Chapter 1

Introduction

This dissertation describes the implementation and evaluation of an activity classifier using accelerometer data captured simutaneously from a smartphone and a smartwatch.

The classifier using data from both sources outperforms a classifier using only smartphone data, and the classifier that uses only smartphone data outperforms a classifier using only smartwatch data.

1.1 Motivation

Wearable devices are set to become the next big technology trend. Wrist-worn wearables, including smartwatches, formed the majority of the 21m wearable devices sold year. Analysts predict the Apple Watch will sell between 20m and 40m in its first nine months [8].

One of the primary appeals of wearables is their ability to sense. Like smartphones before them, smartwatches will enhance the ability to collect data about people. This data is important to consumers, who purchase specialised wearables to measure activity, sleep patterns and caloric intake. The data's research potential is also laudable — Apple's ResearchKit will allow medical researchers to access data about their patients with greater ease than ever before [6].

Accurate activity classification therefore has many academic and commercial applications. To be marketable, activity classification solutions must use current consumer devices. Though rudimentary activity classification is available on Android smartphones, an approach that utilises simutaneous collection from a smartphone and smartwatch has not been investigated in any detail.

This dissertation details the implemenation of accelerometer data collection using current consumer devices (an Android smartphone and Android Wear smartwatch), classifies a user's activities and compares this classification accuracy to using only smartphone data and using only smartwatch data.

1.2 Challenges

This project requires knowledge of a variety of disparate areas in computer science.

Writing software for mobile devices requires knowledge of their paradigms and nuances. Mobile devices are also subject to battery life and computational power constraints and particular care must be taken to build a solution that works in practice. A project that utilises built-in sensors also requires an understanding of the features and limitations of those sensors and good knowledge in the APIs that are provided to access them.

The sensors also output data at a high rate and care must be taken to correctly handle the performance and concurrency issues that may arise. Storage and transfer of large amounts of raw data, especially on a memory-limited device such as a smartwatch, also requires special consideration.

The data processing aspects of the project will require an understanding of digital signal processing, Fourier methods, artificial intelligence and machine learning, and statistics.

1.3 Related Work

Activity classification using accelerometer data from body-mounted devices is an active area of research. I highlight three papers and discuss their similarity to this problem. Summaries of their work are found in Table 1.1.

Bao *et al.* [2] detect physicial activities using five biaxial accelerometers worn on different parts of the body: hip, wrist, ankle, arm and thigh. They find that accuracy is not significantly reduced when using just thigh and wrist accelerometers. Futhermore, recognition rates for thigh and wrist data resulted in the highest recognition accuracy among all pairs of accelerometers, with over a 25% improvement over the best single accelerometer results. This supports the viability of this project, with the improvement of being able to use triaxial accelerometers found in consumer smartphones and smartwatches.

Long *et al.* [5] use a single triaxial accelerometer placed on the wrist and use it to achieve an 80% activity classification accuracy in five activities. However, only 50% of all cycling is correctly classified. Bao *et al.* achieve an accuracy of $> 92\%$ by using thigh and wrist data. This would suggest that wrist data alone is not sufficient to accurately classify certain types of activity. Cycling requires periodic leg motion (pedalling) while the hands and wrists move comparably little. Many of the features of motion used in activity classification require frequency domain analysis, and so data that contains periodic motion will be easier to recognise.

Atallah *et al.* [1] focus on two important facets of accelerometer-based activity classifcation: sensor location and useful features. Much like Bao *et al.* they use seven sensors on the chest, arm, wrist, waist, knee, ankle and ear. Of their analysed features, the averaged entropy over three axes, the mean of the pairwise cross-covariance of axes and the energy of a 0.2 Hz window around the main frequency divided by total energy are all highlight as being highly ranked for distinguishing activities. However, this study neglects to use a decision tree classifier in its classification, recommend by both Bao *et al.* and Long *et al.*

	Bao *et al.* [2]	**Long *et al.* [5]**	**Atallah *et al.* [1]**
Activities	Walking, sitting & relaxing, standing, watching TV, running, stretching, scrubbing, folding laundry, brushing teeth, riding elevator, carrying items, computer work, eating or drinking, reading, bicycling, strength-training, vacuuming, lying down, climbing stairs, riding escalator	Walking, running, cycling, driving, sports	Lying down, preparing food, eating and drinking, socialising, reading, getting dressed, corridor walking, treadmill walking, vacuuming, wiping tables, corridor running, treadmill running, cycling, sitting down and getting up, lying down and getting up
Features	Mean, energy, correlation, entropy	Standard deviation, entropy, orientation variation	Mean, variance, root mean square, entropy, correlation, range, energy, primary frequency, skewness, kurtosis
Classifiers	Decision table, nearest neighbour, decision tree, naive Bayes	Decision tree, principle component analysis, naive Bayes	K-nearest neighbors, naive Bayes
Overall accuracy	84%	80%	N/A

Table 1.1: Prior work on accelerometer-based activity classification

Chapter 2

Preparation

This chapter details the work done before the main implementation of the project was started. It details the devices chosen to implement this project and the reasons for choosing them. It then discusses the existing libaries and APIs available for those devices and for the required data processing. Finally I describe software engineering techniques used.

2.1 Requirements analysis

The aim of the project is to classify activities based on accelerometer recordings from a consumer smartwatch and smartphone, and evaluate to what extent the smartwatch is better at helping to classify activities. The requirements to accomplish this can be split into data collection and data processing requirements.

Data collection requirements

1. access tri-axial readings from accelerometer on both the smartwatch and the smartphone;
2. store this accelerometer data temporarily on the internal memory of each device using suitable data structures;
3. transmit this data from the smartwatch to the smartphone using a suitable protocol;
4. store the data permanently on the smartphone, to enable transfer to the computer.

Data processing requirements

1. parse the data into a manipulatable format;
2. preprocess the data, including filtering and splitting into fixed-length bins;
3. extract features from each bin;
4. train classifier(s) on the extracted features;
5. test classifier and record evaluation statistics.

The remainder of this chapter describes work done to ensure these requirements could be fulfilled.

2.2 Introduction to signal processing

The output from any accelerometer is a time-series representing its acceleration. Effectively extracting information from this time-series is central to the success of this project. Knowledge of signal processing is therefore critical.

It is essential to capture as much of the movement as possible. Conversion from continuous physical acceleration to a discrete time-series requires sampling. The Nyquist-Shannon sampling theorem states that a signal can be exactly reconstructed from its samples if the sample rate is greater than twice the highest frequency of the signal.

The highest frequency of a physical activity is not well defined. The activities I hope to classify will vary in their periodicity. Some, like walking, will be very periodic, while others will have no period at all. Considering common period activities like cycling and walking, I anticipate that the frequencies that best describe movement will be present in the 0–5Hz range. Graphs of accelerometer readings for each of the activities I attempt to classify are presented in section **??**.

Frequency domain analysis

Much of the analysis of the accelerometer readings will be done in the frequency domain. A time domain signal can be converted into the frequency domain using a Fourier transform.

The discrete Fourier transform of a sequence of N complex numbers $f_0, f_1, \ldots, f_{N-1}$ is the sequence F_k, defined by:

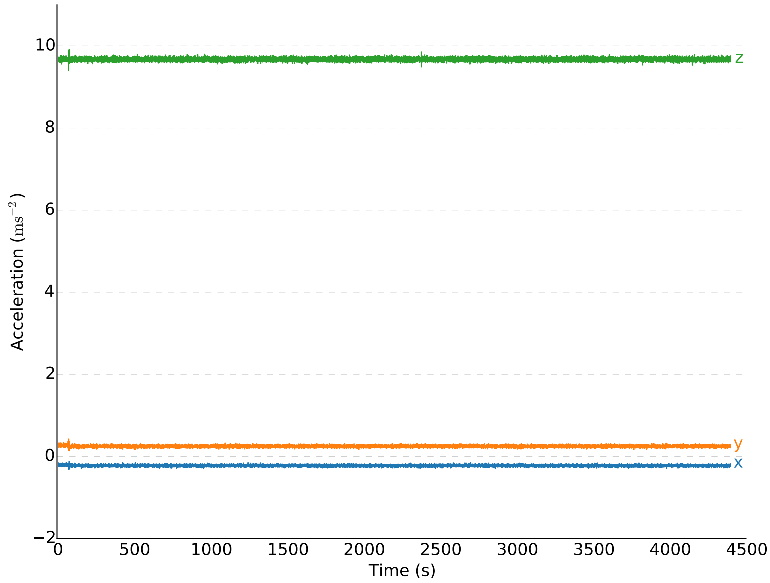

Figure 2.1: The x, y and z axis readings from an hour long accelerometer recording of the phone laying flat on a table. The readings contain noise.

$$F_k = \sum_{n=0}^{N-1} f_n \cdot \mathrm{e}^{-2\pi \mathrm{i} kn/N}$$

The power spectral density of a signal describes how power is distributed over different frequencies. One method of estimating the power spectral density is to take the square of the absolute value of the Fourier transform component:

$$PSD_k = \|F_k\|^2$$

Noise and filtering

The readings from the accelerometer are subject to noise, exhibited in figure 2.1, which plots readings from the x, y, and z axes during an hour long recording with the phone laying flat on a table.

Figure 2.2 plots the distribution of the magnitude of the acceleration, where the magnitude $\|\mathbf{x}\| = \sqrt{x^2 + y^2 + z^2}$. The magnitude, which should be a constant

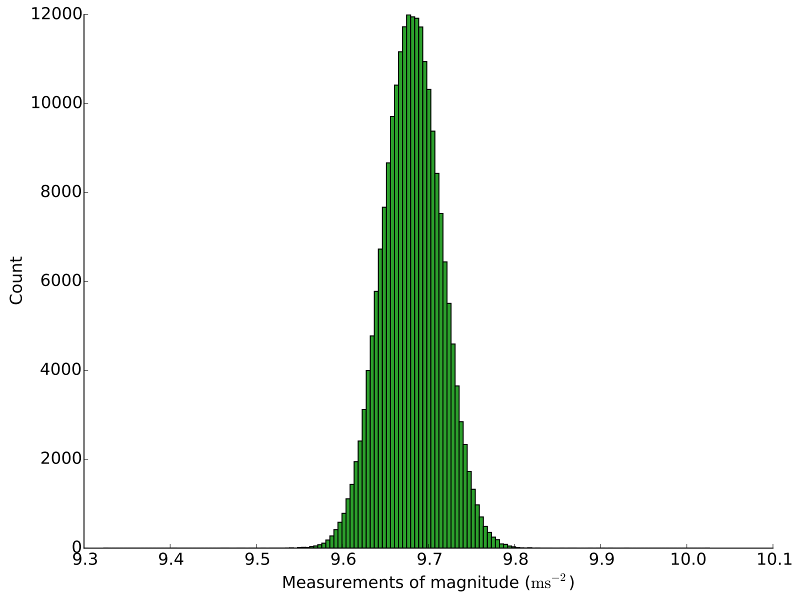

Figure 2.2: Histogram of the magnitude $\|\mathbf{x}\| = \sqrt{x^2 + y^2 + z^2}$ of the data shown in Figure 2.1. The magnitude should measure $\mathrm{g} = 9.81\mathrm{m\,s}^{-2}$. The noise means the accelerometer data is imprecise. The mean of the data is less than g, which indicates the recording is also inaccurate.

$\mathrm{g} \approx 9.81\mathrm{m\,s}^{-2}$, is subject to normally distributed noise.

Figure 2.3 gives a normal probability plot of the same magnitude data. Points on a normal probability plot should form a straight line if they are normally distributed. The straight line of best fit exhibits a coefficient of determination, R^2, which is very close to 1 and therefore it is very likely that the noise is normally distributed.

Noise can be reduced with the application of a low-pass filter. A low-pass filter attenuates signals with a higher frequency than some cutoff, such as the noise exhibited in the signal.

2.3 Hardware devices

The success of this project depends partly on correct selection and understanding of the devices used to collect data. The devices are both required to contain accelerometers accessible to developers.

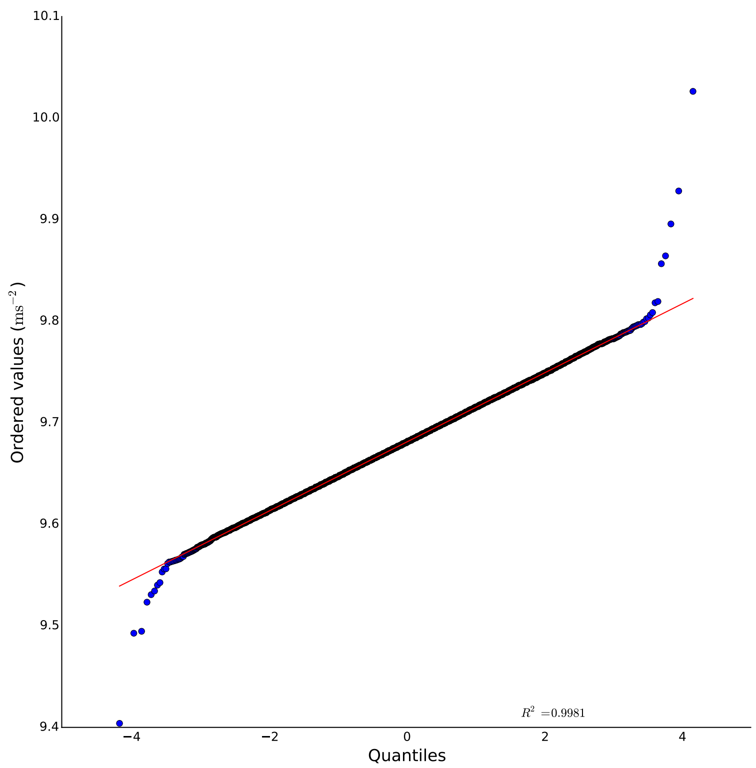

Figure 2.3: A normal probability plot of the magnitude $\|\mathbf{x}\| = \sqrt{x^2 + y^2 + z^2}$ of the data shown in Figure 2.1. Data that is normally distributed will form a straight line when plotted in this way. This data is very likely to be normally distributed.

Android devices were chosen as Android Wear was the most mature platform for developing with wearable devices at the time. It runs on the widest variety of devices and provides developer access to its sensors.

2.3.1 Smartphone

The smartphone chosen for development was the Google Nexus 5. Smartphone technology has advanced to the point that many Android smartphones are homogeneous with respect to this project — they all continue sufficient processing power, internal memory and an accerometer capable of recording data.

The Nexus 5 contains a tri-axial accelerometer capable of recording measurements $\pm 2\mathrm{g}$ on each axis, where $\mathrm{g} \approx 9.81\,\mathrm{m\,s^{-2}}$.

2.3.2 Smartwatch

The smartwatch chosen for development was the Samsung Galaxy Gear Live, running Android Wear. It pairs to any device running Android 4.4 or higher and communicates over Bluetooth.

Wearable devices not running Android typically run either Tizen, an open-source but not widely adopted operating system — such as the Samsung Galaxy Gear 2 — or a proprietary operating system that does not allow access to the raw accelerometer data, for example the Jawbone Up.

There is more differentiation in smartwatches as there is in smartphones, with them varying not just in screen size but also in screen format (round or rectangular), battery life, charging facilities and sensors. Table 2.1 presents an overview of possible smartwatch devices.

2.4 Libraries and APIs

This project makes use of existing libraries and APIs for the data collection, data handling and classification aspects of the project. Preliminary investigation into each of these areas was conducted.

Device	**Samsung Galaxy Gear Live**	**Samsung Galaxy Gear 2**	**LG G Watch**	**Sony Smartwatch 3**
Operating System	Android Wear	Tizen	Android Wear	Android Wear
Processor	1.2 GHz single-core Qualcomm Snapdragon 400	1.0 GHz dual-core Exynos 3250	1.2 GHz single-core Qualcomm Snapdragon 400	1.2 GHz quad-core ARM A7
Memory	512 MB RAM	512 MB RAM	512 MB RAM	512 MB RAM
Storage	4 GB	4 GB	4 GB	4 GB
Sensors	Touchscreen, Accelerometer, Gyroscope, Compass, Heart Rate Monitor	Touchscreen, Accelerometer, Gyroscope, Heart Rate Sensor, 2 MP Camera	Touchscreen, Accelerometer, Gyroscope, Compass	Touchscreen, Accelerometer, Gyroscope, Compass
Radios	Bluetooth 4.0 Low Energy	Bluetooth 4.0 Low Energy	Bluetooth 4.0 Low Energy	Bluetooth 4.0 Low Energy, GPS, NFC, Wi-Fi
Battery	300 mAh	300 mAh	400 mAh	420 mAh
Notes		Pairs only with Samsung devices		

Table 2.1: An overview of possible smartwatch devices. The Samsung Galaxy Gear Live was the device eventually chosen.

2.4.1 Android Sensor API

The Android platform Sensor API is implemented using a publisher-subscriber model. Listeners to a particular sensor must be registered and must implement an `onSensorChanged()` method. The `onSensorChanged()` method is called whenever the sensor reports a new value. A `SensorEvent` object is provided, containing a timestamp at which the data was reported together with the new data.

The rate at which `onSensorChanged()` is called is user-'suggested'; though it can be specified by the user, it can also be altered by the Android system. In practice, this means that the difference in timestamps not constant but is approximately equal to the specified delay. A histogram of timestamp differences for a particular 1 hour recording is given in figure 2.4.

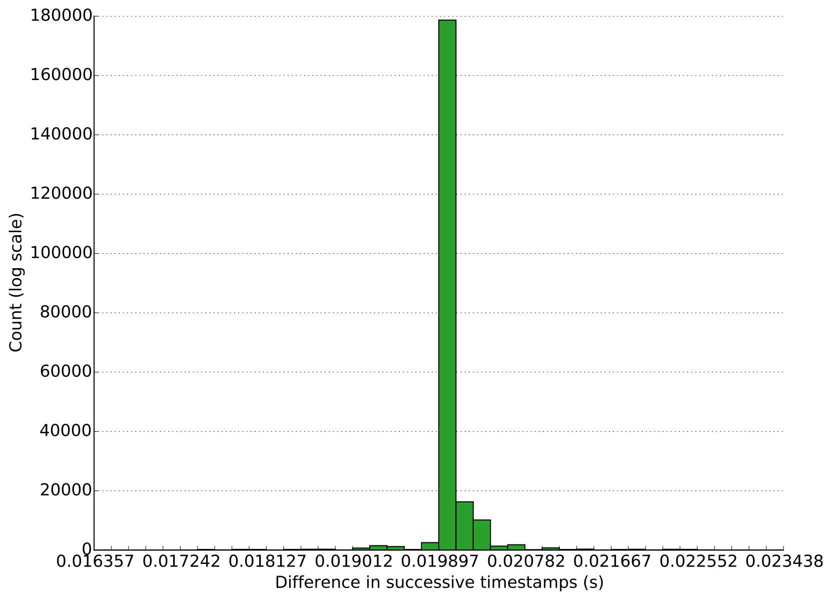

Figure 2.4: Histogram of the differences in successive timestamps of a one hour accelerometer recording from the Nexus 5 smartphone. The sample rate was set to 50 Hz. 0.02002s accounted for 75% of the differences. Thus the actual sample rate is approximately the user-suggested sample rate.

Android provides both acceleration and linear acceration sensors, related by

$$\text{acceleration} = \text{linear acceleration} + \text{gravity}$$

They each provide a timestamp represented as a long and three float values representing the acceleration of each axis in $\mathrm{m\,s^{-2}}$ at that timestamp. Table 2.2 gives

Timestamp	X acceleration	Y acceleration	Z acceleration
ns	$\mathrm{m\,s^{-2}}$	$\mathrm{m\,s^{-2}}$	$\mathrm{m\,s^{-2}}$
Long	Float	Float	Float
2 bytes	1 byte	1 byte	1 byte

Table 2.2: Data from the accelerometer sensor provided to the `onSensorChanged()` method.

a graphical representation of the data returned.

Curiously, the timestamp returned as part of the data is documented only as "The time in nanosecond at which the event happened" [4]. Futher exploration reveals that the timestamp is not defined against any particular zero-base, but rather the time since the device was powered on [3, 7]. The implication of this for the project is that while the timestamp can be relied on for intervals between measurements, it cannot be used between different sets of recordings or across devices.

2.4.2 ES Sensor Manager

I explored this but didn't end up using it. Should I write about what it is and why I didn't end up using it?

2.4.3 Android Wear Data API

As discussed in section 2.3.2, the only radio present in the Samsung Galaxy Gear Live is Bluetooth. To transfer any recorded data from the watch, it must first be transferred to the paired smartphone. The Android Wearable Data Layer API allows communication between Android handheld and wearable devices. It provides three methods of communication between devices:

- **Data items** provide data storage with automatic syncing;
- **Messages** are good for remote procedure calls but do not carry data;
- **Asset objects** for sending binary blobs of data.

The data layer synchronises data between the handheld and wearable. To do so, the Wearable Data Layer API requires the registration of a listener service, much like the Sensor API. The listener service listens for data layer events, such as the creation of asset objects or when messages are received.

2.5 Choice of tools

2.5.1 Programming languages

Java was chosen as it is the native programming language used on Android. Although it is possible to write code for Android in programming languages other than Java, for example by using the Java Native Interface, doing so would not benefit the project. Java is taught in Part 1A and Part 1B of the Computer Science Tripos. The Android SDK builds on principles covered in the course but is complicated by having to manage interactions with the Android operating system.

XML is Android's standard markup language. All user-interface components are written in XML. The project includes a user interface to configure and control the recording of data.

Python 3.4 was chosen as the data processing language due to its easy of use and the strength of its data processing, signal processing and machine learning libraries:

- **NumPy** is a scientific computing library and the basis for the other three libraries below.
- **Pandas** provides extensions to NumPy that enable easier processing of time-series data.
- **SciPy** provides signal processing tools and other statistical features.
- **Scikit Learn** provides machine learning classifiers and utilities to work with them.

All of NumPy, SciPy, Pandas and Scikit Learn are open-source and licensed under the BSD license.

2.5.2 Development Environment

Two powerful IDEs, Android Studio and PyCharm were used for the development of the Android app and the Python data pipeline respectively. Android Studio is available for free from Google, while PyCharm is provided free for educational use by JetBrains. Both include advanced debuggers.

Though the Android SDK contains a device emulator, it runs slowly and cannot simulate sensors. Developing the Android apps is therefore done by connecting them to a computer and running new versions of the code. This also enables access to the device's logs from the development environment. I made extensive use of logging to determine that the program was executing as expected.

2.6 Software engineering techniques

2.6.1 Development methodologies

I used a combination of development methodologies for the project. The data collection apps were developed using a waterfall methodology, while the data processing was developed using an Agile methodology.

Waterfall models are excellent when the end goals of the project are known and can be well specified. The goal of the data collection apps can be easily stated: write apps for the smartphone and smartwatch that will allow user collection of accelerometer data.

The data processing and machine learning elements of the project required an Agile methodology. The goal here is less well defined — classify activities with the greatest accuracy — and the implementation to achieve the goal is far more experimental.

2.6.2 Version Control and Backups

I used three separate Git repositories for the data collection code, the data processing code and the dissertation respectively. The Git repositiories were synced to GitHub at each commit. Version control allowed me to follow a *implement–test–commit* pattern when writing code.

GitHub also served as one method of backup. Each GitHub repository is publicly accessible such that I can continue implementation even if my primary development computer crashed and I was also locked out of my GitHub account. In addition, I backed up periodically to Dropbox and to an external hard drive. The external hard drive backup retained old copies of files when they are updated. This gives four replications of my entire project, with two of these able to access previous versions of the code.

2.7 Summary

In this section I presented:

- an overview of digital signal processing;
- information on the smartphone and smartwatch used;
- details of key APIs used including the Android Sensor API and the Android Wear Data API;
- development tools and software engineering techniques.

Chapter 3

Implementation

3.1 Data collection

This section contains details of the components built to access the accelerometer data and transfer it to a computer.

Because both the smartwatch and the smartphone both run Android, it is possible to create components that are shared between the devices, reducing the amount of code I am required to write and to test, resulting in less redundancy, less complexity and ultimately a more reliable implementation. Both the `AccelerometerListenerService` and the `AccelerometerDataBlob` are shared between both devices.

3.1.1 Accessing the accelerometer

The `AccelerometerListenerService` is responsible for receiving readings from the accelerometer and delivering them to the data structure responsible for storage.

As described in Section 2.4.1, the Sensor API utilises a listener methodology. It is required to create and register a listener that implements `onSensorChanged()`.

Performance considerations

Because the accelerometer can update its values at a rate of over 50Hz, it is vital that any implementation of `onSensorChanged()` be non-blocking and ideally be very quick to execute. Any expensive computation or IO operation has to be moved to a separate thread.

If the execution of `onSensorChanged()` takes longer than $\frac{1}{\text{sample}-\text{rate}}$, requests for `onSensorChanged()` will queue and eventually lead to exhaustion of memory or dropping of data.

For this reason the data structure used, discussed in Section 3.1.2, is very lightweight and `onSensorChanged()` is only responsible for passing data to it.

Concurrency considerations

Because `onSensorChanged()` can be called at such a high rate, it is possible that new calls to the method can be made while previous calls are still executing. Data corruption could result from improper handling of asynchronicity.

The documentation for the Sensor API is not explicit about whether calls to `onSensorChanged()` queue on the same thread or whether they can be dispatched asynchronously. For this reason, the `AccelerometerListenerService` was designed to be thread-safe by using Java concurrency primatives.

Power consumption considerations

Recording data from the accelerometer can be computationally expensive. This increase in computational overhead translates to an increase in power consumption in battery powered devices such as the smartphone and the smartwatch. It is for this reason that care should be taken to minimise power usage where possible while still collecting all the required data.

One tradeoff that had to be made was between collection strategies. One strategy is to record data at a specified sample rate from when the recording is turned on until it is turned off. An alternative strategy is to record a window of data at set intervals and sleep the remainder of the time. For example, one might set the accelerometer to record 10 seconds of data every 50 seconds.

Though this strategy saves battery power as the device turns off the accelerometer between recordings, a continous recording approach was taken in this project in order to have as much data as possible with which to train. In addition, the battery life was not severly impeded by the continous recording approach.

Typically, Android will power off the display and later the CPU after a period of user-inactivity. Powering off the CPU means that the device will stop recording accelerometer data, and so it is required to maintain a wake-lock which keeps the CPU from powering off. It is also important to remember to release the wake-lock once accelerometer recording is complete. Otherwise, the device's CPU will remain on even when the device appears to be on standby, using battery.

Sampling rate

In ideal conditions, it would be sensible to sample at as fast a rate as possible: the resultant data can always be downsampled afterwards if it is not required. As per the Nyquist-Shannon sampling theory, discussed in Section 2.2, our sample rate should be greater than twice the highest frequency of the signal. Because it isn't possible to know what the highest frequency is going to be, it would be reasonable to sample at a far higher rate.

However, picking a very fast sample rate in this context has two potential downsides: battery life drain and the size of resultant data. I investigated whether either battery life or the size of the resulting data would be a limiting factor of sample rate.

The impact on power consumption of increasing the sample rate was negliable. One possible reason for this is that sampling using the accelerometer at all has high fixed costs and increasing the sample rate has lower marginal costs.

Recall from Table 2.2 that each measurement has a total size of 20 bytes. At a sample rate of 50Hz, we produce data at approximately 1 KBps or 3.6 MB per hour. The most memory-constrained device is the smartwatch, which only has 512 MB of RAM but 4 GB of internal storage. A data structure that stores the accelerometer data to the internal storage rather than to memory is required, but a sample rate of 50Hz produces a storable amount of data on reasonable-length activity recording.

Another potential concern regarding data size is the transfer from the smartwatch to the smartphone. The only connection available is Bluetooth. The Bluetooth connection empircally has a maximum transfer rate of no more than 150 KBps, meaning an hour of activity data will take approximately 30 seconds to transfer.

3.1.2 Storing accelerometer data

The data structure to hold the accelerometer data is required to be:

	DataItem	**Asset**
Advantages	• no separate data fetching step • simpler, more reliable receiver code • negligable transmission time	• no hard size limit • can create an Asset from a File without storing it in memory
Disadvantages	• 100 KB size limit • have to insert byte arrays	• some constructors don't seem to work • transmission of large files takes a noticable amount of time

Table 3.1: Advantages and disadvantages of using the `DataItem` and `Asset` to transmit data from the smartwatch to the smartphone.

- **fast** because it will be accessed many times per second and cannot block;
- **on-disk** rather than in-memory, because the smartwatch may not have enough free memory to store all the accelerometer data for lengthy recordings;
- **thread-safe** as it is unclear whether calls to `onSensorChanged()` are queued or concurrent.

The data structure decided on was a temporary random-access file with buffered writing. The data is written as bytes through an output buffer. The output buffer is maintained in memory and is flushed when it reaches capacity. The capacity of the output buffer was set to 20000 bytes as data is only written in multiples of 20 bytes and the smartwatch is comfortably able to keep 20 kb in memory. This equates to data being saved to disk approximately every 20 seconds.

3.1.3 Transmitting accelerometer data

The accelerometer data has to be transmitted from the smartwatch to the smartphone before it can be transferred to a computer. As discussed in Section 2.4.3, there are two methods to transfer data between the smartwatch and the smartphone: a DataItem and an Asset. Their advantages and disadvantages with respect to this project are highlighted in Table 3.1.

Because the DataItem has a 100 KB limit, an alternate transmission and storage system would have to have been built, where the smartwatch collects 100 KB of data and sends that to the smartphone while it continues to record. It is then reassembled at the smartphone receiver.

I consider this solution inferior to the Asset implementation, which allows transmission of any size of data.

3.1.4 User interface

Smartphone

Smartwatch

3.2 Activities and data collection method

3.3 Data processing

3.3.1 Importing

3.3.2 Preprocessing

3.3.3 Grouping and binning

3.3.4 Feature extraction

3.3.5 Machine learning

3.4 Summary

Bibliography

[1] Louis Atallah et al. “Sensor placement for activity detection using wearable accelerometers”. In: *Body Sensor Networks (BSN), 2010 International Conference on*. IEEE. 2010, pp. 24–29.

[2] Ling Bao and Stephen S Intille. “Activity recognition from user-annotated acceleration data”. In: *Pervasive computing*. Springer, 2004, pp. 1–17.

[3] *Documentation Enhancement: SensorEvent timestamp*. 26th Apr. 2013. URL: `https://code.google.com/p/android/issues/detail?id=7981` (visited on 28/03/2015).

[4] Google. *SensorEvent — Android Developers*. URL: `http://developer.android.com/reference/android/hardware/SensorEvent.html` (visited on 28/03/2015).

[5] Xi Long, Bin Yin and Ronald M Aarts. “Single-accelerometer-based daily physical activity classification”. In: *Engineering in Medicine and Biology Society, 2009. EMBC 2009. Annual International Conference of the IEEE*. IEEE. 2009, pp. 6107–6110.

[6] *ResearchKit for Developers*. 23rd Mar. 2015. URL: `https://developer.apple.com/researchkit/`.

[7] *SensorEvent timestamp field incorrectly populated on Nexus 4 devices*. 13th June 2013. URL: `https://code.google.com/p/android/issues/detail?id=56561` (visited on 28/03/2015).

[8] “Wearable technology: The wear, why and how”. In: *The Economist* (14th Mar. 2015). URL: `http://www.economist.com/news/business/21646225-smartwatches-and-other-wearable-devices-become-mainstream-products-will-take-more`.