CONTINUOUS PERSONALIZED FALL DETECTION AND

DATA COLLECTION

by

Shaun Coyne

HONORS THESIS

Submitted to Texas State University
in partial fulfillment
of the requirements for
graduation in the Honors College
May 2020

Thesis Supervisor:

Anne H.H. Ngu

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

Duplication Permission

ACKNOWLEDGEMENTS

Finally, I wish to acknowledge the support of my co-workers, friends, and family. Each of these people had some contribution, whether it be from the technical input of co-workers to the moral boost provided by loved ones—it kept me going.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

CONTINUOUS PERSONALIZED FALL DETECTION AND DATA COLLECTION

by

Shaun Coyne

Texas State University

May 2020

SUPERVISING PROFESSOR: ANNE H.H. NGU

Falls are very real problems for elderly people. Falling is the leading cause of injury and death in older Americans according to the Centers for Disease Control and Prevention. Currently there is no software system for reliably detecting falls for elder people that is unobtrusive, affordable, and easily accessible. While the idea of using a wrist worn smart watch to collect accelerometer data to predict falls has been proposed before, the fall detection algorithms are all trained using simulated fall data from healthy young adults. Any system trained for fall detection that utilizes only a simulated dataset cannot be recommended for real-world application as it does not reflect the characteristics of fall from the target population. We

propose a solution that will move fall detection into the real world by being capable of collecting real data from elder patients unobtrusively. This system utilizes an online database and a GPU server to collect labeled data from each user and retrain the model specific to that user dynamically. This allows the system to learn the patterns in a user's linear acceleration data combined with a pre-existing synthetic dataset to determine if a user has fallen or not. That is, it will be capable of personalizing the model to each user to improve precision while maintaining the recall. Additionally, it is capable of reliably collecting all linear acceleration data (falls or activities of daily life) from a user via positive and negative feedback data. This will help to provide future research with real-world fall datasets to create even more reliable models. This system is built to scale using Couchbase, Tensorflow, and commodity smartwatch and smartphone devices.

CHAPTER I

INTRODUCTION

Falls have been a well explored problem due to their prevalence across elder people.
Its significance in mortality rates in elder people is well known and research into
detecting falls continue to illustrate how difficult of a problem it is to address.
According to the U.S. Center of Disease Control and Prevention, one in four
Americans aged 65 and older falls each year. Many find this statistics to be a very
real part of their lives, such as within my REU project team where one of our
member's grandmothers suffered from a very bad fall. While she had the life alert
system, she opted not to wear it at various parts of the day. This resulted in her
being left alone on the floor without any way to call for help for an entire day. Since
then, she had to be placed in assisted living in case she was to fall again.

Wearable smart watches have been bringing health monitoring, and thus fall
detection, closer to a reality. However, a generic algorithm such as Apple's "hard
fall" detection or even more advanced deep learning models have proven to be
ineffective at covering all cases of falls and ADL (Activity of Daily Living) data.
Our previous work [1] was able to show that using simulated data from young and
healthy adults, we can cover most falls as well as ADLs by utilizing a

personalization strategy. This strategy involved a deep learning model trained on simulated falls, plus labeled ADL data collected from the user (feedback data) while wearing the watch installed with a generic model. This feedback data was used to create a personalized fall detection model that had over 90% recall with very few false alarms. However, there are two main issues with this personalized fall detection system.

First, the dataset is not only small, but also not representative of the target users. It is not possible (or ethical) to collect real fall data from elderly people in any controlled environment. Falls need to occur naturally to be realistic. The system we are proposing here will allow elderly users to wear (and use) the watch and collect their real-world acceleration data due to fall or ADL naturally. Although we do not hope that anyone gets hurt for the sake of collecting fall data, if the user (watch wearer) does happen to fall while wearing the watch with our fall detection system, at least we will be able to use that data to better detect future falls for elderly people.

The second issue is that creating a personalized model in the previous system was done manually. The Fall Detection App is designed to save the collected data on the phone using a CSV file format. A programmer has to splice the data together in Microsoft Excel and manually organize the data in a file manager to prepare it for re-training. This is not scale-able and leaves room for human error.

Moreover, the Fall Detection App would eventually fall behind in recording data as more and more data are collected and eventually crash as the system tried to read and write to a large csv file to make a prediction.

We aim to solve the above problems by automating the entire personalization process, from the user initially wearing the device, to getting feedback or ADL data from the user, re-training the new model, and uploading the new model to the Fall Detection App automatically. Some of the challenges for automating the personalization process include accounting for the long model training time, a faster storage system on the phone, and automatic archiving or transferring of data to the server periodically. Our solution involves the selection of a storage system called Couchbase being used for the storage of collected data and the re-trained model. The Couchbase database allows for robust and efficient storage on the phone, a central place to store all the data, and fast retrieval of archived data for re-training. Finally, we also made the additional improvement of creating the Fall Detection App UI (User Interface) on both the watch and the phone. This was done because elderly people have difficulties keeping up with devices that are not directly attached to them and may find a phone difficult to retrieve from their pocket. A watch's UI allows them to easily interact with the Fall Detection App running on the phone while the phone is not within the reach of the user. In this thesis, I contribute a system that can robustly collect labeled accelerometer data from any

number of users. I also show that this data can be used to personalize models which can then be uploaded back to the existing Fall Detection App to increase precision for the users. Finally, I demonstrate that all of this can be automated completely, and that the system's user interface is very easy to use.

CHAPTER II

BACKGROUND

A recent survey in Fall Detection systems shows much progress in using machine learning to detect falls given accelerometer data [2]. The datasets used to train models are all synthetically created by utilizing fall data collected in controlled experiments with primarily young, healthy adult participants simulating a fall. This data also includes ADL data collected from the participants. Some of these datasets are publicly available. There was a wide range of success, however the most success was achieved using custom hardware mounted on the chest or waist. Unfortunately, chest or waist mounted fall detection systems can be invasive, uncomfortable, or embarrassing for users to wear in public. Other systems that range from infrared monitoring [3] to location monitoring [4] all require wearable custom hardware [2]. It is not reasonable to setup a custom array of cameras and sensors throughout a home to predict falls; not only is it invasive, but also it does not help seniors who go outside of the detection area. This is the motivation for smartwatch-based fall detection. It is portable and does not require video/ cameras or custom devices. Moreover, many of the custom hardware solutions involve mounting the system in a very specific area of the person, usually around the waist or chest. If the devices

position is altered, it does not work properly. Additionally, even a small device such as a pendant can be difficult to use, and overall frustrating as pointed out [5]. Thus, we propose a watch-based fall detection system as a familiar device that an elder person would be more inclined to use.

Another challenge of fall detection is false positives. A survey paper in ICSTCC notes the various strategies used to help combat false positives [6] however, it remains an unsolved issue. This is in large part due to a small dataset problem. Not only are the datasets synthetic and not representative of the target population but also, they are relatively small with limited variation in types of falls and ADLs. When taken to the real world, any activity not represented in the training set can lead to a false positive. This could lead to hundreds, if not thousands of incorrect alarms when scaled to a single nursing home [6]. There have been some proposed ideas to reduce this problem such as detecting relevant context to a fall. Specifically, it has been proposed that if you can detect a fall and someone lying still, then they have truly fallen [7] [8] [9]. This strategy greatly reduced the false positives. However, this assumes expert knowledge on a dataset that does not exist. Currently, there is no dataset of elder people falling or performing ADLs while wearing a watch-based accelerometer sensor. There are various instances in which a fall occurs but acceleration data could continue to remain active. These cases could be things such as Parkinson's, seizures, injury or any instance in which the user is

awake but unable to get up or dial for help. We do not know to what proportions of elderly falling resulted in total stillness vs continued movement. While false positives are annoying, false negatives can be deadly. Therefore, any proposed system will need to rely solely on its ability to learn patterns of falls and ignore patterns in ADLs without expert knowledge. We do not investigate falls in conjunction with seizures or Parkinson diseases in this thesis.

Most recently, personalization has been used to reduce false positives. This has been achieved by two strategies. Both systems utilized some form of generic model that was trained on a synthetic dataset from wrist worn devices. The first system utilized a Bag of Words strategy to collect labeled FP (False Positive) data from the user. Each ADL was added to the bag and future detected falls were compared against these previous ADLs. If the data was similar, then it was an ADL. Otherwise, it was a fall. After the detected fall, the user could confirm if this was a fall or another ADL [10]. This personalized bag of words was able to reduce some of the false positives without affecting recall. This system also attempted to use common ADLs for transfer learning, such that new users could benefit from this labeled data. However, it was found that most of the labeled data in the bag was never encountered again. Meaning the bag kept growing as new ADLs kept being received. This does not scale well for mobile devices. There was a severe lack of commonly occurring ADLs and this prevented transfer learning from being a

practical solution.

The second approach for personalization was in our earlier work [1]. We demonstrated that we could maintain the high recall (sensitivity) of a wrist-worn watch based fall detection as well as increase the precision (specificity) by collecting personal false positive data and including that in the dataset and then re-train the model. This strategy uses an RNN ensemble model trained with simulated dataset to detect if a user has fallen given the user's accelerometer data collected from the watch. After a couple rounds of various ADLs (Activity of Daily Living) performed by the user, the collected data is used to re-generate a new model, thus creating a "personalized" fall detection model. This personalized model was nearly twice as precise as the generic model (a model trained without personal ADL data). This process shows that we can create a system that learns the difference between falls and ADLs without expert knowledge. It needs only feedback from the user to increase its precision.

CHAPTER III

METHODOLOGY

In this chapter we will explain how to implement personalization and achieve automated retraining. The system contains multiple parts including: a wearOS application, an Android application, a Couchbase Server, and a GPU server. This chapter will be organized into the following subsections: Train From Scratch—an explanation of how we achieve personalization; the Fall Detection App—how we get positive and negative feedback data; Data collection and Inference—how we get sensor data from the watch and make predictions on the phone; Data storage—how we store the data in a mobile couchbase instance; Database Synchronization—how we synchronize the mobile database with the cloud/server database; Automation of personalization—how we decide when to retrain; Data Trimming—how we select data for re-training; Dataset Creation—how we create training and test datasets; Model generation—how we create new models; Model validation—how we determine which model is the best.

3.1   Training From Scratch

We utilize a similar personalization strategy described in previous work [1].
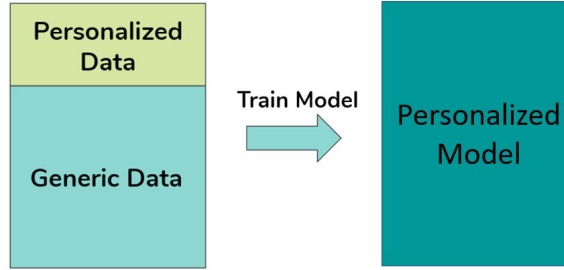
Figure 3.1: Personalized Model Creation

This personalization strategy reduces false positives (increases precision) by combining a generic dataset containing synthetic TP (True Positive) and ADL data with personal ADL data collected from the user. Our initial model is trained using the same dataset used to create the generic model, and is described in this earlier publication [11]. In prior work, we utilized personalization to improve the precision of the fall detection model. The experiments were successful in increasing precision, however the methods used would not be able to scale in the real world. Previously, we asked users to perform a very specific set of ADLs and we recorded the associated wrist acceleration data to a CSV file. We appended this newly collected personal ADL data to the original training set to create new models. We repeated this process again to create models that were highly tuned to the users personal ADL patterns. This process can be represented in figure 3.1 Our latest implementation changes this process slightly to allow for an easier method of labeling data. A user is asked to wear the watch running the generic model during an hour of high activity. There is a list of various ADLs we want them to perform
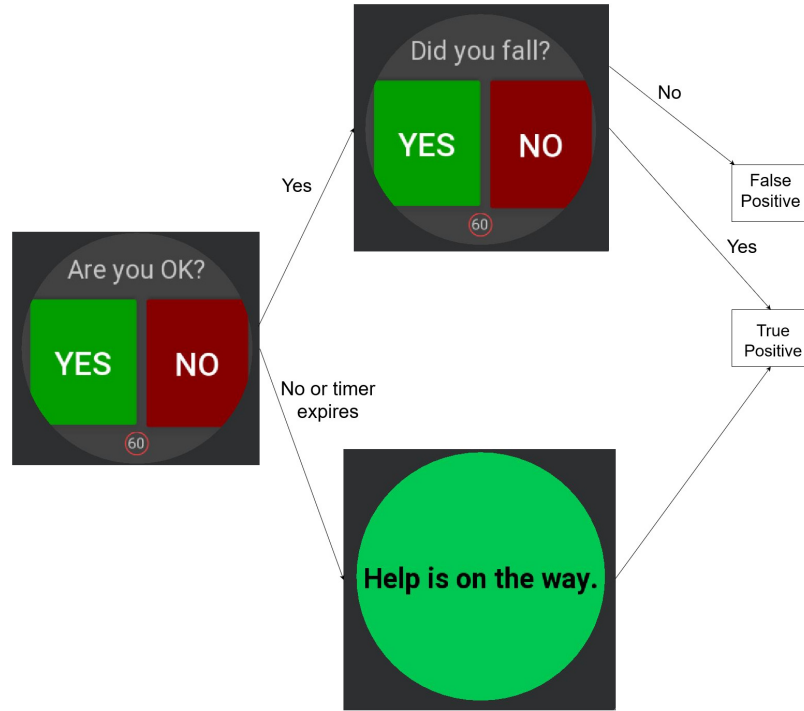
Figure 3.2: User interface display after a fall is detected

at the beginning of this hour, after that, they can carry on with any ADLs they normally perform in the average day. During this time period, the system will generate many false alarms that the user will label through the watch's UI. See Figure 3.2. Once they have completed this process, they can either turn off the system (deactivate the app and take off the watch) or continue to wear the watch.

Once the device/watch is being charged and connected to WiFi, the feedback data will be upload automatically. Unlike the previous system, the user can perform as many ADLs as they want in an uncontrolled environment, starting and stopping as they please. This is due to our new pre-processing script that identifies the high
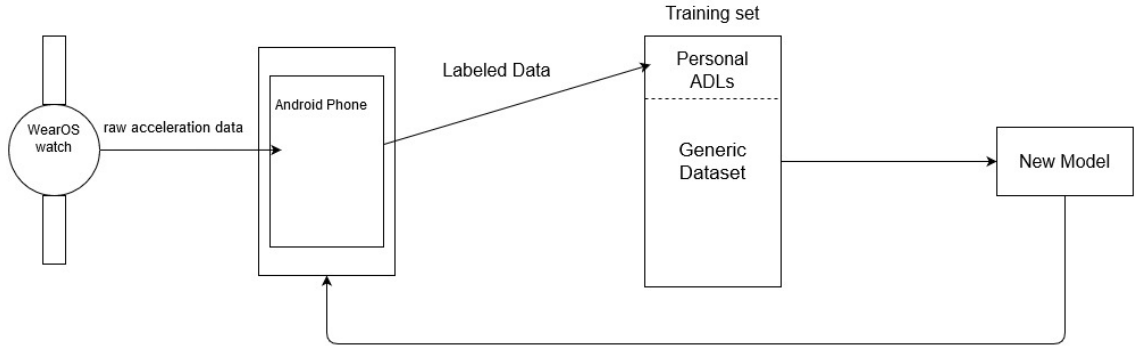
Figure 3.3: Flow of the automatic personalization strategy

accelerometer data and discards very low ones (for example, if a user is sitting on the couch for a long period of time watching a movie, there is no need to upload all those very low acceleration data). We trim some data around the target high accelerometer data with a small buffer of low accelerometer data in between each end. The data is then appended to the original training set, and a new model is created. This model can be automatically pushed onto the phone. This completes the process for creating personalized models. An overview of this process can be seen in figure 3.3.

3.2   The Fall Detection App

The original android Fall Detection App had all of the user interface on the phone. When it comes to starting and stopping the application this is fine. User first needs to create a profile (see figure 3.5) and then click start activity (see figure 3.4). However, when a fall was detected the UI's prompt appeared on the phone. This

meant the user had to stop what they are doing, retrieve the phone from the pocket or nearby purse, and then finally react to the prompt which is analogous to labelling the data. This has to be done before the timer expired. More importantly, if it was actually a fall, they could have trouble accessing the device and would have no way of knowing if help is on the way.

Figure 3.4: Screenshot of what the user sees when opening the app.
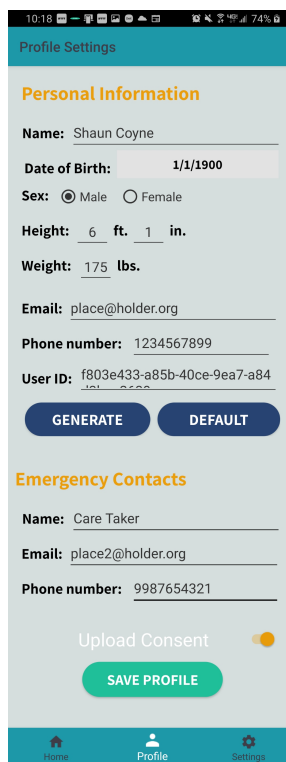
Figure 3.5: Screenshot of how users create a profile.
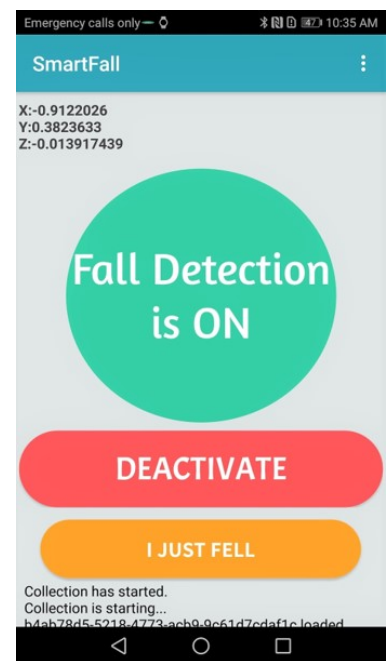
Figure 3.6: Screenshot of what the user sees when fall detection is running.

The user interface was changed significantly since the last version of our system. We retain the profile creation screen as well as the start and stop screen on the phone. See figure 3.6. We moved the labeling of the data (i.e giving feedback) to

13

Figure 3.7: Screenshot of the watch connecting to the phone.
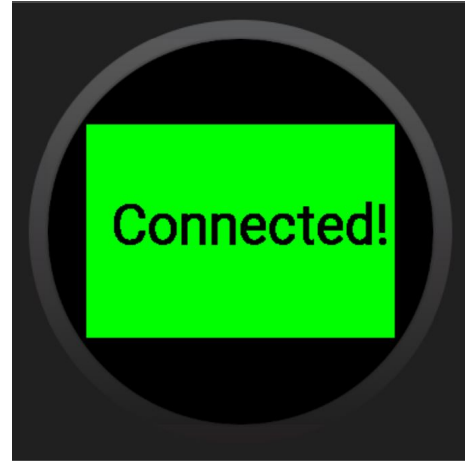
Figure 3.8: Screenshot of the watch connected to the phone.

the watch. See figure 3.2. The phone is used to activate and de-activate the Fall Detection App. Users must create a profile, which is stored locally on the phone. The profile contains various personal information as well as the including emergency contact information. This information is used to send alerts to a caretaker if the system detects that the user has had a fall. The profile creation is only done once. Once a profile is created and selected, Fall Detection App can be activated. In order to activate it, the user must open the App on the phone then open the App on the watch as well. The watch will display "Connecting" (figure 3.7) while it confirms two-way communication with the App on the phone, then displays "Connected" (figure 3.8). At this time, the user can hit "Activate" and the Fall Detection will be running. The user can then place the phone in their pocket or carry it around on a

14

belt (the phone must be within the blue tooth range of the watch). The watch App

runs in the background, so the user can continue using the watch for other purposes

while this application is running. There is no need to interact with the App unless

the system detects that a fall has occurred.

Once a fall is detected, the watch will prompt the user to label the data. See

Figure 3.2. A dangerous fall is when a user presses the "I need help" button when a

fall is detected, or if the user fails to respond to the prompt within 60 seconds. In

this case, the following prompt appears on the watch and phone. See figure 3.2. The

prompt now appears on the watch for easier labeling. Users do not have to retrieve

the phone out of their pocket which makes labeling false positives easy, and more

importantly makes responding to true positives immediate and accessible. Some

elder people may forget where the phone is being placed after a fall or the phone

maybe in a nearby counter or in a purse which is not within reach. Additionally, the

phone could be locked, and it can be difficult for elder people to unlock smartphones

when in a hurry or in a panicking situation. The watch does not have to be

unlocked to respond to prompt. To label a false positive data, the user simply looks

at the watch and follows the two prompts. Being able to call for and see that help is

on the way directly from the watch is crucial to keeping the user calm if a fall

occurs. Of course, if the user is completely incapacitated, the timer will call for

assistance after 60 seconds. The labelled data will be be uploaded to the server

automatically if the user has given the consent when setting up the profile. The uploading of data is done in batches of 20 documents and when there is a wifi connection. We can configure the frequency of upload. For example, we can set for every 15 minutes, upload all data collected batch by batch. When the user deactivates the Fall Detection App, any remaining data collected during this period is uploaded to the sever before shutting down the App. For privacy concern, the only data being uploaded is the labeled accelerometer data with a UUID attached. No profile information is ever uploaded to the server.

3.3   Data Collection and Inference

The Fall Detection App on the phone and the watch App for retrieving sensor data from the watch work together to achieve fall detection. The watch has a linear acceleration sensor that is always on once the watch app is activated. Since we do not know when a fall may occur, it is crucial that we are always monitoring linear acceleration data of the user. Thus, strapping on a watch is the best solution for both comfort and continuous monitoring. The inference logic or detection of fall is performed on the phone due to the availability of better computation power. For our system, we opted to use Android WearOS watch and Android phone devices as they are commodity based hardware, very well adopted by the general public, affordable price and well maintained.

We capture sensor data from the watch by simply subscribing to the sensor using Android's "Sensor Manager". However, sending this data over Bluetooth is non trivial. The requirements for fall detection are that the sampling rate must be high and samples must be received in order. We determine that data need to be received/sampled about every 32 ms for the model to have sufficient information to make predictions of a fall accurately. We tried two methods of communication provided by Android's API for communicating between Android devices over Bluetooth. The first was the DataLayer API. This method involves "putting" data items in a shared buffer for other devices in the Bluetooth network to download. When we put a sample data from the watch, the "onDataChanged()" method was invoked to download the sample. Unfortunately, this method did not satisfy the high frequency requirement (i.e 32 ms). The buffer space in which the watch would put samples was overwritten every time a new sample was generated, this happened on average about every 20ms. The phone's "onDataChanged()" API was programmed by the vendor to be invoked every hundred or more milliseconds and thus was missing large amount of data that had been written over. The other method we tried was the Message API. This method involves sending messages over Bluetooth directly to a connected device. This is done by registering both devices with each other to send messages back and forth. While the initial registering of the connection is inconsistent and must be invoked many times, once a connection is

17

established, the communication between the devices is very reliable while they are in range. The phone received messages, on average, every 20 ms. This is consistent with the rate of data being produced at the watch's sensor. 20 ms is also sufficient for fall detection, which requires that data be collected for a duration of 32 ms to capture sufficient data pattern for a fall. This Message API is selected for our implementation.

With data successfully being streamed to the phone, the fall prediction is made using a pre-trained deep learning model created in TensorFlow 2.0. This model consists of 4 RNN (Recurrent Neural Network, a form of deep learning model that captures temporal relationship between data points) models in ensemble to make predictions for fall. Predictions are made on a window of data that is 35 samples in length. This window is designed as a sliding window with 99% overlap which means the window is slide by one sample in each prediction. This is done to ensure that every one of the 35 sample points in a window contributes to the pattern of a fall. Each prediction will output a probability of a fall and we average the last 20 probability of prediction together to infer fall or not fall. Basically, if the averaged probability reaches a threshold of 0.3, we determine that a fall has occurred. The 0.3 threshold was determined via experimentation and it can be adjusted for different users. The queue can be visualized in Figure 3.9. The old system stored the collected samples using a CSV file, keeping track of where to slide
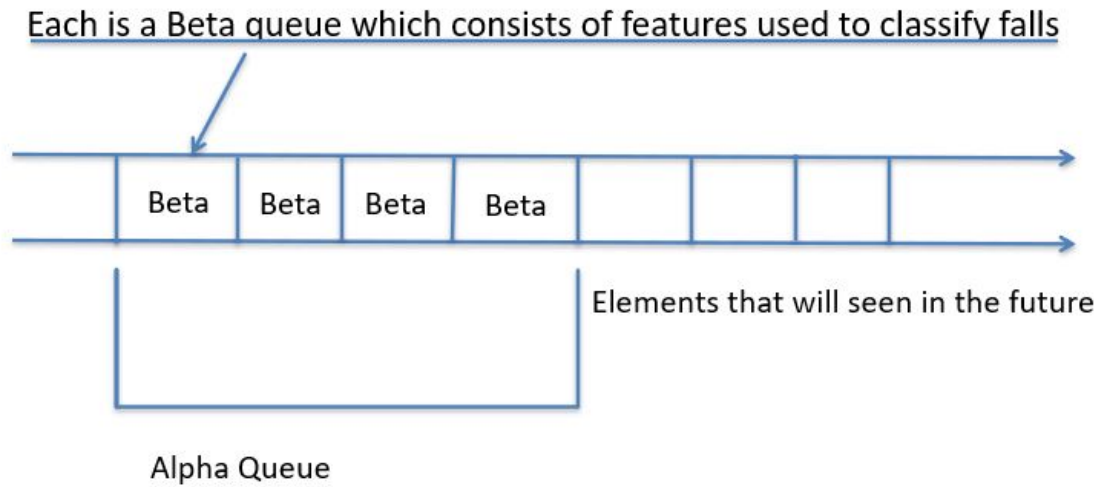
Figure 3.9: A visualization of the Alpha and Beta queue.

the window to by storing the line number of CSV file in memory. Predictions were made by reading the file from beginning and re-wind to the line number recorded. All labels for the data were saved to separate CSV files and were later combined together for training during pre-processing. This solution was inadequate for the long term usage of the App, since the file will get bigger and bigger when the App is being used continuously for a period of time. The prediction will get more and more behind because of reading and writing to a larger and larger file. We mitigate this problem by storing collected data in a queue data structure in memory until a prediction has been determined. This allowed us to only archive collected data once, and never have to read them from a physical file (which is very time consuming) in order to make predictions. Additionally, it provides a more precise method for labeling the collected data (False Positive or True Positive).

19

The structure of the queue is as follows: the main queue is designed with a length of 20 (the number of predictions we wish to average over). We shall call this the *alpha queue* from now on. Each item in the alpha queue is a queue of samples (linear acceleration data), these we will call them beta queues. Each *beta queue* is of length 35. A sample is simply one instance of linear acceleration data and the timestamp from when it was recorded. The beta queue starts as an empty queue and is populated every 32 ms with the latest data from the watch's sensor. Once the beta queue has reached its max length of 35 samples, the content is copied into the alpha queue. That same 35 samples in the beta queue is then used as input to the RNN Model for prediction, and the prediction result is stored in a *heuristics queue*. Since beta queue is limited to store 35 samples, so whenever a new sample has arrived, we pushed the new sample into the head of beta queue, and pop the oldest sample off the tail of beta queue. After 34 samples have been popped, we again save the content of the beta queue into the alpha queue and proceed to make another prediction, and store that prediction result in the heuristics queue. This process repeats continuously. Once the alpha queue is full (reached 20), we are ready to start making the final inference on the last 20 predictions by checking whether the averaged probability is greater than the 0.3 threshold.

```
1 import java.util.ArrayList;
2 import java.util.LinkedList;
```

```java
import java.util.Queue;


public class PredictionQueueDL extends Thread {
    final float threshold = 0.3;
    final int hBatchSize = 20;
    final int frequency = 1; //This controls overlap, we overlap
    34/35 samples
    final int maxQueueSize = 35;
    private Queue<String[]> beta = new LinkedList<String[]>;
    private int samplesAdded = 0;
    private int sampleSize = -1;
    private Queue<Float> pastHeuristics = new LinkedList<Float>();
    private Queue<Queue<String[]>> alpha = new LinkedList<Queue<
    String[]>>();

    public void enqueueSample(String[] newSample) {

            while (beta.size() >= maxQueueSize) { //pop the oldest
    value from the beta queue
                beta.poll();
            }
            beta.add(newSample.clone()); //put the latest sample in
    the beta queue
            samplesAdded++:
```

```
23
24          //wait for the queue to fill.
25          if (beta.size() >= maxQueueSize) {
26              if (samplesAdded >= overlap) {
27
28                  //reset counter
29                  samplesAdded = 0;
30                  //push this beta queue to the alpha queue
31                  alpha.offer(beta);
32                  //make inference on beta queue
33                  currentHueristic = predict(beta);
34                  pastHeuristics.add(currentHeuristic);
35                  currentHeuristic = average(pastHeuristics);
36                  if (currentHeuristic > threshold &&
    pastHeuristics.size() >= hBatchSize) {
37                      //this is a fall. Let's trigger a fall event
    .
38                      save(alpha);
39                      fallDetected();
40                      beta.clear();
41                      alpha.clear();
42                      pastHeuristics.clear();
43                      numOfHeuristics = 0;
44                      currentHeuristic = 0;
```

```
45                        }

46                        //not a fall

47                        else if (currentHeuristic <= threshold &&
    pastHeuristics.size() >= hBatchSize && !apha.isEmpty()) {

48                            while (pastHeuristics.size() >= hBatchSize)
    {

49                                pastHeuristics.pop();

50                                save(alpha.pop()); //save the oldest
     beta queue

51                            }

52                        }

53                    }

54                }

55            }

56        }

57    }

58 }
```

If the system infers that a fall has occurred, we empty the alpha queue by saving
the data to a storage medium with unknown label at this point. Otherwise, we pop
the oldest item (one beta queue) out of the alpha queue and save that as true
negatives. This system of queues can be used with any method of data archiving as
long as it can write the data fast enough. We tested saving data as CSV file and to

a Couchbase database and both were fast enough to perform the I/O in the same thread as the predictions without the total time exceeding 32 ms (total time was around 12-20 ms) which meant no extra overhead was required for multithreading.

3.4 Data Storage

To save data locally on the device (phone) we use a local Couchbase instance as it supports very fast I/O and can use the same document structure as the central server database. All data stored on the device is in a document structure described in the Database section (use section reference). There are 4 document types related to accelerometer data stored in the database. These correlate to true positive ("TP"), false positive ("FP"), true negative ("TN"), false negative ("FN") data. When a fall is detected, the entire alpha queue data is to be saved. We do not yet have a label for this data and are unaware if it is a true positive or false positive. The first step is saving this data immediately as it could contain True Positive fall data which is rare and valuable. To save the data we first remove overlapping data (due to 99% overlapping window), such that the samples are still in temporal order, and each sample is unique. To remove overlapping data, we pop each beta queue, take the first sample from each queue and discard the rest. However for the very last beta queue, we save all of the data in it. This gives a final size of 54 samples to be saved. Since we do not know the label of this data yet, we save it with a label of

"??" to denote that it is unclassified but important data. After getting feedback

from the user from the UI's prompt, we can now update the label of the "??" as

TP(True Positive) or FP(False Positive). Majority of data saved is true negative

data since fall is a rare event. Data is determined as true negative when it is popped

off the Alpha queue because the Alpha queue reached its max size. Thus the data

sent to be saved as true negative is a single beta queue of 35 samples. As described

earlier, this beta queue only contains one unique sample. We save the oldest item in

this queue and discard the rest to remove overlap. This gives us 1 new sample to

save to the database. However, to save I/O overhead, we write this data to the disk

at intervals of 375 samples. This means each true negative document will contain

375 samples. Finally, there is the case that a fall has occurred but was not detected,

in real world use, it is not going to be practical to immediately label false negative

falls as the elderly person will be alone. But, in controlled experiments, we designed

a a button on the phone that says "I just fell" that allows for the recording of a

timestamp marking the moment a fall occurred but is missed by the fall detection

system. This is used to later manually label sections of the data as fall data. This

false negative information is saved to database in a simple meta record consist of

timestamp and UUID. Expert knowledge will have to be used to identify at what

exact time the fall actually occurred and re-label some of the true negative data

accordingly. This is because the timestamps are generated based off the reaction

time of the person pressing the "I just fell" button. Because a human is pressing the button, the delay between a fall and pressing the button varies.

3.5 Database Synchronization

Synchronizing data to a server database is a core part of achieving automation. We needed to create a method to upload the labeled data to the server, re-train a new model, and then download the new model to the phone. The phone stores all data in Couchbase documents on the phone's disk. This allows for fast storage onto the device in a compact format as well as the ability to upload them using JSON format to the server's database. While Couchbase does have great tools for syncing mobile devices to a server database, due to university firewall policy we were forced to build our own method of syncing data via PHP. An overview of this design can be seen in figure 3.10.

We upload saved documents in large batches periodically to avoid continuous usage of the phones Wi-Fi connection. This can be done during the fall detection session, or when the device is charging. Once data are confirmed to be uploaded to the server database, we delete them from the phone's database to free up storage. A Tracker Document is designed to keep track of synchronization between the phone and the server databases. The Tracker documents are also uploaded to the server database, but never deleted from the phone. Profile information is never uploaded
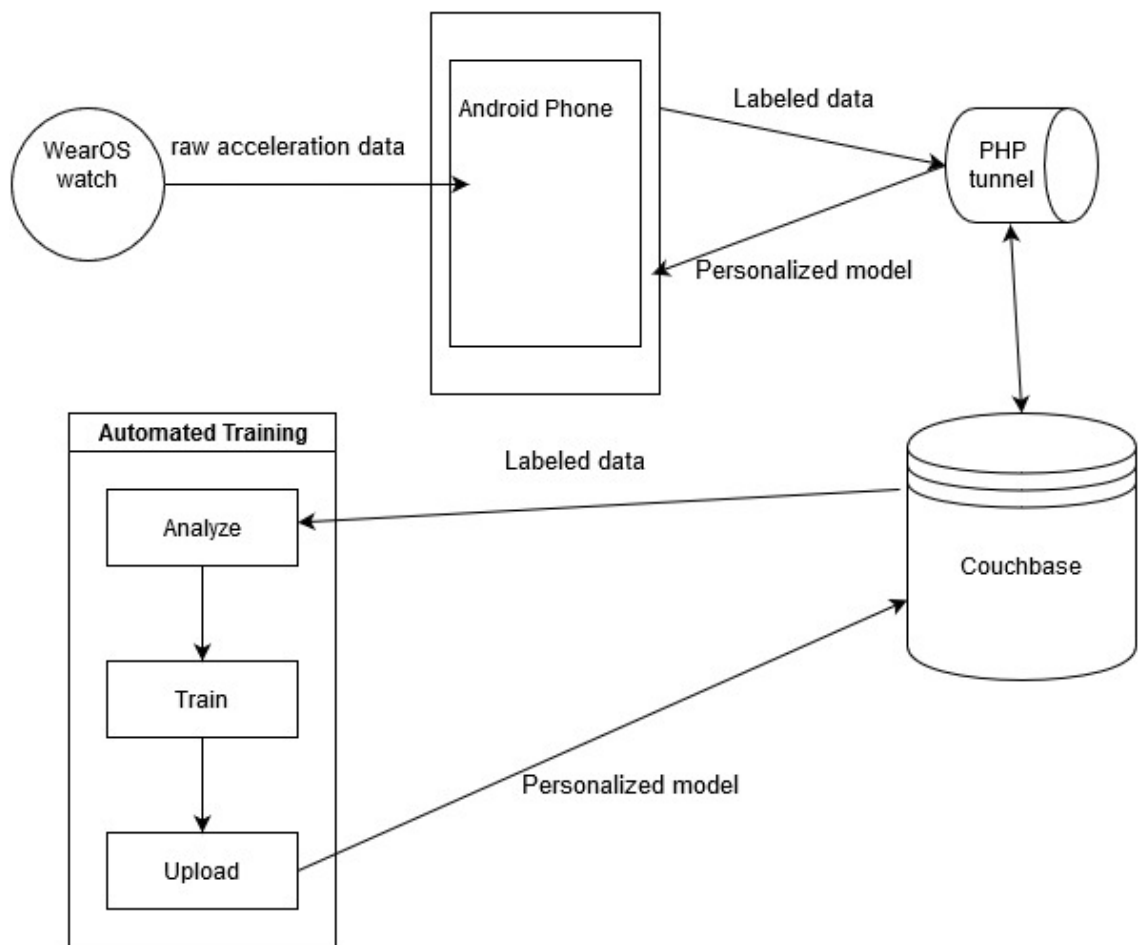
26

Figure 3.10:   Overview of the system.

to the database and remains on the phone until the app is uninstalled. All documents have a UUID field. A UUID is a 32 digit string that is generated using random number during a profie creation. Here is an example of a UUID: 9d94c957-5f4b-49ba-b555-b31db45ca9f7. The only association of the user to their data is with the UUID that is attached to all documents. This is to protect the privacy of users. Each time the user starts their fall detection system, we verify that the phone is connected to Wi-Fi. This is because when the user clicks activate, we first query the database for the latest model using their UUID. If the phone does not already have this model, it is then downloaded and loaded into memory. Fall detection will the proceed to start with the latest model.

3.6   Automation of Personalization

The archived data with the correct labels on the server can now be used to create new models. There are several steps involved in this process, from evaluating the labeled data in the database to uploading the new model to the phone automatically. All scripts used in the automation are written in Python. The system is divided into two parts; both parts are always running and depend on each other. The first part is the scripts which retrieve data from the database, evaluate the information stored in the database, determines if we need to train a new model, and requests a new model from the second part. The second part of this system handles training,

validation, and uploads the new model to the database. These parts are separated as they can be run concurrently and multiple queues on multiple GPU servers can be created. This allows the system to scale as necessary. In summary, the first part of the system handles all data analysis and interaction with the database. The second part of the system manages training and validation of the models.

3.6.1   Part One:

The following sections detail part one of the system.

3.6.1.1   Criteria for Re-Training

Automation begins with analyzing the archived FP data in the database to see if we need to train new models. Since each user's data is tracked using the Tracker documents, we simply need to grab the latest tracker document for each user. This tracker document contains the first and last document ID that contains data relevant to the model the user is currently using. For more detailed information, please see the database section. With these two document ID's, we can select all the data recorded while the user was wearing the watch running the latest fall detection model. With the relevant data retrieved, we want to evaluate how the current model is preforming. If the model is preforming well, there is no need to retrain. Our method for training put emphasis to retain high recall performance. Since true

29

fall is a rare event, we cannot expect true positive data to be captured, so we do not expect to improve recall but just ensure that it does not get worst. However, we need to ensure that there are improvements to precision to reduce the number of false positive alarms. In previous work, it was determined that best metric for evaluating fall detection precision in real-world is the spike score. A spike is a point of data in which the acceleration is greater than the average acceleration squared. Spikes are located by evaluating each datapoint in chronological order. When a spike is located, the next 16 datapoints are ignored as they likely contain the peak data points of high acceleration data and would all be considered spikes. The total number of spikes is calculated for the data and is denoted as num_spike. The spike score is a relationship between the number of false positives (num_fp) and the number of spikes such that the greater number of spikes with the lowest number of false positives is the best score. This gives the equation (1 – num_fp / num_spike). We subtract from the number one to meet the "bigger is better" paradigm. Because false positives typically only occur when a spike is present, num_spike is $>=$ num_fp. We choose .98 to be the threshold for which if the spike score achieves, we no longer need to train. This means that 2 percent or less of all high acceleration activities result in a false alarm.If the spike score does not reach the threshold, the data is sent to be prepared for training. There are two things we consider in preparing this training data. First, does the system already have a test set generated for the user?

And secondly, we need to ensure we trim the dataset such that it contains mainly high acceleration data. We cover these in the next sections.

3.6.1.2 Data Trimming

Adding too much low acceleration data into the dataset will result in highly unbalanced fall data set. Thus we need to trim off most of the low accelerometer data as it makes up a large majority of all captured data. This trimming process removes data that is not within 750 datapoints from a spike (roughly 24 seconds). Additionally, each spike has a buffer around it that is of size 250 datapoints (roughly 6 seconds). This means after training, each spike is at most, 30 seconds away from each other. The data remains in chronological order; however, now all long periods of low acceleration data is removed. We leave some low acceleration data as this is representative of falls. There is always some low acceleration data before and after a fall, because we use an LSTM layer in the model we need to make sure we keep this pattern. Additionally, it is important to retain a variety of ADL data, so some low acceleration data should be included in between the high acceleration data.

3.6.1.3 Dataset Creation

In preparing the dataset, the first thing we do is query the database for a test set. If a test set does not exist for the user, we will take a portion of the current data and

reserve it for the test set. The first 200 spikes are reserved for the test set, the rest is used for training. This test set is appended to the generic test set before uploading, which leaves a final size of 2.45MB. Both the test set and the training set undergo the same trimming process. If we needed to use some of the current data to create a test set for the user, we need to guarantee that there is still enough data left over to use for training. The condition for training is that the training set must contain at least 50 spikes because 50 spikes allows for sufficiently diverse set of ADLs to be included. If this condition is not met, the entire process for the user is canceled at this point and nothing is changed in the database. Otherwise, a new test set is created, it is written to a CSV file locally on the server and uploaded to the database. If a test set is available, it is downloaded from the database. The database is then queried for the last training set used for this user along with its version. If no previous dataset existed, the generic set is used since the user must have been using the generic model. This previous dataset is downloaded and loaded into memory. The version number is retained in memory and incremented by one. The new training set is appended to the old dataset, then written to a CSV file with the new version number. This CSV is uploaded to database as the latest training set. Finally, the file paths of the new training set and the testing set are passed to the training queue along with the user ID and the current version number. We chose to pass the file paths instead of the file contents to reduce memory usage since

there is an unknown number of users that could be queued.

3.6.2 Part Two

The following sections detail part two of the system.

3.6.2.1 Model Generation

Creating models in TensorFlow using the TFS (Training From Scratch) method has been described in earlier works and described in section?? . To recap, the TFS method discards the previous model files and trains a new model from a random initialized state on the new dataset. This new dataset is a combination of the original dataset with new data appended to the end as described in the trimming section. This training is very expensive in terms of server time, with a typical training time of 28 minutes (including validation). To manage this, we utilized a queue to schedule training automatically. Jobs are submitted to the queue as an array consisting of the UUID, version number, training dataset path, and testing dataset path. This means the queue is capable of growing very large and we do not limit the maximum size of the queue. The training thread periodically checks the queue every few minutes. If the queue is populated, it runs each job one at a time until the queue is empty again. Because the queue is synchronous with the model manager, it is possible the queue can grow while a job is being processed.

Processing a job is the same as running the pipeline that includes loading the dataset, apply preprocessing, train and validate the model. Each job is processed one at a time per GPU. This takes about 30 minutes. In our experiment, we only have access to one GPU server, so all jobs ran one at a time. If we had a second GPU, jobs could be run two at a time. This scales linearly.

3.6.2.2 Model Validation

Once training is complete, it is time to upload the models. Validation of the models is very important and needs to be taken into account when uploading models to phones. We do not want the system to update a user with a model that has poor recall. This could be life or death for the user as a FN (False Negative) is dangerous. Therefore, we need to ensure that the model we are uploading is better than its predecessor. We determine that a model is better than its predecessor if when generating a PR curve, there is an increase in precision and a recall of .9 or above. Thus, the system checks by comparing the PR curve of this new model against the current best model for the user. If the new model is better, we replace it with the previous model. Otherwise, we save the new model to the database and do not set it as the best model. These PR curves are generated by executing both models over the same test set. This test set contains a reserved subset of the generic data which is not included in any of the training sets as well as some additional personal ADL

data that was set aside from training. This ensures a fair comparison between all the models, including the generic model. The phone can now query the database for the "best" model and download it if necessary. This ensures the phone does not download a model that could miss falls even if it has a lower false positive rate.

CHAPTER IV

DATABASE

The database used for this project is Couchbase. Couchbase is an opensource NoSQL document-oriented database. We utilize a Couchbase server hosted on a university Ubuntu machine as well as a local Couchbase instance stored on the android device. Due to a university policy, all external access to a database needs to be tunneled through PHP. Couchbase was chosen for its ability to scale, fast read/write performance, and JSON formatting. This makes it easy to store data on the phone and upload/download through PHP. The NoSQL structure allows for us to quickly query just the data we need across multiple types of documents. This means we can store additional data in the database, such as data from other sensors, without impacting query execution time. Additionally, since we can utilize document types, we can store TP, TN, FP, FN data in separate documents and re-compile them later. This means less processing on the phone and the ability to query any combination of types of data. Fast processing of data on the phone is essential since we cannot allow the application to fall behind. Currently we save all data as any of it could be important. Data needs to be saved to the disk at least as fast as we receive it, and ideally on the same thread as the queue. This minimizes

overhead and minimizes battery consumption. Document structure for data storage is the same on the phone and on the server. The server utilizes two Couchbase "buckets" to separate all the raw accelerometer data from the dataset and model files. This is because in Couchbase it is advantageous to have similar documents in the same bucket for better indexes and faster queries. Documents of type: "tracker", "TP", "FP", "TN", and "FN" are in one bucket and documents of type: "model", "csv", "blob" are in the other bucket. This has no significance other than the benefit of faster queries and better organization. The following diagrams are visuals representation of each type of document in the database with descriptions describing their purpose. It should be noted that all documents in the database contain a Document ID. The is a unique key for all documents in Couchbase database. This is simply a unique identifier and could effectively be a random string. Most documents also contain a UUID field. This is a unique identifier that is generated randomly for each user in the system. This allows us to keep data associated with a user without using any personal information.
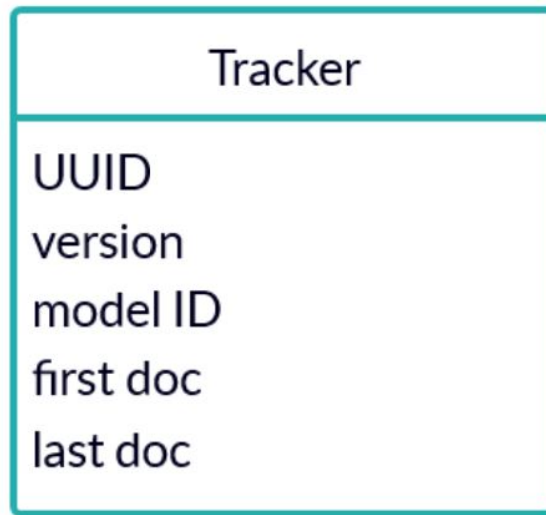
Figure 4.1: A visualization of the tracker document.

The tracker document is created by the Fall Detection App on the phone. It associates the Fall Detection model the app was using when it recorded the data. As new models are downloaded to the phone, newer tracker documents are created to track which data was recorded with which model. It contains the ID of the first and last document recorded when using the model. It also contains the model ID, which is the document ID of the model file the app is currently using. It contains a version, which allows the tracker documents to be ordered from the oldest to newest. Of course, it also contains the UUID which tells us which user this tracker document belongs to.

Figure 4.2: A visualization of the true negative document.

The true negative document is by far the most common document in the database. It holds the majority of accelerometer data. The comparable time field is a java timestamp that allows us to easily order documents in the order that data was collected. Since comparable time is generated when the document is created, in some cases we also use the "first timestamp" field for the same purpose to verify order as this value is generated at the time the sample was collected. The fields containing accelerometer data and the timestamps field are all arrays of exactly 375

samples. These fields are in chronological order such that each item at a given index

is associated with items at the same index in the other arrays. This is how we retain

the order of all the samples collected. The length of 375 samples was chosen as it is

a large enough to reduce overhead of creating many documents, but small enough to

be easy to transfer in batches and download from the database.

Figure 4.3: A visualization of the false positive document.

The false positive document is the same as the true negative document except

that it's length for the arrays is fixed at 54. This length is explained in the data

collection section and is a result of the amount of data needed to determine if a fall

has occurred.

Figure 4.4: A visualization of the ?? document.

The "??" document is the same as the true negative document except that it's

length for the arrays is fixed at 54. This length is explained in the data collection

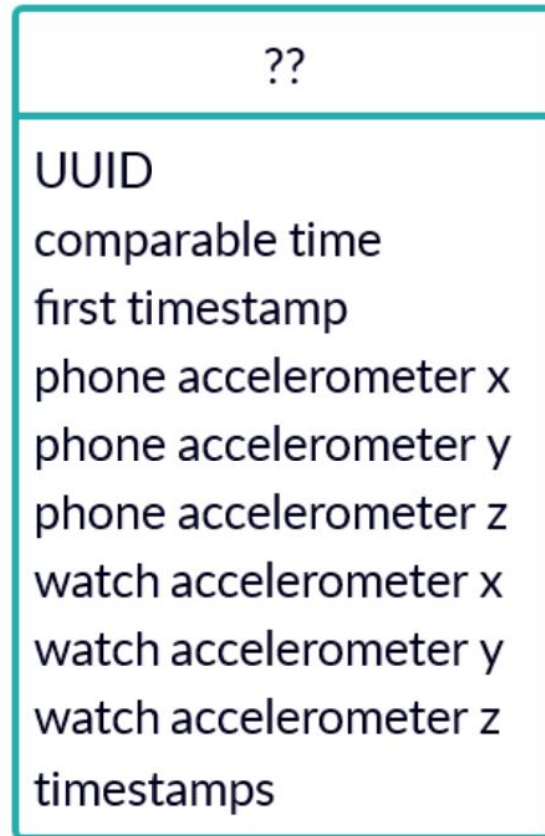section and is a result of the amount of data needed to determine if a fall has

occurred. The type is "??" because we only know that the model has detected this

as a fall, but the user was unable to label the data. This is a rare occurrence but can be the result of an app crash, or a lost Bluetooth message that carried the label information from the watch to the phone. Typically, this document only exists on the phone as a ?? type for a short amount of time until the label is retrieved from the user and they type of the document is updated to the correct label.

TP

UUID
comparable time
first timestamp
phone accelerometer x
phone accelerometer y
phone accelerometer z
watch accelerometer x
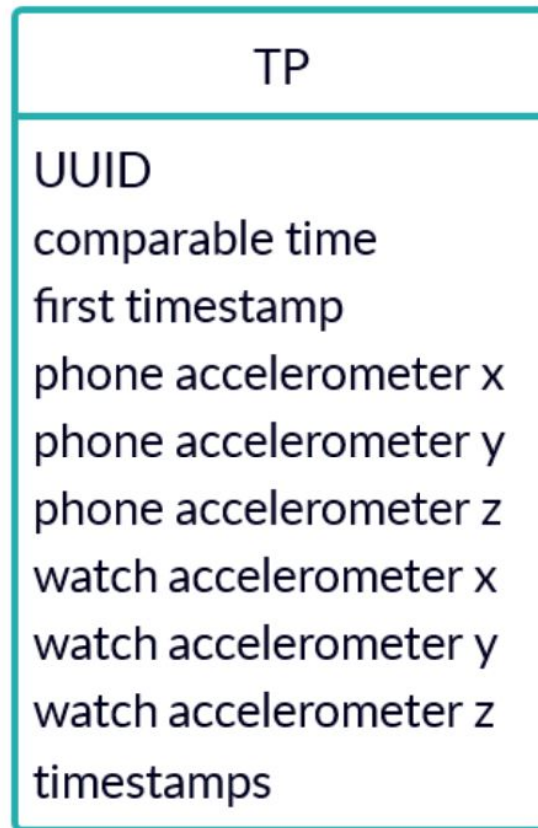watch accelerometer y
watch accelerometer z
timestamps

Figure 4.5: A visualization of the true positive document.

The true positive document is the same as the true negative document except that it's length for the arrays is fixed at 54. This length is explained in the data

collection section and is a result of the amount of data needed to determine if a fall
has occurred.

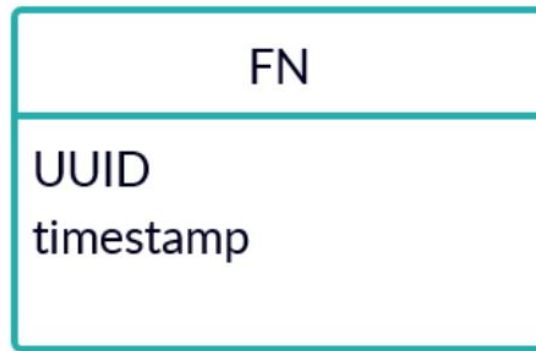Figure 4.6: A visualization of the false negative document.

The false negative document is simply a recording of the timestamp when the
user presses the "I just fell" button. It is not utilized by the automated system as it
is impossible to know what data the fall is actually occurring in. Expert knowledge
is required to review the data near this fall and determine where the false negative
truly is.
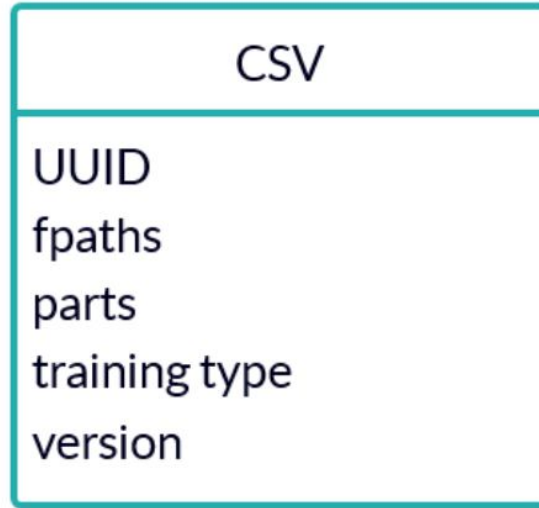
Figure 4.7: A visualization of the CSV document.

The CSV document is used to store the datasets. They contain a training type parameter that identifies which training strategy was used, in this thesis they were all "TFS". The version is the version of the dataset which incremented the same as the model file. Except the for test dataset, that version is always -1. The fpaths is the file name of the csv file. The parts is all the document IDs for the blobs that holds the bytes of the file. Small files can be contained in one blob, but for very large csv files, multiple blobs are used.
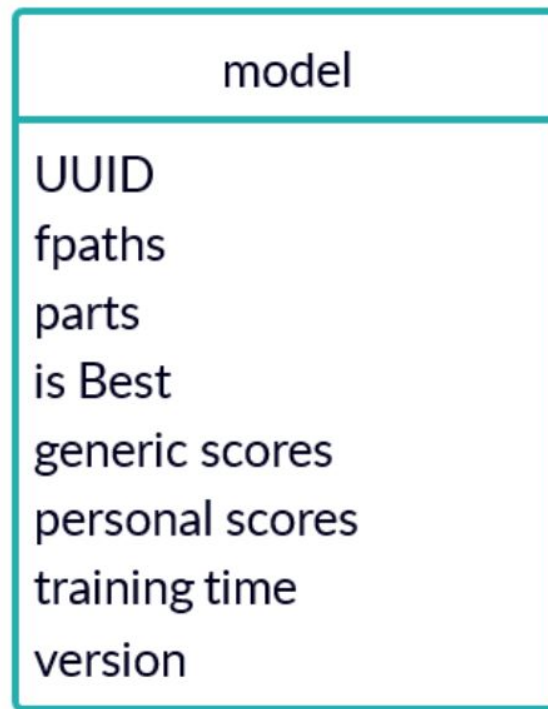
Figure 4.8: A visualization of the model document.

The model document is used to store models. The fpaths field stores the filenames of all the models (in our case, 4 model names). The parts field holds the respective document IDs for the blobs that hold the data for each model. The version field keeps track of the latest model while the "is Best" field is a Boolean that determines which model is the best. Only one model per user can have this field be true. The generic scores field is a map of all the statistics generated during offline validation (including the PR curve) when the model was tested on the generic test set. The personal scores is the same, except for when the model was

tested on the personalized test set. The training time is the number of seconds it
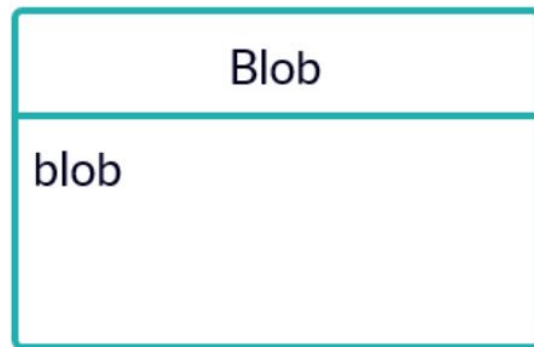took to generate this model.

Figure 4.9: A visualization of the blob document.

The blob document is used to store the bytes contained that make up files.
This is a byte array stored in the blob field. Blobs made for model files also have an
extra field, "weight", which stores the weight of the model (this is used for the
ensemble prediction).

CHAPTER V

EVALUATION

To evaluate our system, we first confirm that the system as a whole can repeat results of the previous manual method for personalizing fall detection models. We do this by recreating the previous experiment with one user from the previous paper [1]. We ask that user to wear the watch as he performs a set of prescribed ADLs, and provides feedback for false positive predictions. We then add the false positive feedback data to the training set and train a new model for that user and upload that model to the phone. The user is then asked to wear the watch with this new model for 5 more hours with moderate amount of movement. At the end of this period, we calculate the spike score for that user. The re-trained model got a spike score of .97. This score is close to the previous manual system that achieved a spike score of .98. The user also completed the simulated fall test and achieved 18/20 falls. This is consistent with the manual system.

We then proceed to evaluate the system's ability to continuously personalize fall detection. This means we should not control the data collection sessions like in the previous session. To do this, we ask the user to wear the watch for one hour for a calibration period. This calibration period is expected to generate a lot of false

positives which the user will have to label (by giving feedback). At the end of the calibration period, a new personalized model is created and uploaded to the phone. From this point on, the user will wear the watch daily. During the night, when the watch is being charged, the training scripts on the server will run to see if there is a need to train a new model for that user and if so, it creates the model and uploads it back to the phone. This process is explained in the methodology section 3.6.1. The user wore the watch each day for one week with remarkably few false positives. At the end of the seven days testing period, we found the system did not create any other model besides the one created after the initial calibration period because the spike score was consistently around .99. This means while our goal is to continuously personalize the model, in actual fact, only one personalized model was ever created. This personalized model has low false positives but when the same user conducted the simulated fall test at the end of the week, only 7/20 falls were detected (a very low recall). This indicates that there is a problem with our re-training strategy. The main difference between this test and the previous test is that we do not limit the amount of data collected in the calibration period. This means our training method is not suitable for automatic calibration and needs a controlled calibration.

During this one week testing period, we recorded the average time the battery lasted from a full charge until the watch died. Since the watch always died before the phone, we stopped recording battery life for the phone once the watch died. The

watch lasted on average 3.7 hours (taken from 3 times the watch was worn continuously from 100 percent to 0 percent) and the phone on average had 41 percent battery left after the watch died.

Finally, we also evaluate the scalability of the automated personalizing system. We utilize two stress tests against the system. The first tests Couchbase ability to store large number of documents quickly. We utilized a second server connected to the same network as the Couchbase server via gigabit Ethernet. This server uploaded data at full gigabit speed until long after the database size exceeded the memory cache of Couchbase. Couchbase was able to successfully save all data uploaded to it at the maximum speed allowed by our hardware.

We then test the system ability to process personalizing multiple users' fall models. We do this by uploading feedback data for 21 users for personalizing each of their model, allowing the system to run all night, and then confirming that new models were generated for all of these users. It took 11.18 hours to train 21 users.

CHAPTER VI

FUTURE WORK

Our system is successful in robustly collecting labeled data from users and automatically generating new models. We expect our system to be valuable in future work to collect real datasets and test other strategies for fall detection. It will enable fast and robust deployment of fall detection models. It will also allow researchers to get immediate feedback on the performance of the models. The user interface on the watch and reliable data continuous collection on the phone should allow anyone to be able to use the system. Through the PHP synchronization, data can be uploaded from anywhere in the world. We also show our system is capable of generating and versioning new datasets and models automatically and can deploy these models via the same PHP synchronization. Model generation scales linearly and is only limited by the number of GPUs. As you increase the number of GPUs, the capacity for training new models increases linearly. The end result is a continuous personalized fall detection and data collection system that can scale indefinitely.

There are only three challenges faced when implementing our system. The first is that our validation strategy using PR curve failed to accurately represent the

model's performance in real life. A personalized model could have a significantly worse PR curve yet preform much better in real world testing. At first, our test set only consisted of the reserved generic data. We considered that this worse PR curve in the generic model may be because the improvements are expected on personal ADL data and not generic ADL data. We then adjusted the system to reserve some personal ADL data as described earlier in this paper. However, this brought no improvements. We believe that this is due to the training parameters (number of epochs, number of neurons, batch size) to be highly optimized for the generic model. Since we are training the nature of the training set, we can't expect these parameters to be optimal for the new training set. Further investigation would be required to be certain of this. However, it is likely this is a drawback of TFS training and PR curves could be a good method for evaluating other training methods. Another issue faced is that TFS training fails to retain recall when the number of added personal ADLs grows too large. We know this can happen within one hour of high acceleration activities, however we do not know exactly at which point this occurs and we would not expect this point to the be same for all users and all activities. We know from previous work that some added personal ADLs provides great improvements to false positive rates and we were able to replicate that result once again in this study. However, we cannot devise a way to resolve this problem in a practical way. Without a way of validating a model offline, we can't

know if a model is actually better or worse until testing it. Additionally, it would be impractical to generate many different models from scratch for each user just to identify how much training data is needed. In future work we plan to address the problem of TFS training and online validation by utilizing incremental training. In this form of training, we will simply reload the generic model into TensorFlow and train a few personal ADLs at a time, with some generic true positives and ADL mixed spliced in. Then re-freeze the model. This should allow us to retain the highly optimized parameters for the generic model (hopefully solving the PR curve issue), as well as efficiently let us increment our added ADLs until there is no longer an improvement since we don't have to start from scratch. In conclusion, we believe future work researching incremental training will mitigate both of these issues.

Finally, there remains the issue of poor battery life. Although our system proved to be an effective means for collecting real data in any environment. It suffers from an unacceptable battery life. Our target battery life is 8 hours, such that we can reasonably expect an elder person to be able to wear the watch all day without having to remove the watch to charge. We hope that in future work we can resolve this issue by taking advantage of the optimized TensorFlow 2.0 models by running them directly on the watch instead of uploading the data to the phone. Hopefully, the significant reduction in Bluetooth communication will provide a significant boost to battery life.

CHAPTER VII

CONCLUSION

Our work demonstrates a feasible way of collecting a real dataset of accelerometer data from elderly people in hopes of being able to generate real datasets for future algorithms. While our current algorithm is still not ready for deployment, we show that that we can achieve a fall detection system with great recall and a very small amount of false alarms via automated personalization. This work paves the way for creating a fall detection system that can be tailored to each elderly patient. The infrastructure for collecting and labeling data is reliable and secure, while preserving patient privacy. It requires no intervention from developer and after a brief calibration period, is convenient for a user to wear. The system provides a robust way of deploying fall detection models with the ability to immediately review performance and generate new models. All data is organized in a Couchbase server where it is versioned and labeled and can be downloaded to .csv files.

BIBLIOGRAPHY

[1] "Personalized fall detection system," *To appear in the proceedings of the 5th IEEE PerCom Workshop on Pervasive Health Technologies, Austin, TX*, 2020.

[2] R. Anita and K. Anupama, "A survey on recent advances in wearable fall detection systems." *BioMed Research International*, 2020. [Online]. Available: http://libproxy.txstate.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=edsdoj&AN=edsdoj.5da88a9647b24010abace1695311550e&site=eds-live&scope=site

[3] F. Riquelme, C. Espinoza, T. Rodenas, J.-G. Minonzio, and C. Taramasco, "ehomeseniors dataset: An infrared thermal sensor dataset for automatic fall detection research." *Sensors (Basel, Switzerland)*, vol. 19, no. 20, 2019. [Online]. Available: http://libproxy.txstate.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=mdc&AN=31640148&site=eds-live&scope=site

[4] M. Shastry, M. Asgari, E. Wan, J. Leitschuh, N. Preiser, J. Folsom, J. Condon, M. Cameron, and P. Jacobs, "Context-aware fall detection using inertial sensors and time-of-flight transceivers." *2016 38th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC), Engineering in Medicine and Biology Society (EMBC), 2016 IEEE 38th Annual International Conference of the*, pp. 570 – 573, 2016. [Online]. Available: http://libproxy.txstate.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=edseee&AN=edseee.7590766&site=eds-live&scope=site

[5] G. Demiris, S. Chaudhuri, and H. J. Thompson, "Older adults' experience with a novel fall detection device." *TELEMEDICINE AND E-HEALTH*, vol. 22, no. 9, pp. 726 – 732, 2018. [Online]. Available: http://libproxy.txstate.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=edswsc&AN=000383602600003&site=eds-live&scope=site

[6] A. Fanca, A. Puscasiu, D.-I. Gota, and H. Valean, "Methods to minimize false detection in accidental fall warning systems." *2019 23rd International Conference on System Theory, Control and Computing (ICSTCC), System Theory, Control and Computing (ICSTCC), 2019 23rd International Conference on*, pp. 851 – 855, 2019. [Online]. Available: http://libproxy.txstate.edu/login?url=http://search.ebscohost.com/login.aspx? direct=true&db=edseee&AN=edseee.8885996&site=eds-live&scope=site

[7] V. Mirchevska, M. Luštrek, and M. Gams, "Combining domain knowledge and machine learning for robust fall detection." *Expert Systems*, vol. 31, no. 2, pp. 163 – 175, 2014. [Online]. Available: http://libproxy.txstate.edu/login?url=http://search.ebscohost.com/login.aspx? direct=true&db=bsu&AN=95980090&site=eds-live&scope=site

[8] I. Chandra, N. Sivakumar, C. B. Gokulnath, and P. Parthasarathy, "Iot based fall detection and ambient assisted system for the elderly." *Cluster Computing: The Journal of Networks, Software Tools and Applications*, vol. 22, no. Suppl 1, p. 2517, 2019. [Online]. Available: http://libproxy.txstate.edu/login?url=http://search.ebscohost.com/login.aspx? direct=true&db=edssjs&AN=edssjs.F1868389&site=eds-live&scope=site

[9] T. Pham Van, T. Duc-Tan, N. Dinh-Chinh, A. Nguyen Duc, D. Dang Nhu, S. El-Rabaie, and K. Sandrasegaran, "Development of a real-time, simple and high-accuracy fall detection system for elderly using 3-dof accelerometers." *ARABIAN JOURNAL FOR SCIENCE AND ENGINEERING*, vol. 44, no. 4, pp. 3329 – 3342, 2018. [Online]. Available: http://libproxy.txstate.edu/login?url=http://search.ebscohost.com/login.aspx? direct=true&db=edswsc&AN=000462305100030&site=eds-live&scope=site

[10] J. R. Villar, E. de la Cal, M. Fañez, V. M. González, and J. Sedano, "User-centered fall detection using supervised, on-line learning and transfer learning." *Progress in Artificial Intelligence*, vol. 8, no. 4, p. 453, 2019. [Online]. Available: http://libproxy.txstate.edu/login?url=http://search.ebscohost.com/login.aspx? direct=true&db=edssjs&AN=edssjs.4769A85D&site=eds-live&scope=site

[11] T. R. Mauldin, M. E. Canby, V. Metsis, A. H. H. Ngu, and C. C. Rivera, "Smartfall: A smartwatch-based fall detection system using deep learning," *Sensors*, vol. 18, no. 10, 2018.