AutoMix

Release 2.0

Jan 29, 2020

Contents

CHAPTER 1

Tutorial

1.1 Example

Suppose we have 10 samples from a random process X that we know not much about.

$$X = \{0.2, 0.13, 0.35, 0.17, 0.89, 0.33, 0.78, 0.23, 0.54, 0.16\}$$

We will assume the samples are independent, identically distributed (IID) from some unknown probability density distribution (PDF). The nature of the problem indicates that it could come from a Normal distribution, a Gamma distribution or perhaps a Beta distribution. We don't have any other information on the parameters,

So we propose three models to account for what we know:

- **Model 1**: Normal distribution with arbitrary parameters.

  $$p_1(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x - x_0)^2}{2\sigma^2}\right)$$

  This model has undertermined parameters σ and x_0.

- **Model 2**: Beta distribution

  $$p_2(x) = \frac{1}{B(\alpha, \beta)} x^{\alpha-1}(1 - x)^{\beta-1}$$

  This model has undetermined parameters α and β under the restriction $\alpha > 0$ and $\beta > 0$.

- **Model 3**: Gamma distribution

  $$p_3(x) = \frac{\beta^\alpha x^{\alpha-1} e^{-\beta x}}{\Gamma(\alpha)}$$

  This model has undetermined parameters α and β under the restriction $\alpha > 0$ and $\beta > 0$.

Note: The dimensions for the problems are 2 for all 3 models.

We would like to find the probability of each model given the evidence (called posterior in a Bayesian analysis.)

Assuming independent, identically distributed (IID) samples, the probability of the evidence given the model (or likelikhood) can be easily calculated

$$p(X|M) = \prod_i p_M(x_i)$$

The probability of the model given the evidence is then

$$p(M|X) \propto \prod_i p_M(x_i) p(M)$$

In what follows we will assume all models are equally likely and set $p(M) = 1/3 \forall M$.

The AutoMix sampler needs to be provided with a function that returns the target distribution (the posterior in our example), for each model. The function prototype must be in the form:

```
double targetDistribution(int model, int model_dim, double *params);
```

The log-posteriors need to be implemented up to an additive constant. A quick implementation of the log-PDFs could be:

```
#include "float.h"
#include <math.h>

int nsamples = 10;
double data_samples[] = {0.2,  0.13, 0.35, 0.17, 0.89,
                         0.33, 0.78, 0.23, 0.54, 0.16};

double logp_normal(double sigma, double x0) {
  double prod = 0;
  for (int i = 0; i < nsamples; i++) {
    double x = data_samples[i];
    prod += -(x - x0) * (x - x0);
  }
  prod = -nsamples * log(sigma) + prod / (2.0 * sigma * sigma);
  return prod;
}

double logp_beta(double alpha, double beta) {
  if (alpha <= 0.0 || beta <= 0.0) {
    return -DBL_MAX;
  }
  double prod = 0.0;
  for (int i = 0; i < nsamples; i++) {
    double x = data_samples[i];
    prod += (alpha - 1.0) * log(x) + (beta - 1.0) * log(1.0 - x);
  }
  prod +=
      nsamples * (loggamma(alpha + beta) - loggamma(alpha) - loggamma(beta));
  return prod;
}

double logp_gamma(double alpha, double beta) {
```

(continues on next page)

(continued from previous page)

```
  double prod = 0.0;
  for (int i = 0; i < nsamples; ++i) {
    double x = data_samples[i];
    prod += (alpha - 1.0) * log(x) - beta * x;
  }
  prod += nsamples * (alpha * log(beta) - loggamma(alpha));
  return prod;
}
```

Now that we have defined our problem and log-posteriors, we can set up AutoMix to generate samples from the posteriors of each model.

1.2 AutoMix Sampler

Below is an exanple of a minimal call to generate 1,000 samples from a given problem:

```
#include "automix.h"

int main() {
  int nmodels = 3;
  int model_dims[] = {2, 2, 2};
  double initRWM[] = {0.5, 0.5, 2.0, 2.0, 9.0, 2.0};
  amSampler am;
  initAMSampler(&am, nmodels, model_dims, logposterior, initRWM);
  estimate_conditional_probs(&am, 100000);
  burn_samples(&am, 10000);
  int nsweeps = 100000;
  rjmcmc_samples(&am, nsweeps);
  freeAMSampler(&am);
  return 0;
}
```

This is a simple set-up for an AutoMix program. Let's analyze it by parts.

The first line includes the AutoMix header file, where the *amSampler* structure and automix functions are defined:

```
#include "automix.h"
```

To initiate the *amSampler* struct, we need to set 4 things:

- The number of models we will use (3 in our example):

  ```
  int nmodels = 3;
  ```

- The dimensions of each model:

  ```
  int model_dims[] = {2, 2, 2};
  ```

- A function that returns the **logarithm** of the probability density function (PDF), (or the log-posterior distribution in the context of a Bayesian analysis) for each of the models in our problem. It must have the prototype:

  ```
  double logposterior(int model, double *params);
  ```

  And an implementation:

```
double logposterior(int model, double *params) {
  if (model == 1) {
    return logp_normal(params[0], params[1]);
  } else if (model == 2) {
    return logp_beta(params[0], params[1]);
  } else if (model == 3) {
    return logp_gamma(params[0], params[1]);
  }
  return 0.0;
}
```

- The initial value of the parameters for every model. This should be given as a 1d continuous array with an appropriate initial value for each model. For example the Beta distribution is bounded to [0, 1] and any initial value has to be in that interval as well. This is to avoid starting the chain in a forbidden region of the parameter space or very far from the average values of the parameters:

  ```
  double initRWM[] = {0.5, 0.5, 2.0, 2.0, 9.0, 2.0};
  ```

  In our case we start our chain with values $\sigma = 0.5$, $x_0 = 0.5$ for the Gaussian model; $\alpha = 2.0$, $\beta = 2.0$ for the Beta model; and $\alpha = 9.0$, $\beta = 2.0$ for the Gamma model.

Once all of the above is defined for our problem we can init our *amSampler*:

```
amSampler am;
initAMSampler(&am, nmodels, model_dims, logposterior, initRWM);
```

The next step is to estimate the conditional probabilities for our posterior model to create a Proposal Distribution with a multi-modal Normal mixture:

```
estimate_conditional_probs(&am, 100000);
```

We just need to pass the automix sampler struct and the number of sweeps we want for the estimation.

It is recommended to "burn" some initial samples to let the MCMC chain achieve convergence. We do this with the following line:

```
burn_samples(&am, 10000);
```

Finally we can create however many RJMCMC samples as we want:

```
int nsweeps = 100000;
rjmcmc_samples(&am, nsweeps);
```

1.3 Run Statistics

Our previous main program has one fundamental problem: the lack of output.

AutoMix saves the run statistics in different C data structures.

The most important is the struct *runStats*. In *amSampler* is the member `st`. There, we will find several statistics, including the parameters sampled during the RJMCMC run. The two most important ones are `st.k_summary` and `st.theta_summary`. For a full description of all the parameters, see *Public API*.

Just as an example, let's print out the relative probability of each model:

```c
#include <stdio.h>

int main() {
  ...
  rjmcmc_samples(&am, nsweeps);

  printf("p(M=1|E) = %lf\n", (double)am.st.ksummary[0] / nsweeps);
  printf("p(M=2|E) = %lf\n", (double)am.st.ksummary[1] / nsweeps);
  printf("p(M=3|E) = %lf\n", (double)am.st.ksummary[2] / nsweeps);

  freeAMSampler(&am);
  return 0;
}
```

This should print something like the following on screen:

```
p(M=1|E) = 0.792750
p(M=2|E) = 0.023890
p(M=3|E) = 0.183360
```

CHAPTER 2

Compilation

2.1 Library Compilation

To compile the automix library, simply type `make` on the command line:

```
$ make
```

This default command compiles only the library with optimization flags `-O3` on.

If you need to link your program against a library with debug information, type:

```
$ make DEBUG=1
```

instead.

If the compilation went without problems, you should see the `libautomix.so` file created.

Typically, libraries are installed in the `/usr/local/lib` directory, and include files such as `automix.h` are installed in `/usr/local/include`. If the default install location is enough for your needs, type:

```
$ sudo make install
```

`sudo` privilege may be needed to copy files into the `/usr/local` directory.

If you prefer to install in a different location you can provide it with the `PREFIX` command line flag:

```
$ make install PREFIX=/my/preferred/path/
```

2.2 Compiling your program against libautomix

If your program is contained in a single `main.c` file, and assuming no extra libraries or include files are required:

```
$ cc main.c -lautomix -o myprogram
```

should be enough to compile. If `/usr/local/` is not a default search place for your libraries, add a `-L/usr/local/lib` and a `-I/usr/local/include` to the compilation line:

```
$ cc main.c -L/usr/local/lib -I/usr/local/include -lautomix -o myprogram
```

You should have a `myprogram` executable compiled.

2.3 Compiling test and tutorial

Along with the library you can compile and run the tests:

```
$ make test
$ ./test
```

Or the tutorial provided in this documentation:

```
$ make tutorial
$ ./tutorial
```

2.4 Compiling "user*" legacy programs

Warning: The user examples are deprecated on version 2.x. If you want to compile AutoMix using the old interface with `user*` files, please refer to version 1.x of this package.

The original AutoMix was program-oriented and offered several example programs. To compile the example programs type:

```
$ make examples
```

After compilation you should see the programs `amtoy1`, `amtoy2`, `amcpt`, `amcptrs`, `amddi`, and `amrb9`.

You can compile each one of them individually as well, for example:

```
$ make amtoy1
$ ./amtoy1
```

2.5 Clean

To remove the executables and object (.o) files, type:

```
make clean
```

CHAPTER 3

Public API

3.1 Functions:

All public functions require as a first argument a pointer to *amSampler* structure.

int initAMSampler(amSampler *am, int nmodels, int *model_dims, targetFunc logpost, double *initRWM);

To initialize *amSampler* you need to provide:

- the number of models in *nmodels*;
- the dimensions for each model in *model_dims*;
- the logarithm of the target distribution (or posterior distribution in a Bayesian analysis) in logpost (see *targetFunc*);
- and an array with initial conditions for each model in *initRWM*.

initRWM is a flattened array, with the initial values in contiguous order for each model. If *initRWM* is passed as *NULL*, then the initial values are randomly selected from a uniform distribution in [0, 1).

initAMSampler allocates necessary memory for most array in the rest of the structures.

void freeAMSampler(amSampler *am);

Once you finish working with *amSampler*, you should free the memory with a call to *freeAMSampler*.

void estimate_conditional_probs(amSampler *am, int nsweep2);

Before running RJMCMC sweeps, you need to call `estimate_conditional_probs()` to estimate the conditional probabilities of your models, to adjust for an appropriate proposal distribution.

The statistics collected during the `estimate_conditional_probs()` calls, are stored in the `cpstats` member inside the *amSampler* struct. For a full description of `cpstats`, see *condProbStats*.

void burn_samples(amSampler *am, int nburn);

It is advised to burn some samples at the begining of a RJMCMC run to let the Markov chain converge.

void rjmcmc_samples(amSampler *am, int nsweep);

This is the function that will provide the RJMCMC samples.

The statistics collected during the `rjmcmc_samples()` calls, are stored in the `st` member inside the *amSampler* struct. For a full description of `st`, see *runStats*.

3.2 C struct's

runStats
Contains statistics of the *rjmcmc_samples* call.

naccrwmb
[unsigned long] Number of accepted Block-RWM.

ntryrwmb
[unsigned long] Number of tried Block-RWM.

naccrwms
[unsigned long] Number of accepted Single-RWM.

ntryrwms
[unsigned long] Number of tried Single-RWM.

nacctd
[unsigned long] Number of accepted Auto RJ.

ntrytd
[unsigned long] Number of tried Auto RJ.

theta_summary
[*double* ***] The theta parameter for each model, for each sweep. theta_summary[i][j][k] is the θ_k component for the j-th sweep for the i-th model.

theta_summary_len
[*int* *] the number of sweeps for theta_summary in model k. theta_summary_len[1] = 20 means that the model 1 has 20 θ samples.

ksummary
[*int* *] the number of times the model was visited.

amSampler

ch
A *chainState* instance containing the chain state of the RJMCMC.

jd
A *proposalDist* instance containing the proposal distribution for the chain.

cpstats
A *condProbStats* instance that holds statistics for the estimation of conditional probabilities (if ran).

st
A *runStats* instance that holds statistics for the RJMCMC runs (if ran).

doAdapt
A *bool* value indicating if Adaptation is to be performed or not during the RJMCMC runs. Default value is *true*.

doPerm
A *bool* value indicating if Permutation is to be performed or not during the RJMCMC runs. Default value is *false*.

student_T_dof
int type. The degree of freedom of the Student's T distribution to sample from. If 0, sample from a Normal distribution instead. Default value is 0.

am_mixfit
A *automix_mix_fit* value. Default is *FIGUEREIDO_MIX_FIT*.

seed
An unsigned long value to initialize the random number generator. Defaults to clock time.

condProbStats

rwm_summary_len
TBD.

sig_k_rwm_summary
TBD.

nacc_ntry_rwm
TBD.

nfitmix
TBD.

fitmix_annulations
TBD.

fitmix_costfnnew
TBD.

fitmix_lpn
TBD.

fitmix_Lkk
TBD.

proposalDist
The proposal distribution is of the form:

$$p(\theta) = \sum_{k=1}^{k_{max}} L_k \exp\left(-\frac{1}{2}\left(\theta_k - \mu_k\right) B \cdot B^T \left(\theta_k - \mu_k\right)\right).$$

nmodels
The number of models in M (referred as K_{max} in thesis, p 143)

nMixComps
The numberof mixture components for each model (refferred as L_k in thesis, p144)

model_dims
The dimension of the θ parameter space for each model.

lambda
The relative weights for each mixture component for each model. lambda[i][j] is the weight of the j-th mixture component for model M_i.

mu
The mean vector for the mixture component for each model. mu[i][j][k] is the k-th vector index for the j-th mixture component for model M_i.

B
$B \cdot B^T$ is the covariance matrix for each mixture component for each model. B[i][j][k][m] is the k,m index of the B matrix for mixture component j of model M_i.

sig
Vector of adapted RWM scale parameters for each model.

chainState
Struct to hold the MCMC chain state

theta
The current value of θ_k in the chain.

pk
TBD.

log_posterior
The current value of the log-posterior.

current_model_k
The current model *k* in the chain.

mdim
The dimension of the current model in the chain.

current_Lkk
TBD

nreinit
TBD.

reinit
TBD.

pkllim
TBD.

doBlockRWM
A *bool* value to indicate if the chain should do a Block-RWM. This is done every 10 sweeps.

isBurning
A *bool* that indicates wether the chain is burning samples.

sweep_i
The index of the chain.

3.3 Typedef's and Enum's

targetFunc
targetFunc is the prototype of the log-posterior function that must be passed to *amSampler* upon initialization.

```
typedef double (*targetFunc)(int model_k, double *x);
```

bool
bool is a typedef for int. C does not have a bool type but int is just as good.

```
typedef int bool;
```

automix_mix_fit
Enum to specify whether using Figuereido or AutoRJ in conditional probabilities estimation.

```
typedef enum { FIGUEREIDO_MIX_FIT = 0, AUTORJ_MIX_FIT } automix_mix_fit;
```

CHAPTER 4

License

The AutoMix sampler is free for personal and academic use, but users must reference the sampler as instructed below. For commercial use permission must be sought from the author. To seek permission for such use please send an e-mail to d_hastie@hotmail.com outlining the desired usage.

Use of the AutoMix sampler is entirely at the user's own risk. It is the responsibility of the user to ensure that any conclusions made through the use of the AutoMix sampler are valid. The author accepts no responsibility whatsoever for any loss, financial or otherwise, that may arise in connection with the use of the sampler.

The AutoMix sampler is available from http://www.davidhastie.me.uk/AutoMix. The sampler may be modified and redistributed as desired but the author encourages users to register at the above site so that notice can be received of updates of the software.

Before use, please read this README file bundled with this software.

The AutoMix package is a C program for Unix-like systems, implementing the automatic Reversible Jump MCMC sampler of the same name described in Chapters 4, 5, and 6 of David Hastie's Ph.D. thesis (included in `docs/thesis`).

While the original AutoMix is highly useful, the fact that it can only be used as an executable can limit its applicability. **LibAutoMix makes the core algorithms of Source Extractor available as a library of stand-alone functions and data structures.** The code is completely derived from the original AutoMix code base and aims to produce results compatible with it whenever possible. LibAutoMix is a C library with no dependencies outside the standard library.

CHAPTER 5

What is Reversible Jump MCMC?

Reversible Jump Markov Chain Monte Carlo (RJMCMC) extends the standard MCMC sampler to include a discrete random variable `k` that represents the index of a *model*. So, instead of sampling from the usual parameter space of a given distribution, RJMCMC will also sample across different models (distributions).

The final samples of a RJMCMC reflect the probabilities of the parameters of each model, but also the relative probabilities of the models themselves.

CHAPTER 6

Main advantages of AutoMix:

- Reversible Jump MCMC allows sampling from several distributions (models) simultaneously.
- The different models may have different number of free parameters (dimension).
- The relative frequency of sampling for different models is proportional to the probability of the model. That means that AutoMix can be used as a model selection sampler.
- AutoMix requires minimum input from the user.
- AutoMix automatically adapts proposal distributions with a multi-modal Normal mixture.

Caution: Potential users should carefully understand the limitations of using the AutoMix sampler. The reliability of results from this sampler depends on many factors, including the scaling of the parameters and the degree of multimodality of the within-model conditionals of the target distribution.

CHAPTER 7

Where to go from here?

For a quick tour of the library, check out the *Tutorial*.

To compile your program against the AutoMix library see *Compilation*.

For a description of the full public API with statistics information, see *Public API*.

Index