new 'Edit Settings' use case defined earlier in this report section.

Removed: Edit Alert Settings

Replaced by new 'Edit Settings' use case defined earlier in this report section.

Use Case Diagram

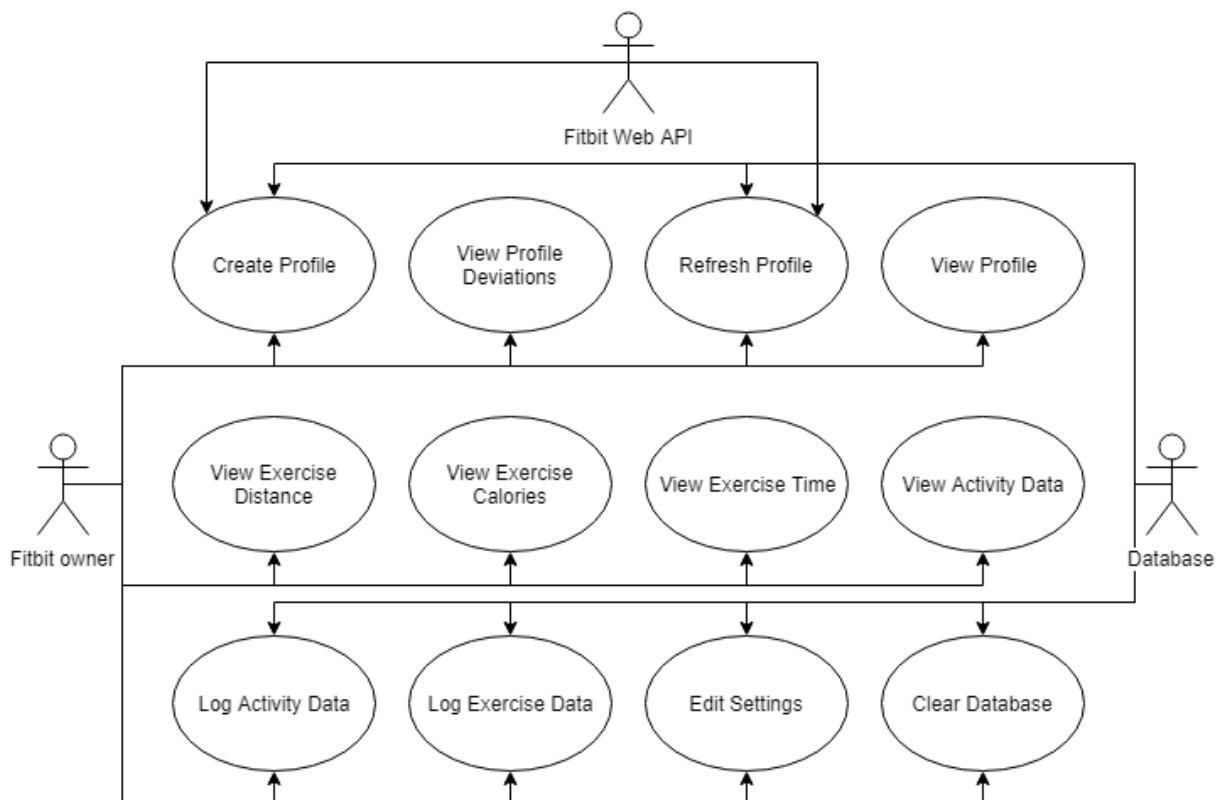


Figure 1 Use Case Diagram

Above is a use case diagram that shows which use cases the user and parts of the system are involved in.

Data Mining and the Dynamic User Profile

One of the aims of this project is to see how users can be profiled using their data. This led to the idea of creating a dynamic user profile. This is a profile that changes over time in accordance with the user data. This profile consists of average values for activities such as steps and calories, but the application is capable of more than this. This section of the report defines how the application creates models based on the user data, which enable the application to classify new user data based on data that is already stored in the application. Every time data is imported, classifiers are created from existing data. These models classify the new data and deviations are created as a result. Next time data is imported, the data from the previous import is included when creating the models, so in theory, the models should get more accurate with time.

Data Types

Decision Tree Data Mining

When looking for appropriate data mining algorithms to use for the project, it was decided that a decision tree algorithm would potentially be useful in the context of this project. A tutorial from a Google developer was found at <https://www.youtube.com/watch?v=LDRbO9a6XPU>. Code to accompany this tutorial was found at https://github.com/random-forests/tutorials/blob/master/decision_tree.ipynb

[14]. This tutorial used Python to implement a decision tree and its component parts. This code was rewritten in JavaScript for this project, and it contains three data types: Question, LeafNode, and DecisionNode. Descriptions of these data types below are heavily based from the descriptions by the author of the original Python code [14].

Question

This data type represents a question in the form of "is $x \geq y$ " if x and y are numerical, or "is $x == y$ " otherwise. Questions are used to partition a dataset [14] into smaller datasets of rows that are true or false for the question.

DecisionNode

A decision node is used to ask a question [14] and store a reference to the two children nodes, created from the true and false datasets for the question.

LeafNode

A leaf node is used to classify the data [14]. It maps the class to the number of times the class appears in the rows from the training data that reached the leaf [14].

Deviation

This data type was implemented to represent profile deviations. Deviations are created for incorrect classifications of hourly data or data from a six-hour window (one of 00:00-05:59, 06:00-11:59, 12:00-17:59 and 18:00-11:59). It consists of six pieces of information:

1. Date – the date that this deviation applies to.
2. Time of day – the time of day that this deviation applies to. If the deviation is hourly, then this value is from the range 0-23, each representing an hour from the day (e.g. 0 would represent 00:00-00:59). If the deviation is six-hourly, then the value is from the range 0-3, where each value represents one of the four possible windows throughout the day (e.g. 2 would represent 12:00-17:59).
3. Time type – this defines whether the deviation is hourly or six-hourly.
4. Category – this defines what part of the user profile has deviated from the norm, e.g. 'Steps', or 'Activity Level'.
5. Expected value – the value that was expected based on data or the model from the user profile.
6. Actual value – the actual value. In the case of 'Activity Level' deviations, this value represents what the data mining model classified the dataset as. For other values, such as 'Steps', this value is simply how many steps the user did in the timeframe defined in part 2 of this list.

Algorithms

Decision Tree Data Mining Approach

Decision Tree

A decision tree was adapted from one written in Python by J Gordon which can be found at https://github.com/random-forests/tutorials/blob/master/decision_tree.ipynb [14] as mentioned in the 'Data Types' section. This decision tree consists of a recursive algorithm to build the tree, which makes use of other, smaller algorithms. The noteworthy algorithms are explained in this section of the report.

Build Tree

Algorithm:

1. Find the question that gives the most information gain.
2. If the information for the question is 0, return a leaf node containing a reference to the dataset.
3. Otherwise, partition the dataset into two datasets, one for data points that provide an answer of 'true' to the question, and another for 'false' data points.
4. Recursively run 'Build Tree' on the 'true' dataset.
5. Recursively run 'Build Tree' on the 'false' dataset.

Find Best Split

Algorithm:

1. Initialise the following variables to these values:
 - a. Current best gain = 0
 - b. Current best question = null
 - c. Current uncertainty = gini(dataset)
2. For each column of the dataset (steps, calories, etc.) excluding the classifier column (final column):
 - a. For each distinct value in this column:
 - i. Create a question for the value
 - ii. Partition the dataset into true and false datasets.
 - iii. Calculate the information gain from this split.
 - iv. If the calculated gain is better than 'current best gain', then set 'current best gain' to the calculated gain and set the 'current best question' to the question.
 - b. Return the best gain and question.

Partition

Algorithm:

1. Initialise two empty arrays, one for the 'true' data points, and one for the 'false' data points.
2. For every data point in the dataset to be split:
 - a. If the data point satisfies the question used for the split, then add the data point to the 'true' array, otherwise add the data point to the 'false' array.
3. Return the arrays.

Classify

Algorithm:

1. If the node is a leaf, return the predictions associated with the node.
2. Otherwise (the node is a decision), if the data point to be classified satisfies the question associated with this node, then recursively call Classify on the data point and the 'true' branch of the node. If the data point does not satisfy the question, call Classify on the data point and the 'false' branch of the node.

Gini

Input: A set of data points.

Algorithm:

1. Count the number of instances of each class within the dataset.
2. Initialise impurity to 1.
3. For each class in the dataset:
 - a. Calculate the probability of the class within the dataset (number of instances of class / total number of data points)
 - b. $\text{Impurity} = \text{impurity} - (\text{probability})^2$
4. Return impurity

Data pre-processing

Before the algorithms above can be used, the dataset must be in a certain format. This format is an array of arrays. Each inner array represents a data point of the form [steps, calories, floors, distance, heartrate, class], where the final item, 'class', is the activity level associated with the data point.

Furthermore, there are other issues to tackle at this stage, namely missing datasets and grouping datasets.

Missing Datasets

Fitbit automatically populates fields like steps and calories with a default value, even when the user is not wearing their device. Of all the fields that are used within this application, the only one that does not contain this guessed data is heart rate. Therefore, it was necessary to design an algorithm to match the other datasets to the heart rate dataset.

Input: An object containing daily data for all types (in the form of objects)

Algorithm:

1. Create an empty array containing empty array for each type of data (steps, calories, floors, distance, heart rate).
2. For each type of data
 - a. If we are not looking heart rate currently, iterate over the length of the heart rate dataset:
 - i. Get the minute-by-minute data for heart rate and the current data type.
 - ii. Iterate over the minute-by-minute heart rate data:
 1. Get the time of the current minute.
 2. Iterate over the other type of minute-by-minute data from the current index of the heart rate data to the final index of the other data type's data.
 - a. If the current minute of the other data type matches the heart rate minute, then a matching dataset has been found. Push this data point to the relevant array created in step 1.

- b. Otherwise, iterate over the heart rate dataset and add each point into the heart rate array created in step 1.
3. Return the 2D array created in step 1, which is now populated.

Grouping Datasets

Fitbit provides an activity level for the user at any given minute. This is one of: 'sedentary' (referred to as 'not active'), 'lightly active', 'fairly active' and 'very active'. This is useful as this activity level is used as a classifier for the user's data, however there is an issue in that when the user's data is grouped into hourly groups, there is no classifier for data at this granularity. The workaround for this was to process the minute-by-minute data into the desired format and create a decision tree for each hour. A dataset containing the average values for this hour can then be created and classified using this decision tree to create a classifier for the grouped dataset.

Algorithm:

1. For every hour of data:
 - a. Create a decision tree from the minute data in the hour
 - b. Create a set of average values for the hour
 - c. Classify the average dataset using the decision tree, in order to provide a classification for the hour

Graph Data Mining

This method was used to group minute-by-minute datasets into hourly datasets and create a model based on these. The approach is as follows:

1. Calculate the frequency of all the activity levels within the **minute** datasets.
2. Calculate an average value for each activity per hour, so that there is a value for each hour of the dataset.
3. Plot a graph of average hour reading against the number of hours where the average is less than the average we are looking at (maximum y value on the graph would be the number of hours in the dataset minus 1).
4. This graph is used to define the values needed to reach the four activity levels classes. For example, if 40% of values are 'not active', then one must find the 40% mark on the y axis and then read the corresponding x axis. Any values between this x value and 0 would be classed as 'not active'. If the next class covered 10% of minute values, then the x values related to y values between 40% and 50% would be in this class, and so on.
5. Once complete, there is a set of values for each activity. This set of values defines the boundaries between the activity classes.
6. The values in the hourly datasets should be compared to these boundaries, to find a set of potential classes for the hourly dataset. The highest class from these is taken as the class of the hourly dataset.

To classify new hourly datasets against the model (boundaries), the following approach was used:

1. Combine new minute datasets into hours, as before.

2. Predict a classification for the new hourly datasets using the boundaries created using the old data.
3. Create a new model (set of boundaries) from a combination of the old and new data. This model should be more accurate than the previous model.
4. Use this new model to find a classification for each new hourly dataset. These classifications can be compared against the predictions to test the accuracy of the model.

Data Pre-processing

Missing Datasets

The same principles applied to missing datasets as with the decision tree mining method. Datasets for all activities other than heart rate had to be processed so that they matched the heart rate dataset.

User Interface Design

This section of the report describes the way the application looks and the reasons behind the choices made.

Colour Usage

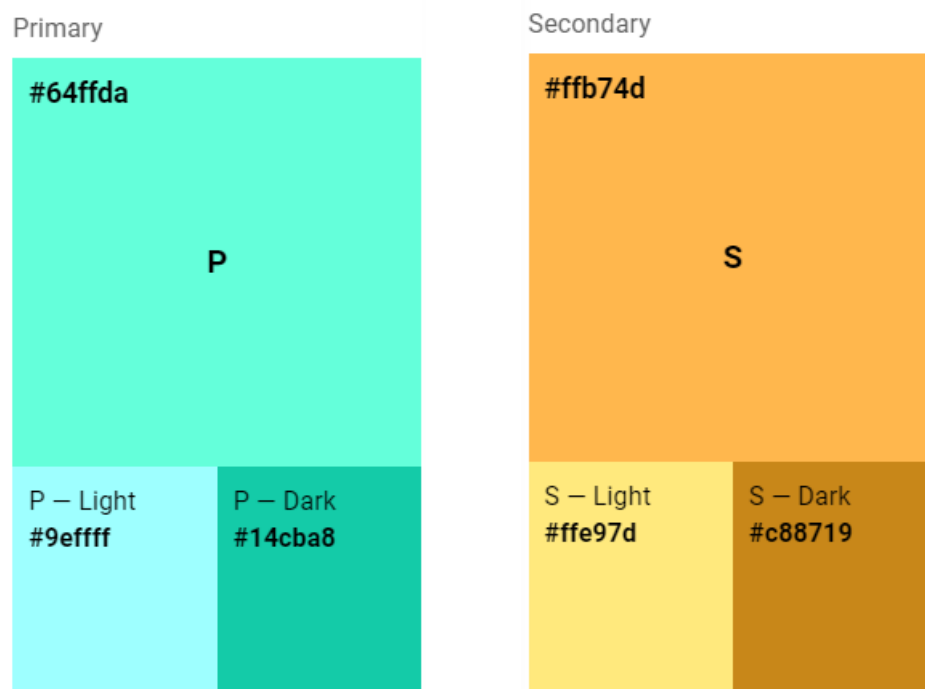


Figure 2 Colour scheme

The figure above shows the main colours used in the application. Material Design's colour tool [15] was used to create the colour scheme. This scheme can be found at <https://material.io/tools/color/#!/?view.left=0&view.right=0&primary.color=64FFDA&secondary.color=FFB74D>. The primary colour is used for the navigation bar, the light primary colour is used for buttons, and the dark primary colour is used for the Android status bar and icons throughout the application. The secondary colour is used for the profile icon and all graphs in the application.

The 'Accessibility' tab on the colour tool website showed that each of the six colours above are appropriate to have black text on them.

Screen Designs

Profile Page

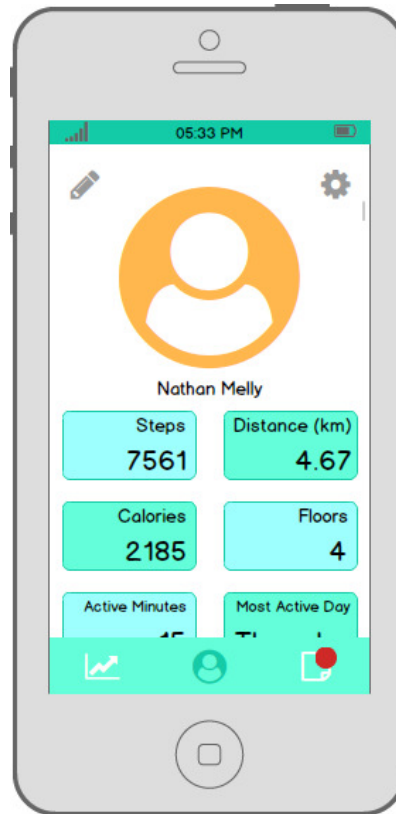


Figure 3 Profile Page Design

Notable Features

Feature	Description and Justification
Profile icon	This is to inform the user that they are on the profile page.
Profile value cards	These cards provide a summary of data to the user. They exist in pairs on each row and are centrally aligned so that the user's eye progresses down the screen, as opposed to any other direction. The font size of the value on each card is larger than the title of the card so that the user's eye is drawn to the value.
Navigation bar	This is the main navigation bar of the application. This allows the user to switch between the Profile, Exercises, and Alerts tabs. This is on every page of the application and so it uses the primary application colour to ensure that this main colour is on every page of the application. When the user is currently in one of the three sections of the application, the corresponding icon on the navigation bar is dark, so that it is clear to the user where they are in the application.
Edit button	This button navigates the user to an 'Edit Profile' page. It is not a primary function of the application, so it is coloured grey to ensure that it does

	not attract attention, however the grey is dark to create enough contrast with the background so that the button is easy to find, should the user need it.
Settings button	This button navigates the user to the 'Settings' page. It is not a primary function of the application, so it is coloured grey to ensure that it does not attract attention, however the grey is dark enough that the button is easy to find, should the user need it. The icon is a gear, which is commonly used to indicate settings within an application, so the functionality of the button is clear to the user.

Activity Page

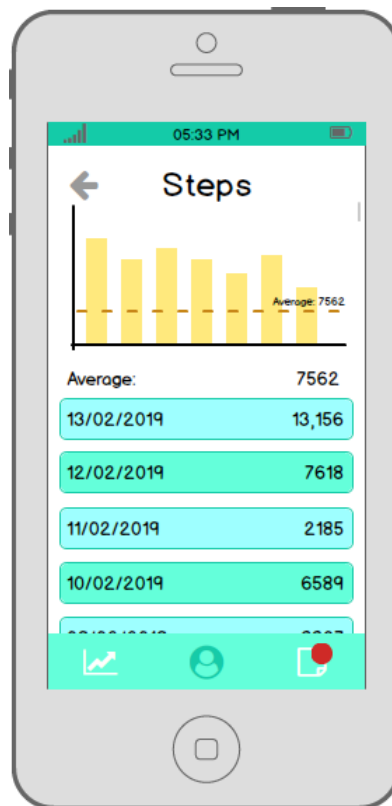


Figure 4 Activity Page Design

Notable Features

Feature	Description and Justification
Title	This tells the user what activity they are currently viewing.
Graph	The graph shows how the user data changes over time. The horizontal line shows the average value of the activity for the user, allowing easy comparison between this and their daily values.
Back button	This button returns the user to the Profile page. The arrow is a common symbol for a back button, so the function of this button is obvious to the user. The button is grey because the button's function is not a key function of the application, so it does not need to catch the user's eye.

List	This lists the date and value for each of the data points relating to the activity. This list is ordered in reverse-chronological order so that the most recent dates are at the top of the list. This is because the user is likely to be most interested in their recent activity, so this should be the first thing that they see.
------	---

Exercises Page

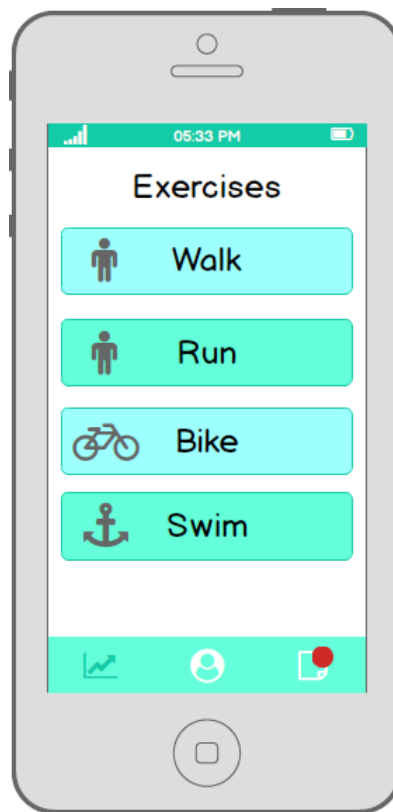


Figure 5 Exercises Page Design

Notable Features

Feature	Description and Justification
Title	This tells the user what page they are currently viewing.
Exercise Cards	These cards show the user what exercise types are available for them to view. They are large and easy to press. Their simple style means that it is clear to the user that pressing on any of the cards will allow them to view more detail about the selected exercise type.

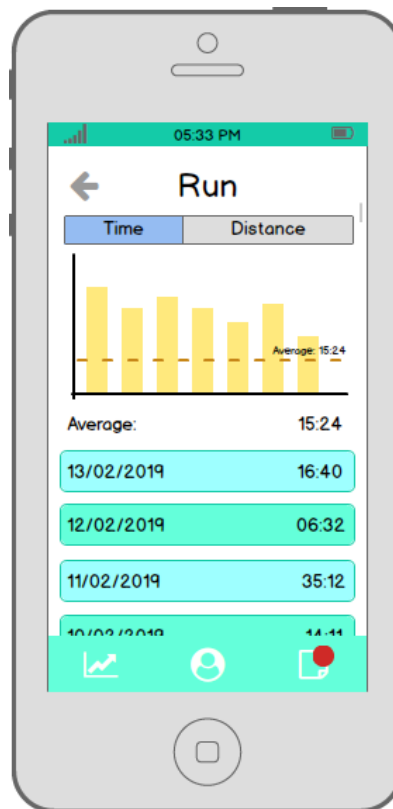


Figure 6 Exercise Page Design

Notable Features

Feature	Description and Justification
Title	This tells the user what exercise type they are currently viewing.
Graph	The graph shows how the user data changes over time. The horizontal line shows the average value of the chosen data category for the user, allowing easy comparison between this and their other sessions.
Back button	This button returns the user to the Exercises page. The arrow is a common symbol for a back button, so the function of this button is obvious to the user. The button is grey because the button's function is not a key function of the application, so it does not need to catch the user's eye.
List	This lists the date and value for each of the data points relating to the exercise type. This list is ordered in reverse-chronological order so that the most recent dates are at the top of the list. This is because the user is likely to be most interested in their recent exercise sessions, so this should be the first thing that they see.

Alerts Page

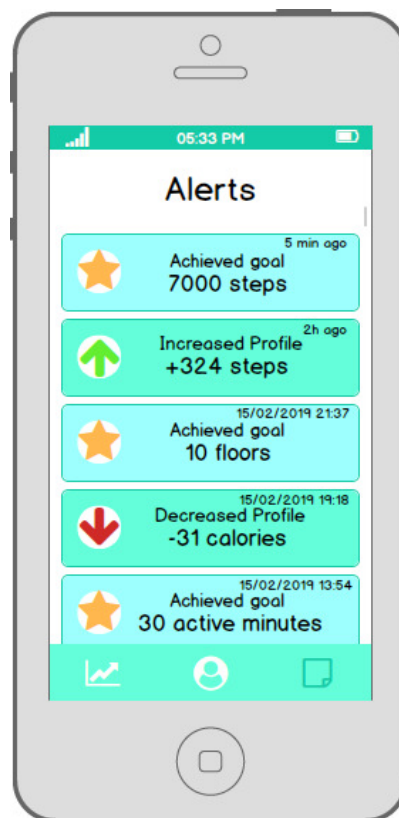


Figure 7 Alerts Page Design

Notable Features

Feature	Description and Justification
Title	This tells the user what page they are currently viewing.
Cards	<p>The cards show the user information about their behaviour. The cards can show the user how their profile has changed as new data has been imported into the application. The cards could also show whether the user has achieved certain goals they aimed to meet.</p> <p>The icons on the cards are used so that the user can tell what kind of information is on the card at a glance.</p>

Heuristic Evaluation

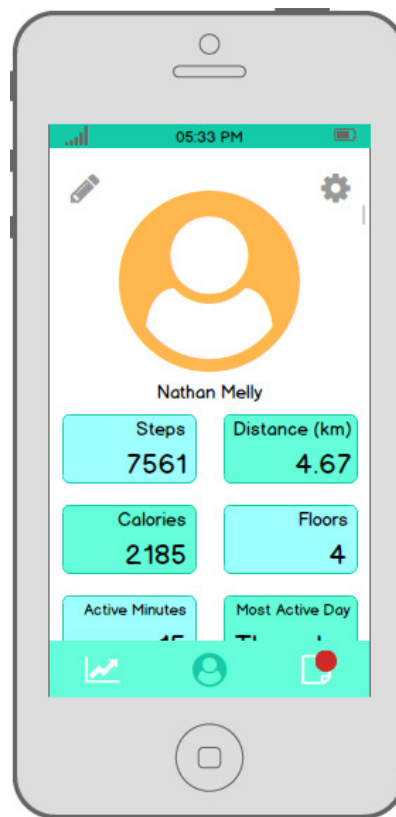
One of the main aims of this project was to create an application that is intuitive and usable. This section of the report is an evaluation of the screens against Nielsen's '10 Usability Heuristics for User Interface Design' [16]. The heuristics are as follows:

1. Visibility of system status
2. Match between system and the real world
3. User control and freedom
4. Consistency and standards
5. Error prevention
6. Recognition rather than recall

7. Flexibility and efficiency of use
8. Aesthetic and minimalist design
9. Help users recognise, diagnose, and recover from errors
10. Help and documentation

Each screen will be evaluated on how well or poorly the heuristic has been used, if at all.

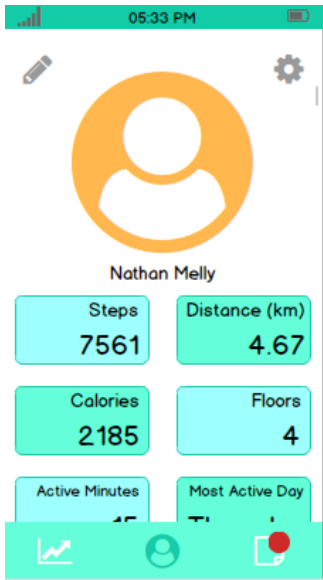
Profile Page

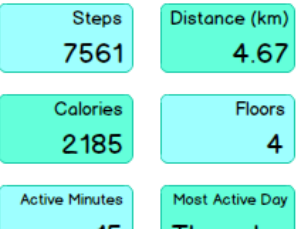


Principle	Comments
1.	Good: <ul style="list-style-type: none"> The navigation bar clearly shows which section of the application the user is in. The red icon on navigation bar alerts the user to the fact that there are new alerts in the alerts tab.
2.	Good: <ul style="list-style-type: none"> A unit is displayed for all values that require one. This will be familiar to the user and aid in their understanding of the value. The phrasing on the cards aligns with words used by Fitbit, so these should be familiar to the user.
3.	Good: <ul style="list-style-type: none"> The user is free to navigate to whichever part of the application they wish.
4.	Good: <ul style="list-style-type: none"> The button for the 'edit profile' page is a pencil, which is commonly associated with editing information in applications and websites. The button for the settings page is a gear symbol, which is commonly associated with settings, so this would be familiar to the user. Bad:

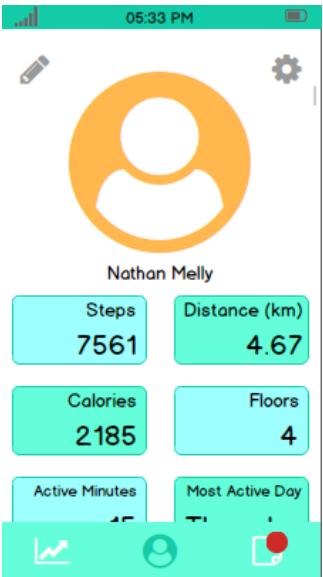
	<ul style="list-style-type: none"> • The functionality of the orange profile icon is unclear. • The cards are different colours, which could cause the user to think that they have different functionality.
5.	
6.	Good: <ul style="list-style-type: none"> • A range of actions are clearly available to the user in the form of buttons. • The cards have a title which gives the user an indication of the information they would find if they pressed the card.
7.	Bad: <ul style="list-style-type: none"> • There is nothing to suggest that the user must pull the screen down to refresh the profile.
8.	Good: <ul style="list-style-type: none"> • The page is well-structured and symmetrical. • The cards contain the minimum amount of information possible to fulfil their task, reducing clutter on the screen, and making important information easily visible. Bad: <ul style="list-style-type: none"> • The orange icon takes up almost half the screen. • The layout of the cards makes it unclear where the user should be looking.
9.	
10.	

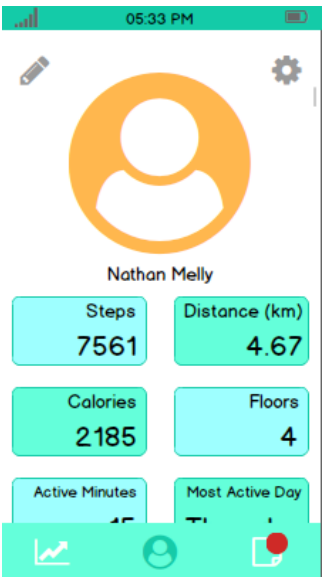
Problem: The purpose of the orange profile icon is unclear	
Heuristic Violated	4
Severity	2
Description	The orange icon takes up a large portion of the page, and it coloured in a way that draws attention to it, however the only functionality it has is that it shows that the user is on the profile page.
Alternative Solutions	Replace the icon with a page title that reads 'Profile'. Replace the icon with a profile picture (this could be taken from the user's Fitbit profile).

Evidence	
----------	---

Problem: The cards are different colours	
Heuristic Violated	4
Severity	3
Description	The cards are different colours, which could cause the user to think that they have different functionality.
Alternative Solutions	Make all cards the same colour.
Evidence	

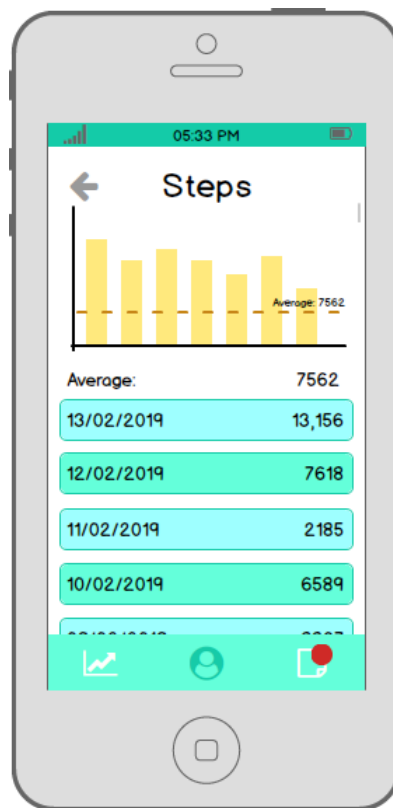
Problem: Refresh action unclear	
Heuristic Violated	7
Severity	4
Description	There is nothing to suggest that the user must pull the screen down to refresh the profile. This may leave the user confused about how to complete this vital task.
Alternative Solutions	<p>Add some text to tell the user to pull the screen down.</p> <p>Use the profile icon's colour to indicate when data has been loaded. This would also solve the first problem.</p>

Evidence	
----------	---

Problem: The profile icon is too large	
Heuristic Violated	8
Severity	2
Description	The orange icon takes up almost half the screen. The size is unnecessary and causes space on the page to be wasted.
Alternative Solutions	Reduce the size of the icon.
Evidence	

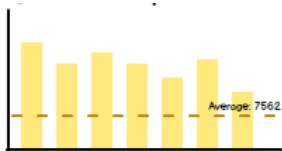
Problem: Card layout unclear	
Heuristic Violated	8
Severity	3

Description	Having two cards per row makes it unclear where the user should be looking.
Alternative Solutions	Put only one card on each row to ensure that the user's eye moves down the page.
Evidence	<p>The evidence shows a grid of six activity tracking cards arranged in three rows and two columns. The cards are:</p> <ul style="list-style-type: none"> Row 1: Steps (7561) and Distance (km) (4.67) Row 2: Calories (2185) and Floors (4) Row 3: Active Minutes (15) and Most Active Day (Tuesday)

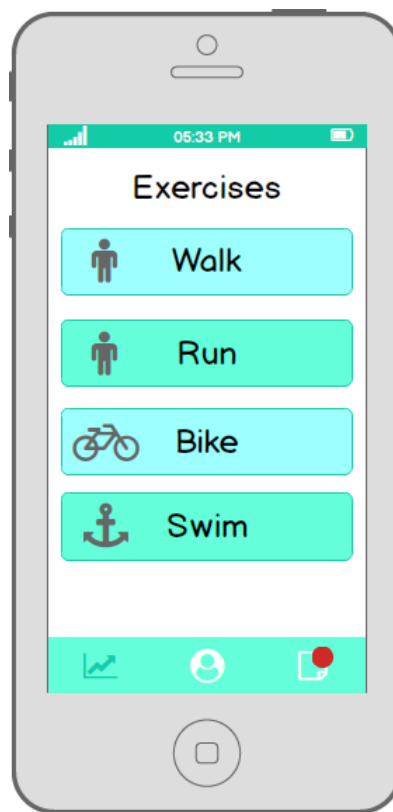


Principle	Comments
1.	Good: <ul style="list-style-type: none"> The title at the top of the screen is clear and tells the user what data they are looking at.
2.	Good: <ul style="list-style-type: none"> The date is in a format that is easy to understand. The information appears in a logical order (newest to oldest)
3.	Good: <ul style="list-style-type: none"> There is a back button in the top left corner so that the user can leave this page at any time. The user can press any of the buttons on the main navigation bar in order to return to one of the three main application pages.
4.	Good: <ul style="list-style-type: none"> The title matches the title of the corresponding card on the profile page. Bad: <ul style="list-style-type: none"> The cards are different colours, which could cause the user to think that they mean different things.
5.	
6.	Good:

	<ul style="list-style-type: none"> The title of the page reminds the user which card they pressed on the profile page The functionality of the back button in the top left is clear to the user as they would be familiar with pressing an arrow to go back. <p>Bad:</p> <ul style="list-style-type: none"> As there are no labels on the graph, the user is required to compare the graph to the list to find out which point on the graph corresponds to which value in the list.
7.	<p>Good:</p> <ul style="list-style-type: none"> As the main navigation bar is still present, the user can access the exercises tab and alerts tab directly from here, instead of having to go back to the profile page first.
8.	<p>Good:</p> <ul style="list-style-type: none"> There are only two main sections on the page, and there is nothing that does not need to be here.
9.	
10.	





Problem: No labels on graph	
Heuristic Violated	6
Severity	3
Description	There are no labels on the graph. This means that the user is required to compare the graph to the list to find out which point on the graph corresponds to which value.
Alternative Solutions	<ul style="list-style-type: none"> Add labels. Display the value and date when the user presses on the bar. This is preferred as the value will only be displayed temporarily, meaning that the minimalist design of the screen remains intact.
Evidence	

The problem of cards being different colours and potential solutions were described in the Profile Page evaluation.

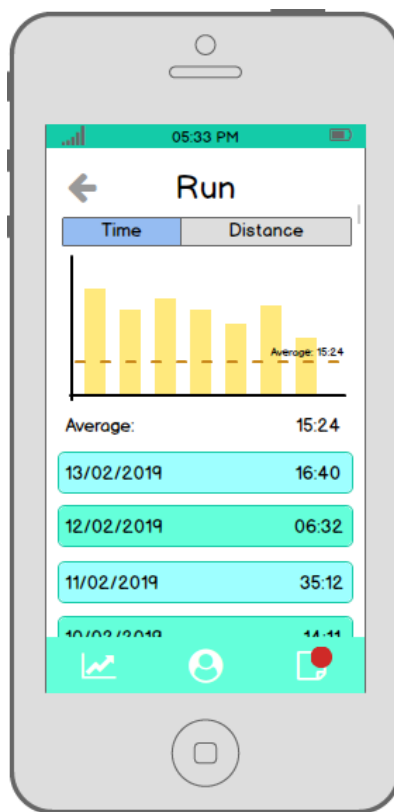


Principle	Comments
1.	Good: <ul style="list-style-type: none"> The title at the top of the screen is clear and tells the user which section of the application they are in.
2.	Good: <ul style="list-style-type: none"> The icons on the cards attempt to make the meaning of the cards clearer. Bad: <ul style="list-style-type: none"> The icons on the cards do not fit the title particularly well (although this is because of a limited range of icons on Balsamiq).
3.	Good: <ul style="list-style-type: none"> The user can press any of the buttons on the main navigation bar in order to go to any of the three main application pages. The user can press any of the cards to view exercise data.
4.	Bad: <ul style="list-style-type: none"> The cards are different colours, which could cause the user to think that they mean different things.
5.	Good:

	<ul style="list-style-type: none"> There is nothing on this page that could lead to an erroneous state.
6.	
7.	
8.	Good: <ul style="list-style-type: none"> There are only cards on this page, so the design is simple and clear.
9.	
10.	

Problem: The icons do not match the titles well	
Heuristic Violated	2
Severity	4
Description	The icons do not match the titles of the cards particularly well, so they are not fulfilling their purpose of making the meaning of the cards clearer.
Alternative Solutions	Change the icons.
Evidence	 Walk  Run  Bike  Swim

The problem of cards being different colours and potential solutions were described in the Profile Page evaluation.

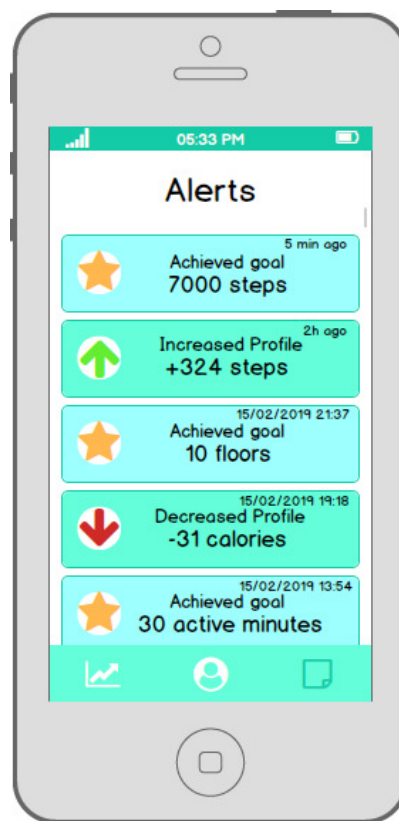


Principle	Comments
1.	Good: <ul style="list-style-type: none"> The title at the top of the screen is clear and tells the user what data they are looking at.
2.	Good: <ul style="list-style-type: none"> The date is in a format that is easy to understand. The information appears in a logical order (newest to oldest)
3.	Good: <ul style="list-style-type: none"> There is a back button in the top left corner so that the user can leave this page at any time. The user can press any of the buttons on the main navigation bar in order to return to one of the three main application pages.
4.	Good: <ul style="list-style-type: none"> The title matches the title of the corresponding card on the exercises page. Bad:

	<ul style="list-style-type: none"> The cards are different colours, which could cause the user to think that they mean different things.
5.	
6.	<p>Good:</p> <ul style="list-style-type: none"> The title of the page reminds the user which card they pressed on the exercises page The functionality of the back button in the top left is clear to the user as they would be familiar with pressing an arrow to go back. <p>Bad:</p> <ul style="list-style-type: none"> As there are no labels on the graph, the user is required to compare the graph to the list to find out which point on the graph corresponds to which value in the list.
7.	<p>Good:</p> <ul style="list-style-type: none"> As the main navigation bar is still present, the user can access the exercises tab and alerts tab directly from here, instead of having to go back to the profile page first.
8.	<p>Good:</p> <ul style="list-style-type: none"> There are only two main sections on the page, and there is nothing that does not need to be here.
9.	
10.	

The problem of cards being different colours and potential solutions were described in the Profile Page evaluation.

The problem concerning the lack of labels on the graph has been discussed in the Activity Page evaluation.



Principle	Comments
1.	Good: <ul style="list-style-type: none"> The navigation bar clearly shows which section of the application the user is in. There is no red circle on the Alerts tab icon anymore as the tab has been opened.
2.	Good: <ul style="list-style-type: none"> The language used on the cards clearly tells the user how their profile has changed. The date is in a common format that is familiar to the user. The time is in 24hr format, which removes confusion as to whether the time displayed is morning or afternoon. Green is used for profile improvements (good) and red is used for profile deteriorations (bad).
3.	Good: <ul style="list-style-type: none"> The user can press any of the buttons on the main navigation bar in order to go to any of the three main application pages.
4.	Bad:

	<ul style="list-style-type: none"> The cards are different colours, which could cause the user to think that some are more important than others.
5.	Good: <ul style="list-style-type: none"> There is nothing on this page that could lead to an erroneous state.
6.	Good: <ul style="list-style-type: none"> The words used on the cards (such as 'steps') match those used in other areas of the system.
7.	
8.	Good: <ul style="list-style-type: none"> There is nothing on this page that does not need to be here. The cards are arranged in a way that makes the user's eye move down the page in a logical way.
9.	
10.	

The problem of cards being different colours and potential solutions were described in the Profile Page evaluation.

Implementation

Software Architecture Diagram

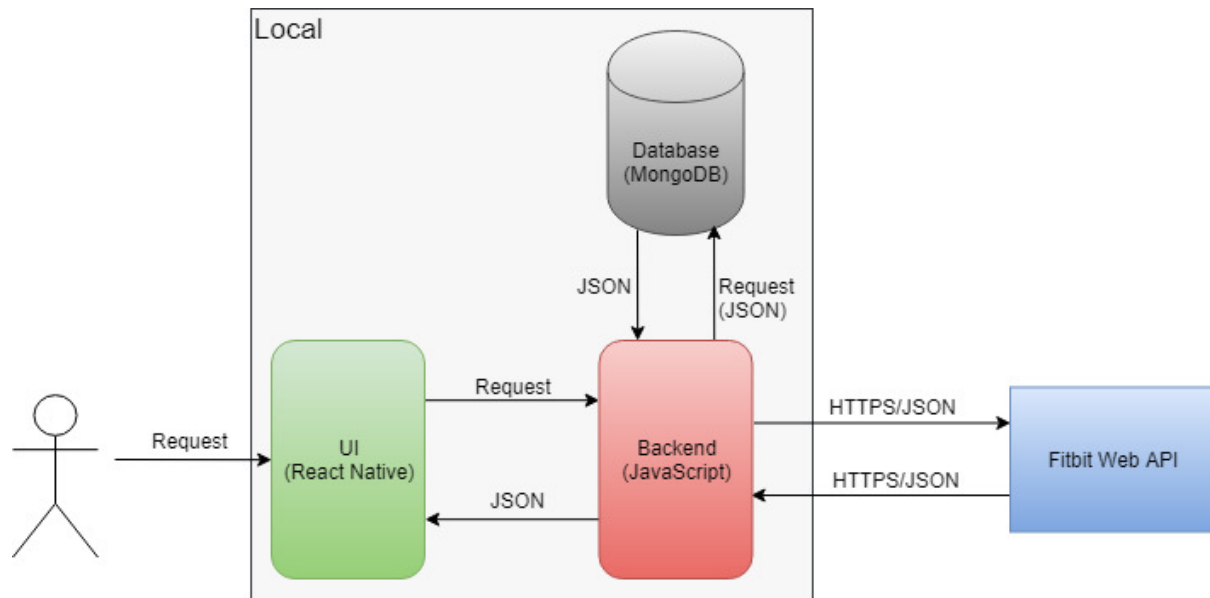


Figure 8 Software Architecture Diagram

The software architecture diagram above shows the technology used for each part of the application, as well as how these parts interact with each other.

The user interacts with the user interface, written in React Native. The UI requests some data from the backend, written in JavaScript. The backend interacts with the Fitbit Web API via a HTTP request when requesting user data, before receiving the data in JSON format and storing it in the database. The raw data is then retrieved from the database for processing before being passed to the user interface to be presented to the user.

Data Flow Diagram

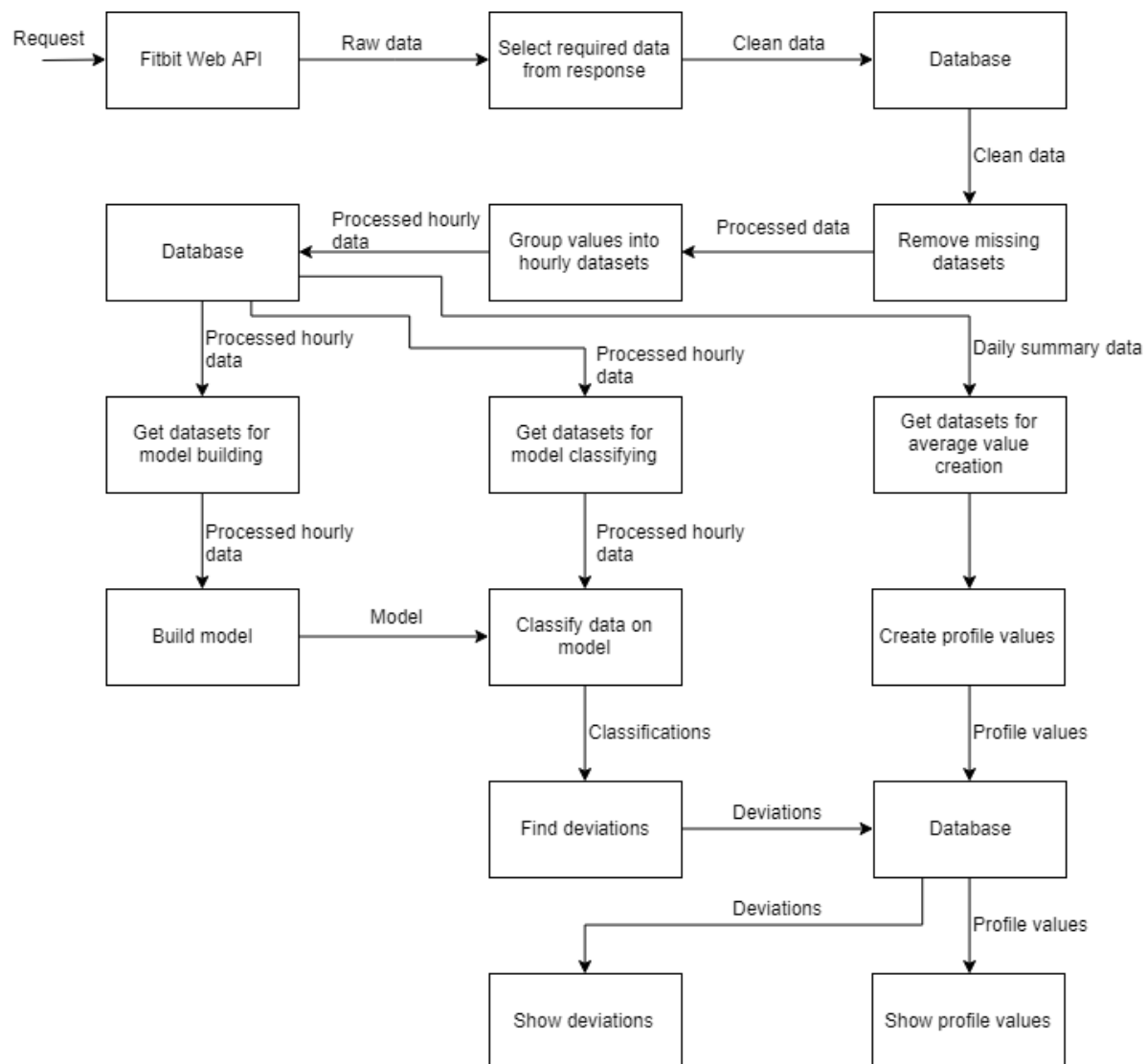


Figure 9 Diagram of data flow within system

Figure 9 shows how data moves within the system from when it is requested until it is displayed in the application. First, a request is sent to the Fitbit Web API, and the relevant values are taken from the response and stored in the database. Next, the minute data is processed into hours. Before this can happen, 'missing' (auto-populated) values must be removed as they correspond to times when the user was not wearing their Fitbit device. After this, the minute values are aggregated into hourly average datasets in order to build the models and classify the data. The newly imported data is classified on a model created on the existing data, and deviations are created from these and stored in the database. Daily total values for data types such as steps etc. are then used to create the profile values, which are stored in the database. Finally, the deviations and profile values are retrieved by and displayed in the Alerts tab and Profile tab, respectively.

Class Diagram

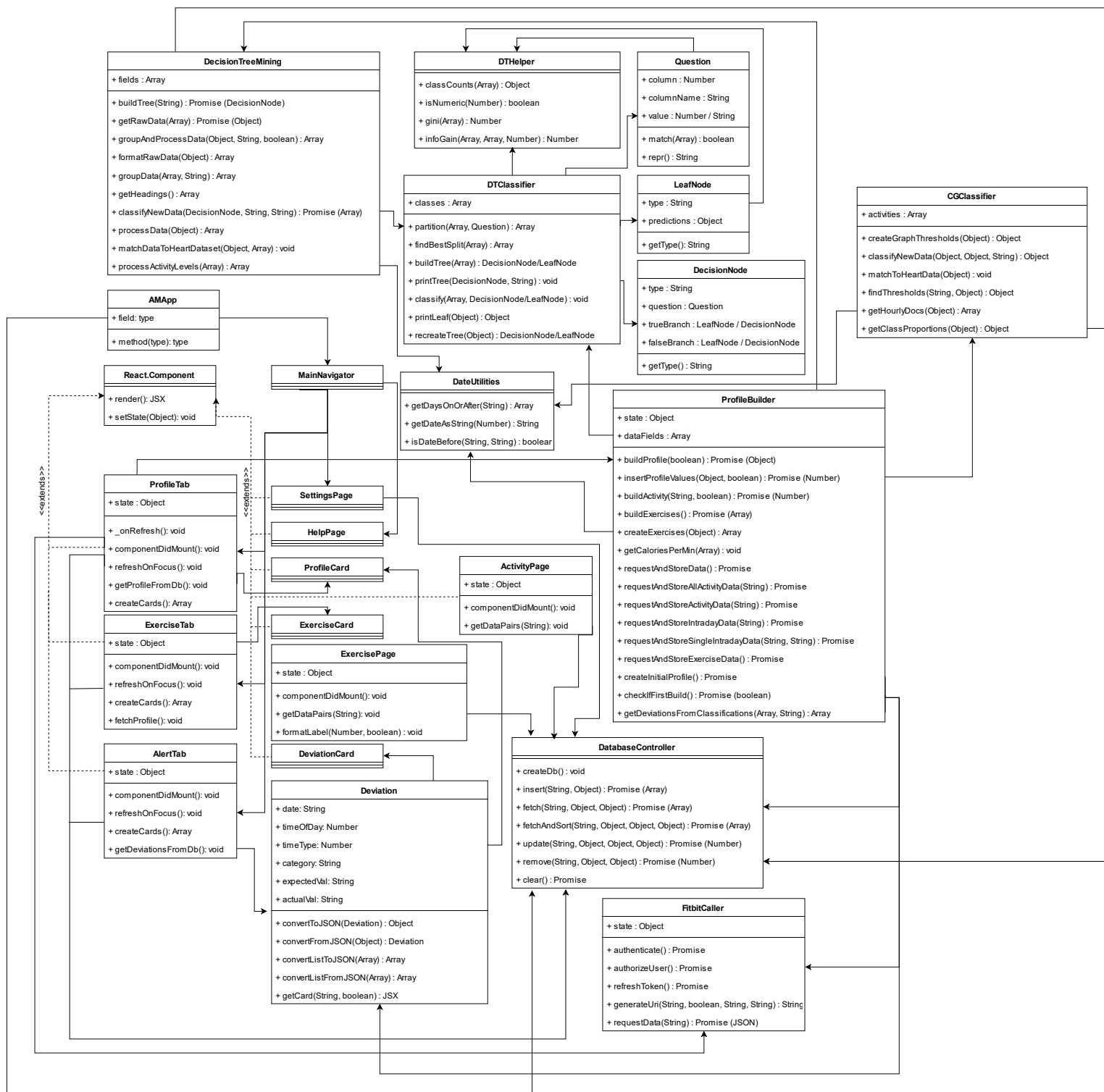


Figure 10 Class Diagram

The diagram above shows how each class within the application interacts with each other. The 'React.Component' class on the left side of the diagram is from the React package that comes with React Native. Only the fields and methods from this class that have been used in the application have been added to the class diagram to prevent unnecessary information being added.

UI Code

Main Tabs

The application is split into three main areas, each consisting of a tab that is accessible from the main navigation bar within the application. These areas/tabs are the Profile, Exercises, and Alerts tabs.

The files for these tabs contain an object called 'cardMap', which contains information about how certain predefined cards should be displayed (such as what icon to use, and what unit to use if the card displays a value).

Profile Tab

The largest file out of the three tabs is ProfileTab.js. This page within the application has a 'RefreshControl' object that allows the user to pull the screen down to refresh. This calls the `_onRefresh` method, which in turn calls methods from other files that request new data from Fitbit and build the profile.

This file also contains a method called 'getProfileFromDb' which is used to load the profile from the database (if it exists) when the application is opened.

Information stored in the profile is displayed on the screen in the form of cards. These cards are created from this information in the 'createCards' method.

This file contains a 'render' method which is inherited from `React.Component`. This returns markup that is used to display items on the page.

Exercises Tab

When this tab gains focus, exercise data is loaded from the database. Exercise types are gleaned from this data, and a card is displayed for each one. Pressing on these cards will take the user to a page that shows the data for that specific exercise type.

This page is designed so that a user can only attempt to view data for exercises that they have completed. Initially, an option for common exercises such as 'Run', 'Walk', and 'Sport' were going to be included, however the solution that only shows cards for exercise types that the user has done is better as the application is more tailored to the user, and there is no way that the user can view a page containing no data.

Alerts Tab

On focus, this page retrieves data from the 'deviations' datastore. This data is split into three categories: activity level deviations, profile deviations, and profile updates. Activity level deviations are deviations that were created based on classifications using the decision tree data mining model. Profile deviations are created when the user has not matched a value in their profile for a given activity. Profile updates tell the user how values in their profile have changed since the last profile build.

This file also contains a 'createCards' method, which is called in the render method, just like in the other two 'tab' files. Unlike in the other two files, this method also adds a picker to the page, that allows the user to filter what they see on the page between the three categories listed earlier, and 'all'.

Activity and Exercise Pages

The activity page is where the user is taken when they press any of the cards in the activity section of the profile page. On navigation, the activity page accepts parameters consisting of the title of the page (usually the activity name correctly formatted with capital letters), the 'key' (usually just the activity name. This dictates which data store the DatabaseController file should query when providing data for the page), and a value (which is the value related to the activity that shows on the card on the profile page for the user). These parameters allow the page to show different data, meaning that it is necessary to only create this one dynamic page for all activities, instead of one for each.

The page consists of a graph showing how the user's data has changed over time, followed by a list of values related to the current activity. A key method in this file is 'getDataPairs'. This uses the key parameter that was passed to the page to retrieve data from the relevant part of the database, sorted by date in reverse chronological order, to be displayed in the list. This method returns pairs consisting of a date and the value for the activity on that date. It is ordered by using the 'fetchAndSort' method from the DatabaseController.js file and specifying sort conditions, so that the most recent day's data is displayed first in the list.

The exercise page is like the activity page in that it can show data for various exercises and is not specific to any exercise. The page has an almost identical layout to the activity page, and the only notable difference is that there is a picker at the top of the exercise page. The exercise page is accessed by pressing any of the cards on the exercises tab. The exercises page displays data from sessions of the selected exercise type. As each exercise session has values relating to the duration and calories burned during the exercise, the picker at the top of the screen allows the user to choose which of these values they wish to see displayed in the graph and list on this page. For the exercise types 'Treadmill' and 'Run', the user can also choose to view the distance covered during exercise sessions.

Cards

Cards appear in three places within the application: the profile tab, the exercises tab, and the alerts tab. There are three types of card ('ProfileCard', 'DeviationCard', and 'ProfileUpdateCard') and their implementation is simple. Each card simply extends React.Component and has a render method that returns JSX code that defines how the card should be rendered. The information to be displayed by the card is passed in via 'props', which is standard in React Native. These props are combined with the JSX markup to display the cards that are seen in the application.

Other Pages

Help Page

This page is a simple page that contains only a render method, used to display text that provides guidance to the user on aspects of the application.

Settings Page

The settings page contains two buttons, with very simple tasks. The first button clears all the data within the database. The other button initiates a rebuild of the user profile.

Navigation

Navigation.js

This is where the navigation for the application is defined. The main 'tab' navigation bar at the bottom of the application is split into three sections, which correspond to the three tabs within the application. Within the navigation file, each of these buttons corresponds to a separate 'stack' navigator for each of the three application parts. The 'ProfileStack' stack navigator contains all the pages that are reachable from the profile tab. The 'ExerciseStack' stack navigator contains all the pages that are reachable from the exercises tab. The 'AlertStack' stack navigator contains all the pages that are reachable from the alerts tab. A diagram showing how the navigators are combined is shown below.

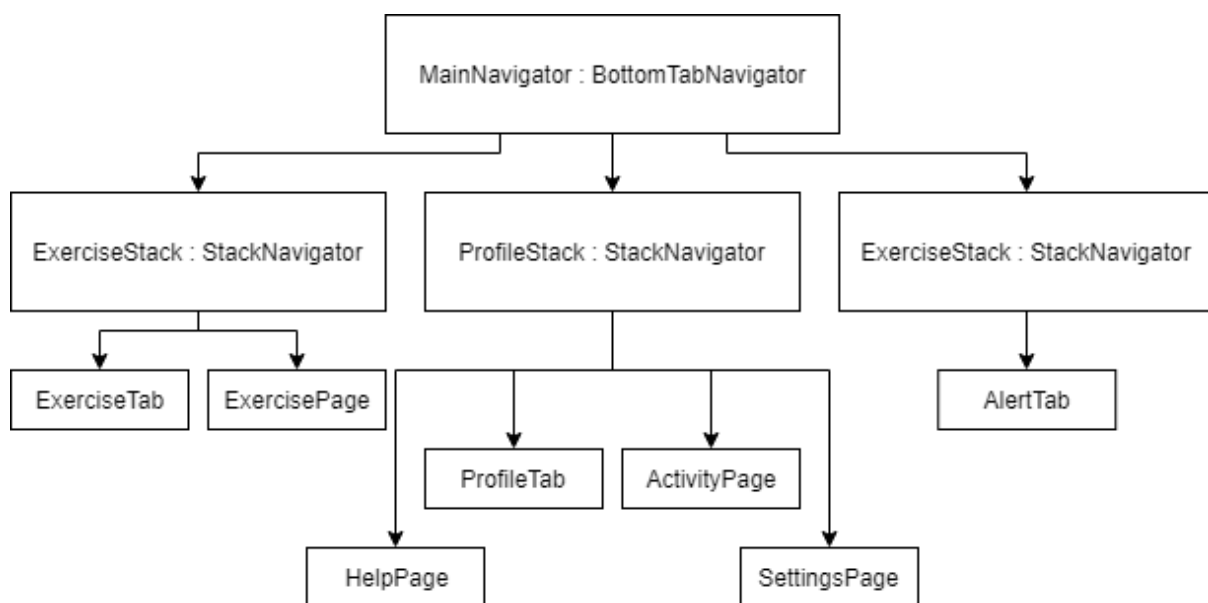


Figure 11 Navigation

Profile Building Code

ProfileBuilder.js

This file is where all profile-related activities happen, from requesting data, to calculating the values that are seen on the profile page, to building a model based on the user and calculating deviations. The method that is called to start all the above is 'buildProfile'.

The first thing that happens in this method is the creation of an object called 'newProfileValues'. This object stores all the information that will be added to the profile in the 'profile' collection of the database.

The method then checks if this is the first time a profile has been built. If so, an entry is created in the database within the call to 'ProfileBuilder.createInitialProfile()'. The profile is retrieved from the database, ready for updating later. Figure 12 below shows the profile being readied.


```

//Check if a profile exists already
let isFirstBuild = await ProfileBuilder.checkIfFirstBuild();

//Create an initial profile object if this is the first build
if (isFirstBuild) {
    await ProfileBuilder.createInitialProfile();
}

//Get the profile object from the db
let profiles = await DatabaseController.fetch("profile", {}, {});
let profile = profiles[0];

```

Figure 12 Getting a Profile ready to update

The code then requests new data from Fitbit if the user is refreshing the profile. Figure 13 below shows this call (note, the 'rebuild' variable is true if the user is only rebuilding the profile from the existing data, and false if the profile is being refreshed, thus new data is required). There are also calls to methods here that process the minute data into hourly datasets to be used by the two classifiers, however this code will be discussed in the data mining section.

```

if (!rebuild) {
    //Requests the data from Fitbit and stores it
    await ProfileBuilder.requestAndStoreData();

    //Convert to hourly datasets in correct format for DT
    await ProfileBuilder.processToDTHours();

    //Convert to hourly datasets in correct format for CG
    await ProfileBuilder.processToCGHours();
}

```

Figure 13 Requesting data

The next snippet of code iterates through all the activity types and creates the profile value for each type. These values are added to the 'newProfileValues' object to be stored in the database. Figure 14 and Figure 15 show the creation of the values in the activity section of the profile page. The method in Figure 15 creates an average of the values for each activity type, to be shown on the profile page.

```
//Iterate over the types of data we defined above.
for (const dataField of ProfileBuilder.dataFields) {

    //Build the profile value for the relevant activity
    let value;
    if (dataField !== "distance") {
        value = await ProfileBuilder.buildActivity(dataField, true);
    } else {
        value = await ProfileBuilder.buildActivity(dataField, false);
    }

    //Add the new value into the object to be stored later.
    newProfileValues.activities[dataField] = value;
}
```

Figure 14 Creating profile activity values

```

static buildActivity(activity, integer) {
  return new Promise(function(resolve, reject) {
    DatabaseController.fetch(activity, {}, {}).then(function(docs) {
      var totalValue = 0;
      const totalDocs = docs.length;
      for (var i = 0; i < totalDocs; i++) {
        totalValue += parseInt(docs[i].total);
      }
      let output = totalValue / totalDocs;
      if (integer) {
        resolve(Math.round(output));
      }
      else {
        resolve(parseFloat(output.toFixed(2)));
      }
    }).catch(function(error){
      reject(error);
    });
  });
}

```

Figure 15 Building profile activity values

After this, three variables (dtHourClassifications, graphHourPredictions, and graphHourClassifications) are initialised to null. These are variables to store predictions and classifications from both classifiers during testing.

Next in 'buildProfile', a call to 'ProfileBuilder.buildExercises' performs a similar function to 'buildActivity' in the figures above. This method and those that it calls provide the values that are shown in the cards in the 'Exercises' section of the profile page.

The next section of the code is where the decision tree data mining takes place. First, the dates required for classification are found, then data for building the tree and data to be classified are retrieved from the database. Figure 16 shows this and the creation of the decision tree.

```
let dtDaysToBuild = await DatabaseController.fetch("dtDatasets", { date: {$nin:
datesToClassify}});
```

```
let dtDaysToClassify = await DatabaseController.fetch("dtDatasets", { date: {$in:
datesToClassify}});
```

```
let dtClassifier = new DTClassifier(ProfileBuilder.getHeadings());
```

```
let buildDatasets = [];
```

```
for (const day of dtDaysToBuild) {
```

```
    let dayData = day.datasets;
```

```
    for (const dataset of dayData) {
```

```
        buildDatasets.push(dataset);
```

```
    }
```

```
}
```

```
//Build a tree using the classifier object and the dataset
```

```
let tree = dtClassifier.buildTree(buildDatasets);
```

Figure 16 Decision Tree creation

Next is some code that is used to classify new data and was originally in DecisionTreeMining.js, however due to a problem with the database (described in the problems section), this code was moved to this file. This code is discussed in the Data Mining part of this section of the report. The output of this code is an array of classifications, which is then passed into another method to create an array of deviations. Figure 17 and Figure 18 show this.

```
let deviations = ProfileBuilder.getDeviationsFromClassifications(dtClassifications, "hour");
```

```
if (deviations.length > 0) {
```

```
    await DatabaseController.insert("deviations", deviations);
```

```
}
```

Figure 17 Creation of deviations after classification

```

static getDeviationsFromClassifications(classifications, timeInterval) {
    let deviations = [];

    for (const classification of classifications) {
        let keys = Object.keys(classification.predicted);
        let predicted = keys[0];
        if (classification.actual !== predicted) {
            deviations.push(new Deviation(classification.date,
                classification.timeOfDay,
                timeInterval,
                "activity",
                predicted,
                classification.actual));
        }
    }
    return deviations;
}

```

Figure 18 Creation of deviations (detail)

The next piece of code is used to create and use the graph-based classifier. First, the datasets for building and classifying are retrieved, as with the decision tree classifier. For the graph classifier, a third dataset is created which is a combination of the two datasets (i.e. all data available in the database).

Next, a classifier is created using the build dataset, and the dataset to be classified is classified using it. The output of this is a prediction of the class of the new data, based on existing data.

Following this, the 'updateCGProportions' method updates the stored frequencies of each activity level (as this information is used during the creation of the graph-based classifier) to account for the new data. A new classifier is then created using the third dataset mentioned above and these updated class frequencies. The dataset containing the new data is classified again using this new classifier to give a final classification. If the classification for any dataset is different to its classification on the first classifier, then the prediction was incorrect. It should be noted that the classifications from the graph-based classifier are **not** used to create deviations within the application. This is because it is only necessary to use one classifier for this purpose, however if the graph-based classifier is more accurate in testing, then some future work should be to replace the decision tree classifications with the classifications from this classifier when creating deviations. Currently, this code is in the application for the purpose of comparison with the decision tree classifier.

```

    let datesToClassify = DateUtilities.getDaysOnOrAfter(lastBuildDate);

    let cgDaysToBuild = await DatabaseController.fetch("cgDatasets", { date: {$nin:
datesToClassify}, $not: {_id: "proportions"}});

    let cgDaysToClassify = await DatabaseController.fetch("cgDatasets", { date: {$in:
datesToClassify}});


    let allCGDays = [];
    for (const day of cgDaysToBuild) {
        allCGDays.push(day);
    }
    for (const day of cgDaysToClassify) {
        allCGDays.push(day);
    }


    let cgPredictClassifier = await ProfileBuilder.buildCGClassifier(cgDaysToBuild);
    let cgPredictions = ProfileBuilder.classifyCGData(cgDaysToClassify, cgPredictClassifier);


    await ProfileBuilder.updateCGProportions();


    let cgClassifyClassifier = await ProfileBuilder.buildCGClassifier(allCGDays);
    let cgClassifications = ProfileBuilder.classifyCGData(cgDaysToClassify, cgClassifyClassifier);

```

Figure 19 Creation and use of graph-based classifiers

Finally, there is a call to `ProfileBuilder.removeIntradayDatasets`. This method is used to remove all intraday (minute) data from the database. This is because of a storage limit that was discovered (see Problems below). The data has already been processed into hourly datasets and stored in that format by this point, so there is no need to keep the minute data in the database.

Deviation.js

This file contains a class that models a ‘deviation’. A deviation contains six fields, which are described earlier in the report. These fields are passed in to the class using the constructor. A key method in this class is ‘getCard’. This is a method that is called on an instance of ‘Deviation’ and returns JSX markup that is used to render a card on the screen. This method returns a ‘DeviationCard’ that is displayed on the Alerts tab. This method is also able to return

a 'ProfileUpdateCard'. This is a card that displays how a profile value has changed. This is technically not a deviation, but as the two cards are so similar, 'Deviation' objects are used within the application to display changes to the user profile.

The other methods in this file are called 'convertToJSON' and 'convertFromJSON'. These are necessary as 'Deviation' objects are stored as JSON in the MongoDB database. When they are retrieved, the 'convertFromJSON' method is used to recreate Deviation objects from the results of the database query.

The only other thing to note about this file is that there is a cardMap object, like the ProfileTab file. This is because the 'getCard' method returns a card, so cardMap is used to define how this card should look.

Problems

1. **Database Limit:** It was discovered very late in the project that there is a limit to the amount of data that can fit in the database. Every day of intraday data contains up to 1440 datasets, and up to 7 days are requested each time (this limit was imposed due to Fitbit request limits). These datasets were originally being stored in the application, and the grouping into hourly datasets was done during each profile refresh, until it was discovered during testing that only around 12 days of data would fit into the database. Upon further investigation, the 'react-native-local-mongodb' package was built on top of React Native's AsyncStorage storage mechanism (this was not mentioned clearly in the documentation), which has a limit of 6MB on Android devices. There is no mention of this 6MB limit in the React Native documentation as far as I can tell either, meaning that it was unlikely that I would discover this issue until I began testing the application with many days of data.

As a workaround to this issue, time was spent reworking major aspects of the application, namely the entire data mining process. The application now processes data into hourly datasets as soon as it is imported into the application, before storing these hourly datasets. The data mining is then carried out on these hourly datasets as it was previously, however certain methods required altering. Minute data that was imported during the refresh is then deleted at the end of the refresh flow to ensure that the space within the database is freed for future intraday data that requires processing. Because this rework was done so late into the project, many methods were copied from the data mining files into ProfileBuilder.js to be worked on independently of the actual data mining code (to ensure that the original code remained intact if this rework failed). These methods have been altered slightly, however there was not enough time to move them back into their original files. As a result, there are many lines of code near the bottom of ProfileBuilder.js that should exist elsewhere. As a result of this rework, I was able to get over 70 days of data into the application during testing (there was potential for more as the database did not fill), making it a success.

Data Mining Code

Due to a problem that was encountered with the database (see 'Problems' section in the 'Profile Building Code' section of the report), some of the data mining code was temporarily moved into ProfileBuilder.js to be worked on, however this code became permanent due to

a lack of time at the end of the project. The data mining code is still discussed in this section of the report, however.

Decision Tree Mining

The code for the decision tree data mining was adapted into JavaScript for this project from a Google tutorial [14]. The code was originally in Python and can be found at https://github.com/random-forests/tutorials/blob/master/decision_tree.ipynb. The adapted code can be found in the following files: 'DTClassifier.js', 'DTHelper.js', 'DTNodes.js', and 'Question.js'. There is a method in 'DTClassifier.js' that is not from this tutorial, called 'recreateTree' which was used to recreate the tree from the object that is retrieved from the database, however this is no longer used as a result of the data mining rework.

The file 'DecisionTreeMining.js' contains the methods that were originally used to process the data and group the minute datasets into hour and six-hour datasets for the decision trees. Processing is now done when data is first imported, as a result of the database issue mentioned earlier (**important:** processing now only processes the data into hourly datasets and not six-hour datasets as a result of the rework). This processing is done in ProfileBuilder.procesToDTHours. This method creates hourly datasets from the minute data. Whilst the code to process the data has been moved from DecisionTreeMining.js, this method still calls other methods from DecisionTreeMining.js.

One of the first things to happen in this method is the retrieval of minute data, which is then put into an object.

```
//Create object to hold all minutes data and heart data to match it to
let activityDoc = {};

//Iterate over all activity types
for (const activity of ProfileBuilder.dataFields) {
    activityDoc[activity] = await DatabaseController.fetchAndSort(activity, {intraday: {$exists: true}},
    {}, {date: 1});
}
```

Figure 20 Retrieval of intraday data

Next, two methods from DecisionTreeMining.js are called. The 'formatRawData' method is used to match the size of datasets for all activities to the heartrate datasets, to remove missing datasets, and format the data for every minute into an object containing the value for every activity for that minute, as well as the date and time of the minute. The 'groupData' method is used to group these minute datasets into hour datasets. This is done by simply iterating through each minute object and aggregating the values from the objects that have the same date and hour and putting these values into a new object for the hour. The hourly object also contains an array of minute datasets in the form [steps value, calories value, floors value, distance value, heart value, activity level], where activity level is the class.

Next in processToDTHours, hourly average datasets of the form [steps value, calories value, floors value, distance value, heart value, activity level] are created by dividing each value in the hourly object by the number of minute datasets in the hour. A decision tree is then

created using the minute datasets for that hour, and the average hourly dataset is classified on this tree. Figure 21 shows this process.

```

//Group the data
let grpData = DecisionTreeMining.groupData(data, "hour");

//Process the data
let procData = [];
let processedTimeData = {};

//For each day
for (var i = 0; i < grpData.length; i++) {
    let groupKeys = Object.keys(grpData[i]);
    for (var j = 0; j < groupKeys.length; j++) {
        let key = groupKeys[j];
        let hourData = grpData[i][key];

        //Divide by the number of minute datasets (there is no guarantee that there are 60)
        let numDatasets = hourData.datasets.length;
        let avgSteps = hourData.totalSteps / numDatasets;
        let avgCals = hourData.totalCalories / numDatasets;
        let avgFloors = hourData.totalFloors / numDatasets;
        let avgDist = hourData.totalDistance / numDatasets;
        let avgHeart = hourData.totalHeart / numDatasets;

        //Create average dataset to be classified
        let dataset = [avgSteps, avgCals, avgFloors, avgDist, avgHeart];

        let classifier = new DTClassifier(DecisionTreeMining.getHeadings());
        let tree = classifier.buildTree(hourData.datasets); //Create a tree from the minute datasets

        let outClasses = DTClassifier.classify(dataset, tree);
        let outClass = null;
        let outClassConfidence = 0;

        for (const potentialClass of Object.keys(outClasses)) { //Pick the most likely class
            if (outClasses[potentialClass] > outClassConfidence) {
                outClass = potentialClass;
                outClassConfidence = outClasses[potentialClass];
            }
        }

        dataset.push(outClass);
        procData.push(dataset);
    }
}

```

Figure 21 Processing of minute datasets into hourly datasets for decision tree

Finally, the hourly datasets are stored in the database under their date.

```
let docs = [];
for (const date of Object.keys(processedTimeData)) {
  let doc = {
    date: date
  }
  for (const timeOfDay of Object.keys(processedTimeData[date])) {
    if (typeof doc.datasets == "undefined") {
      doc.datasets = [processedTimeData[date][timeOfDay]];
    } else {
      doc.datasets.push(processedTimeData[date][timeOfDay]);
    }
  }
  docs.push(doc);
}

let docsRes = await DatabaseController.insert("dtDatasets", docs);
```

Figure 22 Decision tree hourly datasets stored in database

Decision tree classification is done in ProfileBuilder.buildProfile. This used to take place in DecisionTreeMining.js but was moved as a result of the rework. Figure 23 shows this classification taking place.

```

//Build a tree using the classifier object and the dataset
let tree = dtClassifier.buildTree(buildDatasets);

//Classify using decision tree
let classifications = [];
let correct = 0;

for (const day of dtDaysToClassify) {
  let dayData = day.datasets;
  let timeOfDay = 0;
  for (const dataset of dayData) {
    let actual = dataset[dataset.length - 1];
    let predicted = DTClassifier.printLeaf(DTClassifier.classify(dataset, tree));
    // let classification = `Actual: ${actual}; Predicted: ${JSON.stringify(predicted)}`;
    let classification = {
      date: day.date,
      timeOfDay: timeOfDay,
      actual: actual,
      predicted: predicted
    }
    classifications.push(classification);
    if (Object.keys(predicted).includes(actual)) {
      correct += 1;
    } else {
      console.log(dataset);
      console.log(classification);
    }
    timeOfDay++;
  }
}
return classifications;

```

Figure 23 Decision tree classification

Figure 24 shows an example of a decision tree classifier. The numbers in the prediction objects state how many data points in the dataset used to build the tree were given that class.

```

Is distance >= 0.0030793103847074612?
--> True:
    Predict {"lightly active":6}
--> False:
    Is calories >= 1.5433288137118022?
    --> True:
        Is heart >= 61.666666666666664?
        --> True:
            Predict {"not active":2}
        --> False:
            Predict {"lightly active":1}
    --> False:
        Predict {"not active":15}

```

Figure 24 Example of a decision tree classifier

Graph Mining

The original code for this data mining approach can be found in 'CumulativeGraphClassifier.js', however the code that is used now is at the end of ProfileBuilder.js as a result of the rework that was required. The classifier is effectively a set of boundaries between classes for each type of activity, as described in the 'Algorithms' section of this report.

This method starts similarly to the decision tree, with a method called processToCGHours. The first thing that happens within this method is the retrieval of the data from the database, and the processing of this data to remove values from when the user was not wearing their device. Next, the code finds the frequency of each class within the datasets and stores this information in the database if it does not already exist. Figure 25 shows this.

```

//Create object to hold all minutes data and heart data to match it to
let activityDoc = {};

//Iterate over all activity types
for (const activity of ProfileBuilder.dataFields) {
    activityDoc[activity] = await DatabaseController.fetchAndSort(activity, {intraday: {$exists: true}},
    {}, {date: 1});
}

//Remove invalid datasets
ProfileBuilder.matchToHeartData(activityDoc);

//CREATE PROPORTIONS
//Get the existing activity level proportions
let proportionsRes = await DatabaseController.fetch("cgDatasets", {_id: "proportions"}, {});

if (proportionsRes.length == 0) {
    //Get proportions of new data
    let newProportions = ProfileBuilder.getClassProportions(activityDoc);
    newProportions["_id"] = "proportions";
    await DatabaseController.insert("cgDatasets", newProportions);
}

```

Figure 25 Initial retrieval and processing of data for graph-based classifier

The end of the graph dataset processing method involves creating hourly datasets (the call to `ProfileBuilder.getHourlyDocs` does this) before adding these datasets to objects and inserting them into the database. Figure 26 shows this.

```

let docsToInsert = {};
//Insert new hourly docs into db for every data type
for (const activity of ProfileBuilder.dataFields) {
  let docs = ProfileBuilder.getHourlyDocs(activityDoc[activity]);
  for (const doc of docs) {
    doc.activity = activity
  }
  for (const doc of docs) {
    if (typeof docsToInsert[doc.date] == "undefined") {
      docsToInsert[doc.date] = [doc];
    } else {
      docsToInsert[doc.date].push(doc);
    }
  }
}

let formattedDocsToInsert = [];
for (const key of Object.keys(docsToInsert)) {
  let datasets = docsToInsert[key];
  let doc = {
    date: key,
    datasets: datasets
  }
  formattedDocsToInsert.push(doc);
}

let insertionRes = await DatabaseController.insert("cgDatasets", formattedDocsToInsert);

```

Figure 26 Creation, formatting and storage of hourly datasets

Figure 27 shows the datasets from when the user was not wearing their Fitbit device being removed.

```

static matchToHeartData(activityDoc) {

    //Get the heart data to match the active minute data to
    let allHeartData = activityDoc.heart;

    //Get the keys of the active minute activities only (exclude 'heart')
    let activityKeys = Object.keys(activityDoc);
    activityKeys.splice(activityKeys.indexOf('heart'), 1);

    //For each day of heart data
    for (const day of allHeartData) {

        //Get the date
        let date = day.date;

        //Get the minute-by-minute data
        let heartIntraday = day.intraday;

        //Check each type of active minute data
        for (const activityKey of activityKeys) {

            //Get the data for the current type
            let allMinuteData = activityDoc[activityKey];

            //For each day of active minute data of the current type
            for (const minuteData of allMinuteData) {

                //If the date matches the current heart day, we want to match the two intraday array times
                if (minuteData.date == date) {

                    //Get the minute-by-minute datasets for the current day of the current active minute type
                    let minuteIntraday = minuteData.intraday;

                    //New array to keep values we are interested in
                    let newMinuteIntraday = [];

                    //Iterate over each heart minute
                    for (const heartDataset of heartIntraday) {

                        //Iterate over each active minute minute
                        for (const minuteDataset of minuteIntraday) {

                            //If the times match, we want to keep the active minute dataset
                            if (heartDataset.time == minuteDataset.time) {

                                newMinuteIntraday.push(minuteDataset);

                            }

                        }

                    }

                    //Replace the old day dataset with the new one containing only values from when the device was being worn
                    minuteData.intraday = newMinuteIntraday;

                }

            }

        }

    }

}

```


Figure 27 Removing datasets from when the user was not wearing their device

The main method related to the graph-based classifier is `buildCGClassifier`. This method iterates over each activity type and then over every dataset and finds how many datasets have a value for the activity that is **less** than the one in the current dataset. This information could be used to plot a graph of the values for the activity (x axis) against the number of datasets where the value was lower than the current value (y axis). This graph, along with the frequencies of each class within the data (this information was stored in the database earlier), is used to define the boundaries for each class for the activity. For example, if the frequency of the lowest class ('not active') is 80%, then one would find the value for 80% on the y axis and then read the related value on the x axis. This value would define the boundary between 'not active' and the next class ('lightly active'). If the next class had a frequency of 5%, then the same process should happen as with the lowest class, except the 5% should be added to the previous 80%, meaning the value for 85% should be read. This idea is simulated in the code for this classifier as there was no way to plot a graph. When one would find the frequency value on the y axis, the code instead finds the value before and the value after this. These values and their corresponding x values are used to create a straight line between the two points. The function for this line is found, meaning that the frequency value (y) can be plugged into the equation to find the respective x value. Figure 28 shows this.

```

let totalFq = 0;
for (const fqKey of keys) {
  // console.log(fqKey);
  let frequency = minuteTotals[fqKey];
  let y = frequency * numHours;
  totalFq += y;

  let idx = 0;
  let y1 = 0;
  let x1 = 0;
  let y2 = 0;
  let x2 = 0;
  for (const doc of hourlyDocs) {
    // console.log("Total Fq: " + totalFq + ", numLess: " + doc.numLess + ", value: " + y);
    if (doc.numLess > totalFq) {
      y2 = hourlyDocs[idx].numLess;
      x2 = hourlyDocs[idx].value;
      if (idx > 0) {
        idx--;
      }
      y1 = hourlyDocs[idx].numLess;
      x1 = hourlyDocs[idx].value
      break;
    }
    idx++
  }

  let m = (y2 - y1) / (x2 - x1);
  let c = y2 - (m * x2);
  let x = (totalFq - c) / m;
  thresholds[fqKey.substring(0, fqKey.length - 9)] = x;
}
classifier[field] = thresholds;

```

Figure 28 Simulating a graph reading

Finally, in `ProfileBuilder.classifyCGData`, the boundaries created when creating the classifier are used to classify new hourly values. This is done for each activity by finding which boundary the value fits in. This provides a set of five classifications (as there are five activity types), of which the highest class is taken. Figure 29 shows an example of these boundaries. Using the calories row as an example, anything below 2.3114... is classed as 'not active', and anything between 2.3114... and 4.3295... is 'lightly active' and so on.

```
CGClassify thresholds:
▼ {steps: {...}, calories: {...}, floors: {...}, distance: {...}, heart: {...}} ⓘ
  ▶ calories: {minutesSedentary: 2.3114820145099393, minutesLightlyActive: 4.329518246429304, minutesFairlyActive: 4.850036632791583}
  ▶ distance: {minutesSedentary: 0.005279848219000317, minutesLightlyActive: 0.02513770357584356, minutesFairlyActive: 0.02753974733343478}
  ▶ floors: {minutesSedentary: 0.01694486026485725, minutesLightlyActive: 0.089874436317861, minutesFairlyActive: 0.11455136478799183}
  ▶ heart: {minutesSedentary: 78.99753095272241, minutesLightlyActive: 91.4332890739486, minutesFairlyActive: 97.11220182609111}
  ▶ steps: {minutesSedentary: 7.285850184043221, minutesLightlyActive: 34.62903434636099, minutesFairlyActive: 37.94527941106171}
  ▶ __proto__: Object
```

Figure 29 Graph-based classifier boundaries

Problems

Some problems were encountered during the creation of the data mining functionality. These problems were as follows:

1. **Data mining package incompatibility:** As React Native is JavaScript-based, I was anticipating using a JavaScript-based package from npm to carry out the data mining tasks, but it was discovered that this package was incompatible with React Native. This was a major setback as it sparked a search for a way to carry out data mining tasks late into the implementation phase of the project. I thought that an alternative could be to use a familiar Java package as this is an Android application and React Native allows use of native Java code to implement things that cannot be implemented in React Native. Furthermore, as I had experience with this Java package, this seemed like an ideal solution. It was then discovered that this package contained functionality that was not compatible with the Android device and could not be used as a result. Eventually, an open-source decision tree approach [14] was discovered in pure Python, so this was adapted into JavaScript for the project.

2. **Grouping datasets:**

A classification for a minute of data is not particularly useful for a user. For this reason, it was decided that the data mining should be done on hourly datasets. This introduced the problem of classifying these grouped datasets, as the Fitbit data only contains an activity level for each minute. This led to the decision tree approach to grouping datasets.

The decision tree approach to grouping datasets, where a new tree is created for each hour in order to classify an average dataset for the hour, seems to return low activity levels as classifiers for the hourly dataset. I believe that this is because there are generally more 'not active' minutes in an hour, which outweigh the 'very active' values when creating the average dataset. Future work could involve an investigation into how these hourly classifications would be different if the average dataset was classified on a tree created for the entire day's data (≤ 1440 minute datasets), instead of just the hourly (≤ 60 minute datasets) data. This idea of using all the day's data could also be taken further and tested on data where the data points from when the user is sleeping are excluded.

3. **Duplicate code:**

The problems described above led to a lot of time being spent on unplanned tasks. This meant that often when a piece of code was working for the first time, it was left in the state it was in for the remainder of the project, as there was little time to refactor and tidy up the code base. This means that there are aspects of the data mining code that are very similar between the decision tree and graph classifiers. Given more time, these similar pieces of code would have been refactored into one piece, so that it would only be written once. An example of this repetition is the 'matchDataToHeartDataset' in DecisionTreeMining.js and 'matchToHeartData' in CumulativeGraphClassifier.js.

Fitbit Communication

FitbitCaller.js

This file contains methods that are used to authorise the application with the Fitbit Web API and request data from it.

Methods 1, 2, and 3: authenticate(), authorizeUser(), and refreshToken()

These methods authenticate the user with the Fitbit Web API or refresh the tokens each time the application needs to request data.

Method 4: generateUri(String activityName, boolean intraday, String startDate, String endDate)

Generates URIs that can be used to request data from the Fitbit Web API. The activityName parameter states which activity the application needs data for. The intraday parameter states whether the application is requesting intraday data. The final two parameters specify the start and end dates of the period that the application requires data from.

Method 5: requestData(String uri)

This method is used to request data from the Fitbit Web API. The uri parameter is a URI that was generated by the generateUri method.

Problems

1. Fitbit API Registration:

I planned the application to request the data from Fitbit, however I encountered the issue of registering the application with Fitbit. The registration form requires a website for the application, which this project does not have. After some further investigation, it was discovered that a dummy value could be used for the application. This was a very simple conclusion, however the investigation required to find this solution took a chunk out of the implementation time.

2. Fitbit Authentication:

When planning the project, I did not consider the necessity of authenticating the application with Fitbit. In order to get data from Fitbit, I had to learn about OAuth2 authentication and callback URLs, which I had never encountered before.

Database

DatabaseController.js

This file contains all the methods that create the database and allow the system to interact with it.

Method 1: createDb()

This method creates various datastores. Each datastore behaves like a MongoDB collection. There is a datastore for each element of activity data, as well as exercise data, the user profile, and any deviations.

Method 2: insert(String dbName, Object jsonDoc)

Inserts the given jsonDoc parameter into the datastore that is identified by the dbName parameter.

Method 3: fetch(String dbName, Object jsonDoc, Object project)

Fetches documents from the datastore that is identified by the dbName parameter. The jsonDoc parameter is used to provide query terms, and the project parameter is used to project parts of the document, just like in plain MongoDB.

Method 4: fetchAndSort(String dbName, Object jsonDoc, Object project, Object sort)

This is like fetch(), however the extra parameter, 'sort', is used to sort the returned array of documents.

Method 5: update(String dbName, Object jsonDoc, Object update, Object options)

Updates documents that match the query criteria defined by the jsonDoc parameter within the datastore identified by the dbName parameter. The update parameter contains information about what should be updated within the relevant documents, and the options parameter is used to specify options (such as whether multiple documents should be updated).

Method 6: remove(String dbName, Object jsonDoc, Object options)

Removes documents that match the query criteria defined by the jsonDoc parameter within the datastore identified by the dbName parameter. The final parameter is used to specify options.

Method 7: clear()

This method clears the database. It was originally for debugging, however it has remained in the application so that the user can clear the database if they desire to.

Other Notable Code

DateUtilities.js

This file contains some static methods that assist with date-related activities within the application. All dates within the application are strings with the format "yyyy-MM-dd". This is because Fitbit uses this format in their responses, so this format was adopted in other areas of the application for consistency and simplicity.

Method 1: `getDaysOnOrAfter(String startDate)`

This method takes a parameter which is a date string of the format described earlier. This date is expected to be before the current day. The method systematically creates a string from the current date in the desired format before repeating this process for all the dates up to the date in the parameter. The method returns an array which contains all the dates between the current day and the start date. The purpose of this method is to get a list of dates since the last time the profile was updated, so that the system focuses on requesting and classifying new data.

Method 2: `getDateAsString(Number offset)`

Any date string generated within the application is generated by this method. The method returns the current date in the form “yyyy-MM-dd” if the ‘offset’ parameter is 0. The parameter is used to get dates at an offset from the current day. For example, if the offset is -1, then the method returns yesterday’s date in the desired format.

This method was useful for debugging as an offset of -1 can be used within the method to effectively set the date of the application to the previous day. This meant that a profile could be built before resetting the date to the current day by removing the offset and restarting the application so that the profile can be refreshed with new data. This meant that the classification/deviation functionality could be tested effectively, as deviations are only created on new data, not during the first profile build.

Method 3: `isDateBefore(String dateStringA, String dateStringB)`

This method is simply used to compare which of the two date string parameters is before the other. The method returns ‘true’ if date A is before date B, and false otherwise.

Packages Used

`react-native`

This is the package for React Native [6]. This is crucial to the entire project.

`react-native-app-auth`

This package [17] is an authentication package for React Native. This was used for authentication with the Fitbit Web API [1].

`react-native-local-mongodb`

This is the package [9] that is used to create and manipulate the database. All code used from this package is in `DatabaseController.js`.

`native-base`

This is the package [18] that is used for various on-screen components such as buttons.

`react-native-svg-charts`

This package [19] is used to create the graphs on the activity page and the exercise page.

`react-native-svg`

This package [20] is included as `react-native-svg-charts` depends on it.

react-navigation

This package [21] is used to handle the navigation throughout the application.

User Interface

Profile Page

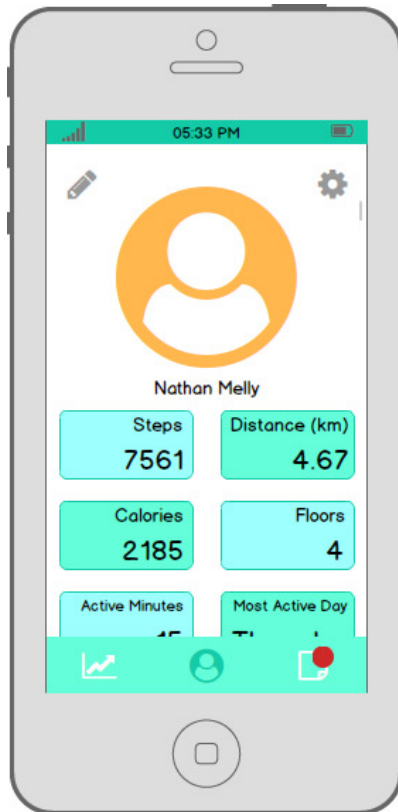


Figure 30 Profile Page Design

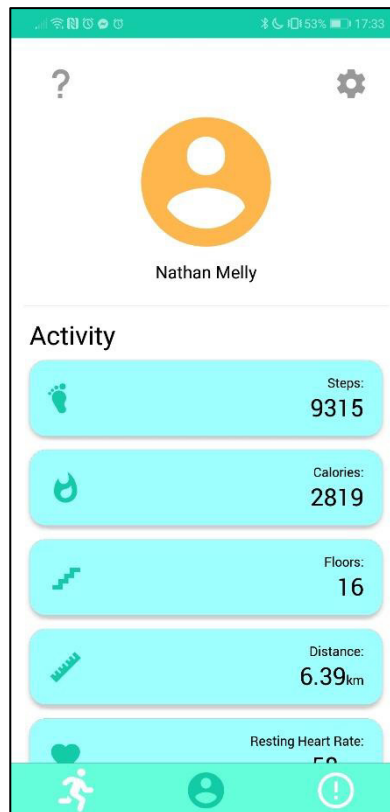


Figure 31 Profile Page Screenshot

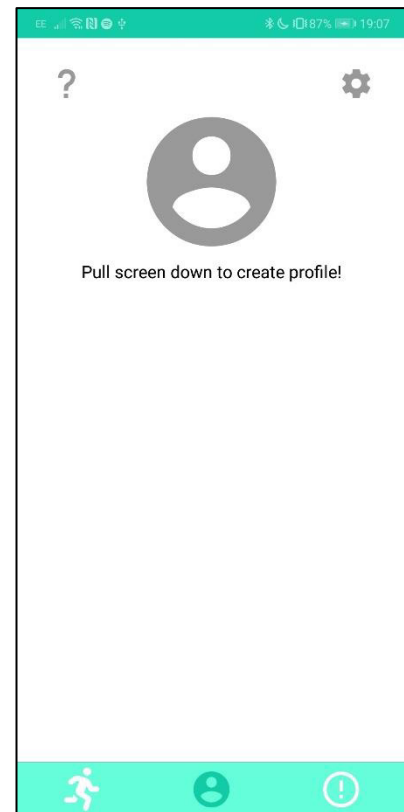


Figure 32 Profile Page Screenshot (no profile)

Differences between design and screenshot

The profile icon is smaller in the implemented application. This is to allow more screen space for cards, which are more interesting and useful to the user.

Another key change is that the edit button was replaced by a help button. During implementation, it was discovered that profile information such as the user's name and date of birth etc. could be requested from Fitbit. This meant that there was not much need for the user to edit profile details in this application as they could do it with Fitbit. The 'Edit Profile' page was then replaced with a 'Help Page' that simply provides the user with a simple guide on how to use the application.

The icons on the navigation bar are also different in the original design. This is simply because the software used to create the design had a limited selection of icons and so during implementation, more appropriate icons were found that aid the user in understanding what each button represents.

The other noteworthy change is the colour and layout of the cards. During implementation, the two-card-per-row layout was tested, however it was not as usable as the one-card-per-row-layout. The alternating colours of the cards in the original design could have wrongly led

the user to believe that the functionality of the cards was different if the colour was different, so a decision was made to keep all the cards one colour.

Activity Page

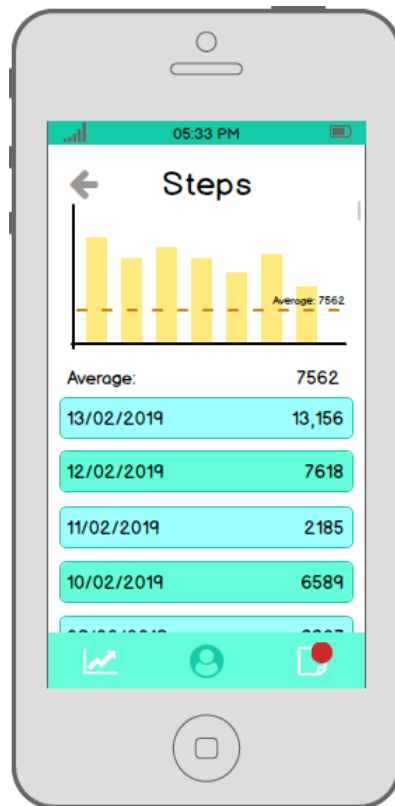


Figure 33 Activity Page Design

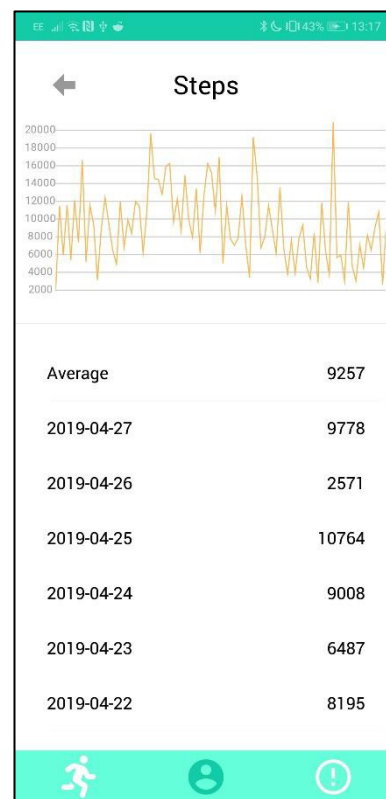


Figure 34 Activity Page Screenshot

Differences between design and screenshot

The most obvious difference between the design and implementation is that the graph is a line chart, however it is a bar chart in the design. This is because the only graph image in the design software was a bar chart, so this was used as a placeholder in the design. The graph shows all the available data for the activity, so a bar chart would not have been an appropriate choice as there would have been too many bars, meaning that the graph would have likely had to be very small to accommodate this.

Another notable difference between the two is the lack of 'Average' line on the chart. This is because the 'react-native-svg-charts' package [19] used in implementation only allows one set of data to be plotted on a chart, meaning that it was not possible to add a second line to the chart. A workaround could perhaps have been found, however it was not deemed important enough to warrant the time that it would have taken to find a solution.

A final difference of note is that the list has no colour in the implementation, whereas it does in the design. This is because I was anticipating creating the list from cards, however a third-party package had a list JSX object that created the simple list in the image on the right. The list in the implemented application looks tidier than the colourful cards in the design, so this change was a positive one overall.

Exercises Page

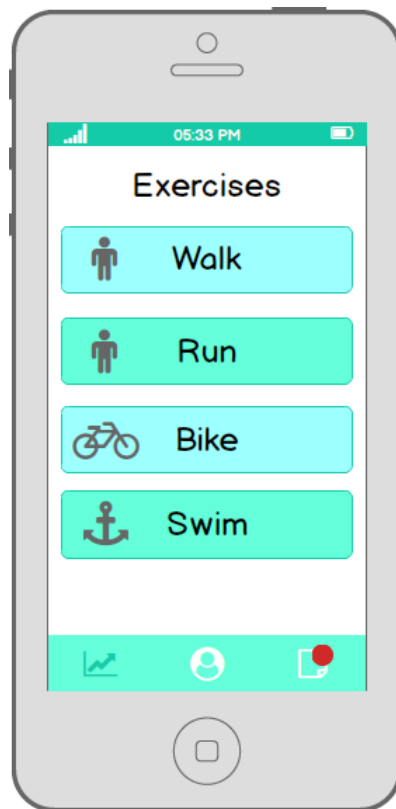


Figure 35 Exercises Page Design

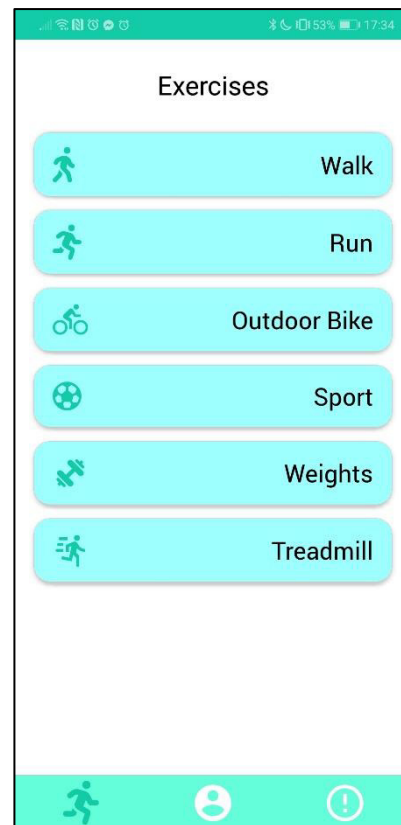


Figure 36 Exercises Page Screenshot

Differences between design and screenshot

The main difference between the two images above is that the cards are all the same colour in the implementation. This is because it was decided that alternating the colours was unnecessary and having different colours on the cards could lead the user to think that they have different purposes.

Exercise Page

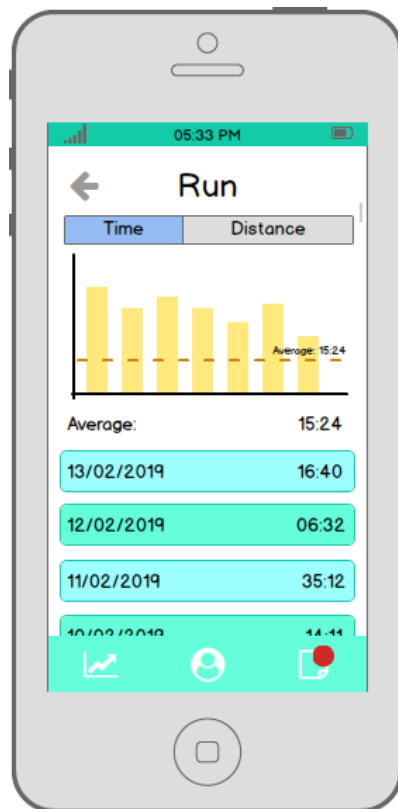


Figure 37 Exercise Page Design



Figure 38 Exercise Page Screenshot

Differences between design and screenshot

The main differences are the graph being a line chart and not a bar chart as designed, the 'Average' value line not being present in the implementation, and the list being colourful in the design. The reasons for these differences have already been discussed in the 'Activity Page' section.

The other difference between the two images is the style of the picker. The picker looks the way it does in the implementation because it is the only style of picker available in React Native and it did not seem necessary to depend on an extra third-party package for such a minor detail.

Alerts Page

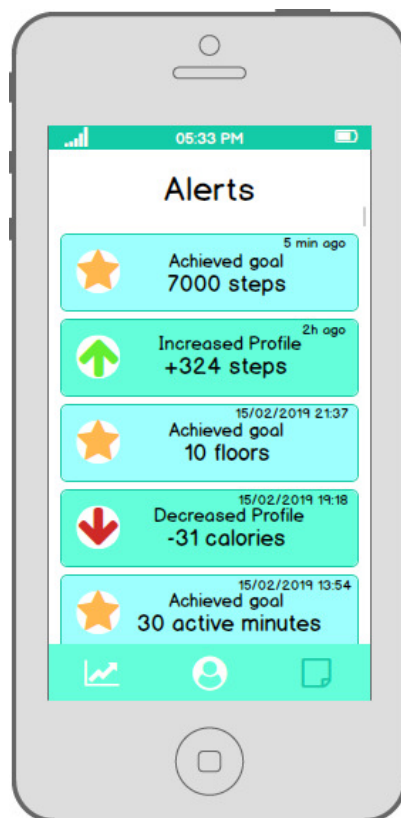


Figure 39 Alerts Page Design



Figure 40 Alerts Page Screenshot

Differences between design and screenshot

The cards in the implemented version are the same colour so that the user does not think that they behave in different ways. They are a lighter colour than the touchable cards in other areas of the application in order to distinguish them from the touchable cards, as these cards do nothing when pressed.

Another difference is that the icons on the cards are the dark primary colour from the colour scheme instead of different colours, as in the design. The text on the cards is also right-aligned so that the user's eye can move in a straight line down the screen, and the user knows where the information will be due to the greater consistency between the cards.

The picker in the implementation allows the user to filter the types of cards they see.

Results and Evaluation

Testing and Evaluation of Data Mining Approaches

In order to compare the classifiers, each was implemented in the main flow of the application and tested on real data. The data was imported in chunks spanning five days each. Data would usually be imported up to seven days at a time, however for the purposes of not hitting Fitbit request limits, two five-day chunks were pulled into the application each hour. The process was as follows:

1. Pick a date over five days from the first day of data (30th January 2019).
2. Set the application date to this chosen date by offsetting the returned value in `DateUtilities.getDateAsString`.
3. Clear the application database.
4. Restart the application.
5. Pull the profile screen down to refresh the profile.
6. Wait while the data is imported and processed.
7. Close the application and lower the offset in `DateUtilities.getDateAsString` by five in order to move the application date forward by five days.
8. Start the application.
9. Pull the profile screen down to refresh the profile.
10. Wait while the new data is imported, a classifier of each type is built from the existing data, and the new data is classified on the classifiers. The results are printed to the console.
11. Copy the results into a CSV file to be evaluated later.
12. Repeat steps 7 – 11 until there is no longer an offset in `DateUtilities.getDateAsString`. The application is up to date.

Following the approach outlined above, the application was tested on 76 days of data (the original offset was 70, however the testing took two days and so these extra days were added to the offset as testing progressed). The initial application date was 20th February 2019, and testing was completed on 3rd May 2019, meaning that the data spanned 15th February 2019 – 2nd May 2019.

The decision to test the application in chunks of five days was to enable the classifiers various opportunities to incorporate more data, which in theory should make them more accurate. Using chunks of seven days would have meant that two tests could not be done safely per hour because there was potential to hit the Fitbit request limit and using chunks of under five days would have resulted in many more tests. For these reasons, five days became the date span of choice.

The results of these tests can be found below. Each row corresponds to the import of five days of data, except for test 15, which had two days imported. The results start from the **second** data import, as this is the first time that data is classified (no classifier is built during the first profile build). Each test tested the classifiers, which were built from **all** data that existed within the application **before** the current import. The newly imported data was then classified on these classifiers. Classification accuracy was worked out by dividing the number of correct classifications by the total number of classifications (apart from in the final two

columns, where the number of correct classifications was instead divided by the total number of classifications **minus** the number of classifications where both classifiers were wrong).

Test Number	Total Classifications	DT Incorrect	Graph Incorrect	Both Incorrect	DT Correct %	Graph Correct %	DT Correct % (ignoring both incorrect)	Graph Correct % (ignoring both incorrect)
1	119	19	1	0	84.03361345	99.15966387	84.03361345	99.15966387
2	120	19	0	0	84.16666667	100	84.16666667	100
3	120	12	7	1	90	94.16666667	90.83333333	95
4	119	8	1	0	93.27731092	99.15966387	93.27731092	99.15966387
5	120	10	4	2	91.66666667	96.66666667	93.33333333	98.33333333
6	120	20	0	0	83.33333333	100	83.33333333	100
7	118	12	1	0	89.83050847	99.15254237	89.83050847	99.15254237
8	119	16	1	0	86.55462185	99.15966387	86.55462185	99.15966387
9	120	19	1	0	84.16666667	99.16666667	84.16666667	99.16666667
10	120	19	1	0	84.16666667	99.16666667	84.16666667	99.16666667
11	120	14	0	0	88.33333333	100	88.33333333	100
12	120	9	1	0	92.5	99.16666667	92.5	99.16666667
13	112	14	0	0	87.5	100	87.5	100
14	117	11	1	0	90.5982906	99.14529915	90.5982906	99.14529915
15	48	4	0	0	91.66666667	100	91.66666667	100
All	1712	206	19	3	87.96728972	98.89018692	88.14252336	99.06542056

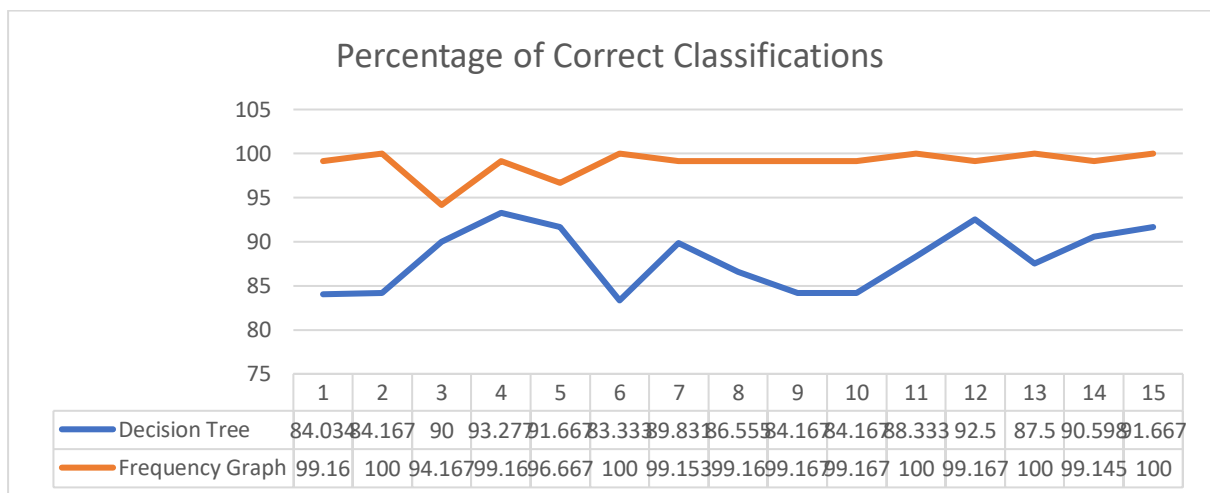


Figure 41 Test results for both classifiers

We can see from the results in Figure 41 that there is a clear difference in the performance of the classifiers. The graph classifier was consistently close to 100% accuracy, and its lowest accuracy was 94.167%. This is a contrast to the performance of the decision tree, which struggled to reach 90% accuracy, although it did manage this a small number of times. One could hypothesise that the decision tree classifier is not accurate because the new data in each test was genuinely inconsistent with the existing data, however because the graph classifier classified the data correctly in most cases, we can assume that this is not the case. There were a number of instances where **both** classifiers provided an incorrect classification,

where we could perhaps assume that the new data was genuinely inconsistent with the existing data, however this happens at such a low rate that these anomalous classifications make a minimal impact on the percentage of correct classifications for each classifier.

We can also see from the results that the decision tree classifier's accuracy does not improve over time. This could be due to one of several potential causes:

1. **The implementation of the classifier.** This is a likely cause as the implemented tree is very basic and the tree can have any number of levels.
2. **The method of grouping the datasets biases the data.** An interesting topic for future work could be to investigate other ways in which minute datasets can be grouped into hourly datasets for this classifier.
3. **The imported data is genuinely dissimilar to the existing data during each test.** This is unlikely, as the graph-based classifier almost always has no problem classifying datasets that the decision tree classifier gets wrong.

Unfortunately, the test results do not help answer the question of how the accuracy of the graph-based classifier changes over time. This is because the results are too similar and are very high from the first test. This meant that there was almost no scope for the graph-based classifier to improve.

Something worth noting for the graph-based classifier is that the classification for every hourly dataset was one of five possible classifications (one each for steps, calories, floors, distance, heartrate). The classifier related to the highest possible class was chosen from the set of five for each hourly dataset. This is not necessarily the best way to get a single classification for the hourly data. Future work should include an investigation into the performance of this classifier when a different classification is taken from the five potential classifications. Moreover, a better way of finding a single classification should be investigated.

One conclusion that can safely found from the tests on these classifiers is that the graph-based classifier is more accurate than the decision tree classifier, at least within the context of this application and its implementation of the mining techniques.

Evaluation Against Requirements

This section goes through each requirement and analyses if it was met or not, before providing a justification for the outcome of each requirement.

Requirement Number	Requirement Met	Justification
<i>Functional</i>		
Must		
1	Yes	A profile is created and displayed in the profile page.
2	Yes	When the profile is refreshed, cards appear in the Alerts tab if the user has a different activity level based on similar data to their model.

3	Yes	The profile can be refreshed once per day. If the user attempts to refresh again, an alert pops up stating that the profile is up-to-date.
4	Yes	Alerts are displayed to the user when their profile rebuild starts and when it is complete.
5	Yes	The user can still use the application after they have initiated a profile refresh.
6	Yes	The classifiers were tested in the section above.
7	Yes	When the user navigates to the activity page, a graph containing data for the chosen activity is displayed.
8	Yes	When the user navigates to the activity page, a list containing data for the chosen activity is displayed.
9	Yes	When the user navigates to the exercises page and 'Time' is selected in the picker, a graph containing durations of their sessions of the selected exercise is displayed.
10	Yes	When the user navigates to the exercises page and 'Time' is selected in the picker, a list containing dates and durations of their sessions of the selected exercise is displayed.
11	Yes	When the user navigates to the exercises page and 'Calories' is selected in the picker, a graph containing the calories burned in their sessions of the selected exercise is displayed.
12	Yes	When the user navigates to the exercises page and 'Calories' is selected in the picker, a list containing dates and calories burned in their sessions of the selected exercise is displayed.
13	Yes	When the user navigates to the exercises page and 'Distance' is selected in the picker, a graph containing the distance travelled in their sessions of the selected exercise is displayed.
14	Yes	When the user navigates to the exercises page and 'Distance' is selected in the picker, a list containing dates and distances travelled in their sessions of the selected exercise is displayed.
15	Yes	Cards are displayed on the profile page that tell the user the average steps, calories, distance, and floors they do each day, and their average resting heart rate.
16	Yes	
17	Yes	
18	Yes	
19	Yes	

20	Yes	Four cards are displayed on the profile page that tell the user their average number of minutes in each activity level per day.
21	Yes	The user's data is retrieved from the database and presented to them when they restart the application.
22	Yes	When the user pulls the profile page down to refresh their profile, data is imported and then the values in their profile are recalculated to include the new data. These values are then updated on the profile page, and an alert is displayed to the user when their profile build is complete.
Should		
23	No	This functionality was not included because the project focussed on testing a second data mining approach as the project progressed.
24	No	This was decided against once implementation started as it was discovered that basic user information can be gleaned from their Fitbit profile, so it seemed unnecessary to get the user to input this information a second time.
25	No	This was not met because the focus moved away from the manual input aspects of the application and moved instead to the comparison of two data mining approaches.
26	No	This was not included as it did not seem straightforward to do using the third-party graph package. It is likely that there is a solution, however this did not seem important enough to justify spending much time on. The average value is displayed in the list on the activity page however.
27	Yes	The most common exercise type is worked out based on how much time the user spends doing each exercise type. Based on my data, this exercise was 'Walk'. This is unsurprising because Fitbit logs exercises when one walks. In future, this could be amended to exclude walking, or exclude automatically logged sessions.
28	No	This was a desirable feature as opposed to a necessary feature. It was not important enough to justify spending time on it when there were issues with other areas of the application.
29	No	There was not enough time to implement this due to issues with other areas of the application, however this would be a good thing to display on the Alerts page in future.
30	Yes	No exercise type is displayed on the exercises tab unless it exists in the user's data.

Could		
31	No	Working with sleep data was defined as an extra challenge if other aspects of the application were complete before the project deadline. This was not the case however, so this requirement remains incomplete.
32	No	Like requirement 31, this was an extra challenge that there was no time for.
33	No	See justification for requirement 31.
34	Yes	This is one of the core aspects of the application. User data is requested from Fitbit before the profile is created.
35	No	This was deemed unnecessary once the application could directly import data from Fitbit. This requirement also presents the issue of the user having to create/import a file onto their Android device, which is extra hassle for the user.
36	No	There was no time to implement this given the focus on getting other areas of the application working.
37	No	As discussed in the justification for requirement 36, there was no time to implement exercise reminders.
Will Not		
38	Yes	There is no way to view other users' data within the application.
Non-Functional		
39	Yes	The application takes less than $n \times 20$ seconds to import data from n days, build the profile, and classify the data.
40	Yes	A heuristic evaluation was done on the UI designs to identify issues with it. Many of these issues were addressed during implementation to produce an application that is intuitive.

Requirements Type	Total Requirements	Number of Requirements Met	Percentage of Requirements Met
Functional	38	26	68.421%
<i>Must</i>	22	22	100%
<i>Should</i>	8	2	25%
<i>Could</i>	7	1	14.283%
<i>Will Not</i>	1	1	100%
Non-Functional	2	2	100%

All	40	28	70%
-----	----	----	-----

The first table above shows which of the 40 requirements were met and provides an explanation of how each one is met (or why it is not met). We can see from the tables above that 70% of all requirements were met by the final application, including 100% of the 'must' requirements, which is a success. Only around 14% of the 'could' requirements were met, which is acceptable as these requirements expressed non-essential functionality to be included if there was enough time within the project. Only 25% of the 'should' requirements were met, which is lower than anticipated. The reason for many of these requirements not being met is that there was originally only going to be one data mining method implemented within the application, but two were eventually created and compared. This work was deemed more important than many of the requirements in the 'should' section, and this is justified by the fact that through testing the two classifiers, we have seen that the new graph-based classifier is much better than the original classifier. This is a conclusion that would not have been possible to make if this work was not carried out.

Future Work

Network Errors

During implementation, no method of handling network errors was put in place. As stated earlier, this project assumes that these network errors will not happen, however this is clearly not the case in the real world. Often, these network errors are out of the user's hands, and so work should be carried out to ensure that these errors have no effect on the state of the application.

Changing Classifier

We have seen from testing that the graph-based classifier was much better at classifying user datasets as the model was more accurate. Going forward, this classifier should be the one used within the application to create the deviations, not the decision tree classifier.

Adjusting Decision Tree Classifier Grouping Method

The decision tree minute datasets were grouped into hourly datasets by creating a decision tree for each hour's data, creating an average dataset from this data, and classifying it on the decision tree. This is not necessarily an effective way of grouping the datasets into hours, and so an investigation into how else this can be handled should take place. One suggestion is to classify the average hourly dataset on a decision tree created from the entire day's data, not just one hour.

Remove Sleep Datasets

When the user is asleep, their activity level is almost certain to be 'not active' by default. Ignoring or removing the datasets from when the user is asleep could drastically alter the classifiers, especially the graph-based classifier, which uses the frequency of classes within the data to decide how to classify unseen datasets. Removing these datasets would make the frequency of 'not active' datasets much lower.

Classifying Days

Currently, the application can only classify hourly datasets. This is likely to be quite useful to users, however it would also be useful if the system could give the user an indication of how active they were for an entire day. This would also reduce the number of requests that would need to be made to Fitbit. Users can request daily total values for a given activity for as many days as they would like in a single request, however if they would like to request data at a smaller granularity, they need individual intraday requests. This is an issue with the request limit of 150 per hour, imposed by Fitbit.

Authentication Bug

There is a bug that causes the application to hang when authenticating occasionally. The cause of this bug is unknown, but it seems to be during a call of a third-party method. This bug should be investigated, as currently there is no way to feedback to the user if the application is hanging.

Further Testing

The application should be tested on data from different timeframes to see how the classifiers adapt to different datasets. Further to this, the data should be added to each classifier in varying increments to see how quickly the classifiers' accuracy changes. The classifiers were tested by adding 5 days of data at a time, however it would be interesting to see how they would behave if data was added in groups of 3 or 7 days for example.

Additionally, the graph-based classifier should be tested using a different classification from the possible five it creates, to see how this affects the accuracy.

Manual User Input

There was not enough time to implement mechanisms that allow the user to manually log exercise sessions and activity data. This should be looked at in future, as it would be beneficial to allow users to log activity from times when they were not wearing their device.

Adapting to Fitbit Devices

This project was completed using a Fitbit Charge 3 to gather data. As not all Fitbit devices track the same data, this application should be tested with other Fitbit models, and adapted as necessary to work with these.

Conclusions

The main aim of this project was to create an application that can profile a user based on their Fitbit data, allows the user to track their activity, and alerts the user when they deviate from their profile. This has been achieved by creating an Android application, using React Native, that allows users to import data directly from their Fitbit profile, before performing some simple data analysis to provide the user with information such as the average number of steps they complete in a day. More complex data mining methods were employed in order to analyse the way a user's activity level changes with their activity data. This project has evaluated the effectiveness of a decision tree and a graph-based classifier for this purpose and has found that the graph-based classifier is the more accurate classifier.

Each time a user imports new data into the application, this new data is compared to their existing data in order to find deviations from their existing profile. These deviations tell the user when they have achieved less than usual in terms of their activity data, such as steps and calories. Deviations are also created using the decision tree classifier (this classifier is used as it was implemented within the flow of the application originally) to tell the user when their data during a given hour of the day provided a different activity level than what was expected by the classifier. This information is displayed to the user within a responsive and intuitive application.

Overall, this project has been a success in terms of creating a usable mobile application to allow users to track and read their Fitbit data and find deviations in their activity level based on this data.

Reflection on Learning

The main thing I have learned during this project is that I should research the capabilities of tools and languages more before using them. The choice to use React Native turned out to be poor because the framework itself is quite new and support is therefore minimal. This framework also proved to be a poor choice for a data mining project as many of the packages I intended to use turned out to be incompatible with it. Furthermore, the underlying database turned out to be incredibly small, which caused issues when testing the application, causing me to have to rework large parts of the application at a late stage in the project (although I do not believe that I would have found this issue with added research as the database restriction is not mentioned in the React Native documentation as far as I can tell). If I were to repeat this project, I would use more appropriate languages such as Python for the data-related aspects, and perhaps store the data on a server.

In addition, I have learned that I should decide on complex areas of functionality earlier in projects. I did not know what data mining techniques I wanted to use until implementation had started, which lead to unnecessary stress when trying to research various solutions before choosing and implementing one. There were also further issues with React Native as any packages I found to potentially aid with data analysis were incompatible. Use of Python or Java would have meant that I could use a variety of well-supported packages for this project.

A personal aim of the project was to learn about a new framework and mobile application development. Whilst I found React Native difficult to work with at times, I found the challenge of learning a new technology as part of the project rewarding. I believe that I was successful in learning about mobile development as I have managed to create an application that is usable and has a range of useful features. This project has given me a new skill and knowledge that I can take forward into future projects that require mobile development experience.

Appendices

Appendix A

Use Cases

Log Activity Data

This use case allows the user to manually log activity data for adding to the profile.

Preconditions

- The application is already running.

Main Flow

1. **Press the Profile icon**

This use case starts when the user presses the profile icon on the main tab bar at the bottom of the application screen. The profile page is displayed.

2. **Press an activity card**

The user presses an activity card on the profile page and is taken to a page that contains detailed data for that activity.

3. **Press the log entry icon**

The user presses the '+' icon on the activity page that they are currently on. The user is presented with a form that asks for information related to the activity.

4. **Complete and submit the form**

The user completes the form and presses the 'Submit' button. The database stores the information that the user entered with the rest of the data for the specific activity that the user has logged an entry for. The user's profile is automatically rebuilt.

Alternative Flows

4A. **Cancel the entry**

The user presses the 'Cancel' button and is asked to confirm that they do not want to submit the entry before cancelling. The user is returned to the activity page that they were previously browsing.

View Exercise Calories

This use case allows the user to view the number of calories burned during exercise sessions that they have completed.

Preconditions

- The application is already running.
- The user has synchronised the application with Fitbit.
- The user has tracked exercises with their Fitbit device.

Main Flow

1. **Press the Exercise icon**

This use case starts when the user presses the exercise icon on the main tab bar at the bottom of the application screen. The exercise page is displayed and contains a list of cards containing exercise types that the user has completed such as 'Run', 'Walk', and 'Cycle'.

2. **Press an exercise card**

The user presses a card on the exercise page and is taken to a page that contains detailed data for that exercise. For example, if the user presses the 'Run' card, they are taken to a page containing a drop down menu that is set to 'Calories' by default, a list of the number of calories burned in each run, and a graph showing the calories burned in each run.

View Exercise Time

This use case allows the user to view the time taken to complete exercise sessions.

Preconditions

- The application is already running.
- The user has synchronised the application with Fitbit.
- The user has tracked exercises with their Fitbit device.

Main Flow

1. **Press the Exercise icon**

This use case starts when the user presses the exercise icon on the main tab bar at the bottom of the application screen. The exercise page is displayed and contains a list of cards containing exercise types that the user has completed such as 'Run', 'Walk', and 'Cycle'.

2. **Press an exercise card**

The user presses a card on the exercise page and is taken to a page that contains detailed data for that exercise. For example, if the user presses the 'Run' card, they are taken to a page containing a picker menu that is set to 'Calories' by default, a list of the number of calories burned in each run, and a graph showing the calories burned in each run.

3. **Select the picker menu**

The user presses on the picker menu, and a list of possible views is displayed. This list contains 'Calories' and 'Time' for all exercise types, and 'Distance' for activities where this is available.

4. **Press 'Time'**

The user returns the exercise screen, which now shows a picker menu that is set to 'Time', a list of times for each session of the selected exercise, and a graph showing the time taken for each session.

View Exercise Distance

This use case allows the user to view the distance covered in exercise sessions where this value was recorded.

Preconditions

- The application is already running.
- The user has synchronised the application with Fitbit.
- The user has tracked exercises with their Fitbit device.

Main Flow

1. **Press the Exercise icon**

This use case starts when the user presses the exercise icon on the main tab bar at the bottom of the application screen. The exercise page is displayed and contains a list of cards containing exercise types that the user has completed such as 'Run', 'Walk', and 'Cycle'.

2. **Press an exercise card**

The user presses a card on the exercise page and is taken to a page that contains detailed data for that exercise. For example, if the user presses the 'Run' card, they are taken to a page containing a picker menu that is set to 'Calories' by default, a list of the number of calories burned in each run, and a graph showing the calories burned in each run.

3. **Select the picker menu**

The user presses on the picker menu, and a list of possible views is displayed. This list contains 'Calories' and 'Time' for all exercise types, and 'Distance' for activities where this is available.

4. **Press 'Distance'**

The user returns the exercise screen, which now shows a picker menu that is set to 'Distance', a list of distances recorded in each session of the selected exercise, and a graph showing the distance covered in each session.

Log Exercise Data

This use cases allows users to manually log an exercise session.

Preconditions

- The application is already running.

Main Flow

1. **Press the Exercise icon**

This use case starts when the user presses the exercise icon on the main tab bar at the bottom of the application screen. The exercise page is displayed and contains a list of cards containing exercise types that the user has completed such as 'Run', 'Walk', and 'Cycle'.

2. **Press an exercise card**

The user presses a card on the exercise page and is taken to a page that contains detailed data for that exercise. For example, if the user presses the 'Run' card, they are taken to a page containing a picker menu that is set to 'Calories' by default, a list of the number of calories burned in each run, and a graph showing the calories burned in each run.

3. **Press the log entry icon**

The user presses the '+' icon on the exercise page that they are currently on. The user is presented with a form that asks for information related to the exercise.

4. **Complete and submit the form**

The user completes the form and presses the 'Submit' button. The database stores the information that the user entered with the rest of the data for the specific exercise type that the user logged an entry for. The user's profile is automatically rebuilt.

Alternative Flows

4A. **Cancel the entry**

The user presses the 'Cancel' button and is asked to confirm that they do not want to submit the entry before cancelling. The user is returned to the activity page that they were previously browsing.

Edit Profile Information

This use case allows users to edit basic profile information such as name, age, height, weight etc.

Preconditions

- The application is already running.
- The user has created a profile.

Main Flow

1. **Press the Profile icon**
This use case starts when the user presses the profile icon on the main tab bar at the bottom of the application screen. The profile page is displayed.
2. **Press the 'Edit Profile' icon**
The user presses the pencil-shaped icon to open the 'Edit Profile' screen. Here, the user can change their basic profile information such as name and date of birth.
3. **Edit profile and submit**
The user makes whatever changes they wish to their profile before pressing a green tick icon to submit their changes.

Alternative Flows

- 3A. **Cancel the edit**
The user presses the 'Back' button and if the user made changes, they are asked to confirm that they do not want to submit the changes before cancelling. The user is returned to the profile page.

Edit Profile Settings

This use case allows the user to edit settings related to the 'Profile' section of the application.

Preconditions

- The application is already running.

Main Flow

1. **Press the Profile icon**
This use case starts when the use presses the profile icon on the main tab bar at the bottom of the application screen. The profile page is displayed.
2. **Press the 'Profile Settings' icon**
The user presses the gear-shaped icon to open the 'Profile Settings' screen. Here, the user can choose settings such as what level of detail they wish to use for their activity data, and the amount of data they would like the application to use to build their profile.
3. **Edit settings and submit**
The user makes whatever changes they wish to their settings before pressing a green tick icon to submit their changes.

Alternative Flows

3A. Cancel the edit

The user presses the 'Back' button and if the user made changes, they are asked to confirm that they do not want to submit the changes before cancelling. The user is returned to the profile page.

Edit Exercise Settings

This use case allows the user to edit settings related to the 'Exercises' section of the application.

Preconditions

- The application is already running.

Main Flow

1. Press the Exercises icon

This use case starts when the user presses the exercises icon on the main tab bar at the bottom of the application screen. The exercises page is displayed.

2. Press the 'Exercise Settings' icon

The user presses the gear-shaped icon to open the 'Exercise Settings' screen. Here, the user can choose exercise related settings such as units, and how much data they would like to view on the graphs.

3. Edit settings and submit

The user makes whatever changes they wish to their settings before pressing a green tick icon to submit their changes.

Alternative Flows

3A. Cancel the edit

The user presses the 'Back' button and if the user made changes, they are asked to confirm that they do not want to submit the changes before cancelling. The user is returned to the profile page.

Edit Alert Settings

This use case allows the user to edit settings related to the 'Alerts' section of the application.

Preconditions

- The application is already running.

Main Flow

1. Press the Alerts icon

This use case starts when the user presses the alerts icon on the main tab bar at the bottom of the application screen. The alerts page is displayed.

2. Press the 'Alert Settings' icon

The user presses the gear-shaped icon to open the 'Alert Settings' screen. Here, the user can choose settings such as how often they would like to generate alerts.

3. Edit settings and submit

The user makes whatever changes they wish to their settings before pressing a green tick icon to submit their changes.

Alternative Flows

3A. **Cancel the edit**

The user presses the 'Back' button and if the user made changes, they are asked to confirm that they do not want to submit the changes before cancelling. The user is returned to the profile page.

References

1. Dev.fitbit.com. (2019). *Fitbit Development: Web API*. [online] Available at: <https://dev.fitbit.com/build/reference/web-api/> [Accessed 1 Feb. 2019].
2. Fitbit.com. (2019). *Fitbit Official Site for Activity Trackers and More*. [online] Available at: <https://www.fitbit.com> [Accessed 1 Feb. 2019].
3. Strava.com. (2019). *Strava | Run and Cycling Tracking on the Social Network for Athletes*. [online] Available at: <https://www.strava.com> [Accessed 5 Feb. 2019].
4. Myfitnesspal.com. (2019). *MyFitnessPal | MyFitnessPal.com*. [online] Available at: <https://www.myfitnesspal.com> [Accessed 5 Feb. 2019].
5. Juneja, N. and de Lassence, C. (2016). *webscale/Rbitfit*. [online] GitHub. Available at: <https://github.com/webscale/Rbitfit> [Accessed 5 Feb. 2019].
6. Facebook.github.io. (2019). *React Native · A framework for building native apps using React*. [online] Available at: <https://facebook.github.io/react-native/> [Accessed 1 Feb. 2019].
7. MongoDB. (2019). *The most popular database for modern apps*. [online] Available at: <https://www.mongodb.com> [Accessed 8 Mar. 2019].
8. MongoDB. (2019). *What Is MongoDB?*. [online] Available at: <https://www.mongodb.com/what-is-mongodb> [Accessed 19 Apr. 2019].
9. Silva, A. (2019). *antoniopresto/react-native-local-mongodb*. [online] GitHub. Available at: <https://github.com/antoniopresto/react-native-local-mongodb> [Accessed 8 Mar. 2019].
10. GitHub. (2019). *Build software better, together*. [online] Available at: <https://www.github.com> [Accessed 19 Feb. 2019].
11. Git-scm.com. (2019). *Git*. [online] Available at: <https://git-scm.com> [Accessed 19 Feb. 2019].
12. Trello.com. (2019). *Trello*. [online] Available at: <https://trello.com/> [Accessed 11 Feb. 2019].
13. Balsamiq.com. (2019). *Balsamiq. Rapid, effective and fun wireframing software. | Balsamiq*. [online] Available at: <https://balsamiq.com/> [Accessed 12 Feb. 2019].
14. Gordon, J. (2017). *decision_tree.ipynb*. [online] GitHub. Available at: https://github.com/random-forests/tutorials/blob/master/decision_tree.ipynb [Accessed 27 Mar. 2019].
15. Color Tool - Material Design. (2019). *Color Tool - Material Design*. [online] Available at: <https://material.io/tools/color/> [Accessed 12 Feb. 2019].
16. Nielsen, J. (1994). *10 Heuristics for User Interface Design: Article by Jakob Nielsen*. [online] Nielsen Norman Group. Available at: <https://www.nngroup.com/articles/ten-usability-heuristics/> [Accessed 24 Apr. 2019].
17. Formidable. (2019). *react-native-app-auth*. [online] Available at: <https://formidable.com/open-source/react-native-app-auth> [Accessed 6 Mar. 2019].
18. Nativebase.io. (2019). *NativeBase | Essential cross-platform UI components for React Native*. [online] Available at: <https://nativebase.io/> [Accessed 26 Feb. 2019].
19. Lekland, J. (2019). *JesperLekland/react-native-svg-charts*. [online] GitHub. Available at: <https://github.com/JesperLekland/react-native-svg-charts> [Accessed 19 Mar. 2019].

20. react-native-community (2019). *react-native-community/react-native-svg*. [online] GitHub. Available at: <https://github.com/react-native-community/react-native-svg> [Accessed 19 Mar. 2019].
21. Reactnavigation.org. (2019). *React Navigation · Routing and navigation for your React Native apps*. [online] Available at: <https://reactnavigation.org> [Accessed 26 Feb. 2019].