CS 155 Miniproject 2: JustTryIt Team Report

Kangchen Bai, Pengchuan Zhang, Yerong Li

1 Overview

In the project, we work on the **Hidden Markov Model (HHM)** and the **Markov model** for poem generations.

Due to the clear rhyme pattern in Shakespeare's sonnets, the basic strategy in our model training is to train multiple models for different part of the poem. We tokenize the words as features in each group and run the EM algorithm with different number of hidden states to train the HMMs. We developed several improvement techniques to get better poem generating performance: we train the HMM in the reverse direction and learn rhyme dictionary to sample last words that rhyme; we generate lines with total number of syllables as close to 10 as possible. With these techniques, we are able to generate poems which honor the rhyme pattern nor the iambic pentameter in Shakespeare's sonnets, and thus sound like Shakepeare's. We apply the same improvement techniques to train the Markov models as well.

The Hidden Markov Model is interesting due to the visualization and interpretation of hidden states. Looking into the model with 5 hidden states, we are able to find a clear relationship between the position of words and hidden states they correspond to. For more information, see Section 4.

For Markov models, we pay special attention to the *2nd Order Markov Model*. In comparison with the first-order Markov model, the 2nd-order Markov model turns out to be a better and more precise model in the task of poem generation. Since there are only, in total, around 3000 different words in Shakespeare's sonnets and we we trained in groups, the number of distinct words in each group is below 1000.

As additional goals, we work on data of Spenser's poetry set and we also work on the rhyme scheme in detail.

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.

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 ending words that rhyme. 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 rhyming dictionary for each group. With this pre-built rhyming dictionary, we generate each line in the reverse direction with pre-sampled ending words that rhyme. For more details, see Section 6.1.

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 6.2.

3 Unsupervised Learning

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

3.1 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, and 100. We are working on model with 100 hidden states because there are around 3000 words in Shakespeare's poetry set and blow 1000 words for each group. In our training of Hidden Markov Model, we maximal the emitting probability of the training set rather than minimizing the Frobenius norm of differeces in **transition matrix** and **observation matrix** for the simple reason that in this unsupervised training, the shape of transition matrix and observation matrix differs a lot : observation matrix has much more elements than transition matrix in most of the test cases, with number of hidden states 5, 10, and 20, for instancce. So we maximize the emitting probability instead. Figure 1 shows when the number of states increases of $\log P$ will increase accordingly. This is to say that choosing number of states according to emitting probability may not be good since if the number of state is too large, say, close to the number of words in each group, the hidden states themselves are not representative.

The Optimal choice for the number of states 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.

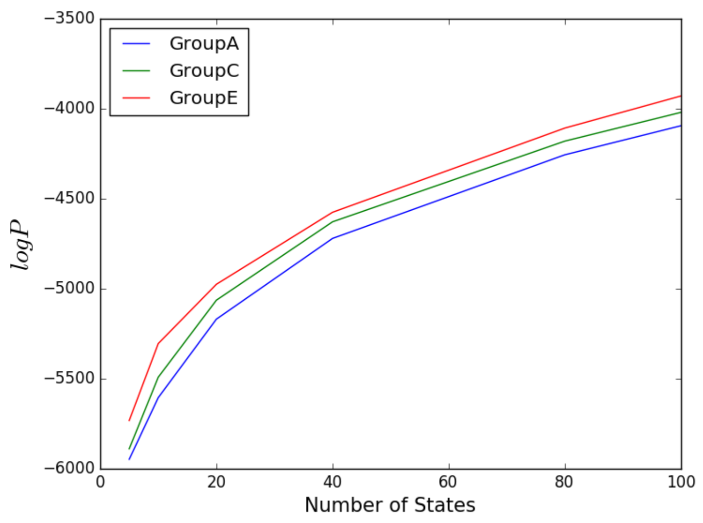

Figure 1: The emitting probability on training set *Group A*, *Group C* and *Group E* the vertical axis is kept as $\log P$

However, 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.

3.2 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 order) 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:

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

What should be mentioned is that we do counting and normalilzation for computing the piror probabilities $p(x_i, x_j)$ 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 $p(x_i, x_j)$ and $p(x_m|x_{m-1})$) is not substantially large, so the running time for 2nd order Markov Chain Model is affordable.

Higher Order Model We also considered higher Markov model, but as a simple reason, higher order may not perform much better than the second order, in the poetry language, especially,we have usually taken 2 word as a phrase. And if the order of the model is too high, actually we may generate a whole line from the original data set, and that is not what we are expecting.

4 Visualization and Interpretation for HMM

Number of hidden states

First, 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 a sonnet.

4.1 Interpreting the hidden states

Here we also present our findings about the interpretations 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

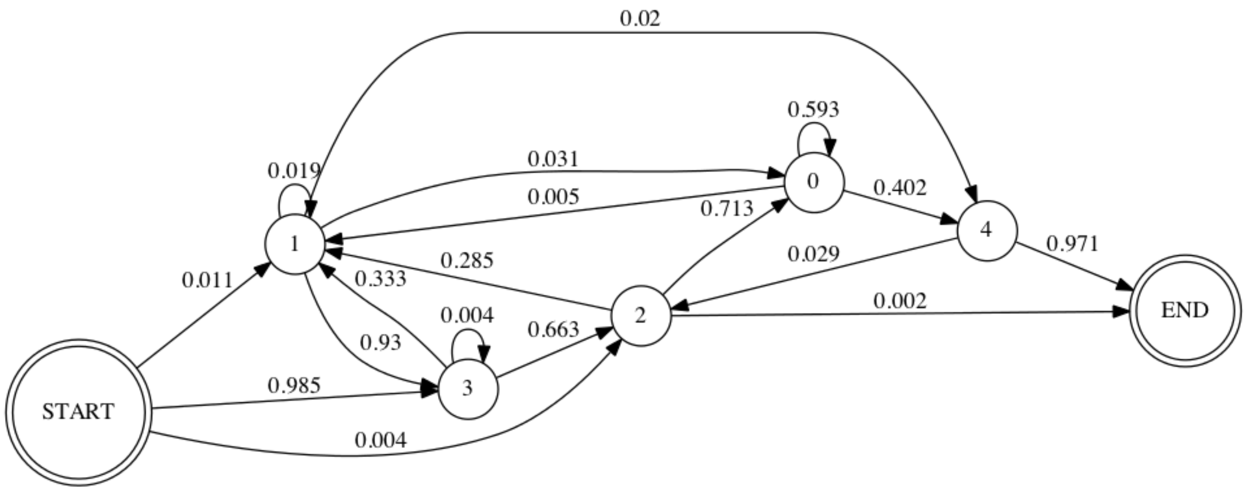

Figure 2: trnasition graphic model with 5 hidden state: In this figure, we ignore transition probability lower than 10^{-4} and we can see that some of the transition between states are ignorable, which is a indication that hidden states do have their own iterpretations

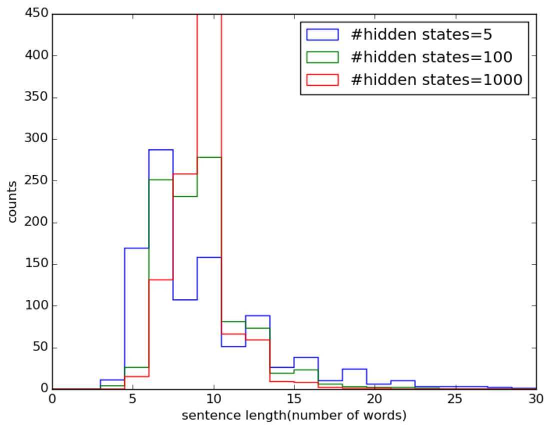

Figure 3: Sentence length distribution with different number of hidden states. When number of hidden states=1000, the length of the sentences are centered arount 8-9.

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.

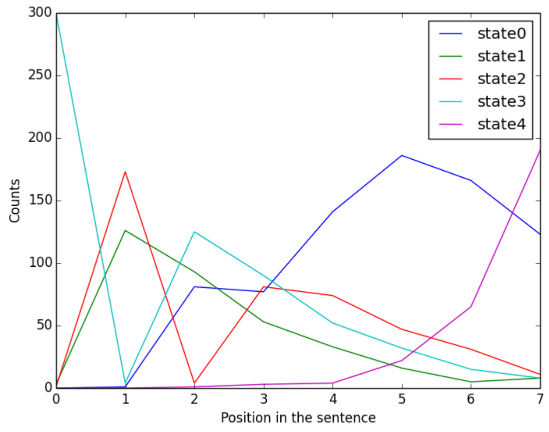

Figure 4: The distribution of different hidden states showing up at different positions in a sentence. Here we assume that a sentence is made of 8 words. If the actual length is greater than 8, then the later positions are also categorized as position 7.

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. 4 illusstrates the relationship between hidden states and positions in a sentence.

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.

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. In this study, the hidden state doesn't seem to bear sentimental meanings because the cor-

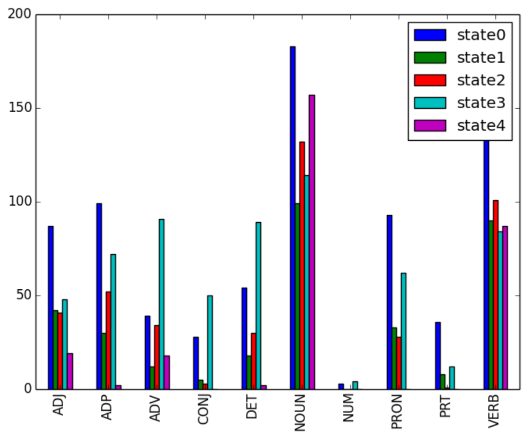

Figure 5: Statistics of corresponding part of speech tag for each state in the maximum likelyhood sequence given by viterbi algorithm

pus is so small and most words appear only once. So there is no way to tell their sentimental polarity. If we have a larger corpus, by using more hidden states, we could dig out more sentimental meanings.

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.

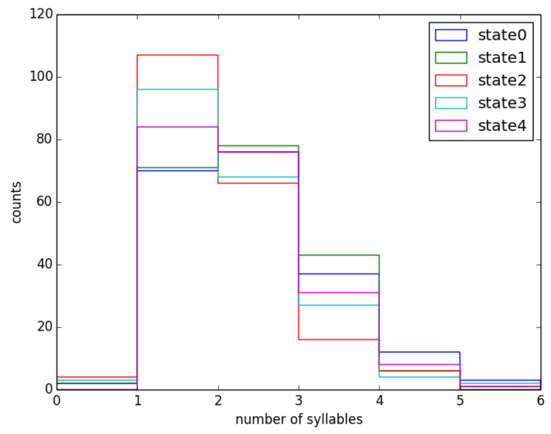

Figure 6: Number of syllables in the most probable 200 emitting words for each hidden state.

1st Hidden Markov Model

To make the improvements we have made clear, let look at a poem generated from HMM with 100 hidden states and trained and generated in a normal order (thus no rhyme dictionary is used and no syllable-counting is done):

look wherefore me but nothing fortune beautys hath soul
and beauty their most twice being
to have make some fairer is thee me seeming
my thou new silvered unbless and till their alone
then me enforced to therefore leaves time
but that see of mock was my hue
and in the finger some self chase
and but shadow even the from twenty me removed thy work from the warmed
that whom with in eye even defeat
where chide thy graces and victor to eye thence where lines may be gaol
the like do child say hold me love anothers writ
that play scope beloved for edge make or thine pupil intelligence
all want in that your adjunct to fair sweet praise
they gluttoning flattered remain

It is not accurate in rhyme, it does not make sense, and the rhythm is not correct.

For improvements, we partition the poetry set into different groups and train HMMs separately. We also generate poems group by group. Which is to say, in function

```
>>> poem_generate(num_of_hidden_states, num_poems)
```

we loop over all groups. In each group, we generate the last word randomly from pre-built rhyming dictionary. For more details on rhyming dictionary, see Section 6.1. In modelhmm member function:

```
>>> generating_random_line_end(self, end_word)
```

we generate one line with the prescribed ending word. Specifically, starting from the ending word, we randomly (according to the trained transition and observation matrices) generate words in the reverse direction until we meet the END state, which marks as the completeness of a sentence. If we enable the syllable-counting feature, we count the number of syllables in each line and after several trials (50 or 500), we pick the line that has the number of syllables closest to 10. Finally, we concatenate these rhyming line-pairs into poems.

Here is one poem we generate from HMM with 80 number of hidden states, we generate at most 5000 lines for counting the syllables. We name it **'Stochastic thought'**:

Stochastic thought

They stain that calls, for every made why,
But unto hammered alchemy nearly,
When with thought restful tigers catch die,
And his tongue the better loved to astronomy be:

For canker hold painter flies thine as bold skill,

In others extremity these thence burthen still to hue see;
When whos love painter even to still still,
Shall by will from decease thee.

Whate'er my true love they to me shade,
Mine to on are, love in thee offences behind way,
What jewel me do not lack fade,
To trial leases forth or thinking of time day?

Lascivious nothing thy of takes becoming words heart in old thee,
This not slight thrice you love she defence see.

We can see that with our improvement techniques, the poems we generate always honor the rhyme pattern. Moreover, all the 14 lines have total number of syllables around 10, which honor the iambic pentameter in Shakespeare's sonnets. However, most lines do not make sense.

2nd order Markov Chain Model

We trained a 2nd-order Markov model for each group with the same improvement techniques as in the HHMs. 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,
And swear that brightness doth not grace the day,
What potions have drunk of siren tears,
My sinful earth these rebel powers array:

Duty so great which wit so poor as mine,
The spirit of love with perpetual dulness;
Nothing sweet boy but yet like prayers divine,
Thy hungry eyes even till they wink with fulness.

Therefore love be of thyself so wary,
Thy black is fairest, in my judgments place,
Of more delight than hawks and horses be,
To guard the lawful reasons on thy face?

Or gluttoning on all or all away,
Yet him for this my love was my decay.

Comparison : *1st Hidden Markov Model* vs *2nd order Markov Chain Model*

- Firstly, both models honor the rhyme pattern nor the iambic pentameter in Shakespeare's sonnets.
- We find in the reasoning of the verses, Markov model makes more sense than the Hidden Markov Model. For example, the **'Hope or Fear?'** reads more reasonable, while **'Stochastic thought'** does not make much sense.

- Since we generate rhyming lines for each group independently, our models can't keep a unified topic across all the 14 lines. Although every line from the 2rd-order Markov model makes some sense separately, the whole poem sounds like a drunk Shakespeare who randomly jumps from one topic to another.

6 Additional researches that we have worked on

6.1 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 total number of syllables in each line varies a lot.

6.2 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.

6.3 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 easily 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.

the dedicated words which writers use
to new found methods and to compounds strange
as fast as thou art too dear for my muse
so far from variation or quick change
reserve them for my love doth well denote
or from their proud lap pluck them where they grew
if that be fair whereon my false eyes dote
at wondrous sight of so celestial hew
prison my heart with silence secretly
and sweets grown common lose their dear delight
but rising at thy name doth point out thee
than when her mournful hymns did hush the night
  so return rebuked to my content
  that it hereafter may you not repent

After examination, we find that the first line "the dedicated words which writers use" is actually borrowed from Shakespear sonnet 82, and the eighth line "at wondrous sight of so celestial hew" is borrowed from Spenser sonnet 3. Other lines are not directly borrowed from either of them, but are kind of mixture of them. For example, the third line "as fast as thou art too dear for my muse" is a combination of "As fast as thou shalt wane so fast thou grow'st" from Shakespere sonnet 11 and "So oft have I invoked thee for my muse" from Shakespere sonnet 78. By introducing additional texts (spenser's poems), we get more variations in the poems we generate.

7 Conclusion

In this project, we trained **Hidden Markov Models (HMMs)** and **2nd-order Markov Models** for poem generation.

Due to the clear rhyme pattern in Shakespeare's sonnets, the basic strategy in our model training is to train multiple models for different part of the poem. We tokenize the words as features in each group and run the EM algorithm with different number of hidden states to train the HMMs. Looking into the HHMs with 5 hidden states, we are able to find a clear relationship between the position of words and hidden states they correspond to. The speech tags of words also show a strong correlation with the 5 hidden states. Since HHMs is a probabilistic model, we are able to generate poems by the HHMs we trained.

After examining the HHMs with different number of hidden states, we found that the more hidden states the HHM has, the more meaningful sentences it can generate. Due to this observation, we decided to also train Markov models. We tried the standard (first-order) Markov model and the second-order Markov model and found that the second-order Markov model works very well to learn short phrases and to generate meaningful sentences. However, we also noticed that sometimes the second-order Markov model just "borrows" sentences from the training dataset. That's why we think higher order may not perform much better than the second order, in the poetry language, especially,we have usually taken 2 word as a phrase. And if the order of the model is too high, actually we may generate a whole line from the original data set, and that is not what we are expecting.

The naive approach to train both the HHMs and the 2nd-order Markov models honors neither the rhyme pattern nor the iambic pentameter in Shakespeare's sonnets. To generate poems whose lines rhyme, we first use the *NLTK* package and the RhymeBrain website `http://rhymebrain.com/en` to build a rhyming dictionary, and then generate each line in the reverse direction with pre-sampled ending words that rhyme. To honor the iambic pentameter, we try to generate lines with total number of syllables as close to 10 as possible. To achieve this goal, we simply keep generating lines until we get a line with exactly 10 syllables. With these additional modifications, we can finally generate Shakespeare-style poems with both the HHMs and the 2nd-order Markov models. Our framework enables us to train our models with additional texts. We include all 139 of Spenser's sonnets in our training dataset and get models with bigger dictionary and generate poems with more variations.

Since we generate rhyming lines for each group independently, our models can't keep a unified topic across all the 14 lines. Although every line makes some sense separately, the whole poem sounds like a drunk Shakespeare who randomly jumps from one topic to another. If we have more time on this project, we may introduce some mechanism to unify all the lines with a common topic. To achieve this goal, one possible solution is to first train a topic model from the dataset. To generate a poem, one can first choose a topic and sample key words from this topic for each line. Finally, one can generate each line with the given key words with either the HMMs or the 2nd-order Markov models. In this way, the given key words in each line can unify all the lines together.