CS 155 Project 3 Report

Introduction

Group Members: June (Joo Eun) Kim, Richard Qing Bao, Matthew (Min Hyuk) Kim
Team: Team Green

June
- Wrote pre-processing code for HMM
- Improved the HMM poem generation algorithm by incorporating rhyme and punctuations
- Generated other forms of poems using the trained HMM

Richard:
- Wrote pre-processing code for RNN
- Trained a character-based LSTM model and generated poems from it

Matthew:
- Tested different hidden states of the HMM
- Wrote naive poem generation code
- Wrote visualization code

Pre-processing

We took different approaches in pre-processing the data for the HMM and the LSTM neural network. For the HMM, we decided to train on each line rather than a whole poem in an attempt to best preserve the iambic pentameter structure of each line, at the cost of not being able to utilize the relationship between neighboring lines. In order to avoid meaningless punctuations in the middle of sentences, we removed punctuations from the raw data, except for those that are part of a word (i.e. I'll, 'tis, summer's). We decided what counts as a word by referring to the given Syllable_dictionary.txt file, which also facilitated looking up syllable information from the file when generating the poem. Similarly, we removed capitalizations. We hoped to keep some words capitalized to preserve simple English conventions and Shakespeare's voice, but it was difficult to come up with a systematic way of distinguishing words that are capitalized because they're at the beginning and those that are always capitalized. For example, the word 'nature' appears in both lowercase and uppercase in the middle of the sentence, and it would be nice to treat those as separate words, but when a capitalized 'Nature' appears at the beginning of a sentence, we do not know which of the two categories it belongs to. Because of this ambiguity, we decided to discard all capitalizations and only keep a few obvious exceptions (I, O, I'll) capitalized for the sake of grammar. We treated hyphenated words as one word, since separating a hyphenated word could potentially result in two meaningless words in poem generation.

For the recurrent LSTM neural network, we preprocessed the data differently, as it is character-based, so we cannot directly manipulate higher level features such as syllabic structure. In order to reduce the vocabulary and improve the training efficiency, we decided to normalize all alphabetic characters to lower-case and used regular expressions to discard all punctuation marks with the exception of apostrophes and commas. We believed that these punctuation marks occur often enough to effectively train on, and that they are the most significant, as they follow the strictest grammatical rules. For instance, an apostrophe is almost always followed by the character “s”. Similarly, a comma is always followed by a space character, or sometimes a new line character to indicate the end of a line in a shakespearean sonnet. Other punctuation marks such as colons and semicolons are much more situational and arbitrary in their usage, so we discarded them. Dashes are also difficult to work with on the character level, as they operate on the syllable level. Question marks and exclamation marks are even more difficult to manage, as they depend on sentiment, and thus operate on the word and line level. Moreover, they are especially rare in the training data. However, we later decided to incorporate question marks as well, because one is included in the seed.

Unsupervised Learning

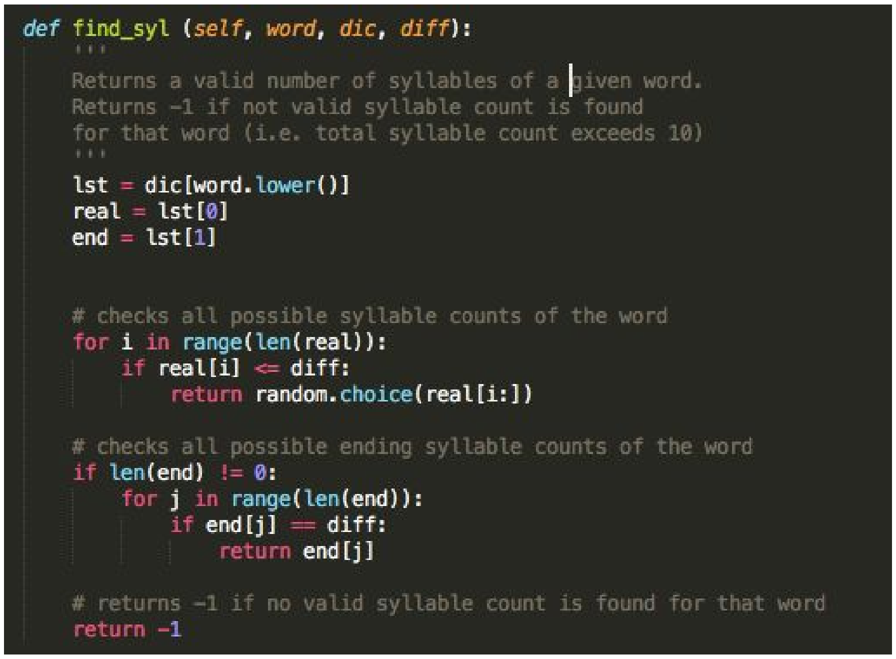
We used the Baum-Welch algorithm (Set 6) to create an unsupervised HMM for poem generation. In unsupervised training, the maximum likelihood problem is given by the equation:

$$\operatorname{argmax} \prod_{i=1}^{N} P(\mathbf{x}_i) = \operatorname{argmax} \prod_{i=1}^{N} \sum_{\mathbf{y}} P(\mathbf{x}_i, \mathbf{y}).$$

The Baum-Welch algorithm consists of four steps. First, we randomly initialize the model parameters (transition and observations matrices). Then we compute marginal probabilities based on the current parameters using Forward-Backward (E-step). Next we update the model parameters based based on the maximum likelihood estimate (M-Step). Finally, we repeat the E-step and the M-step for a given number of iterations (1000) or until the model parameters converged. We chose the optimal number of hidden states qualitatively by examining the lines generated by models with 1, 2, 4, 8, 15, 20, or 25 hidden states. The lines generated by the HMM with 25 states most accurately resembled Shakespeare's style and made the most sense.

Poem Generation: Hidden Markov Models

We generated the sonnets line by line using the generate_emission() method in the HMM class. generate_emission() takes in a total of three parameters: the total number of syllables needed in a line (M), the reverse of the observation map (obs_map_r), and syl_dict, a syllable dictionary with keys of all the words Shakespeare uses. The values of this dictionary are essentially lists of two lists. The first list represents all possible syllable counts of a word in descending order, and the second list represents all possible ending syllable counts of a word (labeled E) in descending order. An example of a key value pair of this dictionary is {‘word’: [[4], [3]]}. The output of generate_emission() is a list of emissions that can be converted into a line of words with a total of 10 syllables. In the method , we initialize an empty list of observations (emission), an empty list of states (states), and a counter that keeps track of the total number syllables in the line. Next, we probabilistically generate the emissions and states using the transition and observation matrices. We sampled the states (y^i) from $P(y^i|y^{i-1})$ and sampled the observations x^i from $P(x^i|y^i)$ For each observation, we check if the word corresponding to the observation has a valid number of syllables. We check this by calling the find_syl() method:



```python
def find_syl (self, word, dic, diff):
    '''
    Returns a valid number of syllables of a given word.
    Returns -1 if not valid syllable count is found
    for that word (i.e. total syllable count exceeds 10)
    '''
    lst = dic[word.lower()]
    real = lst[0]
    end = lst[1]


    # checks all possible syllable counts of the word
    for i in range(len(real)):
        if real[i] <= diff:
            return random.choice(real[i:])

    # checks all possible ending syllable counts of the word
    if len(end) != 0:
        for j in range(len(end)):
            if end[j] == diff:
                return end[j]

    # returns -1 if no valid syllable count is found for that word
    return -1
```

This method takes in 3 parameters: a word, the syllable dictionary, and a value representing the difference between the total number of syllables needed and the current number of syllables in the line. The find_syl() method checks all possible syllable counts of a word. It does this by randomly selecting a syllable count that is less than or equal to the difference value. If no value meets this threshold, it selects a possible ending syllable count of the word. This value has to be equal to the difference (since it's the last/ending word in the line). We call this method until an observation with a possible syllable count value is found. Then the counter is updated and the whole process is repeated until the total number of syllables in the line reaches M (10 in the case of generating sonnets).

We used this algorithm to generate sonnets line by line for HMM with 1, 2, 4, 8, 15, 20, 25 hidden states. Here are the first 5 lines from each sonnet (except HMM with 25 hidden states):

HMM with 1 hidden states.

I and too I I take he errors pen
Budding doth flowers makes deeds where sharp'st this so
Of lead another day the instant fair
My thee friend those mine which to I so turns
A is give returned good commend learned but

****

HMM with 2 hidden states.

Thou grown most thee less life to that being their
First mind lie none with happier and him not
Give my for that of to to but looks to
Bestow so O my thy on grace thine you
You do if me to thoughts word at and as

****

HMM with 4 hidden states.

More your said lips distraction if princes
Few of may great set when fair is in 'greeing
Shames sunk that grows do I blood that thy self
It brings grief in more affairs is do this
Then proud blot year eyes his time O this of

****

HMM with 8 hidden states.

All to world to not so eyes sourly your
Uneared he with this leaving ward and seen
From painted short-numbered took churl gaol of
Receiving yet never can abuses
It again by it salving it no thou

****

HMM with 15 hidden states.

Swear beauteous when your graces thee them my
So quest is keep more die the creating
Seeing as it lost what I in made but
Despair to that on your more days other
When give and farewell from thou if the it

****

HMM with 20 hidden states.

Whence less the song seen leave or no thine own
That alone on whose their will not and with
Rocks every grave I hold so long-lived luck
Suspect thy most issue fight this gentle
But with being she abuse to me fire

****

HMM with 25 hidden states.

Then but wherein thou should I were abused

Art gone in make be doth so oblivion
Tongue can say folly had doth to sight with
Self thee the distraction and faith that none
On for thy self thou have I audit but
Abysm in cross self-love prove upon ah
Of me ruth assured the why of grace
Woe the motion of my own rain tongue-tied
Sparkling note both may in in brand of deaths
That am heir words of cloud shall they is she
Before olives in their birth and to my
Desert played afterwards deeds vex me muse
Where should so idol souls reason thee spur
On love hidden that which in your canst shine

****

We can see that all the sonnets generated from each model have exactly 10 syllables in each line (based on syllable_dict.py). Some of the lines generated follow iambic pentameter; however, the lines do not follow this specific meter consistently. Since a singular sequence consists of a line, the HMM model generates the poem line by line. Thus there is little to no rhyme in the generated sonnets. As we increase the number of hidden states, the generated lines increasingly make more sense and more accurately resemble the lines in Shakespeare's sonnets. When we trained the model with 1 hidden state, the words in the lines were in random order. This is the case because when there is only a single hidden state, each observation has an equal chance of appearing anywhere in the generated line. The lines retain Shakespeare's original voice and style as most of lines follow a reverse sentence structure. In contrast to typical grammatical structure (Subject(S), Verb(V), Object(O)), Shakespeare uses SOV pattern. The HMM was able to capture these qualities as it learns to generate states representing subject, object, and verbs, and higher transition probabilities from subject to object state and object to verb state.

Poem Generation: Recurrent Neural Networks

For the recurrent LSTM, we used an implementation provided in the Keras Library. The training input (X) to this neural network consists of 40-character long sequences, and corresponding training output (y) is the next character following the sequence. A Recurrent Neural Network utilizes internal memory to keep track of all the previous inputs so that they can continue to influence predictions. It produces an output on the current input sequence, copies that output, and loops it back into the network. Essentially, this allows the immediate past to influence the present. Therefore, by iteratively generating consecutive sequences of characters from a body of text, we are able to teach the algorithm to recognize patterns, such as common character combinations (words) and perhaps even higher level features such as basic grammatical rules.

To generate the training data, we parsed the sonnets from the training text, and for each sonnet, we generated a dense list of 41-character sequences by shifting forward by only 1 character for each consecutive sequence (no semi-redundancy). We repeated this process for each sonnet individually because of concerns over independence, as we felt uncomfortable with the assumption that all sonnets could be combined into one large continuous text. There may be subtle yet significant features in the transition between sonnets. For example, certain combinations of characters (words) may be more prevalent toward the end of a sonnet, and others may be more prevalent at the beginning of a sonnet. Therefore, we did not want to have any “hybrid” sequences mixing those two distinct spaces, as we would never want to generate such sequences if we are only concerned with task of generating a single sonnets.

Subsequently, we converted each of the 41-character sequences to a numerical by mapping the characters to their corresponding indices in a sorted “vocabulary” dictionary of all the characters present in the training data. This mapping was a one hot vector mapping, so each character became a list with only one non-zero entry - the value 1 at the location corresponding to its index in the “vocabulary” dictionary. For each sequence, we extracted the subset of the first 40 values (one hot vectors) as the training input (X) entry, and the 41st character as the corresponding training output (y) entry. Therefore, the shape of the overall training input was a triple nested list: a list of sequences in which each value is a one hot vector corresponding to a single character.

We trained the character-based LSTM model using a single layer of 150 LSTM units, followed by a dense fully connected output layer with a softmax nonlinearity. We trained the model to minimize categorical cross-entropy over 100 epochs, which we found to be sufficient. The validation accuracy of the model steadily increased throughout this process, although toward the end, the marginal gains were very small. After the 100th epoch, we achieved 84% accuracy.

Finally, we saved the model, as well as the “vocabulary” mapping, as files in the directory, allowing us to load them again for the task of generating new sonnets. Sampling from the softmax output of the trained model, we were able to build upon the initial 40-character seed: “shall i compare thee to a summer’s day?\n”.

shall i compare thee to a summer's day?
thou canst not love dead strang, repear,
and nothing thy self, and the dust change,
whilst i not spirit comman ever forgord me

the sweet thief which i swear feeling love
then of your meart, that leaves muse it dissed,
and that in the walls, doubt my love you may?
to your mourned, and what exterming pessed,
the earth can have but each doth nature
therefore in that i in me do is think forsure,
the conting of you, love's fire hath and me

lencly that thou gently decail thee i rey,
and that they sweet self to abloven ever
non's can brooking the time to you, befores,
and grew they refunt to hear this i rey,
and yet the stat times the painting try

We also played around with the temperature parameter, which controls the variance of the sampled text. The following sonnets are the result of generating 520 characters:

```
----- temperature: 0.25
----- Generating with seed: "shall i compare thee to a summer's day?
"
shall i compare thee to a summer's day?
thou canst not love dead strangely lear,
that ther for my side, and say i windrong fair well,
tired that ele the wanco's rantoon barthous thine eye,
therefore to be must revire thee is old,
that the sweet that which it flowers die
but that sweet form of worthles doblows,
lest enours by have pry alice and then

the sweet that my adders stite, and prove change,
that sometime and for miniems not then
giving the time that lougs hath life in me,
and yet thou shalt by the friefing chilest,
who give unself th


----- temperature: 0.75
----- Generating with seed: "shall i compare thee to a summer's day?
"
shall i compare thee to a summer's day?
thou wouldst thou till achivily phory
and muticoon all thy self a lay your time
to know my graces and my love is aw
those wilt, for a womak, commous own losies,
by that by thy death whose large and look,
thy every it thy grace yell all the scortured,
and bras injomech of things injurie
then may to come onloying ye?
my sighttest with his compixed in the mind
to metjoy it that i say shall styrs,
as the dutufers injurie hath new to no
mut those, overrights upon this light?
patulining o'ers but a man right


----- temperature: 1.5
----- Generating with seed: "shall i compare thee to a summer's day?
"
shall i compare thee to a summer's day?
though i auty conterlegt so should
but brauds of blaming fire desired ambeast,
but motiendble hailfully her carnstarly,
things thy extermination storn lose
it mine eye me basable giftles of seem'st,
```

seen my love of love, my verse if i shortife,
to hispuble to me, and destre's eye chued?
appyme that bear, and my actumpled,
and acantul from the flities with my,
thire you for eyes abundance an in praise,
brank in their bad spiled introunting suth stee,
thou be gush i darls and think the breath,
but this in elding, thus refiture's laste
take pleasure give the ready spirit,
thou can i now and to thoughts of felf to be
though heaven's sweet'st friend a

We see that the neural network is able to consistently generate actual words, and that it knows to enter a new line character after approximately 8-11 syllables on average. Grammatically, all the commas and apostrophes are correct. Every comma is followed by a space, and every apostrophe is toward the end of the word, followed by the “s” character. We see that temperature has an interesting effect on the predictions. The sonnet generated with a temperature of 0.25 has a lower frequency of wrong words than the sonnet generated with a temperature of 1.5, but it is generally more boring. This is because there are many repeated patterns. One example is the words “Sweet,” which occurs three times. This demonstrates the low variance produced by a low temperature. On the other hand, the sonnet generated with a temperature of 1.5 has a much more diverse range of words, although there are still some repetitions. However, the words in this sonnet are much more interesting, and also longer on average, with less organized structure. This demonstrates the high variance produced by a high temperature. In addition to tuning the temperature, we also tuned the number of training epochs and the number of LSTM units, but with less of an effect.

We conclude that the recurrent LSTM neural network has the potential to successfully learn higher level features such as sentence structure and sonnet structure. Even our model was able to decide when to end a line close to 10 syllables and insert commas and apostrophes in a way that is grammatically coherent. However, overall, the poems generated with the LSTM are poorer in quality than those generated with the HMM. Although the LSTM is more flexible because it operates at a character level, the HMM has the advantage of more easily incorporating higher level features such as syllable stress and even rhyme, allowing it to generate poems with more meaningful structures.

Additional Goals

Rhyme

As discussed in the HMM poem generation section, our initial naive poem generation algorithm wasn't able to capture rhymes, which is impossible when we train on each sequence rather than a stanza or a whole poem. In order to introduce rhymes into our generated sonnets, we created a rhyming dictionary from the given sonnets, as Shakespeare's sonnets follow a strict rhyming pattern (abab cdcd efef gg). We found two sonnets in the dataset with less than or more than 14 lines, so we discarded them while obtaining rhyming pairs.

```python
# This rhyme dictionary is a list of sets, where all the elements in each set rhyme with each other
rhyme_dict = []

def add_to_rhyme_dict(w1, w2):
    '''
    This function takes in a pair of rhyming words and adds them to the rhyme dictionary.
    '''
    # Store the group that one of the words belongs in, if any
    stored = None

    for group in rhyme_dict:
        if w1 in group:
            if not stored:
                group.add(w2)
                stored = group
            else:
                # Combine two groups
                stored.update(group)
                rhyme_dict.remove(group)
                break
        elif w2 in group:
            if not stored:
                group.add(w1)
                stored = group
            else:
                # Combine two groups
                stored.update(group)
                rhyme_dict.remove(group)
                break

    if not stored:
        rhyme_dict.append({w1, w2})
```

The above code adds a pair of rhyming words to our "rhyming dictionary", which is a list of sets, where all the words in each set rhyme with each other. Then in poem generation, we generated two lines that rhyme by seeding their first words with rhyming words and generating the remaining words in reverse direction, which was done by training our HMM with reversed sequences.

One obstacle to this method was that we're starting our sequence generation with a given observation rather than state, which meant that we had to somehow find a state for that observation to continue the sequence generation. Our initial approach was generating the first sentence in the normal approach (without seeding), and then finding a word that rhymes with the first word of the first sentence to seed the second sentence with. Then we would have to make an assumption that the second sentence has the same first state as the first sentence, which is reasonable under the premise that a state that generated one word is also not unlikely to generate a word that rhymes with it. While implementing this approach, however, we realized that the first word generated for the first sentence is not guaranteed to have a rhyming pair in our rhyming dictionary, which could only be solved through integrating an external rhyming dictionary or a very computationally expensive method of repeatedly generating words until we find one that appears in our rhyming dictionary.

Therefore, we took a different approach with different assumptions. Using Bayes' Rule $P(y \mid x) = \frac{P(x \mid y)\, P(y)}{P(x)}$, we found P(y | x) from P(x | y), which is stored in our observation matrix of the trained HMM. Assuming uniform distribution for P(y) and given that P(x) is constant for a given x, we computed P(y | x) by obtaining P(x | y) for each y and normalized them such that they add up to 1. Then we sampled a state that could have generated the given observation using the computed probabilities. This approach allowed us to generate sentences with a chosen starting word and a sampled likely state, which were simply reversed at then end to get our sentences in the correct order with rhyming words at the end.

Below is our sonnet generated with rhymes, which adheres to the correct rhyming pattern without loss of quality compared to our naive poem generation:

Have tongue as if me says approve now grief
If I it besides where love to so bark
Which my friend thou and thing they play with chief
More leave I with sometime and bright in mark
Lived for his thoughts I for poor heart esteemed
Make me teeth to foist of my equipage
Thou crooked robbery my which thee so deemed
From the death-bed when will nothing poor page
My beated extern and storm-beaten luck
I speak if year to above muse with power
His sweet his what I three all in in pluck
Thy appeal thy shows feeds dear let that flower
Care true thrive and should sees thee millioned sits
That never which mock thing defence befits

Generating other poetic forms

Using the same trained HMM, we generated various poetic forms: Haiku, Petrarchan sonnet, and limerick. We focused on the syllable and rhyming structures that define these forms rather than voice and sentiment, as we used our HMM model trained on Shakespeare's sonnets.

A haiku doesn't contain rhymes and consists of 3 lines, with the first line having 5 syllables, second 7 syllables, and third 5 syllables. Generating a haiku simply required changing the number of syllables using our naive poem generation method.

Thy are to transport
If so if check love so though
Springs which whether I where

A Petrarchan sonnet contains 14 lines with the rhyming scheme ABBAABBA CDCDCD, each line following iambic pentameter. We slightly modified our previous generate_rhyming_sentences function to generalize it to create more than two rhyming sentences.

```python
def generate_rhyming_sentences(hmm, obs_map, n_sentences=2, n_syl=10):
    # Choose a rhyming group with enough words
    groups = []
    for group in rhyme_dict:
        if len(group) >= n_sentences:
            groups.append(group)

    # Choose rhyming words
    words = random.sample(random.choice(groups), n_sentences)

    sentences = []
    for word in words:
        sentences.append(sample_sentence(hmm, obs_map, word, n_syl))

    return sentences
```

Using the modified generate_rhyming_sentences function, we were able to easily create different rhyming groups (A, B, C, D) to create a Petrarchan sonnet:

```python
def generate_petrarchan_sonnet(hmm, obs_map):
    '''
    Generate a Petrarchan sonnet with rhyming scheme: ABBAABBA CDCDCD
    '''
    poem = ''

    # Generate 2 groups of 4 rhyming sentences (A, B),
    # 2 groups of 3 rhyming sentences (C, D)
    A = generate_rhyming_sentences(hmm, obs_map, 4)
    B = generate_rhyming_sentences(hmm, obs_map, 4)
    C = generate_rhyming_sentences(hmm, obs_map, 3)
    D = generate_rhyming_sentences(hmm, obs_map, 3)

    # First stanza
    poem += (A[0] + '\n' + B[0] + '\n' + B[1] + '\n' + A[1] + '\n' +
            A[2] + '\n' + B[2] + '\n' + B[3] + '\n' + A[3] + '\n' + '\n')

    # Second stanza
    for i in range(3):
        poem += C[i] + '\n' + D[i] + '\n'

    print(poem)
```

The above code outputs our Petrarchan sonnet:

Thought my colour verse keeps my crests can made
She love my desire humble to see tomb
Height in like world I knowing strange shall womb
Live delight them in more thee think my shade
Whence thee is so be no and no shouldst fade
My self use lie with compounded of dumb
Fortune's for is beauty maladies come
Give or kind wherein cruel shows since jade

Doth put a perfumed new-fangled impute
Is flattered forth contented in from fair
Lust the their worst thou them women's stop fruit

Past with sweetest name with tiger's repair
On but gilded and since loves of lacked mute
With are charactered rich join with stage air

A limerick consists of 5 lines, where the first, second, and fifth lines rhyme and each have seven to ten syllables, and the third and fourth lines rhyme and each have five to seven syllables. The rhyming lines also have the same verbal rhythm, which we decided to ignore since we are simply using our HMM trained on each line of Shakespeare's sonnets. In order to further simplify things (and also compensate for the loss of rhythm),we also decided to make the rhyming lines to have the same number of syllables, which is randomly chosen at each poem generation within the allowed range (between seven and ten for the first, second, and fifth lines, and between five and seven for the third and fourth).

```python
def generate_limerick(hmm, obs_map):
    '''
    Generate a limerick with rhyming scheme: AABBA
    '''
    # Generate 3 rhyming sentences (A) with n1 syllables,
    # 2 rhyming sentences (B) with n2 syllables

    n1 = random.randint(7, 10)
    n2 = random.randint(5, 7)

    A = generate_rhyming_sentences(hmm, obs_map, 3, n1)
    B = generate_rhyming_sentences(hmm, obs_map, 2, n2)

    print(A[0] + '\n' + A[1] + '\n' + B[0] + '\n' + B[1] + '\n' + A[2] + '\n')
```

This code generates the following limerick:

Worth but lose his wealth not foul slide
O wherein my which hath belied
Thou to your earth be perish
Dost prepare my true cherish
Beauteous love he make an abide

As we can see, it is impossible to generate a haiku with its common theme of nature or a limerick that contains the humorous voice that characterizes limericks without training on datasets consisting of haikus or limericks. And yet, we can see that with the HMM we already trained on Shakespeare's sonnets and simple adjustments to the functions we already wrote, we can easily replicate the different structures of various forms of poems.

Punctuations

While pre-processing our training data, we decided to discard all punctuations except for those part of a word (as defined by Syllable_dict.txt file). Although this was immensely helpful in training the HMM on real words, looking up syllable information on each word, and assembling them into a sentence in a coherent manner, we thought it would be nice to put punctuations back after generating the poem to make our poem more closely resemble the original sonnets by Shakespeare. We tried to look for a pattern in the sonnets; for example, the last line of a sonnet might always end with a period, but we soon found out that that is not true. Thus we decided to handle the punctuations in a more probabilistic way. Our method doesn't use the HMM, and thus our punctuations are not related to the words that come before them, but we borrowed the idea of the observation matrix to come up with a way to assign punctuations in a way that incorporates what we can obtain from the training data.

First of all, we grouped the lines in a poem by their location in the poem structure:

From fairest creatures we desire increase,
That thereby beauty's rose might never die,
But as the riper should by time decease,
His tender heir might bear his memory:
But thou contracted to thine own bright eyes,
Feed'st thy light's flame with self-substantial fuel,
Making a famine where abundance lies,
Thy self thy foe, to thy sweet self too cruel:
Thou that art now the world's fresh ornament,
And only herald to the gaudy spring,
Within thine own bud buriest thy content,
And tender churl mak'st waste in niggarding:
  Pity the world, or else this glutton be,
  To eat the world's due, by the grave and thee.

The lines with the same highlight color belong in the same group, which are assumed to have an equal probability of having a specific punctuation. The different probabilities of each group having a specific punctuation can be organized in a matrix that resembles the observation matrix, where the location groups are analogous to states and the punctuations are analogous to observations.

	,	**.**	**:**	**;**	**?**	**!**	**None**
1st-3rd lines of 1st-3rd stanzas	0.753	0.003	0.383	0.155	0.030	0.005	0.155
4th line of 1st-3rd stanzas	0.121	0.512	0.216	0.013	0.102	0.026	0.009
1st line of the last stanza	0.887	0.007	0.013	0.000	0.007	0.000	0.086
2nd line of the last stanza	0.000	0.993	0.000	0.000	0.000	0.000	0.007

The probabilities were obtained from simply counting from the training set and normalizing such that each row adds up to 1. We can see that the probability of the last line of the poem ending with a period is 0.993, which is what we would expect. Note that we are making a naive assumption that the end of each line is the only place that punctuations occur for simplification.

Using this probability matrix, we sampled a punctuation for each line in the same way we would sample a word given a state.

```python
# Sample punctuation given a location in poem
def get_punc(loc):
    probs = punc_probs[loc]
    rand_var = random.uniform(0, 1)
    next_punc = 0
    while rand_var > 0:
        rand_var -= probs[next_punc]
        next_punc += 1
    punc = next_punc - 1
    if punc == 6:
        return ''
    else:
        return puncs[punc]
```

This punctuation generation process is entirely independent from the poem generation process. We simply attached the punctuations generated from this function to the appropriate locations in the already generated poems to get the following result:

> *Have tongue as if me says approve now grief,*
> *If I it besides where love to so bark,*
> *Which my friend thou and thing they play with chief,*
> *More leave I with sometime and bright in mark:*
> *Lived for his thoughts I for poor heart esteemed,*
> *Make me teeth to foist of my equipage*
> *Thou crooked robbery my which thee so deemed,*
> *From the death-bed when will nothing poor page.*
> *My beated extern and storm-beaten luck*
> *I speak if year to above muse with power,*
> *His sweet his what I three all in in pluck,*
> *Thy appeal thy shows feeds dear let that flower.*
> *Care true thrive and should sees thee millioned sits,*
> *That never which mock thing defence befits.*

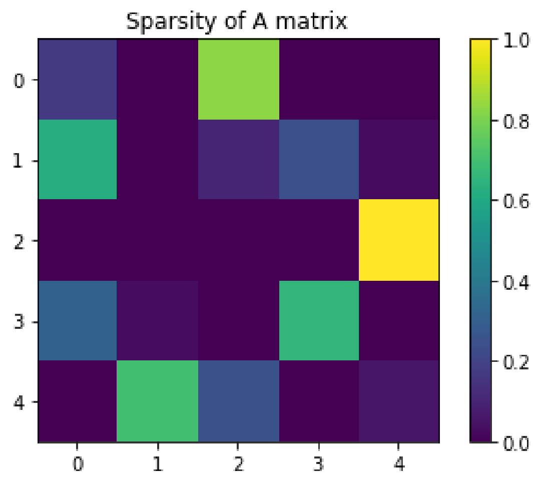
The punctuations do not accommodate the context of each line, but they are likely punctuations at each location, yielding a poem that includes seemingly reasonable punctuations and replicates Shakespeare's original style.

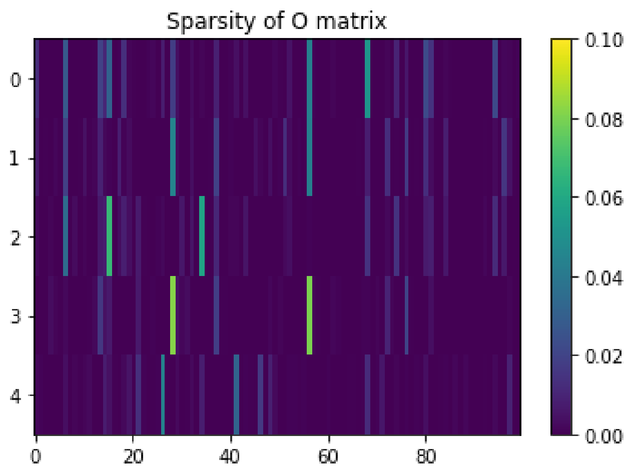
Visualization and Interpretation

We analyzed the HMM models by generating word clouds for each state and creating graphs that depict the sparsity of the transition and observation matrices. We generated these visualizations for the HMM with 5 hidden states. For each state, we found top 10 words that associate with the hidden state. Here are the top 10 words for seven of the states:

State 0: (verb) make, made, take, love, show, say, tell, live, give, change
State 1: (beauty) art, sweet, heart, beauty, gentle, love, muse, fair, time, kind
State 2: (articles) thee, thy, doth, dost, yet, hath, much, hast, although, many
State 3: (life) die, death, white, youth, life, kill, decay, born, winter, form
State 4: (noun) tongue, love, world, men, fear, soul, eye, hand, face, heart

We can see that state 0 mostly consists of verbs. State 1 consists of words relating to beauty while state 2 has a lot of articles. State 3 are words that describe life and state 4 consists of nouns.





The trained A and O matrices are both sparse, but O is more sparse than A. The sparsity along each row i of A means that given a state i, only a few states are achievable in the next state (columns of intensity 0 means 0 probability of transitioning from state i to the state represented by that column). Similarly, the sparsity along each row i of O means that given a state i, only a few observations can be generated from that state (columns of intensity 0 means 0 probability of state i generating the observation represented by that column). Since O is more sparse than A, it could be interpreted as the association between states and observations being stronger than that between different states (transitioning from one to another). Some columns of O have very low intensity across all the rows, which means that the observations corresponding to those columns have overall very low probabilities of occuring (all states are unlikely to generate those observations). Based on the transition matrix, we can see that probability of transitioning from state 2 to state 4 is extremely high (~1.0). This makes sense as state 2 represents articles and state 4 represents nouns. In Shakespeare's sonnets, articles are often followed by nouns. In contrast to typical grammatical structure (subject, verb, and then object), Shakespeare uses SOV pattern. The HMM with large number of hidden states captures there qualities as they learn to generate states representing nouns (subjects and objects), and verbs. In addition, the transition matrix has higher transition probabilities from subject to object state and object to verb state.