Miniproject 2: Generating Shakespearean Sonnets using First-Order Hidden Markov Models

Daniel Gu
Victor Han
Eve Yuan

1 Overview

In this project, we wanted to learn a generative model which can write Shakespearean sonnets. Using Shakespeare's 154 sonnets and Spenser's sonnets from his sonnet cycle *Amoretti* as unlabeled training data, we tried to train a first-order hidden Markov model (HMM) to generate sonnets similar to ones Shakespeare could have written.

A lot of the challenge in this miniproject is that first-order HMMs seem to be too weak to be able to perform this learning task. Sonnets have structure at varying levels of granularity, such as the iambic pentameter, which means that the previous word affects the current word, the syllable count, in which all of the previous words in a line affect the current word, and the rhyme, which is a "long-distance" interaction among line endings. And this is just the *syntactic* structure of the sonnet: the stanzas and couplet of a Shakespearean sonnet also need to be endowed with meaning. So a lot of the challenge was augmenting or clever tweaking our data to try to capture this structure using a first-order HMM.

More concretely, a challenge we ran into was the amount of time needed to train a first-order HMM. An off-the-self implementation such as the one by the Natural Language Toolkit takes a lot of time (a few seconds per iteration for 10 hidden states, significantly more at higher hidden states), and our implementation took similar (and usually longer) times to finish training the HMM. So it took a lot of time to check for errors in the EM algorithm implementation, and after it was debugged, a lot of time to look for a good number of hidden states for our HMM. We were also constrained in time since looking at higher numbers of hidden states took too long. To help alleviate the problem of too much training time, we tried training on subsets of the poems at a time, which did help in the debugging process. However, even with using only a few poems, using large numbers of hidden states still took their toll. Another problem appeared to be that Shakespeare does not repeat words very often. Thus, it seemed like we did not have very much training data on most individual words, and thus their roles were probably difficult for the HMM to figure out. A final problem was that it was difficult to judge the quality of the poems generated. So, what works and what doesn't work was difficult to determine.

Daniel wrote some parsing code as well as used an off-the-shelf implementation of unsupervised training for HMMs (nltk.tag.hmm) to look for the best number of hidden states, and wrote the code to create rhyming poems. Victor wrote most of the unsupervised training code, wrote a little preprocessing code, and also looked for the best number of hidden states. Eve wrote some parsing code, most of the visualization code, and most of the poem generation code.

2 Data Manipulation/Preprocessing

Given text files of Shakespeare and Spenser's sonnets, we parsed them into individual poems, which was represented as a list of its words in order. This seemed best since lines or bigger are too coarse a level of granularity (we don't want to just recite resorted Shakespeare lines), using syllables would cause us to generate imagined words, and parsing words could allow our model to capture parts of the structure such as the meter. We did end up writing a syllable parser anyways, but using it was soon abandoned.

In tokenizing as words, we also made a few other decisions. We kept hyphenated words together as a single token. After all, there must be a reason they are connected, and keeping them together may make our poem more meaningful/consistent. We kept punctuation with the words they come after. We wanted to keep these kinds of voice in the poem, and figuring out where to place them in poem generation would be difficult. Parentheses, however, were removed in preprocessing because the HMM cannot model opening and closing parentheses by itself. Initially, we did not take out parentheses and soon discovered that we got random closing parentheses and opening parentheses with no clear place to put closing counterparts when generating poems.

Other than straight up processing and tokenizing into words, we also did processing based on the number of unique words a poem had. In the function, parseTokLimMin() in parse.py, given a number of poems to search through from Shakespeare and Spenser and a total number n of poems to find, we tokenize only the n poems whose combination yields the fewest number of unique tokens. That is, if a combination of three particular poems uses 176 unique tokens in total, and all other combinations of three poems uses 177+ unique tokens, we choose the set of three poems with 176 unique tokens. The reason for this preprocessing is two-fold. The first is that if we have fewer unique tokens, then we have more data on each individual token, so that the HMM can better train on it. The second reason is that we are very constrained on training time, and having fewer tokens in general reduces the time needed to train.

3 Learning Algorithm

The major algorithm we had to implement was the Baum-Welch algorithm for unsupervised training of first-order HMMs. The Baum-Welch algorithm is a variant of the expectation-maximization algorithm for HMMs. Given training data in the form of unlabeled sequences, we first randomly initialize our transition matrix A and our observation matrix O, which together completely characterizes an HMM. We then run the expectation step, in which we calculate maximum-likelihood marginal probabilities using the forwards-backwards algorithm as a subroutine. In particular, the forwards algorithm calculates given A and O using a Viterbi-like dynamic programming algorithm the values

$$\alpha_Z(i) = \Pr[x^{1:i}, y^i = Z | A, O]$$

that is, the probability of observing the length-i prefix of a sequence x with the ith hidden state y^i being state Z. Similarly, the backwards algorithm is also a DP algorithm which calculates

$$\beta_Z^(i) = \Pr[x^{i:M}, y^i = Z | A, O]$$

that is, the probability of observing the length-$M - i$ suffix of sequence x with the ith hidden state being Z. Once we have these, we can calculate the marginal probabilities according to

$$\Pr[y^i = Z | x] = \frac{\alpha_Z(i)\beta_Z(i)}{\sum_{Z'} \alpha_{Z'}(i)\beta_{Z'}(i)}$$

$$\Pr[y^i = b, y^{i-1} = a|x] = \frac{\alpha_a(i-1)A_{a,b}O_{x^i,b}\beta_b(i)}{\sum_{a',b'}\alpha_{a'}(i-1)A_{a',b'}O_{x^i,b'}\beta_{b'}(i)}$$

Once we have these marginal probabilities, we can then perform a maximization step, in which we update A and O according to these marginals via

$$A_{a,b} = \frac{\sum_{j=1}^{N}\sum_{i=0}^{M_j}\Pr[y_j^i = b, y_j^{i+1} = a]}{\sum_{j=1}^{N}\sum_{i=0}^{M_j}\Pr[y_j^i = b]}$$

$$O_{w,z} = \frac{\sum_{j=1}^{N}\sum_{i=0}^{M_j}1_{[x_j^i=w]}\cdot\Pr[y_j^i = z]}{\sum_{j=1}^{N}\sum_{i=0}^{M_j}\Pr[y_j^i = z]}$$

We can then repeat this until it converges. For our implementation, we stopped when the sum of the Frobenius norms of the difference between the old and new A matrices and of the difference between the old and new O matrices was very small compared to the first algorithm iteration difference.

4 Naive Poem Generation

Using the trained A and O matrices, we can then generate a sequence in accordance with our trained HMM by randomly choosing a start state (according to the distribution described by the HMM), and then randomly choosing state transitions and token emissions in accordance with the probabilities described by the HMM. To enforce the 10-syllable per line structure, we used a tool to count the number of syllables and stop until we had 10 (although, due to the limitations of the tool, some lines may end up with 11 syllables).

[TODO: discuss more.]

5 Better Poem Generation

To create rhyming poems, we first used Shakespeare's and Spenser's sonnets to create a "rhyming dictionary", which internally looks like a list of lists, with each inner list represents a rhyme equivalence class; that is, every word in the list rhymes with every other word in the list. We then seeded the end of each line of our sonnet with matching pairs (if our HMM training set included these tokens) according to the rhyme scheme (Shakespeare's more relaxed $abab\ cdcd\ efef\ gg$ scheme), and then generated each line backwards in accordance with our trained HMM.

6 Model Selection

We didn't do anything too sophisticated to do model selection; we trained our HMMs with different numbers of hidden states, and then examined the poems and chose the number of hidden states which made the poems we thought were qualitatively best. Due to the time required to do unsupervised HMM training with large numbers of hidden states, when using all of the poems, we limited our search to numbers of hidden states below 17.

In the end, using hidden states between 8 and 17 didn't really seem to change much. The poems were all gibbrish. They could somewhat follow proper grammar, but meaning-wise, there was nothing. Meter seemed to be hit or miss. Due to the very random nature of the resulting poems, though, if the poem generation algorithm was run enough times, surely a great, creative poem could pop out.

In order to train on a large number of hidden states, we also used the preprocessing method described earlier to choose just a few poems (3-6 usually) and train on them with numbers of hidden states on the order of the number of unique tokens. One example is where we trained 3 poems with a total of 176 unique tokens using 176 hidden states. The results definitely made more sense and had better meter. However, the cost was that the generated poems had more phrases taken directly from the poems they were trained on. With the number of hidden states equal to the number of poems, however, this was not too much of a problem, as the size of the directly taken phrases seemed to cap at about 4 words. Most of the in the generated poem were still unique to the poem, though, and if phrases were taken, the most common size by far was 2 word phrases.

So, in the end, we mostly experimented on two extremes. One where there was a lot of data and few hidden states, and another where there was not much data and many hidden states. Time constraints would not allow us to test out the case of a lot of data and many hidden states. For these two extremes that we were able to examine though, the pros and cons can clearly be seen. If we want more "creativity," we should go for more data and less states (8-12 seemed fine). If we want to be more Shakespeare-like, we should go for more hidden states.

[TODO: Final choice of model?]

7 Visualization

State	Noun	Verb	Article
0	0.53	0.30	0.00
1	0.12	0.07	0.00
2	0.48	0.36	0.10
3	0.23	0.07	0.19
4	0.51	0.31	0.00
5	0.10	0.05	0.00
6	0.17	0.13	0.00
7	0.28	0.14	0.00

We trained essentially two kinds of HMMs: Part of speech modeling HMM and word modeling HMM. For the former, we trained with numbers of hidden states close to the number of parts of speech (8) in English. For the latter, we used large numbers of hidden states, equal to some fraction of the number of unique words present in the poems we used to train the HMM. For the 8 state HMM, we expected to see some states have high probability of emitting nouns, others verbs and particles. However, as demonstrated by the chart above, we saw multiple states have higher probabilities of emitting all three part of speech types compared to other states. This may be the result of the imperfect methods we use to visualize nouns and verbs however. We determined that there is no good way of determining if a single word, without any context, is a noun or a verb. The method we use is to check words against a large database of already tagged words (specifically nltk's wordnet corpus). However, the corpus we use only has 60k nouns and probably does not include a lot of the older terms used in these poems. Interestingly, state 3 in the chart has a much higher probability of emitting articles compared to other states, and lower probability of emitting nouns and verbs compared to other states. This may indicate that state 3 represents particles.

For the word modeling HMM, many of the states had 0 or 100 percent emission probability of a noun,

verb, or article. This makes sense if each state represents a particular word.

8 Conclusion

[TODO: discuss poem quality and stuff]