HBC: Hierarchical Bayes Compiler

Pre-release version 0.1

Hal Daumé III

September 1, 2007

1 About HBC

HBC is a toolkit for implementing hierarchical Bayesian models. HBC created because I felt like I spend too much time writing boilerplate code for inference problems in Bayesian models. There are several goals of HBC:

1. Allow a natural implementation of hierarchal models.
2. Enable quick and dirty debugging of models for standard data types.
3. Focus on large-dimension discrete models.
4. More general that simple Gibbs sampling (eg., allowing for maximizations, EM and message passing).
5. Allow for hierarchical models to be easily embedded in larger programs.
6. Automatic Rao-Blackwellization (aka collapsing).
7. Allow efficient execution via compilation to other languages (such as C, Java, Matlab, etc.).

These goals distinguish HBC from other Bayesian modeling software, such as Bugs (or WinBugs [3]). In particular, our primary goal is that models created in HBC can be used directly, rather than only as a first-pass test. Moreover, we aim for scalability with respect to data size. Finally, since the goal of HBC is to compile hierarchical models *into* standard programming languages (like C), these models can easily be used as part of a larger system. This last point is in the spirit of the dynamic programming language Dyna [2].

Note that some of these aren't yet supported (in particular: 4 and 6) but should be coming soon!

To give a flavor of what HBC is all about, here is a complete implementation of a Bayesian mixture of Gaussians model in HBC format:

```
alpha     ~ Gam(10,10)
mu_{k}    ~ NorMV(vec(0.0,1,dim), 1)      , k \in [1,K]
si2       ~ IG(10,10)
pi        ~ DirSym(alpha, K)
z_{n}     ~ Mult(pi)                      , n \in [1,N]
x_{n}     ~ NorMV(mu_{z_{n}}, si2)        , n \in [1,N]
```

If you are used to reading hierarchical models, it should be quite clear what this model does. Moreover, by keeping to a very LaTeX-like style, it is quite straightforward to automatically typeset any hierarchical model. If this file were stored in `mix_gauss.hier`, and if we had data for `x` stored in a file called `X`, we could run this model (with two Gaussians) directly by saying:

```
  hbc simulate --loadM X x N dim --define K 2 mix_gauss.hier
```

Perhaps closer to my heart would be a six-line implementation of the Latent Dirichlet Allocation model [1], complete with hyperparameter estimation:

```
alpha      ~ Gam(0.1,1)
eta        ~ Gam(0.1,1)
beta_{k}   ~ DirSym(eta, V)           , k \in [1,K]
theta_{d}  ~ DirSym(alpha, K)         , d \in [1,D]
z_{d,n}    ~ Mult(theta_{d})          , d \in [1,D] , n \in [1,N_{d}]
w_{d,n}    ~ Mult(beta_{z_{d,n}})     , d \in [1,D] , n \in [1,N_{d}]
```

This code can either be run directly (eg., by a `simulate` call as above) or compiled to native C code for (much) faster execuation.

2 HBC Language

The HBC language is designed to look like a standard hierarchical model description. The only significant difference is that all indexing variables have to be qualified, so that we know how large each vector is.

An HBC program is just a sequence of variable declarations. A variable declaration always looks like "`v ~ d`", where `v` is a variable and `d` is a distribution. A variable can either be a simple variable (eg., "`alpha`") or an indexed variable (eg., "`beta_{k}`"). Note that unlike LaTeX, where braces aren't required around single-character subscripts, braces are *always* required in HBC. That is, "`beta_k`" would *not* be valid. If the variable is subscritped, then the distribution must have an associated range that qualifies the subscripts, but more on that later.

The form of a simple distribution is just `Dist(params)`, where `Dist` is one of the build-in distributions (see Section 4), such as `Nor` (Gaussian), `Mult` (multinomial), etc. `params` is just the parameters required for that given distribution (again, see Section 4 for expected paramters and specifics of the paramterizations).

Finally, if the left-hand-side variable is subscribed, then we need to specify ranges. These are of the form `i \in [lo,hi]`, where `i` is the name of the variable, `\in` is a reserved word, and `lo` and `hi` are the extends of the range. For instance `d \in [1,D]` means that `d` varies from the constant value `1` to the value of variable `D` (inclusive). Note that variables here can be subscripted; eg., `n \in [1,N_{d}]` means that `n` ranges from `1` to `N_{d}`, where `d` had better already be qualified.

From the examples above, it may appear that there is a direct mapping from lines in the source code to declarations. This is more-or-less true, but the situtation is a bit more complex. We actually use a *layout*-based parser. Essentially, this means that you can use indentation to indicated continued code. For example, the following two examples are equivalent:

```
var_{i} ~ Dist(params) , i \in [1,I]
```

```
var_{i}
  ~ Dist(params)
  , i \in [1,I]
```

How you lay out your code is entirely up to you.

Comments are Haskell-style. That is, anything beginning `--` is a comment that extends to the end of the line. Block comments (which can be embedded within each other) are begun with `{-` and ended with `-}`.

The final part of the language that is unfortunately necessary are explicit type declarations. Unfortunately, the HBC type system is—I think—undecidable, so sometimes we cannot perform complete type inference. If you get some error or warning, you can try adding explicit type declarations. Type declarations look just like distribution declarations, but the `~` changes to `\in` and you can only use `R` (for real) or `I` (for integer) as a range. For example, we can declare the type of the Gaussian means in a mixture of `K` Gaussians in `dim` dimensions as:

```
mu_{k,d} \in R , k \in [1,K] , d \in [1,dim]
```

Hopefully it will not be necessary to use type declarations frequently.

Note that you can use simple arithmetic operations subscripting. This is useful, for instance, if you want to implement a hidden Markov model. See the distribution of `z` in the example below:

```
A_{k} ~ DirSym(1.0, K)       , k \in [1,K]
B_{k} ~ DirSym(1.0, V)       , k \in [1,K]
z_{n} ~ Mult(A_{z_{n-1}})    , n \in [1,N]
w_{n} ~ Mult(B_{z_{n}}  )    , n \in [1,N]
```

Here, we see that `z_{n}` (the hidden state for position `n`) depends on `z_{n-1}` (the hidden state for position `n-1`). You can't have arbitrary expressions here (it makes it too difficult to compute Markov blankets), but you can have simple additions and subtractions. This raises the question: how are boundary cases handled? Boundary cases are always assigned a value `1`. If you want to handle them specially (eg., by a unique label), then the easiest thing to do is to force all *other* labels to be something other than `1`, so that `1` remains unique. This was a bit of a hard design choice to make, but I think it is the best option.

3 Running HBC

HBC has two primary ways to run: in simulation mode or in compilation mode. In general, you will run one of: `hbc simulate ...` to enter simulation mode or `hbc compile ...` to enter compilation mode. Simulation mode has a few more command line options than compilation mode, so we will talk about it first. However, note that simulation is *really* slow, so you probably will want to compile for real-world experiments. If you run `hbc` without any parameters, it will provide a brief explanation of its usage.

3.1 Simulation Mode

The general format for running HBC in simulation mode is:

```
hbc simulate [options] <HIER FILE>
```

Where `<HIER FILE>` is the name of a file that contains a hierarchical model. There are basically two types of options. One type controls what the simulator does; the other defines data for use in the model.

There are three options that control what the simulator does (aside from the standard verbosity options):

`--iter` Specify how many iterations to run. Eg., `--iter 50` will run 50 iterations.

`--dumpcore` Write the various forms of the internal representation of the model to the screen. This is primarily useful for debugging to see exactly what is going on.

`--dump` By default, the simulator will just print log likelihoods at each iteration. Sometimes you want more information (eg., what is the value of some variable in the model). This is what `--dump` is for; it takes two or more arguments: the first is how often to dump and the second is what to dump. The first option can be one of `last` (to dump the last sample), `best` (to dump the sample with highest log likelihood), some number N (to dump every N iterations) or `all` (to dump at every iteration). After the frequency specification, you list all variables you want it to dump. After all have been listed, you must write a semicolon to tell it that you're done listing variables.

The `--define` option allows you to explicitly define any of the (scalar) variables in your model. For instance `--define K 2` will define that the value of the variable `K` in the model is the constant value 2.

The remaining two options allow you to load data in relatively simple predefined formats. The easiest is `--loadM`, which loads a real-valued square matrix, and takes four arguments. The first argument is the name of the file to load; the second argument is the name of the variable in which to store the loaded matrix. The third and fourth arguments are the names of the variables to store the dimensions (first height, then width). For instance, `--loadM X x N dim` will load data from file `X` and store it in `x`. It will also store in `N` the number of rows in the matrix, and in `dim` the number of columns.

Finally, `--loadD` will load discrete (non-square) data of variable dimensions. This option takes four or more arguments. The first argument is the name of the file. The second argument is the variable to store the data in. The third argument is the variable in which to store the highest read value in the file. The remaining arguments are for storing the various dimensions.

For example, `--loadD testW w V N` will load data from the file `testW` and store it in the variable `w`. It will store the maximum read value in `V` (useful for vocabulary sizes). Finally, since only one dimension paramter was given (in this case, `N`), it will store in `N` the total number of "words" in the file, and `w` will range from `1` to `N`. Words are defined to be any numbers separated by spaces.

On the other hand, `--loadD testW w V D N` specifies something similar but tells us that `w` should have first dimension `D` and second dimension `N_{d}` (where `d` is the first dimension). In this case, line breaks separate "rows" and space/tab separates columns. Finally, you can load three-dimensional data with `--loadD testW w V D N M`, where now new-lines separate `D`, tabs separate `N` and spaces separate `M`.

For example, we can run the `LDA.hier` model included with the distribution (and shown above in Section 1) on the data stored in `testW` by saying:

```
hbc simulate --define K 2 --loadD testW w V D N ';' --iter 10
             --dump best z ';' LDA.hier
```

(We wouldn't actually have a line break – that's just there because the whole command wouldn't fit on one line.)

The output of this on my machine (yours may vary due to random initializations) is:

```
  Adding functions...
  Defining passed-in variables...
  Running other initializations...
  Beginning sampling...
  iter 1  ll -783.151729682117
  iter 2  ll -1064.26454916956
  iter 3  ll -648.8742198307349
  iter 4  ll -798.7226176453835
  iter 5  ll -558.7664318838561
  iter 6  ll -507.99004395602833
  iter 7  ll -462.8881048881344
  iter 8  ll -243.15131153005476
  iter 9  ll -318.09691129450334
  iter 10 ll -589.1882796715172
  == SAMPLE       -243.15131153005476
  == z    [[2, 2, 2, 2, 2, 2, 2], [1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, ...
```

Note that the reported `z` value is from a sample with log likelihood `-243`, which must have been the 8th sample in this case.

It turns out that LDA actually goes a bit berserk if you allow it to sample for both `eta` and `alpha`. We can just define these and not allow it to sample for them by running:

```
  hbc simulate --define K 2 --loadD testW w V D N ';' --iter 10
               --dump best z ';' --define alpha 1 --define eta 1 LDA.hier
```

For this, we get the following output:

```
Adding functions...
Definiting passed-in variables...
Running other initializations...
Beginning sampling...
iter 1  ll -381.88104491175426
iter 2  ll -354.1075078759999
iter 3  ll -323.9761246109454
iter 4  ll -296.9120699713346
iter 5  ll -289.7760768423048
iter 6  ll -290.708995175766
iter 7  ll -270.49744948855454
iter 8  ll -266.3137090960838
iter 9  ll -256.5513739016287
iter 10 ll -264.67874409809696
== SAMPLE       -256.5513739016287
== z    [[1, 1, 1, 1, 1, 1, 2], [1, 1, 1, 1, 1, 1, 1], [2, 2, 2, 2, 2, 2, ...
```

This is a bit more sensical if you look at the data.

As an example of a run with continuous data, we'll use the `mix_gauss.hier` example on the `X` data (generated by a Gaussian at $(2, 2)$ and one at $(-2, -2)$). To run this, we might say:

```
  hbc simulate --dump best mu ';' --loadM X x N dim --define K 2 mix_gauss.hier
```

The solution this finds (after the default of 100 iterations) is:

```
== SAMPLE       -197.49560408973477
== mu   [[2.0913, 2.1203], [-1.9317, -1.6672]]
```

Which seems again quite reasonable given the data.

3.2 Compilation Mode

Compilation mode basically means that instead of simulating the sampler online, to generate code in another language. Currently the only output language that's supported is C. Compilation mode is run by saying:

```
  hbc compile [options] <HIER FILE> <OUTPUT FILE>
```

The extension on the output file is used to determine the target language. For instance:

```
  hbc compile [options] LDA.hier LDA.c
```

Would generate C code.

The options for compilation mode are pretty much the same as for simulation mode, except the verbosity options aren't supported anymore and the `--dump` option isn't supported any more. (If you want to dump, you'll have to edit the C code yourself.)

One additional option is allowed: `--make`, which just runs gcc for you, assuming it's in your path, to create a `.out` file with the same name as the `.c` file (eg., `LDA.c` would become `LDA.out`). If you're on Windows you'll have to rename this to `LDA.exe`.

Recall our example simulation from before:

```
  hbc simulate --define K 2 --loadD testW w V D N ';' --iter 10
               --dump best z ';' --define alpha 1 --define eta 1 LDA.hier
```

We can change this into a compilation by (a) changing `simulate` to `compile`; (b) removing the `--dump` option and (c) adding `LDA.c` to the end. This yields:

```
  hbc compile --define K 2 --loadD testW w V D N ';' --iter 10
              --define alpha 1 --define eta 1 LDA.hier LDA.c
```

We can now compile `LDA.c` (if you look at the top of the `.c` file, you'll find compilation instructions):

```
  gcc -lm samplib.c stats.c LDA.c -o LDA.out
```

And run it:

```
./LDA.out
Allocating memory...
Initializing variables...
iter 1  -379.407
iter 2  -368.162
iter 3  -374.326
iter 4  -369.519
iter 5  -348.602
iter 6  -359.054
iter 7  -321.903
iter 8  -303.877
iter 9  -266.918
iter 10 -264.872
```

This result isn't identical to the simulation result (due to different random initializations), but is reasonably similar.

If you look at the C code, you'll find one function for initializing each variable, one function for sampling each variable, and one function for computing likelihoods. Most of the variable names are supposed to be somewhat informative and there are a few comments saying exactly what distributions are being sampled.

4 Built-In Distributions and Functions

HBC knows about the distributions in the following table. The (unnormalized) pdf that HBC uses is also listed since in cases like the Gamma distribution, there is more than one standard parameterization.

Distribution	**Name**	**Domain**	**Range**	**PDF**
Binomial	`Bin`	$\theta \in \mathbb{R}$	$x \in [0,1]$	$\theta^x(1-\theta)^{1-x}$
Multinomial	`Mult`	$\theta \in \mathbb{R}^D$	$x \in [1,D]$	θ_x
Gaussian (univariate)	`Nor`	$(\mu,\sigma^2) \in \mathbb{R}\times\mathbb{R}$	$x \in \mathbb{R}$	$\exp[-(\mu-x)^2/\sigma^2/2]$
Gaussian (multivariate)	`NorMV`	$(\mu,\sigma^2) \in \mathbb{R}^D\times\mathbb{R}$	$x \in \mathbb{R}^D$	$\exp[-\left\|\mu-x\right\|^2/\sigma^2/2]$
Exponential	`Exp`	$\lambda \in \mathbb{R}$	$x \in \mathbb{R}$	$x^{-\lambda}$
Gamma	`Gam`	$(a,b) \in \mathbb{R}\times\mathbb{R}$	$x \in \mathbb{R}$	$x^{a-1}\exp[-x/b]$
Inverse Gamma	`IG`	$(a,b) \in \mathbb{R}\times\mathbb{R}$	$x \in \mathbb{R}$	$(1/x)^{a-1}\exp[-1/x/b]$
Beta	`Bet`	$(a,b) \in \mathbb{R}\times\mathbb{R}$	$x \in \mathbb{R}$	$x^{a-1}(1-x)^{b-1}$
Poisson	`Poi`	$\lambda \in \mathbb{R}$	$x \in \mathbb{Z}$	$\lambda^x/x!$
Dirichlet	`Dir`	$(\alpha,D) \in \mathbb{R}^D\times\mathbb{Z}$	$\theta \in \mathbb{R}^D$	$\prod_d \theta_d^{\alpha_d-1}$
Dirichlet (symmetric)	`DirSym`	$(\alpha,D) \in \mathbb{R}\times Z$	$\theta \in \mathbb{R}^D$	$\prod_d \theta_d^{\alpha-1}$

HBC knows about the following conjugacy rules: Dirichlet/Multinomial; Normal/Normal; Gamma/Exponential; Beta/Binomial; Gamma/Symmetric-Dirichlet; Gamma/Poisson; Inverse-Gamma/Normal (variance).

The following built-in arithmetic functions are available to you: `+`, `-`, `*` and `/`. These functions are designed to operated on vectors (or matrices) of equal dimension. There are corresponding `.+`, `.-`, `.*` and `./` functions for operating on a scalar and a vector. In other words, `x+y` is valid if either `x` and `y` are both scalars or `x` and `y` are both matrices. If `x` is a scalar and `y` is a vector/matrix, then you have to write `x .+ y`. There is also `^`, but this only works for scalar exponents.

In addition, standard comparators and boolean operators are available: `=`, `~=`, `<=`, `<`, `>`, `>=`, `||` and `&&`. The standard math function `log` and `exp` also work on arbitrary size matrics (or scalars).

Finally, `vec` creates a vector/matrix. In particular `vec(val,dims)` creates a constant vector containing the value `val` with dimensions `dims`. The dimensions must contain both upper and lower bounds. For instance, `vec(0.0,1,N)` creates an N-dimensional column vector of zeros. `vec(0.0,1,N,1,M)` creates a square matrix. Finally, we can create non-square matrices by: `vec(0.0,1,N,1,M_{#1})` creates a non-square matrix. Here, `#1` means "whatever the first dimension is."

5 Compiling HBC

You will need a Haskell compiler if you want to compile HBC from source. I recommend the Glasgow Haskell Compiler `http://haskell.org/ghc/`. Using GHC, you can compile HBC with:

```
ghc --make -fglasgow-exts Main.hs -o hbc
```

With other compilers, your mileage may vary.

6 Bug Reporting

This is a very early release. Hopefully there aren't too many bugs. If something isn't working, please contact me by email at `me` AT `hal3` dot `name`. Please include everything required to reproduce your bug (the hierarchical model, data, etc.). I will try to address bugs ASAP, since I use this program too.

7 Citing

If you use HBC in research, I'd appreciate if you would either put a footnote referencing it, or cite it in a bibligraphy. If you put a footnote, something along the lines of "The implementation of these models was done in HBC, `http://hal3.name/HBC`." If you use a complete citation, please cite as:

- Daumé III, Hal. "HBC: Hierarchical Bayes Compiler." `http://hal3.name/HBC`. 2007.

Finally, I'd love to hear about it's use, so if you want to forward copies of papers to me that would be much appreciated.

References

[1] David Blei, Andrew Ng, and Michael Jordan. Latent Dirichlet allocation. *Journal of Machine Learning Research (JMLR)*, 3:993–1022, January 2003.

[2] Jason Eisner, Eric Goldlust, and Noah A. Smith. Compiling comp ling: Weighted dynamic programming and the Dyna language. In *Proceedings of the Joint Conference on Human Language Technology Conference and Empirical Methods in Natural Language Processing (HLT/EMNLP)*, 2005.

[3] D.J. Lunn, A. Thomas, N. Best, and D. Spiegelhalter. WinBUGS – a Bayesian modelling framework: concepts, structure, and extensibility. *Statistics and Computing*, 10:325–337, 2000.