Machine Learning & Data Mining
Caltech CS/CNS/EE 155
Homework 6

Changhao Xu
UID: 2103530
March 5th, 2020

1 Class-Conditional Densities for Binary Data

Question A:

$$p(x|y=c) = p(x_1|y=c)p(x_2|x_1,y=c)\cdots p(x_D|x_1,\cdots,x_{D-1},y=c) = \prod_{j=1}^{D}\theta_{xjc}$$

Since all the D features are binary, $x_j \in \{0,1\}$,
for each class of C we need 2^{j-1} parameters for $p(x_j|x_1,\cdots,x_{j-1},y=c)$. In total $\sum_{j=1}^{D} 2^{j-1}$ = $O(2^D)$.
Therefore, we need $O(C \times 2^D)$ parameters for C classes.

Question B: Without factorization, since all the D features are binary, $x_j \in \{0,1\}$, and C classes for y, in total we need $O(C \times 2^D)$, which is the same as that with factorization.

Question C: For a small N, Naive Bayes is likely to give lower test set error. This is because full models are more likely to overfit with a small N.

Question D: For a large N, full models are likely to give lower test set error. This is because Naive Bayes is simple and is likely to underfit with a large N, while full models have more parameters and will do better.

Question E: For Naive Bayes,

$$p(y|x) = \frac{p(x,y)}{p(x)} = \frac{p(y)}{p(x)}\prod_{d} p(x^d|y) = \frac{p(y)}{\sum_{i=1}^{C} p(x|y=c_i)}\prod_{d} p(x^d|y)$$

Since we assumed a uniform class prior p(y), O(p(y)) = O(1), and computation complexity of $p(y|x) = O(CD)$.
For full model,

$$p(y|x) = \frac{p(x,y)}{p(x)} = \frac{p(y)}{p(x)}p(x|y)$$

, and computation complexity of D-dimensional vector is O(D). Different from Naive Bayes, full model doesn't take into account every y = c, but their overall probability. Therefore, computation complexity of $p(y|x) = O(D)$.

2 Sequence Prediction

Question A:

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


File #0:
Emission Sequence              Max Probability State Sequence
######################################################################
25421                          31033
01232367534                    22222100310
5452674261527433               1031003103222222
7226213164512267255            1310331000033100310
0247120602352051010255241      2222222222222222222222103


File #1:
Emission Sequence              Max Probability State Sequence
######################################################################
77550                          22222
7224523677                     2222221000
505767442426747                222100003310031
7213413164553611226 7          10310310000310333100
4733667771450051060253041      2221000003222223103222223


File #2:
Emission Sequence              Max Probability State Sequence
######################################################################
60622                          11111
4687981156                     2100202111
815833657775062                021011111111111
21310222515963505015           02020111111111111021
6503199452571274006320025      1110202111111102021110211


File #3:
Emission Sequence              Max Probability State Sequence
######################################################################
13661                          00021
2102213421                     3131310213
166066262165133                133333133133100
53164662112162634156           20000021313131002133
1523541005123230226306256      1310021333133133133133133
```

```
File #4:
Emission Sequence              Max Probability State Sequence
######################################################################
23664                          01124
3630535602                     0111201112
350201162150142                011244012441112
00214005402015146362           11201112412444011112
2111266524665143562534450      2012012424124011112411124


File #5:
Emission Sequence              Max Probability State Sequence
######################################################################
68535                          10111
4546566636                     1111111111
638436858181213                110111010000011
13240338308444514688           00010000000111111100
0111664434441382533632626      2111111111111100111110101
```

Question B: The results using Forward algorithm are as follows:

```
######################################################################
                    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
```

The results using Backward algorithm are as follows:

```
######################################################################
                    Running Code For Question 2Bii
######################################################################


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
```

Question 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
```

Question D:

Changhao Xu
UID: 2103530
March 5th, 2020

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


test case: N_iters=0
Transition Matrix:
########################################################################
2.470e-01   6.958e-02   3.064e-01   3.770e-01
1.846e-01   3.635e-01   2.114e-01   2.404e-01
2.902e-01   1.522e-01   1.328e-01   4.248e-01
3.266e-01   2.547e-01   5.671e-02   3.620e-01


Observation Matrix:
########################################################################
9.093e-02   8.443e-02   1.399e-01   1.539e-01   1.214e-01   6.519e-02   8.377e-02   1.268e-01   9.067e-02   4.306e-02
4.669e-02   2.347e-01   1.511e-02   5.726e-03   1.281e-01   4.333e-02   5.775e-02   1.542e-01   1.871e-01   1.273e-01
1.797e-01   1.101e-01   6.323e-02   1.346e-01   3.886e-02   9.526e-02   1.189e-01   2.195e-02   8.021e-02   1.572e-01
1.381e-01   2.190e-01   6.830e-02   1.514e-01   1.058e-01   1.222e-02   1.477e-01   1.160e-01   7.535e-03   3.392e-02


test case: N_iters=1
Transition Matrix:
########################################################################
2.702e-01   7.208e-02   3.259e-01   3.318e-01
2.003e-01   3.712e-01   2.216e-01   2.069e-01
3.229e-01   1.585e-01   1.439e-01   3.747e-01
3.561e-01   2.646e-01   6.016e-02   3.192e-01


Observation Matrix:
########################################################################
9.941e-02   3.250e-02   1.594e-01   1.280e-01   1.222e-01   1.215e-01   5.508e-02   1.076e-01   1.409e-01   3.342e-02
5.500e-02   9.644e-02   1.872e-02   4.955e-03   1.382e-01   8.785e-02   4.052e-02   1.451e-01   3.061e-01   1.072e-01
1.987e-01   4.211e-02   7.324e-02   1.137e-01   4.002e-02   1.815e-01   8.075e-02   1.931e-02   1.253e-01   1.253e-01
1.880e-01   1.038e-01   9.799e-02   1.553e-01   1.338e-01   2.815e-02   1.223e-01   1.229e-01   1.453e-02   3.325e-02
```

```
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
```

Changhao Xu
UID: 2103530

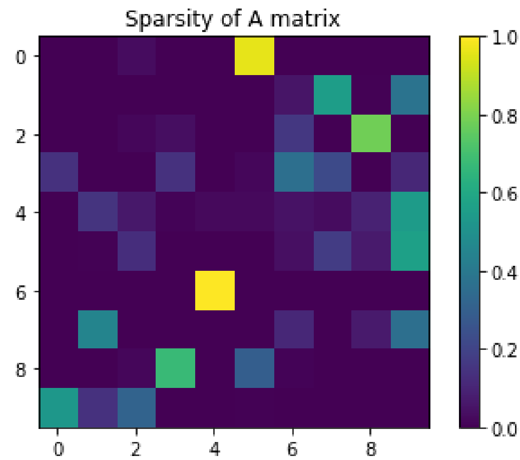
Question E: The transition and emission matrices from 2C and 2D are different, supervised matrices are more uniform, while unsupervised are sparse and nonuniform. 2C (supervised HMM) provides a more accurate representation. This is because for unsupervised 2D, we simply random the initial matrices, which cannot truly reflect Ron's moods. To improve the unsupervised learning data, we might need to initialize the transition and emission matrices A & O more precisely, such as using probability distributions from supervised matrices.

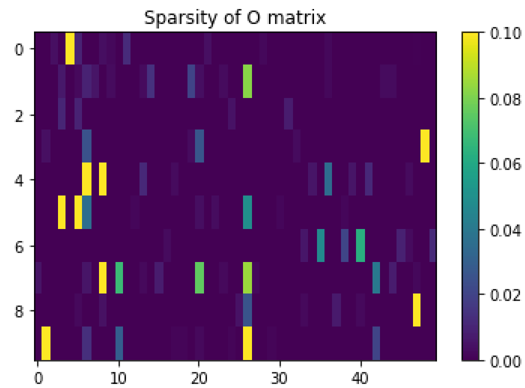
Question F:

```
######################################################################
                     Running Code For Question 2F
######################################################################


File #0:
Generated Emission
######################################################################
45274511250275204412
05407242224726770147
72422713473031327660
42427312074107504324
65147177214245322422


File #1:
Generated Emission
######################################################################
40035770104447412207
45516552177027677212
04317422222000147762
35746073172105617607
05107555006525007447


File #2:
Generated Emission
######################################################################
17726563315712157156
90030125025149435211
62975620039788065245
45692563828145922702
52899330755675696052


File #3:
Generated Emission
######################################################################
03341033031362462112
15126633304022436420
62165152146652334362
12252065242314333013
21214502130111364421
```

```
File #4:
Generated Emission
######################################################################
34202331563531320223
16566131513143621520
43335062640460660662
62431603632366426636
16354030615436116252


File #5:
Generated Emission
######################################################################
32511566801635284208
81164464364484665615
04666240667481480436
60366565318454544314
45422451038680813715
```

Question G:
The trained A and O matrices are both sparse, with most elements near 0. The sparsity of transition matrix A means that for each state, there might be very few states to transit to, and the sparsity of observation matrix O means that for each state, there might be very few observations to belong to.





Question H: As the number of hidden states is increased, sample emission sentences from the HMM becomes more coherent and smooth. When there is only one hidden state, then there is no more transitions, meaning that observation words are just randomly picked from the dataset. Allowing more hidden states will increase the training data likelihood, but when we have too many hidden states for the fixed observation set, this will lead to overfitting and may increase the test error.

Question I: As shown in the figure, I think state 7 is semantically meaningful. This state is filled with

law-related words, such as 'law', 'court', 'representative', 'justice', 'rules'. I think this state represent all law-related words in the dataset, which is distinct from other states.

HMM.py

March 5, 2020

[ ]:
```
########################################
# 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
```

```python
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.  word of tags: N, V, adj

            D:          Number of observations.   fish, sleep

            A:          The transition matrix. L x L

            O:          The observation matrix. L x D

            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)]
```

```python
    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.

        # 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)] #   probability
        seqs = [['' for _ in range(self.L)] for _ in range(M + 1)] # viterbi

        # L  # of states/tags, D: # of observations/  , M: sequence , A: L x L, O: L x D

        for i in range(self.L): # initialize 1st state,    probs[1][i] A_start * O,  probs[0] [0,...,0]
            probs[1][i] = self.A_start[i] * self.O[i][x[0]]

        for seq in range(1, M): #   sequence
            for curr in range(self.L): #   states/tags
                max_value = 0
                max_idx = 0
                for last in range(self.L): #   state
                    if (probs[seq][last] * self.A[last][curr] * self.O[curr][x[seq]] >= max_value):
                        max_value = probs[seq][last] * self.A[last][curr] * self.O[curr][x[seq]] #
                        max_idx = last
                probs[seq + 1][curr] = max_value #       state probs
                seqs[seq + 1][curr] = max_idx
```

```
        max_seq_rev = []

        max_value = max(probs[M])
        max_idx = probs[M].index(max_value) #
↪    state max_value( max_prob)  max_idx

        max_seq_rev.append( str(max_idx) )

        for i in range(M, 1, -1):
            max_seq_rev.append( str(seqs[i][max_idx]) )
            max_idx = seqs[i][max_idx]

        max_seq = max_seq_rev[::-1]

        return "".join(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.
        alphas = [[0. for _ in range(self.L)] for _ in range(M + 1)]

        for i in range(self.L):
```

```python
            alphas[1][i] = self.A_start[i] * self.O[i][x[0]]

        for seq in range(1, M): #   sequence
            for curr in range(self.L): #   states/tags
                sum_value = 0
                for last in range(self.L): #   state
                    sum_value += alphas[seq][last] * self.A[last][curr] * self.O[curr][x[seq]] # Viterbi replaces sum with max

                alphas[seq + 1][curr] = sum_value

            if normalize:
                sum_alpha = sum(alphas[seq + 1])
                for curr in range(self.L):
                    alphas[seq + 1][curr] /= sum_alpha

        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 beta_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)]
```

```
        for i in range(self.L):
            betas[-1][i] = 1 # PPT 74, beta(M) = 1

        for seq in range(-1, -M-1, -1): #   sequence
            for curr in range(self.L): #   states/tags
                sum_value = 0
                for nxt in range(self.L): #   state
                    if seq != -M:
                        sum_value += betas[seq][nxt] * self.A[curr][nxt] * self.
↪O[nxt][x[seq]] # PPT 74,   A_{z,j} A_{j,z}
                    else:
                        sum_value += betas[seq][nxt] * self.A_start[nxt] * self.
↪O[nxt][x[seq]]

                betas[seq - 1][curr] = sum_value

            if normalize:
                sum_beta = sum(betas[seq - 1])
                for curr in range(self.L):
                    betas[seq - 1][curr] /= sum_beta


        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.
```

```
        A_count = [[0. for i in range(self.L)] for j in range(self.L)]
        A_sum = [0. for i in range(self.L)]

        # For each input sequence:
        for y in Y: # Y is a list of lists of length L, so y should be a list,␣
↪each y[i] is a mood tag
            for i in range(len(y) - 1): # A is calculated by
                A_count[ y[i] ][ y[i + 1] ] += 1
                A_sum[y[i]] += 1

        for curr in range(self.L): # to normalize the A matrix
            for nxt in range(self.L):
                self.A[curr][nxt] = A_count[curr][nxt] / A_sum[curr]

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

        O_count = [[0. for i in range(self.D)] for j in range(self.L)] # O = L␣
↪x D, (i, j) is the probability of emitting observation j given state i.
        O_sum = [0. for i in range(self.L)]

        for x, y in zip(X,Y):
            for i in range(len(y)):
                O_count[ y[i] ][ x[i] ] += 1
                O_sum[ y[i] ] += 1

        for curr in range(self.L):
            for nxt in range(self.D):
                self.O[curr][nxt] = O_count[curr][nxt] / O_sum[curr]

        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 iteration in range(N_iters):

            A_count = [[0. for i in range(self.L)] for j in range(self.L)]
            A_sum = [0. for i in range(self.L)]
            O_count = [[0. for i in range(self.D)] for j in range(self.L)]
            O_sum = [0. for i in range(self.L)]

            # for each list
            for x in X:
                M = len(x)      # Length of sequence.
                # expectation step: given A & O matrix, predict probs of y's␣
↪for each training x
                # use forward-backward algorithm,   alpha & beta    [1,...,M]
                alphas = self.forward(x, normalize=True) # (M+1)xL, (i, j)is␣
↪alpha_j(i), probability of observing prefix x^1:i and state y^i = j.
                betas = self.backward(x, normalize=True)

                Marginals = [0. for _ in range(self.L)]
                for seq in range(1, M + 1): # for each prefix x^1:i
                    for curr in range(self.L): #   states/tags
                        Marginals[curr] = alphas[seq][curr] * betas[seq][curr]

                    Marginals_sum = sum(Marginals)
                    for curr in range(self.L):
                        Marginals[curr] /= Marginals_sum # normalized P(y^i␣
↪(curr) | x),     O

                        # Maximization Step: Use y's to estimate new (A,O)
                        O_count[curr][ x[seq - 1] ] += Marginals[curr] #␣
↪seq 1 ,  x[] 0
                        O_sum[curr] += Marginals[curr]
                        if seq != M: # A is calculated by
                            A_sum[curr] += Marginals[curr] # for unsupervised␣
↪learning, there is no tag,    A_count

                #    P(y^i, y^i+1 | x)  A
                for seq in range(1, M):
                    A_update = [[0. for _ in range(self.L)] for _ in range(self.
↪L)]

                    for curr in range(self.L):
                        for nxt in range(self.L):
                            A_update[curr][nxt] = alphas[seq][curr] * self.
↪A[curr][nxt] * self.O[nxt][x[seq]] * betas[seq + 1][nxt]
                            # seq 1 ,  x[] 0
```

```python
                    A_update_sum = sum( [sum(A_update[i]) for i in range(len(A_update)) ] )

                    for curr in range(self.L):
                        for nxt in range(self.L):
                            A_update[curr][nxt] /= A_update_sum # normalized P(y^i, y^i+1 | x)

                    for curr in range(self.L):
                        for nxt in range(self.L):
                            A_count[curr][nxt] += A_update[curr][nxt]

            for curr in range(self.L): # to normalize the A & O matrix
                for nxt in range(self.L):
                    self.A[curr][nxt] = A_count[curr][nxt] / A_sum[curr]

            for curr in range(self.L):
                for nxt in range(self.D):
                    self.O[curr][nxt] = O_count[curr][nxt] / O_sum[curr]
        pass


    def get_data_with_distribute(self, dist): #
        r = random.random()
        for i, p in enumerate(dist):
            if r < p:
                return i
            r -= p

    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 = []
```

```
        y_start = random.randint(0, self.L -1) # starting state is chosen␣
↪uniformly at random
        states.append(y_start)

        for i in range(M):

            # Generate observation/emission x
            #idx_random = random.randint(0, self.D -1)
            #max_value = max(self.O[ states[i] ])
            #x_index = self.O[ states[i] ].index(max_value)
            x_index = self.get_data_with_distribute(self.O[ states[i] ])
            emission.append(x_index)

            # Generate next state y.
            #idx_random = random.randint(0, self.L -1)
            y_index = self.get_data_with_distribute(self.A[ states[i] ])
            states.append(y_index)

        return emission, states[:-1]


    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()
    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.
    '''
```

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

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


    # Randomly initialize and normalize matrix A.
    random.seed(2020)
    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.
    random.seed(155)
    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
```

2_notebook

March 5, 2020

1 Problem 2

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

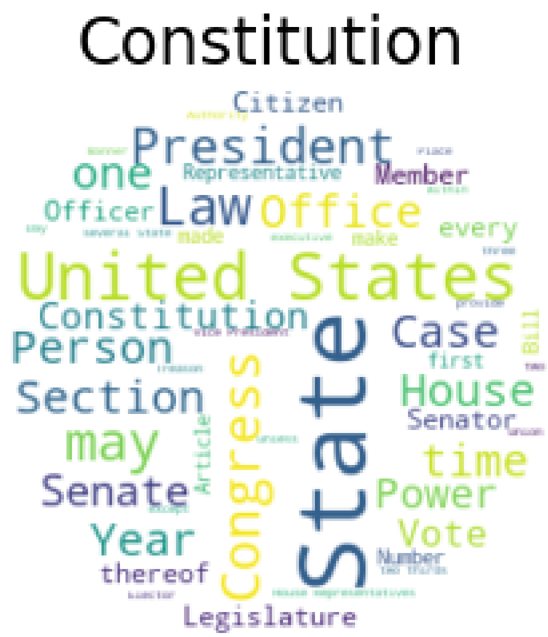
Assuming your HMM module is complete and saved at the correct location, you can simply run all cells in the notebook without modification.

[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
)
```

1.1 Visualization of the dataset

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

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



1.2 Training an HMM

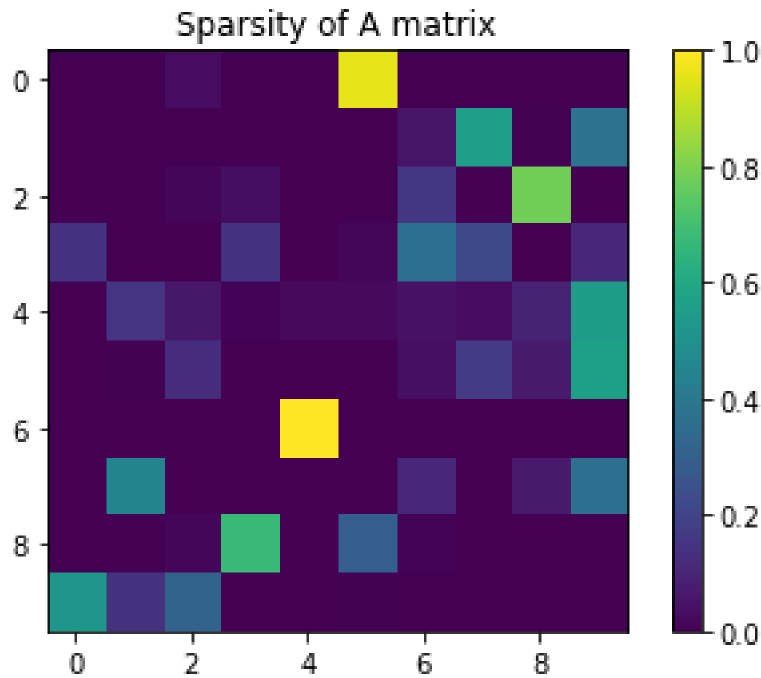
Now we train an HMM on our dataset. We use 10 hidden states and train over 100 iterations:

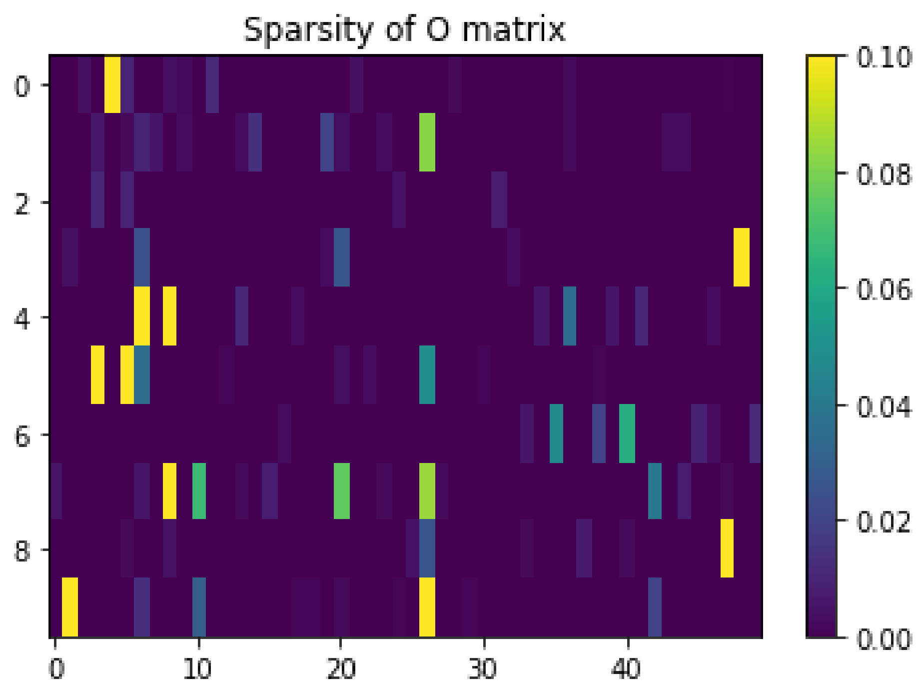
[3]:
```
obs, obs_map = parse_observations(text)
hmm8 = unsupervised_HMM(obs, 10, 100)
```

1.3 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?

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


Sparsity of A matrix
0
2
4
6
8
0
2
4
6
8
1.0
0.8
0.6
0.4
0.2
0.0


Sparsity of O matrix
0
2
4
6
8
0
10
20
30
40
0.10
0.08
0.06
0.04
0.02
0.00

1.4 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.

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

```
Sample Sentence:
====================
War office the concurrence necessary and unless in state obliged to exports
thirds be public thereby the elections states and expiration of an and
counterfeiting…
```

1.5 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?

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

```
Sample Sentence:
====================
Or justice the shall state the shall emit court effect together any pay shall
the and bound of states originated such the member capitation congress…
```

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

```
Sample Sentence:
====================
Legislature tonnage and and president and electors representative deprived
effect or section and number between to that and shall the senate before by the
whom…
```

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

```
Sample Sentence:
====================
```

```
Not cases have of judicial blood impeachment law their the fourths or be shall
of states of the place adjournment concurrence of representatives or on…
```

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

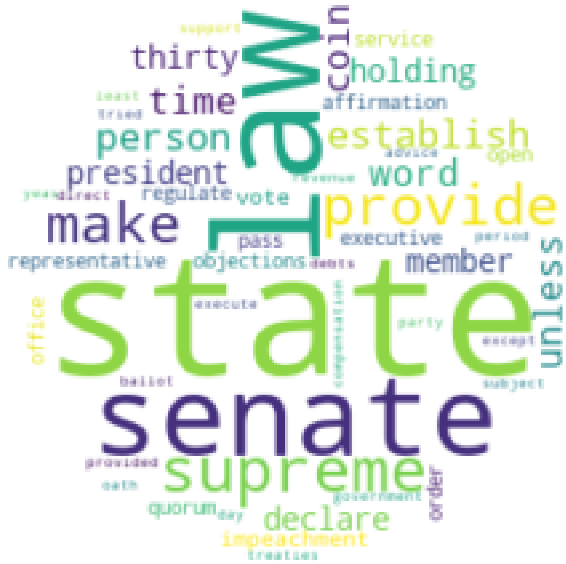
```
Sample Sentence:
====================
To inferior of a monday maritime the judges of behaviour and arsenals and taxes
been yeas concur departments and diminished to work different may a…
```

1.6 Part I: Visualizing the wordcloud of each state

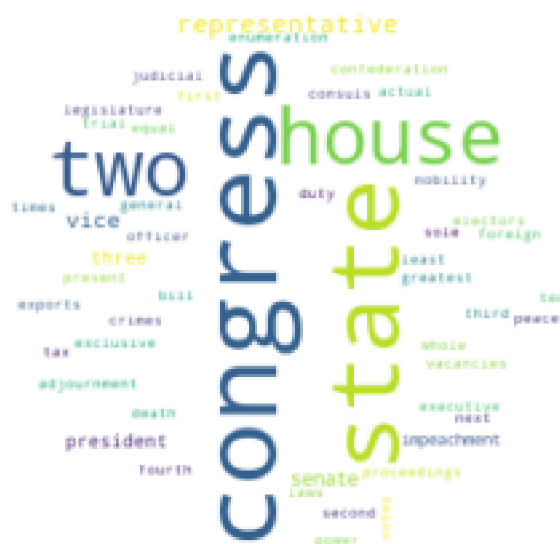
Below, we visualize each state as a wordcloud by sampling a large emission from the state:

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

State 1



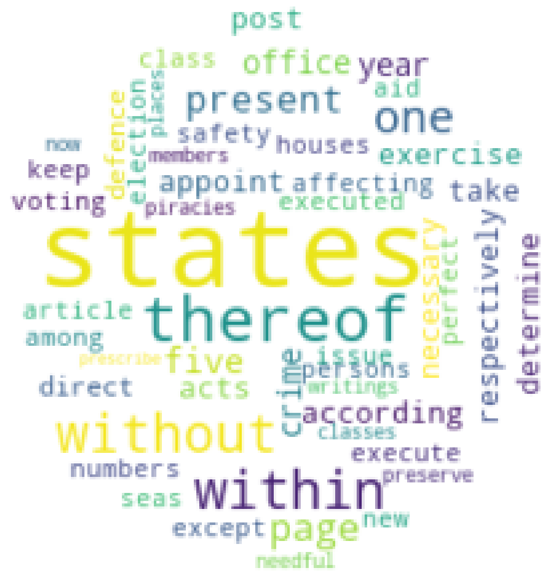
State 2



State 3

State 4

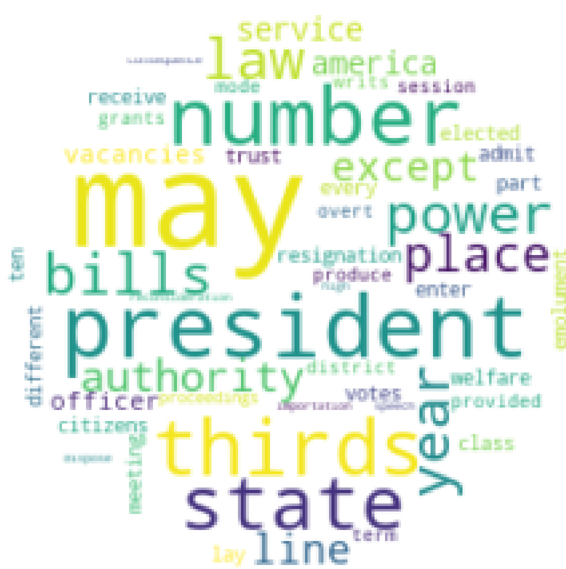
State 5

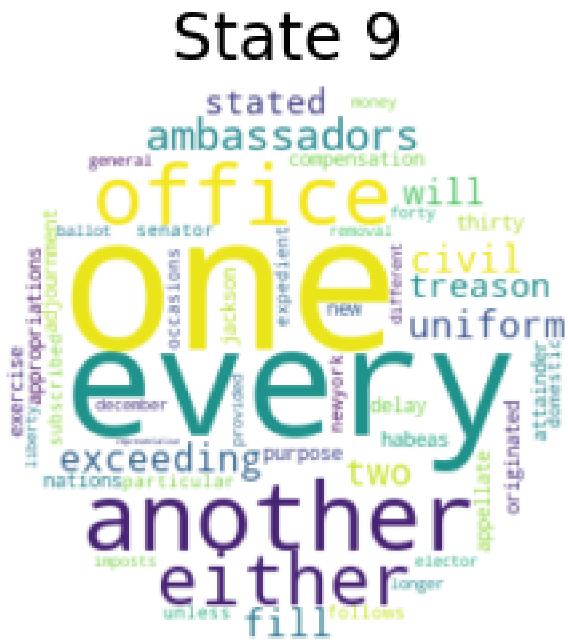


State 6

State 7

State 8





1.7 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.

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

```
Animating…
```

[11]:
```
<IPython.core.display.HTML object>
```

Nor bound in the form determine the consent