ECE 590 Final

Alexander Migala

April 2023

Abstract

In my final project for ECE590: Brain Computer Interfaces with Dr. Collins, I question why the scientific community invests so much time into the P300 Speller, a spelling machine that is used by people with locked-in ALS. I devise a thought experiment about a solution that is better than the P300 Speller, create it in Python, and compare my machine against the P300 Speller using probability. I find that, in the end, my machine ultimately can outperform the P300 Speller, but I recognize that there are other factors a patient with ALS may consider when choosing a device.

Contents

1 Motivation

As I learned about the P300 Speller in class, a widely-used communication device for "locked-in"[1] ALS patients, I wondered why this device is still in such heavy use[2]. The P300 Speller works by incorporating EEG[3] data directly from brain waves, eye-tracking data, or some other combination of inputs[4]. To use it, a patient must first stare at the letter they would like; the Speller flashes over the character set[5], and then correlates the input data with a given character. This, of course, seems like a fair solution, given that it does not require any muscle input to produce an output. However, I was surprised to learn that the P300 Speller is still widely used today, given that it is only able to output around 1 word per minute[6]. I would think that there would exist a better solution in today's world.

For example, Dr. Steven Hawking, a renowned scientist with locked-in ALS, used a custom spelling machine with an input from his cheek muscle, and his EEG waves were not required [7]. Before that, he was using a handheld clicker. Therefore, there are people who have found solutions that are better than the P300 Speller in the modern day.

Taking a naive student approach, I then concluded that I could build a machine that was better than the P300 Speller. Through this process, I shed light on why this problem is much more complicated than it seems.

2 My Solution

As mentioned before, the P300 Speller works by flashing over a series of characters: the alphabet, a host of functional characters, and special characters. This, to me, seemed redundant. If a patient wants so merely spell "hello", why is the machine wasting time by flashing over non-alphabet characters? While it's true that the machine can be optimized to give less weight to these characters, it still stands that a new machine can be devised that gets to the desired character in fewer steps. I estimated that the next-best solution would be to use a binary search methodology, where the patient is prompted between a left and right option, thereby minimizing the selection process to only two options at a time, rather than 26+ at a time. A theoretical selection process to spell "hello" would then look like:

1. Left: 'a-m', Right: 'n-z'. Choose Left
2. Left: 'a-f', Right: 'g-m'. Choose Right
3. Left: 'g-i', Right: 'j-m'. Choose Left

[1] Patients who can no longer voluntarily move any of their muscles
[2] Especially since this device was first brought to the scientific community in 1988[2]
[3] Electroencephalogram
[4] Depending on how locked-in the patient is
[5] As found in class, the *way* in which these are shown directly impacts performance
[6] Metric given in class

4. Left: 'g-g', Right: 'h-i'. Choose Right

5. Left: 'h-h', Right: 'i-i'. Choose Left

The character was selected in a mere five moves, which appears to use much less moves than the P300 Speller, which requires around 19 samples[7] [5]. The number of moves selected is also diminished when autocomplete is used, which is discussed in the next section.

Of course, there is a question of *how* a patient is to use my system. My system theoretically can be setup for any patient that is able to input two distinct interactions or is able to use eye-tracking. This appears to be a plausible setup, given that Hawking, who was almost entirely locked-in, was still able to use his cheek muscle as an input; in his case, this could be combined with eye-tracking to speed up inputs. Thus far, then my machine appears to be theoretically better than the P300 Speller, as it requires fewer instructions to produce a character.

2.1 Autocomplete and Machine Learning

The other aspect of the P300 Speller that frustrated me was that the solutions, at least the ones presented in class, didn't use any sort of language model or, more simply, autocomplete. As such, I wanted to incorporate a simple auto-complete model that would speed up my machine even more so. After researching some solutions, I elected to design a bi-gram and tri-gram language model[8] that updates based on what the patient has input. While I borrowed the statistcal model from GeeksForGeeks, I wrote my implementation [3]. Specifically, my gram model works by:

1. Load-in saved words from the user

2. For each word in the repository, take an n-length substring from it (the "stem" of the word). For every other word in the repository, find other words that have this same stem. Using this information, use a hashmap to store the stems to the endings[9]

3. Do the above for every word and for every n desired up to maximum N (in my implementation, good results were achieved with N=3)

Then, for the model to predict a word, the user begins typing their target. Upon each character addition, the machine searches the tree for a matching stem. If it finds one, it counts the number of possible endings, and measures whether there is an ending with a probability higher than a set threshold[10].

[7] Though this may be modified depending on how accurate the system is intended to be

[8] I also set this up to where higher-order "N"-gram models can used, though they are more resource heavy on the host machine

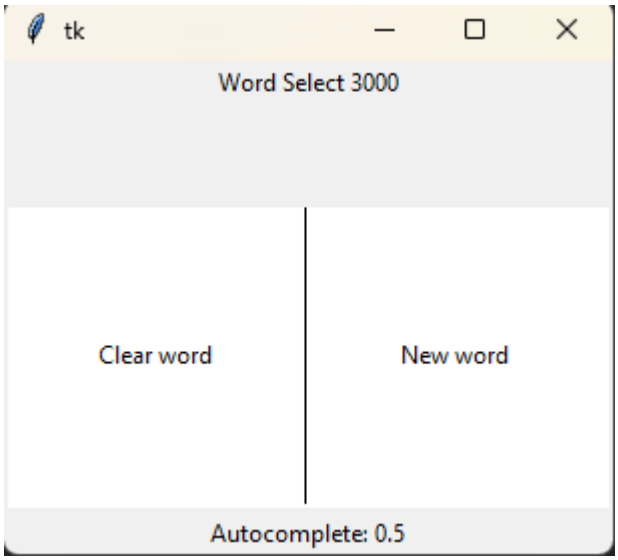
[9] A hashmap was used because it can achieve $\mathcal{O}(N)$ or better performance upon data retrieval

[10] Which can be tuned! In my implementation, I use 0.5

After every finished word, which is denoted when the patient selects the "New Word" option, a word is added to the tree. When the program is off, the autocomplete program may rebuild and store the tree with the old words. The machine, overall, then, learns the patient's most-used vocabulary.

2.2 The Program

To bring my though experiment to life, I wrote a Python GUI[11] program that emulates the machine in question. It is written in Python, which means that it can run on a surprisingly large host of machines, making it widely-accessible[12].



3 Results

In my initial goal, I set out to build a machine that can achieve a character with fewer steps than the P300 Speller. On paper, I appear to have achieved that; however, the real-world results are where I have some problems. Most notably, my machine has a pretty steep learning curve, since we usually don't find ourselves memorizing the n-th division of the English alphabet. Additionally, I didn't include any special characters, or a way to delete a word; frustratingly, the user must hit "New Word" to begin typing a new word, which means that there are garbage words going into the autocomplete model[13]. While my ma-

[11] Graphical User Interface

[12] Specifically, it can even be run on single-board computers such as the Raspberry Pi, meaning that the physical machine could be as small as a handheld device

[13] Though, in complete fairness, this isn't too unlike how non-ALS people use their phone. How many times has your phone autocomplete'd to some nonsense word from before?

chine could likely be expanded to include a delete option[14], there is a bigger problem: it is not fun to use. While I personally have never used a P300 Speller myself, I can say that my implementation is, in a word, boring. Dr. Collins provided anecdotes for P300 Speller study participants in which they simply get bored when using it. I cannot say, with confidence, that my machine has performed any better than the P300 Speller on this front.

3.1 Probability Analysis

Because I do not hold IRB approval, I can't test out my machine on anyone, so I cannot directly measure my words-per-minute (WPM) performance over the P300 Speller. However, I can try to model this metric using probability. I borrow ideas outlined by Karl Sigman from Columbia University [6]. Because I have already demonstrated why my machine would outperform the P300 Speller if two distinct inputs are allowed, I will elaborate on the niche input of eye-tracking, as both implementations can be setup to use this as an input. To do this, I model a patient's eye tracking path as a stochastic process, starting from a random point on the screen. For example, if the user wants to select the left option, their eyes may start anywhere on screen, then gradually, using a random walk, move across to that side. After a certain amount of time[15], the eye-tracker would find that the gaze is on that side. To model this random-walk, I employ the use of a simple[16] Brownian motion. Each individual step that the eyes are allowed to make are independent and follow the same distribution. Because my machine only needs to select between left and right, vertical eye-tracking data is not factored into the model[17]. Thus, an independent eye-moving step is assumed to follow a normal distribution:

$$Z(\Delta x) \sim \mathcal{N}(\mu, \sigma^2) \tag{1}$$

Where μ and σ are approximated is discussed in the next section and Δx is the absolute x-distance between the randomly-chosen initial point and the final point:

$$\Delta x = |r_{ix} - r_{fx}| \tag{2}$$

Given an initial location on the screen, r_{ix}, the final location x-coordinate, r_{fx} is given by the stochastic process:

$$r_{fx} = r_{ix} + \mu\Delta x + \sigma Z(\Delta x) \tag{3}$$

Therefore, the final position has the distribution:

[14] I would put this as a third option, before any character is allowed to be input, though this partially defeats the binary-search aspect I wanted to achieve for this thought experiment

[15] How much time is difficult to say, though the machine would need to be tuned to give enough time for patient to physically move to that side of the screen

[16] Non geometric

[17] It's also not included because the distribution that the eyes would follow becomes multivariate-normal, and this is much harder to work with analytically

$$r_{fx} \sim \mathcal{N}(\mu\Delta x, \sigma^2\Delta x) \tag{4}$$

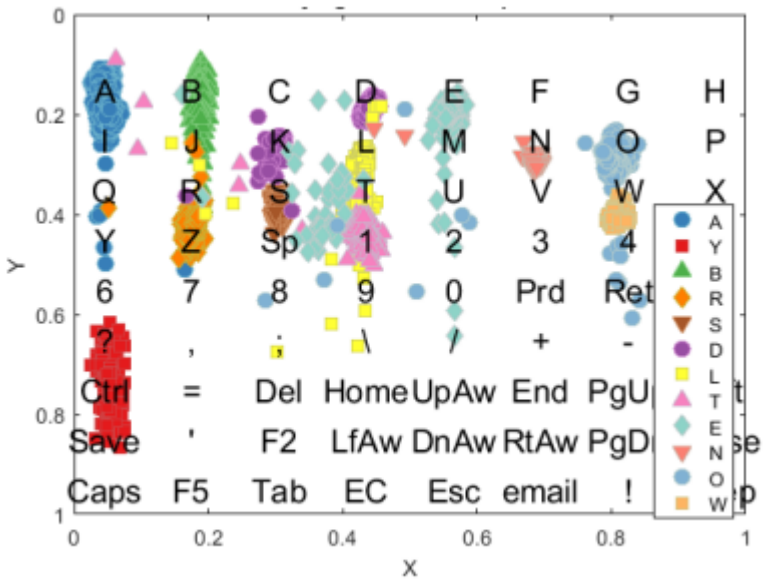
Using this construction, I can then show why my machine is better by examining the worst case scenario. Assuming that the program's window is 100 pixels wide (so 50 pixels per selection site). We let the left side target coordinate to be 25 and the right side target coordinate to be 75. Now, we can approximate the worst case scenario as having to traverse opposite sides. For example, if the user wants to select the left option, but starts on the far right of the screen, the probability of success is given by the cumulative distribution of the possible final x-values on that side of the screen, namely:

$$\int_{-\infty}^{50} \frac{1}{\sigma\sqrt{\Delta x}\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{x' - 25 - \mu\Delta x}{\sigma\sqrt{\Delta x}}\right)^2\right) dx' \tag{5}$$

In the worst case, the initial eye gaze is on the right-most part of the screen, so $\Delta x = 50$[18]. Similarly, the worst-case for the right option is when the user starts their gaze on the left side.

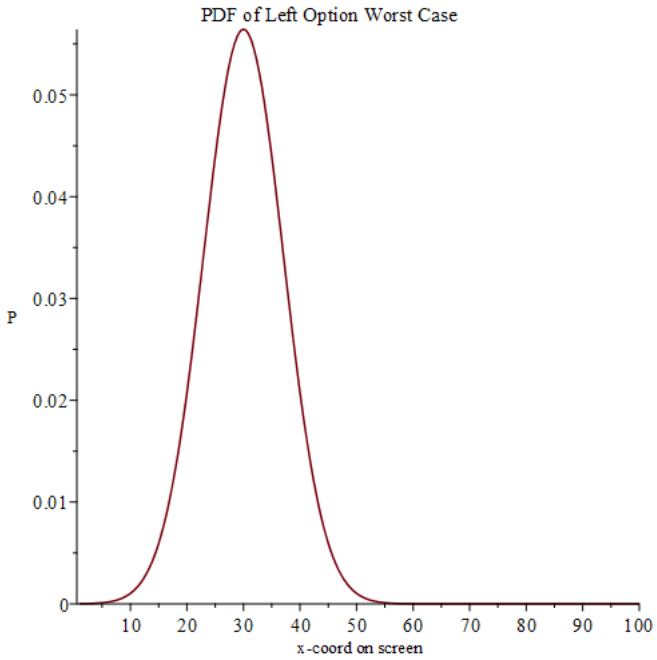
$$\int_{50}^{\infty} \frac{1}{\sigma\sqrt{\Delta x}\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{x' - 75 - \mu\Delta x}{\sigma\sqrt{\Delta x}}\right)^2\right) dx' \tag{6}$$

To evaluate these integrals I needed to find values of μ and σ. I found these in Professor Mainsah's slide-deck [4], on page 16, in this graphic featuring eye-tracking data from an ALS patient:

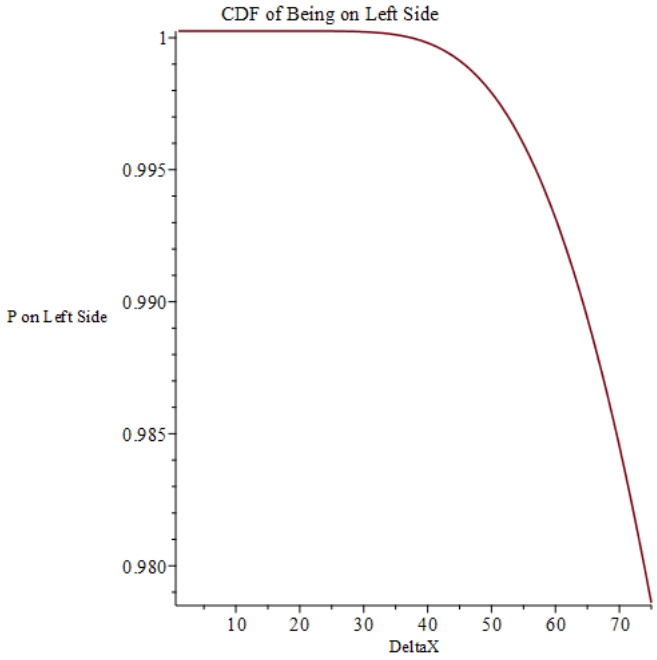


[18] This value is the same for the opposite case as well

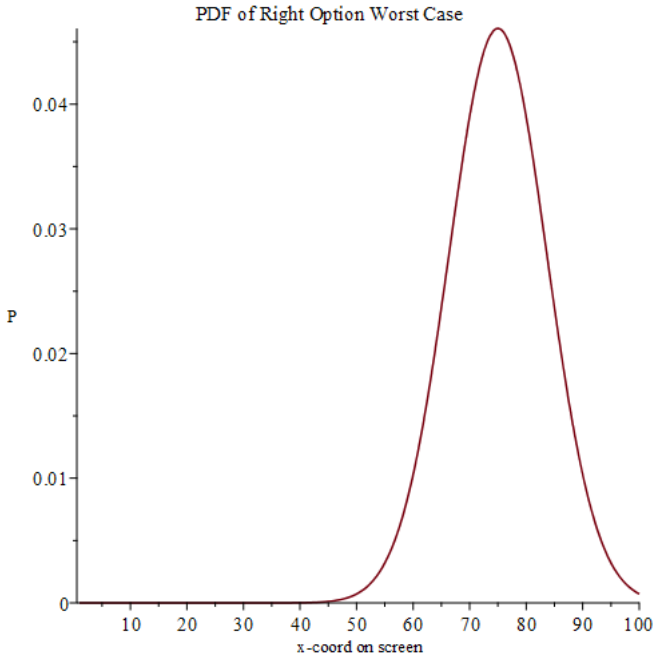
Fortunately, these values are normalized, meaning we can use the percentages right from the figure. From this, we can see that the average distance between the dots, which is the distance between gaze measurements, is very small. For example, take patient A, which appears to have gone "up" towards the A button. The average is around .1%, and the distribution is skewed. The distribution can be approximated to have a $\sigma \approx 1\%$ as well. Using these values, equations 5 and 6 can be solved. In the left-option choice worse case scenario, the probability density function is graphed as:



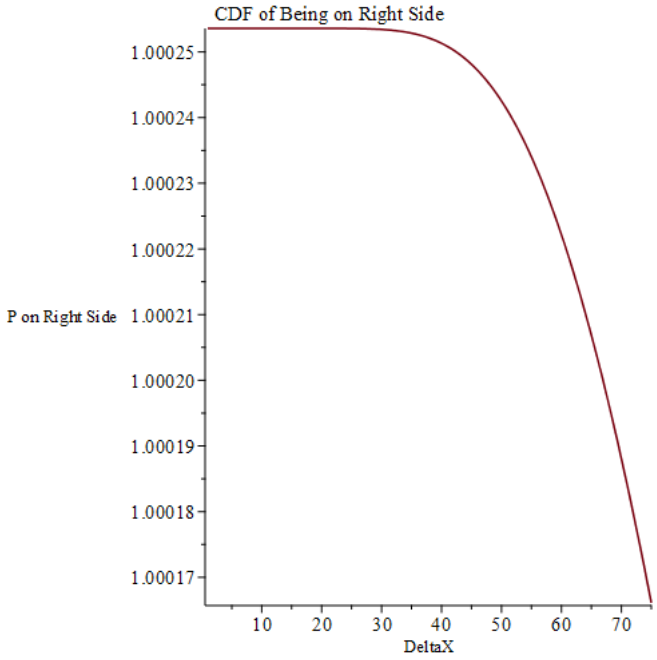
Then, for any Δx, the cumulative density function is:



Similarly, the PDF for the right-option worst case is:



Then, for any Δx, the CDF is:



We can see that, even when Δx is maximal (≈ 75), then the chance the user selects the desired option is greater than 90%. The accuracy could also go down, depending on how developed the autocomplete tree is. For example, for an autocomplete threshold of 75%, the overall accuracy is $0.75 * 0.9 \approx 67.5\%$. This minimum is on par with the P300 Speller, which has an accuracy that ranges from around 61-80% for ALS patients [5]. My accuracy makes some sense given that there are only two options, and I've modeled only the x-coordinate. One area of concern is that Dr. Mainsah's graph didn't give me the time separation between the measurements. Thus, my accuracy analysis is accurate for patients gazing for a sufficiently long enough time, depending on the eye-tracker used. According to these results, then, my machine would likely achieve similar accuracy with fewer moves.

3.2 Success?

Based on my probability, I have created a theoretically better machine than the P300 Speller; however, mine has significantly less output options, and is not convenient to use. Thus, I would speculate that a real-life ALS patient would prefer their standard P300 Speller over my solution. However, in this simple thought experiment, I have demonstrated that it is possible to devise a different solution that can outperform the P300 Speller on some of its metrics; so, why do we still research it? From this experiment, I've found that the P300 Speller is likely still studied because it is the system that has the best support, which matters when someone's life depends on it. For example, I, personally, would not

want to use a slow and glitchy iPhone or computer because it could hinder my everyday workflow. I would estimate that this is even more drastic for patients with ALS. Thus, it is important for their communication system to be well understood, debugged, and tested. Additionally, since ALS patients are few in number (around 18,000 continuously in the US [1]), updating the population with new hardware (that they've probably been using for a long time!) is simply too cumbersome and time-consuming. Therefore, I must conclude from my experiment that scientists should look for alternative solutions to the P300 Speller, but aim to improve the lives of ALS patients who already use the machine by making the software more efficient. Moreover, scientists should focus on designing new systems that are simple, easy-to-learn, low-cost, and require non-EEG input schemas. Said differently, scientists should design a new system that allows for a seamless transition from a P300 Speller.

4 Appendix

Here I include the code for the emulator. The full GitHub repository for the project can be found at: `https://github.com/stlgolfer/ECE590-Final-Project`.

4.1 GUI

```
import pickle
import tkinter as tk
from autocomplete import predict

def bifurcate(arr):
    split = int(len(arr) / 2)
    return arr[0:split], arr[split:]

root = tk.Tk()
root.maxsize(600,600)

left_initial = ['a', 'b', 'c','d','e','f','g','h','i','j'
    ,'k','l','m']
right_initial = ['n','o','p','q','r','s','t','u','v','w',
    'x','y','z']

left_current = left_initial
right_current = right_initial
output_current: [str] = [] # keep this as a character
    array to allow for pushing
mode_select: bool = True # if true, show initial screen
    before left and right initial: either delete current
    word or make new one
AUTO_COMPLETE_CONFIDENCE = 0.5
saved_words: [str] = [] # words to be saved and then
    loaded from for autocomplete

# load autocomplete
nstubtree = None
with open('./autocomplete.pkl', 'rb') as f:
    nstubtree = pickle.load(f)
print(nstubtree)
def display_output():
    global output_current
    typing_area.config(text = ''.join(output_current))

    global nstubtree
    global mode_select
    # global auto_complete_suggestion
```

```
    # attempt an autocomplete
    autocompleted = predict(typing_area.cget('text'),
        nstubtree,AUTO_COMPLETE_CONFIDENCE)
    print(f"Autocompleted: {autocompleted}?")
    if len(autocompleted) > 1:
        typing_area.config(text=''.join(autocompleted))
        output_current = []
        # reset the screen

    mode_select = True
    char_canvas.delete('all')
    char_canvas.create_line(150, 0, 150, 150)
    char_canvas.create_text(75, 75, text='Clear word')
    char_canvas.create_text(225, 75, text='New word')

def do_click(event):
    global mode_select
    global left_current
    global right_current
    global output_current

    if mode_select == True:
        if event.x > 150:
            mode_select = False
            char_canvas.delete('all')
            char_canvas.create_line(150, 0, 150, 150)
            char_canvas.create_text(75, 75, text=f'{
                left_current[0]}-{left_current[-1]}')
            char_canvas.create_text(225, 75, text=f'{
                right_current[0]}-{right_current[-1]}')
        else:
            # also save the word, if not already
            if len(typing_area.cget("text")) > 0:
                saved_words.append(typing_area.cget("text
                    "))
            typing_area.config(text='')
            output_current = []
    else:
        mode_select = False
        char_canvas.delete('all')
        char_canvas.create_line(150, 0, 150, 150)

        # first check if that's the only option left
        if event.x < 150 and len(left_current) <= 1:
            print(f"selected {left_current[0]}")
            output_current.append(left_current[0])
```

```
            display_output()
            left_current = left_initial
            right_current = right_initial
        elif event.x > 150 and len(right_current) <= 1:
            output_current.append(right_current[0])
            display_output()
            left_current = left_initial
            right_current = right_initial
            print(f"selected {right_current[0]}")
        else:
            left, right = bifurcate(left_current if event
                .x < 150 else right_current)
            print(left)
            print(right)

            # and set the global vars to match
            left_current = left
            right_current = right
            char_canvas.create_text(75, 75, text=f'{
                left_current[0]}-{left_current[-1]}')
            char_canvas.create_text(225, 75, text=f'{
                right_current[0]}-{right_current[-1]}')

main_frame = tk.Frame(root, width=300, height=300)

# first thing to do is to have the typing area and then
    two canvas sections
# need a canvas because we'll need to control
text_frame = tk.Frame(main_frame, width=300,height=50).
    grid(row=0,column=0)
typing_area = tk.Label(text_frame, text="Word Select 3000
    ")
typing_area.pack()
# typing_area.configure(state='disabled')

char_canvas = tk.Canvas(main_frame, bg="white", height
    =150, width=300)
# char_canvas.create_rectangle(0, 0, 50, 50)
char_canvas.create_line(150,0,150,150)
char_canvas.create_text(75,75,text='Clear word')
char_canvas.create_text(225,75,text='New word')
char_canvas.grid(row=1,column=0)
char_canvas.bind('<Button-1>', do_click)

auto_complete_suggestion = tk.Label(main_frame, text=f"
    Autocomplete: {AUTO_COMPLETE_CONFIDENCE}").grid(row=2,
```

```
        column=0)

def on_close():
    print("Dumping saved words for autocomplete")
    print(saved_words)
    with open('./saved_words.pkl', 'wb') as file:
        pickle.dump(saved_words, file)
        root.destroy()

root.protocol("WM_DELETE_WINDOW", on_close)
# canvas.pack()
main_frame.pack()
main_frame.mainloop()
```

4.2 Autocomplete Tree Builder

```
import numpy as np
import tensorflow as tf
from keras import layers
from keras.models import Model, load_model
# from keras.utils import plot_model
import matplotlib.pyplot as plt
from convokit import Corpus, download, download_local
import pickle
from collections import Counter
from english_words import get_english_words_set

# need to load in any extra words in saved_words, if
    available
to_append = pickle.load(open('./saved_words.pkl', 'rb'))
words = ['hello'] # given at least first two characters

# keys will be the number of characters available, the
    tree will have the words with associated probabilities
nstubtree = {}

MAX_STEM_LENGTH = 3
for sl in range(2,MAX_STEM_LENGTH+1):
    words_cleaned = [len(x) <= sl + 1 for x in words]
    tree = {} # will have a key with first two characters
        , then it will iterate through the rest of the
        words and append any endings it finds

    # reject all words that have length less than or
        equal to 3
    for wi in words:
        key = wi[0:sl]
        if key not in tree:
            tree[key] = [wi[sl:]] # add its ending
        else:
            # append it's ending
            tree[key].append(wi[sl:])
    nstubtree[sl] = tree

print(nstubtree)
with open('./autocomplete.pkl', 'wb') as f:
    pickle.dump(nstubtree, f)

def predictn(stem: str, tree: dict, threshold: float) ->
    str:
```

```
    if stem not in tree:
        return ''
    # for now just print the options
    counted = Counter(tree[stem])
    print(counted)
    # we now have the counts, so just normalize against
        the total
    for k in counted.keys():
        counted[k] = counted[k]/len(counted.values())
    print(counted)
    # now only return an autocompleted word that meets
        the criteria. select the first one that 'comes to
        mind'
    for candidate in counted.keys():
        if counted[candidate] >= threshold:
            return stem + candidate
    return '' # didn't find one

def predict(stem: str, tree: dict, thresh: float) -> str:
    if len(stem) not in tree:
        return ''
    else:
        # there's a key for it
        return predictn(stem, tree[len(stem)], thresh)
```

References

[1] Als. https://www.mass.gov/info-details/als-lou-gehrigs-disease. Accessed: 2023-04-30.

[2] L.A. Farwell and E. Donchin. Talking off the top of your head: toward a mental prosthesis utilizing event-related brain potentials. https://doi.org/10.1016/0013-4694(88)90149-6. Accessed: 2023-05-02.

[3] Geeks for Geeks. N-gram language modelling with nltk. https://www.geeksforgeeks.org/n-gram-language-modelling-with-nltk/. Accessed: 2023-05-02.

[4] Dr. Boyla Mainsah. Brain-computer interface (bci) datasets (lecture). ECE590: BCI.

[5] E. W. Sellers and E. Donchin. A p300-based brain-computer interface: Initial tests by als patients. https://doi.org/10.1016/j.clinph.2005.06.027. Accessed: 2023-05-02.

[6] Karl Sigman. Notes on brownian motion. http://www.columbia.edu/~ks20/FE-Notes/4700-07-Notes-BM.pdf. Accessed: 2023-05-02.

[7] Wired. How intel gave stephen hawking a voice. https://www.wired.com/2015/01/intel-gave-stephen-hawking-voice. Accessed: 2023-05-02.