1 Tokenizing

The sonnets were initially tokenized by words using the NLTK's function word_tokenizer because word is the smallest entity that we can tokenize in the sonnets such that we can still regenerate sentences with the tokens later. We created two types of data set. First, we consider each line as a sequence, and second, we consider each sonnet as a sequence.

The first change we made after running the learning algorithm was to remove all punctuations. We decided to generate the sentences backward using a seed (i.e. a word) to achieve accurate ending rhythm. Thus, the punctuations were mostly irrelevant because they showed up mostly in the end. In addition, we also find that the model with punctuations generated sentences that sometimes had brackets that did not match or commas that did not make gramatical sense. Therefore, punctuations were removed in our final version.

Furthermore, we counted the bigrams within the sonnets and replaced each pair of consecutive words that showed up frequently in the sonnets as one "word". These word pairs were added into the list of all word tokens. In total, 29 pairs of words were added. Each pair appeared in the sonnets at least 15 times. The top five examples were "my love", "thou art", "my self", "in the ", "that i" which appeared 41, 33, 29, 29, and 28 times respectively.

Lastly, we created a rhythm dictionary that was used in poem generation later to create sentences that had the correct rhymes. For the rhythm dictionary, we collected all ending words in each line of the sonnets that rhymed. For example, "increase" and "decrease" is a pair of words that rhymed. So, they are stored as a tuple in the rhythm dictionary.

2 Algorithm

For unsupervised training of our Hidden Markov Model, we used the Baum-Welch algorithm. We used the TA solution from HW #5 for the implementation. The main three parameters we control were (1) the number of EM iterations, (2) the number of hidden states to use, and (3) whether to train on lines of each poem or the poems themselves . Regarding (1), we used 1000 iterations, because this was enough iterations to show convergence to a very reasonable precision. Regarding (2) and (3), we trained several different HMMs, each with a different number of hidden states (8, 10, 12, 15, 16, 20, 25, 30). For each choice of hidden states, we also trained on each line of the poem, and on each poem itself. We then visualized the HMMs through both the transition matrix and observation matrix (discussed in section 5), and examined the generated poems to determine the optimal number of hidden states based on a qualitative analysis of the HMM structure, and whether to train on lines or poems.

We eventually chose an HMM with 20 hidden states trained on the individual lines of the poem, based on the underlying structure of the HMM and the qualitative quality of the generated poems. When we assess the underlying structure of the HMM, we refer to meaningful patterns in the transition matrix and observation matrix (also whether states generate words with discernible patterns). This underlying structure is discussed in section 5. When using fewer or more states, this structure was less clear, resulting in our decision to use 20 hidden states.

When generating our poem, we also needed to keep track of both syllable count. To do this, we used the NLTK CMU dictionary to keep syllable counts.

3 Poem Generation

First Bare-Bones Implementation

For our initial bare-bones implementation, we originally generated poems by the following steps:

1. Pick a uniformly random states to begin at
2. Initialize an empty line and initialize the syllable count of this line to zero.
3. Sample the next state based on the previous state and the transition matrix probabilities.
4. Add a word to the line based on the current state and the observation matrix probabilities
5. Count the number of syllables in the added word and add it to a syllable count
6. Repeat steps 3-5 until we have created a line with at least 10 syllables. Save this line to our poem.
7. Repeat steps 1-6 until we have generated 14 lines for our poem.

To get the syllable count in step 5, we used the NLTK CMU dictionary to keep syllable counts. By following the steps above, we were able to generate 14-line poems with each line containing *at least* 10 syllables.

4 Poem Generation/Additional Goals

However, we wanted to generate poems with *exactly* 10 syllables. To do this we modified step 6 to get the following algorithm:

Implementation with Syllable Counts (No Rhyme)

However, we wanted to generate poems with *exactly* 10 syllables. To do this we modified step 6, and get the following algorithm:

1. Pick a uniformly random states to begin at
2. Initialize an empty line and initialize the syllable count of this line to zero.
3. Sample the next state based on the previous state and the transition matrix probabilities.
4. Add a word to the line based on the current state and the observation matrix probabilities
5. Count the number of syllables in the added word and add it to a syllable count
6. Repeat steps 3-5 until we have created a line with at least 10 syllables.
   (a) If we have exactly 10 syllables. Add the line to the poem
   (b) If we have >10 syllables, remove the last word, go to the previous state and repeat steps 3-5.
7. Repeat steps 1-6 until we have generated 14 lines for our poem.

This gave poems with exactly 10 syllables. The tradeoff here was that the last word of the line was often restricted in its number of syllables (i.e. if we had 8 syllables in our line already, we had to pick a last word with less than 3 syllables). Thus the last word was picked from a probability distribution that did not exactly match that given by the bare-bones HMM.

Implementation with Syllable Count and Rhyme

Finally, we want to implement rhyme. To do this, we had to generate our poems backward using the following algorithm:

1. Create a dictionary of rhyming pairs based on the rhyming pairs in the shakespeare poems we trained on.
2. For lines 1, 2, 5, 6, 9, 10, 13, randomly choose a word from the rhyming dictionary. For all other lines, choose the rhyming word that goes with the corresponding line (e.g. for line 4, choose the rhyming word from our rhyming dictionary that is paired with the word chosen for line 2).
3. Sample the state based on calculated probability $P(y \mid x)$ where y represents a hidden state and x represents our observation of the first word. We can calculate $P(y \mid x)$ using the formula $P(y^1 = z \mid x) = \frac{\alpha_z(1)\beta_z(1)}{\sum_{z'} \alpha_{z'}(1)\beta_{z'}(1)}$. We use the *forward, backward* functions we wrote for HW #5 in order to get α and β. One we have our chosen first word and have calculated $P(y \mid x)$, we can sample from this distribution to get y^1.
4. Add a word to the line based on the current state and the observation matrix probabilities
5. Count the number of syllables in the added word and add it to a syllable count
6. Sample the next state based on the previous state and the transition matrix probabilities.
7. Add a word to the line based on the current state and the observation matrix probabilities
8. Count the number of syllables in the added word and add it to a syllable count
9. Repeat steps 6-8 until we have created a line with at least 10 syllables.
   (a) If we have exactly 10 syllables. Add the line to the poem
   (b) If we have >10 syllables, remove the last word, go to the previous state and repeat steps 6-8
10. Reverse the generated line (because we have generated the line backwards to ensure proper rhyming), and add it to the poem.
11. Repeat steps 2-10 until we have generated 14 lines.

Generation Result/Discussion

Using this algorithm, we were able to successfully automatically generate poems using an HMM trained on several shakespeare poems that obeyed the syllable count and rhyming pattern of a shakespearean sonnet. However, by implementing rhyme in the above fashion, we had to restrict the choice of first word in every line to one from our rhyming dictionary. Thus the structure of our generation algorithm using HMM was different. In our first step, rather than transitioning to an initial state (based on A) and generating a word (based on O), we had to first choose the word, and then sample the most likely state.

Generating the poems in this fashion, enforcing both rhyming scheme and syllable count, had the effect:

1. Since we chose to cut off the line at 10 syllables as we did, the last word of the line was often restricted in its number of syllables (i.e. if we had 8 syllables in our line already, we had to pick a last word with less than 3 syllables). Thus the last word was picked from a probability distribution that did not exactly match that given by the bare-bones HMM, in order to ensure each line had 10 syllables.

2. Because we were considering transitions backward (to the previous word in the line), the state transition matrix became less intuitive to follow since reading sentences backwards results in different transition structure and is less natural to humans.
3. Sometimes the first word of each line did not fit as a first word, since it was the last word generated in our backwards generation scheme.
4. Our first word of each line was limited to our rhyming dictionary. Furthermore, we had to calculate $P(y \mid x)$ in order to get the first state from the first word,

Below is one of the poems we generated with the proper rhyming scheme and syllable count (chosen for Piazza submission):

> Love my self alteration side granting
> Plague for and bright oaths but sit faces moan
> Sky grow'st best wilt or not to on wanting
> Up turns abide votary shall upon
>
> Not bide canst the that which to oppressed
> Am too black try is fair am be increase
> Usurer in chest state party age rest
> Change of in not to knows my muse decease
>
> Can canker be war him forth him power
> Days loss is ward clock dead eloquence pride
> I still verse with abide play'st call flower
> Again cruel told kindness odour one side
>
> Than to that i west slow and sounds a reap
> All friend great poor tear that all that i leap

5 Visualization and Interpretation

The top 10 words that associate with the hidden state are given in Table **??** (the numbers below the words denote the probability). The number in bracket is the total number of top words to form approximately 0.5 probability. Hidden state 11 and 12 have probabilities that are more diffused and they includes more words that are nouns, adverbs or adjectives. Hidden state 1, 8, 9, and 18 have a more concentrated probabilities, and they tend to be words that are used to connect phrases like particles, conjunctions, and pronouns. Hidden state 10 includes the 5 "wh" words for questions.

Figure **??** shows the observation matrix in which each row is normalized to 0 and 1 for clearer color contrast. Each column represents a token and the columns are sorted by the frequency of the tokens. The top plot shows all tokens, and the bottom plot shows the first 100 tokens. From the plots, we can see that the most common words in general have a higher probability than the less common words. Words like "and" which are very frequently used (473 times) in the sonnets show up in a few hidden states with high probability.

If we also consider the transition matrix (Figure **??**), we can observe some interesting state transitions that makes grammatical sense. For example, hidden state 11 is most likely to transition to hidden state 19 with probability 0.9330. Combinations of the top 4 words "of", "in", "to", and "not" in hidden state 19 with most of the top 10 words in hidden state 11 (e.g. love, i, one, hath, beauty, this, and most) seem to make grammatical sense. Another example is hidden state 1 is most likely to transition to hidden state 12 with probability 0.9836. Combinations of the top words "is" and "time" in hidden state 12 with "the", "my", "a", "so", "such", "their", "his", and "of the" in hidden state 1 also make grammatical sense.

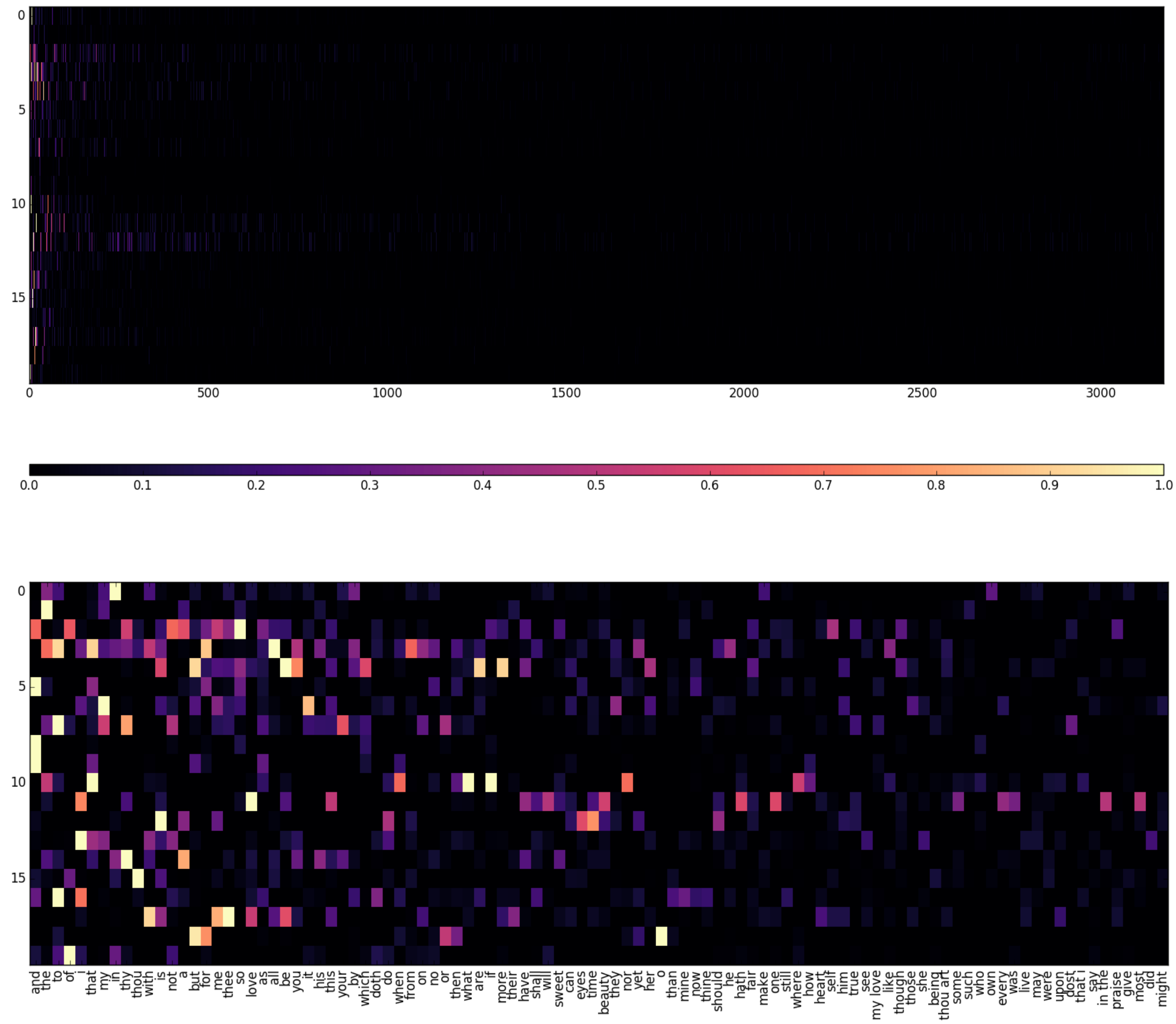

Figure 1: Normalized observation matrix. The top plot shows all tokens, and the bottom plot zooms into the first 100 tokens.

Table 1: Top 10 words for hidden states and the probability. The number in bracket is the total number of top words to form approximately 0.5 probability.

State	Top 10 Words									
0	in	the	by	own	my	with	make	to	show	your
(26)	0.1054	0.0399	0.0352	0.0300	0.0250	0.0244	0.0230	0.0230	0.0149	0.0140
1	the	my	a	so	such	their	his	of the	well	thine
(9)	0.2325	0.0604	0.0481	0.0337	0.0319	0.0283	0.0251	0.0204	0.0164	0.0158
2	so	not	and	of	a	thy	me	self	youth	thee
(50)	0.0364	0.0248	0.0247	0.0234	0.0223	0.0203	0.0195	0.0169	0.0143	0.0140
3	all	to	that	for	the	from	with	you	he	yet
(23)	0.0454	0.0417	0.0413	0.0400	0.0313	0.0305	0.0236	0.0221	0.0192	0.0187
4	be	but	more	are	you	which	is	her	so	i am
(30)	0.0444	0.0407	0.0402	0.0401	0.0332	0.0262	0.0257	0.0205	0.0179	0.0177
5	and	that	for	so	no	now	then	eyes	to me	when i
(35)	0.0985	0.0387	0.0361	0.0311	0.0217	0.0210	0.0152	0.0150	0.0148	0.0128
6	my	it	they	me	i	but	those	this	her	is
(15)	0.0993	0.0854	0.0403	0.0371	0.0271	0.0259	0.0257	0.0228	0.0226	0.0205
7	to	thy	your	my	not	or	the	dost	on	as
(21)	0.0702	0.0565	0.0442	0.0384	0.0333	0.0318	0.0217	0.0215	0.0201	0.0167
8	and	which	so	who	therefore	if	even	not	or	for
(2)	0.3856	0.0657	0.0509	0.0439	0.0202	0.0189	0.0136	0.0134	0.0112	0.0106
9	and	that	as	but	when	how	which	since	for	can
(3)	0.2786	0.0876	0.0825	0.0707	0.0523	0.0423	0.0287	0.0221	0.0193	0.0151
10	if	what	that	nor	when	where	the	let	why	how
(10)	0.0723	0.0719	0.0719	0.0502	0.0492	0.0400	0.0380	0.0336	0.0311	0.0221
11	love	i	one	hath	beauty	this	most	in the	will	part
(61)	0.0354	0.0265	0.0209	0.0208	0.0200	0.0185	0.0178	0.0177	0.0175	0.0149
12	is	time	eyes	do	should	a	sun	summer	age	best
(81)	0.0305	0.0235	0.0183	0.0143	0.0126	0.0116	0.0105	0.0101	0.0100	0.0095
13	i	that	with	not	my	do	did	she	see	is
(31)	0.0914	0.0403	0.0359	0.0352	0.0342	0.0220	0.0197	0.0193	0.0180	0.0168
14	thy	a	his	in	you	your	sweet	have	the	with
(18)	0.0958	0.0783	0.0379	0.0368	0.0294	0.0269	0.0267	0.0252	0.0237	0.0196
15	thou	of	is	in	all	being	and	were	from	day
(31)	0.1502	0.0474	0.0367	0.0202	0.0170	0.0162	0.0136	0.0133	0.0126	0.0114
16	to	i	doth	mine	and	than	shall	as	by	thine
(19)	0.1011	0.0712	0.0368	0.0312	0.0304	0.0244	0.0223	0.0199	0.0197	0.0194
17	thee	with	me	be	love	is	their	all	this	heart
(28)	0.0661	0.0605	0.0547	0.0405	0.0358	0.0269	0.0249	0.0176	0.0168	0.0155
18	o	but	for	or	then	yet	against	save	to	when
(3)	0.1761	0.1683	0.1355	0.0924	0.0595	0.0321	0.0303	0.0241	0.0198	0.0169
19	of	in	to	not	i	doth	and	will	do	of thy
(14)	0.1831	0.0562	0.0431	0.0380	0.0269	0.0249	0.0205	0.0200	0.0166	0.0150

5 VISUALIZATION AND INTERPRETATION

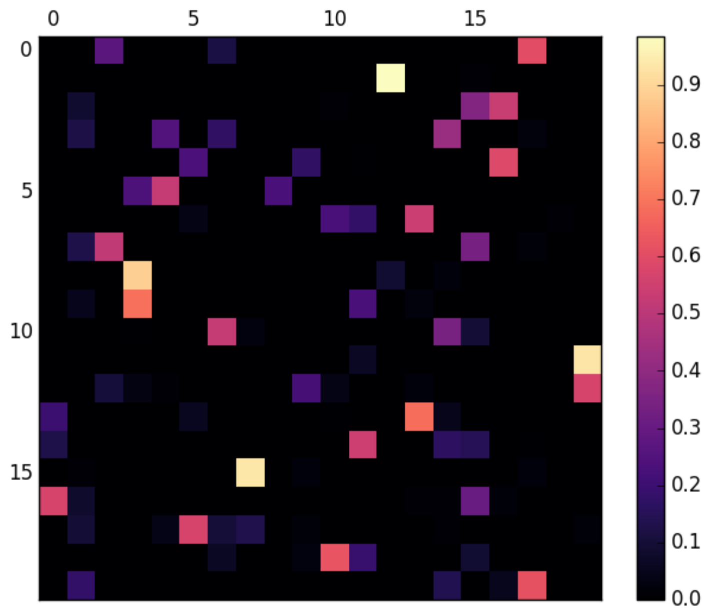

Figure 2: State transition matrix

6 Conclusion

The poems we generated had some structure and seemed to generate some grammatically sensical lines. It was usually able to string together short sequences of words in a grammatically coherent way (though this often broke down with longer sequences). In the "Poetry Generation/Additional Goals" section, we discussed the trade-offs we had to make to enforce rhyme and syllable count in our poem. A few other shortcomings of our poems, and potential remedies, are discussed below.

1. Because we trained our HMM on individual lines, the generated poems had no theme (beyond the coincidental). Thus each line often showed little thematic correlation to adjacent lines. This could be remedied by training on poems rather than lines, but we ended up choosing to train on lines, because the grammatical structure of generated lines was more coherent. With more data, we could consider training on full poems though.

2. By nature of generation with HMMs, words can only have direct relation with the previous word. Therefore it becomes likely for long lines to wander, and difficult to string together many words for a single coherent idea. Thus, in our poems, seldom are there long strings of words forming a single coherent idea.

3. Some words or pairs of words were extremely frequent in the poems, and the prominence of common words like 'the', 'be', and 'he' often threw off the structure of sentences and resulted in part-of-speech sequences that did not make great sense. To remedy this, we could try training our model to learn part-of-speech tags. Google's Speech API is able to identify part-of-speech tags in a given sentence, so we could use this to help our model understand words' part-of-speech tags and generate sentences with proper part-of-speech sequences (e.g. pronoun, noun, adverb, verb, noun).

Despite these issues, we were happy that our poems showed some likeness to shakespearean poetry with mostly sensical lines. Furthermore, we were able to visualize our Hidden Markov Model and extract some structure and learn some interesting things about shakespeare's style and word structure (e.g. he really enjoys ”my love”).

We broke up the work...

6 CONCLUSION