Filip Frahm

s1232367

System Level Integration Practical

Group A: UberVest

INTRODUCTION (Level 1 heading)

With recent news about NSA surveillance, large scale data collection seems to have a negative connotation in our society. While fear for ones' privacy is justified,

large scale data collection can be of tremendous benefit. An application of it that stands out is the imrpovement of our society's health. At the University of

Edinburgh, researcher Andy Sims is using bioinformatics and genetic data to predict which cancer drugs are most appropriate for a breat cancer patient[3].

In San Francisco, Google-backed company Calico[2] gained access to Ancestry.com's genetic database with the intention of using genetic and family tree related data

to improve the human lifespan.[1]

The goal for our project was to create a vest that would 1)have as many health related sensors as possible and 2)process and present that data to the user. We believe

that such a vest would not only be attractive for personal use but that it could, similarly to the examples above, also advance medical research by providing data.

In the duration of the course, a temperature sensor and an electrocardiogram(ECG) were installed to a vest. Further, a website and mobile application were created to

process and present the data. While a respiratory sensor has not yet been integrated to the vest some research regarding future applications of such a sensor has been

completed.

My role in the project was to process data. Data processing included the conversion of data from the ECG to a beats per minute(BPM) value. Additionally, it

was my task to explore potential uses of a respiratory sensor (RESpeck). ADD STUFF ON RESpeck HERE

ECG DATA PROCESSING (Level 1 heading)

SPECIFICATIONS (level 2 heading)

The requirements for the ECG data processing code (BPM code) were to take ECG data from the server and to then return a BPM. Data passed from the server would be in a

python dictionary with unix timestamps as keys and integers representing voltage (voltstamps) as values. Voltstamps theoretically range from zero to 1024 but only

range from zero to around 360 in practice. In available data, data points were typically between 0.001 and 0.05 seconds apart. In order to be able to provide a live BPM

feed, the BPM code was to be applied to a moving window with length between zero and ten seconds.

Figure 1

METHOD (level 2 heading)

The BPM code can be divided into two major parts. The first and more computationally expensive one parses through a given window of data and attempts to identify the

timestamps and voltstamps of heart beats. In the process, more information about the data is collected. The second part of the code uses the previously collected

information to predict how many beats would occur over 60 seconds.

BEAT DETECTION AND STEP STRUCTURE (level 3 heading)

In early stages of the project the approach to finding a beat detecting method was somewhat manual. A method that would theoretically detect beats would be chosen and

then tested against available data. This led to some issues. Firstly, little data was available for testin and the manual approach allowed little flexibility.

As a result, methods would frequently work for small samples, but not be flexible enough to work for larger samples. Secondly, the varying density of datapoints,

requires more versatile methods. A combination of different methods was required to attain a good output.

To avoid the above-listed issues, a computational approach was pursued. Instead of choosing methods manually, a fixed structure (the step structure) was created. The

step structure itself remains the same, but it allows for the choosing of different combinations of methods and benchmarks within it. To determine the optimal

combination of methods and benchmarks a testing environment was created (detailed in TestingITALICS). The computational approach doesn't just try more options and

parse more data than the manual approach, it also easily adapts to new data.

The step structure code starts with the window of data as described before and ends with a dictionary of suspected beats. It does so in four steps outlined in

Figure 2ITALICS. The individual methods(or conditions) for each step are described in Figure 3ITALICS.

Figure 2

Figure 3

Note that for all the step one conditions any of the following five logical statements can be applied to the condition:

1) Whether the condition is met or not has no impact on the selection of the iter\_windowITALICS.

2) If the condition is met, the iter\_windowITALICS will be selected regardless of the results of other methods.

3) If the condition is not met, the iter\_windowITALICS will be selected regardless of the results of other methods.

4) Unless 2) or 3) occurs, this condition must not be met for the iter\_windowITALICS to be selected.

5) Unless 2) or 3) occurs, this condition must be met for the iter\_windowITALICS to be selected.

As a result of testing (detailed in TestingITALICS), the best choice of benchmarks and methods emerged to be HELP.

BEATS PER MINUTE (level 3 heading)

The idea of the BPM is not to project how many beats a user will have in one minute, but rather to let the user know how many beats in a minute he would have at the

current rate. This makes the generation of a BPM value simple (since the beats have already been identified). We simply divide 60 seconds by the average time between

beats.

Simplifying,

Figure 4

ERROR DETECTION AND ADJUSTMENT (level 4 heading)

Since the ECG is easily affected by movement, the BPM code needs to adjust to unusable data. It does so in two ways. Firstly, it flags data windows with extremely high

variance (variance > 60). Secondly, it discards iter\_windowsITALICS that contain zero voltstamps. In practice zero voltstamps only occur as a symptom of extreme noise (the normal

range is 140 to 250). The percentage of iter\_windowsITALICS are used is then computed and used to adjust the BPM formula as described in Figure 5ITALICS below.

Figure 5

This adjustment is not correct. This method was chosen due to time constraints, but an alternative method is suggested implicitly in the criticism below. Given a

data window such as the one in Figure 6ITALICS below, the effect of the error adjustment depends on where the error occurs. The issue with the current method lies in

the fact that data outside of t1 and tn HELP is not considered for the BPM but it is considered in the error adjustment. Figure 7ITALICS discusses different scenarios.

Figure 6

Figure 7

FOR TABLE Index | Scenario | Current Adjustment | Correct Adjustment | Error | Summary

1|

Scenario: Error occurs only in between t1 and tn HELP.

Current Adjustment: 100\*BPM¬old / percentage of windows used HELP

Correct Adjustment: BPM¬old\*(tn-t1)/(tn-t1-error\_duration) HELP

Error Bounds: HELP

Summary: Adujstment is not severe enough.

2|

Scenario: Error occurs only outside of t1 and tn HELP

Current Adjustment: 100\*BPM¬old / percentage of windows used HELP

Correct Adjustment: No adjustment

Error Bounds: HELP

Summary: Adjustment is too severe.

3|

Scenario: Error occurs inbetween t1 and tn HELP and outside of t1 and tn HELP

Current Adjustment: 100\*BPM¬old / percentage of windows used HELP

Correct Adjustment: BPM¬old\*(tn-t1)/(tn-t1-inside\_error\_duration) HELP

Error Bounds: HELP

Summary: Adjustment could be too severe or not sever enough.

In testing, data windows with high error rates (error > X% HELP) were discarded. Nonetheless, the above adjustment could cause significant error. Calculations for the

above error bounds are provided in the CalculationsITALICS seciont (Calculations 1ITALICS).