Jahmm v0.6.1

Jean-Marc François

April 20, 2006

Contents

Jahmm: v0.6.1 User Guide

This document explains how the GPL *Jahmm* library can be used. It's a short because the library itself is quite short and simple. The source code is commented extensively and has been written so the details that are not included in this document should be easily found directly in the source code. The homepage of this library is `http://www.run.montefiore.ulg.ac.be/~francois/software/jahmm/`.

Chapter 1

Main classes

1.1 Hidden Markov Models [1]

The cornerstone class defined in Jahmm is obviously Hmm It holds the basic components of a HMM: state-to-state transition probabilities, observation probability distributions, number of states,... There is no such thing as a "State" class, so each state is simply designated by an integer index; a sequence of states is thus implemented as an array of integers [2].

1.2 Observations

The class Observation implements classes which define an observation.

The ObservationInteger class holds integer observations. To be useful, each kind of observation should have at least one observation probability distribution function (*e.g.* OpdfInteger in this case).

The ObservationDiscrete class holds observations whose values are taken out of a finite set; it matches the distribution OpdfDiscrete.

The ObservationReal class holds real observations (implemented as a double). It can be used together with the class OpdfGaussian (resp. OpdfGaussianMixture), which implements a Gaussian (resp. Gaussian mixture) distribution.

The ObservationVector class holds vector of reals (implemented as doubles). It can be used together with the class OpdfMultiGaussian, which implements a multivariate gaussian distribution[3].

A sequence of observations is simply implemented as a Vector of Observations. A set of observation sequences is implemented using a Vector of such Vectors.

The probability of an observation sequence given a HMM can be computed using the HMM class's probability and lnProbability methods (one can also directly instan-

[1] This document is not a tutorial on HMMs. Please refer to chapter 8 for a few reference papers.
[2] See Section 7.1 to see how to build a simple HMM.
[3] See Section 7.2 to see how to build those distributions.

ciate the ForwardBackwardCalculator class or its scaled version, ForwardBackward-ScaledCalculator, so as to avoid underflows with long sequences).

Chapter 2

Learning

2.1 k-Means clustering algorithm

This algorithm first packs the observations in *clusters* using the general purpose clustering class KMeansCalculator. This class takes elements as inputs and outputs clusters made of those elements. The elements must implement the CentroidFactory interface, meaning that they must be convertible to a Centroid.

A centroid is sort of a mean value of a set of CentroidFactory elements. For example, if the set of elements is the integers values {2,3}, than their centroid could be their arithmetic mean, 2.5.

To use the k-Means learning algorithm, one first instanciates a KMeansLearner object. Remember that the set of observation sequences given to the constructor must also be a set of CentroidFactory objects.

Once this is done, each call to the iterate() function returns a better approximation of a matching HMM. Notice that the first call to this function returns a HMM that does not depend on the temporal dependance of observations (it could be a good starting point for the Baum-Welch algorithm).

The learn() method call the iterate() method until a fix point is reached (*i.e.* the clusters doesn't change anymore).

2.2 Baum-Welch algorithm

One must first instanciate a BaumWelchLearner object in order to use this algorithm.

Unlike the k-Means case, there is no need to use CentroidCompatible observations.

Once it's been done, one can call the iterate(hmm, seqs) method in order to get a HMM that fits the observation sequences seqs better than its hmm argument.

The learn(hmm, seqs) method applies iterate(hmm, seqs) a certain number of times (see the BaumWelchLearner source code), taking a first approximated HMM as an argument. This first guess is very important since this algorithm only finds locally minimum values of its fitting function.

A scaled version of this class, BaumWelchScaledLearner, implements a scaling algorithm so as to avoid underflows when the learning is based on long observation sequences.

Chapter 3

Input/Output

3.1 Observation sequences

Observation sequences and sets of observation sequences can be read from a file. This file must have the following format:

```
obs11 ; obs12 ; obs13 ;
obs21 ; obs22 ; obs23 ; obs24;
```

Tab and spaces are not significant. C-like comments are allowed. Each line holds exactly one sequence and can be CR/LF or CR terminated. Thus, in the above example, the obs1x observations build the first sequence [1]

The observations are read using an ObservationReader class. For example, the format of an ObservationInteger is a simple number (a string made of digits only).

Supported observation types are integer, reals and real vectors.

It is suggested to add the extension ".seq" to *Jahmm* observation sequence files.

3.2 HMMs

HMMs can also be encoded as a textual file. Its syntax is quite straightforward, so it will be explained by means of an example.

```
Hmm v1.0

NbStates 5

State
Pi 0.1
A 0.1 0.2 0.3 0.4 0.0
IntegerOPDF [0.4 0.6 ]
```

[1] See chapter 8 for an example of data file reading.

```

State
Pi 0.3
A 0.2 0.2 0.2 0.2 0.2
IntegerOPDF [0.5 0.5 ]

State
Pi 0.2
A 0.2 0.2 0.2 0.2 0.2
IntegerOPDF [0.5 0.5 ]

State
Pi 0.2
A 0.2 0.2 0.2 0.2 0.2
IntegerOPDF [0.5 0.5 ]

State
Pi 0.2
A 0.2 0.2 0.2 0.2 0.2
IntegerOPDF [0.5 0.5 ]
```

All the white space characters (tab, space, new line,...) are treated the same, and several white spaces are equivalent to a signle one. So, for example, the whole file could be encoded as a single line.

The first line just gives the file syntax's version number. The NbStates keyword of course gives the number of states of the HMM described.

After that comes a list of n state descriptions, where n is the HMM's number of states. This list is ordered, so the i-th element of the list matches the i-th state. Each such description is composed of:

- *The State keyword*.
- *The probability that this state is an initial state.* This probability is preceded bye the Pi keyword.
- *The state-to-state transition probabilities.* Those probabilities are written as the A keyword followed by n an ordered list of probabilities. If the state currently described is state number *i*, then the *j* probability of the list is that of going from state *i* to state *j*.
- *A description of the observation distribution function.*. The exact syntax depends on the type of distribution; it can be found in the javadoc of the relevant class (for example, jahmm.io.OpdfGaussianReader for gaussian distributions).

  The example given above describes integer distributions. It begins with the IntegerOPDF keyword followed by an ordered list of probabilities between brackets. The first probability is related to the integer '0', the second to '1', etc... In the example above, the probability that the first state emits '1' is equal to 0.6.

Chapter 4

Various algorithms

One can find the (probabilistic) distance between two HMMs (a distance similar to the Kullback-Leibler measure) using the KullbackLeiblerDistanceCalculator class [1].

This class generates n long observations sequences of length l using one HMM and finds their probabilities using the other. Parameters n and l can be set using the setNbSequences and setSequencesLength methods.

Pay attention to the fact that this distance measure is not symetric; see the code documentation for more details.

[1] See chapter 8 to find the reference of a paper explaing this distance definition.

Chapter 5

Drawing

A HMM can be converted to a dot file using a HmmDrawerDot class.

If no class has been designed for the particular type of observation one is using, than the generic GenericHmmDrawerDot class can be used.

A HMM can be converted using a command such as:

```
(new GenericHmmDrawerDot()).write(hmm, "hmm.dot");
```

The second argument is the filename (possibly with a path).

This file can be converted to a PostScript file using the graphViz tool:

```
dot -Tps hmm.dot -o hmm.ps
```

Chapter 6

Command-Line Interface

6.1 In a nutshell

A command-line interface (CLI) makes it possible to use most of the library's algorithms without writing any line of code.

In a nutshell, it means that one can create a 5 states Hidden Markov Model that deals with integer observations by simply typing:

```
jahmm-cli create -opdf integer -r 10 -n 5 -o test.hmm
```

This creates a file test.hmm containing the description of a HMM with 5 states (-n argument). Each state is associated with a distribution over integer 0...9 (10 different values, -r argument).

The line above launches a program called jamm-cli, which is supposed to be the command-line program of the Jahmm library. The exact way to launch a Java program is plateform-dependant, but generally amounts to type

```
java be.ac.ulg.montefiore.run.jahmm.apps.cli.Cli
```

Typing this is quite inconvenient, but can generally be simplified. Under Unix, bash users could add a line such as

```
alias jahmm-cli='java be.ac.ulg.montefiore.run.jahmm.apps.cli ↩
    .Cli'
```

to their .bashrc file. Launching the CLI is then as easy as typing jahmm-cli.

6.2 CLI Options

The CLI must be called with a list of arguments that describes what you expect the program to do. If it is called with the single help argument, it prints a short help. When a filename must be given, it can be relative to the current directory or absolute; the special "-" filename opens the standard input (or output) for reading (writing).

The first argument to be given is one of

- create (creates a new HMM description file, see Section 6.2.2).
- print (prints a HMM in a human readable way, see Section 6.2.3).
- learn-kmeans (applies the k-Means algorithm, see Section 6.2.4).
- learn-bw (applies the Baum-Welch algorithm, see Section 6.2.5).
- generate (generates an observation sequence given a HMM, see Section 6.2.6).
- distance-kl (computes the distance between two HMMs, see Section 6.2.7).

Each of those options takes additional arguments.

6.2.1 Observation distributions

Most of the time, you will be required to give a description of the observation distributions associated with the states of a HMM. This can be done *via* the -opdf argument; it must be followed by one of: integer (distribution over a finite set of integers), gaussian (normal distribution), gaussian_mixture (mixture of gaussians), multi_gaussian (normal distribution over real vectors).

When using integer distributions, the -r argument is required; it specifies over which range of integers the distribution is defined. For example, -opdf integer -r 4 defines a distribution over the integers 0, 1, 2, 3..

When using mixture of gaussians, the -ng argument is required; it specifies the number of gaussians this distribution is made of. For example, -opdf gaussian_mixture -ng 3 defines a distribution made of 3 gaussians.

When using normal distributions over real vectors, the -d argument is required; it specifies the dimension of the vectors involved. For example, -opdf multi_gaussian -d 3 defines a distribution whose covariance is a 3x3 matrix.

6.2.2 create

This argument is used to create a new HMM and to write it to a file.

This option requires a distribution specification, the required number of states (specified using the -n parameter) and an output file (specified using the -o parameter).

For example:

```
jahmm-cli create -opdf integer -r 4 -n 3 -o test.hmm
```

The HMM is created with uniform parameters (all the states have the same probability of being initial ad the state transition probability matrix is uniform). As the file created is textual, it can be easily edited by hand.

6.2.3 print

This argument is used to print a HMM in a human-readable way.

This option only requires to provide an input file using the -i argument.

For example:

```
jahmm-cli print -i test.hmm
```

6.2.4 learn-kmeans

This argument is used to apply the k-Means learning algorithm. A learning algorithm finds the parameters of a HMM that best match a set of observation sequences.

This option requires a distribution specification, the resulting HMM's number of states, an input observation sequences file (parameter -is) and an output file (parameter -o).

For example:

```
jahmm-cli learn-kmeans -opdf integer -r 10 -n 2 -is  ↩
   testInteger.seq -o test.hmm
```

6.2.5 learn-bw

This argument is used to apply the Baum-Welch learning algorithm. A learning algorithm finds the parameters of a HMM that best match a set of observation sequences. The Baum-Welch algorithm requires a initial HMM whose parameters are improved according to the provided sequences; it only finds a local optimum.

This option requires a distribution specification, the resulting HMM's number of states, an input observation sequences file (parameter -is), a first HMM estimate (parameter -i) and an output file (parameter -o). The number of iterations of the algorithm can be set using the -ni argument.

For example:

```
jahmm-cli learn-bw -opdf integer -r 4 -is testInteger.seq -ni ↩
    5 -i initial.hmm -o test.hmm
```

6.2.6 generate

This argument is used to generate sequences of observations that match a HMM.

This option requires a distribution specification and an input HMM (parameter -i).

For example:

```
jahmm-cli generate -opdf integer -r 4 -i test.hmm
```

6.2.7 distance-kl

Computes the distance between two HMMs. This distance measure is based on the Kullback-Leibler distance, which is not symmetric.

This option requires a distribution specification, an input HMM (parameter -i) and a second input HMM against which the distance is computed (-ikl parameter).

For example:

```
jahmm-cli distance-kl -opdf integer -r 4 -i hmm1.hmm -ikl ↩
    hmm2.hmm
```

6.3 Example

The following bash script generates sequences of observations that match the ${hmm} HMM. It then learns the parameters of a HMM given those sequences using the Baum-Welch algorithm and prints the distance between the learnt HMM and ${hmm} after each iteration.

```
#!/bin/bash


#
# CLI-related variables
#

cli="java be.ac.ulg.montefiore.run.jahmm.apps.cli.Cli"
opdf="-opdf integer -r 2"


#
# File names
#

tmp_dir="."
hmm="${tmp_dir}/hmmExample.hmm"
initHmm="${tmp_dir}/hmmExampleInit.hmm"
learntHmm="${tmp_dir}/hmmExampleLearnt.hmm"
seqs="${tmp_dir}/hmmExample.seq"


#
# Functions
#

create_files ()
{
    cat > ${hmm} <<EOF
Hmm v1.0

NbStates 2

State
Pi 0.95
A 0.95 0.05
IntegerOPDF [0.95 0.05 ]

State
Pi 0.05
A 0.10 0.90
IntegerOPDF [0.20 0.80 ]
EOF

    cat > ${initHmm} <<EOF
Hmm v1.0

NbStates 2

State
Pi 0.50
A 0.80 0.20
IntegerOPDF [0.80 0.20 ]

State
Pi 0.05
A 0.20 0.80
IntegerOPDF [0.10 0.90 ]
EOF
}

erase_files ()
{
    rm -f ${hmm} ${initHmm} ${learntHmm} ${seqs}
}


#
# Main section
#

# Create sample HMMs
create_files;

# Generate sequences of observations using ${hmm}
${cli} generate ${opdf} -i ${hmm} -os ${seqs}

# Baum-Welch learning based on ${initHmm}
cp ${initHmm} ${learntHmm}
for i in 0 1 2 3 4 5 6 7
```

```
do
 echo $i `${cli} distance-kl ${opdf} -i ${learntHmm} -ikl ${hmm}`
 ${cli} learn-bw ${opdf} -i ${initHmm} -o ${learntHmm} -is ${seqs} -ni 1
done

# Print resulting HMM
echo
echo "Resulting HMM:"
${cli} print -i ${learntHmm}

# Erase the files created
erase_files;

exit 0
```

Chapter 7

Examples

7.1 Building a HMM

This code:

```
Hmm<ObservationInteger> hmm =
     new Hmm<ObservationInteger>(5, OpdfIntegerFactory(10));
```

...creates a HMM with 5 states and observation distributions that handles integers ranging from 0 to 9 (included). The state transition functions and initial probabilities are uniformly distributed. The distribution associated with each state is given by the result of the factor() method applied to the *factory* object (in this case, it returns a uniform distribution between 0 and 9).

This program fragment:

```
Hmm<ObservationInteger> hmm =
     new Hmm<ObservationInteger>(2, new OpdfIntegerFactory(2) ↩
         );
```

...creates a HMM with 2 states and default parameters.

It could be followed by a piece of code setting those parameters to known values:

```
hmm.setPi(0, 0.95);
hmm.setPi(1, 0.05);

hmm.setOpdf(0, new OpdfInteger(new double[] {0.95, 0.05}));
hmm.setOpdf(1, new OpdfInteger(new double[] {0.2, 0.8}));

hmm.setAij(0, 1, 0.05);
hmm.setAij(0, 0, 0.95);
hmm.setAij(1, 0, 0.1);
hmm.setAij(1, 1, 0.9);
```

...in order to get a valid HMM.

7.2 Multivariate gaussian distributions

The following piece of code:

- generates a new gaussian distribution with a given mean and covariance matrix;
- generates 10000 observation vectors according to this distribution;
- finds a gaussian distribution that fits those observations.

```
double[] mean = {2., 4.};
double[][] covariance = { {3., 2}, {2., 4.} };

OpdfMultiGaussian omg = new OpdfMultiGaussian(mean,  ↩
    covariance);

ObservationReal[] obs = new ObservationReal[10000];
for (int i = 0; i < obs.length; i++)
     obs[i] = omg.generate();

omg.fit(obs);
```

7.3 Learning

7.3.1 K-Means

This example finds a HMM that fits the sequences sequences (this argument is a List of List of observations). This HMM has 3 states and uses the distributions build by {\tt OpdfIntegerFactory(4)}.

```
KMeansLearner<ObservationInteger> kml =
     new KMeansLearner<ObservationInteger>(3,
          new OpdfIntegerFactory(4), sequences);
Hmm<ObservationInteger> initHmm = kml.iterate();
```

The iterate() function can be called several times to get better and better HMM models. The learn() method applies iterate() until a fix point is reached.

7.3.2 Baum-Welch

This example is similar to the one given in Section 7.3.1:

```
OpdfIntegerFactory factory = new OpdfIntegerFactory(4);
BaumWelchLearner<ObservationInteger> bwl =
     new BaumWelchLearner<ObservationInteger>(3, factory);
Hmm<ObservationInteger> learntHmm = bwl.learn(initHmm,  ↩
    sequences);
```

The learn(hmm) method iterates the algorithm a certain number of times. Its first argument is an estimation of the resulting HMM (it could be found using an iteration of the k-Means algorithm, see how initHmm has been computed in Section 7.3.1).

7.4 Data file reading

The following data file describes the following two sequences of vectors:

$$\begin{pmatrix}1.1\\2.2\end{pmatrix},\begin{pmatrix}4.4\\5.5\end{pmatrix},\begin{pmatrix}4.3\\6\end{pmatrix},\begin{pmatrix}7.7\\8.8\end{pmatrix}$$
$$\begin{pmatrix}0.5\\1.5\end{pmatrix},\begin{pmatrix}1.5\\2.5\end{pmatrix},\begin{pmatrix}4.5\\5.5\end{pmatrix},\begin{pmatrix}8.\\8.\end{pmatrix},\begin{pmatrix}7.\\8.\end{pmatrix}$$

Those sequences can be encoded in a data file:

```
# A simple data file

[ 1.1 2.2 ] ; [ 4.4 5.5 ] ; [ 4.3 6. ] ; [ 7.7 8.8 ] ;
[ 0.5 1.5 ] ; [ 1.5 2.5 ] ; [ 4.5 5.5 ] ; [ 8. 8. ] ; [ 7. 8. ↩
     ] ;
```

The file must be terminated by a new line. A sequence can span multiple lines if terminated by a backslash (\).

This simple program extract reads this file (here named “test.seq”).

```
Reader reader = new FileReader("test.seq");
List<List<ObservationVector>> seqs =  ↩
    ObservationSequencesReader.
              readSequences(new ObservationVectorReader(),  ↩
                  reader);
reader.close();
```

A 3 states HMM can be fitted to those sequences using a code such as:

```
KMeansLearner<ObservationVector> kml =
     new KMeansLearner<ObservationVector>(3,
          new OpdfMultiGaussianFactory(2), seqs);
Hmm<ObservationVector> fittedHmm = kml.learn();
```

The argument of the constructor of OpdfMultiGaussianFactory is the dimension of the vectors.

Chapter 8

References

Those papers give a good overview on HMM.

- Rabiner, Juang, *An introduction to Hidden Markov Models*, IEEE ASSP Mag., pp 4-16, June 1986.
- Juang, Rabiner, *The segmental k-means algorithm for estimating the parameters of hidden Markov Models*, IEEE Trans. ASSP, vol. 38, no. 9, pp. 1639-1641, Sept. 1990.
- Juang, Rabiner, *A Probabilistic distance measure for HMMs*, AT&T Technical Journal, vol. 64, no. 2, pp. 391-408, Feb. 1985.
- Juang, Rabiner, *Fundamentals of speech recognition*, Prentice All, AT&T, 1993.
- Faber, *Clustering and the Continuous k-Means Algorithm*, Los Alamos Science, no. 22, 1994.

Chapter 9

Changelog

v0.6.0 Major reorganisation, essentially concerning generics.

v0.5.0 Added discrete observations. Ported to Java 1.5.

v0.3.4 Added Gaussian mixture distributions. Began porting to Java 1.5.

v0.3.3 Added real observations.

v0.3.0 Classes ending with IO are now replaced by Readers and Writers.