Using Keystroke Dynamics to Authenticate a User Based on their Typing

Jack Francis

April 20, 2022

Abstract

Hello

Contents

Chapter 1

Introduction

Chapter 2

Survey of Literature

To prepare to start this project, I've studied many different papers from lots of different areas.

To begin with and gain a basic understanding of how keystroke dynamics authentication works, I read Time-frequency analysis of keystroke dynamics for user authentication [1] which presented a very basic system that uses keystroke dynamics along with pure maths-based approach in order to authenticate the user. Unfortunately, this system wasn't a continuous system, but the paper still provided some value explaining the topic and allowing me to get a basic understanding of how it works. I decided to rely on this paper to base my system off because it was well-cited, researched, presented a solid approach I couldn't fault, and it was published recently and built on many others attempts at a similar system.

The paper Keystroke dynamics-based authentication service for cloud computing: Keystroke Authentication Service [2] showcase a different approach from the one above in that it makes use of a machine learning approach which is different from the previous papers purely maths-based approach. This paper also presents a continuous system which was useful when deciding to create my own. After reading and understanding this paper, I decided not to continue with a machine-learning based approach and as such this paper wasn't that useful to me and I couldn't rely on it.

Research into open-source examples was also undertaken with a look at a keystroke analysis system created by Nikolai Janakiev [3]. This system was very interesting and provided a practical look at how some had implemented a theoretical system using machine learning. The system shown was accurate but at the cost of performance. This system was based upon the paper Comparing anomaly-detection algorithms for keystroke dynamics [4] which was a very useful paper on neural network-based keystroke dynamic based authentication.

The paper Keystroke Dynamics-Based Authentication Using Unique Keypad [5] presents a system that is only used when logging in with a unique keypad which although not relevant in my case presented another way to make a keystroke dynamics authentication system. This unique keypad element of this paper wasn't useful due to my system not using a unique keypad. However, the approach they used under the unique keypad was useful and could have been used in any system, with or without a unique keypad. I didn't end up relying on this paper because I judged the approach type to be too theoretical

in nature and not capable of being translated easily into my continuous system due to its performance drain and machine-learning based nature.

The Wigner Distribution: A Time-Frequency analysis tool [6] was a good resource for further understanding the Wigner distribution which is a key algorithm in the system. The paper goes into extreme detail about this algorithm and as such was very useful in combination with the paper which originally presents the approach using the algorithm. Although the source is old, and as such may have been eclipsed by new sources, it was detailed and cited in a few other papers I read on this topic. Therefore, I'm going to rely on it when doing that part of the project.

I believe that my current research has given me a good understanding of already existing keystroke dynamics authentication approaches and will allow me to produce a working system, but I believe that I will have to do further research when I implement my secondary aim which is to ensure that the system is lightweight and not a hindrance to the user.

Chapter 3

Design and Implementation

The following procedure from section 3.2 onwards is a rough implementation of the validation procedure by Ramin Toosi and Mohammad Ali Akhaee in their excellent paper 'Time–frequency analysis of keystroke dynamics for user authentication'. [6] 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 function.

3.1   Data gathering and Forming

My program relies on capturing the users keystrokes and then processing them and then comparing them using a similarity measure. In order to do this, I decided to use the Keyboard Library REFERENCE HERE as it is a lightweight, secure and modern library that makes capturing keystrokes easy. In my project, I make use of the hook function of the library which is used to "hook" onto a users keyboard and record all of the users actions in and create keyboard events for each action. The record function which makes use of this hook function is shown in figure 3.1. The code snippet is very simple, first the program will 'hook' onto the keyboard using the Keyboard Libary 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 that be pressing or releasing a key. Further information such as the type of the action (whether it was an a press or a release), which key is this action happening on and a a highly accurate time stamp of when the event occurred. Figure 3.2, 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' which are both self-explanatory. The next field is the scan-code which is a field

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

Figure 3.1: Record Function

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

Figure 3.2: The keyboard representation of a user pressing "h"

I don't use but is useful for identifying keys easily. After this, is the name of the key which in this case is 'h' as we pressed the 'h' key down. The next field is the time since the epoch in seconds which is useful as it is this precise time-stamp that is used to do the rest of the calculations. The other three fields are device, modifiers and whether or not the user used a keypad. None of these I use in my program and as such 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 convert them into something 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 3.3 shows what the data in 3.2 looks like after going through this.
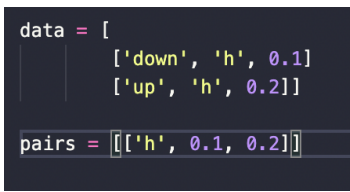
```
['down', 'h', 0.1]
```

Figure 3.3: 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 3.3.

In this form the data is unable to be used for anything, as it currently takes the form of a number of individual actions seem to have no relation to one another. Therefore, the next step is to form pairs from the data. A pair is formed of when 'down' action and one 'up' action where the key field matches and the 'down' is before the 'up'. The reason this is done is that it allows the program to work half as much data which reduces the number of unnecessary data points and allow the program to be able to form words using these pairs.

There are two main rules to follow when pairing the data. Due to the nature of how the data is collected, it is currently stored in chronological order which is very useful. In many cases, the user will press and release a

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

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

Figure 3.4: Example of simple pairing

key in quick succession without pressing any other keys. Due to the chronological nature of the data, pairing theses 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 3.4 shows an example of this type or pairing. However, this type of nice easy matching is not always the case.

In some cases, a user may press more than one key down at once. 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 3.5. 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. If the current action doesn't have a matching key and opposite action next in the array, then the program will iterate through the rest of the array starting from the current point in it searching for the next entry with a matching key and an opposite action that is after the current. If it finds one it will then 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 highly unlikely that a user will hold down a key for the entire interval. If in a rare case this occurs however, this is handled. If the method cannot find a pair, 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 3.5: Example of simple pairing

The finished algorithm is shown in Figure 3.6. The reason for the error handling is that when coming to the end of the data, attempting to access the next element to check if it can be paired up results in an index error.

The resulting pairing algorithm shown in figure 3.6 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 a 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 many different possibilities in the word forming function. This function takes in the list of pairs and returns an array of word objects. The reason I decide to

```python
def rawPairs(self):
    """
    Converts the array from the process function into key pairs

    Returns:
        2D array: Consisting of a pair of actions from the array
            above.
    """
    pairsArray = []
    for i in range(len(self.processed)):
        try:

            if (self.processed[i][2] == 'down' and
                self.processed[i+1][2] == 'up' and
                self.processed[i][0].lower() ==
                self.processed[i+1][0].lower()):
                # If the next value in the array is the up action
                pairsArray.append([self.processed[i][0],
                    self.processed[i][1], self.processed[i+1][1]])
            else:
                # Otherwise, search for the next opposing action and
                    pair them up
                for x in range(i, len(self.processed)):
                    if (self.processed[x][0].lower() ==
                        self.processed[i][0].lower() and
                        self.processed[x][2] == 'up' and
                        self.processed[i][2] == 'down'):
                        pairsArray.append([self.processed[i][0],
                            self.processed[i][1],
                            self.processed[x][1]])
                        break
        except IndexError:
            pass
    return pairsArray
```

Figure 3.6: Pair Forming Function

pivot to an object orientated approach at this point in time is that these words are heavily utilised and I would like to have methods attached to them. For example, it is far easier to generate the Key Down Signal mentioned in following sections on a word by word basis rather than having one function in the main body of the calculation class. This reduces the amount of lines written and makes code easier to read and understand.

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 the pair is added to the 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 and in some cases further action will be taken depending on what type the break pair takes. 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.

The state of the data after this section is simple. The data is an array of word objects. Each word object is in essence an array of pairs with associated timings attached. This whole section could be defined as the pre-processing of the data to get it into a format that can be used in order to perform similarity calculations.

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 3.7 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 users 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 on average time saving of XX 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
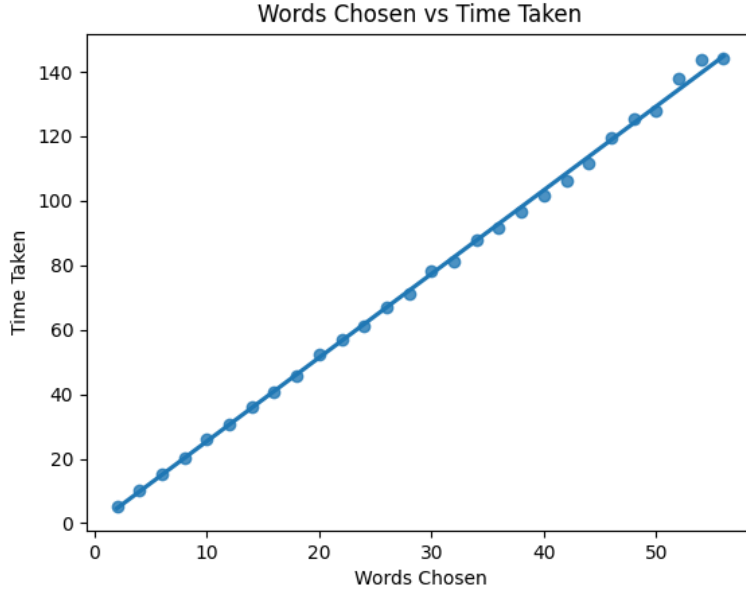
Figure 3.7: Comparing words chosen to time taken in seconds

using the calculation if figure 3.1 Figure 3.7 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. Figure The test data along with the raw data collected and the code can be seen in Appendix A.

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 an acceptable level of coverage.

The process to select 4 words from each interval is simple. Given the word list returned by the word former, the program performs the calculation shown in Figure 3.1 where w is the word list n is the number of words in the words lists returned by the word former and where k is the amount of words to be chosen.

$$EveryX(w) = \frac{n}{k} \tag{3.1}$$

This value returned by this equation is then used in the main body of the function. First of all the program chooses the first value in the words list and adds it to the output. It will select the value returned by figure 3.1. After doing this, it will add on the value returned by the equation again and select the word at that index again. This will keep happening until either the amount of words selected by the program is the number of words to be chosen. The resulting words are then returned by the function to be used in the rest of the program.

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 such as Dynamic Time Warping (DTW) and the correlation coefficient. 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). [6]

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 as the level of points being checked increases. The current level of this is set to 4 decimal points which seems to provide a good balance between accuracy and performance. However this is customisable.

$$KDS(w) = \sum_{i=w_1}^{w_n} K(w_i) \tag{3.2}$$

K is the next step of the algorithm and is heavily based on the KDS algorithm shown in [6]. 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.

$$K(t) = \sum_{x=1}^{n_k} h(t, n_i^1) - h(t, n_i^2) \tag{3.3}$$

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.
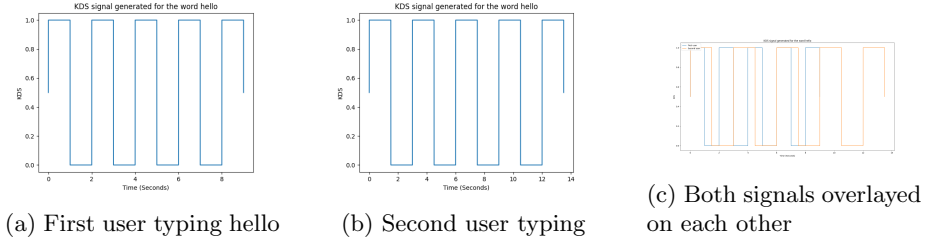
(a) First user typing hello   (b) Second user typing   (c) Both signals overlayed on each other

Figure 3.8: KD Signals generated by a user typing hello

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.

$$h(x_1, x_2) = \begin{cases} 1 & \text{if } x_1 > x_2 \\ 0.5 & \text{if } x_1 == x_2 \\ 0 & \text{if } x_1 < x_2 \end{cases} \tag{3.4}$$

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[6]

3.2.2 Output

The resulting signal can be shown easily in graph format. Figure 3.8a and figure 3.8b show the same user typing the same word twice in two different intervals. Figure 3.8c 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

3.3.1 Path

3.3.2 Cost Matrix

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. This is not done on the warped data that is produced by the Dynamic Time Warping which contrasts to the 2D correlation co-efficient. 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 interval. The confidence interval 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 interval is 0.8, then the user is rejected and asked to re-authenticate. Only the euclidean distance and the 2D correlation go into calculate the similarity score. The semantics are used to either raise the confidence interval 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 3.9. The data stored in the file and the data collected for that interval are then compared against one another. If for ex-

Figure 3.9: Example of a users semantics file

ample, the user in the past has used the shift key and in this interval has also done so, then the confidence interval 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 interval is raised by 0.02. The reason of such a small change in the confidence interval is that users habits regarding this are not typically set in stone and as such any large change will likely lead to reduced accuracy. For example, users will typically use shift when there is a single letter to be capitalised and will use caps lock when capitalising multiple keys.

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 6.2 provides a visual representation of the first part of the validation procedure. A clearer version is provided in Appendix B.

Figure 3.10: Example of a distances dictionary after validation calculations done

As shown in figure 6.2, 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 decompressed. 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

12

in a dictionary with the key being the index in the chosen array. Figure 3.10 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 (6.2), these are adjusted based on the results of the semantics validation. The values these take is further discussed in Chapter 4: Results and Discussion. The distances dictionary is iterated through and each value checked. Figure 3.5 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.

$$\begin{cases} None & e == None \ or \ c == None \\ True & c >= i \ and \ e <= b \\ True & c >= i \ and \ e >= b \\ False & Otherwise \end{cases} \tag{3.5}$$

The value returned by this function is then adding into an array which is used in the next step. Figure 3.11 shows what resulting array formed by putting the data in 3.10 through this part of the validation procedure. This example presumes that the confidence intervals for Euclidean Distance and correlation co-efficient are 1000 and 0.84 respectively.

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

Figure 3.11: The result of the data in figure 3.10 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 3.11 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 3.12 showcases what happens if only one word is in chosen and an output returned to it. If the word is true, then the person 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. At the start of my project, I was planning on creating a separate register and login system however I dismissed due to the fact that users don't want another login system and that the default windows login screen does everything to a greater standard than I ever could. In order

to perform this action, I load in the default python library sub process [3] 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. Keyboards and user storage are explained in 3.7. If the opposite happens and the user is re-authenticated successfully, then the word files and semantics are updated and the program returns true. The update process is explained in 3.6.
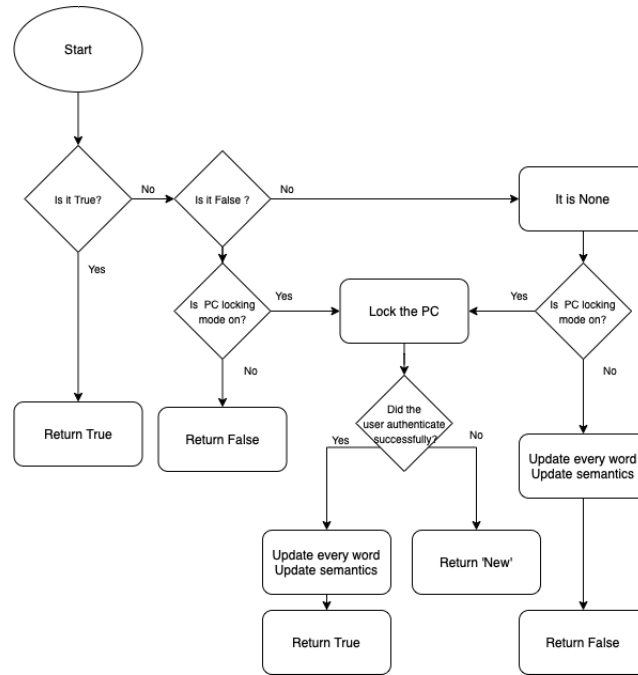

Figure 3.12: Flowchart which illustrates decision tree if only one word is chosen

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. This also needs to be handled correctly. For example, determine what needs to happen if the user types one word correctly, one incorrectly and the program has never seen another word before is no easy task. 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 The simplest validation result. True and an empty array are returned.

The runner which runs the validation process then continues on to the next interval.

Possibility 2: Some words pass validation and some words have never been seen before. Another simple result. 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 consists of locking the computer. If the user re-authenticates successfully then the program has confidence in the user, and all the words are then updated in the update function. However, if the user authenticates as a different user, then the 'New' is returned which instructs the rest of the program to create a new user.

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 and will create a new user if the user logs into a different account.

No matter what happens, two values are returned to the main program body.

- Talk about selecting values Done

- What effect does semantics have? Done

- Weighting of Euclidean vs correlation Done

- Chosen auth method Done

- In all cases what happens? Done

    - If not seen word before
    - If all validated
    - If all bar one are validated
    - etc

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.

15

$$euc(a, b) = \sqrt{\Sigma_{i=1}^{n}(a_i - b_i)^2} \qquad (3.6)$$

Figure 3.6 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 3.13. 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[1]. This is attempting to do this myself in pure python resulted in an incredibly slow processing times. The function used by scipy is much faster due to it being written in C++ which whilst having less function is far better at doing these functions with large datasets.

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 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 3.10. 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 3.13. 2D correlation co-efficient returns around 0.95 for both, whist euclidean distance returns 180 and 300 which are wildly differing figures.

Figure 3.13: Euclidean Distance compared to 2D correlation co-efficient

$$f(a, b) = \frac{\Sigma_{i=1}^{n}(a_i - \overline{a})(b_i - \overline{b})}{\sqrt{\Sigma_{i=1}^{n}(a_i - \overline{a})^2}\sqrt{\Sigma_{i=1}^{n}(b_i - \overline{b})^2}} \qquad (3.7)$$

Figure 3.7 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 3.14.

The code is self-explanatory, with the Xmean and Ymean variables simply calculating for the two input arrays respectively. The output of this function of

```python
# 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 3.14: Code Implementation of correlation co-efficient

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 some training happens. Otherwise, the program has nothing to go on and as such locks the computer at every interval. This is obviously not what I want to happen as not only is it incredibly annoying for the user but also the learning process would be incredibly 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 expored 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. Therefore this method already starts to struggle. 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."

```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 3.15: The main runner function with training intervals in

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. Furthermore, it contains a wide variety of punctuation which tests the programs ability to process this. 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, then what stops an imposter user from altering this training data. Unfortunately, nothing in my program does as yet. 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. Furthermore, more data is always better in a system such as this. 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 3.15 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 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.

The training class simply inherits from the calculation class and contains much the same data with two function added, the save all KDS and the save semantics functions.

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 actual function is simple and is shown in figure 3.16. It takes indexes of the words that failed validation or have never been seen before as input. If the array input is the same length as the chosen array (i.e. all words have failed validation or have never been seen before) then the chosen array is imply used instead. Otherwise, the words are then taken from the chosen array and stored in the intruderWords array.

After this step the program is simple, it iterates through the intruderWords array and for each word contained within, generate the KD signal, compress it and then open or create the relavent word file and store this data within.

Once it reaches the end of the array, it simply returns and the program moves on.

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. 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 3.17 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

```python
def update(self, indexes):
    """Used to update word files with new data. Used typically after
        the user has re-authenticated

    Args:
        indexes (array): The indexes of the words that need updating
            in the chosen list
    """
    if len(indexes) == len(self.chosen):
        intruderWords = self.chosen
    else:
        intruderWords = [self.chosen[indexes[i]] for i in
            range(len(indexes))]
    for x in intruderWords:
        fileName = x.word+'.json'
        Kds = x.compress()
        with open(self.pf.getKeyboardPath()+'/'+fileName, 'w') as
            write_file:
            json.dump(Kds, write_file)
        write_file.close()
```
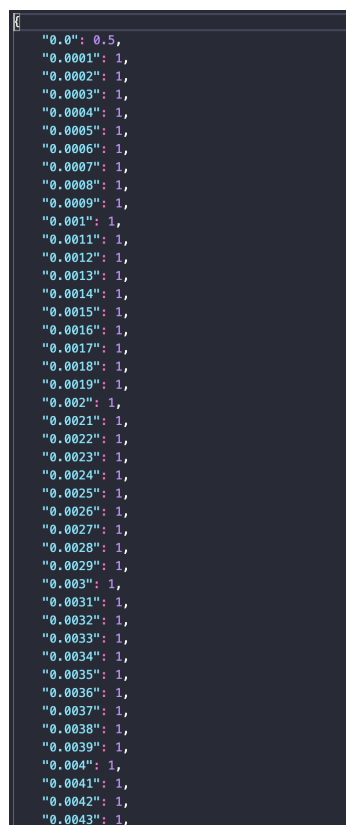
Figure 3.16: Update Implementation

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 3.17 shows a sample of the representation of the data stored without compression whilst figure 3.18 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 3.18 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 3.17 without compression is approximately 91 kilobytes whilst the compressed version shown in 3.18 is ap-

20

```
{
    "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,
```

proximately 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[5] 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 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 3.18: 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

(a) Pause UI   (b) Play UI

Figure 3.20: Example of play/pause UI in both states

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[2]. 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 3.19.

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 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.
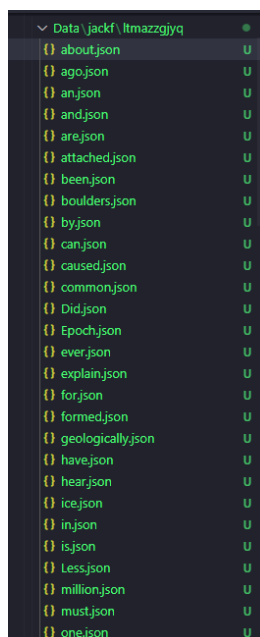
3.8 Pausing

Figure 3.19: 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

22

```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 3.21: Code for controlling calculation threads

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 3.20 showcase both states of this simple section of the UI.

Figure 3.21 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

- 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

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

Bibliography

[1] SCIPY Community. Scipy.spatial.distance.euclidean¶, Feb 2022.

[2] Python Software Foundation. Getpass - portable password input, Apr 2022.

[3] Python Software Foundation. Timeit - measure execution time of small code snippets, Apr 2022.

[4] Python Software Foundation. Timeit - measure execution time of small code snippets, Apr 2022.

[5] Ankur Naskar. What is the average typing speed, average words per minute?, May 2020.

[6] Ramin Toosi and Mohammad Ali Akhaee. Time–frequency analysis of keystroke dynamics for user authentication. *Future generation computer systems*, 115:438–447, 2021.

Appendix A

This appendix contains the test data used in figure 3.7 and the raw data in a table format for this test.

The Test Paragraph

The data consisted of a user typing the following paragraph. The paragraph consists of differing lengths and multiple forms of capitalisation and punctuation.

"Hello my name is Jack, I like to do lots of work and play counter strike. I am from Waterbeach although I am going to be living in Landbeach soon. This will be fun with my friend Alex and will ensure that I can relax and continue to work for Ben. I am looking forward to it."

Results in Table Form

The results in table form are here. As you can quite clearly see the time increased linearly. The speed of this is impacted by the speed of the computer it is running on. A faster computer will mean that the results are significantly faster.

Number of Words Chosen	Time

Code Used

The code used to do this is below. It consists of me using running the validation method with different amounts of words chosen each time until we reach the max number of words chosen.

The code here is very simple and makes use of the rest of my source code. The timeit function[4] is part of the default python installation provides a useful function that allows me to accurately time how long functions take.

The data stored in the times dictionary is then plotted with the keys plotted against the values.

```python
times = {}
# The typing data is loaded in from a file
file = open("data.pickle", 'wb')
data = pickle.load(file) # And then unpacked
start = 1649857380.5629547 # The start time of when the data was
    collected in epoch time
prof = pf.User_Profile() # The user profile used
presave = t.Training(data, start, prof, 1,0) # Saving all word
    files immediately

interval = i.Calculation(data, start, prof, 1)
for x in range(2, interval.noWords, 2):
  print(x)
  interval.chosenAmount = x
  interval.chosen = interval.choose()
  start = timeit.default_timer()
  _, _ = interval.validation(mode='rnl') # Test mode, ensures the
      pc does not lock or update the data
  times[x] = timeit.default_timer() - start
```

Figure 6.1: Word Selection Testing

Appendix B

This contains details regarding the validation procedure.
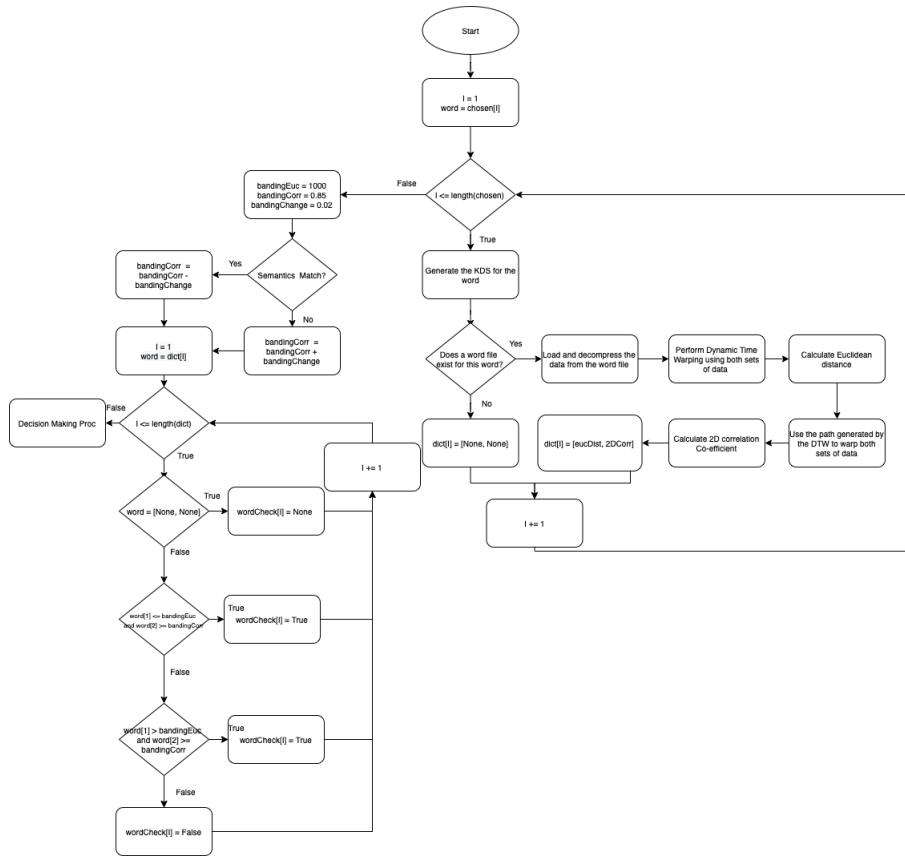
First Validation Proc Flowchart

Figure 6.2: FLowchart of first element of validation procedure.

Appendix C

This chapter contains details of the training data test.