Using Keystroke Dynamics to Authenticate a User Based on their Typing

Jack Francis

April 26, 2022

0.1 Abstract

Contents

Chapter 1

Introduction

Chapter 2

Survey of Literature

In order to create this project, it was necessary to do some research into the project.

In order to do some research onto what a system based on keystroke authentication would involve, I read *Time-frequency analysis of keystroke dynamics for user authentication*[1] which presented a highly theoretical one time system that used a maths based approach using a dynamic time warping algorithm along with a Wigner distribution in order to perform similarity calculations. The presented system only worked with small word samples. AUnfortuntly, the system wasn't a continuous system, but the paper still provided value in how to implement a validation measure. The validation function in my produced system is a highly modified version of the system set out in this paper. I decided to rely on this paper because it was published recently, well-cited, well researched and presented a solid approach I couldn't find any problems with.

Another paper I read in the opening stages of this project is *Keystroke dynamics-based authentication service for cloud computing: Keystroke Authentication Service.*[2] This paper showcased a continuous machine learning approach as opposed to the one time maths based approach showcased previously. Whilst I did not use the approach set out in this paper, the different viewpoint showcased was useful when making the decision in what approach to take. The reason I didn't end up choosing this approach because whilst the paper is relatively recent, it is not well-cited and the paper lacks detail in what is actually happening. Furthermore, I choose to go with a maths based approach because I felt this was more challenging. I decided I couldn't rely on this paper because of these reasons and as such it wasn't that useful to me.

Once I had made my decision in what approach to take, I read papers about algorithms used in certain steps of my validation function. The paper *Fast-DTW: Toward Accurate Dynamic Time Warping in Linear Time and Space*[3] presented the Dynamic Time Warping algorithm. The paper has been well-cited, presented the algorithm in great detail, was easy to understand and really helped me get an understanding of what the algorithm does. The paper is also peer reviewed and the authors well reputed. Therefore this paper was useful to me as it allowed me to get an understanding of one of the key algorithms in my project.

Whilst doing research on Dynamic Time Warping, I also read *Fuzzy clustering of time series data using dynamic time warping distance*[4] which used

Dynamic Time Warping in order to improve their clustering algorithm. Whilst the use case is not exactly the same as mine, the paper was still useful in order to see how they had prepared their data and what the results after going through the algorithm was. The paper is not well-cited but still proved to be useful due to it's subject matter and the clear approach stated within. This paper and the previous one allowed me to get a clear and deep understanding of how to implement and what the results are of the algorithm and as such are useful.

I also did some research into the Wigners Distribution which I was planning on implementing inside the validation procedure. The paper *Wigner Distribution*[5] originally published in 2009 provided an exploration of this distribution on the context of quantum physics. Whilst the subject matter was different and it was difficult to understand to of the quantum maths elements which limited my understanding, I would still class this paper as useful. It allowed me to make a decision if this was strictly necessary to implement in my program and if the proposed algorithm provided enough accuracy to overcome it's performance hit. *The Wigner Distribution: A Time-Frequency analysis tool*[6] further increased my understanding of the distribution and proved useful in explaining details which were not explained by the previous papers.

Research into open-source examples was also undertaken with a look at a keystroke analysis system developed by Nikolai Janakiev[7]. He created a theoretical one time system which uses machine learning in order to create models and work out how a user typed. This was useful as it was interesting to see how other people have approached the project. The approach detailed in there is based on the paper *Comparing anomaly-detection algorithms for keystroke dynamics*[8] which was of huge help when formulating my approach even though I didn't follow this approach.

The final paper I read was *Keystroke Dynamics-Based Authentication Using Unique Keypad*[9] which showcased an approach using a unique keypad that the user uses to login. Whilst the keypad element isn't useful in my case, the approach they used to record and process the data from the keypad was useful and influenced my decision making when it came to the system. However, I didn't end up relying on this paper due to the highly theoretical nature of the system and it was not capable of being easily translated into my system.

Chapter 3

Design and Implementation

I decided to create my system in python due to is flexibility, the large amount of documentation available, and my familiarity with it. Furthermore, python contains multiple packages such as NumPy, which makes doing advanced maths simple, easy, and provides excellent performance. Due to how my program operates, many data points are generated for each word. As such, having fast packages and function is essential to keeping the program as lightweight as possible and avoiding a potential performance impact on the user.
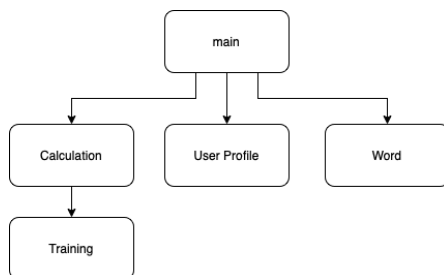

Figure 3.1: Basic Class Structure

It also makes interacting with the various elements of my system simple due to a large number of packages available. I used an object-orientated approach in my system. Figure 3.1 is a visual representation of the class structure used in my program. The main file is where the program starts and runs from. When the raw data is collected in the main file, it is passed into the Calculation class where the preprocessing, word forming, and validation procedures happen. Training inherits from the Calculation class and shares all methods and attributes with a saving to file function simply added on. The User profile is self-explanatory and just provides a simple way to store the user's details. The Word class contains details of each word produced by the program. It's a simple storage mechanism with many functions, such as generating the KD signal and compression attached among others.

My system is a relatively complex system that differs slightly from the one set out in my interim report. My plan from my interim report is shown in figure 3.2. It's significantly more straightforward than my actual system plan shown in figure 3.3. One main difference is the lack of a database to hold the user login information. Upon creating such a database, I felt this was unnecessary and, as such, omitted this from my final system in favour of using a Windows-based authentication system. This was done as too many authentication systems can confuse the user, and there is no better security than the windows one. A new
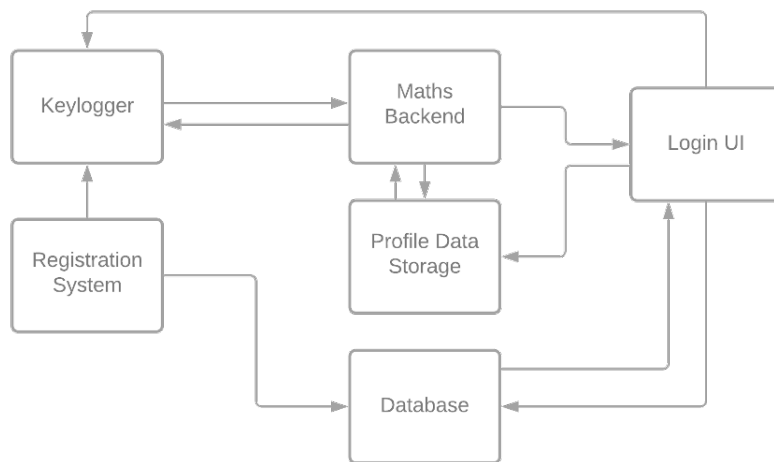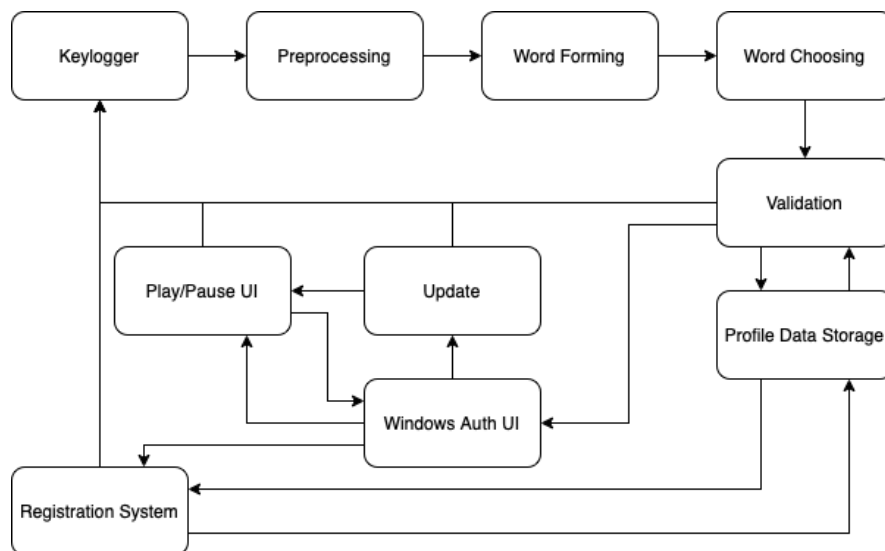
Figure 3.2: Original System Plan

Figure 3.3: Actual System Plan

feature in my final system plan is the play/pause UI which allows the user to pause the system should they do something involving sensitive information they do not want to be recorded. Another new step is the update step; if the user initially fails validation but then re-authenticates correctly using their windows username and password, the relevant word objects are updated. I went for a maths-based approach rather than using popular approaches such as machine learning because machine learning approaches typically have a lengthy learning phase that consumes a lot of the user's time and a lot of the user's system resources. Furthermore, a maths-based approach allows the system to be much more lightweight, and it's simpler to develop and for the user to understand what is going on.

The entire system works at intervals. An interval is defined as a period in which the data is collected. For example, if the interval were 60 seconds, then data would be collected for 60 seconds, processed, and then the next interval would start. If the interval was shorter, then while the program would be theoretically more secure, it would suffer from a lack of data collected, which would affect the system's overall accuracy. Furthermore, having a shorter interval would lead to the program doing many more calculations, which would affect the user's system performance. Choosing an interval of 60 seconds allows me to balance accuracy and performance. While I tried other interval times such as 5, 30, and 45 seconds, these all suffered from a lack of data collected or started to affect performance severely. In particular, the 5-second interval never collected enough data to be able to make a call and sometimes took longer than five seconds to perform the calculations, which started to cause a problem due to the gaps in coverage.

Each step is explained in more detail below. The user, upon first login, starts in the registration procedure. It is here that the system learns how the user types. It creates the profile data storage and then stores the data learned inside. The system then records the user's keystrokes for the interval. Once this has been completed, the data goes through the preprocessing steps and is then formed into words. For each interval, only a select few words are chosen for validation. Once the words are chosen, they are put through validation. This involves loading the learned data from the profile data storage and comparing the two using validation procedures. If they pass validation, the system moves on to the next interval and repeats the steps. If the words fail validation, the user is asked to re-authenticate using windows authentication UI. If they pass this, the failed words are updated in the validation procedure, and the system continues. If the user fails, a new user is created using the registration system.

While all this is happening, the play/pause UI is running in a separate thread. Should the user ask the program to pause using this UI, they are asked to re-authenticate, and if this succeeds, the whole system is stopped until the user asks the program to continue again.

The validation implementation is a rough version of one shown by Ramin Toosi and Mohammad Ali Akhaee in their excellent paper 'Time–frequency analysis of keystroke dynamics for user authentication'. [1] The paper is theoretical and describes an approach for performing validation on one word and then comparing them. It is, in essence, a one-time system, while mine is a continuous system that aims to keep the user safe. I have implemented their validation approach in my project while adding data gathering, word-forming, word selection, word storage, and update functionality. In figure 3.4, you can see the two

maths backends compared to one another. The main difference is that Wigners Distribution is omitted in the actual system. This is because this distribution is slow in my use case, and the improvements in accuracy are not worth the performance loss. The other difference between the two is the introduction of the Euclidean distance. This was introduced as a secondary validation measure to allow greater validation. This and the Correlation coefficient feed into the final similarity score.

The validation implementation is a rough version of one shown by Ramin Toosi and Mohammad Ali Akhaee in their excellent paper 'Time–frequency analysis of keystroke dynamics for user authentication'. [1] The paper is theoretical in nature and describes an approach for performing validation on one word and then comparing them. It is in essence a one-time system whilst mine is a continuous system that aims to keep the user safe. In my project, I've modified and implemented their validation approach whilst adding data gathering, word forming, word selection, word storage and update functionality. In figure 3.4 you can see the two maths backends compared to one another. The main difference between the two is that Wigners Distribution is omitted in the actual system. This is because this distribution is slow in my usecase and the improvements in accuracy are not worth the performance loss. The other difference between the two is the introduction of the Euclidean distance. This was introduced as a secondary validation measure in order to allow the greater validation. It's this and Correlation co efficient that feed into the final similarity score.

3.1 Data gathering and Forming

My program relies on capturing the user's keystrokes, processing them and comparing them using a similarity measure. In order to do this, I decided to use the Keyboard Library[10] as it is a lightweight, secure and modern library that makes capturing keystrokes easy. In my project, I use of the hook function of the library, which is used to "hook" onto a user's keyboard and record all the user's actions and create keyboard events for each action. The record function which makes use of this hook function is shown in figure 3.5. The code snippet is very simple; first the program will 'hook' onto the keyboard using the Keyboard Library mentioned above, record all keystrokes until the interval has passed and then stop recording. The start time of the interval and an array of Keyboard Events are then returned to the main body of the program. The start time of the interval is recorded and returned as it used further on in order to be able to place keyboard events on a time line in the context of the interval.

A keyboard event is generated every time the user does something on the keyboard, whether pressing or releasing a key. Further information such as the type of the action (whether it was a press or a release), which key is this action happening on and a highly accurate time stamp of when the event occurred is also recorded. Figure 3.6, shows an example of a keyboard event produced by the function when the user presses down the 'h' key.

The first element in the array is the action, this can be either 'up' or 'down'. The next field is the scan-code, which I don't use but is useful for identifying keys. The next field is the name of the key which in this case is 'h'. After this is the time since the epoch in seconds, which is useful as this precise time-stamp is used to do the rest of the calculations. The other three fields are

7
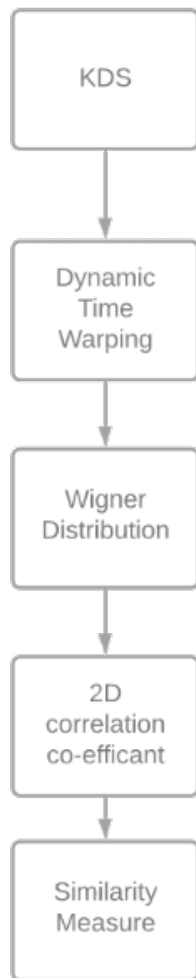
(a) Interim report maths backend
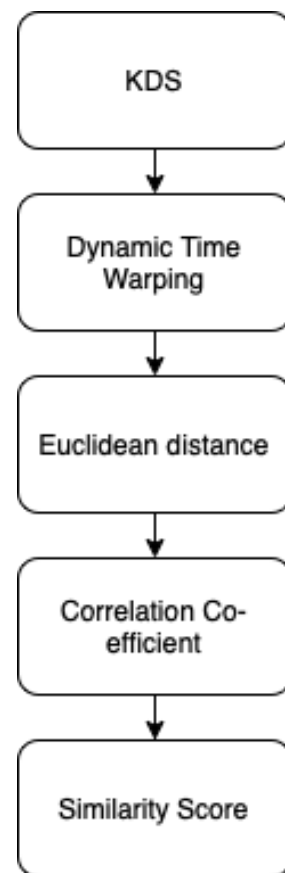
(b) Actual System maths backend

Figure 3.4: New and Old Validation Plans

```python
def record(interval):
    recorded = []
    startTime = time.time()
    keyBoardHook = keyboard.hook(recorded.append)
    time.sleep(interval)
    keyboard.unhook(keyBoardHook)
    return recorded, startTime
```

Figure 3.5: Record Function

```
['down', 35,  'h', 1649693924, None, None, 'N']
```

Figure 3.6: The keyboard representation of a user pressing "h"

device, modifiers and whether the user uses a keypad. None of these I use in my program and are discarded almost immediately.

A small amount of pre-processing is then done on this data before it is paired up. The first step is to remove the scan code, keypad, modifier and device from each keyboard event and in order to make them lighter and more usable. The next step is to take the start time that is returned by the record function and subtract this from the time stamp in each keyboard event to get the time that the action occurred in the interval. Figure 3.7 shows what the data in 3.6 looks like after going through this.
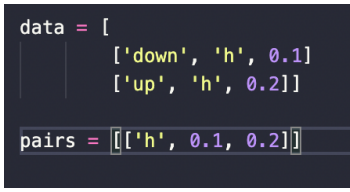
```
['down', 'h', 0.1]
```

Figure 3.7: Keyboard representation of a user pressing 'h' after pre-processing

The data collected at this point is stored as a 2D array with each sub array corresponding to an action. An example sub array is shown in 3.7.

In this form, the data cannot be used for anything, as it currently takes the form of several individual actions that have no relation to one another. Therefore, the next step is to form pairs from the data. A pair is formed of one 'down' and one 'up' action where the key field matches and the 'down' is before the 'up'. This is done is that it allows the program to work with half as much data, which reduces the number of unnecessary data points and allows the program to be able to form words using these pairs.

There are two main rules to follow when pairing the data. In many cases, the user will press and release a key in quick succession without pressing any other keys. Due to the chronological nature of the data, pairing these types of presses is easy. All that is needs to be done is to iterate through the pairs and when we come to a 'down' action then simply select the next value in the array if it is an 'up' action and the key matches. Figure 3.8 shows

```
data = [
        ['down', 'h', 0.1]
        ['up', 'h', 0.2]]

pairs = [['h', 0.1, 0.2]]
```

Figure 3.8: Example of simple pairing

9

an example of this type of pairing. However, this type of nice easy matching is not always the case.

In some cases, a user may press multiple keys down at once. For example, this might occur when the user is capitalising words using 'shift' or when the user is typing fast, so they may be already pressing down the next key before releasing the previous. An example of what the data will look like when this is the case is shown in Figure 3.9. Applying the previous method in which we pair up keys with matching key types and opposite actions which are next to each other would result in the output shown in the pairs array. As you can see, this is not correct and would only lead to one pair where there should be two.

In order to fix this, it is necessary to include another case in the code. Suppose the current action does not have a matching key and opposite action next in the array. In that case, the program will iterate through the rest of the array starting from the current point, searching for the next entry with a matching key and an opposite action after the current. If it finds one it will pair them up. The key thing we assume for this to work is that every action has an opposite action. In nearly all cases, we assume this to be true as it is improbable that a user will hold down a key for the entire interval. If the method cannot find a resultant action event, then it is still added to the pairs list with an end time of the length of the interval.

```
data = [
        ['down', 'h', 0.1]
        ['down', 'shift', 0.11]
        ['up', 'shift', 0.2]
        ['up', 'h', 0.23]]

pairs = [['shift', 0.11, 0.2]]
```

Figure 3.9: Example of simple pairing

The resulting pairing algorithm has a time complexity of $O(n^2)$. In a program which is all about speed and minimal impact to the user, it is essential that the program has the lowest time complexity as possible. Due to the complicated nature of how users type, I believe this is the best time complexity for a problem of this nature.

3.1.1 Forming Words

After forming the pairs, the next step is to form words from these pairs. The words that the program forms are essential as it is this that the program uses to compare users. In English words take many forms, as such it is needed to account for numerous different possibilities in the word forming function.

A word is defined in my program as a sequence of pairs bounded by punctuation, white space or the use of modifiers such as 'shift'. In latter stages of this report, I refer to these as break pairs. The one notable exception to this rule is when an apostrophe or a hyphen is detected. If this occurs, then the program will check the previous pair and the pair afterwords and if both are letters and not numeric or punctuation, the pair including the punctuation is added to the current word.

The data at this stage takes the form of a 2D array. The program will iterate through the 2D array it is given and check the key that the pair matches. If it is a letter or a number then it is added to another array which is used to store the current word being formed. If a break pair is found, it is not added to the current word, the current word is used to form a word object which is then

saved to an output array. If the break pair is a white space pair, then the pair is simply skipped. However, if the break pair is a modifier such as 'shift' or 'ctrl' then the relevant entry in the semantics dictionary is updated for that user. This dictionary is used in the validation section of the project and is another indicator on how a user types. Backspace handling is done separately. If the user has pressed backspace, then the last letter added to the word is removed from it. The program can handle multiple backspaces even if they delete the entirety of the current word. If this occurs, the previous word object is popped off the array to be the current word and the last letter of this new current word is removed.

When the program gets to the last pair in the input array, if the pair is not a break pair then the pair is added to the current word and the current word forms a word object which is then saved to the output array.

The program will then return the output array which at this time is formed of word objects and the semantics dictionary. The output array is then saved to the wordsOut attribute of the Calculation class while the semantics is saved to the semantics attribute in the class.

3.1.2 Data Selection

If the program was to go through and check every single word for similarities, the cost in terms of time would be excessive and would make the program unfit for use especially if the user typed quickly during the interval. For example, if the program checked every word and the user typed 56 words in a 60-second interval, the time taken would be over 2 minutes as shown in figure 3.10 which while highly accurate and secure would render the program unusable as the time taken to process and perform all the similarity calculations would be in excess of the interval and as such would lead to a lower degree of accuracy and security. Furthermore, this would severely impact the performance of the user's computer and as such go against one of the main aims of the project.

As such, it is necessary to use a sampling method to choose words from the list of words chosen by the word forming function. While this is less accurate than checking every word, the performance gain over checking every word is huge with a large time saving per interval. Choosing how many words were selected was the next problem I endeavoured to fix.

I conducted a number of tests measuring how long the entire validation procedure took. Initially I started with 2 words chosen per interval with one chosen every at the start of the interval and one at the end. I then increased the number of words chosen by two each time with the interval remaining the same. At each testing point, the interval remained the same with a word selected evenly throughout the interval. Figure 3.10 shows the results of such a test. The time taken to perform the calculation increases linearly as the number of words chosen increases. If the word chosen is particularly long, then the time taken increases. The test data was the same for all tests with the user typing a paragraph containing 57 words of differing lengths.

After performing the test, I settled on the program choosing four words per interval. Choosing four words took under 10 seconds and allowed the program to get coverage every quarter of the interval which is acceptable.

The process to choose the words is simple, it essentially will take one word from each quarter of the interval. This allows the program to get a good element
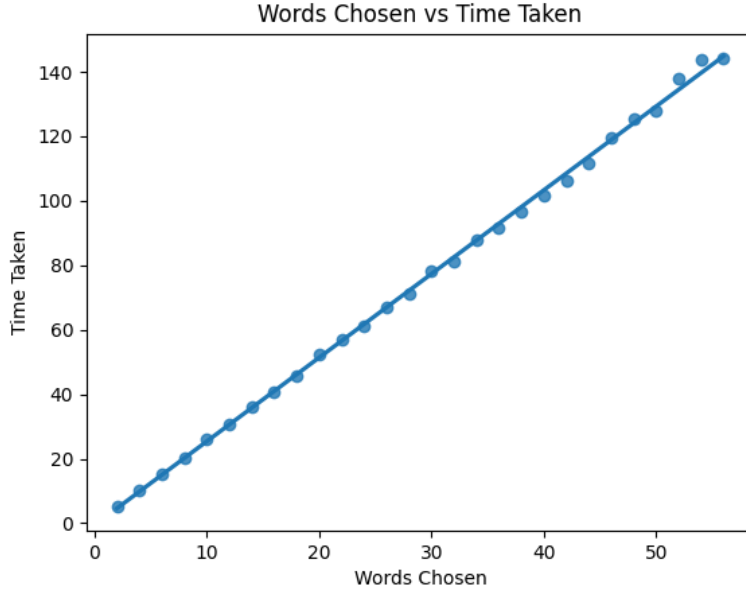
Figure 3.10: Comparing words chosen to time taken in seconds

of coverage.

3.2 KD Signal

Once the raw keystroke data has been formed into words and the words chosen, the next step is to transform the data from a word object made up of keystroke pairs into numerical data that can be used by later algorithms. The best way to do this is to transform the data into a measure of how many keys are being held down at a particular point in time. The resulting output is known as a key down signal (KDS). [1]

To convert a word into a key down signal, the start and end times of the word being transformed are used. Assume that w is the array of times that key actions occur in a particular word. w_1 being the time of the first action and w_n being the time of the last action. This part will loop through all timestamps until it ends with the final time which is denoted by w_n. The accuracy of this step is paramount as it is the level of detail that is the base accuracy for the rest of the steps. A higher accuracy means that the program will check more data points within this range at the cost of reduced performance. The current level of this is set to 4 decimal points which seems to provide a good balance between accuracy and performance.

$$KDS(w) = \sum_{i=w_1}^{w_n} K(w_i) \tag{3.1}$$

K is the next step of the algorithm and is heavily based on the KDS algorithm shown in [1]. n is the array of key presses that is used in the previous step. This

step of the algorithm iterates through all the key presses and uses a modified Heaviside step function denoted by h which is run twice per pair with the time input from the previous denoted as t and the 'down' action denoted as n_i^1 and the 'up' denoted as n_i^2. The value returned by the Heaviside step function with the 'up' action is subtracted from the value returned by the 'down' function.

$$K(t) = \sum_{x=1}^{n_k} h(t, n_i^1) - h(t, n_i^2) \qquad (3.2)$$

The reason for this subtraction is that the purpose of this measure is to return the number of keys pressed down at the time input. Once a key has been released it is essential that the key is removed from the measure. For example, if a pair exists with down action time being at 1 second and up action time being 1.1 seconds. At time, 1.5 the equation will equal $1 - 1 = 0$. However, if the time put in is 1.05 then the equation will be $1 - 0 = 1$ which indicates that one key was being held down at this particular time.

For every time input, each pair is checked with the sum of all the results stored in a dictionary along with the time input as the key. It's this dictionary that forms the KD signal and is used in further steps.

3.2.1 Heaviside Step Function

This is the bottom layer of the KD signal algorithm. It is a modified version of the Heaviside Step function.

$$h(x_1, x_2) = \begin{cases} 1 & \text{if } x_1 > x_2 \\ 0.5 & \text{if } x_1 == x_2 \\ 0 & \text{if } x_1 < x_2 \end{cases} \qquad (3.3)$$

The modification done is very simple, the only change is the addition of a third case which tests if the two times are equal to one another. Due to the nature of the use case for my project, there is a relatively high chance that the two times are equal to one another. In this case this means that the user at this time is currently in the process of performing that action whether that be pressing or releasing the key. The theoretical basis for this function is taken from "Time–frequency analysis of keystroke dynamics for user authentication" by Ramin Toosi and Mohammad Ali Akhaee[1]

3.2.2 Output

The resulting signal can be shown easily in graph format. Figure 3.11a and figure 3.11b show the same user typing the same word twice in two different intervals. Figure 3.11c is both sets of data overlaid on the same graph. As you can see, even with the same user typing it, both sets look very different and normalisation becomes essential in order to be able to compare the data accurately.

3.3 Dynamic Time Warping

The data in my project requires normalising to be useful. The algorithm that I chose to use is called the Dynamic Time Warping algorithm. Two sets of data,
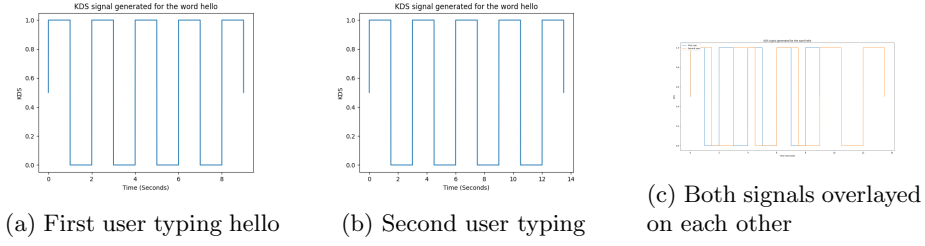
(a) First user typing hello

(b) Second user typing

(c) Both signals overlayed on each other

Figure 3.11: KD Signals generated by a user typing hello

(a) First after DTW

(b) Second after DTW
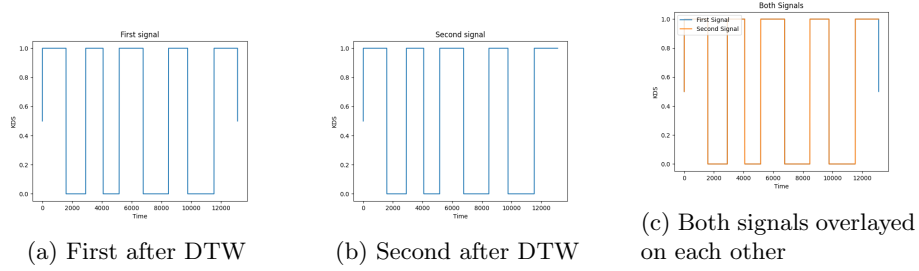

(c) Both signals overlayed on each other

Figure 3.12: KD signals after going through DTW

one loaded in from the word file and the other generated from that interval is input into this function in order to normalise it. As both sets of data are input, the Dynamic Time Warping algorithm attempts to change both sets of data in order to make them as close as possible. The output is two sets of normalised data which can then be compared against one another to provide a similarity score.

In my program, I chose to use the fastdtw library [11]. Like the name suggests, this library provides a really fast implementation of the dynamic time warping algorithm with the algorithm taking only $O(n)$ time complexity[11].

The output of the two datasets shown in figure 3.11a and figure 3.11b both put through dynamic time warping is shown in figure 3.12.

Shown in figures 3.12a and 3.12b is the results of the data in figures 3.11a and 3.11b after going through dynamic time warping. Despite going through the process, you can see that they haven't changed much and look the same. This is because both signals were typed by the same user and as such are very similar. However, the first signal has an extra datapoint on the end which causes it to go on longer than the other signal. This is not a huge issue however, as the validation systems explained below returned the values of 1.0 and 0.99999 which showcases that both signals are the same.

3.3.1 Path

The implementation i chose to use returns two values, the distance and the path. The distance is used in the validation step to calculate euclidean distance. The path is more important however. It is what is used to "warp" the data. Theoretically, it is the least costly path through a cost matrix. It essentially is a pairs of indexes in the two datasets which map up. It states which values

need to go into the datasets.

3.4 Validation Measures

Finally the data is in a state where we can perform validation. In order to improve the accuracy of the program, I use two different validation methods along with the semantics collected in the word forming stage.

The two methods I use along with the semantics are euclidean distance which is done in the dynamic time warping stage. This has secondary importance compared to the 2D correlation co-efficient. The reason for this, is that the euclidean distance is simply a measure of the distance between two points and is not a good indicator for data such as this. Therefore, I use this as a rough figure and use the correlation co-efficient as the main method.

The values produced by these validation methods are then put together to form a value between 0 and 100. This is then compared against the confidence level. The confidence level is a integer which determines how similar the data loaded in from the file and the data collected in the interval have to be. For example, if the validation methods in the interval (Similarity Score) return a value of 0.75 and the confidence level is 0.8, then the user is rejected and asked to re-authenticate. When developing this project, I settled on a confidence level of 0.8. Only the euclidean distance and the 2D correlation go into calculate the similarity score. The semantics are used to either raise the confidence level or lower it depending on the value returned by the semantics validation.

The semantics validation is very simple. In the words step of the validation procedure, the semantics of the data input is collected and stored in the class. Each user will also have a semantics file stored in their data directory. An example of a users semantics file is shown in figure 3.13. The data stored in the file and the data collected for that interval are then compared against one another.

```
{'shift' : 1,  'caps lock' : 0}
```

Figure 3.13: Example of a users semantics file

If for example, the user in the past has used the shift key and in this interval has also done so, then the confidence level is lowered by 0.02. However, if the user in the past has always used the caps lock key and never shift then they don't match and the confidence level is raised by 0.02. The reason of such a small change in the confidence level is that users habits regarding this are not typically set in stone and as such any large change will likely lead to reduced accuracy.

The validation methods are explained further in sections 3.4.1 and 3.4.2 respectively. Here I will outline the entire validation process. The flow chart in figure 3.15 provides a visual representation of the first part of the validation procedure.

Figure 3.15 explains the validation procedure.

As shown in figure 3.15, the words chosen in the first step are iterated through. If a word already has a word file stored in the users directory then that is loaded in and de-

```
distances = {1 : [0.0, 1.0],
             2 : [340, 0.84],
             3 : [1000, 0.74],
             4 : [None, None]}
```
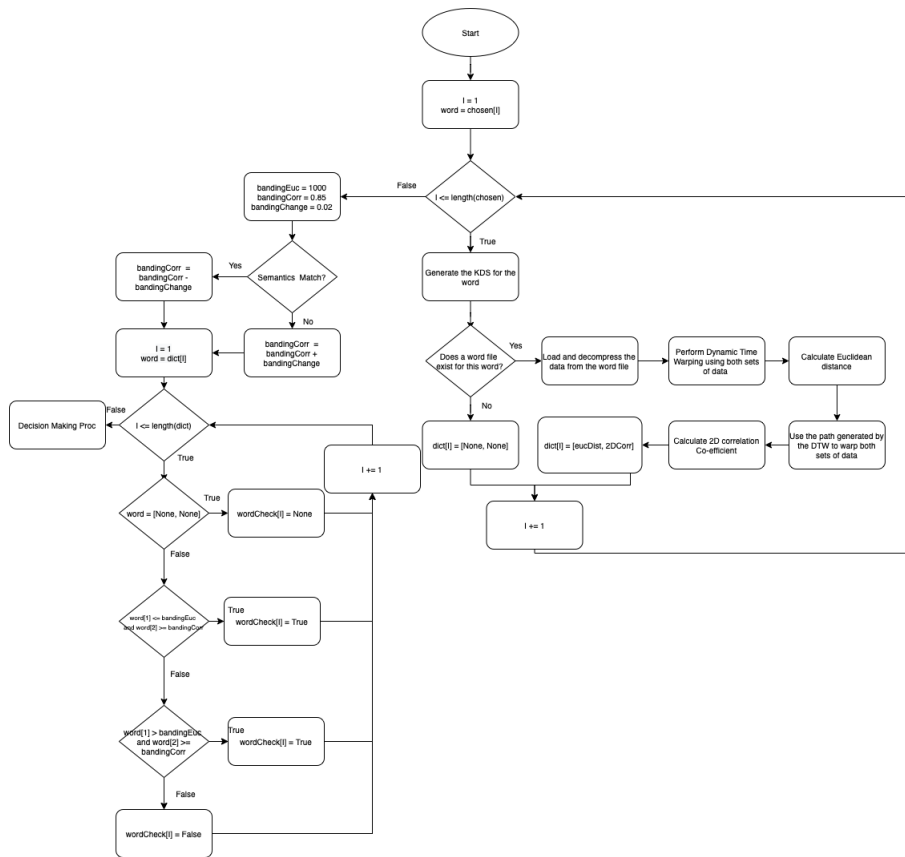
Figure 3.14: Example of a distances dictionary after validation calculations done

Figure 3.15: Validation Procedure

16

compressed. Otherwise, the procedure just returns two None values. The two data sets, one loaded in and the other generated from the current interval are then both put through Dynamic Time Warping where the euclidean distance is calculated and stored. The path generated by the DTW is then used to 'warp' both datasets. It is at this stage that the 2D correlation co-efficient is used in order to calculate one element of the similarity score. The Euclidean distance and the correlation co-efficient are then stored in a dictionary with the key being the index in the chosen array. Figure 3.14 showcases an example distances dictionary. In the example, the first two elements were typed by a genuine user with the third being typed by an imposter and the last word has not been seen before so the validation procedures just returned None.

After the distances dictionary has been formed with all values inside. The next step is to categorise the values. First of all we have to define the confidence level for each validation procedure. As shown in the flowchart (3.15), these are adjusted based on the results of the semantics validation. The distances dictionary is iterated through and each value checked. Figure 3.4 below shows what happens in each individual case where e is the Euclidean Distance, c is the value returned by the correlation co-efficient, i is the confidence level for the correlation co-efficient and b is the confidence level for the Euclidean Distance.

$$
\begin{cases}
None & e == None \ or \ c == None \\
True & c >= i \ and \ e <= b \\
True & c >= i \ and \ e >= b \\
False & Otherwise
\end{cases} \tag{3.4}
$$

The value returned by this function is then adding into an array which is used in the next step. Figure 3.16 shows what resulting array formed by putting the data in 3.14 through this part of the validation procedure. This example presumes that the confidence levels for Euclidean Distance and correlation co-efficient are 1000 and 0.84 respectively.

```
wordCheck = [True, True, False, None]
```

Figure 3.16: The result of the data in figure 3.14 after going through further validation

Transforming the data this way allows the order of the data to be preserved. Furthermore, it allows the next stage of the validation to simplified heavily as rather than working with variable data, we know to need to only consider whether the value is one of True, False or None which vastly simplifies the complicated system used to make a decision which is explained in the next step.

The "wordCheck" array shown in 3.16 is then used in the next step in order to finally perform an action based on the data. Despite what happens next only three outcome occur. Figure 3.17 showcases what happens if only one word is in chosen and an output returned to it. If the word is true, then the person

17

checks out and the program moves on to the next interval. However, if the word doesn't pass the validation checks or has never been seen before, then further processing has to be done.

This is very complex. If the user fails validation then their pc is locked and they are forced to re-authenticate. In order to perform this action, I load in the default python library sub process [12] which I then use to call the windows lock dll file. This enables me to lock the user out of the computer easily and efficiently without much wasted code or overhead in terms of performance. If the user has changed when the user does re-authenticate, then 'New' is returned. This tells my main body of the program that it is a new user and as such a new keyboard is created for that user. If the opposite happens and the user is re-authenticated successfully, then the word files and semantics are updated and the program returns true.
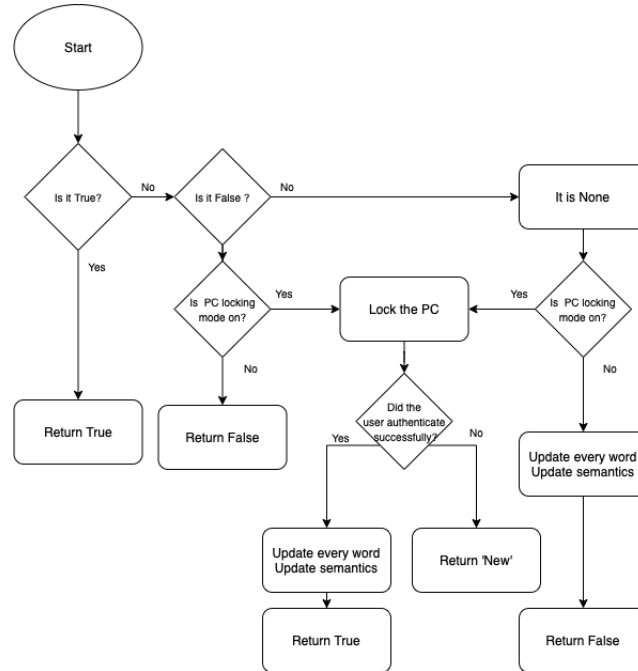
Figure 3.17: Flowchart which illustrates decision tree if only one word is chosen

The validation function can take in as a parameter a 'mode'. This determines what happens should validation fail. The default mode is 'r' which stands for real. This allows the program to do the entire process including locking the users computer if validation fails. 'rnl' stands for real no lock which is mainly used for testing and demonstration purposes because it stops the users pc locking and just returns false. A 't' flag also exists, this indicates that the program is in test mode and when in test mode the program will simply save every word.

Typically the user will type multiple words in an interval. As the result outputted above from the wordCheck can take any one of three values for each word, there are plenty of variations. These are all explained below.

Possibility 1: All words pass Validation and wordCheck consists of just True values True and an empty array are returned. The runner which runs the

18

validation process then continues on to the next interval.

Possibility 2: Some words pass validation and some words have never been seen before. This results in the program creating word files for those which have not been seen for using the update function which is detailed in 3.6. True and an empty array is then returned and the program as a whole moves onto the next interval.

Possibility 3: All words have never been seen before or fail validation.

When a combination of never seen before and failing validation is detected, then the program has no confidence that the user is genuine, therefore it starts the re-authentication procedure which is explained above.

Possibility 4: The chosen only consists of words which the program has never been seen before.

Much like possibility 3, the program has no way of knowing whether the user is genuine or not. Therefore it takes the safest course of action and forces the user to re-authenticate. It then does the exact same as possibility 3 does.

Possibility 5: All words fail validation If all words fail validation, then the program is confident that an imposter is attempting to use the computer. The function locks the pc and demands re-authentication from the user. Like in possibility 3 and 4, it will update the words that fail validation if the user passes re-authentication.

3.4.1 Euclidean Distance

The first validation method used is the Euclidean Distance. It is a system of measure between two data points. In my system it is used on the warped data in order to generate the distance between them. The overall score is calculated by adding up all the distances between the data points output by the KD signal function and the data points input from the word file. This is results in a very good benchmark figure to determine how close the two signals are together.

$$euc(a, b) = \sqrt{\Sigma_{i=1}^{n}(a_i - b_i)^2} \tag{3.5}$$

Figure 3.5 shows the mathematical formula for euclidean distance where a and b are the two KD signals, and n is number of data points. This then returns a value like we can see in the first element of the array in 3.18. The value returned is self-explanatory with a smaller value indicating less difference between signals than a large one. In my program rather than calculate this value myself, I used scipys implementation of this function[13]. TAttempting to do this myself in pure python resulted in slow processing times. The function used by scipy is much faster due to it being written in C++ which is far better at doing these functions with large datasets produced by the KD signal function.

3.4.2 Correlation Coefficient

2D correlation co-efficient is used because it is relatively lightweight, quick, easy to implement and provides easy to interpret results. As opposed to the value produced by the euclidean distance, the output returned by the 2D correlation co-efficient can only take a

```
Doing test:  3
DTW Time:  5.545889800000001
{0: [180.0, 0.9564968019233274]}
Doing test:  4
DTW Time:  5.269914400000001
{0: [1400.0, 0.6066430892542846]}
Doing test:  5
DTW Time:  6.120747300000005
{0: [0.0, 1.0]}
Doing test:  6
DTW Time:  6.435730300000003
{0: [0.0, 1.0]}
Doing test:  7
DTW Time:  6.655205900000006
{0: [520.5, 0.9087348358125797]}
Doing test:  8
DTW Time:  6.9057584999999975
{0: [560.5, 0.9086232941766443]}
Doing test:  9
```

19

range of values between -1 and 1. If the function returns 1, then the two signals input are exactly the same, 0 if they are radically different and -1 if they are the same again. An example of the output returned can be seen in the second element of the dictionary in figure 3.14. In shortened form, the 2D correlation figure essentially returns a percentage of how close the two input signals are.

This measure is used after the dynamic time warping and take the two input signals as inputs. This is weighted higher in my system due to the stronger accuracy and ease of use. One example of why the this is more accurate than euclidean distance is shown in the figure 3.18. 2D correlation co-efficient returns around 0.95 for both, whist euclidean distance returns 180 and 300 which are wildly differing figures.

$$f(a, b) = \frac{\Sigma_{i=1}^{n}(a_i - \overline{a})(b_i - \overline{b})}{\sqrt{\Sigma_{i=1}^{n}(a_i - \overline{a})^2}\sqrt{\Sigma_{i=1}^{n}(b_i - \overline{b})^2}} \tag{3.6}$$

Figure 3.6 shows the equation that my implementation is based upon where a and b are the two warped arrays passed in, n is the length of either array, a_i and b_i are the values at positions i in each array respectively. My code implementation is shown in figure 3.19.

```
# Correlation Coefficient
cov = 0
XSum = 0
YSum = 0
Xmean = sum(ff_warped)/len(ff_warped)
Ymean = sum(ii_warped)/len(ii_warped)

 for i in range(len(ff_warped)):
 cov += (ff_warped[i] - Xmean)*(ii_warped[i] - Ymean)
    XSum += math.pow(ff_warped[i]-Xmean, 2)
    YSum += math.pow(ii_warped[i]-Ymean, 2)

correlationCoefficient =
    cov/((math.sqrt(XSum)*(math.sqrt(YSum))))
```

Figure 3.19: Code Implementation of correlation co-efficient

The code is self-explanatory, with the Xmean and Ymean variables simply calculating the mean for the two input arrays respectively. The output of this function of this is then stored along with the euclidean distance in a dictionary to be used for further processing.

3.5 Training

When a new user is created, it is essential that training happens. Otherwise, the program has nothing to go on and as such locks the computer at every interval. This is incredibly annoying for the user but also the learning process would be slow. Therefore it is necessary to add a training phase. There are two potential approaches to a training phase. The first is to have a dedicated text that the user types out and the second is that for the first x intervals all word data is saved and then used as a bank. I explored both of these attempts in my project.

The first one I explored was the dedicated training phase with a dedicated sample text that the user types out. The difficulty with this wasn't making the UI or ensuring that the data was captured, it was coming up with the text itself. In order for this to work properly, a text was needed that contained all of the most common words that a user uses along with a wide range of punctuation. This in itself is difficult as each user is different. My program currently use the text shown below as the dedicated text.

"Since they are still preserved in the rocks for us to see, they must have been formed quite recently, that is geologically speaking? What can explain these striations and their common orientation? Did you ever hear about the Great Ice Age or the Pleistocene Epoch? Less than one million years ago in fact some 12000 years ago an ice sheet many thousands of feet thick rode over Burke Mountain in a southeastward direction. The many boulders frozen to the underside of the ice sheet tended to scratch the rocks over which they rode. The scratches or striations seen in the park rocks were caused by these attached boulders. The ice sheet also plucked and rounded Burke Mountain into the shape it possesses today."

This piece of text was chosen as it was long enough to get a wide variety of punctuation in but so long that the user gets bored. A wide variety of words is also used in order to attempt to get the best coverage available.

The next approach I tested was having the first 5 intervals for a new user simply recording and saving all words that are used. This gives better coverage than the other approach because it allows the program to learn the users habits and common words. This approach is also deals well with different users and allows greater insight into the user. While this method is good at learning users, it does potentially open up a security issue. If all words, are simply being saved and stored, nothing stops an imposter user from altering this training data. This approach has another disadvantage, it doesn't learn how the user uses punctuation which the other approach does.

Whilst exploring both of these, I came to the conclusion that a mixture of both training methods would be the best approach. This would ensure that the program gets the best of both worlds. Therefore, I decided to use both systems with the first 5 intervals after the user finishes training being the training intervals. The reason 5 was chosen as the training intervals is because it minimises the security risk caused by an imposter user whilst still proving to be an effective data gathering tool.

The code in 3.20 highlights the main "runner" code. This takes the data in from the record function, if training iterations is turned on and the number of training iterations has been exceeded, then the validation function is run with that data and a decision returned. If that happens the pc goes through the authentication specified by the validation procedure. However, if the opposite

```python
def runner(id, prof, stop):
    count = 0
    while True:
        data, start = record(interval)
        if stop():
            break
        if trainingItersYN == False or count > trainingIters:
            inter = i.Calculation(data, start, prof, 1)
            if stop():
                break
            decision, index = inter.validation(mode='r')
            if decision == False:
                break
        else:
            if count <= trainingIters and len(data) != 0:
                inter = t.Training(data, start, prof, 1,0)
                count += 1
                if stop():
                    break
```

Figure 3.20: The main runner function with training intervals in

is true and it is a training interval then the data is simply saved as specified by
the training class which simply saves the KD signal generated for that word.

3.6 Update

The way users type change over time. This is caused by a number of factors such
as age or change in desk setup. For example, the way a user types at home will
be different to the way users type at a coffee shop. Therefore, it is important
that my program can learn and adapt. This is where the update function comes
in. This step is done after the validation step and is called in possibility where
a user has failed validation but re-authenticated successfully. Essentially this is
called where the program has got it wrong.

   The function is very simple. It will iterate through all words that have
failed validation and simply update the word files with the data collected from
the current interval.

3.7 Storage

In order for this project to work correctly, it is essential that the program has
a way of storing how users type certain words. It is necessary to build up a
storage of the words that the user has typed in the past. In my project this
is done by storing the KDS generated in the KD step and storing it in a json
file. The reason for storing it this instead of storing it inside a database is
that the data is highly variable and as such it is not possible to store the data
inside a database. For example, one signal might have 90,000 datapoitns while
another might only have 5000. As such, it's not practical to store this kind

of highly variable data inside a database. Each file is named after the word contained and represented by the KD signal inside. For example, if the word was "hello" the file would be named hello.json. Storing it this way ensured easy access to the data within and simple identification of which data belonged to which individual. Storing just the KD signal as opposed to storing the pairs or the raw keystroke data collected in the first place has many advantages. The first major advantage in storing just the KDS is that a malicious user cannot easily read how a user types just by looking at the files. Secondly, this type of data is easy to read by the program. Thirdly it stops the program having to perform the KDS element of the validation procedure every time is loads in a file resulting in less performance impact. Simply loading in and comparing is far easier than processing all of the data again and then comparing it. Figure 3.21 shows a small excerpt of one of these word files.

The main problem with storing the data this way is that file sizes can become a problem. Generating a KD signal generates a lot of data points and when these are all stored, the size of each individual word file starts to become an issue. Users also tend to use a lot of words and once the user has been using the program long enough, the size of the users directory can become huge. One way to fix this would be to reduce the amount of data points being generated for each word. Unfortunately this isn't feasible as accuracy is negatively impacted. Therefore the solution it to compress the data points when they are saved to the file and then decompress them when they are loaded back in.

Figure 3.21 shows a sample of the representation of the data stored without compression whilst figure 3.22 shows the same data after compression. Due to the way the KD signal is generated, there are a large amount of duplicate values. My compression algorithm works by storing the range in values at which the KD signal is the same. In figure 3.22 you can quiet clearly see that between the timestamps of 1.5649 and 1.6368 the value did not change from 1. Previously all of the timestamps between these two values would have been stored leading to a bloated file size that contained a lot of unnecessary information.

This type of compression makes a huge impact on the file sizes. For example, the file storing the data in figure 3.21 without compression is approximately 91 kilobytes whilst the compressed version shown in 3.22 is approximately 735 bytes. This is a large saving in terms of space, with the compressed version being 99% smaller than the compressed version. Whilst 91 kilobytes may not seem like a lot, when the average office worker types at approximately 30 words per minute[?] for approximately 8 hours a day leading to around 14400 words typed per day. If even a small percentage such as 10% of these words are

{
  "0.0": 0.5,
  "0.0001": 1,
  "0.0002": 1,
  "0.0003": 1,
  "0.0004": 1,
  "0.0005": 1,
  "0.0006": 1,
  "0.0007": 1,
  "0.0008": 1,
  "0.0009": 1,
  "0.001": 1,
  "0.0011": 1,
  "0.0012": 1,
  "0.0013": 1,
  "0.0014": 1,
  "0.0015": 1,
  "0.0016": 1,
  "0.0017": 1,
  "0.0018": 1,
  "0.0019": 1,
  "0.002": 1,
  "0.0021": 1,
  "0.0022": 1,
  "0.0023": 1,
  "0.0024": 1,
  "0.0025": 1,
  "0.0026": 1,
  "0.0027": 1,
  "0.0028": 1,
  "0.0029": 1,
  "0.003": 1,
  "0.0031": 1,
  "0.0032": 1,
  "0.0033": 1,
  "0.0034": 1,
  "0.0035": 1,
  "0.0036": 1,
  "0.0037": 1,
  "0.0038": 1,
  "0.0039": 1,
  "0.004": 1,
  "0.0041": 1,
  "0.0042": 1,
  "0.0043": 1,

Figure 3.21: A sample of an uncompressed word storage file

23

unique, then the resulting uncompressed word
data would take up 131040 kilobytes or 131.4
megabytes worth of space. However, if the
data is compressed then the same amount of
words would only take up 1.05 megabytes. In
this particular case this results in savings of 130 megabytes which is a huge
amount.

```
[
    {"key": [1.5647, 1.5647], "value": 0.0},
    {"key": [1.5648, 1.5648], "value": 0.5},
    {"key": [1.5649, 1.6368], "value": 1.0},
    {"key": [1.6369, 1.6369], "value": 0.5},
    {"key": [1.637, 1.664], "value": 0.0},
    {"key": [1.6641, 1.6641], "value": 0.5},
    {"key": [1.6642, 1.772], "value": 1.0},
    {"key": [1.7721, 1.7721], "value": 0.5},
    {"key": [1.7722, 1.7955], "value": 0.0},
    {"key": [1.7956, 1.7956], "value": 0.5},
    {"key": [1.7957, 1.865], "value": 1.0},
    {"key": [1.8651, 1.8651], "value": 0.5},
    {"key": [1.8652, 1.9731], "value": 0.0},
    {"key": [1.9732, 1.9732], "value": 0.5},
    {"key": [1.9733, 2.0206], "value": 1.0},
    {"key": [2.0207, 2.0207], "value": 0.5},
    {"key": [2.0208, 2.1287], "value": 0.0},
    {"key": [2.1288, 2.1288], "value": 0.5},
    {"key": [2.1289, 2.2002], "value": 1.0}
]
```

Figure 3.22: KD representation of a user typing hello after compression

Performance is important to the success of the program but reducing this
slightly in order to save storage space on the users computer is acceptable. The
next step is where to store these word files.

Each user has a "User Profile". This essentially stores information about the
current user such as the profile name, their current keyboard, whether they are
a new user or not and how many keyboards they have. When a user profile is
created as a result of either a new profile being created in the training procedure
or "New" being returned from the validation procedure, a new folder is created
in the Data folder inside the programs home directory which is named after
the users windows user name. The python library has a built in function called
"getpass" which simply interfaces with windows in order to get the current users
information and provide a secure interface for password input[14]. In order to get
the current users information, I use the getuser function from this library. Each
keyboard that the user has is stored inside the users folder. The users word
information is stored inside the relevant keyboard along with their semantics
storage file. A typical folder structure is shown in figure 3.23.

The reason it is done this way is that users type differently on different
keyboards and it made sense to have a way to store all the different profiles
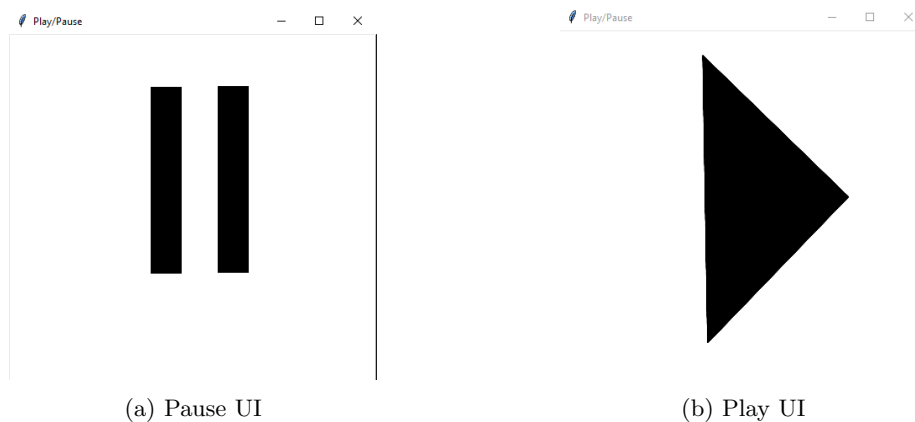
(a) Pause UI

(b) Play UI

Figure 3.24: Example of play/pause UI in both states

that the user may have. The semantics file is also stored inside the keyboard folder. This is for the reason that users may use caps lock on one keyboard and shift on another. It ensures that the program can really narrow down for certain users and really learn the user. The user profile class stores the list of keyboards along with the path to the current keyboard.
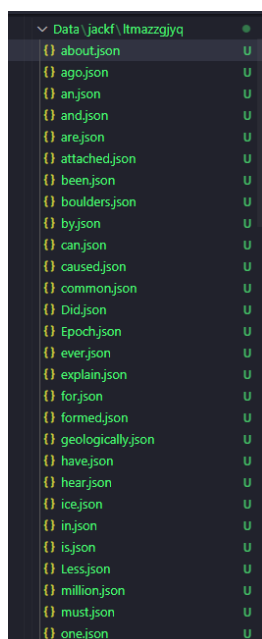
3.8 Pausing

Figure 3.23: Example of folder structure after light usage by a user with one keyboard

One main problem with my system was the lack of user privacy. For example, a user wouldn't want the program to pick up and store details about their bank password. Therefore, it becomes necessary to introduce a pause functionality. This is very simple and consists of a simple user interface screen which consists of one button which is shaped like a pause button. When the button is pressed the pause symbol changes to a play and the system stops recording the users keystrokes until the button is pressed again. The UI that powers this functionality runs on the main thread whilst the calculation happens on another thread. When the button is pressed to pause the calculation, the stop thread flag is switched to True which stops the thread and causes it to return. As such, when the calculation is switched on again a new thread is created and the calculation continues.

However, this system is open to abuse because an imposter could simply switch the system off, use the system and then switch is back on. In order to combat this, when the user decides to pause the program, they are prompted to re-authenticate using the windows login system. This essentially combats this problem and ensures that data collection remains. Figure 3.24

25

```python
def stop():
 global stop_threads
 if button.cget('image') == 'pyimage2':
     cmd='rundll32.exe user32.dll, LockWorkStation'
     subprocess.call(cmd)
     button.configure(image=imgPlay)
     button.image = imgPlay
     stop_threads = True
     for worker in workers:
         worker.join()
 else:
     button.configure(image=imgPause)
     button.image = imgPause
     stop_threads = False
     tmp = threading.Thread(target=runner, args=(0, prof, lambda:
         stop_threads))
     tmp.start()
```

Figure 3.25: Code for controlling calculation threads

showcase both states of this simple section of the UI.

Figure 3.25 showcases the function that is run everytime the button is pressed. It essentially checks what image is being displayed. If the pause image is being displayed, then the program calls the windows UI in order to authenticate the user. If they pass this, then the image is changed, the stop threads flag is switched to true and all threads stop. If the play image is being displayed, then the image is switched and the calculation sub-process is started back up again.

Chapter 4

Results and Discussion

In order to be able to test the accuracy of my system it is necessary to first define some measures. False Positive (FP) is used to define the case when the system lets in an imposter user. False Negative (FN) is the opposite of this where the system rejects a user it shouldn't have. True Positive (TP) is where the system makes the correct decision and allows in a user correctly. True Negative (TN) is where the system makes the correct decision and rejects a user where it should have. Hold time (HT) is the time in which a key is held down or the time between the when a key is pressed and when a key is released. The float time (FT) is the time between actions. It can be defined as the time between the user releasing a key and pressing down the next key.

In order to test the system, it was necessary to create some data. In order to do this, I made a function which would generate test data with timings I specified. This enables me to generate lots of test data simply and easily. Furthermore, it allows reproducibility in the data collected. If the data was simply a user such as myself typing, it is almost impossible to get the same timings in terms of typing more than once. This function can be seen in Appendix C. That function can only deal with data that is linear (where a 'up' action is always followed by a 'down' action).

4.1 Test 1: Validation Measures

The first tests I conducted on the system consisted of checking if results returned by the validation procedures are what is expected. In this test, the confidence level is irrelevant as we are not measuring this. The test data consists a singular word "geographically". The first test in this set of tests consisted of me generating 10 sets of test data using the same word each time. Each set of test data increased the hold time by 0.1 whilst keeping the float time the same.. The program was in 'rnl' which ensured that it did not lock the pc or update the data. This was done in order to ensure that all tests were tested against the same dataset. The reference dataset which was loaded in each time consisted of the 5th set of test data. This test should return an inverted bell-shaped curve due to the reference data point being in the middle.

Figure 4.1 shows the results of the testing. As shown, the test data results in a inverted bell curve which is expected. The smallest euclidean distances
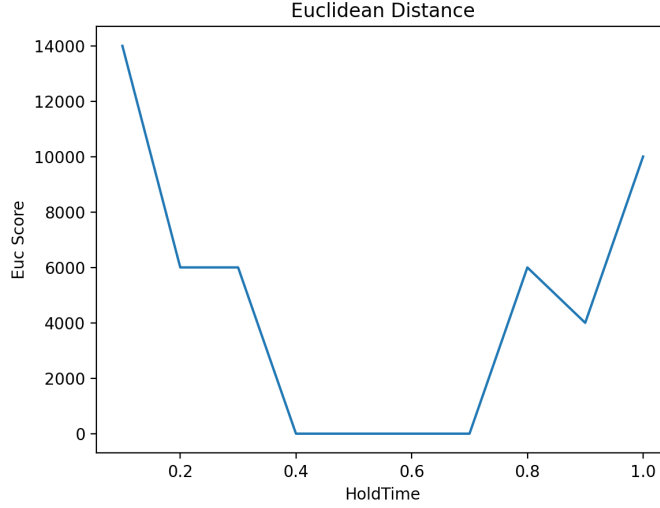
Figure 4.1: Euclidean Score

being in the middle as these are the most similar to the reference data that all datapoints are checked against. This proves that the euclidean distance is working as expected. Interestingly, the euclidean distance score is not a perfect inverted curve due much higher results on the left side of the graph rather than the right. I'm not too sure why this happens but I believe it may be because of the way in which Euclidean distance is calculated. It's because of results like this that the euclidean score is used as a estimation and a secondary measure to the 2D correlation co-efficient.

Figure 4.2 graphs the hold time against the 2D correlation score. Similar to the graph shown in figure 4.1 it takes the shape of a bell shaped curve. Unlike the euclidean graph this is a far more uniform graph which highlights once again that the 2D correlation co-efficient is a better measure than euclidean distance. The table below shows the raw data of such a test.

HT	Euc	Corr	Dec
0.1	13997	0.56	False
0.2	6001	0.72	False
0.3	6001	0.73	False
0.4	1.5	0.99	True
0.5	1.5	0.99	True
0.6	1.5	0.99	True
0.7	1.5	0.99	True
0.8	6000	0.74	False
0.9	4000	0.83	False
1.0	10000	0.58	False

The raw data points to another interesting anomaly in the data, 0.9 if following the trend should have a euclidean distance greater than in 0.8 and a 2D correlation co-efficient less than in 0.8. However, this is not the case. This small
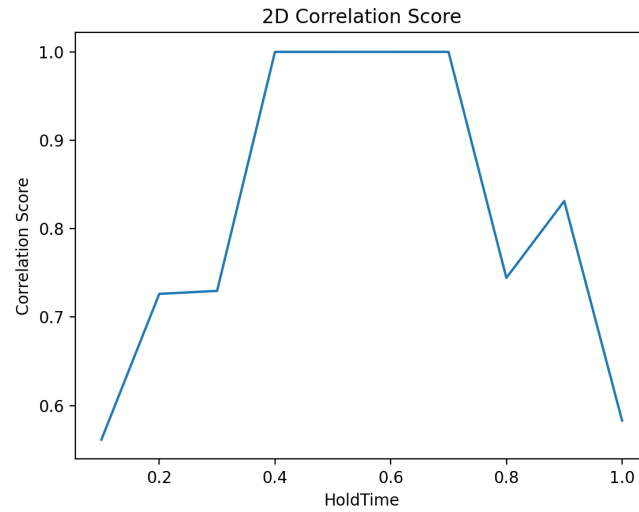
28

Figure 4.2: 2D Correlation Co-efficient Score

anomaly doesn't affect the system in a huge way. Calculating the accuracy of this data we can see that 9/10 data sets follow the trend leading to an accuracy of 90

4.2 Test 2: Decisions

The next logical test to do is to test whether the decision being made. This is where the measures defined in the first paragraph of this section come into use. This test essentially tests whether the system can tell the difference between a genuine and an imposter user. In order to do this, I generated and saved some test data. Due to my program only choosing four words from an interval, each interval is comprised of only four words that are the same for each test. Any more and the program would not be able to test each interval correctly. No data is updated in the intervals and the sets of data are tested against a genuine sample of the users data. Different combinations are stored in these intervals. For example, in the first interval, all four words might be genuine whereas in the next interval, there might be three imposter words and one genuine. All possibilities are tested and written below. A copy of the data is stored inside the gitlab under test/data.

The first possibility is that one word in the four is an imposter user

- Test Results

- Calc and use FP, FN, TP, TN - get a percentage

- Discuss in relation to validation measure

- Mention struggling with small words maybe???

- Speed, security??

Chapter 5

Critical Appraisal

5.1 Summary and Critical Analysis

Whilst the produced system differs from the word explained in the outline as shown by figures 3.2 and 3.3, the functionality remains the same. I would go as far as to say that the produced system is an upgrade on the proposed system. The produced system is accurate, lightweight, secure and as such fulfils all of the original aims. Whilst some functionality around login has not been implemented due to not being needed, the expanded functionality added far outweighs this. The expanded functionality being the compression and decompression function which vastly reduce the amount of storage space being used in the user's computer, the update functionality which allows the system to learn the user's typing throughout the life cycle of the program and the multiple stages of the training functionality which allow the system to get a comprehensive and deep knowledge of how the user types. By changing my original plans and moving away from a database approach due to the high variability in the amount of data produced for each word by the system, I've reduced the performance cost because inserting and removing data from a database is a costly operation in terms of performance.

The word forming and choosing is also another way in which the functionality of the proposed system has been expanded in the produced system. In my proposed system, I planned on checking the entire intervals keystrokes and not forming words. By doing first forming words and then checking a small sample of these, the resulting system is faster and more accurate. Instead of forming one profile for an entire user, which is broad and not particularly accurate as a result, the system produces a mini profile for each word. The more data points per user which are generated by this approach allows the system to be more accurate.

The overall system is accurate and secure, as shown by figure 4.2. The system is the only data that is stored is a highly compressed version of the KDS signal. No data is transmitted off the user's system as well, ensuring that user's who are concerned about the data being shared with a third party such as advertisers can use the system without worry. The addition of the play/pause functionality is also good for users who may be worried about the system recording sensitive information. Therefore, it is clear to see that the produced system is highly

accurate and secure. When combined with the weightlessness of the program, its clear to see that the system is fit for use.

However, the produced system does suffer from some shortcomings. As mentioned previously, whilst an imposter user accessing the data stored inside the word files is not a concern due to the nature of the data within, the program suffers from a security standpoint due to the word files being named after the word within. However, because of the play/pause functionality, the user should pause the program every time they need to enter sensitive information which is a small relief from this issue. In the future, I would seek to encrypt the file names in order to mitigate this issue. The validation measure is only as good as the number of data points collected. Therefore, for words such as "hi" which are going to generate relatively few data points, the system struggles with providing an accurate validation score leading to a few cases of false negatives or positives. This is a hard issue to fix however, the obvious solution is simply to omit word with a few data points to be selected for validation. This throes up more problems however, what if small words are the only options in that interval? Another option would be to simply go into greater detail and generate more data points for those words. Unfortunately due to the nature of creating a UI in python, the provided UI for the system is rather ugly and not pleasant to look at. If I was to do this project again this is the main thing I would do differently.

Overall, I would say that producing the system went well with no major problems. The actual element I thought would be hardest and take the most effort in the actual validation system itself was easier than expected. In contrast, planning out the data flow from the raw keystroke data to the validation result took more planning and steps than expected. The UI was more of an issue that it needed to be. This is because I need to play around with different threads as I needed my program to be able to do two things at once.

5.1.1 Aims and Objective

In order to judge the success of the project, it is necessary compare the current system against the original aims and objectives of the project which are listed below.

1. To produce a lightweight key logger that can accurately log all inputs accurately and securely

2. To create a graphical system that allows the user to register or login

3. At an interval set by the user, the system will create a profile for that user based on their typing since the last interval and check this against the profile created in the registration system

4. If the user doesn't match, asks the user to re-authenticate

I feel that my project has quite comfortably succeeded in point 1. The key logger produced records all keystrokes along with the action associated with them. Furthermore, the impact on performance is minimal with the keylogger in particular not being noticeable to the user thus satsifying the accurate and lightweight requirements. The validation procedure is unfortunately heavier and

may slow down the user system a small amount. If I was to do the project again, I would reduce the performance impact on the user of the validation procedure by either implementing a faster procedure to generate the data points or by improving the storage solution. In order to implement a faster procedure, I would rewrite the project in a faster but more basic language such as C++. This would dramatically improve the speed of the validation procedure but would drastically increase the amount of code written due to the lack of suitable packages. Whilst other storage systems such as databases were explored, I would aim to go for a storage solution that could store highly variable data such as the one produced by this program, securely, speedily and with minimal impact on the user's performance and disk space. One example of a solution is a big data storage system which uses ai in order to store the data. This would also speed up the program because then the compression and decompression functions would no longer be needed which both take up $O(n)$ time each. Furthermore, this would allow me to implement clustering which would further speed up the program by allowing similar record to be stored next to each other. Overall, the produced system comfortably fulfils this aim.

My project partly succeeded in point 2. I did create a registration system but not in the sense that I meant here. In my original aims and objectives, the registration system would allow the user to sign up using username and password. In my produced system, I quickly dismissed the idea of a login and registration system as explained above. This was because I felt that the need there was no need for a system such as this, because I could just use the windows registration, login and authentication systems. These provide a more secure, fast and overall better experience for the user than any I could create myself. However, I did still create a registration system, this registration system simply was simply used to learn how the user types. This was a massive success and is a relatively complicated system that is explained better in the training section of Design and Implementation. So whilst I didn't create a login system, or the exact registration system I set out in my original aims and objective, I still feel that the produced system fulfils this aim as all the functionality that would have been provided by these systems with a extra provided by the play/pause UI, is in the produced system.

My produced system achieves the aim set out in point 3. It uses intervals and the words typed in those intervals in order to form a profile of sorts for that user which is then compared against the data learnt for that user. However, in the produced system, the user is not able to modify the interval, this is something I would change if I were to make improvements to my program. I would allow the user to choose from a small subset of possible option as allowing the user to choose any time period could easily be abused by an imposter user by setting the period to a huge amount. Bar this, the produced system not only fulfils this aim but eclipses it with the produced system going far beyond this aim.

The final aim is also fulfilled. In my produced system, if the user fails validation, then the whole system is locked using the windows locked screen and the user is asked to re-authenticate thus fulfilling this aim.

5.1.2 Time plans

5.2 Social, Sustainability, Commercial and Economic Context

5.3 Personal Development

Doing this project, enabled me to grow my skill base and as such improve my knowledge of many elements of the project life cycle. Whilst doing this project, I primarily learnt about data and how to manipulate it. I've learnt a lot about a number of different algorithms that I had no clue existed such as the Dynamic Time Warping algorithm in use in my project and the Wigners Distribution that I learnt about it in preparation for this project. I also learnt a lot about the use of git during the development life cycle. One example of how I've applied this new knowledge is my use of issues, branches and merge requests inside my Gitlab. This allowed me to develop new features whilst also safeguarding the work I've already done. Doing the project has also allowed me to further my knowledge of python which has allowed me to code in a more confident manner in the future. I've also developed further independent learning skills such as timekeeping due to the need to keep to deadlines. Doing this project whilst also juggling other university work and a part time job has ensured that I've come out of this project, not only with greater knowledge of the algorithms and language I was using but also with better time-keeping skills and a more hard working nature.

1. Summary and crit analysis

   - System works very well - provide examples using test data??
   - System is lightweight and secure
   - Compared to og planned, system is more complicated
   - Struggles with smaller words - less data points
   - NEED TO COME BACK TO THIS, NOT DETAILED AT ALL

2. Impact

   - Benefits
     - better security in combo with other sec methods
     - Lightweight and users won't notice
     - Doesn't spy on people due to only storing KDS and can turn off when user is doing something sensitive
     - Can be adapted to be used in the real word easily
   - Risks
     - Greater surveillance
     - Could easily be adapted maliciously - key logger
     - NEED MORE

3. Personal Development

- Maths, maths, maths
- Further git knowledge???
- Exp Project Development
- MORE

Chapter 6

Conclusion

Comparing the produced system and the planned system brings up a number of differences between the two. Whilst the produced system is more complex as a result of the increased features in play, it still is true to the original aims and objectives. For example, the aims stated the proposed system should be lightweight, accurate and secure and I feel that to a large extent this has been achieved. The aims also stated that the system should have the ability to form profiles for users which I feel as though my system does to an even higher standard than originally stated. The planned system should also have the ability to lock the user out which is inside the completed system. Overall, I believe that the system produced meets the original aims of the project and as such can be judged a success.

Bibliography

[1] R. Toosi and M. A. Akhaee, "Time–frequency analysis of keystroke dynamics for user authentication," *Future generation computer systems*, vol. 115, pp. 438–447, 2021.

[2] A. Abo-alian, N. L. Badr, and M. F. Tolba, "Keystroke dynamics-based user authentication service for cloud computing," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 9, pp. 2567–2585, 2016.

[3] S. Salvador and P. K.-F. Chan, "Fastdtw: Toward accurate dynamic time warping in linear time and space," 2004.

[4] H. Izakian, W. Pedrycz, and I. Jamal, "Fuzzy clustering of time series data using dynamic time warping distance," *Engineering applications of artificial intelligence*, vol. 39, pp. 235–244, 2015.

[5] R. O'connell, *Wigner Distribution*, pp. 851–854. 07 2009.

[6] R. B. Pachori and P. Sircar, "Time-frequency analysis using time-order representation and wigner distribution," in *TENCON 2008 - 2008 IEEE Region 10 Conference*, pp. 1–6, Nov 2008.

[7] N. Janakiev, "Biometric prediction on keystroke dynamics," Jun 2018. Accessed: 10/11/20221.

[8] K. S. Killourhy and R. A. Maxion, "Comparing anomaly-detection algorithms for keystroke dynamics," in *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, pp. 125–134, 2009.

[9] M. Choi, S. Lee, M. Jo, and J. S. Shin, "Keystroke dynamics-based authentication using unique keypad," *Sensors*, vol. 21, no. 6, 2021.

[10] Boppreh, "Keyboard." https://pypi.org/project/keyboard, Jan 2016. Accessed: 30/11/2021.

[11] K. Tandi, "Fastdtw." https://pypi.org/project/fastdtw/, Mar 2015. Accessed: 10/01/2022.

[12] P. S. Foundation, "Subprocess - subprocess management." https://docs.python.org/3/library/subprocess.html, Apr 2022. Accessed: 10/03/2022.

[13] S. Community, "Scipy.spatial.distance.euclidean." https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.euclidean.html, Feb 2022. Accessed: 20/01/2022.

[14] P. S. Foundation, "Getpass - portable password input." https://docs.python.org/3/library/getpass.html, Apr 2022. Accessed: 20/02/2022.