1 Class-Conditional Densities for Binary Data [25 Points]

Problem A [5 points]: Parameters of Full Model with Factorizing

Solution A.:

$$\theta_{xjc} = P(x_j|x_{1,\ldots,j-1}, y)$$

Assuming a uniform class prior:

$$P(x|y) = P(x_1|y)P(x_2|x_1, y)P(x_3|x_{1,2}, y)\ldots P(x_j|x_{1,\ldots,j-1}, y)$$

In this factorization, there are 2^D possible sequences for every possible state $c \in C$, so there are $2^D C$ parameters needed to represent the factorization. This becomes O($2^D C$).

Problem B [5 points]: Parameters of Full Model without Factorizing

Solution B.: *Without factorization, we end up with the same number of parameters as the factorized state ($2^D C$). There are still 2^D possible sequences for each case. So, it appears there is no benefit to doing the factorization (at least in terms of the number of parameters stored), the number of parameters remains O($2^D C$).*

Problem C [2 points]: Naive Bayes vs. Full Model for Small N

Solution C.: *If we assume the number of features is fixed, and there are very few training cases, it is likely that Naive Bayes will give lower test set error. We can view this in the lens of the bias variance tradeoff. Naive Bayes has much fewer parameters, so with a small amount of data, this kind of high bias, low variance model will be superior.*

Problem D [2 points]: Naive Bayes vs. Full Model for Large N

Solution D.: *If, however, the sample size is large, it is likely that the full model will give lower test set error (assuming that the test dataset comes from a similar distribution to the train dataset). We have enough data to train a more highly parameterized model and lower bias.*

Problem E [11 points]: Computational Complexity of Making a Prediction Using Naive Bayes vs Full Model

Solution E.: *Assuming all the parameter estimates have been computed, the computational complexity of making a prediction for Naive Bayes is O(DC), getting D features for each class* $c \in C$*. However, the full model would be O(D + C) (which could be simply considered O(D), since* $C << D$*), because it is just the cost to index. The full cases ends up with C tables that can be indexed. Interestingly, it appears that the full model provides a computational advantage in the prediction stage. This could be very useful in cases where you have the ability to front-load your computation, but need faster prediction.*

2 Sequence Prediction [75 Points]

Problem A [10 points]: Max-Probability State Sequences for 6 Trained HMMs

Solution A.:

```
######################################################################
                     Running Code For Question 2A
######################################################################


File #0:
Emission Sequence             Max Probability State Sequence
######################################################################
25421                         31033
01232367534                   22222100313
5452674261527433              1031003103131333
7226213164512267255           1310331000033100313
0247120602352051010255241     2222222222222222222222103


File #1:
Emission Sequence             Max Probability State Sequence
######################################################################
77550                         10313
7224523677                    2222221003
505767442426747               222100003310003
7213413164553611226 7         1031031000031033310 3
4733667771450051060253041     2221000003222223103222223


File #2:
Emission Sequence             Max Probability State Sequence
######################################################################
60622                         11102
4687981156                    2102020202
815833657775062               021011111111112
2131022251596350501 5         0202011111111111102 2
6503199452571274006320025     1110202111111102021110202


File #3:
Emission Sequence             Max Probability State Sequence
######################################################################
13661                         13333
2102213421                    3131310213
166066262165133               133333133133133
5316466211216263415 6         2000002131313100213 3
1523541005123230226306256     1310021333133133133133133
```

Figure 1: Output from 2A.py

```
File #4:
Emission Sequence             Max Probability State Sequence
######################################################################
23664                         01124
3630535602                    0111201244
350201162150142               011244012441244
00214005402015146362          11201112412444011124
2111266524665143562534450     2012012424124011112411124


File #5:
Emission Sequence             Max Probability State Sequence
######################################################################
68535                         10102
4546566636                    1111111102
638436858181213               110111010000002
13240338308444514688          00010000000111111102
0111664434441382533632626     2111111111111100111110102
```

Figure 2: Second Half output from 2A

Problem B [17 points]: Probability of Emission Sequence for 6 Trained HMMs

Solution B:

```
[(base) Alexanders-MacBook-Pro-3:code afarhang$ python 2A.py          -3:code afarhang$ python 2Bii.py


######################################################################   #########################################
                     Running Code For Question 2A                        ode For Question 2Bii
######################################################################   #########################################


File #0:
Emission Sequence             Max Probability State Sequence              Probability of Emitting Sequence
######################################################################   #########################################
25421                         31033                                       4.537e-05
01232367534                   22222100313                                 1.620e-11
5452674261527433              1031003103131333                            4.348e-15
7226213164512267255           1310331000033100313                         4.739e-18
0247120602352051010255241     2222222222222222222222103                   9.365e-24


File #1:
Emission Sequence             Max Probability State Sequence              Probability of Emitting Sequence
######################################################################   #########################################
77550                         10313                                       1.181e-04
7224523677                    2222221003                                  2.033e-09
505767442426747               222100003310003                             2.477e-13
72134131645536112267          10310310000310333103                        8.871e-20
4733667771450051060253041     2221000003222223103222223                   3.740e-24


File #2:
Emission Sequence             Max Probability State Sequence              Probability of Emitting Sequence
######################################################################   #########################################
60622                         11102                                       2.088e-05
4687981156                    2102020202                                  5.181e-11
815833657775062               021011111111112                             3.315e-15
21310222515963505015          02020111111111111022                        5.126e-20
6503199452571274006320025     1110202111111102021110202                   1.297e-25


File #3:
Emission Sequence             Max Probability State Sequence              Probability of Emitting Sequence
######################################################################   #########################################
13661                         13333                                       1.732e-04
2102213421                    3131310213                                  8.285e-09
166066262165133               133333133133133                             1.642e-12
53164662112162634156          20000021313131002133                        1.063e-16
1523541005123230226306256     1310021333133133133133133                   4.535e-22


File #4:
Emission Sequence             Max Probability State Sequence              Probability of Emitting Sequence
######################################################################   #########################################
23664                         01124                                       1.141e-04
3630535602                    0111201244                                  4.326e-09
350201162150142               011244012441244                             9.793e-14
00214005402015146362          11201112412444011124                        4.740e-18
2111266524665143562534450     2012012424124011112411124                   5.618e-22


File #5:
Emission Sequence             Max Probability State Sequence              Probability of Emitting Sequence
######################################################################   #########################################
68535                         10102                                       1.322e-05
4546566636                    1111111102                                  2.867e-09
638436858181213               110111010000002                             4.323e-14
13240338308444514688          00010000000111111102                        4.629e-18
0111664434441382533632626     2111111111111100111110102                   1.440e-22
```

Figure 3: Combining the output of 2A and 2Bii(Backwards) to recreate the table

```
                    Running Code For Question 2Bi
######################################################################


File #0:
Emission Sequence              Probability of Emitting Sequence
######################################################################
25421                          4.537e-05
01232367534                    1.620e-11
5452674261527433               4.348e-15
7226213164512267255            4.739e-18
0247120602352051010255241      9.365e-24


File #1:
Emission Sequence              Probability of Emitting Sequence
######################################################################
77550                          1.181e-04
7224523677                     2.033e-09
505767442426747                2.477e-13
72134131645536112267           8.871e-20
4733667771450051060253041      3.740e-24


File #2:
Emission Sequence              Probability of Emitting Sequence
######################################################################
60622                          2.088e-05
4687981156                     5.181e-11
815833657775062                3.315e-15
21310222515963505015           5.126e-20
6503199452571274006320025      1.297e-25


File #3:
Emission Sequence              Probability of Emitting Sequence
######################################################################
13661                          1.732e-04
2102213421                     8.285e-09
166066262165133                1.642e-12
53164662112162634156           1.063e-16
1523541005123230226306256      4.535e-22


File #4:
Emission Sequence              Probability of Emitting Sequence
######################################################################
23664                          1.141e-04
3630535602                     4.326e-09
350201162150142                9.793e-14
00214005402015146362           4.740e-18
2111266524665143562534450      5.618e-22


File #5:
Emission Sequence              Probability of Emitting Sequence
######################################################################
68535                          1.322e-05
4546566636                     2.867e-09
638436858181213                4.323e-14
13240338308444514688           4.629e-18
0111664434441382533632626      1.440e-22
```

Figure 4: Output from 2Bi(Forwards)

Problem C [10 points]: Learned State Transition and Output Emission Matrices of Supervised Hidden Markov Model

Solution C.:

```
######################################################################
                     Running Code For Question 2C
######################################################################


Transition Matrix:
######################################################################
2.833e-01   4.714e-01   1.310e-01   1.143e-01
2.321e-01   3.810e-01   2.940e-01   9.284e-02
1.040e-01   9.760e-02   3.696e-01   4.288e-01
1.883e-01   9.903e-02   3.052e-01   4.075e-01


Observation Matrix:
######################################################################
1.486e-01   2.288e-01   1.533e-01   1.179e-01   4.717e-02   5.189e-02   2.830e-02   1.297e-01   9.198e-02   2.358e-03
1.062e-01   9.653e-03   1.931e-02   3.089e-02   1.699e-01   4.633e-02   1.409e-01   2.394e-01   1.371e-01   1.004e-01
1.194e-01   4.299e-02   6.529e-02   9.076e-02   1.768e-01   2.022e-01   4.618e-02   5.096e-02   7.803e-02   1.274e-01
1.694e-01   3.871e-02   1.468e-01   1.823e-01   4.839e-02   6.290e-02   9.032e-02   2.581e-02   2.161e-01   1.935e-02
```

Figure 5: Learned State transition and output emission matrices for ron.txt

Problem D [15 points]: Learned State Transition and Output Emission Matrices of Unsupervised Hidden Markov Model (use the seeds specified in the Piazza post)

Solution D.:

```
######################################################################
                     Running Code For Question 2D
######################################################################


Transition Matrix:
######################################################################
8.029e-04   5.274e-01   8.376e-05   4.717e-01
1.856e-03   6.830e-01   2.922e-01   2.303e-02
6.214e-01   8.278e-13   3.747e-01   3.940e-03
2.051e-02   7.025e-01   2.042e-02   2.566e-01


Observation Matrix:
######################################################################
1.577e-01   4.863e-02   1.835e-01   4.863e-02   2.072e-01   3.686e-21   6.874e-02   2.040e-02   2.653e-01   1.458e-35
1.120e-01   5.788e-02   1.132e-01   1.021e-01   1.309e-01   9.003e-02   8.507e-14   1.870e-01   1.437e-01   6.304e-02
9.063e-02   7.887e-02   1.014e-16   1.937e-01   6.975e-02   2.191e-01   1.481e-01   9.426e-17   5.298e-02   1.468e-01
3.079e-01   1.320e-01   9.116e-02   3.171e-02   5.488e-03   1.068e-06   2.835e-01   6.466e-02   8.364e-02   4.515e-07
```

Figure 6: Learned state transition and output emission matrices for Ron's third roommate

Problem E [5 points]: Compare 2C and 2D

Solution E.: *Transition Matrices: 2D provided a much sparser transition matrix. For example, let's compare moving from State 3 to state 2 (2C: 9.7e-2, 2D: 8.2e-13). This actually seems to be the case across all states. While the supervised case gives a* $10^{-1\,or\,-2}$ *transition probability, 2D provides a much larger range from* $10^{-13\,:\,-1}$*. There tends the probability mass much concentrated much more in 1 or 2 specific states to transition to.*
Observation Matrices: These also appear a bit sparser now. There are some probabilities that have dropped to basically zero: $10^{-21,-16,-14}$*. This makes sense, if Ron is in a grumpy mood and unwilling to become happier, he might be exceedingly unlikely to want to listen to his most upbeat happy song.*
I think that the unsupervised (2D) models provide more accurate representations of Ron's moods and how they affect his music choices.
Though supervised methods are considered more accurate in many applications, their success comes down to how the data were generated. And in this case, the states are determined (probably arbitrarily) by a roommate's observations. It is unlikely that the roommate is a perfect state estimator, furthermore, he might not have even picked the states that explain the music choices optimally. The Unsupervised method, however, finds the maximally likely states (assuming a given number of states) to explain the data. However, as a counterpoint, if the labeled states were instead coming from Ron (self-labeled) or perhaps some sort of behavioral scientist who was an expert at this, the supervised version might be better. To improve the supervised method, this could be done, or perhaps the number of states and their identities could be more appropriately tuned. In fact, I think this is a case where a hierarchical representation of mood states could be very useful.

Problem F [5 points]: Generating Emission Sequences

Solution F.:

```
File #0:
Generated Emission
######################################################################
27615152416007476475
57462054470367515665
15245431355575665767
77642550452152772114
04516111725410443215


File #1:
Generated Emission
######################################################################
75453017254323450075
66545662546061535530
72402761521502340504
52654270651454544575
74746677706457125074


File #2:
Generated Emission
######################################################################
40042249921877379225
96702732355217724551
25703569114694636792
19361697903277594303
82512630802727306125


File #3:
Generated Emission
######################################################################
64261051465133016563
13105115626116514501
61660361221426222114
54651361001213031636
60643224001061313113


File #4:
Generated Emission
######################################################################
42361606421320451314
56033636134126012435
12006346056536651066
06166130032636166120
02263642353245253322


File #5:
Generated Emission
######################################################################
21684881314428822731
34264463800366322434
64454060803411403266
83418288114041386150
02248425414633118640
```

Figure 7: Generated Sequences

Problem G [3 points]: Sparsity of Trained A and O Matrices

Solution G.: *The transition and observation matrices are pretty sparse. We can see mostly dark areas, with a handful of bright pixels, perhaps more visibly in the transition matrix. It's hard to directly compare the two as they are different sized matrices. The sparsity of the transition matrix could imply that there are specific directionalities or neighborhoods of states. This makes sense, as we these are trained on the constitution. The Constitution is made of words and words have very important syntactic relationships; random classes of words are not often put together. If we view these states as an unsupervised part of speech, then this makes sense. Sparsity in the observation matrix also makes sense. If the current state is something resembling a noun, then a word that is considered in this group would occur with a much higher probability than something we might consider a verb. However, rows of this observation matrix will likely have more nonzero entries, as the founders had an impressive vocabulary, and in language there are many words that one could choose for a given spot in a sentence.*

Problem H [5 points]: Hidden States vs. Sample Emission Sentences from HMM

Solution H.: *The sample emission sentences improve as the number of hidden states is increased. For example, the 1 state model produced a fragment including "all be coin or between one of and." This is mostly incomprehensible, and there are words that don't really make sense together ('of', 'and'). The higher state examples are still not really meaningful, and would never fool a native speaker, but they appear to have a more understandable syntactical structure.*

In the special case of one state, it seems that the state information is completely lost. Instead, it resorts to relying completely on the Observation Matrix (which has only 1 row). It seems that words are sampled proportional to their frequency in the corpus, and that's it, like some sort of bag of words.

In general, when the number of hidden states is unknown, allowing more hidden states will increase the training data likelihood. However, if the number of states becomes greater than or equal to the number of tokens, we end up with uninformative states that just memorize the words. So, like many things, there is a tradeoff; more states means a more parameterized model that might inappropriately separate every word into a unique state. I imagine some sort of hierarchical hidden markov model might be useful both for more accurate predictions, but also to use the learning as a way to classify unknown states of some data-generating process at differing granularities.

Problem I [5 points]: Analyzing Visualization of State

Solution I.: *First, I wanted to mention that the word 'state' shows up as a high probability word in 5 of the 9 states. This is likely because it is one of the most commonly used words, and perhaps 9 states becomes more finely granulated such that 'state' in different contexts falls into different states.*

Figure 8: State 1 word cloud

This state appears to represent words pertaining to the structure of the United States government. We have words like 'law', 'representative', 'court', 'justice', 'ambassadors', 'year', 'war', 'foreign'. The words in this state appear to be primarily (but not all) nouns related to how a government should work, including different branches of the government ('court'), foreign relations ('ambassadors', 'war'), and individuals with power in the government ('senator', 'representative').

Problem 2

In this Jupyter notebook, we visualize how HMMs work. This visualization corresponds to problem 2 in set 6.

Assuming your HMM module is complete and saved at the correct location, you can simply run all cells in the notebook without modification.

In [1]:
```
import os
import numpy as np
from IPython.display import HTML

from HMM import unsupervised_HMM
from HMM_helper import (
    text_to_wordcloud,
    states_to_wordclouds,
    parse_observations,
    sample_sentence,
    visualize_sparsities,
    animate_emission
)
```

Visualization of the dataset

We will be using the Constitution as our dataset. First, we visualize the entirety of the Constitution as a wordcloud:

In [2]:
```
text = open(os.path.join(os.getcwd(), 'data/constitution.txt')).read()
wordcloud = text_to_wordcloud(text, title='Constitution')
```

Training an HMM

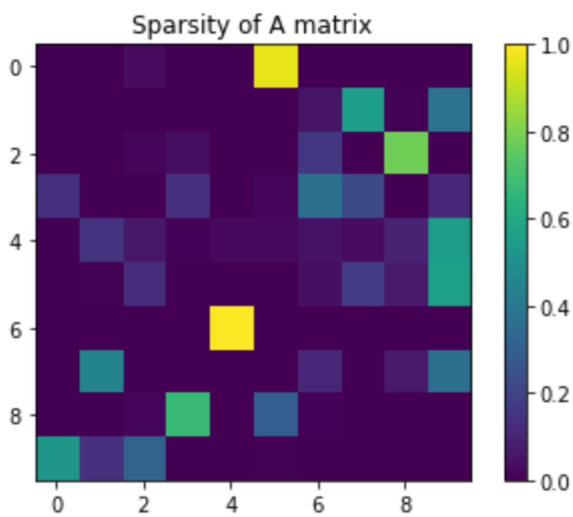
Now we train an HMM on our dataset. We use 10 hidden states and train over 100 iterations:

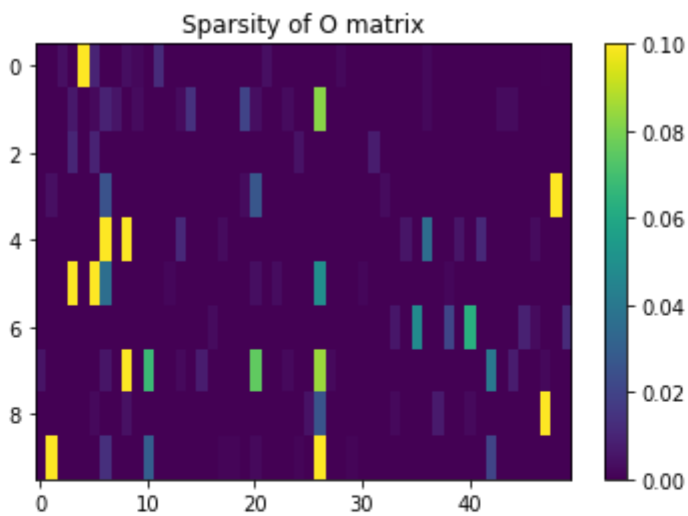
In [3]:
```
obs, obs_map = parse_observations(text)
hmm8 = unsupervised_HMM(obs, 10, 100)
```

Part G: Visualization of the sparsities of A and O

We can visualize the sparsities of the A and O matrices by treating the matrix entries as intensity values and showing them as images. What patterns do you notice?

In [4]:
```
visualize_sparsities(hmm8, O_max_cols=50)
```





Generating a sample sentence

As you have already seen, an HMM can be used to generate sample sequences based on the given dataset. Run the cell below to show a sample sentence based on the Constitution.

In [5]:
```
print('Sample Sentence:\n====================')
print(sample_sentence(hmm8, obs_map, n_words=25))
```

```
Sample Sentence:
====================
Them to quorum captures or our shall not any which accept states the ma
ke protect the congress section by representative on the own states con
gress...
```

Part H: Using varying numbers of hidden states

Using different numbers of hidden states can lead to different behaviours in the HMMs. Below, we train several HMMs with 1, 2, 4, and 16 hidden states, respectively. What do you notice about their emissions? How do these emissions compare to the emission above?

In [6]:
```
hmm1 = unsupervised_HMM(obs, 1, 100)
print('\nSample Sentence:\n====================')
print(sample_sentence(hmm1, obs_map, n_words=25))
```

```
Sample Sentence:
====================
Organizing all be coin or between one of and forfeiture state seven hou
se be the states likewise in several blood in erected the of in...
```

In [7]:
```
hmm2 = unsupervised_HMM(obs, 2, 100)
print('\nSample Sentence:\n====================')
print(sample_sentence(hmm2, obs_map, n_words=25))
```

```
Sample Sentence:
====================
And to be general the officers be and courts shall constitute any excep
t suffrage six notwithstanding of to the manner prejudice one nominate
the and...
```

In [8]:
```
hmm4 = unsupervised_HMM(obs, 4, 100)
print('\nSample Sentence:\n====================')
print(sample_sentence(hmm4, obs_map, n_words=25))
```

```
Sample Sentence:
====================
Legislature more on his grant regulate crimes excises coin office and b
y law on which manner as the useful during post diminished by meet t
o...
```

In [9]:
```
hmm16 = unsupervised_HMM(obs, 16, 100)
print('\nSample Sentence:\n====================')
print(sample_sentence(hmm16, obs_map, n_words=25))
```

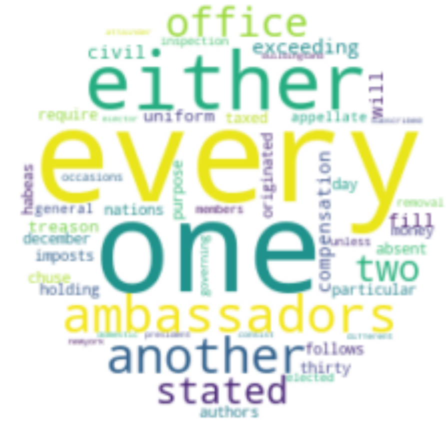
```
Sample Sentence:
====================
Disapproved establish the direct days or the make discoveries a require
either of the section or to the excepting or of treason the first hous
e...
```

Part I: Visualizing the wordcloud of each state

Below, we visualize each state as a wordcloud by sampling a large emission from the state:

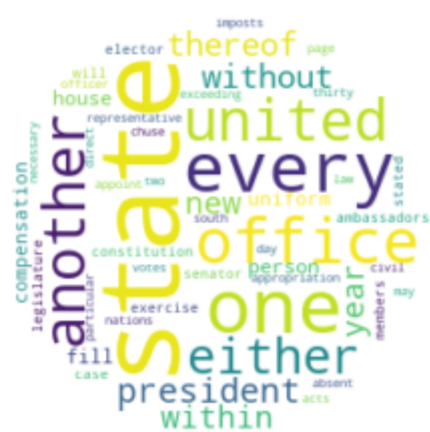
In [10]:
```
wordclouds = states_to_wordclouds(hmm8, obs_map)
```

State 0

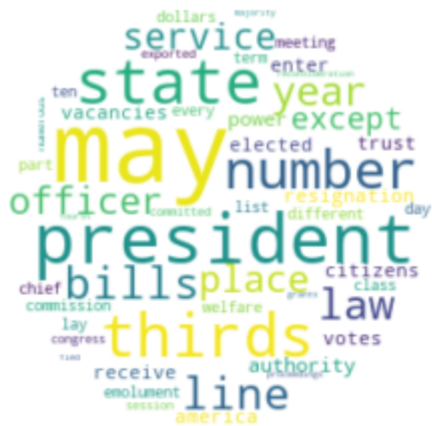


State 1

State 2

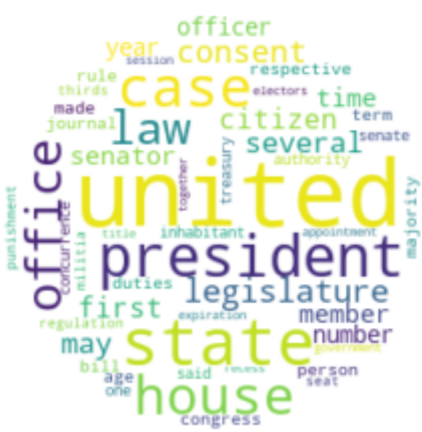


State 3

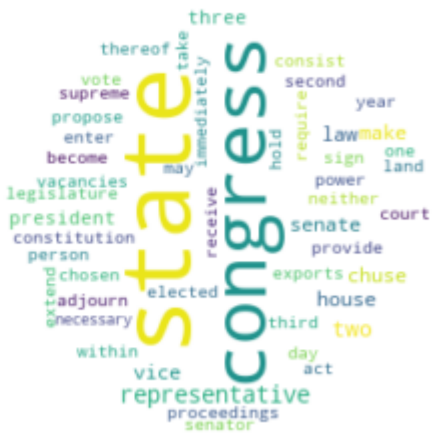


State 4

State 5



State 6



State 7

State 8

Visualizing the process of an HMM generating an emission

The visualization below shows how an HMM generates an emission. Each state is shown as a wordcloud on the plot, and transition probabilities between the states are shown as arrows. The darker an arrow, the higher the transition probability.

At every frame, a transition is taken and an observation is emitted from the new state. A red arrow indicates that the transition was just taken. If a transition stays at the same state, it is represented as an arrowhead on top of that state.

Use fullscreen for a better view of the process.

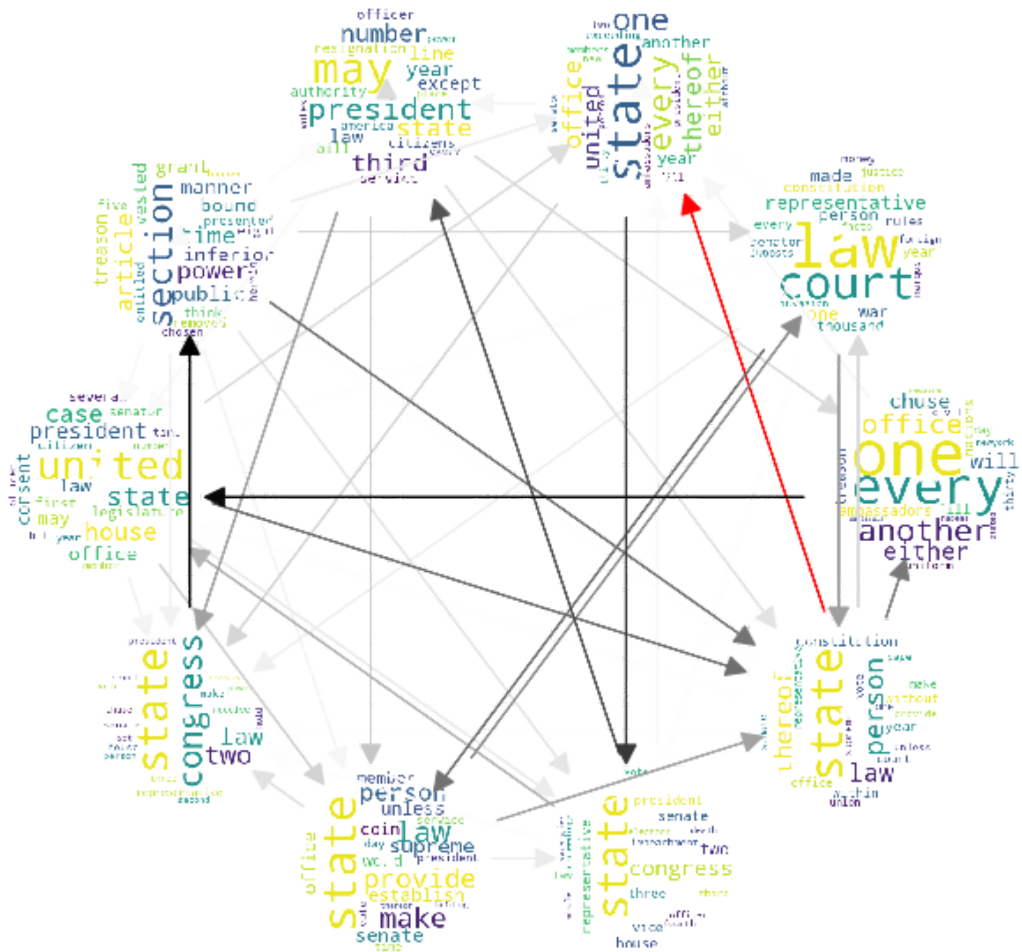
In [11]:
```
anim = animate_emission(hmm8, obs_map, M=8)
HTML(anim.to_html5_video())
```

```
Animating...
```

Out[11]:

0:00 / 0:09

Be not bound as three and of the



```
In [ ]:
```

```
In [ ]:
```

```
########################################
# CS/CNS/EE 155 2018
# Problem Set 6
#
# Author:       Andrew Kang
# Description:  Set 6 skeleton code
########################################

# You can use this (optional) skeleton code to complete the HMM
# implementation of set 5. Once each part is implemented, you can simply
# execute the related problem scripts (e.g. run 'python 2G.py') to quickly
# see the results from your code.
#
# Some pointers to get you started:
#
#     - Choose your notation carefully and consistently! Readable
#       notation will make all the difference in the time it takes you
#       to implement this class, as well as how difficult it is to debug.
#
#     - Read the documentation in this file! Make sure you know what
#       is expected from each function and what each variable is.
#
#     - Any reference to "the (i, j)^th" element of a matrix T means that
#       you should use T[i][j].
#
#     - Note that in our solution code, no NumPy was used. That is, there
#       are no fancy tricks here, just basic coding. If you understand HMMs
#       to a thorough extent, the rest of this implementation should come
#       naturally. However, if you'd like to use NumPy, feel free to.
#
#     - Take one step at a time! Move onto the next algorithm to implement
#       only if you're absolutely sure that all previous algorithms are
#       correct. We are providing you waypoints for this reason.
#
# To get started, just fill in code where indicated. Best of luck!

import random
import numpy as np

class HiddenMarkovModel:
    '''
    Class implementation of Hidden Markov Models.
    '''

    def __init__(self, A, O):
        '''
        Initializes an HMM. Assumes the following:
            - States and observations are integers starting from 0.
            - There is a start state (see notes on A_start below). There
              is no integer associated with the start state, only
              probabilities in the vector A_start.
```

```
        - There is no end state.

    Arguments:
        A:          Transition matrix with dimensions L x L.
                    The (i, j)^th element is the probability of
                    transitioning from state i to state j. Note that
                    this does not include the starting probabilities.

        O:          Observation matrix with dimensions L x D.
                    The (i, j)^th element is the probability of
                    emitting observation j given state i.

    Parameters:
        L:          Number of states.

        D:          Number of observations.

        A:          The transition matrix.

        O:          The observation matrix.

        A_start:    Starting transition probabilities. The i^th element
                    is the probability of transitioning from the start
                    state to state i. For simplicity, we assume that
                    this distribution is uniform.
    '''

    self.L = len(A)
    self.D = len(O[0])
    self.A = A
    self.O = O
    self.A_start = [1. / self.L for _ in range(self.L)]


def viterbi(self, x):
    '''
    Uses the Viterbi algorithm to find the max probability state
    sequence corresponding to a given input sequence.

    Arguments:
        x:          Input sequence in the form of a list of length M,
                    consisting of integers ranging from 0 to D - 1.

    Returns:
        max_seq:    State sequence corresponding to x with the highest
                    probability.
    '''

    M = len(x)      # Length of sequence.
    A = self.A
    O = self.O
```

```
        L = self.L
        A_start = self.A_start

        # using list comprehensions for more concise code
        # The (i, j)^th elements of probs and seqs are the max probability
        # of the prefix of length i ending in state j and the prefix
        # that gives this probability, respectively.
        #
        # For instance, probs[1][0] is the probability of the prefix of
        # length 1 ending in state 0.
        probs = [[0. for _ in range(self.L)] for _ in range(M + 1)]
        seqs = [['' for _ in range(self.L)] for _ in range(M + 1)]

        # initialization case
        for state in range(L):
            # log probability of transition into starting state + generating
             x[0]
            # probs[0][state] = (np.log(self.A_start[state]) +
             np.log(self.O[state][x[0]])).tolist()
            probs[0][state] = A_start[state] * O[state][x[0]]

            seqs[0][state] = str(state)

        # Viterbi for all rest of states
        # loop over all observations
        for prev_x in range(M-1):
            # loop over all states
            for state in range(L):
                k = prev_x
                y = state
                # (cur_prob, append_state) = max(((np.log(probs[k][y_prev]) +
                 np.log(A[y_prev][y]) + np.log(O[y][x[k+1]])).tolist(),
                 y_prev) for y_prev in range(L))
                (cur_prob, append_state) = max((probs[k][y_prev] *
                 A[y_prev][y] * O[y][x[k+1]],y_prev) for y_prev in range(L))


                probs[k+1][y] = cur_prob
                seqs[k+1][y] = seqs[k][append_state] + str(y)

        k += 1
        (prob, state) = max((probs[M-1][state], y) for y in range(L))
        max_seq = seqs[M-1][state]
        return max_seq


    def forward(self, x, normalize=False):
        '''
        Uses the forward algorithm to calculate the alpha probability
        vectors corresponding to a given input sequence.
```

```
    Arguments:
        x:          Input sequence in the form of a list of length M,
                    consisting of integers ranging from 0 to D - 1.

        normalize:  Whether to normalize each set of alpha_j(i) vectors
                    at each i. This is useful to avoid underflow in
                    unsupervised learning.

    Returns:
        alphas:     Vector of alphas.

                    The (i, j)^th element of alphas is alpha_j(i),
                    i.e. the probability of observing prefix x^1:i
                    and state y^i = j.

                    e.g. alphas[1][0] corresponds to the probability
                    of observing x^1:1, i.e. the first observation,
                    given that y^1 = 0, i.e. the first state is 0.
    '''
    M = len(x)      # Length of sequence.
    # Hmm, seems like have to use M+1 and not M
    alphas = [[1. for _ in range(self.L)] for _ in range(M+1)]


    # initialization here
    for state in range(self.L):
        alphas[1][state] = self.A_start[state] * self.O[state][x[0]]
        # if normalize:
        #     C = np.sum(alphas[1])
        #     alphas[1] = np.multiply((1/C), alphas[1]).tolist()

    if normalize:
        C = np.sum(alphas[1])
        alphas[1] = np.multiply((1/C), alphas[1]).tolist()

    # have normalization option for underflow issues.  keep running algo


    for k in range(1, M):
            # for state
            for y in range(self.L):
                # similar to Viterbi, but summing instead of max.
                 actually identical, obviously don't return the seq.
                alpha = sum(alphas[k][y_prev] * self.A[y_prev][y] *
                 self.O[y][x[k]] for y_prev in range(self.L))
                alphas[k+1][y] = alpha
            if normalize:
                C = np.sum(alphas[k+1])
                alphas[k+1] = np.multiply((1/C), alphas[k+1]).tolist()

    return alphas
```

```
def backward(self, x, normalize=False):
    '''
    Uses the backward algorithm to calculate the beta probability
    vectors corresponding to a given input sequence.

    Arguments:
        x:          Input sequence in the form of a list of length M,
                    consisting of integers ranging from 0 to D - 1.

        normalize:  Whether to normalize each set of alpha_j(i) vectors
                    at each i. This is useful to avoid underflow in
                    unsupervised learning.

    Returns:
        betas:      Vector of betas.

                    The (i, j)^th element of betas is beta_j(i), i.e.
                    the probability of observing prefix x^(i+1):M and
                    state y^i = j.

                    e.g. betas[M][0] corresponds to the probability
                    of observing x^M+1:M, i.e. no observations,
                    given that y^M = 0, i.e. the last state is 0.
    '''

    M = len(x)      # Length of sequence.
    betas = [[0. for _ in range(self.L)] for _ in range(M + 1)]

    # initialization here
    for state in range(self.L):
        betas[M][state] = 1.0
        # if normalize:
        #     C = np.sum(betas[M])
        #     betas[1] = np.multiply((1/C), betas[M]).tolist()

    if normalize:
        C = np.sum(betas[M])
        betas[1] = np.multiply((1/C), betas[M]).tolist()

    # have normalization option for underflow issues.  keep running algo

    # This time we are running backwards, so start at M, go to 0
    for k in range(M, 0, -1):
            # for state
            for y in range(self.L):
                # similar to Viterbi, but summing instead of max.
                 actually identical, obviously don't return the seq.
                beta = sum(betas[k][y_next] * self.A[y][y_next] *
                 self.O[y_next][x[k-1]] for y_next in range(self.L))
```

```
                betas[k-1][y] = beta
            if normalize:
                C = np.sum(betas[k-1])
                betas[k-1] = np.multiply((1/C), betas[k-1]).tolist()


        return betas


    def supervised_learning(self, X, Y):
        '''
        Trains the HMM using the Maximum Likelihood closed form solutions
        for the transition and observation matrices on a labeled
        datset (X, Y). Note that this method does not return anything, but
        instead updates the attributes of the HMM object.

        Arguments:
            X:          A dataset consisting of input sequences in the form
                        of lists of variable length, consisting of integers
                        ranging from 0 to D - 1. In other words, a list of
                        lists.

            Y:          A dataset consisting of state sequences in the form
                        of lists of variable length, consisting of integers
                        ranging from 0 to L - 1. In other words, a list of
                        lists.

                        Note that the elements in X line up with those in Y.
        '''

        # Calculate each element of A using the M-step formulas.
        # Determine number of sequences
        N = len(X)
        # A: transition matrix
        for a in range(self.L):
            for b in range(self.L):
                a_numer = 0
                a_denom = 0

                for j in range(N):
                    M = len(X[j])
                    for i in range(M-1):
                        if (Y[j][i] == a) & (Y[j][i+1] == b):
                            a_numer += 1
                        if (Y[j][i] == a):
                            a_denom += 1
                self.A[a][b] = a_numer / a_denom



        # Calculate each element of O using the M-step formulas.
```

```
    for z in range(self.L):
        for w in range(self.D):
            o_numer = 0
            o_denom = 0

            for j in range(N):
                M = len(X[j])
                for i in range(M):
                    if (Y[j][i] == z) & (X[j][i] == w):
                        o_numer += 1
                    if (Y[j][i] == z):
                        o_denom += 1
            self.O[z][w] = o_numer / o_denom

    pass


def unsupervised_learning(self, X, N_iters):
    '''
    Trains the HMM using the Baum-Welch algorithm on an unlabeled
    datset X. Note that this method does not return anything, but
    instead updates the attributes of the HMM object.

    Arguments:
        X:          A dataset consisting of input sequences in the form
                    of lists of length M, consisting of integers ranging
                    from 0 to D - 1. In other words, a list of lists.

        N_iters:    The number of iterations to train on.
    '''

    for each_iteration in range(N_iters):
        # Primarily sticking to list comprehensions for more readable code
        # initialize the 0 matrices for A numerator and denominator and O
         num/denom
        a_num = [[0. for _ in range(self.L)] for _ in range(self.L)]
        a_den = [[0. for _ in range(self.L)] for _ in range(self.L)]
        o_num = [[0. for _ in range(self.D)] for _ in range(self.L)]
        o_den = [[0. for _ in range(self.D)] for _ in range(self.L)]

        for j in range(len(X)):

            M = len(X[j])


            # initialize the probabilities.
            probs = [[0. for _ in range(self.L)] for _ in range(M)]
            joint_probs = [[[0. for _ in range(self.L)] for _ in
             range(self.L)] for _ in range(M)]
```

```
        # time for E/M
        # E:
        # Calculate alphas and betas using forward and backward methods
        alphas = self.forward(X[j], normalize=True)
        betas = self.backward(X[j], normalize=True)

        # probabilities now
        for i in range(M):
            # marginal
            probs[i] = [alphas[i+1][z] * betas[i+1][z] /
             sum([alphas[i+1][w] * betas[i+1][w] for w in
             range(self.L)]) for z in range(self.L)]

        for i in range(M-1):
            joint_margin_denom = sum([sum([alphas[i+1][a] *
             betas[i+1+1][b] * self.A[a][b] * self.O[b][X[j][i+1]]
                for a in range(self.L)]) for b in range(self.L)])

            for a in range(self.L):
                for b in range(self.L):
                    cur_joint_prob = alphas[i+1][a] * betas[i+1+1][b]
                     * self.A[a][b] * self.O[b][X[j][i+1]]
                    joint_probs[i][a][b] = cur_joint_prob /
                     joint_margin_denom

        # so many current iterator variables in use, be very careful.
         removing all cases of ii, jj, i_, etc and replacing with
        # single letters
        # M:
        # update A
        for a in range(self.L):
            for b in range(self.L):
                a_num[a][b] += sum([joint_probs[i][a][b] for i in
                 range(M-1)])
                a_den[a][b] += sum([probs[i][a] for i in range(M-1)])
        # update O
        # over states
        for z in range(self.L):
            # over tokens
            for w in range(self.D):
                o_den[z][w] += sum([probs[i][z] for i in range(M)])
                for i in range(M):
                    if X[j][i] ==w:
                        o_num[z][w] += probs[i][z]

    self.A = [[a_num[i][j] /a_den[i][j] for j in range(self.L)] for i
     in range(self.L)]
    self.O = [[o_num[i][j] / o_den[i][j] for j in range(self.D)] for i
     in range(self.L)]
```

```
def generate_emission(self, M):
    '''
    Generates an emission of length M, assuming that the starting state
    is chosen uniformly at random.

    Arguments:
        M:          Length of the emission to generate.

    Returns:
        emission:   The randomly generated emission as a list.

        states:     The randomly generated states as a list.
    '''

    emission = []
    states = []

    # start state randomly
    states.append(np.random.choice(self.L))
    emission.append(np.random.choice(self.D, p=self.O[states[0]]))
    # now loop over
    for i in range(M-1):
        states.append(np.random.choice(self.L, p=self.A[states[i]]))
        emission.append(np.random.choice(self.D, p=self.O[states[i]]))


    return emission, states


def probability_alphas(self, x):
    '''
    Finds the maximum probability of a given input sequence using
    the forward algorithm.

    Arguments:
        x:          Input sequence in the form of a list of length M,
                    consisting of integers ranging from 0 to D - 1.

    Returns:
        prob:       Total probability that x can occur.
    '''

    # Calculate alpha vectors.
    alphas = self.forward(x)

    # alpha_j(M) gives the probability that the state sequence ends
    # in j. Summing this value over all possible states j gives the
```

```
        # total probability of x paired with any state sequence, i.e.
        # the probability of x.
        prob = sum(alphas[-1])
        return prob


    def probability_betas(self, x):
        '''
        Finds the maximum probability of a given input sequence using
        the backward algorithm.

        Arguments:
            x:          Input sequence in the form of a list of length M,
                        consisting of integers ranging from 0 to D - 1.

        Returns:
            prob:       Total probability that x can occur.
        '''

        betas = self.backward(x)

        # beta_j(1) gives the probability that the state sequence starts
        # with j. Summing this, multiplied by the starting transition
        # probability and the observation probability, over all states
        # gives the total probability of x paired with any state
        # sequence, i.e. the probability of x.
        prob = sum([betas[1][j] * self.A_start[j] * self.O[j][x[0]] \
                    for j in range(self.L)])

        return prob


def supervised_HMM(X, Y):
    '''
    Helper function to train a supervised HMM. The function determines the
    number of unique states and observations in the given data, initializes
    the transition and observation matrices, creates the HMM, and then runs
    the training function for supervised learning.

    Arguments:
        X:          A dataset consisting of input sequences in the form
                    of lists of variable length, consisting of integers
                    ranging from 0 to D - 1. In other words, a list of lists.

        Y:          A dataset consisting of state sequences in the form
                    of lists of variable length, consisting of integers
                    ranging from 0 to L - 1. In other words, a list of lists.
                    Note that the elements in X line up with those in Y.
    '''
    # Make a set of observations.
    observations = set()
```

```python
    for x in X:
        observations |= set(x)

    # Make a set of states.
    states = set()
    for y in Y:
        states |= set(y)

    # Compute L and D.
    L = len(states)
    D = len(observations)

    # Randomly initialize and normalize matrix A.
    A = [[random.random() for i in range(L)] for j in range(L)]

    for i in range(len(A)):
        norm = sum(A[i])
        for j in range(len(A[i])):
            A[i][j] /= norm

    # Randomly initialize and normalize matrix O.
    O = [[random.random() for i in range(D)] for j in range(L)]

    for i in range(len(O)):
        norm = sum(O[i])
        for j in range(len(O[i])):
            O[i][j] /= norm

    # Train an HMM with labeled data.
    HMM = HiddenMarkovModel(A, O)
    HMM.supervised_learning(X, Y)

    return HMM

def unsupervised_HMM(X, n_states, N_iters):
    '''
    Helper function to train an unsupervised HMM. The function determines the
    number of unique observations in the given data, initializes
    the transition and observation matrices, creates the HMM, and then runs
    the training function for unsupervised learing.

    Arguments:
        X:          A dataset consisting of input sequences in the form
                    of lists of variable length, consisting of integers
                    ranging from 0 to D - 1. In other words, a list of lists.

        n_states:   Number of hidden states to use in training.

        N_iters:    The number of iterations to train on.
    '''
```

```
# Make a set of observations.
observations = set()
for x in X:
    observations |= set(x)

# Compute L and D.
L = n_states
D = len(observations)
random.seed(2020)
# Randomly initialize and normalize matrix A.
A = [[random.random() for i in range(L)] for j in range(L)]

for i in range(len(A)):
    norm = sum(A[i])
    for j in range(len(A[i])):
        A[i][j] /= norm
random.seed(155)
# Randomly initialize and normalize matrix O.
O = [[random.random() for i in range(D)] for j in range(L)]

for i in range(len(O)):
    norm = sum(O[i])
    for j in range(len(O[i])):
        O[i][j] /= norm

# Train an HMM with unlabeled data.
HMM = HiddenMarkovModel(A, O)
HMM.unsupervised_learning(X, N_iters)

return HMM
```