

Typing Mania

Francis Weate z5160184

November 2021

Contents

1	A Typing Algorithm	2
1.1	Typing efficiency	2
1.2	Dataset:	2
1.3	Similarity Between Words	3
1.3.1	Why adjacency list	3
1.3.2	String Similarity Metric	4
1.3.3	Recommendation:	6
1.3.4	Popularity:	6
1.4	Word Rank:	7
1.5	Random Surfer	7
1.5.1	Law of Large Numbers:	8
1.5.2	Results of Surfing:	8
1.6	Recommendation:	9
1.6.1	Back to Alpha:	11
1.6.2	Program Example:	11
1.7	Testing:	12
1.7.1	Testing alpha:	13
1.8	Measuring Typing Speed:	16
2	Code:	17
2.1	word_rank.py	17
2.2	word_similarity.py	19
2.3	create_similarity_graph.py	20
2.4	recommend_words.py	21
3	Conclusion	23
3.1	Potential Improvements	23
3.2	What I learnt	23

1 A Typing Algorithm

<https://github.com/frankers3/TypingMania>

As someone who has recently begun to touch type. I saw this as the perfect opportunity to create an Algorithm to help me learn. Most touch typing tutors online often use pre-written text files (typing.com) or made up words (keybr.com) to teach touch typing. While I definitely see both websites as useful resources, I believe that they are efficient in helping someone to learn how to touch type, as they help with learning key positioning. However once someone knows the basics, it is much more useful to focus on a more algorithmic approach to finding words to type.

A recommendation system that finds words similar to badly typed words could be very beneficial to someone looking to increase their typing speed.

1.1 Typing efficiency

What constitutes typing efficiently? A common typing measurement is Words Per Minute (WPM). WPM is in fact a misnomer, as the calculation does not actually use words at all, but instead uses substitutes a word for $\frac{\text{characters}}{5}$. So if someone typed 1000 characters in 4 minutes, we could calculate their WPM as follows:

$$WPM = \frac{\frac{1000}{4}}{5} \quad (1)$$

therefore $WPM = 50$ in this case. Instead of ignoring words entirely and just looking at random keystrokes, it is more useful to see how the words that we use impact our typing speed. For example, someone in their everyday life will type the word 'useful' much more frequently than they will type the word 'noncom' (a word for a non-commissioned officer). For this reason it would be much more useful to be able to type "useful" quickly instead of the word "noncom". For that reason I would argue that WPM is not a useful metric for gauging a person's typing speed. Consider a word w from a dictionary of length d , let s_w equal the speed required to type it, and p_w equal its popularity.

$$T_s = \sum_{k=1}^d \frac{s_k p_k}{\text{total}(p)} \quad (2)$$

This metric makes more sense in terms of typing purposes as more common/popular words are weighted more highly.

1.2 Dataset:

The dataset I used for this project is <https://www.kaggle.com/ratatman/english-word-frequency> which "contains the counts of the 333,333 most commonly-used single words on the English language web, as derived from the Google Web Trillion Word Corpus." This is very useful as it allows for a gauge of the popularity

of each word and thus its usefulness in learning how to type it. However some words in this data-set were not at all useful, as the dataset is made from search terms, there are a lot of spelling mistakes such as "seach" and random acronyms "fft" (fast Fourier transform), which is a great algorithm, but the acronym is hardly useful in a typing system.

To resolve this issue I took the intersection of this dataset along with the unix dictionary found in "/usr/share/dict/words". This got rid of random acronyms and spelling mistakes from the dataset. Leaving only 61,426 words in the dataset, as opposed to 333,333. Meaning that I removed 81.6% of entries from the kaggle dataset.

1.3 Similarity Between Words

It is useful to know words which are typed similarly to each other for multiple reasons as it helps with recommending words to type as well as establishing how popular a word is. To map words based on similarity I created an adjacency list graph containing words with a similarity of greater than 0.4 in the dictionary to that word. For example the top 10 words in the adjacency list for the word "the" are:

```
the,"(['thee', 0.8333333333333334), ('they', 0.75), ('them',
      0.75), ('then', 0.75), ('thew', 0.75), ('he',
0.6666666666666666), ('there', 0.6666666666666666), ('these',
0.6666666666666666), ('theme', 0.6666666666666666), ('theft',
      0.6666666666666666), )...]
```

These words all have a distinct similarity with respect to how they are typed to the word "the". Not only is this graph useful for obtaining the overall rank of all the words in the dataset, but it also helps the recommendation system find similar words to recommend.

1.3.1 Why adjacency list

while an adjacency matrix is $O(1)$ for retrieving the weight at $edge_{ij}$, however its space complexity of $O(n^2)$ is an issue. with 61,426 words and assuming a 4 byte edge weight that would take up $4 \times 61426^2 = 15,092,613,904$ of 15 Gigabytes of space. Along with this issue, using all that space would be wasteful since the similarity of words such as "the" and "microscope" would be compared and stored in the graph. These words are clearly not similar at all and comparing their similarity is trivial and not helpful in the functionality of the program.

An adjacency list made much more sense as it takes up less space and allows for only fairly similar nodes to be connected. The graph itself is kept in a local file "graph.csv" with an array of offsets being used by the recommendation system to instantly access a node from the graph. The weights of the graph are defined

as the degree of similarity that word i has with word j . so at row i of the adjacency list, if j has a high enough similarity to i , an edge of weight similarity will be added to the graph.

1.3.2 String Similarity Metric

There are multiple methods to determine the similarity of two strings to each other:

- **Hamming distance:** the distance of two strings from each other is calculated by adding up the positions that they differ, so the hamming distance of "step" and "stat" is 2. This means that the algorithm runs in time $O(n)$ where 'n' is the length of the two strings. The two main limitations of this model for our purposes is that firstly the string must be of equal size. Secondly (and the main problem) is that with this problem, the substrings within the strings are more informative of similarity than just letter positions. For example when looking at similar strings to "the", the word "he" is much closer to it than the word "toe", even though both distances would be 1. This is because of the "he" substring in "the" and we must find some way of expressing that similarity in our model. An example with bits can be seen below:

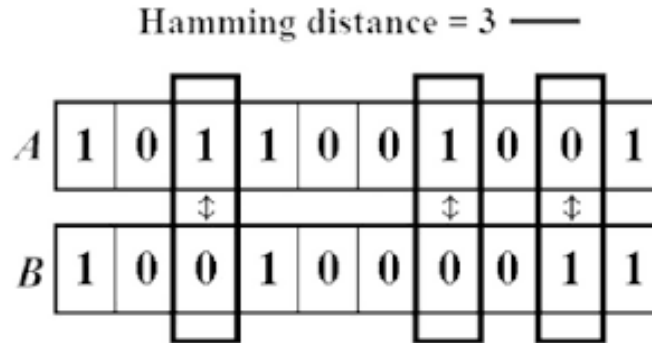


Figure 1: Hamming Distance of 10 bits

- **Levenshtein distance:** The Levenshtein distance works on the minimum number of single character insertions, deletions and modifications needed to make string1 into string2. To implement this requires filling out a $m \times n$ matrix where 'm' and 'n' are the two string lengths, so the runtime of this algorithm is $O(m \times n)$. It is a more useful metric than Hamming distance due to its ability to work with different strings lengths. Like Hamming distance however, it still fails to take into account the substrings within the string, which is something that needs to be taken into account for the

similarity two strings when they are typed. For example looking at the word "the", the word "tap" has the same Levenshtein distance as the word "their". This model clearly does not accurately display the similarity of strings with respect to typing. Both algorithms' uses are more focused on finding typing mistakes by the user rather than words that are actually typed similarly.

- **Custom formula with Convolution:** For this algorithm the similarity between two strings should be based off the amount and the length of the substrings within both strings. Convolution was used here to find all the substrings within two strings. However due to the relatively small string length, the overhead cost of implementing a fast fourier transform $O(n \log(n))$ seemed to outweigh the benefit it would provide. Therefore an $O(n^2)$ system was used, since n (string length) in this case was small, a lot of concern should not be placed on the scalability of the algorithm for when n is large.

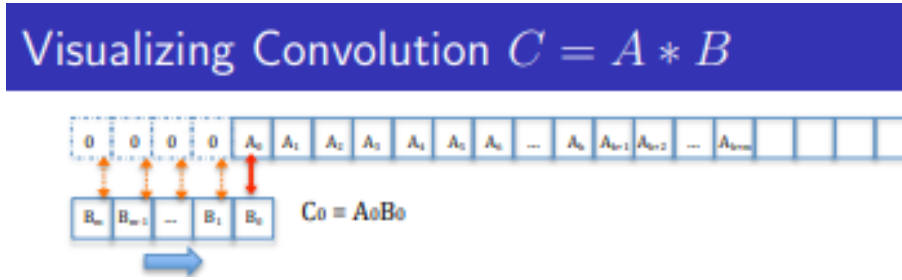


Figure 2: Convolution of two strings

One thing of note is that when using this convolution method, substrings within other substrings were not counted. For example if the substring "ing" was found, then "in", "ng", "i", "n", "g" were all ignored. Once all the substrings have been found, we need to calculate similarity. The actual strings and substrings are not help anymore, only their lengths. Let s_1 and s_2 be the length of two strings with a set of the length of substrings w_n with n elements.

$$similarity = \frac{\sum_{k=1}^n w_k^2}{s_1 \times s_2} \quad (3)$$

This equation for word similarity is very useful. Longer substrings within strings suitably compensated by the power of two. Meaning that one substring of length five is way more valuable than five substrings of length one. Since typing is all about how keystrokes are positioned next to each other, this is a very useful feature.

- One fallback of this method is looking at strings of vastly different sizes, that still have common features. For example if we compute the similarity of "theorize" and "the" we get

$$\frac{3^2 + 1^2}{3 \times 8} = \frac{10}{24} \quad (4)$$

this hardly seems like a fair similarity value for the two strings especially when considering the fact that "the" is the substring within "theorize" and thus matches it completely. For this reason I adjusted the value by taking the \log_e of the lengths of each string and multiplied it by this factor:

$$\frac{2}{2 + \log_e\left(\frac{s_1}{s_2}\right)} \quad (5)$$

so if string one substantially smaller than string two, then it will be ranked more highly. Although I did not settle on this modified string for similarity, it is worth noting that the similarity calculation is in need of some polishing in how it works especially when considering strings of vastly different size.

1.3.3 Recommendation:

If a user is slow at typing a word, then it can be assumed that they will also be slow at typing similar words. And so a typing system that can give words to practice on based on metrics of other words, could drastically improve your typing speed. Coupled with taking the popularity of words into account, this system could make learning how to type faster much easier.

1.3.4 Popularity:

the most popular word in the dataset is "the", but that does not mean that it is necessarily the most useful word to know. A less common word which is more similar to a lot of more common words should, in theory, be a more useful word to learn. Since learning how to type this word fast, will also help in typing similar words to it fast. In this case we find that the word "in" is actually the most useful word to know in the dataset, mainly because of its similarity to so many other words.

1.4 Word Rank:

Just like how webpages are ranked by the pagerank algorithm, the words in the dataset can be ranked like this as well. The algorithm is used to page nodes in a graph by the number and quality of the links that are pointing to them. as shown below:

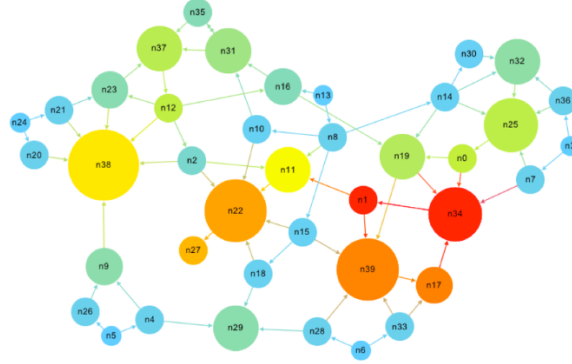


Figure 3: An image of a nodes being ranked by pagerank

With an adjacency list of words, the ranking algorithm takes into account the degree of similarity between two words along with the popularity of the word. So the weight given from node a to node b is given by *similarity* \times *popularity* where a higher weight means the nodes are more closely connected. So a **random surfer** model was developed to iterate over the graph and find accurate rates for the words.

1.5 Random Surfer

One way to find the ranking of the words, is to randomly surf the graph. Since by randomly traversing the graph space, a random surfer will naturally visit more popular destinations (in this case words) more often. However this model can lead to issues without some tweaks.

- **Dangling nodes:** A dangling node is a word which does not have any outgoing links to any other words in the graph. How should a random surfer tackle this issue?
- **Disconnected:** A graph might be disconnected or not very well connected and thus a random surfer might be stuck in only part of the whole graph giving unnecessarily high weights to nodes.

So fix this issue we have an α term, where $1 - \alpha$ represents the probability that a random surfer will teleport to a random node in the graph. For surfing this graph the α was kept at 0.85. This allows the whole graph to be suitably

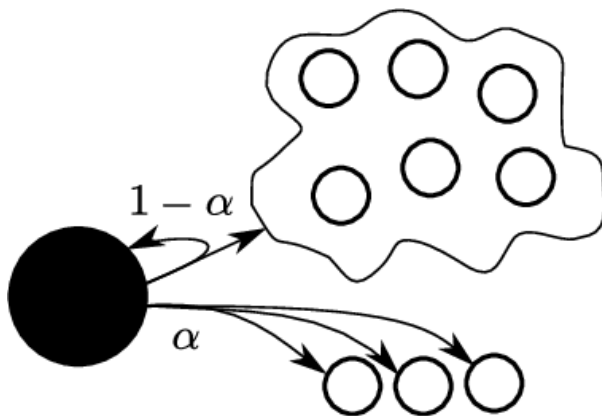


Figure 4: A random surfer choosing where to walk next

explored. As shown below:

With an α of 0.85, there is an 85% chance that the nodes will go one of the current nodes connections. But there is also a 15% chance that the node will go to any node in the network (including the current node and its connections).

1.5.1 Law of Large Numbers:

This algorithm might seem very chaotic. How could it possibly give accurate results about anything when so much of it is random?

The Law of Large Numbers states that as the iterations of this algorithm trend towards infinity. The sampled value equals the expected value. We can prove this using Chebyshev's inequality.

$$P(|\bar{X} - \mu| \geq \epsilon) \leq \frac{Var(\bar{X})}{\epsilon^2} = \frac{Var(X)}{n\epsilon^2} \quad (6)$$

And as n trends towards infinity the term trends to 0.

1.5.2 Results of Surfing:

After surfing for many iterations i.e one hundred billion. The word rank given by the surfer should very closely reflect the ranks of the words. Unlike with traditional Pagerank that was used on pages on websites, the landscape of this graph is quite different. Here is a list of the top twenty-five words found by the algorithm:

- 1 in,2996714
- 2 the,2081008
- 3 a,1821925


```

4  and,1362967
5  i,1255542
6  all,1197426
7  is,1136977
8  to,1123147
9  on,1027609
10 at,999468
11 an,996782
12 as,944560
13 or,906364
14 it,870291
15 for,761940
16 are,697984
17 that,670956
18 be,476380
19 e,449694
20 this,440784
21 re,395011
22 of,386790
23 see,382558
24 us,366126
25 s,332346
26 can,327143
27 will,326362
28 one,326130
29 he,303469
30 here,293988
31 ...
32 ...

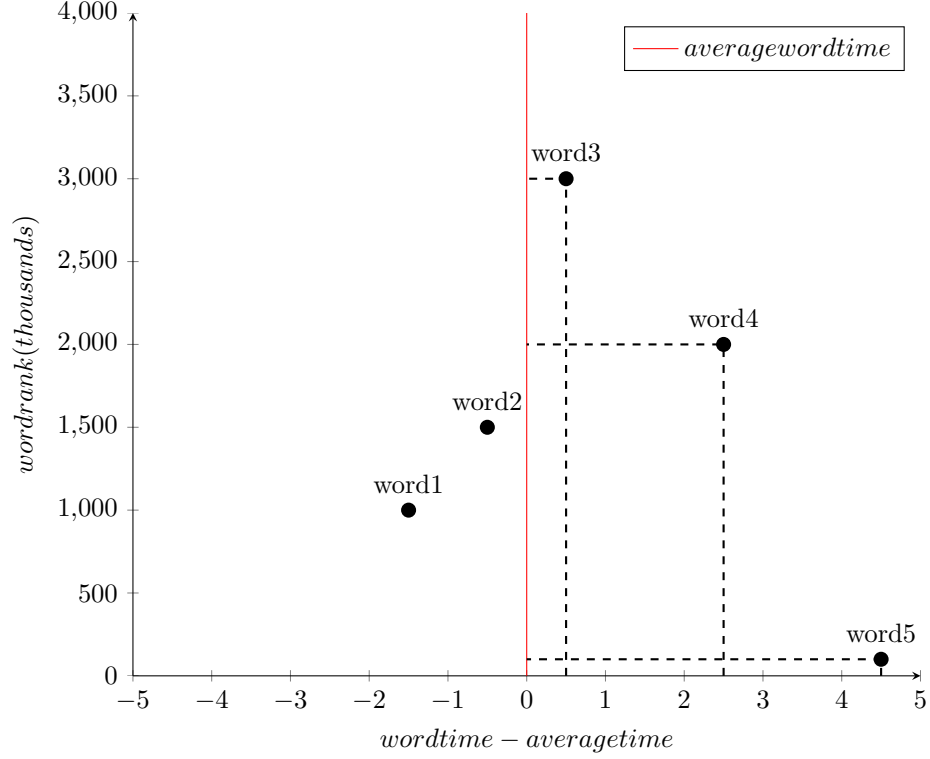
```

Even though in the original kaggle dataset 'the' was listed as the most popular word, after applying wordrank 'in' was regarded as the most popular word. This is other high ranking words such as 'line' and 'info' point to this word, where 'the' has lower ranking words. In total 977 words point to 'in', whereas only 230 point to 'the'.

1.6 Recommendation:

The system now has a ranking of the usefulness of each word through its wordrank as well as a graph containing the similarity of words. Based off this information we can identify the importance of a word based off its rank, if a user is typing an important word very slowly, the system should highly recommend that word and words similar to it for practice.

Similar to the word rank algorithm above. The Recommendation algorithm is going to randomly surf the graph. The goal is to find similar words to the previous under-performing words and recommend those word for the user the practice.



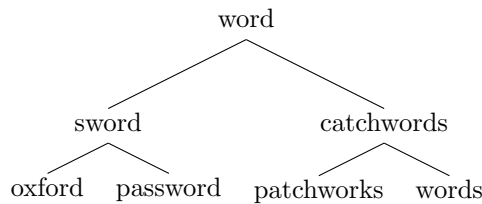
To get the slower words that have been typed we simply take the average of all the words that have been typed so far, and subtract that from that word's time. Clearly if this number is positive it means that the letters in the word were typed slower than average. The fast words such as word2 and word1 are ignored, since the algorithm is meant to focus on slower words. The code then examines the words that are slower than average and in this case looks at word3, word4 and word5. It then retrieves the word rank of each slower word to see how useful learning how to type this word is. The metric used to rank the inefficiency of learning a word is then calculated:

$$Inefficiency = (wordtime - averagetime) \times wordrank \quad (7)$$

In this case it is the area of the dashed box shown for each of the words. Meaning that word4 is the most inefficient word. All three words are then added to a sorted inefficiency list ranked from most inefficient to least inefficient. Once a list of inefficient words has been created, similar words to this inefficient word can be found in the similarity graph. for example as seen in the graph.csv file:

```
word,"(['words', 0.8), ('sword', 0.8), ('wordy', 0.8),
('worded', 0.7083333333333334), ('reword', 0.7083333333333334),
('swords', 0.6666666666666666) ... ('wordsmiths', 0.4),
('wordlessly', 0.4), ('catchwords', 0.4)]"
```

We take two random words from this list, and add them to the next paragraph iteration. So if a user under-performed on 'word', then 'swords' and 'catchwords' could be added to the next iteration.



The structure that is being made is essentially a binary search tree. As the user iterates over more and more paragraphs this tree will keep on expanding. Essentially honing in on the worst phrases for typing possible for that specific user, the user will have to type similar words to 'word' multiple times. Thus improving their typing of that word. However a problem with this is that it prevents new words from being explored in the typing system, and only constrains the user to a very specific subset of words.

1.6.1 Back to Alpha:

Just like in the word rank algorithm used above, we will use α here again, to make sure that new words are being added for the user to explore. The goal is to prevent the user from being cluttered with the same words over and over, and allow the system to accurately find the worst words in the entire dictionary for them to practice, this is similar to the 'Exploration vs Exploitation' problem in machine learning, except in this case it is human learning. With this alpha variable, the user only sees $(1 - \alpha) \times \text{paragraphsize}$ of similar words, and the rest are randomly selected based of their wordrank. This is another example of the system deliberately favouring more useful to know words over less useful ones. So what number should α be?

With some trial and error, I estimated alpha to be best at 0.75

1.6.2 Program Example:

The program begins by getting random words from the dictionary based on their wordrank. It is fairly obvious from looking at the paragraph that words with high word-ranks are there such as:

```

cleanliness table disposable exploiting wide none structural comments an parts pylon
mailing is questioned hangar contactable sea you free specific defining scorpions
force as fawning price documentary that cocky a

```

Figure 5: The start of the program

"that" 17th, "is" 7th, "free" 31st, "as" 12th, "to" 8th, "list" 83rd, "will" 27th,
 "an" 11th

but also uncommon words such as "contactable" and "pylon". The program will get roughly half of these nodes based on the user's typing performance and get 2 similar words for the next paragraph.

```

Welcome to the typing Program!

stabler disable manliness splines sables disposals e int the are photo no lubrication
baffle m spigot electronic contact really living using deal use did home discotheques
shout graphics top terms

```

Figure 6: The second paragraph

(also worth noting that the word list has not been Censored). Since "table" was the worst performing word here, the words "stabler" and "disable" are recommended by the system. In this example Shuffle was turned off, when generally it defaults to on. "cleanliness" was second least performing so "manliness" and "splines" are recommended and so on. So all these words appear in the next iteration i.e paragraph.

1.7 Testing:

How should the success of the algorithm be measured?

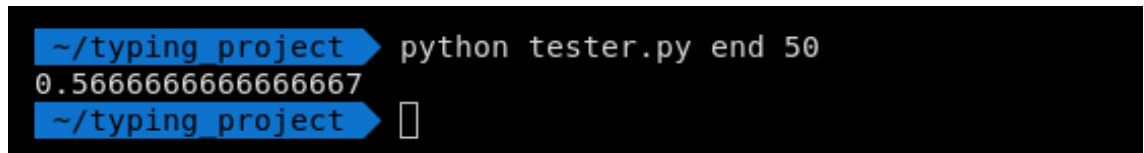
If the algorithm continually recommends popular and under-performing words for the user to type, then It is successful. Instead of manually testing the system based of paragraphs of words. I wrote a testing file to test how effective system is at recommendations. The file essentially passes in fake word data about how long a word has taken to be typed. For most words in the paragraph the time taken was in pythonic terms

```
Tw=random.random()+ random.random()+ random.random()
```

But the tester finds words with a certain substring e.g. "end" and applies a slightly different time value:

```
Tw=1.5+random.random()
```

The point is to simulate a user having trouble typing a specific substring. After simulating the typing in multiple paragraphs the testing function then returns the percentage of words in the final paragraph with that specific substring. Here is the result of the substring "end" on 50 iterations:

A terminal window with a black background and blue prompt characters. The first prompt is '~ /typing project' followed by the command 'python tester.py end 50'. The output is '0.5666666666666667'. The second prompt is '~ /typing project' followed by a cursor.

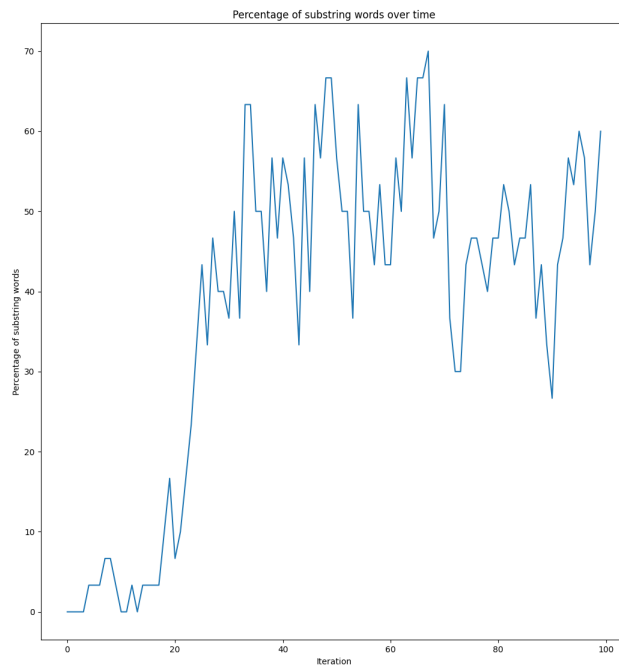
```
~/typing project ➤ python tester.py end 50  
0.5666666666666667  
~/typing project ➤
```

Figure 7: tester

This is a significant result that was repeatedly occurring when I ran the program. 57% of the final paragraph had the substring "end" in it, which is huge considering that only 0.83% of the data-set contains words with the substring "end". Keeping in mind this is with an alpha of 1, next we will explore the effects of changing the alpha on our tester function.

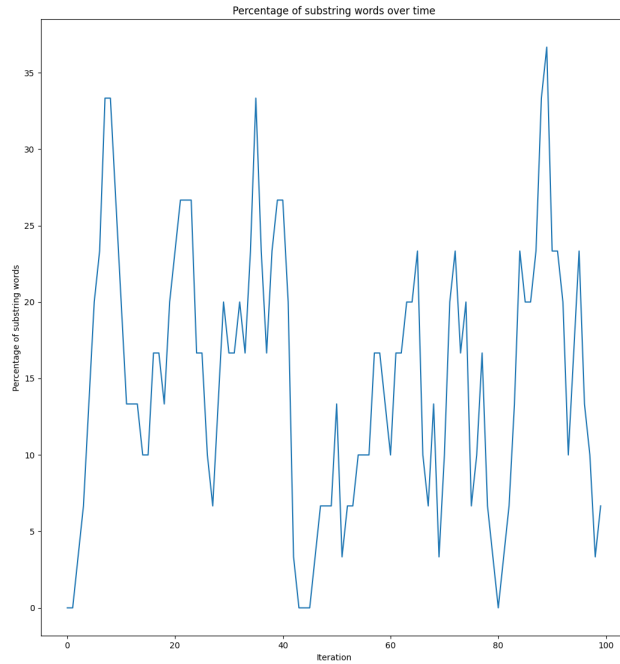
1.7.1 Testing alpha:

With a slight modification to the test file and the creation of a graphing file in python, we can show the percentage of current words with a substring in it. With alpha equal to one the result is:



alpha=1

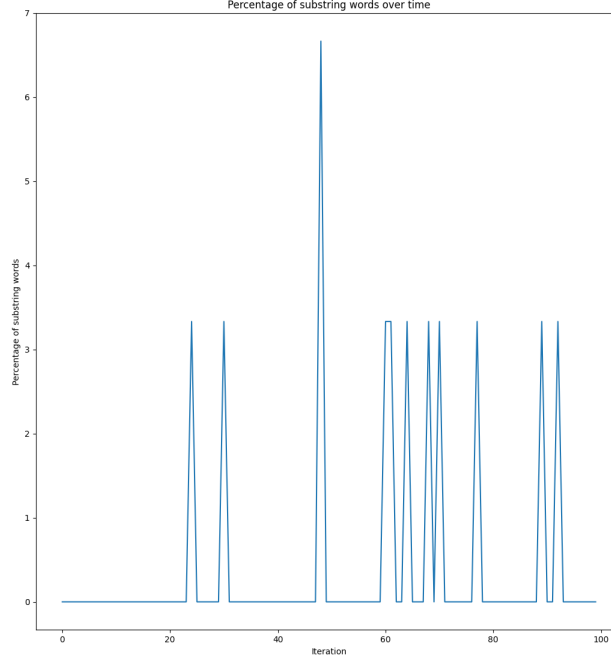
When we lower the alpha the graph changes:



$\alpha=0.85$

Unsurprisingly there is more fluctuation in the lower alpha graph, since more random nodes are added to the paragraph on each iteration. However an interesting thing to note is that the percentage increases quite quickly from the first iteration, this is because the random alpha words in the list allow for more exploration and thus it finds more words with the substring in it.

Considering that the area underneath the graph represents the amount of times a user types the substring, both are fairly effective at making a user type the substring, but the 0.85 alpha allows for more exploration and helps find inefficient words faster. As a comparison this is what the graph looks like when presented with random words ($\alpha = 0$).



alpha=0

The graph never reaches over the 10% mark at spends over 85% of its iterations at 0. Further pointing to the algorithm's success at recommending similar words. This graphing was done with multiple substrings yielding similar results. Because of these results I decided to kept the alpha at 0.85.

1.8 Measuring Typing Speed:

Going back to the first page: Consider a word w from a dictionary of length d , let s_w equal the speed required to type it, and p_w equal its popularity.

$$T_s = \sum_{k=1}^d \frac{s_k p_k}{total(p)} \quad (8)$$

p_w here can be pretty clearly defined as the wordrank of w . With this we can calculate the effective speed of a paragraph (if the system has the wordrank of a word along with the time taken to type it). With this in mind the system now has an EPWM metric.

2 Code:

To run the code just type "python start_program.py" Here is some code used in the assignment: (Note: I only included the code that implemented the features talked about above) files not included are (tester.py, start_program.py, prune_data.py, graph_test.py, analyze_words.py)

2.1 word_rank.py

this function creates a file called word_rank.csv, full of the word ranks from the word list.

```
1 #This is very similar to the pagerank algorithm and it follows a  
2 random walk to find the ranks  
3 #of each word in the top words freq file.  
4 #In the situation of typing: a word's rank should be given by its  
5 own rank + (similar_words_rank*similarity)  
6 import csv  
7 import random  
8 def random_surf(iterations ,alpha):  
9     #takes a while since the file size is over 300MiB  
10    word_graph = load_graph_file_to_memory()  
11    #word_list contains the amount of iterations spent on a  
12    particular word  
13    word_list = load_list_file_to_memory()  
14    popularity = load_initial_popularity_from_memory()  
15  
16    top_popularity = popularity['the'][0]  
17    current_word = random.choice(word_list)  
18    #this is the model starting its random surf  
19    #we dont worry about dangling nodes in this model, since there  
20    are none  
21    for iteration in range(0,iterations):  
22        popularity[current_word][1]+=1  
23        if iteration % 10000 == 0:  
24            print(str(int(10000*iteration/iterations)/100) + "%")  
25            if random.random() > alpha:  
26                #we find a random node and start surfing from that  
27                current_word = random.choice(word_list)  
28                #increment the value of the word  
29                else:  
30                    #we traverse to a connected node  
31                    current_word_edges = word_graph[current_word]  
32  
33                    #this next term represents the weights that we put on  
34                    this node  
35                    # popularity and similarity  
36                    edge_weights = []  
37                    word_weight = 0  
38                    new_current_node = None  
39                    if len(current_word_edges) == 0:  
40                        #dangling node  
41                        current_word = random.choice(word_list)  
42                    else:  
43                        for edge in current_word_edges:
```

```

39         word_weight += edge[1] * popularity[edge
40             [0]][0] / top_popularity
41         edge_weights.append(word_weight)
42
43         rand = random.random() * word_weight
44         for weight in range(0, len(edge_weights)):
45             if rand < edge_weights[weight]:
46                 new_current_node = current_word_edges[
47                     weight][0]
48                 current_word = new_current_node
49                 break
50
51     make_csv(popularity, word_list)
52 def load_list_file_to_memory():
53     list_file = open('csv_files/top_words_freq.csv', 'r')
54     nodes = []
55     for row in csv.reader(list_file):
56         nodes.append(row[0])
57     list_file.close()
58     return nodes
59
60 def load_initial_popularity_from_memory():
61     list_file = open('csv_files/top_words_freq.csv', 'r')
62     words = {}
63     for row in csv.reader(list_file):
64         words[row[0]] = [int(row[1]), 0]
65     list_file.close()
66     return words
67
68 def load_graph_file_to_memory():
69     file = open('csv_files/graph.csv', 'r')
70     graph = {}
71     for row in csv.reader(file):
72         key, edge_list = convert_string_to_edge_list(row)
73         graph[key] = edge_list
74     file.close()
75     return graph
76 def convert_string_to_edge_list(string):
77     name = string[0]
78     result = string[1]
79     result = result.strip(' ')[(' ').split(' '), (' ')]
80     list_tuple = []
81     if result[0] == '':
82         return name, list_tuple
83     for i in result:
84         vals = i.strip('\n').split('\n', ' ')
85         list_tuple.append((vals[0], float(vals[1])))
86     return name, list_tuple
87
88 def make_csv(popularity, word_list):
89     file = open("csv_files/word_rank.csv", 'w')
90     writer = csv.writer(file)
91     sorted_list = []
92     for word in word_list:
93         sorted_list.append([word, popularity[word][1]])

```

```

94     sorted_list.sort(key=get_pop, reverse=True)
95     for word in sorted_list:
96         writer.writerow((word[0], word[1]))
97     file.close()
98 def get_pop(word):
99     return word[1]
100 if __name__ == '__main__':
101     random_surf(100000000, 0.85)

```

2.2 word_similarity.py

Calculates the similarity of 2 given strings:

```

1  #finds the similarity between 2 words (
2  # similarity in terms of typing and not in terms of meaning)
3  import math
4
5  def similarity(string1, string2):
6  #for this convolution to work we must first add padding to one of
   the strings
7  #runtime O(n^2)
8      similarity = 0
9      #get string lengths
10     string_length1 = len(string1)
11     string_length2 = len(string2)
12     string_length = 0
13     padded_string = ""
14     non_padded_string = ""
15     divisor = string_length2 * string_length1
16     #below variables are being initialized depending on which
       string is bigger
17     if string_length1 > string_length2:
18         #divisor = string_length1 * (string_length2 + 1) / 2
19         string_length = string_length1 + 2 * (string_length2 - 1)
20         padded_string = string1.center(string_length)
21         non_padded_string = string2
22     else:
23         #divisor = string_length2 * (string_length1 + 1) / 2
24         string_length = string_length2 + 1 * (string_length1 - 1)
25         padded_string = string2.center(string_length)
26         non_padded_string = string1
27     for i in range(0, len(padded_string)):
28         similarity += recurse_string(i, 0, padded_string,
                                       non_padded_string, 0)
29
30     #since padded_string > non_padded_string we divide by the
       maximum value padded_string can have
31     # which is the triangle number of string length
32     similarity = similarity / divisor
33     return similarity
34
35 def recurse_string(i, j, string1, string2, val):
36     if i >= len(string1) or j >= len(string2):
37         return 0
38     if string1[i] != string2[j]:
39

```

```

40         return recurse_string(i+1, j+1, string1, string2, 0)
41     else:
42         return 1 + 2*val + recurse_string(i+1, j+1, string1,
            string2, val+1)

```

2.3 create_similarity_graph.py

This code was only ran to create the similarity graph which is subsequently stored in a csv file for easy access.

```

1  from word_similarity import similarity
2  import csv
3  def create_graph():
4      word_freq = open("csv_files/top_words_freq.csv", "r")
5      reader = csv.reader(word_freq)
6      word_list = []
7      for row in reader:
8          word_list.append((row[0], row[1]))
9      word_freq.close()
10     graph = {}
11     flag = 0
12     for i in word_list:
13         graph[i[0]] = list()
14         flag+=1
15         if flag %10 == 0:
16             print(flag)
17         for j in word_list:
18             if i[0] != j[0]:
19                 add_to_list(200, graph[i[0]], j[0], similarity(i
                    [0], j[0]))
20     output = open("csv_files/graph.csv", "w")
21     writer = csv.writer(output)
22     for key, value in graph.items():
23         print(key, value)
24         writer.writerow((key, value))
25     output.close()
26     #determines whether to add element to the adjacency list
27     def add_to_list(length, list, element, similarity):
28         add_sorted(element, list, similarity)
29
30     def add_sorted(element, list, similarity):
31         if similarity >= 0.4:
32             for i in range(0, len(list)):
33                 if float(list[i][1]) < similarity:
34                     list.insert(i, (element, similarity))
35             return
36             elif i == len(list) -1:
37                 list.append((element, similarity))
38             return
39         if len(list) == 0:
40             list.insert(-1, (element, similarity))
41     if __name__ == '__main__':
42         create_graph()

```

2.4 recommend_words.py

Firstly the similarity graph must be made, and then the work rank file. After this recommend words contains the logic for recommending words for the next paragraph.

```
1 import numpy
2 import csv
3 import os
4 import random
5 import time
6 import math
7 # the class that handles the recommendation logic for words.
8 class RecommendWords:
9     hash_table = {}
10    word_ranks = []
11    rank_lookup = {}
12    rank_total = 0
13    def __init__(self):
14        #create a hash table of all our words so we can retrieve
15        them from the file
16        #in constant time
17
18        rankfile = open("csv_files/word_rank.csv","r")
19        rankfile_reader = csv.reader(rankfile)
20
21        for row in rankfile_reader:
22            self.rank_total += int(row[1])
23            self.word_ranks.append((row[0], self.rank_total))
24            self.rank_lookup[row[0]] = int(row[1])
25        rankfile.close()
26
27        wordfile = open("csv_files/top_words_freq.csv","r")
28        wordfile_reader = csv.reader(wordfile)
29        graphfile = open("csv_files/graph.csv")
30        row = graphfile.readline()
31        offset = 0
32        while row:
33            wordrow = next(wordfile_reader)
34            self.hash_table[wordrow[0]] = offset
35            offset = graphfile.tell()
36            row = graphfile.readline()
37
38        wordfile.close()
39        graphfile.close()
40
41    def convert_string_to_list(self, string):
42        result = string.strip(' ')[(' ').split(' '), (' ')]
43        list_tuple = []
44        for i in result:
45            vals = i.strip('\'').split('\', ' ')
46            list_tuple.append((vals[0], float(vals[1])))
47        return list_tuple
48
49    def get_word_edges(self, word):
50        file = open("csv_files/graph.csv")
```

```

51         offset = self.hash_table[word]
52         graphfile = open("csv_files/graph.csv")
53         reader = csv.reader(graphfile)
54         graphfile.seek(offset,0)
55         edge_string = next(reader)[1]
56         return self.convert_string_to_list(edge_string)
57
58     def get_paragraph(self, word_metrics, alpha, number_of_words):
59         #contains a list with item [wordname, average_word_time]
60         paragraph = []
61         #effective words per minute
62         ewpm = 0
63         if len(word_metrics) == 0:
64             for _ in range(0, number_of_words):
65                 paragraph.append(self.get_weighted_random_word())
66         else:
67             word_time_list, average = word_metrics
68             slow_word_list, ewpm = self.get_inefficiency_list(
69                 word_time_list, average)
70             #worst_percentage is the length of the list that we
71             will iterate over
72             #
73             worst_percentage = min(math.floor(alpha *
74                 number_of_words/ 2), len(slow_word_list))
75             for i in range(0, worst_percentage):
76                 #for these underperforming nodes we want to get 2
77                 similar nodes from the wordlist and add them to
78                 the next paragraph
79                 edges = self.get_word_edges(slow_word_list[i][0])
80                 paragraph.append(self.get_random_edge(edges))
81                 paragraph.append(self.get_random_edge(edges))
82
83                 random_words_to_add_to_paragraph = number_of_words -
84                     worst_percentage*2
85             for _ in range(0, random_words_to_add_to_paragraph):
86                 paragraph.append(self.get_weighted_random_word())
87
88             #random.shuffle(paragraph)
89             return paragraph, ewpm
90
91     def get_random_edge(self, edges):
92         rand_weight = random.random()
93         potential_edge = random.choice(edges)
94         while rand_weight > potential_edge[1]:
95             potential_edge = random.choice(edges)
96             rand_weight = random.random()
97
98         return potential_edge[0]
99
100     def get_inefficiency(self, inefficiency):
101         return inefficiency[1]
102     #returns a tuple (word, inefficiency)
103     def get_inefficiency_list(self, word_time_list, average):
104         index = 0
105         slow_word_list = []
106         ewpm = 0
107         total_rank = 0
108         while index < len(word_time_list):

```

```

102         #the user typed this word more slowly than the average
           word they wrote
103         slow_word = word_time_list[index][0]
104         # essentially how slow did we type this word * how
           useful is it to type fast
105         rank_look = self.rank_lookup[slow_word]
106         ewpm += rank_look / word_time_list[index][1][0]
107         total_rank += rank_look
108         inefficiency_scale = (word_time_list[index][1][0] -
           average) * rank_look
109         if word_time_list[index][1][0] > average:
110             slow_word_list.append((slow_word, inefficiency_scale)
           )
111
112         index+=1
113         slow_word_list.sort(key=self.get_inefficiency, reverse=True)
114         ewpm = ewpm * 60 / (5*total_rank )
115         return slow_word_list, ewpm
116
117     def get_weighted_random_word(self):
118         random_word_num = random.random() * self.rank_total
119         for i in self.word_ranks:
120             if i[1] > random_word_num:
121                 return i[0]

```

3 Conclusion

3.1 Potential Improvements

- the user interface is very barebones and uses the terminal control library curses. This mainly because the UI was not the focus of the project. This could be improved to a client server model running on a webpage.
- currently all the data is stored in csv_files, it would be more optimal to store it in databases such as PostgreSQL for word_rank.csv, word_list.csv and similarity_graph.csv
- One additional feature that I thought about implementing but did not actually add, is some sort of key positional graph that could be used to modify the similarity function of two strings. For example "p" and "o" share a similar location on the keyboard, so therefore should be treated more similarly than "p" and "z" for example. I ended up not implementing this, since I'm on the fence about if it would add unnecessary complexity to the algorithm and its implementation might be shaky.

3.2 What I learnt

- I learnt a lot about string similarity theory in the process of doing this assignment Hamming distance and Levenstein distance we only a few metrics I looked at. It is ironic that the lack of a suitable algorithm for my purposes forced me to learn a lot on the subject.

- How to implement a pagerank like algorithm was interesting, and fairly since the random surfer will accurately find the ranks just from increasing iterations as shown by the law of large numbers.
- a stronger understanding of how the α variable works in a pagerank algorithm and the trade-off from having the α too high or too low.
- This was very different from anything I have written before. A lot of the implementation was trial and error to see what worked and what did not. I think I made 10+ graph.csv files containing different weights to see what worked with the wordrank algorithm.
- Increased my EWPM from 45 to 55 over the course of this assignment, partially from typing in the program but also from the typing of this report.

References

- [1] Ignjatovic, A., 2021. DFT, DCT and convolution. [online] Cse.unsw.edu.au. Available at: http://www.cse.unsw.edu.au/cs4121/lectures_2019/DFT_DCT_CONV_short.pdf [Accessed 15 November 2021].
- [2] Ignjatovic, A., 2021. Google PageRank and Markov Chains. [online] Cse.unsw.edu.au. Available at: http://www.cse.unsw.edu.au/cs4121/lectures_2019/pagerank_slides_short.pdf [Accessed 11 November 2021].
- [3] Ignjatovic, A., 2021. Recommender Systems. [online] Cse.unsw.edu.au. Available at: http://www.cse.unsw.edu.au/cs4121/lectures_2019/recommender_systems_short.pdf [Accessed 11 November 2021].
- [4] Wu, G., 2021. String Similarity Metrics – Edit Distance. [online] Available at: <https://www.baeldung.com/cs/string-similarity-edit-distance> [Accessed 8 November 2021].
- [5] Goel, A., 2021. MSE 233 Lecture 8: Applications of PageRank to Recommendation Systems. [online] Web.stanford.edu. Available at: <https://web.stanford.edu/class/msande233/handouts/lecture8.pdf> [Accessed 12 November 2021].