Generating Shakespearean Sonnets with Hidden Markov Models

Ritwik Anand, Audrey Huang and Dryden Bouamalay

March 14, 2016

1 Overview

Shakespeare's sonnets beautifully convey complex ideas such as love and mortality while following a strict meter. The difficulty in generating a Shakespearean sonnet through machine learning lies in reproducing a timeless voice in a very limited structure while training on a small data set, only 154 sonnets. Literary elements such as meter, rhyme, theme, and tone must be examined, yet tokenizing then classifying each word or line in the training data by its properties is a difficult problem with imperfect solutions. Furthermore, improving each element gives some sort of tradeoff in the generated sonnet; perfecting meter may destroy voice, while focusing too much on eloquency may sacrifice structure.

We considered sonnet generation to be composed of three general stages: preprocessing, unsupervised learning, and poem generation. Training and poem generation were improved by factoring in iambic pentameter, sonnet structure, rhyme, and parts of speech, which were parsed from the input sonnets in the preprocessing step.

All three group members contributed equally to discussing implementation and improvement ideas. Audrey was responsible for preprocessing and visualization/interpretation, Dryden implemented the unsupervised learning algorithm, and Ritwik completed naive poem generation. All three members were involved with improving poem generation, and wrote their respective portions of the report.

2 Preprocessing

Tokenizing

We want to avoid overcounting the number of unique words or observations in Shakespeare's sonnets because our transmission and observation matrices are already sparse. When we reduce overcounting, there are fewer entries in these matrices and each word lends more information. Though there are many levels of preprocessing, the least drastic of which could involve removing capitals and the most drastic of which could involve conglomerating verb tenses, ideally it simplifies our HMM without removing much diversity and content from our sonnets.

The simplest method of tokenizing each sonnet is splitting each line into words by spaces.

However, each line ends with some form of punctuation, most commonly a comma or colon, and many lines have internal quotes, commas, and question marks. The naive tokenizing method would thus treat a word ending with punctuation as a different word from the word itself (e.g. 'word,' is different from 'word') even though they have the same meaning in a line. In addition, if a word was the first word in a line or a proper noun, it would be capitalized and considered different from the same all-lowercase word. Other confounding factors include possessive words (e.g. beauty, beauty's), words apostrophed to preserve meter (e.g. over, o'er), hyphenated words (e.g. love-god), plurals (e.g. word, words), and verb tenses (e.g. go, going, gone).

After naively tokenizing, we had 4726 unique words. Our first action was to make all words lowercase, which decreased the number of unique words to 4445.

After that, our primary concern was the use of punctuation, which overcounted the number of diverse words. We first tried substituting all punctuation–save hyphens–with empty strings using Python's regular expression package, re. However, this changed the meaning of words with internal apostrophes by turning possessive nouns into plurals (e.g. beauty's to beautys) and meter words to senseless syllables (e.g. o'er to oer). We wanted to keep words that had internal apostrophes and hyphens, but not quotations or other forms of punctuation. We considered removing all apostrophes at the begining and ends of words, which would eliminate quoted statements such as in the line ("Sings this to thee, 'Thou single wilt prove none'.", sonnet 8, line 14), but this got rid of meter words such as "'gainst" and "'tis". As a result, we settled with removing all punctuation but hyphens and apostrophes. This took care of the majority of the confounding problems while preserving the meanings of most words, save for quoted statements which are rare. This decreased the number of unique words we encountered to 3232, a decrease of about 25%. We kept hyphenated words and did not tokenize any words as bigrams.

We also removed sonnets 99 and 126 because they didn't have the standard 14-line format, which made parsing rhymes difficult. The unique word count afterwards was 3204.

Syllables

Shakespeare's sonnets are written in iambic pentameter, and each line is 10 syllables long. We hoped to reproduce this syllable count in our sonnet, but we first needed to create a dictionary with each unique word as key and the number of syllables as its value. Later in our implementation, we will use this dictionary to check if a generated line has the right number of syllables. To implement this dictionary, we used *cmudict* from NLTK's corpus module. *cmudict* has, for each word, a value corresponding to its pronunciation. For example, the key 'beautiful' has value ['B', 'Y', 'UW1', 'T', 'AH0', 'F', 'AH0', 'L']. The number of syllables is equivalent to the number of strings representing vowel sounds, in this case 'UW1', 'AH0', and 'AH0', which correspond to three syllables. The vowel sound ending in '1' is emphasized.

However, the sonnets possess apostrophe'd and hyphenated words, archaic language, plurals/tenses, and proper nouns not present in *cmudict*'s pronunciation dictionary. Of the roughly 3000 unique words, about 470 has pronunciations could not be found. There's probably a better or more efficient way of doing this, including existing algorithms on the internet, but we used syllable-counting rules enforced by a series of if-statements to determine the number of syllables in these remaining 470 words. Roughly, the number of syllables in a word is equal to the number of vowel letters minus the number of diphthongs, or strings such as 'ea' made by putting two vowels together. We also took into account exceptions such as having a silent 'e' or syllable-producing 'y' at the end. Due to the complexity of the English language, it's difficult to account for all edge cases and correctly count the number of syllables for all words. However, by checking the dictionary, the vast majority appear correct.

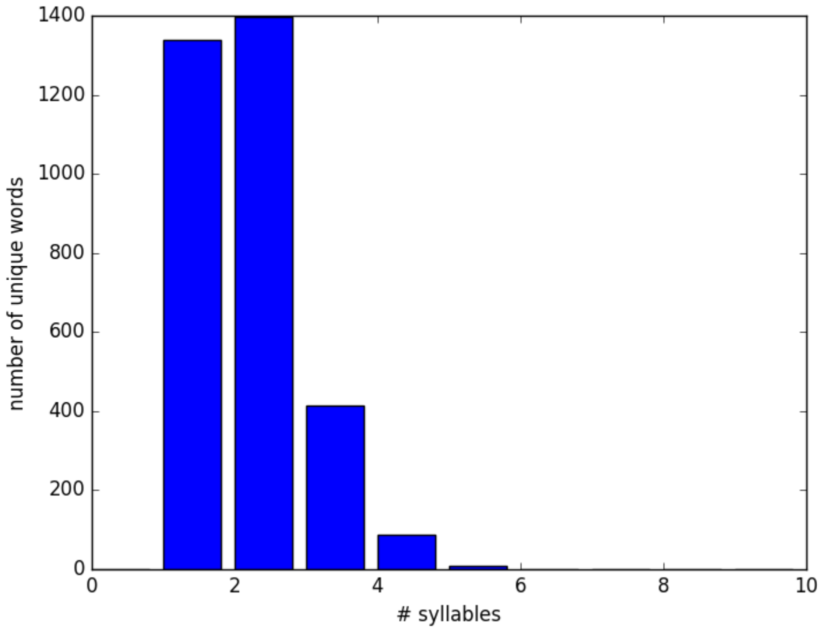

Figure 1: The syllable distribution for all words.

[H]

Indexing

Our HMM is trained on the lines from each poem. We created a dictionary of unique words, where each word was mapped to an index from 0 to 3203 which represented the observation state. We used this mapping to procure a list of lists, each of which represented the sequence of words in a line but in index form. This list was the observed sequence input to the EM algorithm.

Meter

Another feature we planned to reproduce in our generated sonnet is Shakespeare's iambic meter. Similar to other literary elements we considered, we first needed to map each unique word to the pattern of stresses in its pronunciation. Just as for syllables, we could exploit

NLTK's *cmudict* to determine stresses, since the vowel sound in an emphasized syllable ends with '1' and '0' otherwise. However, this still left hundreds of words unaccounted for, and we weren't sure of a directed and accurate approach we could use to discern their stress patterns. There is a large amount of uncertainty and noise, for example, single-syllable words have nebulous stress depending on their importance, meaning, and position in the line. Instead, we generalized this step to the following set of rules. Even-syllable words have emphasis starting on the first syllable and alternate unstressed/stressed after that, and odd-syllable words have emphasis starting on the second syllable and alterate unstressed/stressed afterwards. As a result, the stress pattern is purely dependent on the number of syllables, which makes things easier. There are clear exceptions, such as 'beautiful', which has stress on the first syllable but an odd number of syllables. However, we focused on this aspect of Shakespeare's sonnets less, since it has a less obvious overall effect on the quality of the poem compared to say, rhymes. We also hoped the HMM itself, or playing with/regulating the syllable count or number of words would replicate meter.

Number of Words

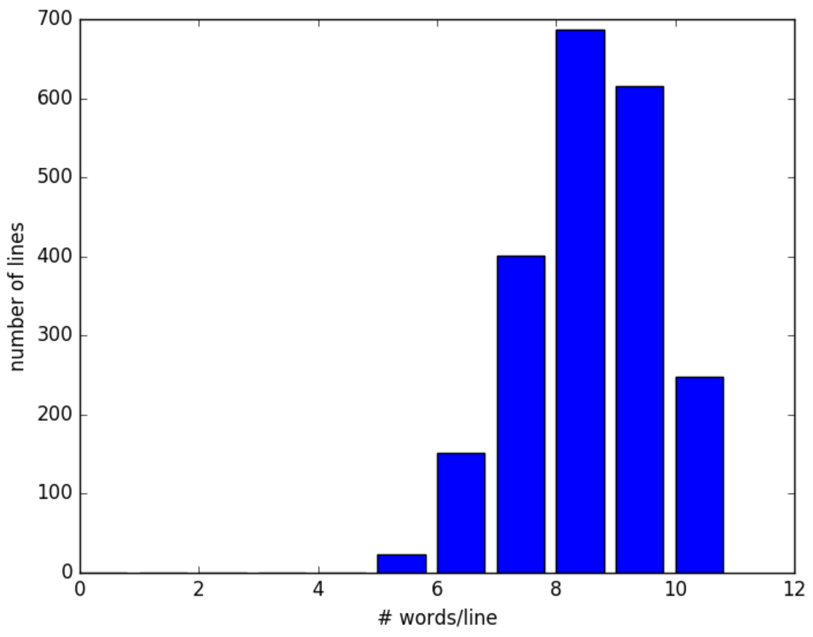

Figure 2: The distribution of number of words per line.

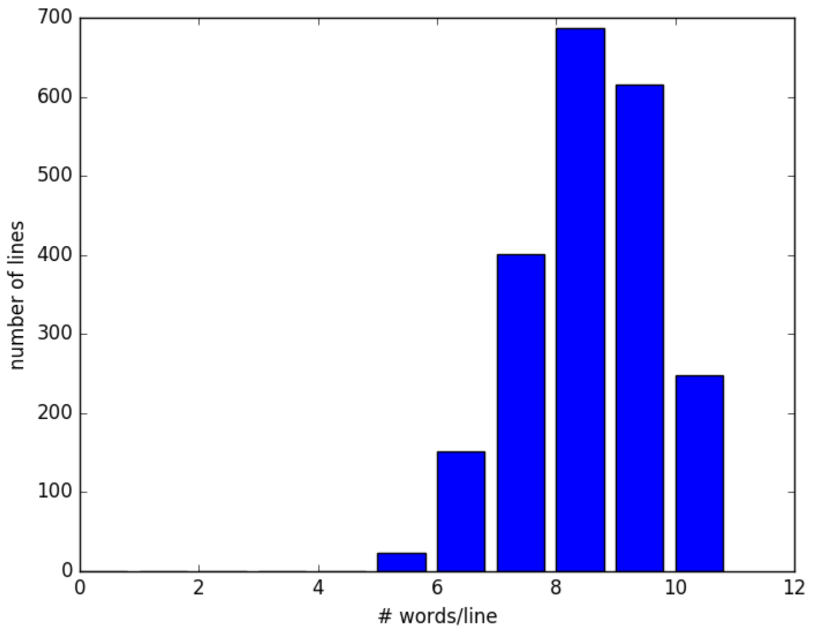

Figure 3: The distribution of word copies in Shakespeare's sonnets. The highest usage number is 485 for the word 'and', followed closely by 'the', 'to', 'my', 'of', 'i', 'in', and 'that', all of which are found in over 300 places.

Rhyming

In each quatrain, every other line rhymes. To reproduce this structure later, we first created a dictionary where each key is a unique word, and its value is a list of words it rhymes with. Parsing this out of the input file was much easier. We essentially parsed the last word of each line of a poem into a list, so that the 0th index represented the last word of the first line and so on until the 13th index for the 14th line. We then entered the 0th index-2nd index words, 2nd index-0th index words, and so on as key-value pairs into our dictionary based on Shakespearean sonnet structure.

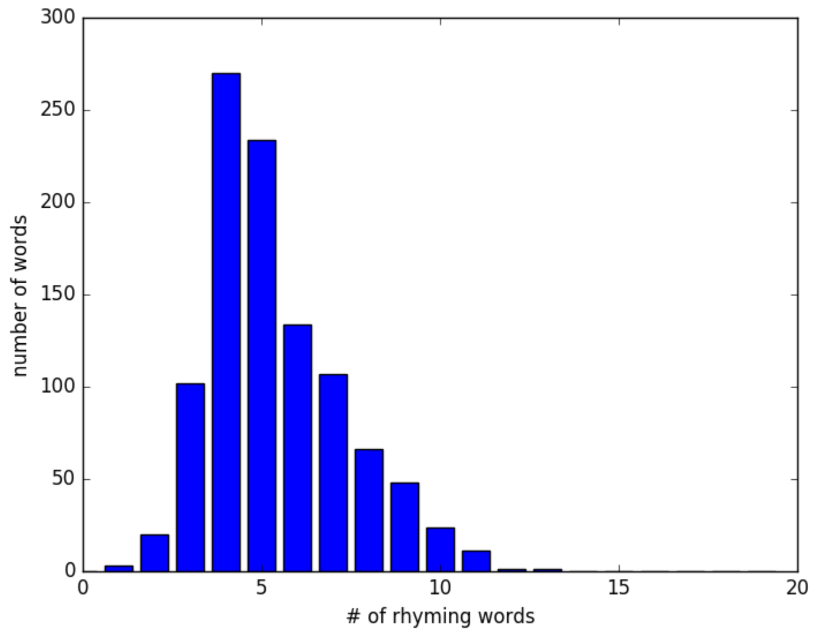

Figure 4: This graph displays the distribution of number of rhyming words for the set of unique word used in Shakespeare's sonnets.

Parts of Speech

Sentence structure produces a transition matrix for parts of speech. For example, a sentence often starts with an article, proceeds to a noun described by some adjective, and does some action with a verb. We hoped to consider this logical grammatical flow in our HMM, but first we needed to annotate each word in our training set by creating a dictionary of parts of speech with value being a list all the words that are that part of speech. NLTK has a pos_tag function which automatically determines the part of speech for each list of input tokens. NLTK classifies each token into one of 35 parts of speech. After creating this dictionary, we can convert the sonnets into a list of parts of speech, determine a transmission and emission matrix. In this case, each part of speech is a state, and each state's emissions are the words that belong to that part of speech from our dictionary. We can generate a naive sonnet using this approach, which will likely be poor because it will consider nothing but grammar and leave out important aspects such as theme and cohesive meaning.

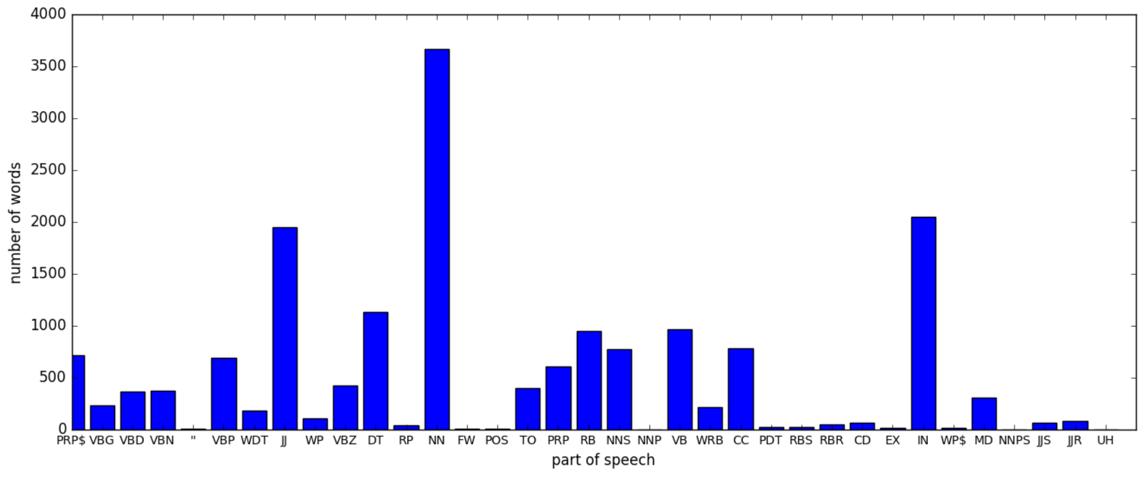

Figure 5: NLTK's pos_tag module classifies words in one of 35 parts of speech. This graph shows the number of words in Shakespeare's sonnets for each part of speech. Nouns 'NN' are by far the most common by a factor of 2.

3 Unsupervised Learning

To perform unsupervised learning for our HMMs, we implemented the Baum-Welch algorithm. The Baum-Welch algorithm uses the Expectation Maximization (EM) algorithm to find the maximum likelihood estimate for the transition and observation matrices given some set of observations.[1]

In particular, the Baum-Welch algorithm begins by randomly initializing (with normalization) the transition and observation matrices A and O, along with a starting state distribution S. By convention, each entry A_{ij} of A is the probability of transitioning from the j^{th} state to the i^{th} state, and each entry O_{wz} is the probability of emitting observation w while in state z. During unsupervised learning, we do not have access to the hidden states for the observed emissions, so we must update the A and O matrices via the following marginals during the maximization step:

$$A_{ab} = \frac{\sum_{j=1}^{N}\sum_{i=0}^{M_j} P(y_j^i = b, y_j^{i+1} = a)}{\sum_{j=1}^{N}\sum_{i=0}^{M_j} P(y_j^i = b)} \qquad O_{wz} = \frac{\sum_{j=1}^{N}\sum_{i=0}^{M_j} 1_{[x_j^i = w]} P(y_j^i = z)}{\sum_{j=1}^{N}\sum_{i=0}^{M_j} P(y_j^i = z)}$$

where N is the total number of samples in our dataset, M_j is the length of the sample in question, and $1_{[x_j^i = w]}$ is an indicator function that returns 1 if $x_j^i = w$, and 0 otherwise.

To perform the maximization step, we must compute the above marginals. To do so, the

[1]https://en.wikipedia.org/wiki/Baum%E2%80%93Welch_algorithm

Baum-Welch algorithm implements the Forward-backward algorithm during the expectation step. The Forward-backward algorithm begins with the randomly initialized matrices A and O and the starting state distribution S. For consistent notation, let the observations be denoted X and the hidden states by Y. By means of dynamic programming we compute two matrices α and β, where: [2]

$$\alpha_{ij} = P(X_1 = x_1, \ldots, X_j = x_j, Y_j = y_i | A, O, S)$$
$$\beta_{ij} = P(X_{j+1} = x_{j+1}, \ldots, X_{M_j} = x_{M_j} | Y_j = Y_i, A, O, S)$$

Given α and β, we can compute the above marginals by combining α and β into two other matrices γ and ξ: [3]

$$\gamma_{ij} = P(Y_j = y_i | X, A, O, S) = \frac{\alpha_{ij}\beta_{ij}}{\sum_{k=1}^{M} \alpha_{kj}\beta_{kj}}$$
$$\xi_{ijk} = P(Y_k = y_i, Y_{k+1} = y_j | X, A, O, S) = \frac{\alpha_{ik}A_{ij}\beta_{j(k+1)}B_{j(k+1)}}{\sum_{k=1}^{M} \alpha_{kM_j}}$$

These matrices contain the marginals we need in the maximization step above, and thus combining γ and ξ we perform the maximization step.

Our implementation is contained in `baum_welch.py` and is thoroughly documented. In brief, the code takes a list of samples, assumed to be lists of integers, that appropriately index into the A and O matrices. To generate the list of samples, we took the line(s) from the poems from which we wish to train a particular HMM and map those words via a dictionary to an index. This dictionary maps any word in our sample space to a distinct index, up to the maximum number of distinct words in our training set. Given this list of samples, we train a single HMM by performing the Baum-Welch algorithm on the input list.

In the interest of efficient implementation, most of the steps above are vectorized, and thus the code runs relatively quickly (e.g, 10 states takes less than a minute to train, and 100 states will take several minutes). The implementation converges with reasonably trained A and O matrices and outputs the trained matrices as `.npy` files to prepare for poem generation.

4 Visualization and Interpretation

We analyzed the emission probabilities of each hidden state in our HMM for enrichment in a particular part of speech, word "category", syllable count, or poem placement. The idea is that the states might have been trained to be "responsible" for an element of the sonnet, (for example, one state might emit primarily nouns and another past-tense verbs) and transmitting between states produces a line.

As shown in the pre-processing section, a small subset of words are highly enriched while the majority are only seen once. We first normalized each of the probabilities in O by the frequency of the word's appearance in the training data.

[2]https://en.wikipedia.org/wiki/Baum%E2%80%93Welch_algorithm
[3]https://en.wikipedia.org/wiki/Baum%E2%80%93Welch_algorithm

Parts of Speech

Next, we determined whether particular states were enriched in a particular part of speech by finding the probability of each state emitting each part of speech. Earlier, we used NLTK to classify each of our words into 35 parts of speech in a dictionary relating word to a list of the parts of speech it takes in the poems. We created an $N \times M$ zero-matrix, where N is the number of states and M is the number of parts of speech, and each cell records the total probability of that state emitting that part of speech.

Next we filled in this matrix. We mapped each column of O to the unique word its index corresponds to, and used our dictionary to determine the part of speech it most commonly takes, then added the probability of the state emitting that column to its corresponding cell in our matrix. Each state seems to have extremely similar emitting probabilities for each part of speech in the same order of magnitude. Nouns were also had the highest total probability (between 0.056-0.057) of emission for all states, which could be accounted for by the fact that nouns are the majority of words present in the sonnets.

Word Categories

Every sonnet, even each quatrain, has a particular theme. Stanzas trying to convey a particular idea are enriched in words of a particular category, for example happiness or sadness. We then wanted to see if certain states had been trained to emit words that belonged to any such category. We printed out the top 10 words by normalized probability that each state emitted. The full list can be found in the pickle file $'./pickle/top\_words.p'$. For example, the top words for the first state was [”love’”, ’lengths’, ’theirs’, ’wanting’, ’cheer’, ’erred’, ’thank’, ’enjoys’, ’shifts’, ”y’have”]. Many of our states shared top words such as ”love’”, ”’now and ”’will”, noticeably uncommon words with apostrophes. Words such as ’hair’ and ’wire’ were also frequent top words, but are not frequently present in sonnets. There was no particular theme we could attribute to each state.

Placement in Poem

In the sonnets, the 1st/2nd quatrains, volta, and couplet have different voices. In addition, the first/last word in each line may have different roles in establishing the flow and tone. We wanted to see if particular states specialized in emissions in the 1st/2nd quatrains, volta or couplet, and if they specialized in first or last words. We recorded a list of first and last words in the 1st/2nd quatrains, volta, and couplet separately. We then recorded the total probability of each state emitting a first word in each of these stanza groups, and of each state emitting a last word. To do this, we summed the probabilities of indices corresponding to words that matched those in our lists of first and last words. We found that the probabilities were almost identical for each state in all of these cases. The probabilities for the states emitting a first/last word in each stanza group is recorded below:

	1st word	last word
1st/2nd quatrains	0.070	0.071
volta	0.035	0.036
couplet	0.0178	0.0178

The probability of emitting a word is proportional to the number of lines in each stanza group; the 1st/2nd quatrain group has 8 lines, the volta group has 4 lines, and the couplet has 2 lines. Similarly, we can see the emission probability decrease by a probability of 2 after each row. There seems to be no specialization or pattern here.

Syllables

We considered the possibility that each state is responsible for emitting a particular length of word, although from *Fig. 1* we can see that the vast majority of words are 1-2 syllables. We determined the average number of syllables emitted by each state. All were between 1.71 and 1.73 syllables, which makes sense given the syllable-word distribution.

5 Poem Generation

Using our Baum-Welch algorithm, we trained A and O matrices using 50 hidden states with a tolerance of 0.001.

Once we initialized the start, transmission and observation matrices we had the tools required to generate poems based on our model. To generate our poems we would generate the quatrains simultaneously and adding in the couplet at the end. Each line of each quatrain was generated independently as well except for rhyming lines where we would force the last word of each pair of rhyming lines to rhyme with each other.

Before we could generate our lines we needed some kind of stopping condition to know when we could stop generating our sequence of hidden states. We considered using syllables as our stopping condition but this doesn't work out too well since the words are generated at the end and the same state can emit different words. We ended up finding the normal distribution that determines the length of an arbitrary line for the sonnets. Then for each line we would pick a length based on this distribution and stop generating the sequence of hidden states once we reached this length.

To generate each line we would randomly pick a start state with probabilities weighted by out start vector. We would then transition to the next state with probabilities determined by our transition matrix. Once we have our sequence of states we would convert each state to a word with probabilities weighted by our observation matrix, giving us our line. We then repeated this for each line for our sonnet.

With our naïve poem generation very little of our generated sonnet resembled the classic Shakesphere sonnet. The syllables were a mess and not very consistent, the stress pattern rarely fit and rhyming never happened. The poems didn't make much sense at all, with many non-sensical and non-grammatical structure appearing with high frequency. However as we increased the number of states we could see improvement in our poems. We could see a lower frequency of non-sensical sentences and grammatical structure and if we look less closely at the generated poems we could see a similar structure to the poem. As we only selected words that shakesphere used in his previous poems, many of which aren't commonly seen some of shakesphere's voice was retained in our generated poems.

With the above algorithm (along with the improvements we mention in the improvements section) we generated the following poem:

So shape doth it fair impiety your fruit,
Then tongues love forgetfulness even i,
Hath seem for expire when whole let verse mute,
Whole forgetst tongues perfect do married be.

Can love teach belong thy or after ranged,
Dwell give pleasing chase invent all despise,
Of doth whole self not nor whole live exchanged,
No in through every with user wit eyes.

Love no sweet equal my fine with fulfil,
Wantonly prisoner pitch your fill of greeing,
With thou of count graces hurt light to "will,
If not held compare me so on seeing.

Sad depends art foot is thievish so erred,
Shame grew shalt of thy so year they transferred.

We can see that Shakespeare's *abab cdcd gg* rhyming scheme was reproduced in our sonnet, save for one rhyme-pair which was likely due to a slant rhyme from Shakespeare's original sonnets that was parsed into our rhyming dictionary. All other generated poems had the correct rhyming scheme, but we chose this one because it was most coherent.

Recall that the third quatrain or *volta* confers a change in theme or voice. The first two quatrains use words like 'love' and 'pleasing' which confer a happier tone, while the third quatrain uses words like 'prisoner' and 'hurt' which imply a change to a more forlorn tone. The final couplet also uses words such as 'sad' and 'shame' which complete a transition to a darker voice toward the end of the poem.

6 Improvements to Poem Generation

Sonnet Structure

Each sonnet has the same general structure, three quatrains followed by a couplet. However, the third quatrain is called the *volta* and has a change in tone or content. The couplet forms the last two lines in the poem, and acts as its conclusion of sorts. To improve our sonnets, we could have trained our HMM separately on the first two quatrains, the third quatrain, and the couplet because these three parts of the sonnet play different roles and likely have different voices and content.

Rhyme

To capture the rhyme of the sonnets we considered the rhyming lines in each quatrain and the couplet as a pair. We generated the first line normally, for the second line we would only pick the last word such that it rhymed with the last word of the first. To accomplish this task we first created a rhyming dictionary which had each unique word that occurred as a last word in each of the lines and we added every corresponding rhyming word used in the poems. We wanted to keep the structure of our poem consistent so we created a new start vector to initialize the rest of our second line. We considered every state that emits the picked rhyming word and multiplied the chance of emission by the corresponding transmission column for the state. This gives us a new start vector from which we use to generate the rest of the line. We repeat this for every pair of rhyming lines in our poem to add the necessary rhyming structure of a sonnet.

Stress

Our idea for adding the stress pattern in our poem was to create four lists dividing our set of words used: words that start with stress, words that start with unstress, words that end in stress and words that end in unstress. To keep the pattern we note that a word ending in stress shouldn't transition to a word with stress and same with unstress. We would apply this after we generated our hidden states for each line. After we emit the first word we would simply zero the probability of the next state emitting a word that wouldn't follow the stress pattern. If we ever reached a scenario where we couldn't pick any word by somehow zeroing out all possible emissions we would throw out the hidden-states and generate a new one. However we were unable to implement this idea as we didn't know how to initially categorize the stress patterns of a word.

Syllables

To implement a proper syllable count we simply created a dictionary which mapped each word that appeared in shakesphere's sonnets to the number of syllables that word contained. We then would check the number of syllables of each generated line and if it was not equal to 10 we would throw out the geenrated line and re-do it. This was reasonable since we used the average length as our stopping condition so the number of syllables of each generated line would be very likely to be very close to 10 syllables on average.

Post-Processing

Every line of Shakespeare's sonnets ends with some form of punctuation, most frequently a comma or a period for the last line of each quatrain/couplet. Since we didn't keep punctuation save for apostrophes and hyphens and made all letters lowercase during pre-processing, as sa simple post-processing step we capitalized the first letter of each line, added a period to the last line of each quatrain/couplet, and added a comma to the end of the other lines. We also added a newline between each stanza to reproduce sonnet structure.

7 Conclusion

We performed unsupervised training of 154 of Shakespeare's sonnets via the Baum-Welch algorithm and generated sonnets with the trained hidden markov model. The generated sonnets incorporated some rhyming as well as a 10-syllable count for each line.

From our generated poems, it is clear that the hidden markov model captures some of Shakespeare's voice. However, semantically and grammatically the poems certainly suffer. This is not surprising since HMMs, like markov chains, operate on the markovian assumption, i.e., that the dependence for each term in the sequence is only on the previous term.

In other words, we may be able to transition from one word to the next in a somewhat reasonable manner, but globally the poem is not very coherent. This is to be expected from our model. At the same time however, it is quite exciting that we performed completely unsupervised learning and were able to train states that captured some of Shakespeare's style.

7.1 Improvements

One source of error is the level of pre-processing. Notably, we left in apostrophes from quotes and conjunctions, as we didn't have an easy way of separating the two. This caused the same words to be considered unique tokens. When we looked at the most probable emissions from each of our states, many states had infrequent, apostrophe'd words. To improve pre-processing, we could develop a method of removing grammatical apostrophes from words and treating conjunctions such as "o'er" the same as the un-conjuncted word, in this case "over". Other points of improvement include developing a better method of counting syllables for words not in *cmudict* using already-established algorithms, or considering the use of bigram tokens.

It's difficult for us to determine how well our HMM performed. It generated a poem that kind of made sense and rhymed, but lacked in grammar, clarity, and meter. In addition, the states seemed to all have basically the same purpose and emission probabilities for the factors we examined. It would have been interesting to compare its performance to an off-the-shelf hmm generator such as hmmlearn. In addition, we could have considered factoring in pronunciation in writing our poem instead of just removing them, which would have proved very a difficult task.
If we had more time, we would have tried creating a dictionary of word to pronunciation/meter or tokenized words as stressed/unstressed. syllables in order to try to reproduce the iambic meter. Our transition matrix would have thus represented a bipartite graph going between stressed/unstressed states. We could have also factored in the part-of-speech idea into generating the poem, where each HMM state represents a part of speech and going from one state to another is like composing a sentence grammatically. As mentioned before, all of these options have tradeoffs. Focusing too much on structure and grammar sacrifices coherence, and vice versa. We would have liked to develop a method of combining all of these factors in training an HMM and outputting a poem, but the problem is difficult and very open-ended