Java HMMPak v1.2
User Manual

Troy L. McDaniel
Center for Cognitive Ubiquitous Computing (CUbiC)
Arizona State University

Table of Contents

I. Introduction

Hidden Markov models are truly an indispensable tool when attempting to recognize temporal or modeling sequences. All one has to do is just look at the amazing progress speech recognition has made and you'll see the power of HMMs. But now they're finding uses in many other areas such as gesture recognition, biometrics, computer vision, computational biology, and many other application areas. For this reason, an easy-to-use but powerful HMM package is crucial. Hopefully, HMMPak will meet these requirements.

Before continuing with this manual, a solid understanding of hidden Markov models is crucial towards the reader's ability to utilize all this package has to offer. For an excellent tutorial on HMMs, please see [1]. And for a more advance tutorial, see [2].

II. Getting Started

HMMPak can be downloaded from http://www.public.asu.edu/~tmcdani/hmm.htm. Both a Java and a C++ implementation exist. This manual is for the Java v1.2 implementation. Three directories can be found within the WinZip file for HMMPak v1.2. *HMMPak* includes the actual HMMPak Java package, *Demo* includes a demo file for the package, and *Manual* includes a copy of this manual.

If you already know how to set up Java packages, feel free to skip this paragraph. The folder *HMMPak* may be copied to the location where your packages are kept. For example, on my system all my favorite Java packages are at C:\packages. Next, set your CLASSPATH to point to this location if you haven't done so already. This allows Java to find the HMM class. If you need more information regarding packages and commands to check and set the CLASSPATH, an excellent tutorial can be found at http://jarticles.com/package/package_eng.html.

Once HMMPak has been successfully installed, any Java program can use it. To use HMMPak, simply have your program import it, as shown below.

```
import HMMPak.*;
```

Obviously the first step towards using this package is creating an HMM. This task can be done in many different ways and thus deserves some discussion. In the next section, we'll cover the manual creation of HMMs.

III. Creating an HMM

Before we begin looking at all the things hidden Markov models are capable of, it makes since to begin with the simple task of creating one. Actually, this can be done several ways. We can create an HMM on our own, manually filling in probabilities, or we may decide to load one from a file. An HMM can also be constructed from training data. All these methods will eventually be discussed at some point in this manual, but for right now we'll look at the first case, which is creating an HMM "by hand." Loading an HMM from a file can be found in Section XI, and learning an HMM from training data can be found in Section IX.

If we already know the states and probabilities of an HMM, or would just like a simple, random HMM to demonstrate certain properties, then creating an HMM manually is ideal. To create, for example, an HMM with three states and three distinct observation symbols, we can call the constructor as shown below. The first parameter is the number of states and the second is the size of the output vocabulary, both three in this example.

```
HMM myHmm = new HMM(3, 3);
```

An HMM with initial, transition, and bias probabilities equal to zero doesn't do us much good, so let's manually fill in these values (shown below). Again, this is great if we know the probabilities before hand or if "made up" values are just fine for whatever reason. Our states are 0, 1, and 2, and lets call our observation symbols Red, Green, and Blue. Each state has a symbol distribution, and each symbol has an index into these distributions. And therefore, indices are used to refer to observation symbols. Here, Red, Green, and Blue have the indices 0, 1, and 2, respectively.

```
// initial probabilities
myHmm.pi[0] = 0.5;              // state 0
myHmm.pi[1] = 0.25;             // state 1
myHmm.pi[2] = 0.25;             // state 2
// transition probabilities
myHmm.a[0][0] = 0.2;            // state 0 to 0
myHmm.a[0][1] = 0.6;            // state 0 to 1
myHmm.a[0][2] = 0.2;            // state 0 to 2
myHmm.a[1][0] = 0.25;           // state 1 to 0
myHmm.a[1][1] = 0.5;            // state 1 to 1
myHmm.a[1][2] = 0.25;           // state 1 to 2
myHmm.a[2][0] = 0.35;           // state 2 to 0
myHmm.a[2][1] = 0.15;           // state 2 to 1
myHmm.a[2][2] = 0.5;            // state 2 to 2
// bias probabilities
myHmm.b[0][0] = 0.25;           // Red at state 0
myHmm.b[0][1] = 0.25;           // Red at state 1
myHmm.b[0][2] = 0.5;            // Red at state 2
myHmm.b[1][0] = 0.2;            // Green at state 0
myHmm.b[1][1] = 0.3;            // Green at state 1
myHmm.b[1][2] = 0.5;            // Green at state 2
myHmm.b[2][0] = 0.45;           // Blue at state 0
myHmm.b[2][1] = 0.05;           // Blue at state 1
myHmm.b[2][2] = 0.5;            // Blue at state 2
```

Notice that certain sets of probabilities must sum to one. For example, the set of initial probabilities should sum to one, among other sets. Your HMM may not be very accurate if this property is not satisfied, but the algorithms will still run.

If you've looked through the API for HMMPak, you've probably seen an overloaded constructor with a third input parameter—namely *dimensions*. This constructor is used during training since we must know the number of dimensions that our symbols contain, plus the actual output vocabulary. This information is not required using the first approach and may therefore be ignored. Here, we've identified an output vocabulary of Red, Green, and Blue, but again this isn't necessary when the first approach is used.

Next, we'll discuss in detail all of the member variables contained within the HMM class. You've already seen a few in this section. We'll cover these again but in more detail plus touch on the rest.

IV. Getting to Know Your Member Variables

From the previous section, you've already been introduced to four important member variables: *pi*, *a*, *b,* and *dimensions*. Member variable *pi* is the set of initial probabilities. It's an array of doubles with a length equal to the number of states, starting at state 0. This member variable, as well as any other public member variable, can be accessed using Java's "dot" operator, as shown in Section III. Member variable *a* is the set of transition probabilities. It's a matrix of doubles with the rows and columns being states.

Member variable *b* is the set of bias probabilities. It's a matrix of doubles with the rows being states and the columns being symbols. For example, some observation symbol may be assigned to index six within this matrix. As we move down through the rows, we visit each state's symbol distribution. The last member variable you were introduced to in Section III was *dimensions*. If you recall, this is the number of dimensions of our observation symbols, and may not be used in some cases depending on how an HMM was created, as described in Section III.

Member variable *numStates*, as the name implies, is the number of states an HMM contains. The size of our output vocabulary, i.e., the number of observation symbols, is member variable *sigmaSize*. And lastly, member variable *V* is the actual output vocabulary. It's a matrix of doubles, with columns being dimensions and rows being indices matching those of the symbol distributions. Similar to *dimensions*, whether or not this matrix is used depends on how an HMM is created.

Of course the methods of this HMM class are also important, and you'll eventually learn about each one as you go through this manual. Next, we will learn how to generate an observation sequence from an HMM. Not only does this tell us quite a bit of information about the structure of an HMM, it's also used for determining the distance between HMMs, discussed in Section XII.

V. Generating an Observation Sequence

Once an HMM has been created, we can use it to generate an observation sequence. The actual sequence returned is in fact based on the probabilities that make up an HMM. The sequence returned is actually a list of indices. Each index in this list corresponds to an observation symbol. Recall that each state has a symbol distribution. Symbol *X* will always have the same position, i.e., index, within each distribution.

The code below shows how to use the *observationGeneration* method. This particular call generates a sequence of length 5. The return value is always an array of integers—as you know, these values are indices.

```
int[] o;
o = myHmm.observationGeneration(5);
```

Next, we'll learn how to actually determine the probability of a certain sequence of observation symbols occurring on a hidden Markov model. This can be done using the Forward-Backward procedure, described next.

VI. Forward-Backward Procedure

Say we would like to know the probability of a certain sequence of observations symbols occurring on an HMM. Finding this type of probability is simple thanks to the Forward algorithm, or the Backward algorithm, whichever you prefer.

Say we want to know the probability of the sequence Red, Green, Blue occurring on the hidden Markov model created in Section III. Finding the probability of this sequence is accomplished in three steps. The first step is creating an integer array containing the sequence in question in the form of indices, where each index correspond to an observation symbol in the symbol distributions. (If you created an HMM using real training data, you may not know the index of each symbol. The HMM class contains a method to convert real symbols into their respective indices. You can learn about this method in Section X.) The code that completes the first part is shown below.

```
int[] o;
o = new int[3];
o[0] = 0;       // Red
o[1] = 1;       // Green
o[2] = 2;       // Blue
```

The second step is the actual method call. The methods *forwardProc* and *backwardProc* do not return a single probability but instead the entire Forward and Backward variable tables, respectively. Code for both types of calls is shown next.

```
double[][] alpha;
double[][] beta;
alpha = myHmm.forwardProc(o);
beta = myHmm.backwardProc(o);
```

The final step is to obtain the desired probability from the alpha, or beta, table. Let's look at how it's done with the alpha table first. Remember that we don't care what state we end at, so we need to sum up the last column of the alpha table to get the desired probability, as shown below.

```
double prob;
for(int i=0; i < 3; i++)
        prob += alpha[i][2];
```

Recall that the rows of the alpha table are states and the columns are time, whose total length is equal to the length of the observation sequence since one symbol is generated per unit time. And therefore, the alpha table is a 3x3 table in this example. Now, let’s look at how it’s done using the beta table. Remember that we don’t care which state we begin at, but a little bit more than just summing up the first column has to be done due to the nature of this variable, as shown below.

```
double prob;
beta = myHmm.backwardProc(o);
for(int i=0; i < 3; i++)
        prob += myHmm.pi[i]*myHmm.b[i][o[0]]*beta[i][0];
```

The beta table for this example, similar to the alpha table, is a 3x3 table where the rows are states and the columns represent time. We’ve now seen how to calculate the probability of an observation sequence on a hidden Markov model using either the Forward or Backward algorithm. (As mentioned earlier, you may use either algorithm; the final probability will be the same no matter which algorithm is used.) Next, we’ll learn how to use an algorithm that can find a state sequence in an HMM such that this state sequence gives us the highest probability of the occurrence of a given observation sequence.

VII. Viterbi Algorithm

At some point it may be useful to determine the state sequence for an observation sequence that yields the highest probability of that particular observation sequence occurring. The Viterbi algorithm makes them seemingly difficult task quite easy. The running example seen in the previous sections will again be used here to demonstrate how to use the *viterbi* method.

First, we need to decide what observation sequence we’re interested in, and set up an array holding this sequence. This array will be an integer array consisting of indices. Setting up an array for the observation sequence Red (index 0) followed by Blue (index 2) is shown below.

```
int[] o;
o = new int[2];
o[0] = 0;       // Red
o[1] = 2;       // Blue
```

Now, we can just send this array to the *viterbi* method to determine the state sequence. However, the output of *viterbi* is a little tricky and deserves an explanation. What viterbi returns is a 2xT matrix, where T is the length of the input observation sequence. Entry (0, 0) represents the minimized weight, which can be turned into the state optimized probability. The second row, i.e., entries (1, 0) to (1, T-1), holds the actual state sequence. Shown below is code extracting these values and printing them after the method call.

```
double[][] answer;
answer = hmm.viterbi(o);
// recall that if w is our minimized weight, state optimized probability = e^(-w)
double stateOptProb = Math.exp(-1*answer[0][0]);
System.out.println("The state optimized probability is " + stateOptProb);
System.out.print("Optimal path is ");
        for(int i = 0; i < o.length; i++) {
                System.out.print((int)answer[1][i] + "");
                if(i != o.length-1)
                        System.out.print(", ");
        }
```

This concludes our discussion on how to use the *viterbi* function. Next, we'll discuss the very important task of training HMMs. The two methods are the Baum-Welch algorithm and K-Means Learner. The latter learns an HMM from scratch using training data, and therefore, we must also cover the required file format for training data plus converting real symbols into their respective indices. For this reason, we've divided training into two sections: Section VIII covers the Baum-Welch algorithm and Section IX covers learning an HMM from training data.

VIII. Baum-Welch Algorithm

The Baum-Welch algorithm is the first training algorithm we'll look at. Baum-Welch increases the probability of a certain observation sequence occurring on an HMM. Using Baum-Welch requires that an HMM already exists. Using the running example seen throughout the previous sections, let us train *myHmm* using the observation sequence Red, Green, Blue, which we know corresponds to indices 0, 1, and 2 respectively. Below is the code to initialize an integer array of indices and call the *baumWelch* function.

```
int[] o;
o = new int[3];
o[0] = 0;
o[1] = 1;
o[2] = 2;
myHmm.baumWelch(o, 9);// number of steps = 9
```

Now, the probability of Red, Green, and Blue on this HMM has been increased, with the best case being a maximized probability. The number of steps determines the number of re-estimations. At some point, the algorithm will converge and the probability will be maximized. Different values for the number of steps should be tried to determine where maximization occurs.

Next, we'll learn how to train an HMM from scratch using training data. We'll learn about the appropriate file format for training data and of course K-Means Learner.

IX. Learning an HMM from Training Data

Before using *kmeansLearner* to cluster training data and calculate all the necessary probabilities, we'll need to learn about the training data file format. Once this format is known, programs can be written that export data using this format, and so all kinds of data may be fed directly into the K-Means Learner algorithm. Section VIII.A covers the file format, while Section VIII.B covers how to use the *kmeansLearner* function.

VIII.A. Training Data File Format

The file format for storing training data is straightforward. We'll take a look at the header first, and then the data portion. Below is an example training data file that we'll be working with.

```
3 3 4
90.0 45.0 30.0 135.0 90.0 45.0 179.8 135.0 89.8
90.5 45.2 29.8 135.2 90.3 44.9 180.3 135.2 89.7
90.3 44.9 29.6 134.8 90.1 45.2 180.4 135.1 90.0
89.9 45.0 30.3 135.0 89.9 45.1 180.0 135.0 90.0
```

The first element in the header is the number of dimensions. The next entry is a space followed by the length of the observation sequences. Next, we have another space followed by the number of observation sequences in the training data. From this example, these values are 3, 3, and 4, respectively.

The data portion, as the name implies, contains the actual observation sequences that make up the training data. The data portion should be in the form of a matrix with rows being observation sequences and columns being symbols. Before we examine a single row in detail, white space deserves some discussion. There should be a carriage return between the header and the data portion, as shown above. Also, there should be a space between each symbol in a single row. And at the end of each observation sequence, including the last, there should be a carriage return.

Now, let's examine the first row. Since the length of each observation sequence is 3, there are should be three symbols in this row, and in fact there is. Even though it looks like there are nine symbols, remember that each symbol has three dimensions. Therefore, the first three values of the first row, namely (90.0 45.0 30.0), is one observation symbol with three dimensions.

With the details of the training data file format out of the way, the next subsection will cover K-Means Learner, which can learn an HMM from a training data file of the format just discussed.

VIII.B. K-Means Learner

The K-Means Learner algorithm can take as input a training data file and from it, learn an HMM. The code below shows how to call the *kmeansLearner* method.

```
HMM newHmm = HMM.kmeansLearner("training.txt", 3, 5);
```

The first parameter is the name of the training data file. The second parameter is the number of desired states, which is three in this example. And the third parameter is number of desired iterations, which is five in this example. Basically, the larger this value is, the more accurate your clusters will be. But when the change of the means is below a threshold, the algorithm will stop anyways.

We now have an HMM, namely *newHmm*, which has been learned from real training data. Whether that data is speech data or gesture data, it doesn't matter. Any kind of training data can be used as long as it obeys the proper file format.

However, one piece of the puzzle is missing. We don't know the indices of our observation symbols, which are required if we want to use a method such as *forwardProc*. There's a method, namely *convert*, which handles this task—specifically the task of taking symbols and converting them into their respective indices. Next, Section X discusses how to turn real observation symbols into indices.

X. Transforming Real Symbols into Indices

If an HMM has been trained using real training data, the indices of every symbol in the training data, which could number in thousands, may not be known. Therefore, we need a way to transform real observation symbols into indices. The *convert* method handles this task elegantly.

Let's continue the example from Section IX. Say we want to determine the probability of the occurrence of observation sequence (90, 45, 30), (135, 90, 45), and (180, 135, 90). The first task is to set up a double array containing this actual sequence, as shown below. The double array *seq* is 3x3 since the observation sequence is of length three and each symbol consists of three dimensions, respectively.

```
double[][] seq = new double[3][3];
seq[0][0] = 90.0;  seq[0][1] =  45.0;  seq[0][2] =  30.0;  // symbol #1
seq[1][0] = 135.0; seq[1][1] =  90.0;  seq[1][2] =  45.0;  // symbol #2
seq[2][0] = 180.0; seq[2][1] =  135.0; seq[2][2] =  90.0;// symbol #3
```

Once our observation sequence is set up, *convert* takes it as an input parameter to obtain an integer array containing the respective indices for each symbol. Let's look at an actual call to *forwardProc* using the HMM trained in Section XIII.B, shown below.

```
double[][] alpha;
alpha = newHmm.forwardProc(newHmm.convert(seq, 3, 3));
```

The first parameter of convert is our observation sequence created in the previous step. The second parameter is the number of dimensions each symbol consists of. And lastly, the third parameter is the length of the observation sequence.

This completes converting real symbols into indices. Next, we'll take a look at how to write an HMM to a file and how to read an HMM from a file. An HMM file also has a specific file format, and this file format will be discussed as well.

XI. Reading and Writing

Reading and writing HMMs are very useful tasks. For example, we may wish to save an HMM for later use or perhaps use an HMM belonging to someone else. We'll cover both reading and writing in this section.

Say we have an HMM called *myHmm* that we wish to write to a file, say *hmm.txt*. The command to do this is shown below.

```
myHmm.write("hmm.txt");
```

Reading an HMM from a file is just as easy. Below is the code that accomplishes this task. The file we're reading from is *hmm.txt*. Lastly, note that the load method is static, and thus it's called using the Class name rather than through an object.

```
HMM myHmm;
myHmm = HMM.load("hmm.txt");
```

Before continuing to the next section, the file format of an HMM file deserves some discussion. Below is an example HMM file. The HMM within this file is the Red, Green, Blue HMM example used throughout this manual.

```
3 3 -1
0.5 0.25 0.25
0.2 0.6 0.2 0.25 0.5 0.25 0.35 0.15 0.5
0.25 0.25 0.5 0.2 0.3 0.5 0.45 0.05 0.5
```

Let's look at the header first. The values, in order, are the number of states, the size of the output vocabulary, and the number of dimensions. The number of dimensions is set to -1 since it's not used in this example. The second row includes the initial probabilities. The third row includes the transition probabilities. And finally, the fourth row, or more appropriately "block" since it will most likely occupy more than one row, includes the bias probabilities.

This was a simple example; usually you'll care about the number of dimensions and the actual output vocabulary. If we did, the number of dimensions would be greater than zero. Also, the actual output vocabulary would be included in the HMM file, immediately following the bias probabilities.

In the following section, will learn how to calculate the distance between two HMMs. Distance helps us determine how similar, or dissimilar, two hidden Markov models are.

XII. Distance Between HMMs

Calculating the distance between two HMMs can be accomplished using the *hmmDistance* method. Say, for example, we have two HMMs, namely *hmm1* and *hmm2*. The code below shows how to calculate the distance between these two HMMs.

```
double distance;
distance =  hmm1.hmmDistance(hmm2);
```

For the results of this method to make any sense, your HMMs should obviously have the same output vocabulary and output vocabulary size. Furthermore, this method uses indices to check distance and thus symbols should have the same index within each model’s symbol distribution. If output vocabulary sizes differ, a value of -1 is returned.

Next, we’ll learn the simple task of printing HMM values on the console. Although simple, it’s often quite useful for debugging and visualization.

XIII. Printing

Although a very simple task, displaying the probabilities that make up your HMM, including initial, transition, and bias probabilities, is very useful. The code below shows how to print the probabilities of an HMM on the console. Probabilities are displayed in the order of initial, transition, and bias probabilities.

```
myHmm.print();
```

Displayed on the console will be all of the probabilities in an easy-to-read format. This section concludes a thorough discussion of the functionality of HMMPak.

XIV. Bibliography

[1] R. Dugad and U. B. Desai, “A Tutorial on Hidden Markov Models,” *Published Online, May 1996.* See http://vision.ai.uiuc.edu/dugad/.

[2] L. R. Rabiner and B. H. Juang, “An introduction to hidden Markov models,” *IEEE ASSP Mag*., pp. 4-16, Jun. 1986.