1 Speech Recognition using CDHMMs

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

1.0.1 Continuous Density Hidden Markov Models

Some of the most powerful applications of Hidden Markov Models, speech and voice recognition, result from allowing the observation space to be continuous instead of discrete. 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 1.1 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.

In a GMMHMM, we seek to model a hidden state sequence $\{\mathbf{x}_1, \dots, \mathbf{x}_T\}$ and a corresponding observation sequence $\{O_1, \dots, O_T\}$, 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

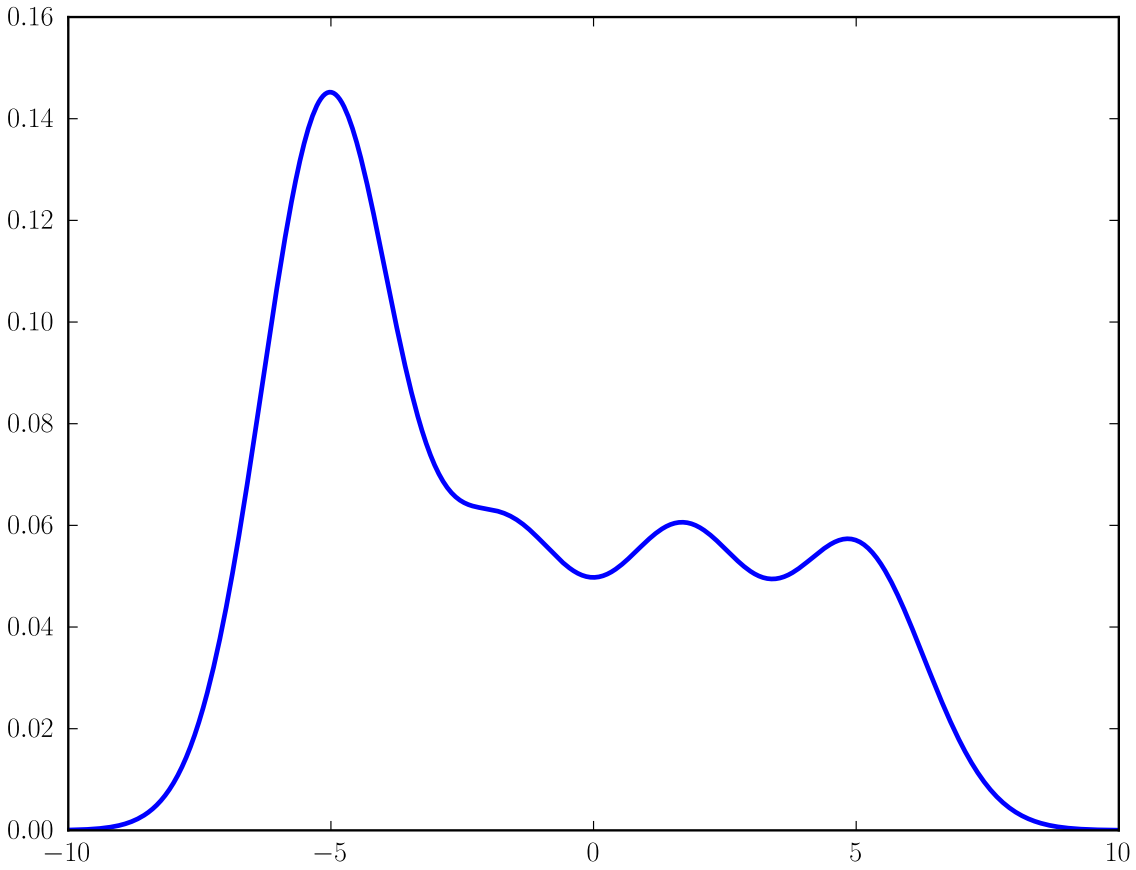

Figure 1.1: The probability density function of a mixture of Gaussians with four components.

$i = 1, \ldots, N$, 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]])   # state transition matrix
>>> pi = np.array([.8, .2])   # initial state distribution
>>> 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]
```

Once we have a GMMHMM, we can randomly choose the first state based on the initial state distribution π. As explained above, to sample from a GMMHMM, we draw a sample from one of the Gaussians $\mathcal{N}(\mu_i, \Sigma_i)$, with component i chosen according to the probably weights $\{c_1, \ldots, c_M\}$. We

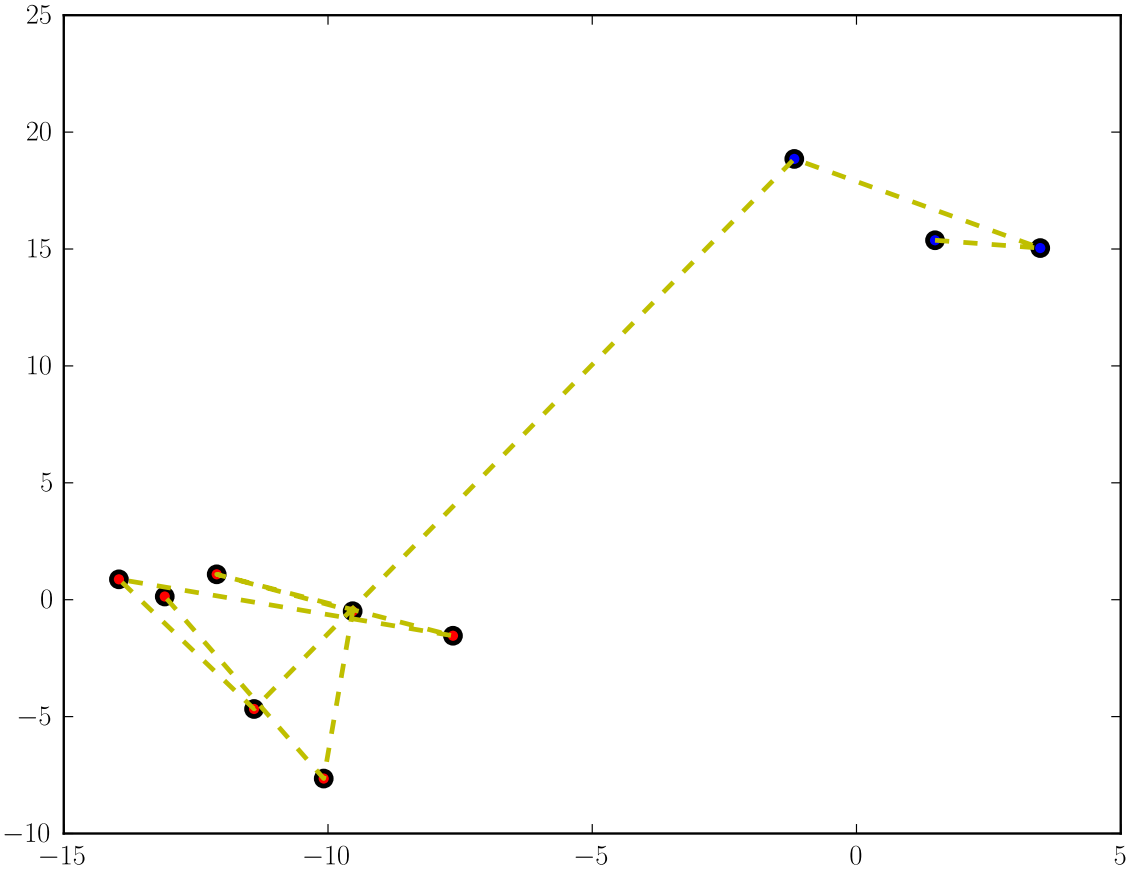

Figure 1.2: An observation sequence generated from a GMMHMM with one mixture component and two states. The observations (points in the plane) are shown as solid dots, the color indicating from which state they were generated. The connecting dotted lines indicate the sequential order of the observations.

can draw a random sample from the GMMHMM corresponding to the second state. Once we have the sample, we use the transition matrix A to determine the second state.

```
# choose initial state
>>> state = np.argmax(np.random.multinomial(1, pi))

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

Figure 1.2 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
```

Test your function by running it on the gmmhmm given in the example, with $n\_sim = 4$. Print out the each state and its corresponding observations.

Note, K is the same K that defines the gmmhmm and is identified through the gmmhmm, so you don't need to set it.

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. Samples.zip contains 31 recordings for each of the words/phrases mathematics, biology, political science, psychology, and statistics. These audio samples are 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*.

Extract the MFCCs from each sample using code from the file MFCC.py. Store the MFCCs for each word in a separate list. You should have five lists, each containing 31 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 as hmm
>>> startprob, transmat = initialize(5)
>>> model = hmm.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 (reinitializing and creating a new model). Use $n\_components = 5$ and $initialize(5)$. Keep the best model for each word (the one with the highest log-likelihood). This process may take up to a couple of hours. 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.