CS155 Group Project 3 Report

Halle Blend, Georgia Malueg, Rachael Kim, Jayden Nyamiaka

March 2023

1 Introduction

Used 1 Late Hour

1. **Team Name:**
   Everybody Likes Jayden
2. **Group Members:**
   Halle Blend, Georgia Malueg, Rachael Kim, Jayden Nyamiaka
3. **Colab Link:** Code
4. **Piazza Link:** Poem Submission
5. **Division of Labor:**
   Halle: Data Pre-Processing, Unsupervised Learning, Poem Generation for HMM, and implementation of rhyming /counting syllables.
   Georgia: Visualizations and Interpretation (Code and Analysis), Additional feature 3 (Additional text - Spencer)
   Rachael: Additional feature 1 (Implementing Haiku, collecting example Haiku poems), Additional feature 2 (Implementing rhyming sonnets - debugged Halle's rhyming sample sentences code, wrote rhyme dict gen and generate rhyming sonnet)
   Jayden: Visualizations and Interpretation (Analysis), Additional feature 1 (Collecting example Haiku poems, feature analysis), Helped with Additional feature 3

2 Pre-Processing

Hidden Markov Models

We kept most of our pre-processing of the provided data file, *shakespeare.txt* simple for our HMM implementation in accordance with the implementation from HW6. We based the function *sonnet_parse_observations* off of the function *parse_observations* from HW6. The difference between the two functions is that in *sonnet_parse_observations*, we filter out lines of size ≤ 1 to remove the numbered headers and the empty lines because these would not contribute to our modeling in a positive way. Thus, each line was split into words where the words were now lower-case and punctuation/excess characters were removed. After examining *shakespeare.txt*, we decided to exclude all punctuation and other various special characters because we did not think it would impact our ability to reconstruct sonnets similar to Shakespeare's voice. We also decided to remove all capitalization in order to simplify and standardize the sonnet. The only exception we made was to capitalize the beginning of each sentence and certain words such as I and O in order to keep some sense of a proper sonnet format. The only pre-processing decision we did not implement was that we did not split hyphenated words. Thus, our poems could potentially have words that used to be hyphenated but are now merged together without punctuation.

We also did additional pre-processing with the provided data file, *syllable_dictionary.txt* in order to properly count the number of syllables. We did this by creating two dictionaries to track syllable counts. To properly match the words, the words in the syllable dictionary were processed the same way we removed capitalization and any punctuation and special characters as above. We created two dictionaries with these processed words where one dictionary maps the word to its syllable count, and the other dictionary maps the observation number of the word (from *obs_map*) to the syllable count. We implemented it this way because the first dictionary, *syllab_dict*, was used to print out the number of syllables in each line while the other dictionary, *syllab_dict_2*, was used in the function *sonnet_generate_emission* to generate emissions based on the number of syllables. We also note that we kept a one-to-one mapping of a word to its number of syllables throughout this process. For cases where words had end-of-line syllable counts in addition to their real syllable counts, we randomly selected the syllable count that appeared first and just disregarded the other count. We did this to maintain simplicity and efficiency with the other functions. In addition, there were just a few words that had more than one syllable count, so this did not have a major effect on our poem generation.

Next, we looked at *shakespeare.txt* again in order to introduce rhyme schemes to our poems. Similar to above, we created two dictionaries to track which words rhyme with other words. To properly match the words, the words in the rhyme dictionary were processed the same way as we mentioned above where the capitalization, punctuation, and special characters were removed. We created two dictionaries with these processed words where one dictionary maps from the word (string) to a list of words (strings), and the other dictionary maps from the observation number of the word (from *obs_map*) to a list of words (strings). We implemented it this way because the first dictionary, *rhyme_dict*, was used to compare between rhyming words for debugging and direct string word access while the other dictionary, *rhyme_dict_2*, was used in the function *rhyme_generate_emission* to generate emissions with a required rhyming word. We also note that when generating these dictionaries, we checked for a desired number of rhyming words when we wanted to create a certain rhyming scheme. Thus, we always made sure we had enough rhyming words for each scheme. This is implemented in both functions *rhyme_generate_emission* and *rhyme_sample_sentences*. We used dictionaries for rhyming as described above in order to easily access a list of words that rhyme with a given word (that wasn't the word itself).

3 Unsupervised Learning

As directed in the project description, we used the Baum-Welch algorithm that we implemented in HW6. As we looked at in HW6, we suspected that increasing the number of hidden states would give us phrases that made better sense. Thus, we experimented with the number of hidden states. We varied the number of hidden states between 1 and 15. As we expected, the models with more hidden states seemed to produce poems that sounded better. Therefore, to test we used smaller numbers of hidden states for a quicker run time. However, we mainly produced poems with models with 15 hidden states. Also, we mainly used 100 epochs. Due to run-time efficiency, we only considered models with at most 15 hidden states.

4 Poem Generation 1: Hidden Markov Models

We will describe our process for Naive Poem Generation using Hidden Markov Models (HMM) for a 14-line sonnet without altering exact syllable counts, rhymes, and other additional implementations. We followed a similar process as in HW6. We used *obs* created from our function *sonnet_parse_observations* based off of the function *parse_observations* from HW6 to train our unsupervised HMM. We used the function *generate_naive_sonnet* to generate a sonnet from the trained HMM. The function generates 14 lines of poetry line by line using the function *sample_sentence* from HW6 while also properly splitting it into quartains and couplets. Simply, *sample_sentence* was called 14 times in the naive poem generation process. Also, we counted the syllables in each line after generating the sentences, capitalizing words such as I and O. In the *sample_sentence* function, we retrieved the words from the list of emissions made from the function *generate_emission* from HW6. This function accounts for the number of words instead of the number of syllables for each line, so we tried to get as close to 10 syllables per line as possible by randomly choosing between 7 or 8 words per line. Also, we note that we implemented the proper syllable count as part of the Additional Goals section that will be discussed below.
Below are the poems that were naively generated using our functions with hidden states of 1, 10, and 15, respectively.

```
Palate other of the got needs my thou
Frown leave image my be main taste your
Burn cheer livery gored his dead and each
Ere when or your with is memory more

About rich add not times your my beauteous
Loves seeing swear how answer others shall such
It to is ripe was natures the thy
Whose shun of it from and suborned glass

Most to the vermilion breath is perceive worst
Eyes that in you basest of I worth
My rest call me absent thy beauty me
Terms amis haste make flower where think so

  I four strong brain seen and besides lie
  And oblivion divine purity be with and your
```

```
Clouds to sooner alive live such crime
Beautys old that usest maturity are nature
Others love to gilded lives increase on
My I night fire with would divert

No my self yet of and for
More image by farthest kindhearted far black
Fall you now against of this most
Dwells or before for my firm those

No desire thou woo eyes in you
Self of at their the as the
Thy numbers grief that that thou hymns
When for despite bounty sight proud him

  How my parts were my sun to
  Who my can eyes thee minutes some
```

```
Thy a where whence as no and
Thee more lie and water before him
Self the sure and day with had
Admit female it this and loves doth

Writes praise it be nothing eternal both
Woe looks read under I can thee
Bright ye here chest to this suspect
Fulness love made me indeed not but

Cry thou see give bark my my
Do you a growth the are abuse
Angel and my sweets fire she it
With now their and the map forth

  Dead peace this fortunes so bereft the
  Am my that ill best in the
```

After examining these poems, we see that they do not make much sense. Unlike a proper Shakespearean sonnet, there is poor rhythm and no rhyming pattern. The syllable count displayed is close to 10 syllables per line. This happens because there are 7 or 8 words per line. This helps to keep partial structure and rhythm but lacks consistency. However, there is some expression of Shakespeare's voice in the sense that the words are not very modern. Although the poems do not make a lot of sense, they still have remnants of Shakespeare's voice. Thus, it seems that increasing the number of hidden states up to 15 helps generate lines of poetry that are more realistic and make more sense where the seem to resemble lines from Shakespeare's sonnets. This makes sense since with more hidden states, we are able to keep proper sentence structure instead of randomly placing words like when we have 1 hidden state. Therefore, with more hidden states, we would likely see more related words in all the different states, helping to keep phrases that are more accurate to what is in Shakespeare's sonnets. Generating poems in a naive manner is not quite effective, but is still useful in attempting to read and analyze the generated poems. We can try to improve this process by implementing ideas from the Additional Goals section.

5 Additional Goals

1. Haiku (including sample sentence generator by syllables)

We implemented the model for a different form of poem, Haiku. Haiku is a short, unrhymed poem originally from Japan. Traditional Haiku poems have three lines of 17 syllables in total. The first and the third lines will contain 5 syllables each, and the second line will have 7 syllables. Another important feature of Haiku is the theme: Haiku contains *kireji* (cutting words) and *kigo* (seasonal reference). We first started the process by collecting examples of Haiku poems which we can use to train our model. We processed our Haiku data the way we processed Shakespearean sonnets - we added each word appearing in our data into observations map, which we used to train our model, along with the provided syllables dictionary.

Challenges

Here's the link to our Haiku train data: Haiku train data
In general, we had difficulty sourcing good enough train data for our model for the following reasons:

1) Theme features

Before collecting training data for the Haiku model, we had to think how to include the features of Haiku. Features like *kireji* or *kigo* were hard to include. However, we thought we could incorporate *kigo* by simply including words that relate to seasons and nature. Although it would've been difficult to force the presence of *kigo* words (i.e without having something like a *kigo* specific ditionary), we modeled *kigo* by carefully choosing our training data. Since most Haikus already use a lot of metaphors and abstract imagery that exhibit *kigo*, training our model on poems that we saw exhibited ample *kigo* resulted in our generated Haiku displaying *kigo* as well. Thus, we modeled *kigo* without having to explicitly identify which words counted as seasonal references. On the other hand, we quickly learned *kireji* could not be implemented in English. *Kireji* actually refers to specific Japanese characters that cause pauses or tone switches when used in the right place such that Japanese *kireji* is closer to English punctuation than words. There is no exact English equivalent for *kireji*, so due to the language barrier, *kireji* is omitted in English (that includes Japanese Haiku that's translated to English and Haiku that's originally written in English). So in English, Haiku simply doesn't exhibit *kireji*, so we had to give up on implementing this feature.

2) 5-7-5 style

We decided to keep the 5-7-5 style in our poem generator, which is definitely the most distinctive style that characterizes Haiku poems. We initially tried to create training data using Haiku from one poet, just like how we did for the sonnets. However, we had to soon give up on this idea because of the following reasons. First, a lot of the prominent Haiku poets were Japanese poets (usually from a few hundreds of years ago). So, their Haiku were originally written in Japanese, and thus the 5-7-5 rule is followed in Japanese, but not in the English translation of the poem. Second, a lot of modern Haiku experiment with breaking the 5-7-5 rule. So, unlike Shakespearean sonnet, it was nearly impossible for us to consist our training data of Haiku from a singular poet that also have the features (such as 3 lines, 5-7-5 rule, 17 total syllables) we were looking for. Instead, we incorporated Haiku written by multiple poets. We also decided that finding poems that stick to the Haiku rules are not too important for our model training, because we are training our model based on parsed observations (in other words, mappings of words to the frequency), so the structure only becomes important when we generate the poem.

Output

We trained our Haiku poem generator with 5 and 15 hidden states.

```
### Naive Poem Generation: Haiku
### Number of hidden states: 5
### Number of iterations: 100

(syllables: 5) Lovely the the my
(syllables: 7) Green either comes leaves trees star green
(syllables: 5) Bow towards her the it
```

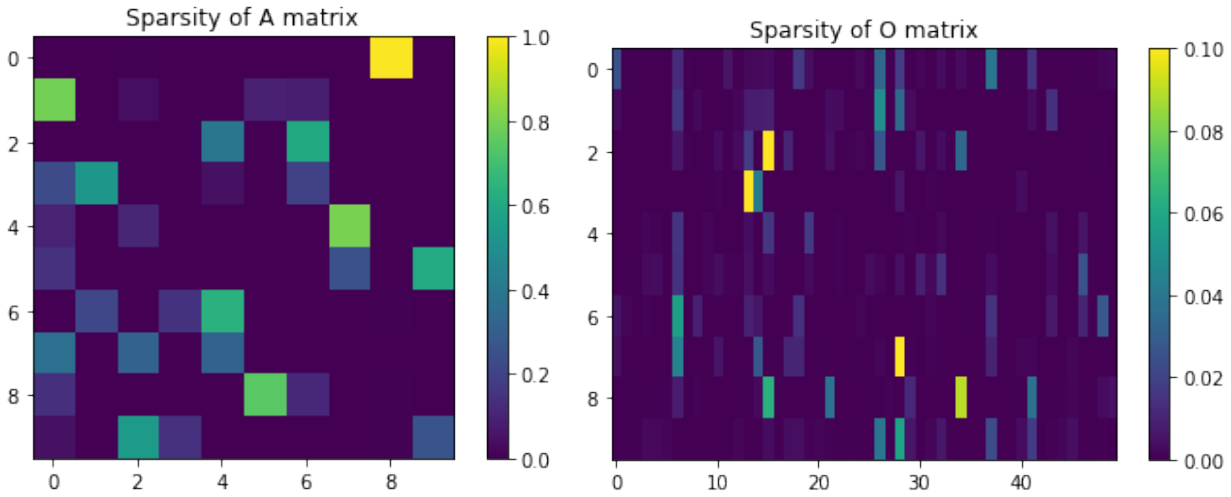
```
### Naive Poem Generation: Haiku
### Number of hidden states: 15
### Number of iterations: 100

(syllables: 5) On the fingers how
(syllables: 7) Beside i up shade morning
(syllables: 5) We twilight a way
```

We implemented a function called sample_sentence_syllable and sonnet_generate_emission which generates sample sentences based on the number of syllables, not the number of words. Thus, our Haiku perfectly follows the 5-7-5 pattern, even though our poems don't make much sense like our Sonnet example. We also notice that the poem contains *kigo* (seasonal reference), even though we never did any data preprocessing to include this feature (such as 'Green', 'trees', or 'twilight'). Just as we expected, our model learned *kigo* because the Haiku we used as training data had profuse seasonal references. Other than *kireji* (which isn't even incorporated in English), our model generation implemented all the features of Haiku successfully.

6 Visualization & Interpretation

We decided to do our visualizations on our basic model with 10 hidden states. In order to analyze the model, we used the same visualization methods from HW6. Using the method *visualize_sparsities*, we visualized the sparsities of the trained model matrices A and O. Using *states_to_wordcoulds*, we were able to inspect the words that were most representative of each of our hidden states. Finally, using *animate_emission*, we analyzed how our model connected hidden states to generate emissions for our sonnets. The visualizations can be seen throughout the following discussion.

From the sparsity visualizations of trained matrices A and O, we can see that they are both very sparse with few values over 0.5. This comes as no surprise since we're using the same unsupervised training method from HW 6, and the HW6 unsupervised model matrices possessed similar sparsities. The extreme sparsity tells us that certain transitions and observations between states and emissions are much more defined compared to others. In other words, our model focused on certain connections and gave them much more probabilistic weight, revealing a certain structure/pattern to the sonnets that are generated. We can most likely attribute the learned grammar and sentence patterns consistent in the our generated poems to this sparsity and its resulting defined connections.



Analyzing the word clouds of our model, we can see the words that are most representative for each state. For each of the 10 hidden states, the following 10 words are most representative of the state.

State 0: thou, make, eye, let, upon, show, give, tell, art, see
State 1: thou, yet, mine, lest, look, nothing, place, gentle, one, long
State 2: thy, thou, love, make, must, say, think, live, now, nothing
State 3: O, dost, stand, although, music, ere, came, deaths, boy, hour
State 4: love, thee, time, heart, best, life, world, thought, heaven, others
State 5: love, self, eye, heart, beauty, day, fair, part, will, hours
State 6: sweet, love, true, dear, though, self, every, poor, good, yet
State 7: doth, hath, will, may, now, see, mine, might, thus, bring
State 8: thy, thee, thine, doth, mine, love, whose, times, ill, better
State 9: thou, still, doth, yet, away, eyes, dost, thine, hand, kind

The first and potentially most obvious observation is that some states look like they are repeats of others and that many states share the same most representative words. For example, the words "thou", "thy", and "love" appear in the top 3 most representative words of at least 3 states each. This is probably due to the fact that Shakespeare used these words so often and the fact that they can be used in many different grammatical contexts. Thou, thy, and love are used in so many of Shakespeare's sonnets - thou and thy because they are some of the most used pronouns and determiners in old English, and love because most of Shakespeare's sonnets had love as a theme. Additionally, these words are flexible when it comes to grammar. Love can be used as a verb, a noun, and in many different phrases and similarly, thy and thou come after many different words since they are used so often. Since these words come after so many different words and are used so often, our model learns this flexibility and puts them in multiple different states.

Additionally, we can use the most representative words of each state to interpret some semantic meaning of some states. For example, we can reasonably surmise State 0 is action/verb words, State 3 is existential/dark nouns and topics, State 4 is general nouns/places, State 5 is endearment, and State 8 is possessive and descriptive words. Here, our hidden states seem to capture the types of words and parts of speech, rather than number of syllables or stressed vs unstressed words. This gives us some idea for why our model makes its connections between states and lets us better understand how our model works overall. The word clouds can be seen below.

State 0

State 4

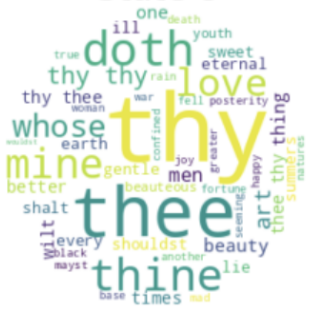
State 1

State 5

State 2

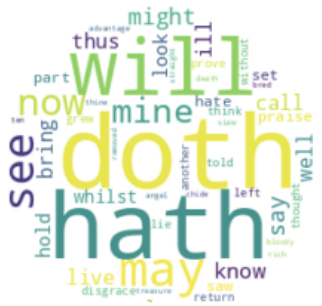
State 6

State 8



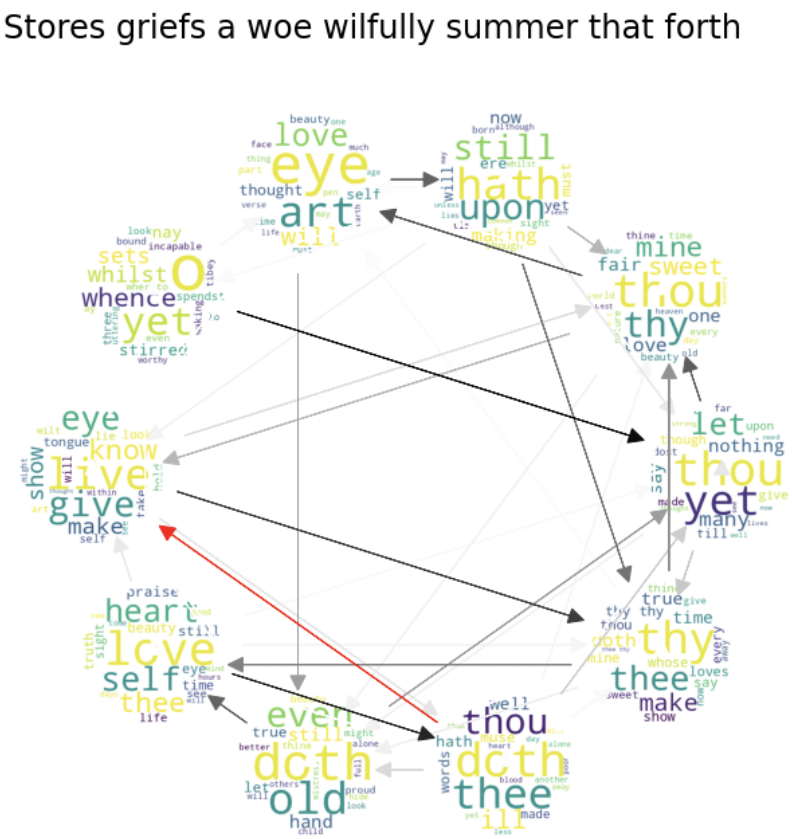
State 3

State 7



State 9

Finally, the example emission diagram below lets us see an emission and the transition probabilities between hidden states for our model. The darker lines show the connections that are most probable from each hidden state to another at each step. The sentence "Stores griefs a woe willfully summer that forth", although it doesn't make much sense, we can see some semblance of grammar and sentence structure. Although it is not perfectly grammatically correct, it is much better than a random jumble of words lined together. Analyzing this model we can see why certain states are more connected than others. Their semantic meanings that we inferred up above let us reason why our model has connected them to produce more grammatically correct and sensible sentences.



7 Extra Credit

Additional Feature 2: Rhyme

We implemented rhyming very easily by storing dictionaries for the rhymes Shakespeare used. Since the form of Shakespearean sonnets, all share the same rhyme pattern, we know which words at the ends of which lines rhyme, so we hard-coded which lines had ending words that rhymed and stored those rhymes in pairs in dictionaries. Thus, given the training data of Shakespearean sonnets, we were able to generate dictionaries for the rhyming pairs. To generate sonnets with rhyme pattern, we simply generated rhyming pairs for each pair of lines that needed to rhyme. Then, we generated the model using all the previously described training methods above, except in the opposite direction. Thus, we started from the rhymes, then generated the lines of the sonnets in the opposite direction. Doing this, we successfully generated rhyming sonnets.

Below are some rhyming sonnets generated by our model with the feature implemented:

```
Rhyming Poem Generation with 1 Hidden State

Limping me that living shall that adieu
My hymns are straight with feeding and thou rare
To love she give reap delivers that you
Wherefore dost needst I name making or care
To a as thy me but all wantonly
Thing hell book what patience day excellence
And whilst mine for and those never bide dye
Tears to have uncertain when thorns expense
You him thy to tongues remover dear will
Thievish glass from I willingly the trust
Flower chide fled other not any is ill
This swift a know in to respect you thrust
  Saw might is eyed I shall on to translate
  Find disperse or makst nimble spend prime state
```

```
Rhyming Poem Generation with 10 Hidden States

Gone guard bare hand all then wilt thee me thine
Loves thy bore tonguetied and which nor proceed
Not I hath beauty a perfumes thy mine
I time all noted ceremony weed
Day my stand bears silent moan controlling
Might a give which outward even thy from check
Goodness grace invention it me rolling
Own it may gold thee think own do they beck
It on him such strangle being making blot
Of this my by still sufferance my mock prove
Die good so am cured want veins to truth got
Doth to some thou brand sight of turns fire love
  Queen that moan you or beauty argument
  His for youths that unlettered in invent
```

Additional Feature 3: Incorporating Additional Texts

The additional data file containing Spenser's sonnets in the Amoretti was implemented in the "Add in Spencer's Sonnets" section of the code. The combined authors' resulting observations and observations map generated both rhyming and non-rhyming sonnets. The following image is an example of a generated rhyming sonnet followed by an image of a generated non-rhyming sonnet.

```
My see to of worst the eyes shall heart wrong
Poor with and forth hence since be lively thrive
So youths richly princes art since belong
When to seeing kill less you to derive
Thy strange seek will ruth she absence sweet sang
Is grace my defaced fiery of commits
Was extreme nights a and not thoughts my hang
Faith trenches thee than most your most befits
To will gave old the of being be shall hell
Cruel to length fruit than sire or no his
Self the not will patience tender not well
After when to is her sweet them will is
  When but her on in as their in faces
  Is three my sweet and to show should graces
```

```
Bids then bases nor calms in rest with
That thou of this she one that victors
Chose sorry faces merit as she to attend
Nought should death graceth all you whereof this

When all love emprize sweet to fade to
Bough such unto my brow debt day your
My fine the thee false hercules the your
Cuckoo gildst dear merchants faults of with sun

Both these worth disdaineth in the wound or
Sacred mirth ornament is fry is is friends
Made loves perfection which truths in thereon doth
Gravity work thee tomb no sundry they life

  Besiege brains be with a damsel debate who
  Most taste green your heaven are I act
```

While the HMM successfully produced sonnets, the resulting sonnets tended to make less sense and have darker themes than the sonnets produced only by Shakespeare's poems.