CS 155 Miniproject 2: JustTryIt Team Report

Kangchen Bai, Pengchuan Zhang, Yerong Li

1 Overview

2 Data Manipulation

The basic strategy in our model training is to train multiple models for different part of the poem. Therefore, we have the following pre-processing data manipulation.

Grouping Shakespear's sonnets enjoy the clear rhyme scheme *abab cdcd efef gg*. Moreover, lines with different rhymes have quite different sentence structure. For example, the first and third lines *aa* have quite different sentence structure from the last two lines *gg*. Based on this obersevation, we decide to train 6 models for these 6 parts, namely *a, b, c, d, e, f, g*. Therefore, we first group all the lines in the same part of the poems and get six corpuses, namely `groupA, groupB, groupC, groupD, groupE, groupF, groupG`.

Punctuations Shakespear's sonnets contain various punctuations (e.g., " ,", " '", " -"). We delete all punctuations except " -", and thus the words "Feed'st" and "'This" become "Feedst" and "This". For words with hyphen " -", we manually delete it or replace it with empty space " ". There are in total 83 hyphens in `Shakespear.txt` and it is very easy to deal with hyphens manually. After this stage, we have six corpuses and every corpus contains hundreds of lines without any punctuations. In poetry generation, we took our own punctuation scheme based on the writing style of Shakespeare's and our punctuations in each line of the poem are generated **automatically** by programs.

Tokenization We tokenize the words as features and use the method `text.CountVectorizer` from `sklearn.feature_extraction` to preprocess every corpus. In simple, `CountVectorizer` lowercases all the words and builds a dictionary between the words and the natural numbers $\mathbb{N}$. The output of this tokenization step is six corpuses with sequences of natural numbers. These six corpuses will be the input of our model-training algorithms, e.g., HHM and 2nd-order Markov model.

To achieve better poem-generating performance, we generate each line in the reverse direction with pre-sampled rhyming ending words. We keep generating lines until we get a line with exactly 10 syllables. In order to achieve these additional goals, we have the following pre-processing data manipulation.

Generating rhyming dictionary We use the *NLTK* package and the RhymeBrain website `http://rhymebrain.com/en` to build a dictionary for rhyming words. For more details, see Section 6. NEED SOME TEXT Briefly summarize how we are using this dictionary.

Counting syllables in each word We use the *NLTK* package, the *PyHyphen* package and our own-written function `count_syllables()` to count the number of syllables in each word. These three methods have their own advantages and disadvantages, and we combine them to get the most accurate syllables-counting. For more details, see Section 5.

3 Unsupervised Learning

We worked on Hidden Markov Model and Markov Model in this project.

Hidden Markov Model

In training HMM, we tried on several number of hidden state in our model and chose the numjber of state with the highest emission probabilities as the favorate model in the project. To be specific, we tried on models with 5, 10, 20 , 40, 80, 100, 500, 1000. We are working on model with 1000 hidden states because there are around 3000 words in Shakespeare's poetry set. Usually, the way to choose the right number of hidden state is to do cross validation: train a model on one part of the corpus (training set) and calculate the emitting probability of the other half of the corpus (testing set) and choose the number that can maximize the emitting probability of the testing set. But since the fact that most of the words in the corpus appears only once, we abandon the cross validation approach. We choose the best hidden state number by inspecting the quality of the generated sentences. We decide that the larger the number is, the better the generated poem will be. We will elaborate this problem at a later section.

Markov Model

1st order Markov Chain Model

The relationship between HMM and original Markov Chain Model We see in history that Hidden Markov Model is necessary when the number of observation states (number of words in Shakespeare's Sonnets, in this case) is unavoidable large. Besides this, we need Hidden Markov Model because we are supposed to get some intuitions in the grouping of words and hidden states are bringing us information. But in the case that the number of different words is affordably large (there are around 3000 words in this case), Markov Chain proves a more sophisticated model.

With this consideration and with the purpose of generating more reasonable verse set, we also worked on **Markov Chain model** in this project. In the basic (1st) Markov Chain Model the joint probability is given by

$$p(x_{1:M}) = p(x_1)p(x_2|x_1)p(x_3|x_2)...p(x_M|x_{M-1}) = p(x_1)\prod_{m=2}^{M} p(x_i|x_{i-1}) \tag{1}$$

But when we first get our trial on this **first order Markov Chain Model**, it does not give us perspective result, because this is extremely similar to what we have done in our **(1st order) Hidden Markov Model**, as what we have stated above, instead of tokenized the words into phrases, we tried the second order model instead, for the simple reason that this will do the tokenization automatically and is much more subjective in tokenization.

2nd order Markov Chain Model

In the **second order Markov Chain Model**, the assumption on the transition probability is:

$$p(x_m|x_{1:m-1}) = p(x_m|x_{m-1}, x_{m-1}) \tag{2}$$

So, different from the first order model, the joint probability in the 2nd order Markov Model gives:

$$p(x_{1:M}) = p(x_1, x_2)p(x_3|x_2, x_1)p(x_4|x_3, x_2)...p(x_M|x_{M-1}, x_{M-2}) = p(x_1, x_2) \prod_{m=3}^{M} p(x_m|x_{m-1}, x_{m-2})) \quad (3)$$

What should be mentioned is that we do counting and normalilzation for computing the piror probabilities $p(x_1x_2)$ and trains on the transition probabilities $p(x_m|x_{m-1})$. Since we have around three thousand words in Shakespear's Sonnet, we the number of parameters (for $p(x_1x_2)$ and $p(x_m|x_{m-1})$) is not substantially large, so the running time for 2nd order Markov Chain Model in affordable.

Notes on some trials and improvements

Here are some of the trials we have worked on in data manipulation and training:

- NEED SOME TEXT

4 Visualization and Interpretation for HMM

Number of hidden states

F irst, as we increase the number of hidden states, the quality of the generated sentences is increasing. We interpret this phenomenon as such: when the number of hidden states is small, some very different word have to be put into the same state. When generating sequences, not all word pairs under consecutive hidden states are meaningful(actually only a small number of pairs are). If we increase the number of hidden states to be almost equal to the number of words in the corpus, then usual word pairs are easily captured . Another benefit of increasing number of hidden states is that the sentence length is tending to have less variation. We randomly generate 1000 sentences when the number of hidden states equals 5, 100 and 1000 and plot the generated sentence length distribution. We find that as the number hidden states increases, the sentence length is more centered around 8-9, which is the typical length of a line in sonnet.

5pt

Interpreting the hidden states

H ere we also present our findings about the interpretation of the hidden states. The hidden states should bear some meanings in a sense that words likely to be emitted from the same hidden states should more or less share some common features. We find that the words pertaining to different hidden state may have difference in their part of speech tag, syllables and positions in a sentence.

For simplicity, in the following discussion we are always using 5 hidden states example. We plot the transition diagram of the 5 hidden states. From that we can see there is a close connection between hidden states and the position in a sentence. For each sentence in the training corpus, we use Viterbi algorithm to find the most likely corresponding hidden state sequences.We count how many times a certain state appears in a certain position of a sentence. We plot out the distribution of the occurring position of the 5 hidden states in those maximum likelihood sequences. It is clear that each of the 5 hidden states have peaks in different positions of a sentence. State 2 is most likely to be the starting word of a sentence while state 4 end of a sentence.

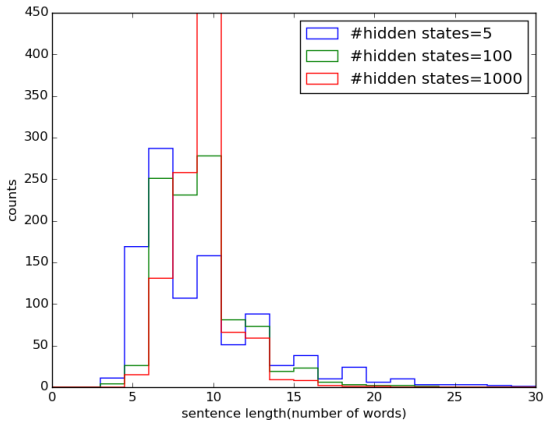

Figure 1: Sentence length distribution with different number of hidden states. When number of hidden states=1000, the length of the sentences are centered arount 8-9.

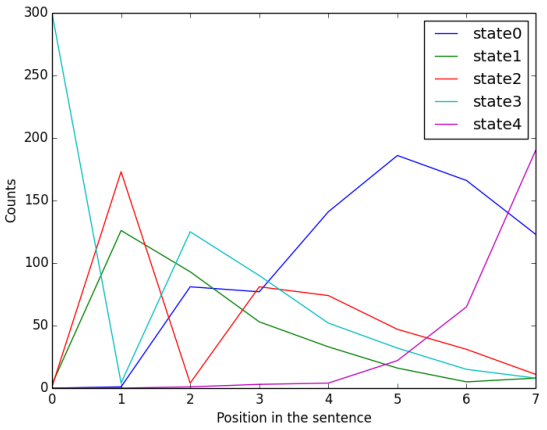

Figure 2: The distribution of different hidden states showing up at different positions in a sentence. Here we assume that a sentence is made of 8 words. If the actual length is greater than 8, then the later positions are also categorized as position 7.

Words pertaining to the 5 hidden states also have different distribution in their corresponding part of speech tags. For each sentence in the corpus, we use the part of speech tags marking tools provided by NLTK package to see the part of speech tag each hidden state is corresponding to and then make a histogram of how many times a hidden states appears in a sentence as a noun, verb, adj, adv,. We find that state 1 is most likely to appear as a noun while state 2 is more likely to appear as verb. Conjunction words like and ,or are more likely to be under state 2 and 3 while state 4, usually at the end of a sentence, is limited to noun and verb.

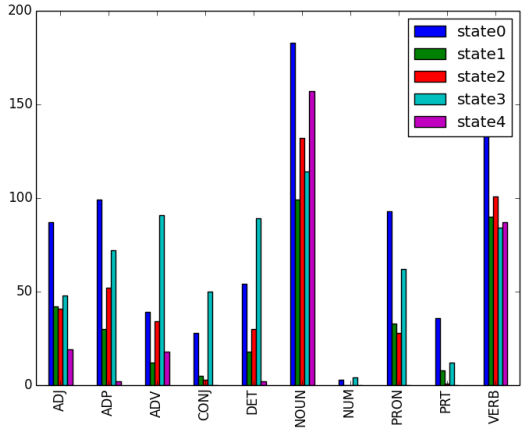

Figure 3: Statistics of corresponding part of speech tag for each state in the maximum likelyhood sequence given by viterbi algorithm

The number of syllables in each state is not so different. We choose the most probable 200 word that is emitted by each of the 5 states and count their number of syllables. The emitting probability has been normalized by word frequency. We didnt find significant differences between each state.

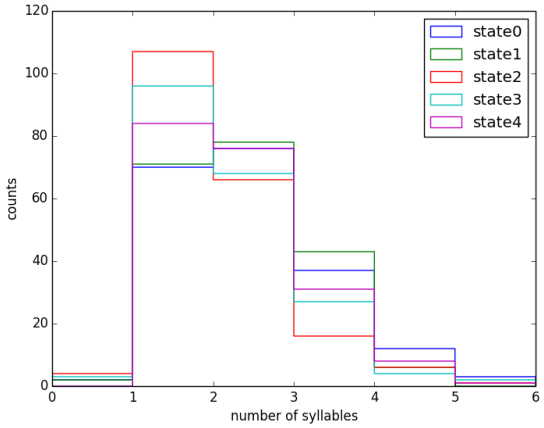

Figure 4: Number of syllables in the most probable 200 emitting words for each hidden state.

To make a conclusion, the five states has clear meaning representing the position in a sentence. It also has some relationship to the part of speech tags. But the two effects are not independent. It might be that a certain state have to be the first state in a sentence so that it prefers some part of speech tags rather than others.

5 Poetry Generation

We present results from the models we worked on in this project. As stated above, we do counting in the poetry generation to make sure that each line in our poem consists of 10 syallables.That is to say, we actually generation our poetry line by line, at each position of line, we repeated generate lines until we get a 10-syllable line: we only took our pick of lines with 10 syllables. In counting the syllables, we use dictionary from *NLTK* and package *PyHyphen* to break words into syllables, we did not truncate lines during the counting, so each line is supposed to end up in the *END* state (this is the same for both Hidden Markov Model and Hidden Markov Model). We only took our pick at sentence level.

1st Hidden Markov Model
2nd order Markov Chain Model

Here is a poem we generated from our reversed-trained 2nd order Markov Model, with automatically marked punctuation, we name it **'Hope or Fear?'**:

Hope or Fear?

Applying fears to hopes, and hopes to fears,
The worser spirit woman coloured ill,
What potions have drunk of siren tears,
Thus far for love my love suit sweet fulfil:
The mortal moon hath her eclipse endured,
Deaths second self that seals up all in rest;
Incertainties now crown themselves assured,
On both sides thus is simple truth suppressed.
And by this will be gainer too,
And all things turns, to fair that eyes can see,
How can it how can loves eye be true,
Thy proud hearts slave and vassal wretch to be?
Thine by thy beauty tempting her to thee,
So long lives this and this shall ever be.

6 Additional researches that we have worked on

Rhyme

The naive approach to generate poems does not honor the rhyme pattern in the Shakespear's sonnets. However, it is actually not difficult to introduce rhyme in our group-based poem generating algorithms. There are two steps to generate poems with rhymes: first, build a rhyming dictionary; second, seed the end of the line with words that rhyme, and then do HMM generation in the reverse direction.

To build a rhyming dictionary, we pick out the last words of pair of rhyming lines. If these two words rhyme, we add it to the rhyming dictionary. For example, "increase" and "decease" rhyme with the same phonetic "IY-S" (`cmudict` phonetic form), and thus we generate an phonetic item "IY-S" in the rhyming dictionary with two words "increase" and "decease". In Sonnet 11, we find another pair "increase" and

"cease" also rhyme with "IY-S". Then, we add "cease" into the item "IY-S". Sometimes, the last words of pair of rhyming lines do not rhyme, like "die" and "memory". In this case, we only check these two words separately whether they can be added into some existing phonetic item. For example, "die" can be added into the phonetic item "AY" and "memory" can be added into the phonetic item "IY". After traversing all the rhyming lines twice, we build a rhyming dictionary for each corpus.

We combine the *NLTK* package and the RhymeBrain website `http://rhymebrain.com/en` to identify whether two words rhyme or not and the phonetic they rhyme. For the word which exists in `cmudict` (in *NLTK*), we use *NLTK* to get its phonetics. For example,

```
>>> phondict = nltk.corpus.cmudict.dict()
>>> phondict['increase']
[[u'IH0', u'N', u'K', u'R', u'IY1', u'S'], [u'IH1', u'N', u'K', u'R', u'IY2', u'S']]
```

If either of the pronunciation rhymes with other word, like "cease", we think that these two words rhyme. For the word which does not exists in `cmudict`, our script *automatically* picks an auxiliary word which rhyme with this word from the RhymeBrain website `http://rhymebrain.com/en`, and use the phonetics of the auxiliary word to analyze the rhyme. For example, "fulfil" does not exists in `cmudict`. Our script will go to the RhymeBrain website and pick the word "foothill". Then we use

```
>>> phondict['foothill']
[[u'F', u'UH1', u'T', u'HH', u'IH2', u'L']]
```

to analyze whether "fulfil" and "will" rhyme, and the answer is yes!

In this case, we train the HMM or 2rd-order Markov model in the reverse direction. To do this, we only need to reverse every line in the input corpus. To generate a poem, we first seed the end of the line with words that rhyme, and then generate lines with the reverse-direction-trained model. The following are several rhyming lines generated by trained HMM for `groupG` with number of hidden state 80.

```
as with proves replete in thee writ,
even see shall accessary used must find and herself enfeebled mine it,
she this and thee praise,
then love away night seat is one days:
this even had in lived their young part,
yet subjects knife what right winter thee heart;
and thou feelst find bear wretchcd your store line,
that other not may it of shows this mine in writ mine.
pity mayst be you made to praise best,
the thee be him and length time thou am still breast,
```

We can see that the lines alway rhyme, but the number of syllables varies a lot.

Controlling the total number of syllables in a line

There are several ways to control the total number of syllables in each line and ideally to make it exactly 10. We take a very simple approach to do this: repeatedly generating lines until the total number of syllables

is 10. Sometimes, it takes a long time to get a line with exactly 10 syllables. Therefore, we randomly generate at most 50 lines, and keep the line whose total number of syllables is closest to 10.

To count the total number of syllables in each line, we need to count the number of syllables in each word. We combine the *NLTK* package, the *PyHyphen* package and our own-written function `count_syllables()` to count the number of syllables in each word as accurate as possible. If a word is in `cmudict`, the *NLTK* gives us the right answer. For examples, "increase" has 2 syllables according to `cmudict`. If a word is not in `cmudict`, we use *PyHyphen* to count the syllables. For example,

```
In[4]: from hyphen import Hyphenator
In[5]: h_en = Hyphenator('en_US')
In[6]: len(h_en.syllables(unicode('fulfil')))
Out[6]: 2
In[7]: len(h_en.syllables(unicode('air')))
Out[7]: 0
```

We can see that the *PyHyphen* package is not so accurate to identify the number of syllables in a word. Therefore, we also write our own function `count_syllables()` (see file `countvowel.py`) to correct possible mistakes made by the *PyHyphen* package.

With this approach to control the total number of syllables in a line, we get rhyming lines all of which have total number of syllables nearly 10. The following are several rhyming lines generated by trained 2rd-order Markov model for `groupG`.

```
which die for goodness who have lived for crime,
but were some child of yours alive that time,
to give back again and straight grow sad,
this told joy but then no longer glad:
lo thus by day my limbs by night my mind,
for thee and for my name thy love and am blind;
think all but one and me most wretchcd make,
till then not show my head where thou mayst take,
so till the judgment that your self arise?
so long lives this and dwell in lovers eyes,
```

We can see that the lines alway rhyme, and the total number of syllables in each line is nearly 10.

Incorporating additional texts

Our framework enables us to train our models with additional texts. We include all 139 of Spenser?s sonnets in our training datasets. With the same process, i.e., pre-processing, rhyme dictionary learning, model training (for both HMM and 2rd-order Markov model), we can easy get models which have a larger dictionary. The training time nearly get doubled because we have nearly double sized training data. The following is one poem from our trained 2rd-order Markov model, with rhyming and controlling-the-total-number-of-syllables.

7 Conclusion