Machine Learning & Data Mining
Miniproject 3

Caltech CS/CNS/EE 155
Released March 6^{th}, 2020

---

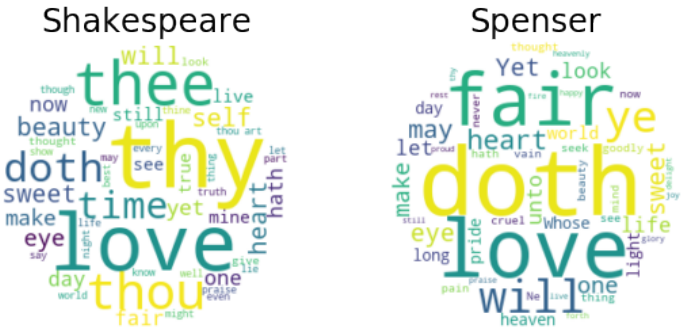
1 Introduction

- **Group Members**
  Liting Xiao.
- **Team Name**
  Turtle School Son Goku.
- **Division of Labor**
  The whole of the project, including data exploration, data pre-processing, poetry generation via HMM and RNN, visualization and interpretation, and writing of this report.
- **Github Repo**
  https://github.com/litingxiao/william-wanna-shake-pear

2 Basic Pre-Processing

In this section, I talk about the basic pre-processing techniques used for the naive Hidden Markov Model (HMM) and the naive Recurrent Neural Network (RNN) implementation. More "advanced" pre-processing endeavors geared towards more "advanced" HMM and RNN will be discussed in Section 6 Additional Goals.

Before pre-processing, let's visualize the two datasets *shakespeare.txt* and *spenser.txt* using *WordCloud*. (Note that the basic pre-processing and implementation in Sections 2–5 used ONLY *shakespeare.txt*. Advanced pre-processing and implementation in Section 6 will include the additional dataset *spenser.txt*.)



From the *WordCloud*s, we can see that Shakespeare used certain old English words very often, such as "thee", "thou", "thy", "doth". And both Shakespeare and Spencer talked about "love" a lot in their sonnets.

- **Word-based Basic Pre-processing for Naive HMM**

  For the naive HMM, I tokenized the dataset into words, and I considered each line as a singular sequence. This is based on the fact that a line is the smallest unit that contains a complete flow of ordered semantic meanings. Word-based tokenization and line-based sequencing will enable a simple HMM to learn about the relations between neighboring words. And we can avoid non-sensible words

appearing in text generation in implementation using char-based tokenization. Of course, we can also try sequencing each quatrain/couplet, or each poem, but I do not see the benefits of doing them over the simple line-based sequencing.

I tried using *nltk*'s *TweetTokenizer* for tokenization, but it did not handle apostrophes in all cases that well. So the actual easiest way is to use string's *split()* function line by line.

After reading in the input file *shakespeare.txt* line by line, I first made all texts in each line lower case so that the program would treat all words equally and wouldn't take, for example, "Thou" and "thou" differently. Then I splitted each line into words. From there, I removed all numerical numbers 0–9 irrelevant to the poems. Then I removed all punctuations except for apostrophes and hyphens to keep hyphenated words and words containing apostrophes as individual words. Lastly, I removed extra empty lists.

The total number of sequences is 2155. And the complete vocabulary in *shakespeare.txt* processed this way contains 3231 words. To convert words into machine-readable vectors, I assigned a number to each unique word. And I saved the mapping from words to vectors and the inverse mapping from vectors to words into pickle files. I also saved the list of sequences of vectors (of words) into a pickle file.

- **Char-based Basic Pre-processing for Naive RNN**

  Char-based pre-processing contains many repeated processes as the word-based pre-processing. I read in the file as a whole instead of line by line and made all texts lower case. I removed all numerical numbers and punctuations except for apostrophes and hyphens. I removed empty strings and combined all words into one very long string to get n-char sequences. In our case, we chose n to be 40.

  Here we defined 40 characters to be a sequence, which is a sensible choice since roughly each line contains $\sim$40 chars (the median number of chars in a line is 41). Then I sieved through the long string of raw text, advancing 1 char at a time, and obtained 90965 40-char sequences. To convert chars into machine-readable vectors, I assigned a number to each unique char. There are 29 unique chars processed this way (26 English chars and a hyphen, an apostrophe, and a space).

  I tried including the newline character "\n" into the char vocabulary, but it did not work well in later poetry generation since it sometimes generated newlines too early/too late.

  Lastly, I saved the mapping from chars to vectors and the inverse mapping from vectors to chars into pickle files. And I also saved the list of sequences of vectors (of chars) into a pickle file.

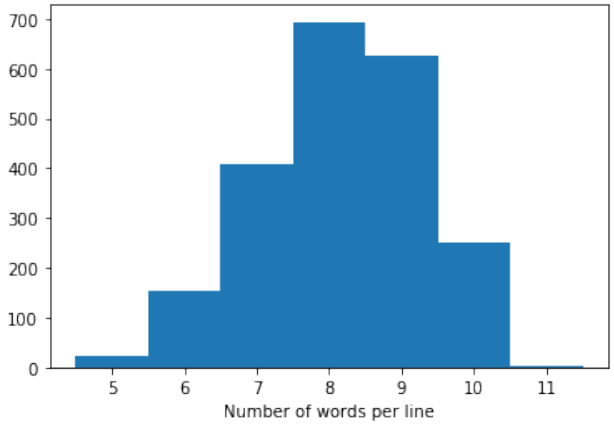
3 Unsupervised Learning: Hidden Markov Models

For the HMM implementation, I did not use any off-the-shelf packages. I used the Baum-Welch algorithm that we implemented in HW6.

After loading the word-based pre-processed data, I trained HMMs with {4, 8, 16, 32} hidden states, outputted sample sentences of 8 words from the trained HMMs, and saved all trained HMMs. Here I generated sample sentences of 8 words, because that's the median number of words in a line in the Shakespearean sonnets. From the sample sentences (see them in notebook *2-HMM-Train-Poems.ipynb*), subjectively, I think that the HMM with 32 hidden states generated sentences that made the most sense. Therefore, in the poetry generation in Section 4, I used the HMM with 32 hidden states.

4 Poetry Generation, Part 1: Hidden Markov Models

To generate a 14-line sonnet, I first randomly chose the number of words in each line with some probabilities. I noticed that the median of the number of words each line is 8, the minimum length is 5 and the maximum length is 11. So I chose randomly from 5–11 with probabilities to mimic the distribution of number of words per line shown in the plot below.



Then I generated a long list of all the words in a sonnet via HMM's function *generate_emission()*. I generated all words in one-go instead of line by line because in this way all the emission states are connected to the previous states, instead of generating a random state at the start of each line. Lastly, I broke the long list of words into lines using the previously sampled number of words per line.

Here's one sample sonnet created this way using *'basic_hmm32'*:

```
Have i mock these and my verses to the
  Uneared things do woman lives lend his me
  To praise favour she spent live in toiled wonder
  Of my way if with told to publish
Past new tend else wealth i his fortune
  Doth nine seeting pride am with the heard
```

```
  Strength of power wretched near
  Jewels crowned gave for fair love love hath
My fair cries point respose so hasten life 'tis
  Vengeful love's worth hast thou beauty of proudly breathe
  Hell rich sport as mock loving child
  Ah breed they black as love with bereft
Good bequest swear in of the prime
  Be seem a black confined wary and this
```

Here's another sample sonnet:

```
That in eve's mounted brave any eye and
  What give thou i my amis but
  Thou most sepulchres thy life old it
  We contrary to in goest poor which praises
Reeks he can hear than fair wail if
  Scythe translate make bark the weed bark balmy very a
  Mud well was buried homage swears thou that with
  Mine are guilty of thy outward and pretty die
The my scanted that bitterness that
  Forsake and eyes the inherit your fortune not of
  Not is eyes a own urge thee welcome
  The want to be in fair proposed that
Is't thunder treasure this longer but lost shall which
  His that works thou leaves did worse it
```

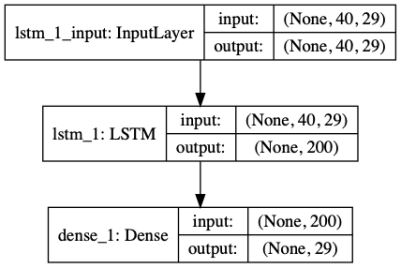
We can see that although the samples contain some sensible phrases, they do not have good rhymes, rhythms, and syllable counts compared to a real sonnet. I don't think the "poems" make much sense except that they contain some valid phrases. They do, for a small part, retain Shakespeare's original voice through the use of old English words and somewhat weird sentence structures. Qualitatively, training with more hidden states seems to generate more valid phrases and words are more likely to be connected to their neighbors.

Unsupervised learning in HMMs constantly updates its state transition matrix A and the observation matrix O through Baum-Welch algorithm with incoming data. With ample data, HMMs learn that some words have higher probabilities to appear near some other words and thus form some cluster-like word groups. This is probably why some common phrases appear in the generated texts. But with limited information, the naive HMM could only do so much. Hopefully, with more information provided in the advanced data pre-processing, HMM will do a better job.

5 Poetry Generation, Part 2: Recurrent Neural Networks

I used *Keras* for the RNN implementation. The first step is to load the char-based pre-processed data described in details in Section 2. After loading the pre-processed dataset, I used it to generate (x, y) pairs for RNN training, with x being each 40-char sequence, and y being the immediate next character after each 40-char sequence x. Then I used '*keras.utils.to_categorical()*' to transform the (x, y) pairs of (vectors, numbers) to (matrices, vectors), with 1 being in the index of the char and 0 otherwise in a vector to signify a specific character. Now, the data are ready for RNN training.

For the RNN model, I used a simple model with 1 hidden layer of 200 LSTM units. And the output layer is a fully-connected dense layer with a softmax nonlinearity. The plot below on the left shows the RNN network flow. The plot on the right again shows the architecture of the RNN model, and it also shows that the model contains 189,829 trainable parameters. I trained my model to minimize categorical cross-entropy for 60 epochs using a batch size of 128, with 'adam' optimizer. I then saved my model.



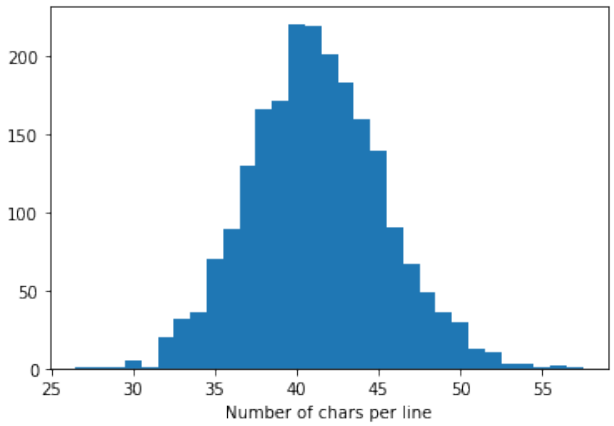

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_6 (LSTM)                (None, 200)               184000
_________________________________________________________________
dense_6 (Dense)              (None, 29)                5829
=================================================================
Total params: 189,829
Trainable params: 189,829
Non-trainable params: 0
_________________________________________________________________
```

After 60 epochs, the model reached a categorical cross-entropy loss of $\sim$0.41, and an accuracy of $\sim$0.87.

```
Epoch 57/60
90965/90965 [==============================] - 43s 475us/step - loss: 0.4204 - acc: 0.8646
Epoch 58/60
90965/90965 [==============================] - 44s 489us/step - loss: 0.4214 - acc: 0.8641
Epoch 59/60
90965/90965 [==============================] - 44s 482us/step - loss: 0.4219 - acc: 0.8627
Epoch 60/60
90965/90965 [==============================] - 44s 482us/step - loss: 0.4095 - acc: 0.8675
```

Finally, to make predictions (generate poems), I added a Lambda layer between the LSTM and the output layer to incorporate temperatures into the model. A higher temperature means a higher variance and vice versa. In generating sample sonnets, I used temperatures of 1.5, 0.75, 0.25.

For the actual implementation, first provide the model with a seed. Then the model makes a prediction of the next char according to the seed, and it advances from there. Here we used a 40-char seed "*Shall i compare thee to a summer's day*". For generating each line of the poem, I chose randomly the number of chars in a line from a Gaussian distribution centered at 45 (=40+5), with a standard deviation of 5. The extra 5 chars act as a buffer for stripping the last "word" (the last few chars after the last space) from the generated line so that we don't have incomplete words. Using a Gaussian to choose the number of chars is justified by the following plot of the distribution of number of chars in each line.



The generated poems with temperatures 1.5, 0.75, 0.25 are shown below. For better comparison, I kept the number of chars each line, respectively, the same for all three temperatures.

For a temperature of 1.5, one generated poem is:

```
Shall I compare thee to a summer's day
  Of you are to outin and for my self
  Rosel and all my friend and your true
  Love alane exes to the very was of hight
Do I not for my self I spend all plack
  Should do a lear that in the wrickle glacter
  Tell come how I am for while in their
  Putcons injures home and beauty looking
When in your sweet self to be so thus did
  The sime the deages ngrest and that beauty that I do
  Cold concest of the tears that I
  In other place or glasoul his
With the trouning that the thought of
  Hearts can beauty to the wirned some with
```

For a temperature of 0.75, one generated poem is:

```
Shall I compare thee to a summer's day
  Of you are to outin and for my self
  Rosel and all my friend and your true
  Love alane exes to the very was of hight
Do I not for my self I spend all plack
  Should do a lear that in the wrickle glacter
  Tell come how I am for while in their put
  Sweets with thoughts my love shall in my
Verse as and and spire as you to the vise
  Of worts do I honour his benunt in thee and love's
```

```
  Love away thick my mistress
  Weep-lime un hearts are dead
So I love thee to the world will not be
  The see all by that I may gazer should be
```

For a temperature of 0.25, one generated poem is:

```
Shall I compare thee to a summer's day
  Of you are to outin and for my self
  Rosel and all my friend and your true
  Love alane exes to the very was of hight
Do I not for my self I spend all plack
  Should do a leaves regue to be so true that is
  Not for my self I spend what not in good
  Fair cortone O that I am forsay despite
Of soul is in my love say on appetit in
  Our to this sing and praise be so I do not love thee
  When I am despised who have whet
  Of thee hast thou then love
When I am seem so compare the still weter
  Delight that I coly better for their
```

From the three generated poems, we see that the naive RNN performs better than the naive HMM. The poems by the naive RNN have better connected phrases and sentence structures than those by the naive HMM. But of course, the naive RNN required more runtime/amount of training data compared to the naive HMM. I ran the training on GPUs in Google Colab.

Comparing the three poems with different temperatures, we see that the poems with the highest temperature 1.5 and the lowest temperature 0.25 are less "stable" in the sense that fewer words/phrases made sense than the poem with a temperature of 0.75. The poem of temperature 0.25 has less variety in word choices. The poem of temperature 1.5 has too much variety so that it contains fewer valid phrases. Overall, the model with a temperature of 0.75 produced the best written poem so far (naive HMM and naive RNN), in terms of words, phrases, and sentences.

6 Additional Goals

- **Advanced Pre-processing**

  - **Additional Texts**

    For training the advanced HMM and RNN models, I went through the same text cleaning and word-based (HMM) / char-based (RNN) tokenization processes as described in section 2 on both provided files *shakespeare.txt* and *spenser.txt* for a bigger Shakespearean poem training set.

    For the advanced HMM, the total number of line-based sequences is 3401, while the total number of unique words is 4391. Note here, 2 sets of sequences were created, one with normal ordering and one with inverted ordering. The reason would become clear in the next sub-subsection *Syllable Count*.

    For the advanced RNN, the total number of unique chars is 38, including [' ', '!', '&', '"', '(', ')', ',', '-', '.', ':', ';', '?'] in addition to the 26 English chars. To produce 40-char sequences, I no long combined all of the poems into one giant string and advanced 1 char at a time. Instead, I used poem-based sequencing, meaning that I created 40-char sequences from the start of a poem to the end of it, then restart on the start of the next poem. By doing it this way, poetic meaning is somewhat preserved within sequences. And hopefully, this will improve later poem generation in terms of sentence structures and poetic meanings. Also, this reduced the amount of sequences used for training to speed up training time. The total number of 40-char sequences processed this way is 137,247.

    (The following advanced pre-processing techniques are specific to the advanced HMM model.)

  - **Syllable Count**

    Most of Shakespearean sonnets have 10 syllables per line. One way to improve the HMM is to incorporate syllable count information.

    So, in pre-processing, I read in the file *Syllable_dictionary.txt* that contains all syllable count info in *shakespeare.txt* and saved the info into a dictionary *word2syllable*. To create a bigger vocabulary, I also obtained syllable count info from *spenser.txt* using a light-weight package *pronouncing* (https://pronouncing.readthedocs.org). Originally, I was using *nltk*'s *cmudict*, but it was way too slow to run. And I found this package to be much faster and much easier to implement. It could easily and instantly output a word's pronunciation list, syllable counts, stresses, and a list of words that rhymes with the given word.

    Using *pronouncing*, I obtained syllable counts of all words from *spenser.txt* and stored those that weren't already in *word2syllable* and saved the dictionary into a pickle file for later use.

  - **Rhyme**

    Shakespeare's sonnets have a strict rhyming rule "ABAB CDCD EFEF GG". A second thing that could be done to improve the HMM is to make generated poems follow this rhyming scheme.

    For each unique word in the vocabulary, I used *pronouncing.rhymes()* method to get a list of all words that rhymes with it. Then I only kept the rhyme words that are also in the vocabulary. Lastly, I saved the dictionary that contains mapping from words to their rhyme words into a pickle file.

- **Rhythm/Stress**
  pronouncing's *stresses()* function also allows easy access to a word's certain pronunciation's stress information. However, by the way I implemented syllable counts into the poetry generation, incorporating stresses will complicate the process a great deal. Hence, considering time constraint, and given that advanced HMM generated poems still lack cohesive semantic meanings, I did not incorporate stresses in poetry generation.

- **Advanced HMM**

  With training data ready, next step here is to train an unsupervised learning HMM using the bigger dataset. I directly chose 32 hidden states given the results in naive HMM. Now that we have the advanced HMM, syllable count dict and rhyme dict, we are ready to generate poems.

  For every 2 lines in a poem, I first generate a pair of words that rhymes. To do this, I randomly choose a start state and an observation until the observation falls into the rhyme dictionary (meaning that until I find a word that has rhyme words in our vocabulary). I then select a random rhyme word from its rhyme words list in our vocabulary.

  These 2 rhyme words are placed at the end of lines. I then start state transitioning and observation generation in the reverse order, starting at the end of a line. Note here since I need to reverse generated word lists in the end, I trained one advanced HMM using sequences in the reversed order. I also trained one advanced HMM using sequences in the normal order. From generated poems, we see that ordering doesn't make a big difference.

  To get desired syllable counts, I keep track of the syllable count in each line. Once a line hits 10, the line is done. If it goes over 10, I delete the word and state that are just generated and retry.

  Once lines are generated, I place each line marked by the rhyme pairs alternatingly in the 3 quatrains and together in the couplet.

  One sample poem generated using normal ordering training is:

```
With just whom in rehearse shame be living
  Heart your fire this to fate feeble of first
  A figure their race wink and thou giving
  How be glory new of all-tyrant worst
Which married up whence breasts love's did that hard
  Make should admire toil not love the on you
  With not again full she fashion regard
  Of down frown thy aught of sleep fair sweet due
Teachest the taken amis tell lie which
  Me her heaven myself end the see spill
  Two chaste that sea nor art my guard to pitch
  Her chain that no may world sweet in goodwill
First-born stoutly and art sweet remove to
  Life my it to be worth paws finds good two
```

One sample poem generated using reversed ordering training is:

```
The large numbers must due did be you sweet
  Love thou have black heaven love hair was stand
  Beauteous the large least in the building seat
  Child be numbers to fair fearless men brand
The happy dress came now would my thy gait
  It they and that clay all she glory that
  Thou was me should tears in they art the mate
  The message and thine once make thine loud at
Beseechers rack for it miss print said thou
  Sue glorious mine freshly affections growth
  Praise to have same feeds in tongue and then how
  Back seeing the black sail so my is loath
Thoughts ever in in one in thine been turn
  True none thou wolf ever or she return
```

I also used the advanced HMM to generate poems of other forms. The simplest to implement is Haiku. Haiku poems are only 3 lines long, and they follow a syllable count scheme of 5-7-5. I generated Haiku poems similarly to how I generated Shakespearean poems.

One sample Haiku poem is:

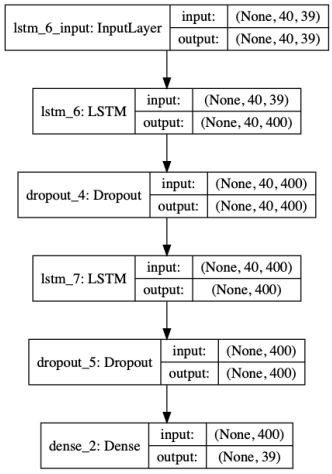
```
Abhor with to work
Foolish art sweet of thy
Truth twice devise
```

- **Advanced RNN**

  One obvious way to improve the RNN is to use a more complex model (besides using more data).

  I pre-processed the bigger dataset in a similar fashion as in Section 2, also char-based. But this time, I kept all punctuations. I didn't keep the newline char in training since it might end line too early/late or end word abruptly. Keeping punctuations is to ask the RNN to figure out when to break up parts of sentences or end a sentence.

  For the advanced RNN, I built a deeper and wider model using 3 LSTM hidden layers of 600 units, each accompanied by 20% dropout. And the output layer is again a fully-connected dense layer with a softmax nonlinearity. The plot below on the left shows the RNN network flow. The plot on the right again shows the architecture of the more advanced RNN model, and it also shows that the model contains 7,321,238 trainable parameters. I trained my model to minimize categorical cross-entropy for 30 epochs using a batch size of 128, with 'adam' optimizer. I then saved my model.



Layer (type)	Output Shape	Param #
lstm_6 (LSTM)	(None, 40, 600)	1536000
dropout_1 (Dropout)	(None, 40, 600)	0
lstm_7 (LSTM)	(None, 40, 600)	2882400
dropout_2 (Dropout)	(None, 40, 600)	0
lstm_8 (LSTM)	(None, 40, 600)	2882400
dropout_3 (Dropout)	(None, 40, 600)	0
dense_5 (Dense)	(None, 40, 39)	23439

Total params: 7,324,239
Trainable params: 7,324,239
Non-trainable params: 0

After 30 epochs, the model reached a categorical cross-entropy loss of $\sim$0.42, and an accuracy of $\sim$0.86.

```
Epoch 27/30
140406/140406 [==============================] - 277s 2ms/step - loss: 0.4363 - acc: 0.8518
Epoch 28/30
140406/140406 [==============================] - 279s 2ms/step - loss: 0.4289 - acc: 0.8555
Epoch 29/30
140406/140406 [==============================] - 277s 2ms/step - loss: 0.4260 - acc: 0.8552
Epoch 30/30
140406/140406 [==============================] - 277s 2ms/step - loss: 0.4204 - acc: 0.8588
```

For poetry generation, I ask the trained advanced RNN to keep generating characters until a specified max_char has been reached. Here, max_char is set to 800. This should contain more than enough characters for a 14-line sonnet, with 40-char per line as its median. Then I break down the generated 800 chars into a list of "lines" separated by line breaker punctuations including **.,!?:;**

I consider each "line" to be a line in the final output with one exception. If a "line" has fewer than 20 chars, I will attach the next "line" to it and consider the new "line" to be a line in the final poem. Then, the first 14 lines are outputted as the final poem.

With this advanced RNN with much more parameters, using a seed from the original text would produce the original poem or something close to it (overfitting!). Therefore, instead, I found some lines in some old English poems not written by Shakespeare or Spenser and used them as seeds (first lines of poems), and generated three sample poems using a temperature of 0.75. (The temperature choice was based on the results from the naive RNN.)

One sample poem is:

```
Great was the glory then gained in the fight.
  'tis thee (my self) no motion show what wealth,
  Some seem so true, in vainted compound with the stormy part.
  With light thereof i do myself that i in many dear delight,
The dost be mind the better part of me,
```

```
  So thou previge the sun,
  For they shall i most ore,
  And the firm soil win of the world,
Unbless some mother.   Present the time with thoughts canst move,
  And i am still with thee,
  When thou from thee that said i could not the greater scath,
Through sweet illusion of her look's delight.
    Therefore i lie with her.
```

Another sample poem is:

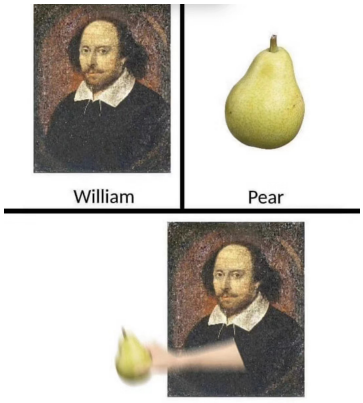
```
Then the powerful king put to the test,
  Both soul doth say thy show,
  The worst of worth the words of weather ere:
  And with the crow of well of thine to make thy large will more.
Let no unkind, no fair beseechers kill,
  Though rosy lips and lovely yief.
  Her wratk doth bind the heartost simple fair,
  That eyes can see! take heed (dear heart) of this large privilege,
And she with meek heart doth please all seeing,
  Or all alone, and look and moan,
  She is no woman, but senseless stone.
  But when i plead, she bids me play my part,
And when i weep, in all alone,
    That he with meeks but best to be.
```

Another sample poem is:

```
My robe is noiseless when i roam the earth,
  Which her fair child expire,
  Shall to a baser made,
  And soon to temper that my wit cannot endite.
When suddenly with thine eye but with thy time decays?
  O fearful meditation,
  Where alack, shall time's best jewel will be took,
  And they that skill not of the world's fresh care,
And me for me than shortune your self still,
  And therefore to be seen so thy great growing age,
  A dearer birth than thine eyes seem so.
  If it be not, the which three times thrice haply hath me,
Suffine in them shall still will play the tyrant,
    The which beholding me with melancholy.
```

From these 3 poems, we could see that the poems now make much more sense. There are some simple sentence structures and most phrases are valid. There exist bags of words that are related to each other throughout the poems.

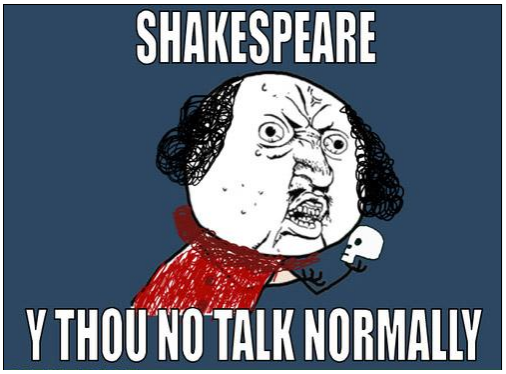
Just for fun, I named this poem-writing RNN to be *William-wanna-shake-pear*.



Now that I have trained a naive HMM, a naive RNN, an advanced HMM, and an advanced RNN. Comparing their performances, I would say that the advanced RNN, *William-wanna-shake-pear*, worked the best (precisely why I named it).

In actual deployment, there are ways to further improve it (with more poem data, maybe use different RNN architectures such as ones with GRUs, maybe use word embeddings, train with more epochs) and deploy *William-wanna-shake-pear* to generate machine-written poems.

Finally, if the trained AI above, *William-wanna-shake-pear*, could really think, I believe its mood would be like this:



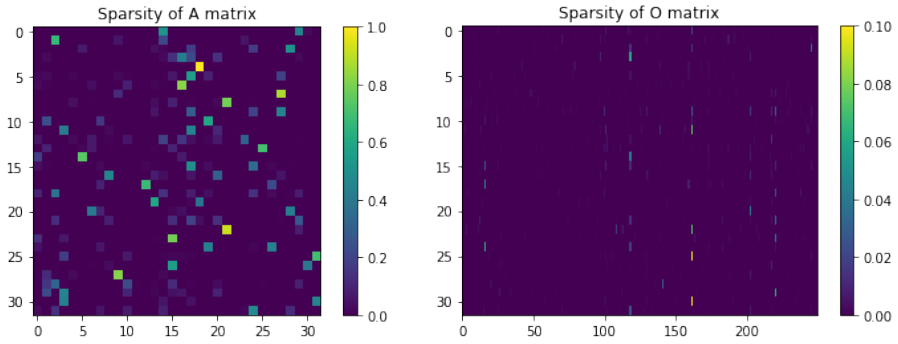
7 Visualization and Interpretation

(Throughout the implementation and the writing of this report, I have included some visualization and interpretation along the way for ease of understanding. Below, I will be presenting additional visualization and interpretation insights.)

I have trained 3 HMMs, one naive HMM, one advanced HMM trained using normal ordering of sequences, and one advanced HMM trained using reversed ordering of sequences. I will analyze only the second one, the advanced HMM trained using normal ordering of sequences. Analyses on the other two could be done in a similar fashion.

With the trained HMM, we can first visualize the **sparsities** of the state transition matrix A and the observation matrix O to get a broad sense of how it works. This is done by using *visualize_sparsities()* function from *HMM_helper.py*.

From the sparsity plots, we can see that some states have stronger preferences to transition to certain designated states (left plot), and some states have high probabilities in generating a few certain words (right plot). Some states have a number of states that they like transitioning to, but some states prefer one or two specific states for transition. Each state collects a bag of words that share some common features, and each observation is a word. Note that we can not show the sparsity of the entire O matrix, only the first 250 indexed observations are shown.



At the state level, let's get a sense of the type of words associated with each state by plotting the *WordClouds* of all 32 hidden states, with **10 most frequent words** plotted for each state. This is done by using *states_to_wordclouds()* function from *HMM_helper.py*.

By examining the top 10 words from each state, words from some states show common features easily. For example, state 5 contains mostly action verbs, such as {sing, give, take, stand, make, live, tell, stay, speak}; state 6 contains some auxiliary verbs such as {must, will, may, might}; state 19 contains mostly nouns {beauty, eye, night, time, love, things, one} and some adjectives that could describe them {fair, old, better}; state 29 is marked by words like {yet, dost, now, though, whether, ye, ere}. State 22 only has 3 words, {weren't, perhaps, lean}. We can also see that a few words appear repeatedly in several different states as among top 10, such as {thy, thee, thou, doth, yet}. This seems reasonable considering they are among the most frequent words in the original datasets. From these examples, we could clearly see that HMM collects words that share some common features into the same states.

At the interstate level, we can plot the **transitions** between states indicated by arrows, with the blackness of arrows indicating the higher values of probabilities. This is done by adapting *animate_emission()* function from *HMM_helper.py*. This plot isn't particularly easy to see because of the huge number of hidden states in the HMM. We can see that most transitions have weak probabilities (white arrows) but there are just a few number of strong transitions (grey and black arrows) between states. And notice that state 16 and 17 are two very popular transition stops for other states.