Shakespeare Bot 5000

Advith Chelikani, Loko Kung, Joon Hee Lee

March 2016

Note that we are using 19 of Advith's remaining 40 late hours on this assignment.

1 Unsupervized Learning

1.1 Introduction/Abstract

In order to implement a pipeline to process and learn a hidden Markov model (HMM) from a series of Shakespearean sonnets, we decided to build our architecture in a way such that the HMM model will remain constant and use modular preprocessing kits to allow for a variety of different tokenization and sequencing methods. The ultimate goal was to have a large selection of preprocessing methods that could all be fed in to the same model and yield results nonetheless. As our code moved closer to the final product, the modularity is sacrificed to produce better models and training sequences.

1.2 Preprocessing

1.2.1 Background Research

Before even writing preprocessing code, we did some manual research to select only the sonnets that followed the Shakespearean archetype, discarding any that deviated from the standard to produce a training set that was representative of our intended population. This meant that Sonnet 99 (15 lines), Sonnet 126 (12 lines), and Sonnet 145 (iambic tetrameter) were all removed from the training set. (Note that in the training file they still exists at the bottom but are not read in by our pipeline.)

1.2.2 Syllables and Meter

In order to take into account the number of syllables and the iconic iambic pentameter, we used an API client to an online dictionary in order to get the number of syllables for each word in our entire training set. We recorded all the syllable counts for the words that the API was able to recognize and proceeded to number the remainder of the words by hand. We then saved a map from the words to their counts so that we could easily access the values along the way.

To account for the stress in the iambic pentameter, we exploited the fact that every line in Shakespeare's sonnets alternate between stressed and unstressed syllables starting with an unstressed syllable. Since we could get the syllable counts from our dictionary above, we parsed through each line from the sonnets and assigned words to either stressed or unstressed for the first syllable. The first word of each line was always unstressed and from there on, we would add the number of the syllables of the word to get the parity of the next. Again, we took this data and saved it into a dictionary where 0 indicated unstressed and 1 indicated stressed.

1.2.3 Rhyming

Since learning to rhyme separated lines is impossible for a first-order Markov chain, we decided that if time-permitted, we would seed the poem generations with predefined rhyming words. Since the words must be in our model, we used the consistent rhyming scheme of Shakespeare's quatrains and couplets to build a list of words that rhymed with one another and would randomly pull from this list to generate the lines.

Initially, we separated the rhyming between quatrains and couplets when returning since this would allow us more leeway to test different structure sequencing with multiple models later on. After all, it is much easier to pool the two together if necessary rather than to separate them if they were pooled to start.

1.2.4 Tokenizers

Since we had to parse the data line by line, a series of tokenizer methods were created such that given a single line, they would tokenize the words in it accordingly. This way, we could then pass in the tokenizing method we wanted to use for the sequencing methods and allow for all permutations of tokenization. Although we initially considered tokenizing on syllables, we realized that not only would this generate "made-up" words that could be nonsensical but also make it difficult to split the words accurately. Because of this, we decided that doing so would hurt our model more than help and hence decided to tokenize on words.

Even though we decided to tokenize on words, we still needed to address how we handled punctuation and newline characters. The only obvious step was to standardize all lines to be lowercase first since we could easily add back capitalization. To handle the ending punctuations, we wrote our tokenizers to either attach the mark to the word to its left or remove them all entirely. Ultimately, though, we chose to use keep the punctuation attached to the words since adding punctuation after generation was unintuitive whereas the Markov model could potentially learn where words with punctuation should be generally placed.

For newline characters, we experimented with including newline characters either by themselves or with ending punctuation to see if the models would respond positively. Unfortunately, this just resulted in poems with random newline characters in between what we designated as lines so we decided against using this form of tokenization.

Finally, on the topic of tokenization, we decided to keep hyphenated words as a single word in our dictionary since we didn't want random words being hyphenated together to make nonsensical phrases. (This could, however, have been an interesting toy to mess with to generate silly poems.)

1.2.5 Sequencing

Similar to the tokenization, we were initially unsure of how we wanted to break the sonnets into training sequences, and as a result, implemented a series of different sequencings and tested on them afterwards. Some approaches we included were to sequence per sonnet, making each sonnet a single sequence, sequence by quatrain/couplets, making each sonnet into three quatrains and one couplet, and sequence by line making every line of each sonnet into a sequence.

Of the different sequencing methods, we found that sequencing on the entire sonnet made generation difficult since regulating aspects such as rhyming and meter proved to be difficult and ineffective. When splitting the text into couplets and quatrains, we found that the training set for couplets tended to be too small and as a result not as effective. Aside from just splitting poems into quatrains and couplets, we also tried to split them into a set of all lines of all quatrains and all lines of all couplets. Similarly, however, the training set for couplets was still smaller than we hoped.

Although we will elaborate more in detail about our implementation of the HMM in the following section, it is important to note that since the time it took to train our models was on the orders of days for anything greater than 20 states, we limited the number of sequences going into our training implementation to be

either a full quatrain or a full couplet (we trained two HMM's to generate a poem.)

When using third-party open-source HMM models where the training algorithms were much more optimized, we were able to train using larger input and for more states. As a result, we decided that since we want to generate each line and have the option of seeding the lines to rhyme that we wanted to train on a set of all lines. Therefore, we sequenced these inputs on a per-line basis.

Furthermore, since rhyming occurs at the end of a line, we found it hard to seed a model at the end of the line rather than the front. (We looked into Markov time reversal models and they are apparently not guaranteed to run like we would expect.) To resolve, this we tried training models on a per-line sequence input where each line is reversed to start. To clarify, by reversed, we mean that the order of the input words are reversed, not the actual words. This yielded models that generated lines from the last word back to the first which allowed us to seed easily. To display the poems, we simply reversed the lines again.

1.3 HMM Implementation

We implemented the standard Baum-Welch algorithm in Python using NumPy. The algorithm was implemented as a class called "HMM", whose main properties include A (transition matrix), O (emission matrix), PI (initial state vector), and token-dict (dictionary between integers and tokens).

Dynamic programming was a key part of our implementation that allowed for greater speeds. After randomly initializing A, O, and PI, the Forward-Backward algorithm was run to calculate all possibles alphas and betas. Using these values, "computeMarginals" was run to precompute all values of $P(y^j = b|y^{j-1} = a)$ (xi) and $P(x^j = w|y^j = a)$ (gamma). This precomputation meant that we did not have to calculate any of these values more than once in the maximization step. Finally, the function "update" was run to use these values of gammas and xis to update A, O, and PI.

To calculate the change in the matrices and initial state vector, we computed the Frobenius (for matrices) and L2 (for PI) norms of the differences between the updated and previous versions of these matrices and vector. These norms were then saved in an array, and the algorithm stopped when the ratio between the most recent norm and the first norm was less than a prespecified value of epsilon.

2 Visualization & Interpretation

When a Hidden Markov Model learns patterns in Shakespeare's texts, it is looking to find patterns in the data by seperating groups of words (or whatever your base building block is) into categories. Though not clearly delineated, these categories will often hold a specific property, be it about the similarity in meaning, length, or syllable count of their elements. We also see information about the relationship between the hidden states (categories), in the form of transition probabilities.

State 1	**State 2**	**State 3**	**State 4**	**State 5**
heats	cool	discased,	heart-inflaming	vowed
quenched	virgin	tripping	distempered	hied
sleeping	lying	boy	quickly	oaths'
love-kindling	drudge	trial	misuse	raised
prize,	abhor	rising	perjured	all-tyrant,
commanded	worship	marvel	fawn	sees
revenge	service	it?	longing	falsely
random	keep'st	correspondence	dying	sickly
fed,	desperate	me!	servant's	there's
divine	preserve	pray	costly	terms

State 6	**State 7**	**State 8**	**State 9**	**State 10**
votary	be.	nymphs	well-seeing	cools
cure,	thrall,	chaste	men's:	legions
thither	perpetual,	love-god	self-example	fire;
valley-fountain	disarmed.	seeting	woo	touch
dian's	warmed,	bed-vow	foolish	maid
cupid	guest.	stays	over-partial	enlighten
honest	desired,	nobler	beseem	warrantise
oaths	new-fired,	cheater	become	refuse
twenty?	cure:	urge	wires,	brightness
breach	endure,	frantic-mad	quest,	insufficiency

Above we see the top 10 words that associate with each hidden state. These words were found by taking the top 10 emission probabilities for each state (after normalization by frequency of occurrence) and mapping their indices to words. Let's try to investigate patterns in the data. With the help of the dictionary at http://www.shakespearwords.com, we were able to find the part of speech and definitions of unknown words like warrantise.

We found that in state 8, an overwhelming majority of the words were directly linked to beauty/sex/love. The two words without a direct connotation related to love, stays and frantic-mad, still could potentially have a love-related meaning in the right context.

In state 2, we found two sets of polarizing words; that is, one set of the words, namely virgin, worship, service, and preserve had a holy/divine connotation and the other set, namely lying, drudge, abhor, desperate had the opposite connotation. This suggests perhaps that state 2 sets the tone for the rest of the line. In a similar vein, in state 4, all the words except for perhaps servant's have a heavy connotation, meaning that the words are not neutral. In other words, they all immediately invoke an emotion or feeling within the reader and set a tone that the following words must follow.

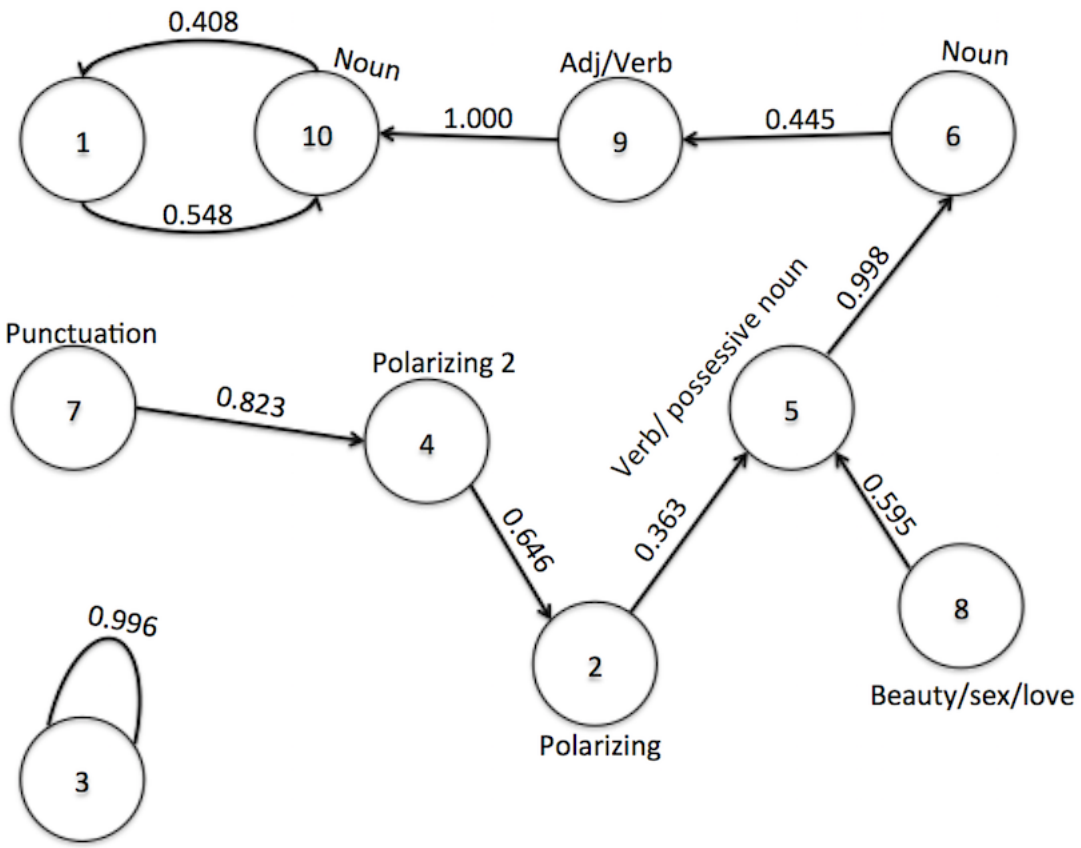
Looking at state 9, we see that all but two of the words are either adjectives or verbs. Similarily, in state 10 we see that all the words but enlighten and refuse are primarily nouns, with refuse also being used as a noun occasionally (sonnet 150) to mean rubbish. In state 7, we see that every single word ends with some sort of punctuation.

Let's also take a look at the most likely state-to-state transitions and with what probability they occur. These values were found simply by looking at the transition matrix and finding the max value and its index.

Transitions

$$P(1 \rightarrow 10) = 0.548$$

$$P(2 \to 5) = 0.363$$
$$P(3 \to 3) = 0.996$$
$$P(4 \to 2) = 0.646$$
$$P(5 \to 6) = 0.998$$
$$P(6 \to 9) = 0.445$$
$$P(7 \to 4) = 0.823$$
$$P(8 \to 5) = 0.595$$
$$P(9 \to 10) = 1.000$$
$$P(10 \to 1) = 0.408$$



Let's interpret these transition probabilities using our speculated state definitions from earlier. We see that the state 7 transitions with very high probability to state 4. Words in state 7 end with punctuation and words in state 4 are heavy connotation (non-neutral) words. This transition makes sense, because following punctuation, we need a heavy word to set the tone for the upcoming sentence or phrase.

State 9 transitions to state 10 with probability 1. As we pointed out earlier, state 9 is comprised of verbs and adjectives and state 10 is comprised eseentially entirely of nouns. This makes sense since often directly following a verb or adjective we find a noun. The noun may be the subject of the adjective's description or the object of the verb's action.

We also see that state 4 transitions to state 2 with high probability. Both these states have words loaded with meaning (non-neutral), hence chaning them together helps to set the tone for the rest of the line (or quatrain/couplet) and invoke emotion in the reader.

In this way we the HMM model helps us to analyize the relationship between groups of words and through it we can learn about the underlying nature of Shakespearean Sonnets.

3 Poetry Generation

3.1 Implemented HMM

3.1.1 Algorithm and Design

For the HMM that we implemented, since we were unable to train on a large input set (the set of all lines), we were restricted in the fanciness of the poems we could generate. Namely, since the models we trained using our own implementation split the data into quatrains and couplets and trained on forwards data (meaning we did not reverse the order of the words we trained on) we were unable to enforce rhyming for the lines. We were, however, able to enforce both syllable count and the iambic pentameter meter based on the maps we made in our preprocessing step.

Since we trained unsupervised models, we did not include the syllables or stressing of words in our training, however, we enforce these rules in our code to generate the poems. When we are generating a quatrain or couplet, we generate on a per line basis, however, since our training data used an entire quatrain/couplets as a single sequence, at the end of generating each line, we save the state that we ended on and start the next line from that state in order to reflect how we trained. For the first line, however, we decided to use the initial distribution outputted by our HMM implementation to randomly select a state to start from since randomly choosing a state potentially start us off in an undesirable state.

Just to be clear, we are generating poems using the models we implemented by randomly selecting a state to start weighted by the initial distribution returned by our model, choosing a word based on the emissions matrix of that state, transitioning to the next state, and repeating the process. Since we want each line to end once we have reached 10 syllables, we parameterize the length of the line by number of syllables and generate words until we have reached that length. We will discuss in more detail how we constrained the length of each line and the meter in the Additional Goals section. In order to generate poems with our implemented models, refer to the method `gpoem_qac_models` in the included source code for poem generation.

3.1.2 Examples

Here, we include two examples of poems generated using our implemented model which applies some restrictions that will be discussed in the Additional Goals section. We follow with a brief discussion of this algorithm and the poems generated by it.

In heart to whom did constant sun, of short
Conceit self praises gift, deep spirit give!
But hied idolatry, but broken, full
If guides to how upon of graciously
But look upon time's face: outworn thy tell
But vassal nothing crooked be which me
To best. Extern seeking autumn fresh make
My creation my lovely answer be,
Whence thou not praise. Not a removed proud that
True, every your ladies face the so
Do o'er brass my like an ornament
I many earth the physic blunt my shoot
But longer lov'st I young. Content, thy though
And tongue: doom strive right, heart's for wake ever

Nativity to full I beauties pipe
He thou the coward buds of dwell thus thou
As thou those muse but a self canker all
Doth sober in play to grace they gav'st
Making for fled, I flatterer my heart
Perfection white friend thou did born then be
Yellow whilst afterwards of most, than why
To every painting and a without
Whilst me which windows slumbers yet did buried,
Having thee, mine to soul the lays your thoughts
Reason the me. To every which thoughts
Lips, others' ceremony youthful kind
Since me to that and so of swear dreading
Shine father, all assure out-going it

The poem on the left was generated using a 20-state HMM while the poem on the right was generated using a 30-state HMM. Clearly, however, both poems sound like gibberish and follow almost no notable kind of consistent punctuation aside from capitalization which we enforce as a post-processing step for presentation purposes. Even with punctuation aside, the sentences fail to follow grammatical rules consistently. Although not shown here, the algorithm also produces random parenthesis without a corresponding closing or opening counterpart since we left those in during tokenization. As a result, it often takes a couple of generations in order to get a poem that doesn't have any unwanted parenthesis.

Looking at the bright side, however, we realize that our model was trained on a very small data set since we had to clump lines together as quatrains and couplets to get reasonable runtimes, and yet we still managed to capture some important aspects with a little help in the generation step, namely notice that the number of syllables for each line is consistently 10 and that, for the most part, each line follows the iconic iambic pentameter that Shakespeare used for his sonnets as well. All in all, it is clear that our model is indeed working and learning on the training set along with the generation process. If we were able to optimize our training more, we could have ran our training on larger inputs and generated more interesting poems.

3.2 Off-the-Shelf HMM

3.2.1 Algorithm and Design

With the off-the-shelf HMM implementation (hmmlearn), we were able to train on a per-line basis with each of the lines reversed so that we could seed it with rhyming words (more on this under Additional Goals). Instead of starting in a random state dictated by a weighted vector like we did for our implemented generations, we instead take a seeded value (the word that we want the line to end with) and find the state that is most probable to emit that word. We then take that to be the start state and generate each line.

Since we have a Markov chain that is backwards, for each word we generate, instead of placing it at the end of the line, we instead put it at the front and add each new word to the front to generate our line. We used some of the same restrictions we talked about in our implementation (again more on this under Addition Goals) to limit the syllables per line. Unfortunately, we realized that if we try to seed, limit the syllables per line, and enforce the iambic pentameter, the program often stalls and never produces anything because there simple aren't that many combinations of words under our model that fit all the criterion. As a result, we decided to allow deviation from iambic pentameter. Finally, in order to produce the rhymes, we are randomly pulling from the pool of rhyming pairs we computed in the preprocessing step. By randomly,

we just mean we have them all in a list and we randomly select one.

3.2.2 Examples

The following two examples were generated using the models that we generated with off-the-shelf implementations. Notice that the poems already include many of the Additional Goals.

Three-fold will thy love, should dart shouldst sweetness,
Spirit who on for till I in the sweet,
I fair and I 'tis are moon of meetness,
Now may I two she in say of love meet,
My charge? Griefs tell be worth that my doth spite,
One, but no hawks which in bosoms set,
Drawn my beauties with public I delight.
That which who eye on crooked carcanet.
My friend, thou blame world if from writers well?
In the own sweet actor for that it art
Had may thee if to that bloody your tell.
Mine frank why in gates my fair things convert:
Is in to have wilt, when I make contains,
Eyes not upon and heaven's my remains.

Lives to follow it clouds thou in great old,
That thou the by needs at why that the store;
To give compass themselves supposed told:
Yet by allege wilt love not 'twixt mightst more,
Art which thou how when pomp, every grow'st,
Bends when the kiss religious indigest,
'gainst child, think it lodged can idly bestow'st,
Admit think poor me to voice sinful best
To with wilful love should feel do cloak,
Gardens thy then was remover the deeds,
Fair, beauty a thy faults of summer smoke?
From the wooing my gifts wrinkles exceeds?
Looking yet as it as for of doth sit,
Say so, at profitless were love on it.

The poem on the left was generated with 30 states while the one on the right was generated with 10. With the backwards seeding and rhyming included, we also notice that much of the punctuation has been smoothed out, namely we don't have random punctuation appearing in the middle of lines anymore. Even though we are no longer restricting the meter of the poems, we see that the stress and unstressed syllable still stand every once in a while indicating that our models are actually learning the pattern even if it is in the slightest way. Finally, although the poems are still mostly gibberish, notice that every once in a while there is a string of words that follow common parts-of-speech, indicating that our models are flawed, but not just randomly generating word after word.

4 Additional Goals

4.1 Improvements

4.1.1 Meter

Since the HMM models were all first-order only, it was essentially impossible for them to learn the number of syllables per line since we did not use some sort of end-of-line indicator when training. Even if we had such an indicator, termination of each line is not guaranteed to occur when it should. As a result, we decided to implement restrictions aside from the models. In order to maintain the number of syllables, we generate each line over and over again until we find a line that matches the correct syllable count. We do this by simply generating full lines over and over again in a while loop where the breaking conditions are that the line match up to our restrictions. We implemented the stressing and unstressing of syllables in the same way. Only when both conditions are satisfied do we proceed to generating the next line.

Since we have two different generation algorithms, we will first talk about the one we used for the models we generated with our implementation. Since we trained on sequences which were entire quatrains or entire

couplets without newlines, we realized that generating line by line using the initial weighted distribution to get a starting state would be inconsistent. As a result, for each quatrain/couplet, we get the start state randomly from the distribution for the first line then proceed to pass on the current state we were in at the end of the line (once we have met the criterion) to the next line and start on that state instead. By doing this, we were able to keep the flow of the quatrains and couplets alike.

For the other algorithm, since our models were trained on a per-line basis, we did not have to worry about carrying states over to the next line. More importantly, since rhyming was our target for this algorithm, instead of concerning ourselves with an initial distribution, we parameterized a seed word instead. To find a starting state for each line, we look for the state with the highest probability of emitting the seed word and start on that state. Since we were seeding words and forcing initial start states in that sense, we realized that applying the same restrictions as above often caused our algorithm to stall and produce no results. With further testing, we found that it was just very unlikely for all conditions to be matched when putting so much constraint of such a small model. As a result, we decided that as a proof of concept, we would omit the iambic pentameter from the restrictions when generating rhymed poems.

4.1.2 Rhyming

Initially, we were unable to get our poems to rhyme shown by the examples under our implementation. However, to improve the quality of our poems, we rewrote much of our poem generation algorithm to allow for seeding of rhyming words. We also had to train entirely new models in order to use our new algorithm since we wanted to generate lines backwards, i.e. generate last word of like then second to last and so on. From the preprocessing step, we had already created a list of pairs of words which Shakespeare himself used as rhymes. Using this, we were able to pairwise-seed lines in the format of Shakespeare's rhyme scheme to generate lines that rhymed. In order to maintain more of Shakespeare's voice, we also chose to keep the rhymes between quatrains and couplets separate so that upon generation, the algorithm was more likely to produce lines more appropriately.

4.1.3 Hurdles, Other Issues, and Conclusion

For the poem generation, hurdles that we would have liked to overcome include better handling of punctuation (especially parenthesis which often result in the need for regeneration), improvement on consistency of meter, and improving the overall coherence of the poems. For the coherence of the poems, we realize that the main issue we would need to address would be to supply more training data. Since we are seeding in order to get rhyming anyways, we could have tried to train on snippets of other Shakespearean works just to keep his voice and sentence structure. This would also produce more variation in the text and possibly allow for constraints on meter along with seeding for rhymes.

Overall, in our poem generating stage, we tried a couple of different models with 5, 8, 10, 20, 30, 50, 75, and even 100 states just to see if they would perform better. Unfortunately, since the generated poems are all pretty incohesive there was no real way to compare which models performed better compared to one another. When we tried different number of hidden states, we had expected to be trading creativity for better poetry, however, it seemed that since we were only training per line, structure was not the priority in what our models learned. This was also the limiting factor in the lack of coherence in the sentence structures.

In summary, our models managed to generate Shakespearean sonnets that definitely retains his original style and most of his structure. We tried different ways to improve the models to make the poems more realistic and meaningful, including the addition of rhyming and meters copied from Shakespeare's foundations. The poems often fail to handle punctuation correctly due to a sub-optimal tokenization process with respect to punctuation, but manages to produce relatively reasonable sentences when seeded with rhymes. Finally, since our models often stalled when trying to require all constaints, we sacrificed meter and rhyming interchangeably to produce two different set of results.