**EE155: Shakespeare Bot 5000**

*Akshta Athawale, Mannat Singh and Miguel Aroca-Ouellette*

**1. Overview**

For the development of our Shakespeare Bot 5000 our team chose to take an iterative approach, wherein the first goal was functioning preprocessing, HMM training and Naïve Generation upon which incremental improvements were built. The incremental improvements first involved using visualization and qualitative sonnet analysis to determine an appropriate number of hidden states. Following this we added further preprocessing and more complex poem generation which used rhyme, meter and parts of sentence tags to generate a Shakespearian sonnet. Each step was tuned incrementally and qualitative analysis showed that each had an effect in generating a better poem.

We have three members in our team, Akshta Athawale, Mannat Singh and Miguel Aroca-Ouellette. Akshta was in charge of visualization and interpretation. Mannat was in charge of the unsupervised learning and HMM implementation. Miguel was in charge of preprocessing, poem generation and the additional goals. However, there was significant collaboration each task across the team, particularly with regards to debugging and qualitative sonnet analysis.

**Unsupervised Learning**

**2.1 Preprocessing**

Our team chose to tokenize the dataset by word, where a singular sequence would be represented by a line of a poem and each type of stanza (quatrain, volta and couplet) maintains a separate list of tokens. This approach was chosen as Shakespeare uses different intonation and language in each of his stanzas. The quatrain is the body of the poem and introduces the theme or story with lots of adjectives describing the subject of the poem. The quatrains maintain a consistent mood through, except for the volta, which usually represents a change in mood. This is grammatically demarcated by particularly strong adjectives and adverbs, as they have to provide a counterpoint in 4 lines to the 8 of the previous quatrains. By separating the volta from the other quatrains our team hoped to capture this shift in mood, or at least the shift in intensity. Similarly, although the couplet does not bring the same intensity as the volta, it does use different language from the rest of the poem as Shakespeare is bringing a close to the story he has presented. It usually starts with a conjunction (i.e. “or”, “but”, “yet”, “then”) and instead of posing the questions or accusations of the previous lines, it presents a statement or final action. Once again, by separating the couplet from the other stanzas, we hope to capture his literary voice.

Regarding punctuation our team took a simple, but effective approach. When tokenizing the data, all punctuation was stripped except for apostrophes “ ‘ “ and hyphens “-“.This allowed for words implying possession and plural words to stay separate, which should be beneficial as such words appear in different contexts and with different meaning. More importantly, Shakespeare often uses apostrophes within a word as alternate spelling or to skip a syllable, we wanted to keep these words intact and readable when generating our poems; for example, “consum’st” and “murd’rous”.

Hyphens were kept as the hyphenization of two words yields an entirely meaning than the non-hyphenated word, and treating the two words as separate would provide a significantly different meaning, particularly in Shakesperian English; for example, “all-eating” means self consuming, while “all eating” means everyone eats. (Maybe change) Punctuation was then added back in during poem generation.

During preprocessing, we also used the line endings and the known rhyming scheme (*abab cdcd efef gg*) to create a rhyming dictionary which was later used by our poem generation algorithm to ensure rhymes at the end of the lines.

Explain choices

Explain initial choices

Explain what changed

Final choice

Dataset Analysis

**2.2 Unsupervised Learning**

In order to perform unsupervised learning we used the Baum-Welch algorithm as laid out in the lecture slides and HMM notes. The stopping condition was the convergence of the norms of both the state transition and the observation matrix.

How did you choose hidden states

How did you tokenize words

How did you split up data into different sequences

**3. Visualization and Interpretation**

**4. Poetry Generation**

To generate our poems we use three trained HMMs, one for each stanza type, and individually generate each stanza. The stanzas were then combined in the correct order to generate our Shakesperian sonnet. The first letter of each line was capitalized and punctuation at the end of the line was added randomly; with a high probability simply a comma was added, and with a small probability an exclamation mark, question mark or semicolon was added.

Describe poetry generation

Include Naïve sonnet

Comment on quality

* Rhyme, thythm and syllable count
* Does poem make sense?
* Maintain voice?
* How does different number of states affect generation (qualitative)

For good qualitie discuss how HMM was able to capture these

Include non-Naïve sonnet

Compare

**5. Additional Goals**

**5.1 Rhyme**

In order to add rhyming to our Shakespearian sonnets, during preprocessing we generated a rhyming dictionary. In addition, every line of each poem was reversed before being processed by the HMM training algorithm, this allowed us to model reverse transition probabilities (reading sentence right to left) instead of forward transition probabilities (reading sentence left to right).

Using these new state transition and observation matrices we could then generate each line of the poem backwards. This was done by randomly selecting a word with a rhyme in our rhyming dictionary and using that as the seed for the generation of our poem line. If the line, was a rhyming line (i.e. the second *a* rhyme line), then the seed was selected from one of the rhymes of its previous rhyming line (first *a* rhyme line). This was done similarly for each rhyming scheme.

This ended up working very well and yielded much nicer sonnets than the non-rhyming approach. These sonnets also matched much more closely the sonnets written by Shakespeare since the rhyming scheme was an important characteristic of his approach. The only limitation this imposed is that the final word of each line (the seed word in the line generation) had to come from the rhyming dictionary. We felt that although this was a little limiting, since the dictionaries were so large it did not limit the creativity of our Shakespeare Bot too heavily. In the future it would be beneficial to generate rhymes for many of the words in the poems. Perhaps from an external rhyming dictionary source. This would allow for a larger variety in the rhyming schemes and greater variety in each line since there are more possible seed words. See below for an example of a sonnet generated once we added rhyming.

(Rhyming sonnet)

**5.2 Meter**

One of issues we noticed with the naïve poem generation was that without control over the meter, the generated sonnets did not match particularly well the linguistic flow of Shakespeare’s sonnets. In addition, often our lines were simply too long and wordy as our line ending condition was based on word count or transition to an end state.

In order to remedy these problems we decided to incorporate syllable counting. Syllable counting allowed us to generate sentences which roughly 10 syllables (as a Shakespearian sonnet should have), by using the syllable count as a new ending condition for our sonnet line generation. Syllable counting was done using the python *NLTK* [1]library, as well as *pyHyphen* [2] for words that are not present in the *NLTK* library. Although the syllable counting is not 100% accurate, it was close enough for our purpose.

Adding syllable counting also allowed us to keep track of the current syllable stress within a poem when parsing the training poem lines. Using this information, as well as syllable stress information from the *NLTK* library, we were able to provide an estimate of the stress for each syllable for each word; combining the syllable counting stress estimate as well as the *NLTK* stress estimate allowed us to show the multiple possible syllable stresses that each word could (i.e. content as in happy, or content as in material).

When generating each line of our sonnets, we ensured that each subsequent word had the correct starting (or ending when generating backwards) syllable stress in order to maintain the expected Shakespearian meter. This was done by pruning all the words which did not have the appropriate ending or starting stress syllable and re-normalizing a copy of the corresponding column of the observation matrix over this new, controlled distribution.

This yielded particularly large advantages in imitating Shakespeare’s style. Although it was slightly limiting in terms of creativity, since we were on average only pruning ~50% of each word from the observation distribution we still had a large subset of words to choose from. This was in some sense actually a good feature as in guaranteed variety of words within a sentence and avoided repeated highly probably words.

**5.3 Additional Texts**

One of the things we learnt from the Kaggle project was that having a larger dataset is often beneficial. There was not a specific problem we were trying to solve with this addition to our Shakespeare Bot, but we felt that perhaps the larger dataset and rhyming dictionary would be beneficial in adding creativity to our poetry generation and counteract some of the limitations imposed by rhyming and meter.

We added the Edmund Spenser dataset as well as sonnets we fetched ourselves from some of Shakespeare’s contemporaries which abided by the Shakespearian sonnet structure; these included John Keats, John Clare and Robert Frost. The result of our expanded sonnet dataset can be found below.

**5.4 Part Of Speech (POS) Restraints**

We noticed that one of the grammatical difficulties which we were sometimes running into with the sonnet was repeated part of speech (POS) tags. For example, in the generated sentence “Level *inspire* *convert* device perish”, both “inspire” and “convert” are verbs and it is grammatically quite rare for this to happen; one exception would be the sentence “He *began* *working* on his project”. As such, as a general rule we decided to enforce that POS tags of one type could not be repeated sequentially; i.e. a verb could not follow a verb. POS tagging was done using the *NLTK* library and the data was then stored in a dictionary for quick retrieval. Many of the tags provided by the *NLTK* library were too descriptive for our purpose, so they were grouped into broader tags; i.e. “Determiner”, “Predeterminer” and “Wh-determiner” where all grouped under “Determiner”.

This approach yielded favorable results and qualitatively generated grammatically correct sentences more often than before. In addition, it had the added benefit of adding more variety within the poems as highly probably word sequences were now broken by the POS tag of the current line’s randomly generated seed word. Since there are so many words for each POS tag this did not noticeably affect the creativity of our poem generation algorithm.

* What were you trying to fix
* How did you attempt to fix it
* Why did you think what you tried would work?
* Did it succeed?
* If not why?
* What tradeoffs in quality and creativity did this change cause

**6. Conclusion**

In conclusion, our team was successful in implementing a Shakespeare Bot which produced Shakespearian sonnets of a reasonable quality. We took several steps to improve upon the basic HMM implementation and naïve poem generation, and each modification had a visibly positive effect on the quality of the poems. Rhyme scheme and meter were the most important in replicating the Shakespearian style, however controlling for POS was the most important in improving the reliability of generating grammatically correct sentences.

**7. References**

|  |  |
| --- | --- |
| [1] | NLTK Project, "Natural Language Toolkit," 03 03 2016. [Online]. Available: http://www.nltk.org/. [Accessed 10 03 2016]. |
| [2] | D. Leo, "PyHyphen 2.0.5," 2014. [Online]. Available: https://pypi.python.org/pypi/PyHyphen. [Accessed 10 03 2016]. |