CS156 - Assignment 7

Akmarzhan Abylay

December 2020

Assignment 7

1 - Language detection

There are three languages: A, B, and C. Each language uses the same set of symbols: "A, o, e, t, p, g, and k". However, each language uses the symbols differently. In each of these languages we can model everything as P(next symbol | current symbol).

1. There is training data available for each language. This consists of several files each generated by sampling from a Markov model. Using python, build a Markov model for each of the languages.
2. Now use the Markov model and Bayes' rule to classify the test cases. Write down how you used Bayes' rule to get your classifier. Give the full posterior distribution for each test case.

Solution: I first import all of the files for training and testing.

[1]:
```
from glob import glob
import numpy as np

#extracting the data
def extract(file):
    data = glob(file)
    lst = []
    for path in data:
        lst.append(open(path).read())
    return lst

#getting the training data for each language
train_A = extract('symbol/language-training-langA*')
train_B = extract('symbol/language-training-langB*')
train_C = extract('symbol/language-training-langC*')

#getting the testing data for each language
test = extract('symbol/language-test*')
```

We need to first define the initial distribution and I do so by simply counting the number of times each character appears in the data for each language. After I do that, I get three lists with 7 elements each representing the probability of seeing that character in the corresponding language.

[2]:
```
symbols = ["A", "o", "e", "t", "p", "g", "k"]

#initial distributions for each language
def init_dist(file):
    init = np.zeros((7, 1))
    for i, letter in enumerate(symbols):
        for doc in file:
            init[i] += doc.count(letter)

    init = init/sum(init)
    return init

init_A = init_dist(train_A)
init_B = init_dist(train_B)
init_C = init_dist(train_C)

init_test = init_dist(test)
```

Now we define the transition matrix. It will represent the probability of getting from one character to another. As mentioned in class, we can first count how many jumps we made from state `i` to state `j` and normalize it over the sum of all jumps from state `i`. In class, we used the column representation, but in my case, I am using the row representation (i.e., the vertical side represents the current state and the horizontal side represents the next state), which is why we need the sum over the row to be equal to 1. In other words,

$$w_{ij} = \frac{c_{ij}}{\sum_j c_{ij}},$$

where c_{ij} is the matrix with counts and w_{ij} is our transition matrix.

We can also show this by considering the following equation:

$$p(x) = \prod_{ij} w_{ij}^{c_{ij}}$$

We can take the log of $p(x)$ and find a partial derivative over w_{ij}. We get:

$$log p(x) = \sum_{ij} c_{ij} log w_{ij}$$
$$\frac{\partial log p(x)}{\partial w_{ij}} = \frac{c_{ij}}{w_{ij}} \; s.t. \sum_j w_{ij} = 1$$

We can use Lagrange multipliers to do that, so we simply construct a maximization problem and solve it.

$$\mathcal{L} = \sum_{ij} c_{ij} log w_{ij} - \lambda(\sum_j w_{ij} - 1)$$
$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \frac{c_{ij}}{w_{ij}} - \lambda = 0$$

To find the multiplier, we can consider the sum over j to get:

$$w_{ij} = \frac{c_{ij}}{\lambda}$$
$$\sum_j w_{ij} = \frac{\sum_j c_{ij}}{\lambda}$$

Given from the constraint, it should be equal to 1, so $\lambda = \sum_j c_{ij}$, which means that our transition matrix could be defined as:

$$w_{ij} = \frac{c_{ij}}{\sum_j c_{ij}}$$

[3]:
```
def transition_matrix(data):

    M = np.zeros((7, 7))

    #counting how many times we see some letter
    #given another letter previously
    for i, file in enumerate(data):
        for j in range(len(file)-1):
            for s in range(7):
                if file[j] == symbols[s]:
                    for k in range(7):
                        if file[j+1] == symbols[k]:
                            M[s][k] +=1

    #normalizing
    for i, row in enumerate(M):
        M[i] = row/sum(row)

    return M

matrix_A = transition_matrix(train_A)
matrix_B = transition_matrix(train_B)
matrix_C = transition_matrix(train_C)
```

For the classification step, we need to use the Bayes theorem. We want to find the probability of a language given some string. Based on the highest probability within the three languages, we decide which language a string belongs to (i.e., most likely belongs to). Our equation will look like:

$$p(lang|string) = \frac{p(string|lang)p(lang)}{p(string)}$$

Let's break this down into different pieces:

1. **p(lang | string)** - posterior that we aim to find.
2. **p(string | lang)** - likelihood, which defines how likely it is to see the relevant string given that it is from a specific language. We find it by using the transition matrix we specified earlier.

Since we are using Markov chains, we think that the current character's probability only depends on the previous character. For example, suppose we want to find the likelihood of `Ao`. In that case, we will multiply the value for A from the initial distribution by the probability of going from A to o (i.e., w_{Ao}).

3. **p(lang)** - prior probability for each language. Since we are not given any information about that, we assume that it is uniform, so we have $\frac{1}{3}$ for all languages.

4. **p(string)** - evidence. We find it by multiplying each likelihood (i.e., each language to the corresponding string) by the probability of a language and sum them. For example, since we have three languages, it is equal to:

$$p(string) = p(string|lang_1)p(lang_1) + p(string|lang_2)p(lang_2) + p(string|lang_3)p(lang_3)$$

We then choose the one with the highest probability and set it as our predicted language. I am rounding the posterior probabilities to the third decimal in the end for better presentation, so some of the very small probabilities can turn into 0 when they are not actually 0.

[4]:
```
def clf(test):
    for f, file in enumerate(test):
        like_A, like_B, like_C = [], [], []
        for i in range(len(file)-1):
            for j in range(len(symbols)):
                if i==0:
                    init = symbols.index(file[i])
                    like_A.append(init_A[init])
                    like_B.append(init_B[init])
                    like_C.append(init_C[init])
                else:
                    if file[i] == symbols[j]:
                        m = j
                    if file[i+1] == symbols[j]:
                        n = j

        #appending the probabilities to then find the likelihood
        like_A.append(matrix_A[m][n])
        like_B.append(matrix_B[m][n])
        like_C.append(matrix_C[m][n])

        #evidence term
        evidence = 1/3*np.prod(like_A)+1/3*np.prod(like_B)+1/3*np.prod(like_C)

        #posterior probability, we could also skip 1/3, since it is the same
        #for all languages, but I decided to put it anyways
        post_A = 1/3*np.prod(like_A)/evidence
        post_B = 1/3*np.prod(like_B)/evidence
        post_C = 1/3*np.prod(like_C)/evidence

        maxi = max([post_A, post_B, post_C])
```

```
        language = ["A" if maxi==post_A else "B" if maxi==post_B else "C"]
        print("[File %d]" %(f+1))
        print("Posteriors:", [round(post_A[0], 3), round(post_B[0], 3),␣
 ↪round(post_C[0], 3)])
        print("Language:", language[0])
        print("\n")

clf(test)
```

```
[File 1]
Posteriors: [0.011, 0.947, 0.042]
Language: B


[File 2]
Posteriors: [0.763, 0.0, 0.237]
Language: A


[File 3]
Posteriors: [0.048, 0.788, 0.164]
Language: B


[File 4]
Posteriors: [0.115, 0.855, 0.03]
Language: B


[File 5]
Posteriors: [0.174, 0.784, 0.041]
Language: B


[File 6]
Posteriors: [0.258, 0.0, 0.742]
Language: C


[File 7]
Posteriors: [0.024, 0.97, 0.006]
Language: B


[File 8]
Posteriors: [0.018, 0.129, 0.852]
Language: C
```

```
[File 9]
Posteriors: [0.029, 0.048, 0.923]
Language: C


[File 10]
Posteriors: [0.332, 0.0, 0.668]
Language: C
```

2 - Speaker identification

There are three people in a room. Each says about 10 phonemes, before being randomly interrupted by someone else. When they speak they all sound the same, however each person tends to use different phonemes in their speech. Specifically we can model the following transition probabilities that someone will interrupt the current speaker: P(speaker i at time t+1 | speaker j at time t). We can also model the probability over phonemes given a particular speaker: P(phoneme | speaker i). The phonemes are identical to the ones introduced in problem 1 (but the transition matrices are obviously different, since they take a different form altogether).

1. **Write down the update equations that you will need to train a hidden Markov model. Using the information given above, write down a sensible initialization for the transition matrix.**

The main difference between a Markov model and a hidden Markov model is that we now have a complication of hidden states. So, in this case, the speaker is the hidden variable, and the phenomes they say are the observable variables. In addition to the transition matrix (i.e., matrix defining the probabilities of moving from speaker i to speaker j), we now also have the emission matrix (i.e., the matrix representing the probabilities of seeing different observations given that we are in a particular state). The below equations are taken from *this resource* provided in the PCW for 13.2.

We will first define:

$$\xi_t(i,j) = \frac{\alpha_t(i) a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)}{\sum_{j=1}^{N} \alpha_t(j) \beta_t(j)}$$

Here α is the probability of being in a given state given all the past observations. It is given by:

$$\alpha_t(j) = \sum_{i=1}^{N} \alpha_{t-1} a_{ij} b_j(O_t)$$

It is initialized as $\alpha_1(j) = \pi_j b_j(O_1)$ where $1 \leq j \leq N$ and π_j is the probability that the Markov chain will start in state j (i.e., taken from the initial probability distribution).

β is the probability of a given state knowing what comes in the future but not knowing what came in the past. It is given by:

$$\beta_t(i) = \sum_{j=1}^{N} a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)$$

It is initialized as $\beta_T(i) = 1$ where $1 \leq i \leq N$. Here, ξ is the probability of transitioning from state i to state j at time t (normalized). a_{ij} is the transition matrix. O_t is the observation at time t.

Our **first update equation** is:

$$\bar{a}_{ij} = \frac{\text{expected number of transitions from i to j}}{\text{expected number of transitions from i}}$$

$$\bar{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \sum_{k=1}^{N} \xi_t(i,k)}$$

Here, we update the transition matrix.

Our **second update equation** is:

$$\bar{b}_j(v_k) = \frac{\text{expected number of times observing symbol } v_k \text{ given state j}}{\text{expected number of times in state j}}$$

$$\bar{b}_j(v_k) = \frac{\sum_{t=1,\, s.t. O_t = v_k}^{T-1} \gamma_t(j)}{\sum_{t=1}^{T-1} \gamma_t(j)}$$

γ is the probability that we are in some state given what's come before and after (fusion of α and β). Here, we update the emission matrix, and $O_t = v_k$ means we are only considering the observations at time t, which were v_k.

Given these two update equations, we re-estimate the transition and emission matrices. This is the EM-algorithm since we are computing the expected values for γ and ξ from previous (or initial) a and b probabilities (i.e., E-step) and then plug them into the update equations to recompute the a and b (i.e., M-step).

A sensible initialization for the transition matrix is:

$$\begin{bmatrix} & \underline{S1} & \underline{S1} & \underline{S1} \\ S1| & 0.9 & 0.05 & 0.05 \\ S2| & 0.05 & 0.9 & 0.05 \\ S3| & 0.05 & 0.05 & 0.9 \end{bmatrix}$$

This is because each speaker is interrupted after about 10 phenomes. If, for example, speaker 1 is talking at the moment (i.e., said 1 phenome), there is a 0.9 chance that speaker 1 will continue and a 0.05 chance for both speakers 2 and 3 to interrupt (i.e., I assumed that they have equal chances). I also assumed uniformity for the initial distribution so that each speaker has an equal probability of speaking. I randomly initialized the emission matrix but made sure that the rows sum up to 1.

2. **Write your own python code to train a hidden Markov model on the data. You may look at code online, but will need to reference any code that helps you with your implementation.**

I used an existing library *hmm* and *this resource* for this step. Both of them are official `hmm` tutorials. I first needed to turn the data into an appropriate format to fit the hmm library. Since the data was

presented as letters, I turned it into numbers (i.e., I just mapped the letters to numbers, so there is no information loss, just different notations).

[5]:
```
import numpy as np
import matplotlib.pyplot as plt

speaker = open("speaker.txt").read()
symbols = symbols = ["A", "o", "e", "t", "p", "g", "k"]
```

[6]:
```
#making the data appropriate for the hmm function
dict_ = {symbol: number for number, symbol in enumerate(symbols)}
change = lambda x: dict_.get(x)
X = np.array(list(map(change, speaker))).reshape(-1, 1)
```

[7]:
```
import hmmlearn.hmm as hmm

hm = hmm.MultinomialHMM(n_components=3, n_iter=1000)

#transition matrix - choose numbers based on the given info
hm.transmat_ = np.array([[0.9, 0.05, 0.05],
                         [0.05, 0.9, 0.05],
                         [0.05, 0.05, 0.9]])

#emission matrix - arbitrarily set - the rows need to sum up to 1
hm.emissionprob_ = np.array([np.random.dirichlet(np.ones(7)) for i in range(3)])

#initial distribution - how likely one person is to speak
hm.startprob_ = np.array([1/3, 1/3, 1/3])
```

We will train the algorithm by calling the `fit()` method. It accepts a list of sequences we want to predict rather than a single sequence, and since we want to see who is more likely to say each character (i.e., 1000), I reshaped the data in one of the previous steps. We have unobserved (hidden) variables, which is why we use an expectation-maximization algorithm (which was mentioned above). The EM steps are described above in the first part of the answer with all update equations.

The optimal hidden states are then found through the `predict()` method. The library automatically uses the Viterbi algorithm to find the most likely sequence of hidden variables (i.e., speakers) that corresponds to the sequence of observations (i.e., phenomes). This step is usually referred to as "decoding." Below we can see the most likely speakers for each phenome.

[8]:
```
hm.fit(X)
person = hm.predict(X)
person[:100]
```

[8]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 0, 0, 0, 2, 2, 2, 2, 2, 2, 2,
```

```
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

We also can print the posterior probabilities for each observed state (i.e., 1000 phenomes) as below.

[9]:
```
probabilities = hm.predict_proba(X)
probabilities
```

[9]:
```
array([[9.99999969e-01, 2.31580916e-86, 3.09537669e-08],
       [9.60840183e-01, 4.84807796e-03, 3.43117388e-02],
       [9.94686633e-01, 1.06393964e-03, 4.24942709e-03],
       ...,
       [1.24050383e-01, 7.24763273e-03, 8.68701984e-01],
       [3.80811996e-02, 2.75085697e-03, 9.59167943e-01],
       [3.45143119e-02, 8.38983678e-03, 9.57095851e-01]])
```

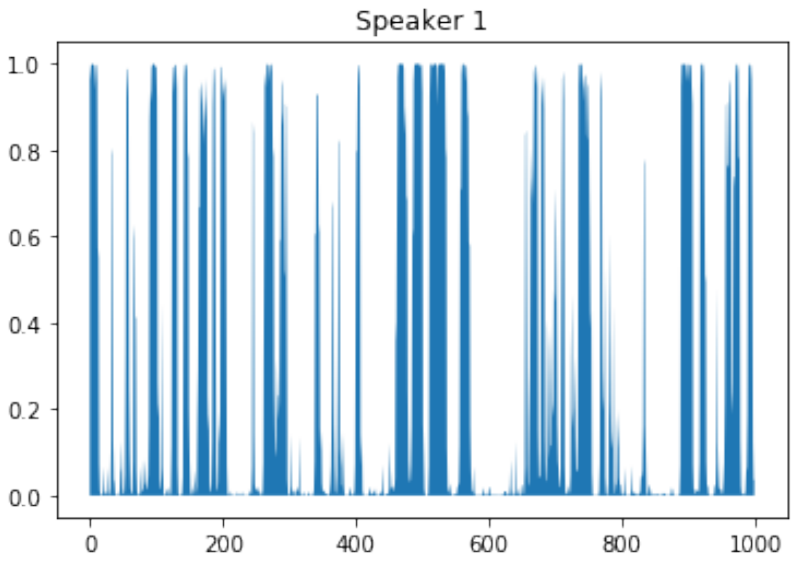
3. **From matplotlb use a stackplot to show the probability of a particular person speaking.**

I plotted the probabilities below. For all speakers, it was pretty obvious when they were talking as their lines are clear and have visible gaps. For example, we can tell that the model thinks the first 14 phenomes were said by speaker 1, the next 19 phenomes were said by speaker 3, the next 2 phenomes were given by speaker 1, and so on. We can observe this from both the predictions and the plot, as we see the probabilities of each speaker speaking for each of the 1000 phenomes.

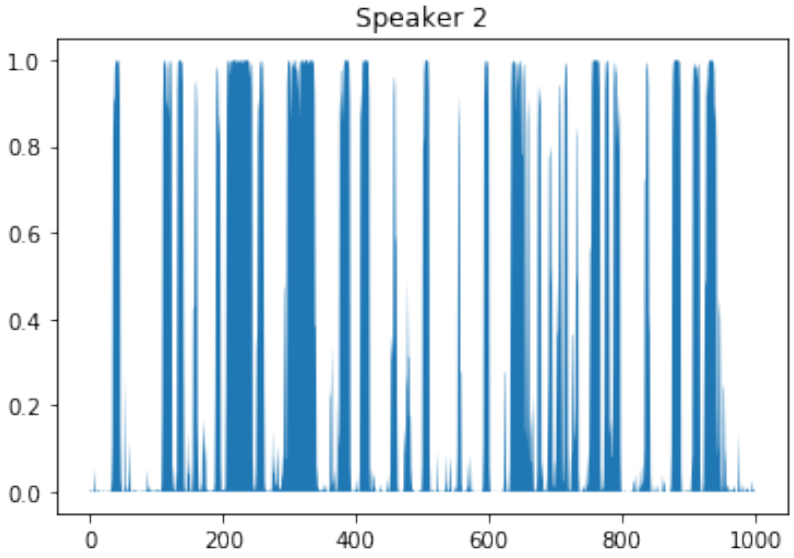
We defined an equal probability for each of the speakers to talk (i.e., 1/3), and the model doesn't really differentiate between the speaker numbers (i.e., 1, 2, or 3), but it can tell us when a new speaker starts speaking.

[10]:
```
x = range(0, len(probabilities))

plt.stackplot(x, probabilities[:, 0])
plt.title("Speaker 1")
```



[11]:
```
plt.stackplot(x, probabilities[:, 1])
plt.title("Speaker 2")
```



[12]:
```
plt.stackplot(x, probabilities[:, 2])
plt.title("Speaker 3")
```