CS111 F21 Homework 5

Michael Glushchenko, 9403890

October 30, 2021

1 Singular Value Decomposition of Languages

1.1 Recreating Moler's Chart

To recreate Moler's chart, we load our text into a list, getting rid of spaces and punctuation, and capitalizing all letters in the process:

```
with open('./Homework/h05/gettysburg.txt') as file:
    raw_text = file.read()
letters = []
for c in raw_text.upper():
    if c.isalpha():
        letters.append(c)
print(letters[:8], '...', letters[-8:])
```

```
['F', 'O', 'U', 'R', 'S', 'C', 'O', 'R'] ... ['T', 'H', 'E', ' E ',' A ',' R ',' T ',' H ']
```

We go on to make the frequency matrix based on the Gettysburg Address text:

```
# count the number of occurances of each pair of characters
A = np.zeros((26,26))
for i in range(len(letters) - 1):
    A[ord(letters[i]) - ord('A'), ord(letters[i + 1]) - ord('A')] += 1
# annnd the last letter is followed by the first
A[ord(letters[-1]) - ord('A'), ord(letters[0]) - ord('A')] += 1
```

The next step is to compute the SVD of the frequency matrix A, check that the row sums are equal to the column sums (correctness check), and compare the individual singular values to the ones Moler got in his paper:

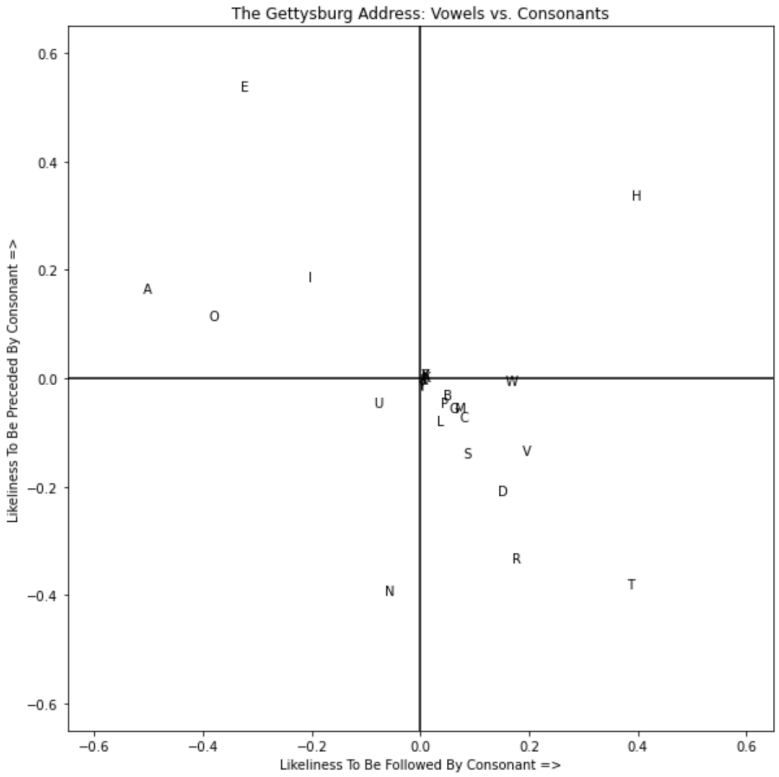
```
print((np.sum(A, axis = 0) == np.sum(A, axis = 1)).all())
# compare our singular values with Moler's research paper
for i in range(len(s)):
    print("s[", i, "]:\n", np.round(s[i], 4))
```

```
True
s[ 0 ]:
 81.7256
s[ 1 ]:
 53.5189
s[ 2 ]:
 45.1604
```

Then, we compute the SVD of A, and use the following code to make the desired

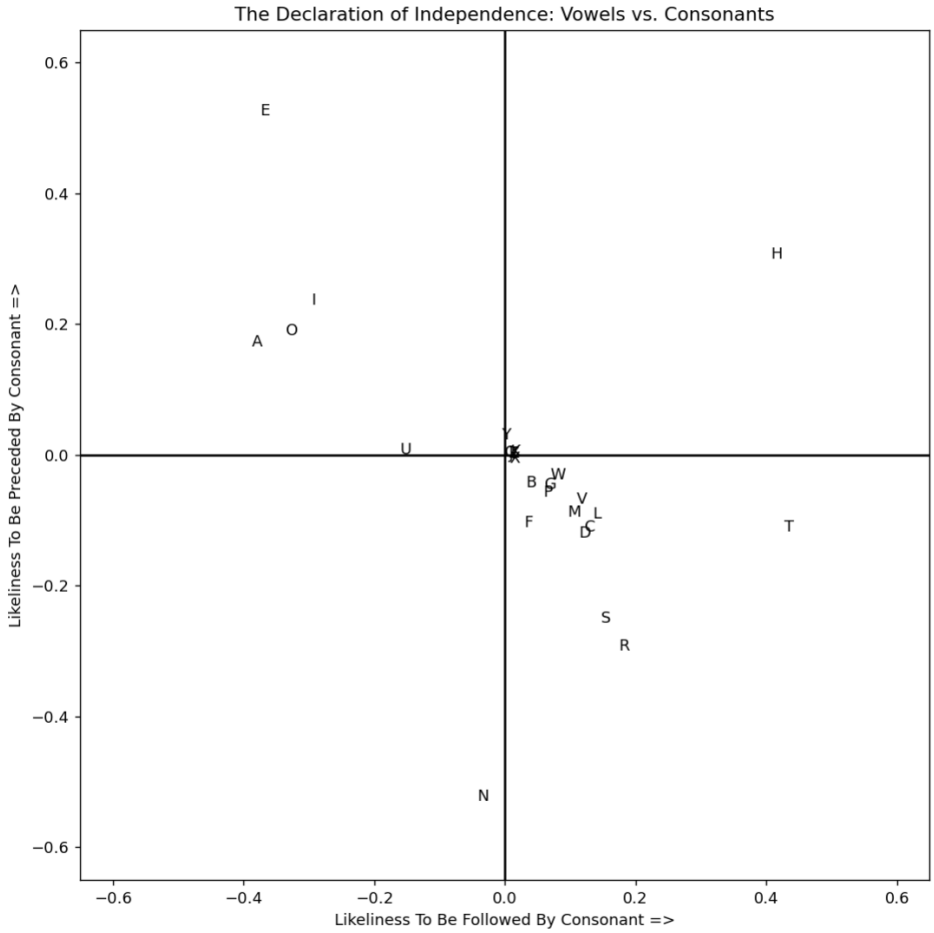
plot for the Gettysburg Address (the screenshot has the title commented out, but it was used when making the plot):

```
x = U[:,1]
y = Vt[1,:]

plt.figure(figsize = (10, 10)) # change size to large
plt.axis('square')

# zoom in and out
limits = (-.65, .65)
plt.xlim(limits)
plt.ylim(limits)

# plot each character in the alphabet
for i in range(26):
    plt.text(x[i], y[i], chr(i + ord('A')))

# x axis and y axis in black
plt.plot(limits, (0, 0), 'black')
plt.plot((0, 0), limits, 'black')

# title and axis labels
# plt.title('The Gettysburg Address: Vowels vs. Consonants')
# plt.title('The Declaration of Independence: Vowels vs. Consonants')
plt.title('Ruslan and Lyudmila: Vowels vs. Consonants')
plt.xlabel('Likeliness To Be Followed By Consonant =>')
plt.ylabel('Likeliness To Be Preceded By Consonant =>')
```



1.2 Attempt on a Longer Text

Without many changes to the code, since the language used is the same, we load the Declaration of Independence into the *letters* list. Running the same code from section **1.1**, we now obtain the following plot:



The overall picture looks very similar to that of the output that the Gettysburg Address produced. A noticeable difference is that the Declaration of Independence text classifies the letter U as a vowel, along with A, O, E, and I, whereas the other text's analysis puts U in its own category, separate from the vowels. Letter Y is in the center of the chart for both texts; letters H and N seem to behave the same. All consonants fall into the same quadrant both times, with obvious minor discrepancies on the exact coordinates of each letter. Since this is a longer text, however, we see some letters being more defined in the analysis of the second text; S is a good example of this – it's close to the middle cluster of consonants in **1.1**, but a clearly-defined consonant in **1.2**, closer to R.

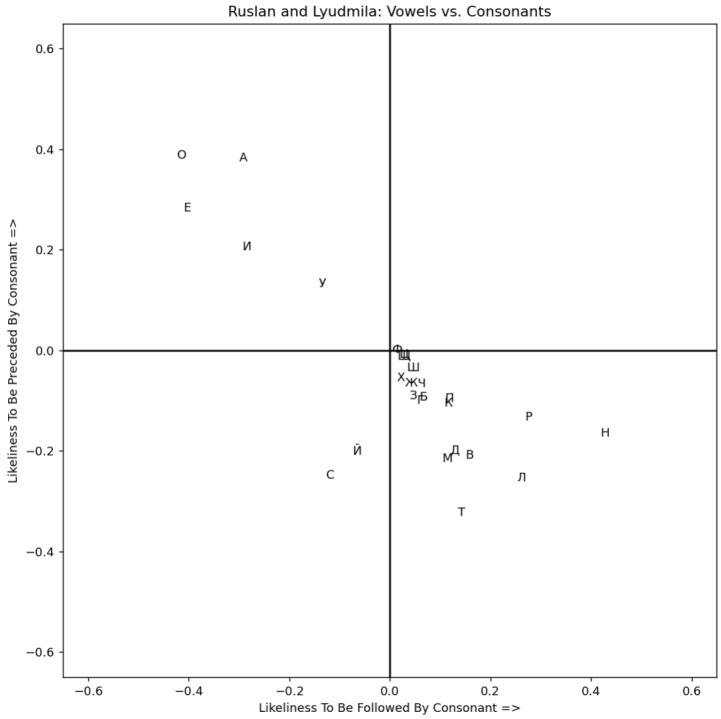
1.3 Analysis On a Different Language

In this section, we attempt to run the same sort of analysis on the Russian language, choosing Pushkin's 'Ruslan and Ludmila' as the our sample text. Since we are now analyzing the Russian language, we want to use the following method for indexing each letter:

```
ord('Ш') - ord('А') # to decode russian characters the same way we decoded english
```

```
24
```

meaning we substract the unicode of the Russian A, not the English A. After changing n to 33, since there are 33 letters in the Russian language, we run the same code we ran in **1.1** and **1.2**. This was the resulting output:



The overall structure of where letters fall seems to be the same here, meaning Russian probably **is** a vowel-follows-consonant language, as English is described by Moler. Furthermore, we can see the upper left quadrant indeed holds the vowels of the Russian language, with the exception of the letters "yu" and "ya," where the difference is actually also displayable in their translation to English requiring two English letters. We can see the consonants are mostly grouped in the lower right quadrant.

The key difference I see is that Russian has more letters bunched up closer to the origin, and more letters in that cluster of consonants slightly below-and-right of the origin.

2 Floating Points

2.1 Loop 1

Here's the code used for this loop:

```
x = 1.0
counter = 0
while (1.0 + x > 1.0):
    x /= 2.0
    cs111.print_float64(x)
    counter += 1
print("\nNum of iterations before halt: ", counter)
```

```
sign      : 0 means +
exponent  : 3cc means 972 - 1023 = -51
mantissa  : 1.0000000000000000000000000000000000000000000000000000

input     : 2.220446049250313e-16
as float64: 2.2204460492503131e-16
as hex    : 3cb0000000000000
sign      : 0 means +
exponent  : 3cb means 971 - 1023 = -52
mantissa  : 1.0000000000000000000000000000000000000000000000000000

input     : 1.1102230246251565e-16
as float64: 1.1102230246251565e-16
as hex    : 3ca0000000000000
sign      : 0 means +
exponent  : 3ca means 970 - 1023 = -53
mantissa  : 1.0000000000000000000000000000000000000000000000000000


Num of iterations before halt:  53
```

As you can see, it does 53 iterations before halting. The second to last input we use, i.e. machine ϵ, is $2.220446049250313e{-}16$ in decimal, and $0x3cb0000000000000$ in hex. The last input in decimal is $1.1102230246251565e{-}16$, and $0x3ca0000000000000$ in hex, this is when the exponent reaches -53, and we don't have anymore bits of precision to represent x in such a way such that $1 + x > 1$. The property displayed is **machine** ϵ, the smallest floating point such that python still distinguishes $1 + x$ as being > 1.

2.2 Loop 2

Here's the code used for this loop:

```
x = 1.0
counter = 0
while x + x > x:
    x = 2.0 * x
    cs111.print_float64(x)
    counter += 1
print("\nNum of iterations before halt: ", counter)
```

```
as hex    : 7fd0000000000000
sign      : 0 means +
exponent  : 7fd means 2045 - 1023 = 1022
mantissa  : 1.0000000000000000000000000000000000000000000000000000

input     : 8.98846567431158e+307
as float64: 8.988465674311579e+307
as hex    : 7fe0000000000000
sign      : 0 means +
exponent  : 7fe means 2046 - 1023 = 1023
mantissa  : 1.0000000000000000000000000000000000000000000000000000

input     : inf
as float64: inf
as hex    : 7ff0000000000000
sign      : 0 means +
exponent  : 7ff means inf or nan


Num of iterations before halt:  1024
```

Evidently, we do 1024 iterations before the loop halts. The second-to-last input in decimal is $8.98846567431158e + 307$, which is $0x7fe0000000000000$ in hex; here, our mantissa is 1 with a bunch of 0s after the binary period, and our exponent of the base is 1023. The input after that, the last input, is ∞, written $0x7ff0000000000000$ in hex. The property displayed here is the largest number less than infinity that is able to be represented in IEEE Standard.

2.3 Loop 3

Here's the code used for the final loop:

```
x = 1.0
counter = 0
while x + x > x:
    x = x / 2.0
    cs111.print_float64(x)
    counter += 1
print("\nNum of iterations before halt: ", counter)
```

```
input     : 1e-323
as float64: 9.8813129168249309e-324
as hex    : 0000000000000002
sign      : 0 means +
exponent  : 000 means zero or denormal

input     : 5e-324
as float64: 4.9406564584124654e-324
as hex    : 0000000000000001
sign      : 0 means +
exponent  : 000 means zero or denormal

input     : 0.0
as float64: 0.0000000000000000e+00
as hex    : 0000000000000000
sign      : 0 means +
exponent  : 000 means zero or denormal


Num of iterations before halt:  1075
```

Finally, we have the property of the smallest number that's still greater than 0 that Python is able to represent in IEEE Standard. This time, the loop runs 1075 times before halting. The second-to-last input used is $4.9406564584124654e{-}324$ in decimal and $0x0000000000000001$ in hex. The next input, the last one, our hex representation of the floating point goes to $0x0000000000000000$, thus representing the actual number 0 (0.0 in decimal). This property shows the floating-point representation for when the exponent of our base reaches 0, but we still have a non-zero fraction, thus resulting in a denormal number, (for the last 53 iterations) until we reach 0 because we ran out of precision *bits* in the fraction part of the floating-point representation.