NPBayesHMM Toolbox: Quick Start Guide

Mike Hughes (mike@michaelchughes.com)

December 10, 2012

Abstract

This guide provides a quick introduction to running MCMC simulations with the NPBayesHMM toolbox. We cover prerequisite installation and configuration details, describe the intended workflow and syntax for running a simulation, and provide some tips on accessing, interpreting, and visualizing results. For an introduction to the BP-HMM as a probabilistic model, see BPHM-Mintro.pdf.

1 Quick Intro

We'll denote the installation directory of the toolbox as `HOME`.

To actually dive into code, see the various demo scripts, located in `HOME/code/demo/`. In particular, check out `EasyDemo`, an "all in one file" introductory script showing loading data, running MCMC, and plotting results.

Unfortunately, you'll definitely need to configure your local toolbox to get these examples to run properly (see next section).

2 Prerequisites

2.1 Dependencies

We assume correctly installed versions of the following libraries:

1. Eigen : a fast matrix operations library, in C/C++

   `http://eigen.tugfamily.org`

   NPBayesHMM expects a pointer to the install directory in the file "EigenLibrary.path" in `HOME/code/`

2. Lightspeed: Tom Minka's Matlab tools for random variable generation.

   `http://research.microsoft.com/en-us/um/people/minka/software/lightspeed/`

   NPBayesHMM expects a pointer to the install directory in the file "LightspeedLibrary.path" in `HOME/code/`

2.2 Compiling MEX code

The dynamic programming routines are coded as MEX files for efficiency. You'll need to compile these from the raw C++ source code. A bash script that does this is provided: `CompileMEX.sh`. Here are the necessary steps.

1. Create a file called "EigenLibrary.path" in `HOME/code/`.

   In this file, put a single line of plain text that indicates a valid file path to the `include/eigen3/` directory of the Eigen installation. In UNIX, this looks like:

   ```
   echo '/path/to/Eigen/include/eigen3/' > EigenLibrary.path
   ```

2. Execute the script `CompileMEX.sh`, no args needed.

   This script compiles the four C++ source files found in `HOME/code/mex/`, into executables that Matlab can use.

2.3 Configure local paths

MCMC simulations create lots of data, especially when all diagnostics are on. This toolbox allows you to choose where to save the output of simulations, etc.

Create a file called "SimulationResults.path" in `HOME/code/`. This should be a plain text that indicates a valid directory where simulation results can be saved.

Also create "ProfileResults.path" in `HOME/code/`. This file indicates where to store browseable HTML results from the Matlab profiler, in case you want to use this feature.

3 Running MCMC

The NPBayesHMM toolbox allows efficient posterior inference for latent structure in sequential data. We employ Markov Chain Monte Carlo to do so. This is an iterative procedure that requires lots of time and computation, so the first rule is to be patient.

3.1 Intended Workflow

Because MCMC can be vulnerable to local optima, we recommend that users always run multiple initializations for any given experiment. We have organized the toolbox to easily support a workflow where users run many "jobs" (experiments that compare model settings or sampler schemes), and also many "tasks" (initializations) for each job. This setup conveniently allows efficient use of clusters where available, but can of course work with just a single desktop computer.

Each individual run of the sampler is given a jobID and a taskID. We'll treat these as integers, but can easily be to be more human-readable strings, like "June30exp" or "ParamA=4". Most of our visualization tools allow easy comparison for within-job and across-job experiments.

3.2 Running an MCMC Simulation

To actually run an MCMC simulation, we use the versatile entry-point function `runBPHMM`, which is called with five arguments:

```
>> runBPHMM( dataP, modelP, {jobID, taskID}, algP, initP )
```

Here's an intuitive example, that runs a short simulation on some toy data with Gaussian emissions. You can see each of these arguments in action.

```
dataP = {'SynthGaussian', 'T', 100};
modelP = {'bpM.gamma', 5 };
outP = {jobID, taskID, 'saveEvery', 5};
algP = {'Niter', 100, 'doSplitMerge', 0};
initP = {'InitFunc', @initBPHMMCheat };
runBPHMM(  dataP,  modelP, outP, algP, initP );
```

As you can see, each of these parameters (`dataP,modelP`) is a cell array that specifies some parameters of the data preprocessing, the model, the MCMC algorithm, and the initialization. These must be in the form of Name/Value pairs, such as 'Niter', 100, which says to set the number of iterations to 100. Only exception to the Name/Value restriction is that `dataP` always starts with the name of the desired dataset, and the output parameters (`outP`) always starts with jobID and taskID.

Of course, there are lots of parameters in play here. The NPBayesHMM toolbox defines a *huge* set of default parameters. You can find these in `HOME/code/BPHMM/defaults/`. All user input passed in can override *any* of these default values. This allows keeping end-user syntax relatively clean, but still allowing lots of flexibility, such as the freedom to disable sampling of certain variables for debugging, or to change the hyperparameters of any model distribution when needed.

Here's a detailed breakdown of input parameters used by `runBPHMM`.

1. `dataP` : dataset preprocessing specification

   Here's where you indicate how many sequences to model, what preprocessing to apply, etc.

   Note that you'll need to edit these defaults when you want to study a new dataset. Alternatively, you can simply pass in a `SeqData` object instead of a cell array to run inference directly on the provided dataset. Please see `HOME/doc/DataGuide.pdf` for details of how NPBayesHMM represents a collection of observed sequential data.

   Relevant defaults: `code/io/getDataPreprocInfo.m`.

2. `modelP` : MCMC algorithm parameters

   Here's where you specify anything that influences the *posterior distribution* of the data that you're trying to model. This includes hyperparameters for the beta process, HMM transition model, and HMM emissions model. Can be empty, {}, if you just want to use defaults.

   Relevant defaults: `code/BPHMM/defaults/defaultModelParams_BPHMM.m`.

3. `outP`: output parameters

   Here's where you specify the jobID, the taskID, and anything else about how to save results to disk or display progress at standard out. Always requires the first two arguments (`jobID, taskID`), which shouldn't be Name/Value pairs.

   Relevant defaults: `code/BPHMM/defaults/defaultOutputParams_BPHMM.m`.

4. `algP` : MCMC algorithm parameters Specifies the number of iterations, which proposal distributions to use, and whether to sample each component variable of the model (useful for debugging).

   Relevant defaults: `code/BPHMM/defaults/defaultMCMCParams_BPHMM.m`.

5. `initP` : MCMC initialization parameters Here's where we specify how to construct the initial state of the sampler. To be versatile, we assume that the user offers a specific function for constructing this state, and this function can take a generic set of function-specific parameters.

The attribute 'InitFunc' provides the function handle of the initializer. The rest of the fields of initP are passed along directly to this function.

For example, to initialize "from scratch" (the default method), use the function `@initBPHMMFresh`, which takes arguments that specify how many states to create, and then draws remaining parameters from their posteriors. Alternatively, to use a known set of ground-truth labels to initialize the state sequence (and, implicitly, the feature matrix) use the function `@initBPHMMCheat`.

Relevant defaults: `code/BPHMM/defaults/defaultInitMCMC_BPHMM.m`.

4 MCMC Output

Any call to `runBPHMM` will produce two binary MAT files.

`Info.mat` contains all fixed parameters used to run the experiment, and the dataset in full. We recommend saving this information so that results are interpretable months/years after a simulation is run.

`SamplerOutput.mat` contains the actual Markov chain history for the sampler. This includes (1) model parameter values ($\Psi = F, z, \eta, \theta$), and (2) diagnostics, such as log probabilities $p(\Psi, \mathbf{x})$ and accept/reject rates for the various proposals.

These two files are always saved in this location:

```
<SimulationResults.path>/<jobID>/<taskID>/
```

You can load this information into Matlab workspace easily via

```
INFO = loadSamplerInfo( jobID, taskID );
OUT = loadSamplerOutput( jobID, taskID );
```

where the variables `INFO`,`OUT` are structs that contain all the necessary fields.

5 Visualizing Results

NPBayesHMM provides several Matlab scripts for visualizing simulation performance. These are documented here (briefly).

5.1 Log Probability Trace

The most important first-pass diagnostic of a sampler run is to compute the joint log probability of the configuration: $p(\mathbf{x}, \Psi)$, where $\mathbf{x}$ is the fixed data [1]. Watching this quantity evolve over the sampler run (especially across many chains) can indicate when mixing problems exist. Of course, seeing this trace plot level off at a high value does *not* in general give any indication of convergence... this is one of the tricky bits about MCMC.

To create this plot, use the function `plotLogPr`, as follows:

Suppose I have three experiments, with jobIDs 1,2, and 3. Each one was run with 10 random initializations. To plot the joint log probability of all chains together color-coded by jobID (with legend labels 'A','B','C'), we can use the syntax

[1] You can see the BPHMMintro.pdf document for details of this computation.

```
plotLogPr( [1 2 3], 1:10, {'A', 'B','C'} );
```

The useful jobID, taskID organization scheme definitely shines here. We can easily compare the mixing behavior of several different sampler settings with this command.

5.2 State Sequences

Showing the discrete HMM state sequence can be informative.

To show the final sample of z from job 1, task 1.

```
plotStateSeq( 1, 1 );
```

If I have a `stateSeq` variable loaded into workspace.

```
plotStateSeq( stateSeq );
```

Note that if ground truth state sequence is present (in the data object), this function will automatically plot it alongside the estimated state sequence, using a relabeling scheme to align the states as best we can.

```
plotStateSeq( stateSeq, data );
```

5.3 Emission Parameters

For toy data problems, showing the learned emission parameters can be helpful in diagnosing correctness.

```
% SEE THE FINAL SAMPLED Theta for job 3, task 1
plotEmissionParams( 3, 1 );
% SEE THE SAMPLED Theta for job 1, task 1 at the 1500-th iteration
plotEmissionParams( 1, 1, 1500 );
```

To visualize a `theta` variable in my active workspace,

```
plotEmissionParams( theta );
% To plot only a certain mixture component (#5)
plotEmissionParams( theta(5) );
% In case I have the entire Psi model structure
plotEmissionParams( Psi );
```

5.4 Feature Matrix Visualization

To visualize the binary matrix F, use the following syntax:

```
% SEE THE FINAL SAMPLED F for job 3, task 1
plotFeatMat( 3, 1 );
% SEE THE SAMPLED F for job 1, task 1 at the 1500-th iteration
plotFeatMat( 1, 1, 1500 );
```

Note that just because F has a positive entry, doesn't mean that feature is actually assigned to any single timestep within the HMM state sequence z. This visualization distinguishes between "available" (F_{ik} is positive) and "active" (actually used in z) features. Of course, this is only possible when this function is provided the `stateSeq` function (which happens implicitly when called with a jobID and taskID).