THE UNIVERSITY OF NEW SOUTH WALES

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Cough Detection Using Mobile Phones

Ronald Huynh (3253669)

Bachelor of Engineering

(Computer Engineering)

Submission Date: 16 October 2012

Supervisor: Dr. Salil Kanhere
Assessor: Mahbub Hassan

Abstract

Despite being in a technologically advanced society, coughing is still a major problem in our society. Many diseases stem from coughing such as lung cancer (from smoking), asthma, pneumonia, bronchitis etc... The severity of such diseases could be reduced if coughing could be detected accurately by a small pocket device.

Previous efforts in cough detection have used a microphone which is fairly accurate but does not perform well in noisy environments and privacy issues are also a concern. This thesis explores the viability of using a built-in accelerometer to detect coughing that mitigate issues of audio based cough detection.

An android smartphone was programmed to collect data for coughing and non-coughing scenarios and analysed in WEKA. Initial results were promising with over 90% accuracy rates with a 10-fold cross validation and a total of 7 features.

Acknowledgments

Firstly, I would like to thank my supervisor Salil Kanhere for providing me with guidance whilst undertaking this thesis. He has given me invaluable advice especially during the data collection stage. A special mention also goes to Sara Khalifa; a PhD student who has taught me how to use WEKA and without her guidance this thesis would not have been possible.

Finally, I would like to thank the people who helped me collect accelerometer data such as Dougy Lee and Jeff Tang when time was limited. The data they collected has saved me the embarrassment of doing it in front of the entire university.

Contents

Chapter 1

Introduction

1.1 Motivation

Coughing nowadays can be a serious issue in our society. Coughing usually is an indication of a more serious issue especially if the cough is ongoing. One example of serious illness is smoker's cough and pneumonia. Such diseases have adverse effects on the quality of life for patients. Whooping cough is another disease which also has adverse effects. Recently whooping cough has re-manifested itself in California [6]. The reason for this is because the regular whooping cough virus has mutated itself where modern medicines are ineffective against this new strain.

It is for this reason that cough detection is vital to our everyday lives, if a small device could be developed to detect how often a patient coughed in a time frame, it would be an invaluable resource for doctors as they don't have to ask the patient about their coughing history; instead they can observe data coming from a small mobile device. Studies have also shown that over 20% of doctor consults are for coughing alone.

1.2 Goals

The overall goal of this thesis is to provide an insight into whether it is feasible to detect cough using a standard tri-axial accelerometer on a mobile device. Standard methods discussed in Chapter 2 were not considered because they have have previously been extensively researched. The main focus for this thesis is to be able to achieve similar detection rates compared to other solutions despite having parasitically induced movement from the activity of the user. Chapter 6 discusses the results of testing with different features and their corresponding accuracy rates. Listed below are the requirements that need to be met for the application which is to be developed for this thesis:

- **Ease of use for target audience**
  Since this application does not have a particular audience, it would be beneficial to cater for all users including children and the elderly. The application should contain a minimalist interface and tested with a variety of people.

- **Easy to debug software**
  Developing applications for a mobile device is radically different to a personal computer. Mobile devices are limited in power and battery constraints means the app must be free of bugs or serious errors. Albeit being a fairly stable environment (mobile devices), bugs tend to occur during the actual usage which means connections to an external debugger with stacktrace or logging functions will help tremendously with debugging.

- **Programming SDK freely available on the internet**
  Nowadays many SDKs (Software Development Kits) are available on the internet. This means that development can be done using corresponding API's and IDE's (Integrated Development Environment) easily and efficiently. Many SDK's also offer virtual devices similar to running VMWare on a desktop computer.

- **Scalability**
  Of course with this application scalability must also be considered. Nowadays the market is flooded with a myriad of versions of the same product. Catering for all versions would be of benefit for the community and the penetration into the application market will be high.

- **Accuracy**
  Accuracy must be of paramount concern. The minimum standard for accuracy should be compared to an audio based solution. It is obvious the chances of detecting coughing during certain activities such as running and extreme sports is very minimal. This means that the scope of this thesis is reduced to common everyday activities such as walking, standing and sitting in front of a computer.

Chapter 2

Background

This chapter introduces the background material for this thesis including the definition and physiology of coughing, current cough detection mechanisms and finally an evaluation of previous work.

2.1 Definition of coughing

According to Wikipedia [7] cough is defined as:
"*A sudden and often repetitively occurring reflex which helps to clear the large breathing passages from secretions, irritants, foreign particles and microbes*"
Depending on the severity of the cough, coughs are placed in the following categories:

- **Acute**
  A cough is classified as *Acute* when the coughing lasts for less than three weeks e.g. common cold, hayfever.
- **Subacute**
  Similarly a cough is classified as *Subacute* when coughing lasts between three and eight weeks.
- **Chronic**
  Finally a cough is considered to be *chronic* if coughing lasts for over eight weeks.

2.2 The cough reflex

The cough reflex is an important component in the physiology of coughing. The cough reflex is triggered by the introduction of foreign particles into the pulmonary irritant receptors located in the respiratory tract. As foreign particles land on these receptors, an impulse is sent from the area to the brain, triggering the body to cough in order to remove these foreign particles. A list of stages that the cough reflex goes through are outlined below:

1. The diaphragm contracts, causing negative pressure surrounding the lungs

2. The negative pressure causes the patient to take a deeper breath than usual
3. The glottis closes and the muscles controlling the vocal cords contract, causing the larynx to close
4. Abdominal muscles contract causing the diaphragm and other expiratory muscles to contract to increase air pressure in the lungs
5. As the abdominal muscles contract, they force air out the larynx and glottis at speeds of over 100 miles per hour.
6. Internal bronchial structures collapse to form slits for foreign particles to exit the lung cavity

2.3 Current cough detection mechanisms

2.3.1 Self Analysis

The simplest and least intrusive method of detecting cough is self evaluation. There are two ways of doing so. One way is to have an external party (e.g. scientists) in a controlled environment counting coughs manually. There are obvious inherent flaws with this method. One would be the necessity to observe the patient for extended periods of time, another would be privacy issues, e.g. if a patient has a phone call with an important company client and the conversation is overheard. This method is not only time consuming, but is costly for scientists to sit down and observe a person coughing, a hardly practical solution nowadays.
An alternative to this is if the patient carries a diary around to log down the frequency of the cough or alternatively carry a simple device in which when they cough, a button is pressed to count the cough. Again this has one flaw as mentioned above - there is the necessity for the patient to be alert when they are coughing in order to log it down.

2.3.2 Audio Based Cough Detection

Audio based cough detection is not a new concept. There have been many studies and experiments conducted to determine the effectiveness of audio based cough detection. Drugman et al. [8] did a study on features in an audio signal in which they could detect coughing from an audio signal recorded from an MP3 player. A patient is placed in a room where an MP3 player is recording ambient noise for a predetermined amount of time. The recordings were then passed through a computer system where 3 different machine learning classifiers were run on the data set. The researchers looked at 3 categories, they were the features describing the spectral contents Zheng et al. [9], measures of noise G. Peeters [10], and prosody-related features Kawahara et al. [11]. The sample audio file was then divided into 2 sections, cough and non-cough.

For the machine learning component, Drugman et al. [8] used 3 different algorithms - ANN (Artificial neural networks), GMM (Gaussian mixture model) and SVM (Support vector machines). They found that all 3 algorithms had varying degrees of success, for example ANN achieved a 94.7% TPR (True positive rate) and 5.5% FPR (False positive rate) with 64 neurons in the hidden layer. GMM achieved the highest accuracy rate at 95.20% TPR and a percentage of 5.73% for FPR involving 16 gaussians and 20 features. Finally SVM produced the worst accuracy rates of the three; 81.87% TPR and an FPR of 0.32%. But it is noteworthy that SVM produced the least false positives where as the other two produced in the neighbourhood of 5.5%. Judging by the results of this article, it is easy to conclude that GMM combined with an audio based cough detection is a good solution. But what if excessive ambient noise is present?

Barry et al. [12] also conducted studies on cough detection using audio signals. The premise of their research stemmed from thresholds of different sound sources. For example, talking quietly exhibits such sound levels where as a jet taking off exhibits a huge blast of decibels. Periods of silence were omitted from the calculation via DSP (Digital signal processing) techniques and passed into a trained artificial neural network. The ANN was trained to classify each signal as a cough or non-cough. The labelling of a cough from an audio signal was determined by a sudden explosive sound followed by a subsequent drop in sound pressure. To combat background noise, σ background at time t is calculated as the minimum σ signal between the start of the window, t - (Δ t background) and the end of the window, t + (Δ t background). Sound events are thus detected when σ signal for a particular window exceeds the threshold value, thresh-peak, multiplied by σ background for that window. Although this procedure means that sound sensitivity varies to a certain extent, it allows for peak detection in noisy backgrounds.

2.4 Issues identified in previous work

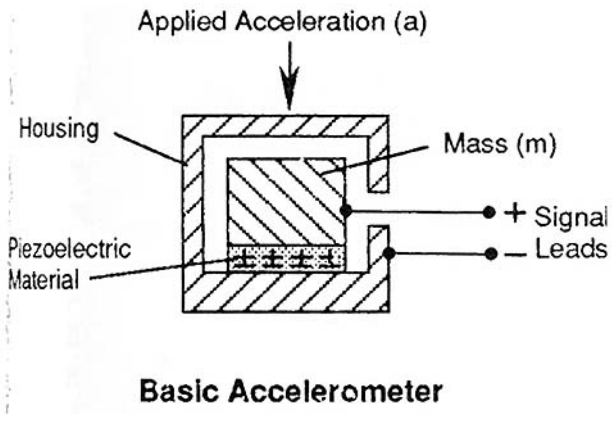
Despite having a novel way of detecting coughing, there are still unsolved problems. One as mentioned before is the manual verification of coughing events. The study still required trained professionals to be present verifying the coughs. Secondly, the tests were performed in a closed laboratory environment where there is obvious inherent bias in these experiments because it has not been tested in varying locations. A location which will surely fail would be for example in a club environment where very loud music is being played.

2.5 Activity Detection using Accelerometers

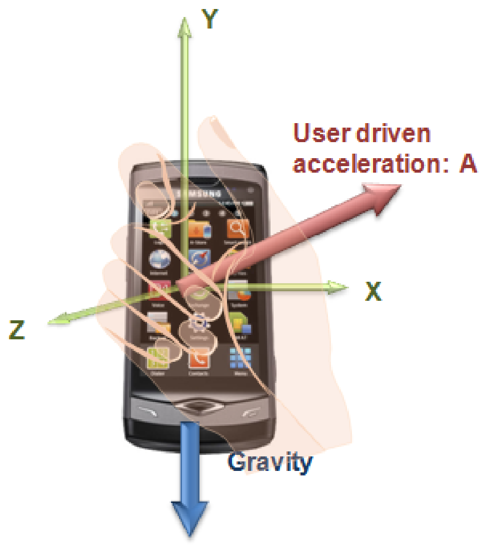
Before we look at cough detection using accelerometers, we must first examine the developments going on in accelerometer activity detection and give an overview of how an accelerometer works. For this section we will be focusing on accelerometers in mobile devices such as a mobile phone or small portable computer.

2.5.1 How an accelerometer works

A regular 3-D Tri-Axial accelerometer can be found in the majority of today's smartphones. An accelerometer is a small electro-mechanical device which measures the total acceleration with respect to a set of x-y-z axes relative to the earth's x-y-z axes. An accelerometer can measure static or dynamic forces such as a stationary phone on a table or on a person holding the phone while running. One method an accelerometer works is via the *piezo-electric effect* - inside the device there are microscopic crystal particles that are stressed when a force is applied to them, which causes a voltage to be generated. From this voltage graphs of acceleration can be produced to measure total displacement in any direction (x, y, z). The second way an accelerometer could be produced is measuring changes to capacitance if 2 microstructures are placed together. Changes in the structures causes a shift in capacitance which can be measured. The capacitance is then converted to a voltage and plotted on a graph.



(a) Basic piezo-electric Accelerometer



(b) Axes for a mobile Phone

Figure 2.1: Piezo-electric accelerometer and mobile phone axes

2.5.2 Uses of accelerometers

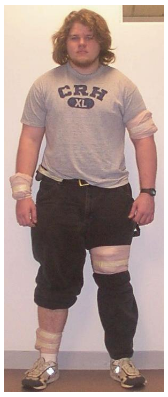
Since accelerometers measure acceleration in 3 dimensions, an accelerometer is especially useful in detecting the orientation of an object. For example if a car is driving up an incline an accelerometer can detect that, if an object is free falling that can also be detected. The question is, therefore, how accurate is the detection of activities using a simple tri-axial accelerometer?

2.5.3 Bao and Intille [1]

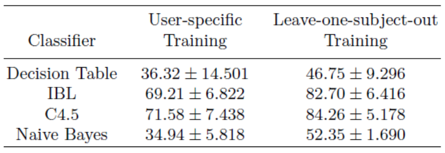
A study by Bao and Intille [1] was conducted to determine whether it was possible to classify common everyday activities by using 5 bi-axial accelerometers strapped to various parts of the human body. With the aforementioned limitations of testing in a laboratory environment, Bao and Intille [1] did not restrict the participants to a laboratory environment. Instead the participants were told to carry out a specific activity in an uncontrolled environment to test the effectiveness of their algorithms. This was done in order to combat the stable consistent values in a laboratory environment; in actual testing the signal is jagged and irregular. During the study they also ran into the problem of self-evaluation - in order to generate user driven results, the participants had to label at what point in time they were participating in which activity. If observers were to be hired it would have been too costly and does not scale well with a larger cohort of users.

Bao and Intille [1] chose 20 common daily activities to test. Each participant was told to label the start and stop duration of each activity and to write down any relevant notes on that activity. Participants were to complete each activity at their own pace and without external party supervision. With the raw data they passed it through the following machine learning techniques: Decision tables, instance-based learning (IBL or nearest neighbour), C4.5 Decision trees (J48) and naive Bayes classifiers. All the machine learning techniques were available in the WEKA data mining suite.

From their results, it was found that decision making algorithms were the most accurate namely the C4.5 decision tree with 89.30% accuracy for sitting, standing and locomotion. The C4.5 decision tree identified sitting down as having a 1G downward acceleration with low energy on the hip and arm. But the results were an average of all the values, this means that any outliers or low accuracy activities were not accounted for. An example was that a C4.5 decision tree had difficulty distinguishing between stretching and riding an elevator with an accuracy below 50%. Other activities such as folding laundry and stretching were often confused because they involved the same muscles and have similar acceleration profiles.



(a) Accelerometers attached to body



Classifier	User-specific Training	Leave-one-subject-out Training
Decision Table	36.32 ± 14.501	46.75 ± 9.296
IBL	69.21 ± 6.822	82.70 ± 6.416
C4.5	71.58 ± 7.438	84.26 ± 5.178
Naive Bayes	34.94 ± 5.818	52.35 ± 1.690

(b) Results of machine learning

Figure 2.2: Accelerometers attached and results of study

At the end of the study, Bao and Intille [1] had a combined activity detection rate of 84.26%. This was significant because the participants were not subjected to controlled laboratory environments, instead they were allowed to carry out tasks at their own will. Before a conclusion is drawn, it must be emphasised that the researchers were using 5 bi-axial accelerometers and an accuracy rate of 84.26% was achieved. Had this been a tri-axial accelerometer the results may have improved compared to a bi-axial accelerometer. If such activity recognition accuracies can be achieved with a simple bi-axial accelerometer, then using a tri-axial accelerometer to detect coughing is definitely a viable idea.

2.5.4 WISDM (Wireless sensing and Data mining) [2]

Another study conducted by Kwapisz et al.[2] also looked into the viability of using accelerometers to determine what activity the user is doing. The experiments were conducted on an android mobile phone with a standard tri-axial accelerometer running at the default polling rate of 20hz. The experiment involved 29 users carrying out a variety of activities with the android mobile phone in the pocket. They developed a custom android application with a GUI where the participant records their name, start/stop times and whether other sensors such as GPS are used. Data was verified by one of the WISDM team members for accuracy.

They split the raw accelerometer data into 10 second chunks as they though 10 seconds was an adequate amount of time to collect data. Data obtained were values for each axis (x, y and z) and the following table outlines the attributes they were looking for.

Feature	**Explanation**
Average	Average acceleration (for each axis).
Standard Deviation	S.D for each axis
Average absolute Difference	Average absolute difference between the value of each of the 200 readings within the ED and the mean value over those 200 values(for each axis)
Average resultant acceleration	The Square root of the (Sum of the squares of each axis) $\sqrt{x^2 + y^2 + z^2}$
Time between peaks	Time in milliseconds between peaks of the wave
Binned distribution	The range of values for each axis (maximum - minimum) divided into 10 equal sized bins then recorded what fraction of the 200 values fell within each of the bins.

Table 2.1: Features Kwapisz et al.[2] looked for in their study

The researchers focused on 6 activities namely sitting, standing, jogging, walking, ascending/descending stairs. Participants were to do each activity for a couple of minutes except standing and sitting down as no movement is required and hence the graph is expected to be stable. Initial results suggested that walking and jogging exhibit predictable and stable patterns as shown in figure 2.3. The repetitive activities can be described by the time between peaks and magnitudes for each respective axes. It is also noteworthy that the Y-axis exhibits the most displacement and this is due to the pull of gravity.

We can clearly see that for walking and jogging that the peaks are further apart for walking and closer together for jogging. This means that distinguishing between jogging and walking is relatively easy. The problem area was when the participant was ascending and descending stairs. All algorithms had trouble identifying those 2 activities with accuracies ranging from 12.2% for the strawman approach and 61.5% for multi-layer perceptron.

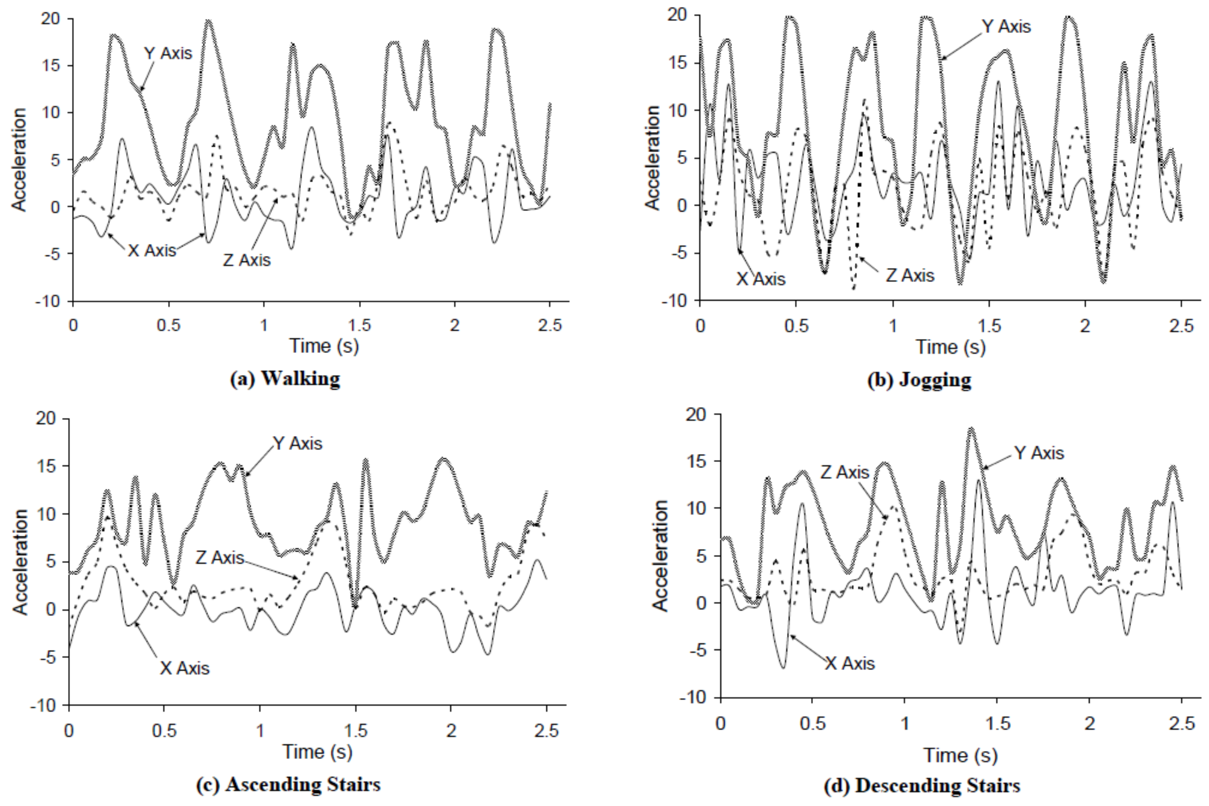
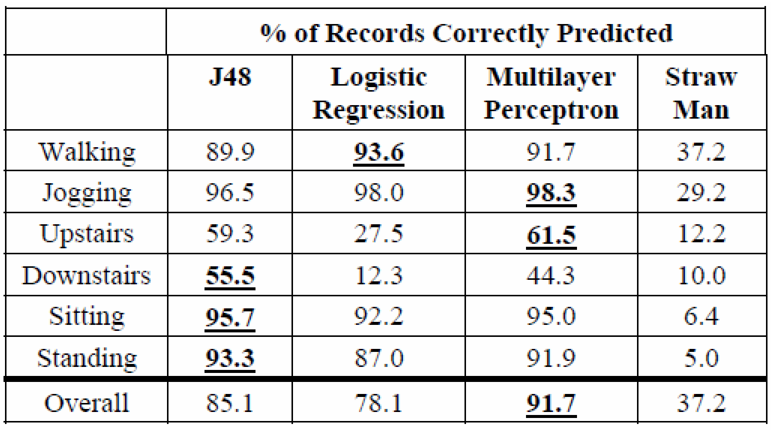


Figure 2.3: Graphs for Acceleration for 4 labelled activities

The researchers used 4 algorithms which were the J48 decision tree, Logistic regression, Multilayer perceptron and the straw man approach. The accuracies are shown in the figure below.



	% of Records Correctly Predicted			
	J48	Logistic Regression	Multilayer Perceptron	Straw Man
Walking	89.9	**93.6**	91.7	37.2
Jogging	96.5	98.0	**98.3**	29.2
Upstairs	59.3	27.5	**61.5**	12.2
Downstairs	**55.5**	12.3	44.3	10.0
Sitting	**95.7**	92.2	95.0	6.4
Standing	**93.3**	87.0	91.9	5.0
Overall	85.1	78.1	**91.7**	37.2

Figure 2.4: Accuracies of tested algorithms

Figure [2.4] suggests that Multilayer perceptron yields the highest accuracy rates. But it is noted that the J48 decision tree also achieves a relatively high accuracy of 85.1%. The straw man approach is a clear failure with accuracy rates tanking to 37.2%. Once again it is obvious that ascending and descending stairs were the most difficult activities to determine due to such similar acceleration signatures. We also observe that walking and jogging detection rates were the highest on average out of all the activities, even exceeding standing and sitting down.

2.6 Conclusion of related work

From Kwapisz et al.[2] and Bao and Intile's [1] research, it is evident that it supports the notion that by using a simple tri-axial/bi-axial accelerometer, the activity of a user can be determined with high accuracy rates (90% or higher) with the best classification algorithms. In addition, Kwapisz et al.[2] conducted their experiments on commercially available mobile phones, which indicates that implementing a system to detect coughing using the accelerometer is possible.

Chapter 3

Data Collection

3.1 Introduction

Data collection was the most important step in this thesis. In order to run the machine learning techniques, a suitable sample size of accelerometer data was collected for 3 different activities and 2 positions of the phone.

3.2 Activities

The 3 activities chosen for analysis were sitting, walking and standing because they were thought to be the most common activities [13]. For each activity the phone was placed in 2 locations: The trouser front pocket and in the hand simulating a user using their phone.

Each test was repeated 10 times for each position without coughing. These activities were done without coughing involved and a total of 60 samples were taken. Each sample without coughing was run over a 20 second period. A 20 second period was chosen because any outliers caused by irregular patterns in the accelerometer readings would be minimised.

The final step of the data collection was to collect samples for the same activities except with coughing involved. This time the coughing samples were taken over a 5 second period because a user would not cough for a prolonged period of time. If the duration was increased to 10 seconds, any spikes caused by a cough would be eliminated thus decreasing chances of detection significantly. A total of 30 traces of coughing were collected for each activity and position.

3.3 Features chosen for analysis

After the data collection phase, it was necessary to research into which features were suitable for detecting cough. Generally most statistical formulas such as mean, standard deviation and kurtosis were natural candidates for the feature set. After researching into features to incorporate, the following list of features was compiled and their formulas shown:

Feature	**Formula**
Mean	$\frac{\sum_{i=1}^{n} x_i}{n}$
Standard Deviation (S.D)	$s = \sqrt{\frac{1}{N-1} \sum_{i=1}^{N} (x_i - \overline{x})^2}$
Kurtosis	$\left\{\frac{n(n+1)}{(n-1)(n-2)(n-3)} \sum \frac{(x_i-\overline{s})^2}{s}\right\} - \frac{3(n-1)^2}{(n-2)(n-3)}$
Skewness	$\frac{n}{(n-1)(n-2)} \sum_{i=1}^{N} \left(\frac{x_i-\overline{x}}{s}\right)^3$
Resultant Acceleration	$\frac{\sum_{i=1}^{N} \sqrt{x_i^2+y_i^2+z_i^2}}{N}$
Root Mean Square (RMS)	$\sqrt{\frac{\sum_{i=1}^{N} x_i}{n}}$
Variance	$\frac{1}{N-1} \sum_{i=1}^{N} (x_i - \overline{x})^2$

Table 3.1: List of features used in cough detection

Chapter 4

Design

This chapter will describe in depth the design process which was used to produce the requisite Android application for this thesis.

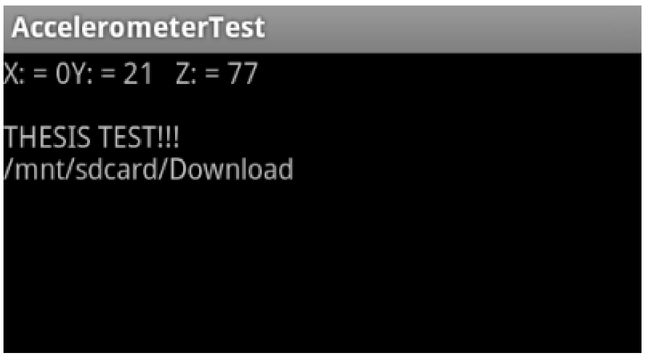
4.1 GUI Design

4.1.1 Notes and Aims

The aim of the GUI for this application was to provide an intuitive and easy to use interface to collect data for various activities such as jogging, walking, sitting and standing. The GUI was written in XML within the eclipse IDE.

4.1.2 Version 1.0

As outlined in chapter 3, collecting data was the most important and time consuming task of this thesis. The first step, therefore was to develop a very simple application to log accelerometer data from a potential user of this system. The simple application did not contain a GUI, instead once the program was running, the x, y, z axes were saved into a .csv file on the phone's local storage.



```
X: = 0Y: = 21   Z: = 77

THESIS TEST!!!
/mnt/sdcard/Download
```

Figure 4.1: The initial android app - a simple data collector

The above figure shows a minimalist interface with only 3 lines of text on the screen. This sort of application was suitable for rough data collection but the thesis required a much more sophisticated data collection method. The pitfall of this application was the user had to manually exit the application to stop collecting data. This has a number of issues. Firstly it is difficult if not impossible to collect a constant number of data points. Secondly the results would be skewed because the user has to place and remove the phone from their pocket, which meant the first and final couple of values would introduce a significant amount of noise.

4.1.3 Version 1.1

With the pitfalls in mind in the first revision, Version 1.1 was aimed to improve the GUI but functionality was to remain the same. The aim at this stage was to introduce incremental changes whilst retaining functionality and the minimalist design/interface. The figure below shows the newly designed GUI:

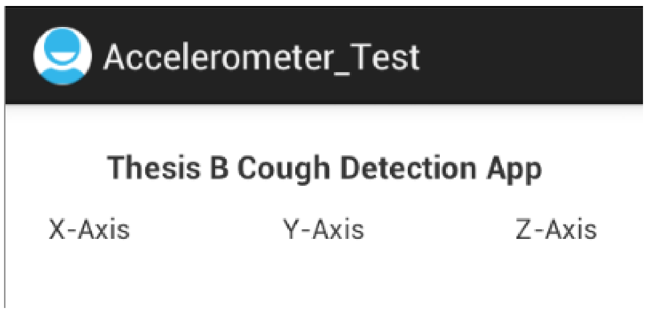
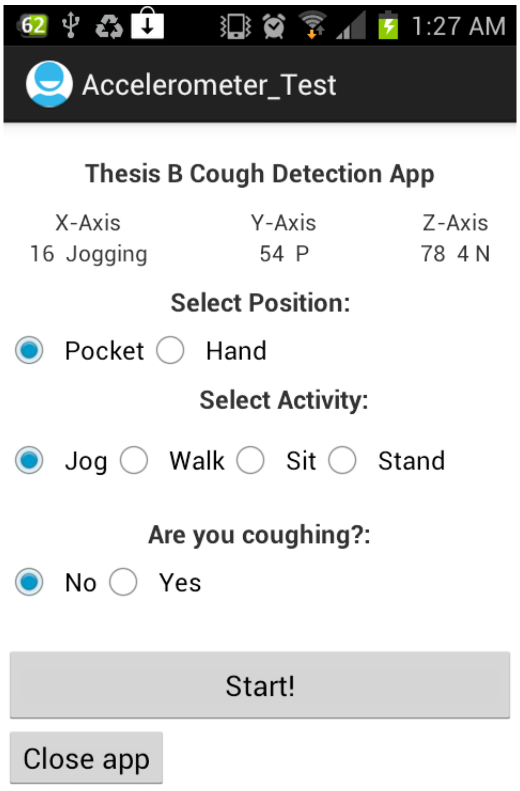


Figure 4.2: Version 1.1 - An incremental improvement in the GUI

It is obvious that the application looked much more fancier but functionality remained the same. This methodology is questionable because it may seem like a better idea to immediately develop a fancy GUI from the start. This was not practical because there were too many developments being made to the back end software that many of the GUI functions were not going to be incorporated into the final product.

4.1.4 Version 2.0

In version 2.0, the most dramatic changes to the GUI was introduced. This version was developed for the sole purpose of collecting data. The GUI was designed to be intuitive for other users to help collect data. By this stage the back end had also made significant changes which will be documented in more detail in chapter 5. The figure below depicts the most recent GUI implementation:



X-Axis	Y-Axis	Z-Axis
16 Jogging	54 P	78 4 N

Select Position: [✓ Pocket] Hand

Select Activity: [✓ Jog] Walk Sit Stand

Are you coughing?: [✓ No] Yes

Figure 4.3: Version 2.0 - The most significant change to the GUI

This GUI design was extremely time consuming because it was expected that the back end software would have also made significant progress. In version 2.0, the most salient features are the included radio buttons and two more buttons towards the bottom - the giant "Start!" button and the "close app" button.

4.1.5 Version 2.1

In version 2.1, most of the functionality for data collection was removed because by that stage all the necessary data to perform analysis was done. The figure below depicts the next version of the GUI:

Version 2.1 also sees a significant change in the GUI. Most notable would be the complete removal of the data collection options present in version 2.0. Two extra buttons have replaced

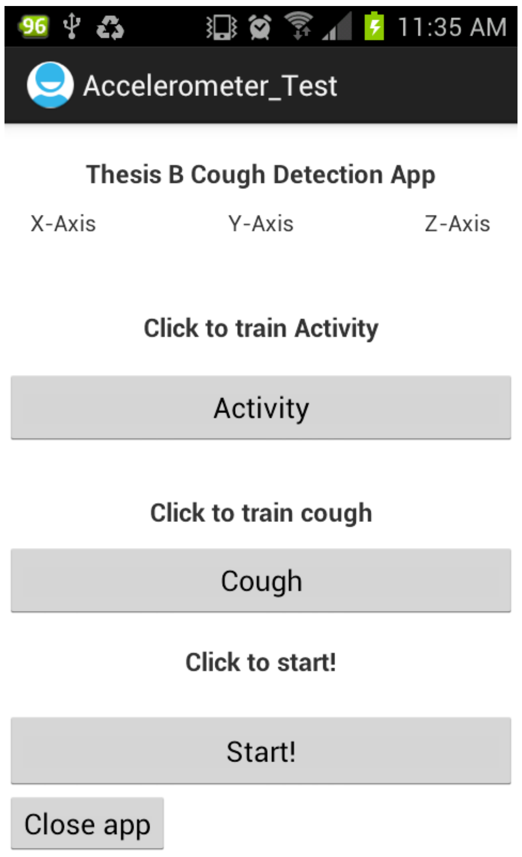


Figure 4.4: Version 2.0 - The most significant change to the GUI

the radio buttons are are labelled ”Train Cough” and ”Train Activity” respectively. The back end has also seen significant changes and the final version of the GUI reflects this. One more thing to note is that after the timer expiry the program will transition to a new screen displaying the results of detection.

4.2 Application Design

This section will describe in detail the design of the cough detector application.

4.2.1 Initial block diagram

During the first portion of the thesis, an idea which was debated was whether to include sound sources to supplement the accelerometer data. With this in mind the initial block diagram was as follows:

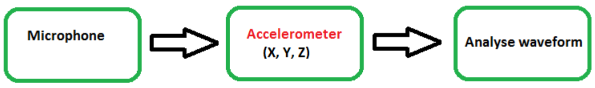


Figure 4.5: Initial block diagram

At this early stage in the thesis, it was difficult to tell what was going to incorporated into the final product. Early ideas included matching the audio recorded with the spikes in the accelerometer data. After some deliberation with my supervisor, it was clear that incorporating audio was not a practical solution. This was because audio cough detection has already been researched thoroughly and relying on the accelerometer would be a better option. Also, incorporating audio would include an extra component which will make development more difficult. The block diagram shown above intentionally displayed very little information because at this stage the scope of this thesis was still being discussed.

4.2.2 Final block diagram

After many weeks of deliberations and discussion, a final decision was made on the program flow. Below is the representation of the combined brainstorming over a couple of weeks.

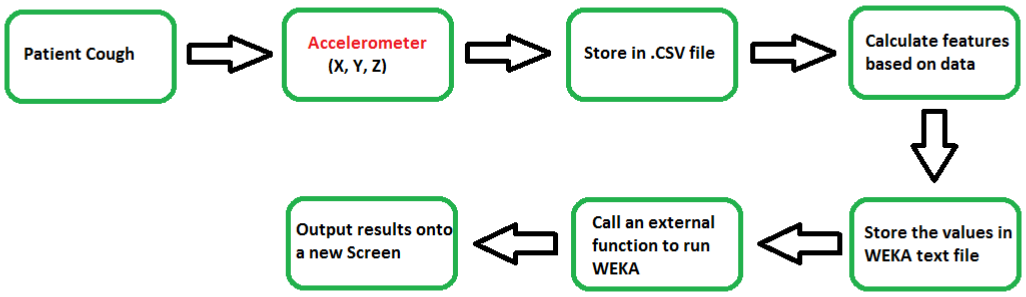

Patient Cough
Accelerometer (X, Y, Z)
Store in .CSV file
Calculate features based on data
Store the values in WEKA text file
Call an external function to run WEKA
Output results onto a new Screen

Figure 4.6: Final block diagram

The final block diagram incorporates the entire back end but leaving out some small steps such as analysis which will be detailed in the next section. I will explain briefly here what happens in each step.

1. A patient carries out their everyday activity with cough or not cough. Phone is placed in their trouser front pocket
2. The phone records accelerometer data for 5 seconds and stores the corresponding values into an array
3. With the accelerometer data the program calculates the values for the features listed in section 3.3
4. The program then stores these values into a WEKA suitable format for analysis and calls another function
5. The results of the function are displayed on the next screen. Note that the program pauses at this stage.

4.3 WEKA file format

WEKA defines a strict file format with an extension of “.arff” with the following format:

```
@relation relation_name
@attribute 'attribute_name' type_of_attribute
...
@data
...
```

The reason why WEKA demands a strict convention is because it expects that the training and testing set of data to be of the same format. The only way to guarantee this is if the file headers are the same. In the event that they are not the same, WEKA will output an error saying that the train and test sets are not compatible.

The WEKA library conveniently provides converters which converts any .csv file into it's equivalent .arff file. Initially the application would save the data as a .csv file and convert it to an .arff file. But this changed because the initial training data file which was used had an attribute "cough" which could take 2 values ({Y,N}) where as the test file could only take N or Y. In this case, the idea of using the .csv converter was scrapped and it was decided that it would be a better solution to manually input the headers.

Below is a snippet of the .arff file displaying only the relevant headers:

```
@relation 'ThesisB_Test'

@attribute 'Mean X' numeric
@attribute 'Mean Y' numeric
@attribute 'Mean Z' numeric
@attribute 'St Dev X' numeric
@attribute 'St Dev Y' numeric
@attribute 'St Dev Z' numeric
@attribute 'Skew X' numeric
@attribute 'Skew Y' numeric
@attribute 'Skew Z' numeric
@attribute 'Res Acc' numeric
@attribute 'Kurt X' numeric
@attribute 'Kurt Y' numeric
@attribute 'Kurt Z' numeric
@attribute 'RMS X' numeric
@attribute 'RMS Y ' numeric
@attribute 'RMS Z' numeric
@attribute 'Var X' numeric
@attribute 'Var Y' numeric
@attribute 'Var Z' numeric
@attribute Cough? {N,Y}

@data
```

Chapter 5

Implementation

This chapter will go in detail the technical aspects of the final Android application, referencing the final block diagram presented in chapter 4. The entire application was developed in java using the standard android SDK available on the internet [14].

5.1 Introduction

The Android operating system was an open source OS designed by Android inc. lead by Andy Rubin and Rich Miner in 2003. The Android operating system was primarily written in C, C++ and java and is based on the ubiquitous Linux kernel. Each application run on Android contains it's own address space pre-allocated by the kernel. In other words, each application runs in a sandbox style and no application cross-talk can occur.

In 2005 Android inc. was bought by Google. The android platform was co-developed by 84 telecommunication companies who were part of the open handset alliance (OHA[1]).

Figure 5.1: Google's Andoid Operating System

5.2 Android environment set up

In order to start development under the android platform, the Eclipse IDE must first be set up. Firstly, we need to download the Android SDK for the particular android version we are

[1]The open handset alliance consisted of major telecommunication companies such as T-Mobile, Sprint, Samsung, Motorola, Qualcomm etc... who supported open source operating systems. The forefront of open source operating systems was Android hence why Google is the current leader of the OHA

developing for . After the relevant SDK is installed, we must then install the ADT (Android Development Tools) in order for the Eclipse IDE to communicate with the mobile phone via a USB interface cable.

Once the eclipse environment is set up properly, the phone must then be set to USB debugging mode so that the ADT can initiate a connection to the phone. To do this we first unlock the screen, then press the menu button and select settings. Then we scroll down and enter Developer options and enable USB debugging mode. USB debugging mode enables the programmer to see debugging details the phone is outputting as well as observing program failure if it occurs. USB debugging also enables the developer to deploy programs directly to the phone and run diagnostics. In the event that a phone is not available, the Android SDK also provides a virtual android device, the draw back is that an emulated device does not have access to the accelerometer and is not indicative of the final product.

5.3 Mobile Device

The development work was primarily done a Samsung Galaxy SII (SGS2) running android 4.0.3 (Ice Cream Sandwich). Outlined below are the specifications of this mobile device:

- **CPU** - Dual core ARM Cortex A9 @ 1.2GHz
- **RAM** - 1GB
- **Sensors** - Accelerometer, gyroscope, light sensor, proximity sensor, compass
- **Screen** - Super AMOLED 480x800 pixel resolution
- **Storage** - 16/32GB Internal storage, supports up to 32GB micro SD cards

As one would notice the SGS2 is a fairly powerful device, which has the implication that the speed of execution on the cough detector application may not scale well to less sophisticated devices. This should not be a concern because the application is designed to execute as little instructions as possible, according to the final block diagram presented in section 4.2.2.

5.4 External libraries

The major library which was incorporated into this thesis was the WEKA library of machine learning techniques. Initially it was thought that the machine learning techniques were to be manually written. But further research on the Google play store revealed applications which incorporated WEKA directly into their software. Thus if these applications allowed WEKA functions to be run on a mobile device, then it is possible to download and import the WEKA libraries directly into an android project.

A copy of the libraries can be found in the following referenced location [15].

Another library which was imported was a math library from apache which allowed functions such as skewness and kurtosis to be calculated without manually writing the formulas. This was good because the formula used previously did not match the formula used in excel when calculating the attribute values in the training data. With the math libraries imported, the comparison between the values in this application and excel were correct.

5.5 Sensor Event Listener

In order to access the various sensors available on an android device, the main activity[2] must extend the **SensorEventListener** interface which is part of the android.hardware package. Such a declaration would typically look like this:

```
public class Accelerometer_Test extends Activity implements SensorEventListener
```

The SensorEventListener interface consists of the following methods which must be implemented:

1. **public void onAccuracyChanged(Sensor arg0, int arg1);**
   This method is called if the accuracy of the sensor has been changed, e.g. the accelerometer polling rate. (This method was irrelevant for this thesis)

2. **public void onSensorChanged(SensorEvent event);**
   This method is called when a sensor detects a change in values, e.g. accelerometer x co-ordinate has increased. (This method was the function which triggered the accelerometer data collection)

3. **protected void onResume();**
   This method is called when a user re-enters the application after pressing and holding the home button. All the sensors that are being used are re-instantiated and the activity resumes execution.

4. **protected void onPause();**
   This method is called when the user presses the home button[3] to return to the android desktop. All currently running sensors are disabled and resources are de-allocated.

5.5.1 onSensorChanged()

The onSensorChanged() method is triggered by an interrupt caused by a sensor hardware event. The method accepts a variable of type **"SensorEvent"** which contains parameters such as the

[2]Android applications are defined as "activities". Each activity has it's own GUI and address space on the device. Each activity does not have access to other activities' variables; instead variables are passed around using Intents and Contexts

[3]Pressing the "Home" button on an android device returns the user to the desktop without killing the currently running application

name of the sensor, accuracy details, time at which the event had occurred and an array containing the values.

Since the accelerometer was the primary sensor, there is an if statement which checks the type of the sensor. If it is of type accelerometer, then read the x, y and z-axes into a variable. The event returns an array of type float[].

```
if (event.sensor.getType() == Sensor.TYPE_ACCELEROMETER){
   x =  (Math.abs(event.values[0])*10);
   y =  (Math.abs(event.values[1])*10);
   z =  (Math.abs(event.values[2])*10);
}
```

The raw accelerometer values retrieved from the device range between +10 and -10 representing a multiplier of the pull of gravity. A stationary phone face up on a desk would measure -1G on the Z-Axis and zero for the other 2 axes. The value was then multiplied by 10 and the absolute value of the result was stored. 10 was a good number because it was enough to show an adequate change upon graphing the values in excel.

5.6 Android File system

The android file system consists of multiple storage locations on a specific device. In the case of the SGS2, there are a total of three locations where persistent data can be stored.

1. Internal Memory allocated to application installation (Limited to 2GB)
2. USB Storage (The amount of in-built flash storage minus internal memory)
3. External_SD card (Max. size 32GB)

The storage location chosen for persistent data was the USB storage. A new directory called "Thesis" was created in the following directory: "/mnt/sdcard/". For this particular model of device, the USB storage is deceptively labeled as an SD card but this did not cause any problems in the application.

New files were created via typical java file operations:

```
File test = new File("/mnt/sdcard/Thesis/", "ThesisB_Test.arff");
test.createNewFile();
```

5.7 Calculating attribute values

In this section I will explain how the calculations were programmed in the application. To see the actual code written please refer to section A of the appendix.

Mean The mean is calculated by summing the x, y and z axes and dividing them by the size of each corresponding array.

Standard Deviation The standard deviation was calculated by taking the square root of the sum of the individual values of each axes subtracted by the mean of the respective axis squared. Finally the result was divided by the size of each corresponding array.

Skewness The skewness was calculated by passing the array for each axis into a variable of type Skewness and calling the evaluate(object[] dataset) method. The result was of type double so it was typecasted into a float.

Average resultant acceleration The average resultant acceleration was calculated by taking the total sum of the square root of the sum of the squares of each axis. Finally the result was divided by the number of data points collected. This feature was the only one which returned one value, others returned 3 values corresponding to their respective axes.

Kurtosis Similarly to skewness, the kurtosis was calculated by calling the evaluate(object[] dataset) method and typecasting the result as a float.

Root mean square (RMS) The RMS was calculated by taking the sum of the square of each value in each of the 3 axes. The final result would equal the square root of the previous result divided by the number of data points.

Variance Finally, the variance is calculated in the same manner as kurtosis and skewness.

After the calculation, the values were stored into an arff file along with the headers from section 4.3. The next step would be to call an external function to initiate WEKA in order to perform a classification on the new data file.

5.8 Using WEKA in the application

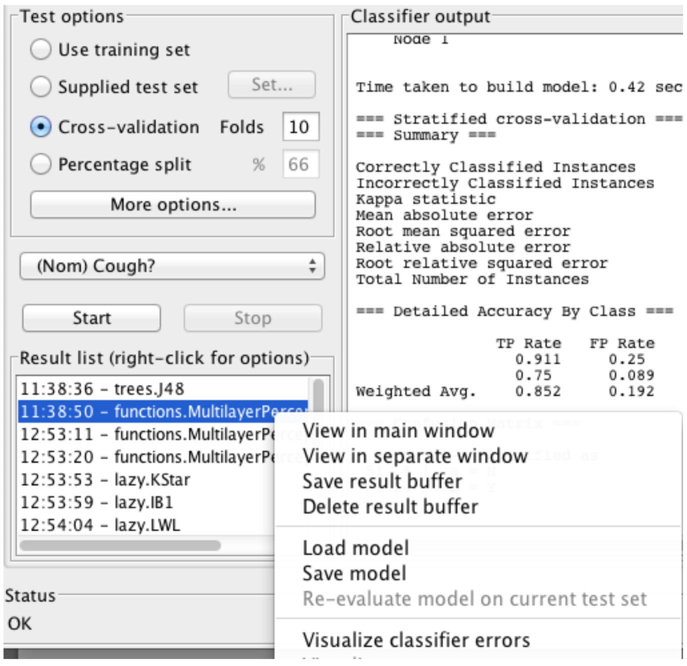
In order to start using the WEKA libraries in the application, the jar file must be imported into the /libs folder of the android project. A java class called weka_test contained all the functionality required to do the analysis as outlined in the below subsections.

5.8.1 Training the classifier

With the data collected outlined in chapter 3, it was necessary to train a classifier in order to run it on a new set of data. There were two methods of approaching this problem.

The Naive Method The most naive method would be to train the classifier every time we want to classify a new set of data against the trained data. This method was very time consuming and battery draining because the data set was fairly large and training algorithms such as multi-layer perceptrons do not come cheap. On the SGS2 it took an average of 4 seconds to train the perceptron thus this method was not an option.

Pre-trained classifier The second method was to train a classifier offline and simply load the saved model every time we want to perform analysis on a new data set. This method was preferred because running WEKA on a single data set is computationally cheap and the training process is time consuming. WEKA supports pre-trained classifiers thanks to java's in-built serialization functionality.



```
11:38:36 - trees.J48
11:38:50 - functions.MultilayerPerce
12:53:11 - functions.MultilayerPe
12:53:20 - functions.MultilayerPe
12:53:53 - lazy.KStar
12:53:59 - lazy.IB1
12:54:04 - lazy.LWL
```

```
    Node 1


Time taken to build model: 0.42 sec

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances
Incorrectly Classified Instances
Kappa statistic
Mean absolute error
Root mean squared error
Relative absolute error
Root relative squared error
Total Number of Instances

=== Detailed Accuracy By Class ===

               TP Rate   FP Rate
                 0.911     0.25
                 0.75      0.089
Weighted Avg.    0.852     0.192
```

Figure 5.2: Saving a previously trained model

5.8.2 Porting the model file

Initially the idea was to run the training on the data set using the multilayer perceptron on the computer and copy the output.model file onto the phone. Below is the snippet of code required to read in a .model file and instantiate a classifier object.

```
Classifier temp = (Classifier) SerializationHelper.read(
"/mnt/sdcard/Thesis/Output_Total.model");
```

Upon calling this line of code, the android application crashed with the error regarding the java serialization class. The error was referring to a mismatched serial identifier between the classifier and the output model. The reason for this might be because when WEKA instantiates a Classifier object and .model file it generates a unique serial ID based on the device it is running on. Since the training and testing devices were not the same, the serial ID's did not match thus causing an application crash. This was only realised when the phone was plugged in and the built-in debugger pointed to this error.

In order to fix this bug, it was necessary to perform the training on the mobile device itself. In order to do this, we must place the .csv or .arff training set into a suitable directory on the SD card and instantiate a new DataSource object pointing to that file. Then we instantiate an Instances object which equals DataSource.getDataSet(). The next step would be to instantiate a new Classifier object according to the chosen machine learning technique[4]. Finally we then must set the classIndex of the Instances object and write the output.model file. The classIndex represents all the features as an array and we subtract 1 from the end to reach cough.

Below is the code for training a multilayer perceptron and outputting the file to the SD card:

```
public void train_cough(String path) throws Exception{
   DataSource source = new DataSource(path);
   Instances data = source.getDataSet();
   Classifier cModel = (Classifier)new MultilayerPerceptron();
   data.setClassIndex(data.numAttributes() - 1); //Point to Cough
   cModel.buildClassifier(data);
   SerializationHelper.write("/mnt/sdcard/Thesis/Output_Total.model", cModel);
}
```

5.8.3 Classifying the activity and cough

During initial research and results, it was found that the user's activity had a bigger impact on detection rates compared to the position of the phone. In order to detect the activity of the user, it was first necessary to edit the training file to remove the coughing and position columns.

A major limitation of WEKA is that classification cannot be done on data which are missing values - we cannot simply omit the "activity" value from the data. The example below depicts what is happening:

```
[Other labels snipped]... Var x, Var y, Var z, Activity
                          21.3,  0.34,  59,    ___?___
```

WEKA demands that all data must be present in order to report accuracy rates - note that WEKA doesn't actually see the value corresponding to the attribute which is being determined. To mitigate this issue I arbitrarily assigned 'Sit' to the activity label so that the data set would be complete. The following lines show the new solution:

```
[Other labels snipped]... Var x, Var y, Var z, Activity
                          21.3,  0.34,  59,    __Sit__
```

[4]With this style of code, it was easy to train multiple classifiers and compare training times.

Now that the data is suitable for analysis, a function called getActivity() is called within the weka_test file. The code for getActivity() is shown below:

```
public String getActivity(String test_location) throws Exception {
   DataSource source = new DataSource(test_location);
   Instances Activity = source.getDataSet();
   Activity.setClassIndex(Activity.numAttributes() - 1);

   Classifier temp = (Classifier) SerializationHelper.read
                   ("/mnt/sdcard/Thesis/Output_Activity.model");

   Evaluation testResult = new Evaluation(Activity);
   testResult.evaluateModel(temp, Activity);
   double[][] temp2 = testResult.confusionMatrix();

   //Main loop
   int column = 0;
   for (int i = 0; i < temp2.length; i++){
      for (int j = 0;j < temp2.length;j++){
      if ((int) temp2[i][j] == 1){
      column = j;
      }
      }
   }
   switch (column){
   case 0:
   return "Sit";
   case 1:
   return "Walk";
   case 2:
   return "Stand";
   default:
   return "error";
   }
}
```

Here is the detailed explanation of the code:

Loading in pre-trained classifier Firstly the previously trained model is loaded and a variable of type "Evaluation" is instantiated.

Evaluating the model The next step would be to run an evaluation of the newly made testing data against the pre-trained model and save the confusion matrix as a 2D array. The next step involving the confusion matrix requires more explanation. The purpose of a confusion matrix is to give a visual representation of the performance of an algorithm in terms of correct and incorrect classification [16]. For example in this scenario the activity of the user can be either standing, sitting or walking. The typical output of the confusion matrix would be as follows:

```
S     ST   W
5     0     1  S
8     2     0  ST
0     0     8  W
```

The diagonal starting from [0,0] to [2,2] across the centre represents the correctly classified instances. All the other rows not on the diagonal represent the incorrect classifications. The first row containing the '1' indicates the algorithm classified one instance of "Standing" as "Walking" and in row two 8 instances of "standing" was classified as "sitting".

Outputting the activity The output of the confusion matrix after classification is as follows:

```
S     ST   W
x     x     x  S
0     0     0  ST
0     0     0  W
```

The position of a '1' in place of the 'x' represents the predicted activity and the two for loops loop through the array to find the position of the '1'. Once it is found the column of the '1' is recorded into a variable. The final step would be to return a string with the predicted activity according to the switch case statements in the final lines of code.

Outputting the cough After retrieving the activity of the user, the same method used for activity prediction is replicated for cough except the missing label is arbitrarily set to 'N'.

```
public String getCough(String test_location) throws Exception {
   ...[snipped]...
   switch (column){
   case 0:
   return "N";
   case 1:
   return "Y";
   default:
   return "error";
```

```
    }
}
```

The final step of the application was to output the results of the activity and cough detection onto a new screen. In order to do this an intent must be created given the class name of the next screen. This was done as follows:

```
    Intent i = new Intent(getApplicationContext(), DisplayResults.class);
    i.putExtra("results_cough", output_cough);
    i.putExtra("results_data", attributes);
    startActivity(i);
```

The next step would be to call the putExtra(String, String) method so that the next activity can access the variables. The final step would be to run the actual activity, displaying the activity and cough values.

Chapter 6

Results and Evaluation

This chapter analyses the performance and feasibility of using an accelerometer on a mobile device to detect coughing. The first section will go through the performance of different machine learning algorithms and the second section will go through the results of actual testing on users.

6.1 Performance of machine learning techniques

In this section, a number of machine learning techniques were chosen from the WEKA library and their performance (accuracy) and confusion matrices recorded down. The algorithms were chosen arbitrarily because WEKA contained over 30 selectable algorithms and the top 3 performing algorithms are presented in this section.

The number of features chosen for analysis was 7. The features and their formulas were documented in chapter 3. The tests were run with 10-fold cross validation with all seven features enabled and setting the cough label as the class which was to be determined.

6.1.1 Artificial Neural Network [3]

An artificial neural network consists of a connection of "neurons" inspired by the connected network in the human brain. A neural network is an adaptive algorithm - meaning the topology changes during the training phase. Neural networks are suited to recognising patterns in a data set which is useful in this application. The default settings for the ANN algorithm was used and no modification to the algorithm was made.

Initial test results with 10-fold cross validation with ANNs were very good. The ANN algorithm correctly classified 90% of the sample data correctly which was a very surprising result. Despite having parasitically induced movement from the user, the algorithm was able to successfully determine if the user was coughing or not. The remaining 10% error rate was due to either false positives or not detecting the cough at all. The confusion matrix for the test is shown below:

```
  a  b   <-- classified as
 53  3 |  a = N
  4 22 |  b = Y
```

The above confusion matrix suggests that ANN is fairly good at reducing false positives with only 3 instances which were wrongly classified. 4 instances of cough was classified as non-cough.

6.1.2 J48 Decision Tree [4]

The J48 decision tree is a WEKA implementation of the C4.5 algorithm first developed by Ross Quinlan. A decision tree uses a greedy technique to induce decision trees for classification. Nodes in each decision tree contain significant features extracted from training data and using these nodes is able to classify unseen data.

The J48 decision tree also had very good detection rates. On average the J48 decision tree achieved a detection rate of 89% which is almost on par with ANN. The power of decision trees is fairly evident in this case, if we put coughing in context, small spikes caused by coughing instances are labeled as features and thus detection rates are very high. The confusion matrix for the J48 decision tree is shown below:

```
  a  b   <-- classified as
 54  2 |  a = N
  7 19 |  b = Y
```

Based on the J48 decision tree confusion matrix, there are 3 more instances where coughing was not detected compared to ANN but one less false positive.

6.1.3 Naive Bayesian classifiers [5]

A naive bayesian classifier is a simple probabilistic algorithm based on applying Bayes' algorithm with (naive) independent assumptions. Each feature present in the data set are viewed as independent entities in which features have no relation with each other.

The naive bayesian classifier did not perform as well as ANN and J48 decision trees but still returned a respectable 86% detection rates. Due to the high variable nature of accelerometer readings, it was expected that algorithms which put weights on individual features rather than seeing the "big picture" performed worse than other algorithms. The confusion matrix for the naive bayesian classifier is shown below:

```
  a  b   <-- classified as
 52  4 |  a = N
  7 19 |  b = Y
```

6.1.4 Discussion

The initial results from a 10-fold cross validation with seven features using WEKA was very encouraging especially because it was initially assumed that detection would be impossible. Accuracy rates across the board were very good ranging from 60% for ZeroR and 91% for Artificial neural networks.

The results presented above support the notion that cough detection is possible on a mobile device, especially with commercially available devices such as the Samsung Galaxy SII. The final remaining question was whether it is possible to detect coughing in real time. i.e. The user is carrying out the aforementioned activities while sitting or standing with the mobile device in their front trouser pocket.

6.2 Real World Evaluation

Although the results presented in the previous sections suggest that a basic tri-axial accelerometer is able to accurately detect coughing, real world testing with actual users was necessary to confirm this hypothesis. The android application developed from the implementation chapter was given to a number of users to install onto their android phones.

A total of 5 people were selected to participate in this experiment, 3 users had a cough at the time and the remaining 2 were healthy individuals. Each user was told to place the phone in their trouser front pocket while doing either of the activities listed in section 3.2 and perform them like how they would usually. The test was performed a total of 10 times in varying times of the day when the users were coughing. For the users who were not coughing at that time, they were told to induce coughing for the purposes of this experiment. A total of 50 samples of coughing with various activities was recorded and the detection results are shown below:

Cough	**Non-Cough**	**Accuracy (%)**
38	12	76%

Table 6.1: Performance of cough detection on real users

The above table suggests that the detection rates for real world testing are fairly substandard compared to results obtained from the 10-fold cross validation done in WEKA. A 76% accuracy rate is still fairly good for this proof of concept because this is the first time that an accelerometer in a mobile phone has been used to detect coughing. Fortunately there are a number of reasons why the accuracy was not on par:

Lack of training samples The lack of training samples presented to WEKA meant that there were only 30 instances of coughing for a particular person. It is obvious that people have different coughing signatures and intensities and the training set only had data from one person. If the training set was expanded to accommodate a multitude of users, then the detection rates of coughing would be increased.

Extra features Extra statistical features such as the range, interquartile range or even features presented in Kwapisz et al.[2] could help increase detection rates.

Different machine learning technique Artificial neural networks was chosen because ANN obtained the highest detection rates out of all the tested algorithms. There is a chance that an alternative is available which is more suited to detecting minute changes in acceleration profiles such as in this application.

6.2.1 Conclusion

With the results presented above, a preliminary conclusion can be made. Firstly the detection rates are still fairly good in the circumstances presented to the algorithms. Secondly if the suggestions above were implemented the accuracy rates may improve significantly. There is clearly a lot more which could be done for this thesis to improve detection rates. But we must also remember that this research is currently a proof of concept so any notable results from real world testing is a bonus.

Chapter 7

Conclusion

The original goal of this thesis was to determine whether it was feasible to detect coughing using a standard tri-axial accelerometer on a mobile phone. The platform chosen for development was Google's Android operating system because of it's reliance on regular java code and eclipse.

The design and implementation chapters of this thesis clearly demonstrate that with a simple android application, it is possible to reliably collect data to determine if the user is coughing or not. Accuracies ranging between 60% for ZeroR and 91% for Artificial neural networks were achieved from data collected from the application.

As a bonus, detection was also implemented on the mobile phone itself and a 76% accuracy rate was achieved. This is an excellent result because possible improvements could eventually increase the accuracy rates to well above 90%.

Appendix A

Calculate Attribute Method

```
public void calculateAttr(){
   int size = values.size()-1;
   double[] tempX = new double[size+1];
   double[] tempY = new double[size+1];
   double[] tempZ = new double[size+1];

   //Calculating Mean
   attributes[0] = sumX/size;
   attributes[1] = sumY/size;
   attributes[2] = sumZ/size;

   //One massive loop to do all the calculations
   for (int i = 0; i <= size; i++){
       xyz temp = values.get(i);

   tempX[i] = temp.x;
   tempY[i] = temp.y;
   tempZ[i] = temp.z;
   /*
    * Standard Deviation
    */
   attributes[3] += Math.pow((temp.x - attributes[0]), 2);
   attributes[4] += Math.pow((temp.y - attributes[1]), 2);
   attributes[5] += Math.pow((temp.z - attributes[2]), 2);
   /*
    * Calculate Resultant Acceleration SUM(sqrt(x^2+y^2+z^2))/ N
    */
   attributes[9] += FloatMath.sqrt((temp.x*temp.x) +
                        (temp.y*temp.y) + (temp.z*temp.z));
   /*
    * Calculate RMS sqrt(SUM(Xi^2) / N)
    */
   attributes[13] += Math.pow(temp.x, 2);
   attributes[14] += Math.pow(temp.y, 2);
   attributes[15] += Math.pow(temp.z, 2);
   }
   /*
    * Standard Deviation
    */
   attributes[3] = FloatMath.sqrt(attributes[3]/size);
   attributes[4] = FloatMath.sqrt(attributes[4]/size);
   attributes[5] = FloatMath.sqrt(attributes[5]/size);
   /*
    * Skewness
    */
   Skewness xS = new Skewness(), yS = new Skewness(), zS = new Skewness();
   attributes[6] = (float) xS.evaluate(tempX);
   attributes[7] = (float) yS.evaluate(tempY);
   attributes[8] = (float) zS.evaluate(tempZ);
   /*
    * Resultant Acceleration
    */
   attributes[9] /=size;
   /*
    * Kurtosis
    */
   Kurtosis xK = new Kurtosis(), yK = new Kurtosis(), zK = new Kurtosis();
   attributes[10] = (float) xK.evaluate(tempX);
   attributes[11] = (float) yK.evaluate(tempY);
   attributes[12] = (float) zK.evaluate(tempZ);
   /*
    * RMS
    */
   attributes[13] = FloatMath.sqrt(attributes[13]/size);
   attributes[14] = FloatMath.sqrt(attributes[14]/size);
   attributes[15] = FloatMath.sqrt(attributes[15]/size);
   /*
    * Variance
    */
```

```
Variance vX = new Variance(), vY = new Variance(), vZ = new Variance();
attributes[16] = (float) vX.evaluate(tempX);
attributes[17] = (float) vY.evaluate(tempY);
attributes[18] = (float) vZ.evaluate(tempZ);
}
```

Bibliography

[1] Ling Bao; Stephen S Intille. Activity Recognition from user-annotated acceleration data. *Proc Pervasive*, 2004.

[2] Jennifer R. Kwapisz; Gary M. Weiss; Samuel A. Moore. Activity recognition using cell phone accelerometers. *ACM SIGKDD Explorations Newsletter*, 2011.

[3] Jeff Dalton; Atul Deshmane. Artifial Neural Networks. *IEEE Potentials*, 1991.

[4] THARA SOMAN; PATRICK O. BOBBIE. Classification of Arrhythmia Using Machine Learning Techniques. *WSEAS Transactions on Computers*, 2005.

[5] G. Potamias; V. Moustakis; M. van Someren. Classification of Arrhythmia Using Machine Learning Techniques. *Proceedings of the workshop on Machine Learning in the New Information Age, 11th European Conference on Machine Learning, Barcelona, Spain, pp. 9-17*, 2000.

[6] Laura Greenhalgh. New strain blamed for whooping cough epidemic. `http://www.cosmosmagazine.com/news/5466/new-strain-blamed-whooping-cough-epidemic`, 2012.

[7] Cough. `www.wikipedia.org/wiki/cough`, 2012.

[8] Thomas Drugman; Jerome Urbain; Thierry Dutoit. Assessment of audio features for automatic cough detection. *19th European Signal Processing Conference (Eusipco11), Barcelona, Spain*, 2011.

[9] F. Zheng; G. Zhang; Z. Song. Comparison of different implementations of mfcc. *The Interspeech Conference*, 2001.

[10] G. Peeters. A large set of audio features for sound description (similarity and classification). *The Cuidado Project*, 2003.

[11] H. Kawahara; H. Katayose; A. de Cheveigne; R. Patterson. Fixed point analysis of frequency to instantaneous frequency mapping for accurate estimation of f0 and periodicity. *In Proc. Eurospeech, volume 6*, 1999.

[12] Samantha J Barry; Adrie D Dane; Alyn H Morice; Anthony D Walmsley. The automatic recognition and counting of cough. *biomed central*, 2006.

[13] Xi Long; Bin Yin; Ronald M. Aarts. Single-Accelerometer-Based Daily Physical Activity Classification. *31st Annual International Conference of the IEEE EMBS Minneapolis, Minnesota, USA, September 2-6*, 2009.

[14] Android sdk. `http://developer.android.com/sdk/index.html`, 2012.

[15] Weka libraries. `http://www.cs.waikato.ac.nz/ml/weka/`, 2012.

[16] Confusion matrix. `http://en.wikipedia.org/wiki/Confusion_matrix`, 2012.