Lab 5

Discrete Hidden Markov Models

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

Given a discrete state-space Hidden Markov Model (HMM) with parameters λ and an observation sequence O, we would like to answer three questions:

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

The answers to these questions are centered around the *forward-backward* algorithm for HMMs. 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 O) that is devoid of human-provided labels (the corresponding state sequence); the algorithm does not rely on human supervision or input.

We assume throughout this lab that the HMM has a discrete state space of cardinality N and a discrete observation space of cardinality M. In this context $\lambda = (A, B, \pi)$, where A is a $N \times N$ column-stochastic matrix (the state transition model), B is a $M \times N$ column-stochastic matrix (the state observation model), and π is a stochastic vector of length N (the initial state distribution). Further, O is a vector of length T with values in the set $\{1, 2, \ldots, M\}$.

Warning

The mathematical exposition in the lab assumes the standard 1-based indexing of vectors and matrices. Be sure to carefully translate the various formulae into 0-based indexing when implementing these methods for Python coding. This means that, in Python, your array containing the observation sequence O will actually have values in the set $\{0, 1, \ldots, M-1\}$ so that they may be used to index the matrix B correctly.

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

```
>>> # toy HMM example to be used to check answers
>>> A = np.array([[.7, .4],[.3, .6]])
>>> B = np.array([[.1,.7],[.4, .2],[.5, .1]])
>>> pi = np.array([.6, .4])
>>> obs = 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, in which you should set the *self* aspects A, B, and pi to be None objects. You will be adding methods throughout the remainder of the lab.

The Forward Pass

Our first task is to efficiently compute $\log \mathbb{P}(O|\lambda)$. We can do this using the *forward pass* of the forward-backward algorithm. We must take care to compute all values in a numerically stable way; we do this by properly scaling values as necessary.

We compute a scaled forward probability matrix $\widehat{\alpha}$ of dimension $T \times N$ as follows: Let $\widehat{\alpha}_{i,:}, B_{i,:}$ denote the i-th rows of $\widehat{\alpha}$ and B, respectively, let $\odot$ denote the Hadamard (or entry-wise) product of arrays, and let $\langle \cdot, \cdot \rangle$ denote the standard dot product. (Note that here, using 0-based indexing and the toy HMM example, $B_{O_3,:}$ would refer to $[.5, .1]$.) Then

- $c_1 = \langle \pi, B_{O_1,:} \rangle^{-1}$
- $\widehat{\alpha}_{1,:} = c_1(\pi \odot B_{O_1,:})$
- For $t = 2, \ldots, T$:

$$c_t = \langle A\widehat{\alpha}_{t-1,:}, B_{O_t,:} \rangle^{-1}$$
$$\widehat{\alpha}_{t,:} = c_t((A\widehat{\alpha}_{t-1,:}) \odot B_{O_t,:})$$

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:

$$\log \mathbb{P}(O|\lambda) = -\sum_{t=1}^{T} \log c_t.$$

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

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

    Parameters
    ----------
    obs : 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_1,c_2,...,c_T]
    """
    pass
```

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

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

The Backward Pass

The backward pass of the forward-backward algorithm 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:

- $\widehat{\beta}_{T,i} = c_T$ for $i = 1, \dots, N$
- $\widehat{\beta}_{t,:} = c_t A^T (B_{O_{t+1},:} \odot \widehat{\beta}_{t+1,:})$ for $t = T-1, \dots, 1$

(Above, A^T is the *transpose* of A, not the T-th power of A.)

It turns out that

$$\mathbb{P}(\mathbf{x}_t = i | O, \lambda) = \frac{\widehat{\alpha}_{t,i}\widehat{\beta}_{t,i}}{\sum_{j=1}^N \widehat{\alpha}_{t,j}\widehat{\beta}_{t,j}}$$

and so we can easily compute the most likely state at time t by

$$\mathbf{x}_t^* = \text{argmax}_i \widehat{\alpha}_{t,i}\widehat{\beta}_{t,i}.$$

This is the solution to the second question posed at the beginning of the lab.

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

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

     Parameters
    ----------
    obs : 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(obs, c)
>>> print beta
[[ 3.1361635   2.89939354]
 [ 2.86699344  4.39229044]
 [ 3.898812    2.66760821]
 [ 3.56816483  3.56816483]]
```

Computing the δ 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 λ that maximize $\mathbb{P}(O|\,\lambda)$. At this stage, we combine the information accumulated in the forward-backward algorithm to produce a three-dimensional array $\widehat{\delta}$ of shape $(T-1)\times N \times N$ whose entries are related to $\mathbb{P}(\mathbf{x}_t = i, \mathbf{x}_{t+1} = j|\,O, \lambda)$, as well as a $T \times N$ matrix $\widehat{\gamma}$ whose entries are related to $\mathbb{P}(\mathbf{x}_t = i|\,O, \lambda)$. The relevant formulae are

$$\widehat{\delta}_{t,i,j} = \frac{\widehat{\alpha}_{t,i} A_{j,i} B_{O_{t+1},j} \widehat{\beta}_{t+1,j}}{\sum_{k,l} \widehat{\alpha}_{t,k} A_{l,k} B_{O_{t+1},l} \widehat{\beta}_{t+1,l}}$$

for $t = 1, \ldots, T-1$ and $i, j = 1, \ldots, N$,

$$\widehat{\gamma}_{t,i} = \sum_{j=1}^{N} \widehat{\delta}_{t,i,j}$$

for $t = 1, \dots, T-1$ and $i = 1, \dots, N$, and finally

$$\widehat{\gamma}_{T,:} = \frac{\widehat{\alpha}_{T,:} \odot \widehat{\beta}_{T,:}}{\langle \widehat{\alpha}_{T,:}, \widehat{\beta}_{T,:} \rangle}.$$

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

```python
def _delta(self, obs, alpha, beta):
    """
    Compute the delta probabilities.

    Parameters
    ----------
    obs : 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

    Returns
    -------
    delta : ndarray of shape (T-1,N,N)
        The delta 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.

```python
>>> delta, gamma = h._delta(obs, alpha, beta)
>>> print delta
[[[ 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 δ probabilities, we are now in a position to choose new parameters $\lambda' = (A', B', \pi')$ that increase the probability of observing our data, i.e.

$$\mathbb{P}(O|\,\lambda') \geq \mathbb{P}(O|\,\lambda).$$

The update formulas are given by

$$\begin{aligned} A'_{i,j} &= \frac{\sum_{t=1}^{T-1} \widehat{\delta}_{t,j,i}}{\sum_{t=1}^{T-1} \widehat{\gamma}_{t,j}} \\ B'_{i,j} &= \frac{\sum_{t=1}^{T} \widehat{\gamma}_{t,j} 1_{\{O_t=i\}}}{\sum_{t=1}^{T} \widehat{\gamma}_{t,j}} \\ \pi' &= \widehat{\gamma}_{1,:} \end{aligned}$$

where $1_{\{O_t=i\}}$ is one if $O_t = i$ and zero otherwise.

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

```python
def _estimate(self, obs, delta, gamma):
    """
    Estimate better parameter values.

    Parameters
    ----------
    obs : ndarray of shape (T,)
        The observation sequence
    delta : ndarray of shape (T-1,N,N)
        The delta probability array
    gamma : ndarray of shape (T,N)
        The gamma probability array
    """
    # update self.A, self.B, self.pi in place
    pass
```

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

```python
h._estimate(obs, delta)
>>> 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]]
>>> print h.pi
[ 0.18816981  0.81183019]
```

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:

- Randomly initialize parameters $\lambda = (A, B, \pi)$
- Compute $\log \mathbb{P}(O|\,\lambda)$
- For $i = 1, 2, \ldots, K$:
  - Run forward pass
  - Run backward pass
  - Compute δ probabilities
  - Update model parameters
  - Compute $\log \mathbb{P}(O|\,\lambda)$ according to new parameters
  - If change in log probabilities is less than ϵ, break
  - Else, 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 A, B, and π using this technique:

```
>>> # assume N and M are defined
>>> A = np.random.dirichlet(np.ones(N), size=N).T
>>> B = np.random.dirichlet(np.ones(M), size=N).T
>>> pi = np.random.dirichlet(np.ones(N))
```

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 $\mathbb{P}(O|\,\lambda)$. 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 A, B, and π. Figure 7.2 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.

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

```
def fit(self, obs, A, B, pi, max_iter=100, tol=1e-3):
```

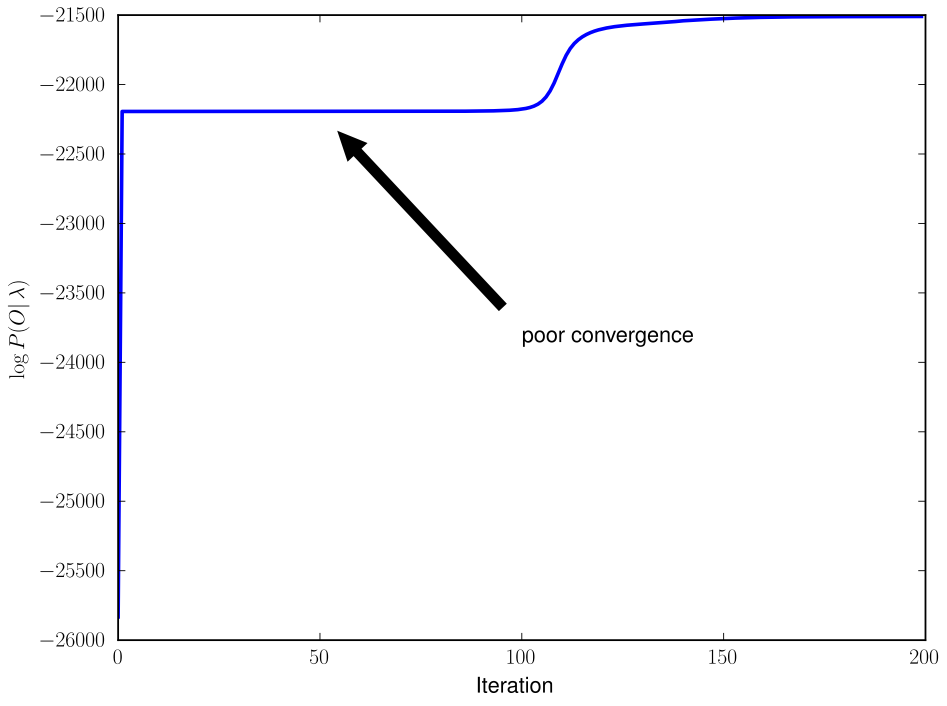

Figure 5.1: The log probabilities for a HMM trained on the Declaration of Independence data with 200 iterations. It takes over 100 iterations for the algorithm to work itself out of a poor local maximum.

```
    """
    Fit the model parameters to a given observation sequence.

    Parameters
    ----------
    obs : ndarray of shape (T,)
        Observation sequence on which to train the model.
    A : stochastic ndarray of shape (N,N)
        Initialization of state transition matrix
    B : stochastic ndarray of shape (M,N)
        Initialization of state observation matrix
    pi : stochastic ndarray of shape (N,)
        Initialization of initial state distribution
    max_iter : integer
        The maximum number of iterations to take
    tol : float
        The convergence threshold for change in log-probability
    """
    # initialize self.A, self.B, self.pi
    # 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 sample code below to accomplish this task:

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

>>> with open("declaration.txt", 'r') as f: # read in the text
>>>     dec = f.read(-1).lower() # convert to lower-case

>>> # next, remove punctuation and newline characters
>>> dec = dec.translate(string.maketrans("",""), string.punctuation+"\n\r")

>>> # create a list of the unique characters in the text
>>> char_map = list(set(dec))

>>> # map each character to its index in char_map
>>> obs = []
>>> for char in dec:
>>>     obs.append(char_map.index(char))
>>> obs = np.array(obs)
```

Problem 7. You are now ready to train a HMM using the Declaration of Independence data. Use $N = 2$ states and $M = 27$ observation values (26 lower case characters and 1 whitespace character), and run for 200 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:

```
>>> for i in xrange(len(h.B)):
>>>     print  "{0}, {1:0.4f}, {2:0.4f}".format(char_map[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. It appears that this distinction is a statistically significant aspect of much of human language.

Lab 6

Speech Recognition using CDHMMs

Lab Objective: *Understand how speech recognition via CDHMMs works, and implement a simplified speech recognition system.*

6.0.1 ▪ Continuous Density Hidden Markov Models

Some of the most powerful applications of HMMs (speech and voice recognition) result from allowing the observation space to be continuous instead of discrete, as we dealt with in the previous lab. These are called Continuous Density Hidden Markov Models (CDHMMs), and they have two standard formulations: Gaussian HMMs and Gaussian Mixture Model HMMs (GMMHMMs). In fact, the former is a special case of the latter, so we will just discuss GMMHMMs in this lab.

In order to understand GMMHMMs, we need to be familiar with a particular continuous, multivariate distribution called a *mixture of Gaussians*. A mixture of Gaussians is a distribution composed of several Gaussian (or Normal) distributions with corresponding weights. Such a distribution is parameterized by the number of mixture components M, the dimension N of the normal distributions involved, a collection of component weights $\{c_1, \dots, c_M\}$ that are nonnegative and sum to 1, and a collection of mean and covariance parameters $\{(\mu_1, \Sigma_1), \dots, (\mu_M, \Sigma_M)\}$ for each Gaussian component. To sample from a mixture of Gaussians, one first chooses the mixture component i according to the probability weights $\{c_1, \dots, c_M\}$, and then one samples from the normal distribution $\mathcal{N}(\mu_i, \Sigma_i)$. The probability density function for a mixture of Gaussians is given by

$$f(x) = \sum_{i=1}^{M} c_i N(x;\, \mu_i, \Sigma_i),$$

where $N(\cdot;\, \mu_i, \Sigma_i)$ denotes the probability density function for the normal distribution $\mathcal{N}(\mu_i, \Sigma_i)$. See Figure 5.1 for the plot of such a density curve. Note that a mixture of Gaussians with just one mixture component reduces to a simple normal distribution, and so a GMMHMM with just one mixture component is simply a Gaussian HMM.

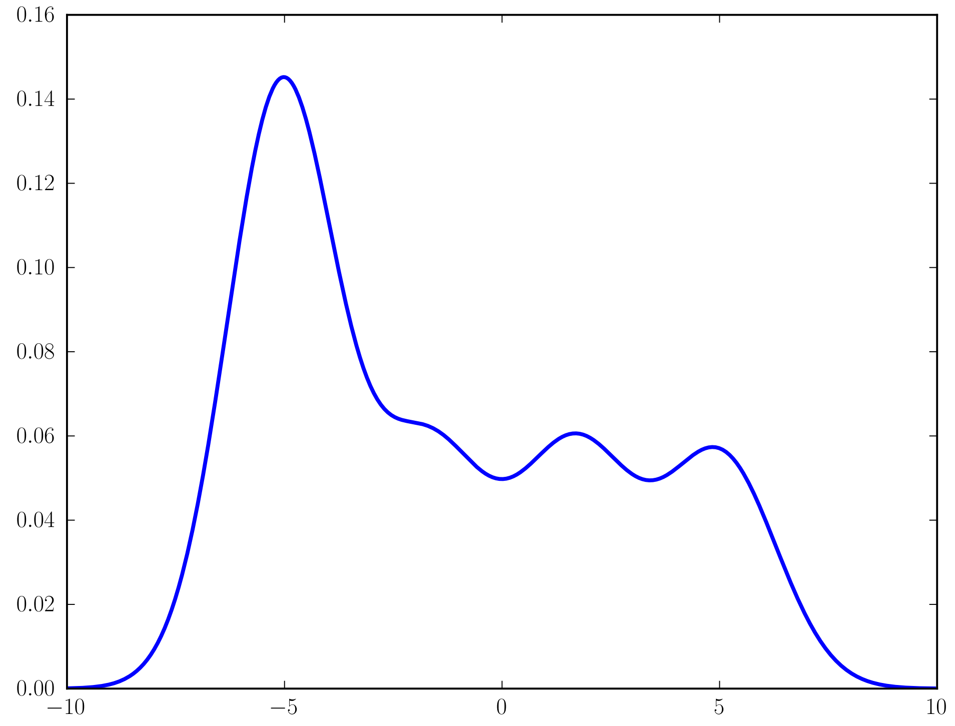

Figure 6.1: The probability density function of a mixture of Gaussians with four components.

In a GMMHMM, we seek to model a hidden state sequence $\{\mathbf{x}_1, \ldots, \mathbf{x}_T\}$ and a corresponding observation sequence $\{O_1, \ldots, O_T\}$, just as with discrete HMMs. The major difference, of course, is that each observation O_t is a real-valued vector of length K distributed according to a mixture of Gaussians with M components. The parameters for such a model include the initial state distribution π and the state transition matrix A (just as with discrete HMMs). Additionally, for each state $i = 1, \ldots, N$, we have component weights $\{c_{i,1}, \ldots, c_{i,M}\}$, component means $\{\mu_{i,1}, \ldots, \mu_{i,M}\}$, and component covariance matrices $\{\Sigma_{i,1}, \ldots, \Sigma_{i,M}\}$.

Let's define a full GMMHMM with $N = 2$ states, $K = 3$, and $M = 3$ components.

```
>>> import numpy as np
>>> A = np.array([[.65, .35], [.15, .85]])
>>> pi = np.array([.8, .2])
>>> weights = np.array([[.7, .2, .1], [.1, .5, .4]])
>>> means1 = np.array([[0., 17., -4.], [5., -12., -8.], [-16., 22., 2.]])
>>> means2 = np.array([[-5., 3., 23.], [-12., -2., 14.], [15., -32., 0.]])
>>> means = np.array([means1, means2])
>>> covars1 = np.array([5*np.eye(3), 7*np.eye(3), np.eye(3)])
>>> covars2 = np.array([10*np.eye(3), 3*np.eye(3), 4*np.eye(3)])
>>> covars = np.array([covars1, covars2])
>>> gmmhmm = [A, weights, means, covars, pi]
```

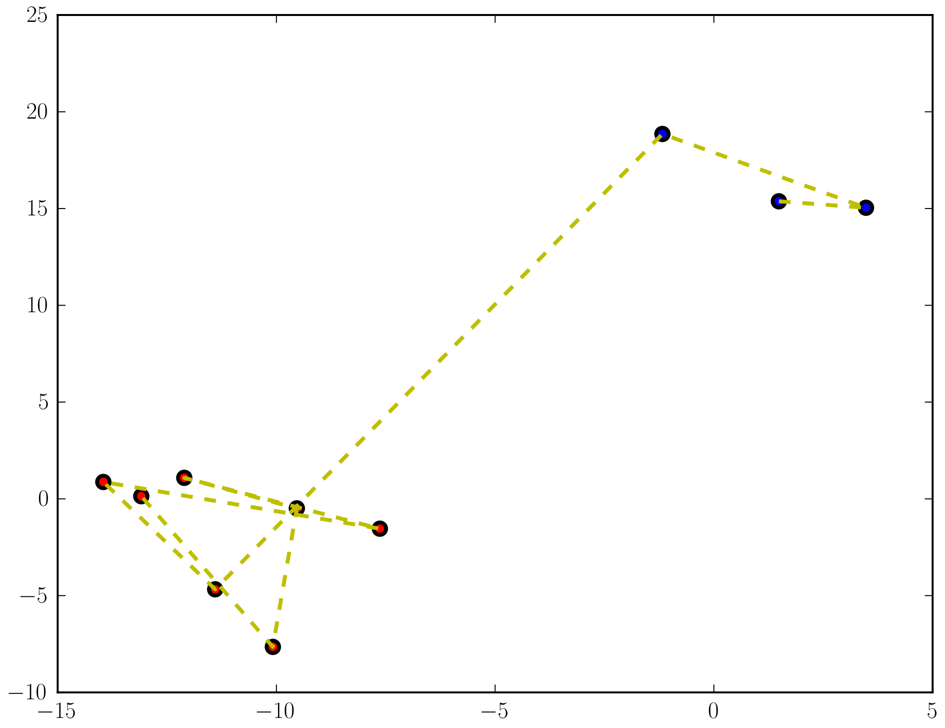

Figure 6.2: An observation sequence generated from a GMMHMM with one mixture component and two states. The observations (points in the plane) are shown as solid dots, the color indicating from which state they were generated. The connecting dotted lines indicate the sequential order of the observations.

We can draw a random sample from the GMMHMM corresponding to the second state as follows:

```
>>> sample_component = np.argmax(np.random.multinomial(1, weights[1,:]))
>>> sample = np.random.multivariate_normal(means[1, sample_component, :], covars↩
    [1, sample_component, :, :])
```

Figure 5.2 shows an observation sequence generated from a GMMHMM with one mixture component and two states.

Problem 1. Write a function which accepts a GMMHMM in the format above as well as an integer n_sim, and which simulates the GMMHMM process, generating n_sim different observations. Do so by implementing the following function declaration.

```
def sample_gmmhmm(gmmhmm, n_sim):
    """
    Simulate sampling from a GMMHMM.
    Returns
    -------
    states : ndarray of shape (n_sim,)
        The sequence of states
    obs : ndarray of shape (n_sim, K)
        The generated observations (column vectors of length K)
    """
    pass
```

The classic problems for which we normally use discrete observation HMMs can also be solved by using CDHMMs, though with continuous observations it is much more difficult to keep things numerically stable. We will not have you implement any of the three problems for CDHMMs yourself; instead, you will use a stable module we will provide for you. Note, however, that the techniques for solving these problems are still based on the forward-backward algorithm; the implementation may be trickier, but the mathematical ideas are virtually the same as those for discrete HMMs.

Speech Recognition and Hidden Markov Models

Hidden Markov Models are the basis of modern speech recognition systems. However, a fair amount of signal processing must precede the HMM stage, and there are other components of speech recognition, such as language models, that we will not address in this lab.

The basic signal processing and HMM stages of the speech recognition system that we develop in this lab can be summarized as follows: The audio to be processed is divided into small frames of approximately 30 ms. These are short enough that we can treat the signal as being constant over these inervals. We can then take this framed signal and, through a series of transformations, represent it by mel-frequency cepstral coefficients (MFCCs), keeping only the first K (say $K = 10$). Viewing these MFCCs as continuous observations in $\mathbb{R}^K$, we can train a GMMHMM on sequences of MFCCs for a given word, spoken multiple times. Doing this for several words, we have a collection of GMMHMMs, one for each word. Given a new speech signal, after framing and decomposing it into its MFCC array, we can score the signal against each GMMHMM, returning the word whose GMMHMM scored the highest.

Industrial-grade speech recognition systems do not train a GMMHMM for each word in a vocabulary (that would be ludicrous for a large vocabulary), but rather on *phonemes*, or distinct sounds. The English language has 44 phonemes, yielding 44 different GMMHMMs. As you could imagine, this greatly facilitates the problem of speech recognition. Each and every word can be represented by some combination of these 44 distinct sounds. By correctly classifying a signal by its phonemes, we can determine what word was spoken. Doing so is beyond the scope of this lab, so we will simply train GMMHMMs on five words/phrases: biology, mathematics, political science, psychology, and statistics.

Problem 2. Obtain 30 (or more) recordings for each of the words/phrases *mathematics*, *biology*, *political science*, *psychology*, and *statistics*. These audio samples should be 2 seconds in duration, recorded at a rate of 44100 samples per second, with samples stored as 16-bit signed integers in WAV format. Load the recordings into Python using `scipy.io.wavfile.read`.

If the audio files have two channels, average these channels to obtain an array of length 88200 for each sample. Extract the MFCCs from each sample using code from the file `MFCC.py`:

```
>>> import MFCC
>>> # assume sample is an array of length 88200
>>> mfccs = MFCC.extract(sample)
```

Store the MFCCs for each word in a separate list. You should have five lists, each containing 50 MFCC arrays, corresponding to each of the five words under consideration.

For a specific word, given enough distinct samples of that word (decomposed into MFCCs), we can train a GMMHMM. Recall, however, that the training procedure does not always produce a very effective model, as it can get stuck in a poor local minimum. To combat this, we will train 10 GMMHMMs for each word (using a different random initialization of the parameters each time) and keep the model with the highest log-likelihood.

For training, we will use the file we have provided called `gmmhmm.py`, as this is a stable implementation of GMMHMM algorithms. To facilitate random restarts, we need a function to provide initializations for the initial state distribution and the transition matrix.

Let `samples` be a list of arrays, where each array is the output of the MFCC extraction for a speech sample. Using a function `initialize()` that returns a random initial state distribution and (row-stochastic) transition matrix, we can train a GMMHMM with 5 states and 3 mixture components and view its log-likelihood as follows:

```
>>> import gmmhmm
>>> startprob, transmat = initialize(5)
>>> model = gmmhmm.GMMHMM(n_components=5, n_mix=3, transmat=transmat, startprob=↩
    startprob, cvtype='diag')
>>> # these values for covars_prior and var should work well for this problem
>>> model.covars_prior = 0.01
>>> model.fit(samples, init_params='mc', var=0.1)
>>> print model.logprob
```

Problem 3. Partition each list of MFCCs into a training set of 20 samples, and a test set of the remaining 10 samples.

Using the training sets, train a GMMHMM on each of the words from the previous problem with at least 10 random restarts, keeping the best model for each word (the one with the highest log-likelihood). This process may take several minutes. Since you will not want to run this more than once, you will want to save the best model for each word to disk using the `pickle` module so that you can use it later.

Given a trained model, we would like to compute the log-likelihood of a new sample. Letting `obs` be an array of MFCCs for a speech sample we do this as follows:

```
>>> score = model.score(obs)
```

We classify a new speech sample by scoring it against each of the 5 trained GMMHMMs, and returning the word corresponding to the GMMHMM with the highest score.

Problem 4. Classify the 10 test samples for each word. How does your system perform? Which words are the hardest to correctly classify? Make a dictionary containing the accuracy of the classification of your five testing sets. Specifically, the words/phrases will be the keys, and the values will be the percent accuracy.