12 Discrete Hidden Markov Models

Lab Objective: *Understand how to use discrete Hidden Markov Models.*

In this lab, we explore Hidden Markov Models (HMMs) with discrete state and observation spaces. Assume the state space $\mathscr{X}$ and observation space $\mathscr{Z}$ are finite sets where $|\mathscr{X}| = n$ and $|\mathscr{Z}| = m$. In addition, a discrete state-space HMM has parameters $\boldsymbol{\theta} = (\boldsymbol{\pi}, A, B)$ and an observation sequence $\boldsymbol{z}$. We would like to answer three questions about an HMM:

1. What is the likelihood that our model generated the observation sequence? In other words, what is $P(\boldsymbol{z} \,|\, \boldsymbol{\theta})$?
2. What is the most likely state sequence $\boldsymbol{x}$ to have generated $\boldsymbol{z}$, given $\boldsymbol{\theta}$?
3. How can we choose the parameters $\boldsymbol{\theta}$ that maximize $P(\boldsymbol{z} \,|\, \boldsymbol{\theta})$?

The first question is answered using the *forward pass* algorithm. For the second question, the approach taken in this lab will be to find the state sequence maximizing the expected number of correct states. The third question is an example of *unsupervised learning*, since we are attempting to learn (or fit) model parameters using data (the observation sequence $\boldsymbol{z}$) that is devoid of human-provided labels (the corresponding state sequence); the algorithm does not rely on human supervision or input.

In this context $\boldsymbol{\theta} = (\boldsymbol{\pi}, A, B)$, where $\boldsymbol{\pi}$ is a stochastic vector of length n (the initial state distribution), A is a $n \times n$ column-stochastic matrix (the state transition model), and B is a $m \times n$ column-stochastic matrix (the state observation model). Further, $\boldsymbol{z}$ is a vector of length T with values in the set $\mathscr{Z} = \{0, 1, 2, \dots, m-1\}$.

Throughout this lab, we will be using the following toy HMM to verify your code.

```
>>> # toy HMM example to be used to check answers
>>> pi = np.array([.6, .4])
>>> A = np.array([[.7, .4],[.3, .6]])
>>> B = np.array([[.1,.7],[.4, .2],[.5, .1]])
>>> z = np.array([0, 1, 0, 2])
```

Problem 1. To start off your implementation of the HMM, define a class object which you should call `HMM`. Then add the initialization method, storing the `self` aspects `pi`, `A`, and `B` as `None` objects. You will be adding methods throughout the remainder of the lab.

The Forward Pass

Our first task is to efficiently compute $P(\boldsymbol{z} \,|\, \boldsymbol{\theta})$. We can do this using the *forward pass* algorithm.

First, let $\alpha_t(i) = P(z_0, \dots, z_t, x_t = i \,|\, \boldsymbol{\theta})$. Then using the law of total probability and $\alpha_t(i)$, we can efficiently compute $P(\boldsymbol{z} \,|\, \boldsymbol{\theta})$ as

$$P(\boldsymbol{z} \,|\, \boldsymbol{\theta}) = \sum_{i \in \mathscr{X}} \alpha_{T-1}(i).$$

Now we must use a rescaled version of the forward pass to prevent the $\alpha_t(i)$s from becoming too small as t gets large. Let $\odot$ denote the Hadamard (entry-wise) product of arrays. The algorithm is as below.

Algorithm 12.1 Forward Pass Algorithm

```
1: procedure Forward Pass Algorithm
2:     for t=0 do
3:         Set $\widehat{\alpha}_0(i) = \pi_i \cdot B_{z_0 i}, \forall i \in \mathscr{X}$
4:         Let $c_0 = 1/\left(\sum_{j \in \mathscr{X}} \widehat{\alpha}_0(j)\right)$
5:         Set $\widehat{\alpha}_0(i) = c_0 \odot \widehat{\alpha}_0(i), \forall i \in \mathscr{X}$
6:     for t=1, ..., T - 1 do
7:         Compute $\tilde{\alpha}_t(i) = \sum_{j \in \mathscr{X}} \widehat{\alpha}_{t-1}(j) \cdot A_{ij} \cdot B_{z_t i}, \forall i \in \mathscr{X}$
8:         Compute $c_t = 1/\left(\sum_{j \in \mathscr{X}} \tilde{\alpha}_t(j)\right)$
9:         Rescale by setting $\widehat{\alpha}_t(i) = c_t \cdot \tilde{\alpha}_t(i), \forall i \in \mathscr{X}$
```

The matrix $\widehat{\alpha}$ will be of use when fitting parameters, but we can compute the desired log probability using the scaling factors c_t as follows:

$$\log P(\boldsymbol{z} \,|\, \boldsymbol{\theta}) = -\sum_{t=0}^{T-1} \log c_t.$$

Problem 2. Implement the forward pass by adding the following method to your class:

```python
def _forward(self, z):
    """
    Compute the scaled forward probability matrix and scaling factors.

    Parameters
    ----------
    z : ndarray of shape (T,)
        The observation sequence

    Returns
    -------
    alpha : ndarray of shape (T, n)
        The scaled forward probability matrix
    c : ndarray of shape (T,)
        The scaling factors c = [c_0, c_1, ..., c_{T-1}]
    """
    pass
```

To verify that your code works, you should get the following output using the toy HMM:

```
>>> h = HMM()
>>> h.pi = pi
>>> h.A = A
>>> h.B = B
>>> alpha, c = h._forward(z)
>>> print(-np.log(c).sum()) # the log prob of observation
-4.6429135909
```

The Backward Pass

The backward pass produces values that can be used to calculate the most likely state sequence corresponding to an observation sequence.

We compute a scaled backward probability matrix $\widehat{\beta}$ of dimension $T \times n$ as follows:

Algorithm 12.2 Backward Pass Algorithm

```
1: procedure Backward Pass Algorithm
2:     When $t = T-1$, set $\widehat{\beta}_{T-1}(i) = c_{T-1}$, $\forall i \in \mathscr{X}$
3:     for $t = T-2, \ldots, 0$ do
4:         Compute $\tilde{\beta}_t(j) = \sum_{i \in \mathscr{X}} A_{ij} \cdot \widehat{\beta}_{t+1}(i) \cdot B_{z_{t+1}i}$, $\forall j \in \mathscr{X}$
5:         Rescale by setting $\widehat{\beta}_t(i) = c_t \cdot \tilde{\beta}_t(i)$, $\forall i \in \mathscr{X}$
```

Problem 3. Implement the backward pass by adding the following method to your class:

```
def _backward(self, z, c):
    """
    Compute the scaled backward probability matrix.

    Parameters
    ----------
    z : ndarray of shape (T,)
        The observation sequence
    c : ndarray of shape (T,)
        The scaling factors from the forward pass
    Returns
    -------
    beta : ndarray of shape (T, n)
        The scaled backward probability matrix
    """
    pass
```

Using the same toy example as before, your code should produce the following output:

```
>>> beta = h._backward(z, c)
>>> print(beta)
[[ 3.1361635   2.89939354]
 [ 2.86699344  4.39229044]
 [ 3.898812    2.66760821]
 [ 3.56816483  3.56816483]]
```

Computing the ξ and γ Probabilities

Having implemented both parts of the forward-backward algorithm, we are closing in on the solution to question three, namely that of fitting parameters $\boldsymbol{\theta}$ that maximize $P(\boldsymbol{z} \mid \boldsymbol{\theta})$. At this stage, we combine the information accumulated in the forward and backward algorithms to produce a three-dimensional array $\boldsymbol{\xi}$ of shape $(T-1) \times n \times n$ whose entries are related to $P(\mathbf{x}_t = i, \mathbf{x}_{t+1} = j \mid \boldsymbol{z}, \boldsymbol{\theta})$, as well as a $T \times n$ matrix $\boldsymbol{\gamma}$ whose entries are related to $P(\mathbf{x}_t = i \mid \boldsymbol{z}, \boldsymbol{\theta})$. The relevant formulae are

$$\xi_t(i,j) = \widehat{\alpha}_t(i) A_{j,i} B_{z_{t+1},j} \widehat{\beta}_{t+1}(j)$$

for $t = 0, \ldots, T-1$ and $i, j \in \mathscr{X}$,

$$\gamma_t(i) = \widehat{\alpha}_t(i) \widehat{\beta}_t(i) / c_t$$

for $t = 0, \ldots, T-1$ and $i \in \mathscr{X}$.

Problem 4. Add the following method to your class to compute the ξ and γ probabilities.

```
def _xi(self, z, alpha, beta, c):
    """
    Compute the xi and gamma probabilities.

    Parameters
    ----------
    z : ndarray of shape (T,)
        The observation sequence
    alpha : ndarray of shape (T, n)
        The scaled forward probability matrix from the forward pass
    beta : ndarray of shape (T, n)
        The scaled backward probability matrix from the backward pass
    c : ndarray of shape (T,)
```

```
        The scaling factors from the forward pass

    Returns
    -------
    xi : ndarray of shape (T-1, n, n)
        The xi probability array
    gamma : ndarray of shape (T, n)
        The gamma probability array
    """
    pass
```

While writing a triply-nested loop may be the simplest way to convert the formula into code, it is possible to use array broadcasting to eliminate two of the loops, which will speed up your code.

Check your code by making sure it produces the following output, using the same toy example as before.

```
>>> xi, gamma = h._xi(z, alpha, beta, c)
>>> print(xi)
[[[ 0.14166321  0.0465066 ]
  [ 0.37776855  0.43406164]]

 [[ 0.17015868  0.34927307]
  [ 0.05871895  0.4218493 ]]

 [[ 0.21080834  0.01806929]
  [ 0.59317106  0.17795132]]]
>>> print(gamma)
[[ 0.18816981  0.81183019]
 [ 0.51943175  0.48056825]
 [ 0.22887763  0.77112237]
 [ 0.8039794   0.1960206 ]]
```

Choosing Better Parameters

After running the forward-backward algorithm and computing the ξ probabilities, we are now in a position to choose new parameters $\boldsymbol{\theta}' = (\boldsymbol{\pi}', A', B')$ that increase the probability of observing our data, i.e.

$$P(\boldsymbol{z} \,|\, \boldsymbol{\theta}') \geq P(\boldsymbol{z} \,|\, \boldsymbol{\theta}).$$

The update formulae are given by

$$
\begin{aligned}
\boldsymbol{\pi}' &= \gamma_0(i) \\
A'_{i,j} &= \frac{\sum_{t=0}^{T-2} \xi_t(i,j)}{\sum_{t=0}^{T-2} \gamma_t(j)} \\
B'_{i,j} &= \frac{\sum_{t=0}^{T-1} \gamma_t(j)\delta_{z_t=i}}{\sum_{t=0}^{T-1} \gamma_t(j)}
\end{aligned}
$$

where $\delta_{z_t=i}$ equals 1 if $z_t = i$, and it equals 0 otherwise.

Problem 5. Implement the parameter update step by adding the following method to your class:

```
def _estimate(self, z, xi, gamma):
    """
    Estimate better parameter values and update self.pi, self.A, and
    self.B in place.

    Parameters
    ----------
    z : ndarray of shape (T,)
        The observation sequence
    xi : ndarray of shape (T-1, n, n)
        The xi probability array
    gamma : ndarray of shape (T, n)
        The gamma probability array
    """
    pass
```

Verify that your code produces the following output on the toy HMM from before:

```
h._estimate(z, xi, gamma)
>>> print(h.pi)
[ 0.18816981  0.81183019]
>>> print(h.A)
[[ 0.55807991  0.49898142]
 [ 0.44192009  0.50101858]]
>>> print(h.B)
[[ 0.23961928  0.70056364]
 [ 0.29844534  0.21268397]
 [ 0.46193538  0.08675238]]
```

Fitting the Model

We are now ready to put everything together into a learning algorithm. Given a sequence of observations, a maximum number of iterations K, and a convergence tolerance threshold ε, we fit a HMM model using the following procedure:

Algorithm 12.3 HMM Fitting Algorithm

```
1: procedure HMM Fitting Algorithm
2:     Randomly initialize parameters $\boldsymbol{\theta} = (\pi, A, B)$.
3:     Compute $\log P(\boldsymbol{z} \mid \boldsymbol{\theta})$
4:     for $i = 0, 1, \ldots, K-1$ do
5:         Run forward pass
6:         Run backward pass
7:         Compute $\xi$ and $\gamma$ probabilities
8:         Update model parameters
9:         Compute $\log P(\boldsymbol{z} \mid \boldsymbol{\theta})$ according to new parameters
10:        if Change in log probabilities is less than $\varepsilon$ then
11:            break
12:        else
13:            continue
```

The most convenient way to randomly initialize stochastic matrices is to draw from the Dirichlet distribution, which produces vectors with nonnegative entries that sum to 1. The following Python code initializes M, $\boldsymbol{\pi}$, A, and B using this technique:

```
>>> # assume N is defined
>>> # define M to be the number of distinct observed states
>>> M = len(set(obs))
>>> pi = np.random.dirichlet(np.ones(N))
>>> A = np.random.dirichlet(np.ones(N), size=N).T
>>> B = np.random.dirichlet(np.ones(M), size=N).T
```

The learning algorithm is essentially an optimization over the parameter space (i.e. the space of tuples of stochastic arrays having the proper dimensions) with respect to the objective function $P(\boldsymbol{z} \mid \boldsymbol{\theta})$. The algorithm is guaranteed to increase the objective function at each iteration, so it is sure to converge. However, the objective function is riddled with local maxima, and so the outcome depends heavily on the randomly selected starting values for $\boldsymbol{\pi}$, A, and B. Figure 12.1 illustrates the issues involved. The log probability stays approximately constant for the first 100 iterations. This indicates that the algorithm is not exploring the parameter space enough, and the parameters found at the 100-th iteration are virtually the same as those found at the first or second iteration. After the first 100 iterations, however, the algorithm is finally able to explore more of the parameter space and hence make better progress toward increasing the objective function. The moral of the story is that you may need to train the HMM a few times, using different starting values, and then keep the model that has the highest log likelihood.

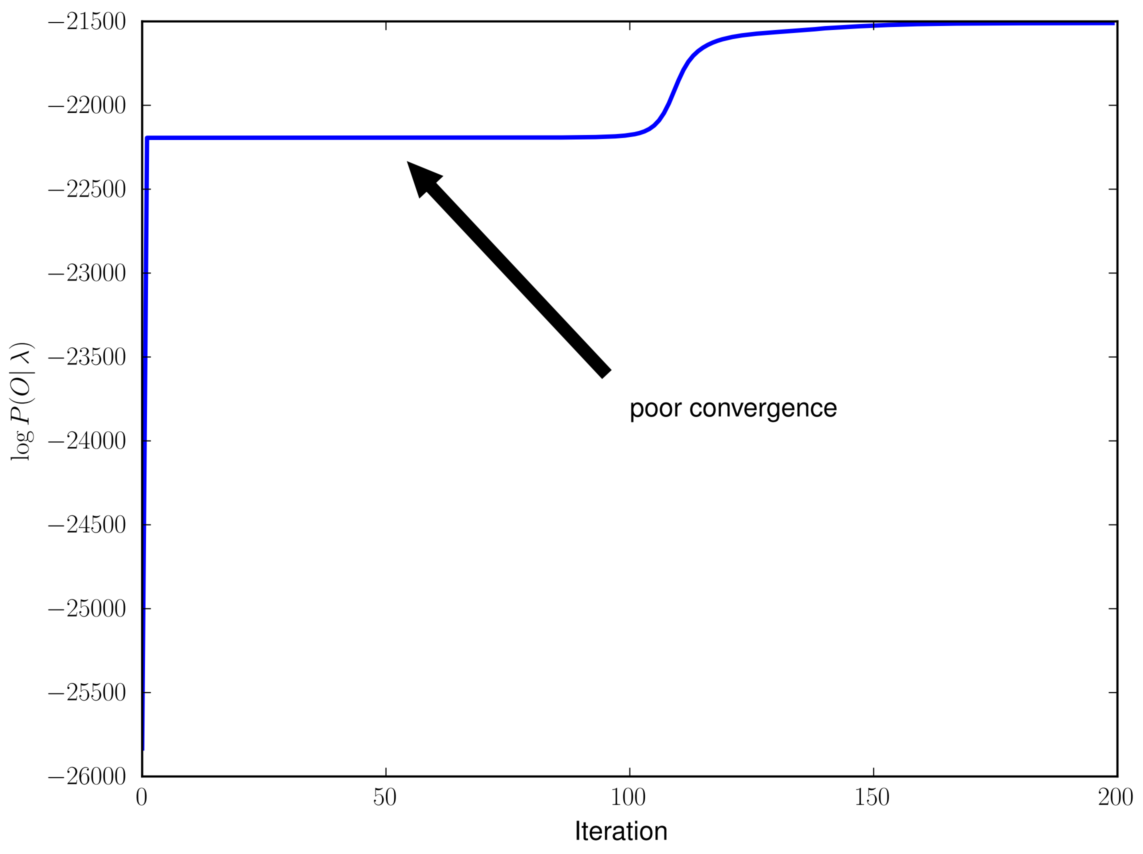

Figure 12.1: The log probabilities for a HMM trained on the Declaration of Independence data with 200 iterations. It takes over 100 iterations for the algorithm to work itself out of a poor local maximum.

Problem 6. Implement the learning algorithm by adding the following method to your class:

```python
def fit(self, z, pi, A, B, max_iter=100, tol=1e-4):
    """
    Fit the model parameters to a given observation sequence.

    Parameters
    ----------
    z : ndarray of shape (T,)
        Observation sequence on which to train the model.
    pi : Stochastic ndarray of shape (n,)
        Initial state distribution
    A : Stochastic ndarray of shape (n, n)
        Initial state transition matrix
    B : Stochastic ndarray of shape (m, n)
        Initial state observation matrix
    max_iter : int
        The maximum number of iterations to take
    tol : float
        The convergence threshold for change in log-probability
    """
    # initialize self.pi, self.A, self.B
    # run the iteration
    pass
```

We now turn to the data found in the file `declaration.txt`. This file contains the text of the Declaration of Independence. We will use the sequence of characters (after stripping out punctuation and converting everything to lower-case) as our observation sequence. In order to convert the raw text into a useable data structure, we need to read in the file, process the string as necessary, and then map the characters to integer values. We provide helper code below to accomplish this task for various files in various languages:

```
>>> import numpy as np
>>> import string
>>> import codecs

>>> def vec_translate(a, my_dict):
>>>     # translate numpy array from symbols to state numbers or vice versa
>>>     return np.vectorize(my_dict.__getitem__)(a)

>>> def prep_data(filename):
>>>     # Get the data as a single string
>>>     with codecs.open(filename, encoding='utf-8') as f:
>>>         data=f.read().lower()  # and convert to all lower case

>>>     # remove punctuation and newlines
>>>     remove_punct_map = {ord(char):
                            None for char in string.punctuation+"\n\r"}
>>>     data = data.translate(remove_punct_map)

>>>     # make a list of the symbols in the data
>>>     symbols = sorted(list(set(data)))

>>>     # convert the data to a NumPy array of symbols
>>>     a = np.array(list(data))

>>>     # make a conversion dictionary from symbols to state numbers
>>>     symbols_to_obsstates = {x:i for i,x in enumerate(symbols)}

>>>     # convert the symbols in a to state numbers
>>>     obs_sequence = vec_translate(a,symbols_to_obsstates)

>>>     return symbols, obs_sequence
```

Now apply this helper code to `declaration.txt`.

```
>>> symbols, obs = prep_data('declaration.txt')
```

Problem 7. You are now ready to train a HMM using the Declaration of Independence data. Use $N = 2$ states and $M =$`len(set(obs))`$= 27$ observation values (26 lower case characters and 1 whitespace character), and run for 150 iterations with the default value for `tol`. Generally speaking, if you converge to a log probability greater than -21550, then you have reached an acceptable set of parameters for this dataset.

Once the learning algorithm converges, analyze the state observation matrix B. Note which rows correspond to the largest and smallest probability values in each column of B, and check the corresponding characters. The code below displays typical results for a well-converged HMM. Note that the `u` before the `"` indicates that the string should be unicode, which will be required for languages other than English.

```
>>> for i in range(len(h.B)):
>>>     print(u"{0}, {1:0.4f}, {2:0.4f}"
                .format(symbols[i], h.B[i,0], h.B[i,1]))
 , 0.0051, 0.3324
a, 0.0000, 0.1247
c, 0.0460, 0.0000
b, 0.0237, 0.0000
e, 0.0000, 0.2245
d, 0.0630, 0.0000
g, 0.0325, 0.0000
f, 0.0450, 0.0000
i, 0.0000, 0.1174
h, 0.0806, 0.0070
k, 0.0031, 0.0005
j, 0.0040, 0.0000
m, 0.0360, 0.0000
l, 0.0569, 0.0001
o, 0.0009, 0.1331
n, 0.1207, 0.0000
q, 0.0015, 0.0000
p, 0.0345, 0.0000
s, 0.1195, 0.0000
r, 0.1062, 0.0000
u, 0.0000, 0.0546
t, 0.1600, 0.0000
w, 0.0242, 0.0000
v, 0.0185, 0.0000
y, 0.0147, 0.0058
x, 0.0022, 0.0000
```

```
z, 0.0010, 0.0000
```

What do you notice about the second column of B? It seems that the HMM has detected a vowel state and a consonant state, without any prior input from an English speaker. Interestingly, the whitespace character is grouped together with the vowels. A HMM can also detect the vowel/consonant distinction in other languages.

Problem 8. Repeat the previous calculation with 3 hidden states and again with 4 hidden states. Interpret/explain your results.
Hint: with 3 hidden states, your print statement will look like the following:

```
>>> print(u"{0}, {1:0.4f}, {2:0.4f}, {2:0.4f}"
             .format(symbols[i], h.B[i,0], h.B[i,1], h.B[i,2]))
```

Now we turn to the Russian file `WarAndPeace.txt`, which is a small subset of the book *War and Peace* by Tolstoy.

```
>>> symbols, obs = prep_data('WarAndPeace.txt')
```

Problem 9. Repeat the same calculation with `WarAndPeace.txt` for 2 and 3 hidden states. Interpret/explain your results. Which Cyrillic characters appear to be vowels?