STAT825 Project Report
`bulkem`: an R package that quickly fits mixture models using the EM algorithm on CUDA hardware

Ian Howson (43164153, ian@mutexlabs.com)
Supervisor: Dr. Maurizio Manuguerra

June 18, 2015

Abstract

One disadvantage of the Expectation-Maximization (EM) algorithm is that it requires a lot of computation time to produce a result. This can be burdensome when many models need to be fit. The `bulkem` R package attempts to address this problem by taking advantage of CUDA hardware.

CUDA is a parallel computing platform available on NVIDIA graphics processing units, widely explored in the computational statistics literature. On the author's hardware, `bulkem` runs around thirty times faster on CUDA hardware than on a CPU when many small datasets need to be fit. For very large datasets (one million observations), CUDA is around 36 times faster.

Using the current market price of CPU vs. GPU compute time, the GPU implementation is slightly more cost-effective for small datasets and twice as cost-effective for large datasets.

To achieve this level of performance even for small datasets, `bulkem` uses task parallelism and a novel parallel summation algorithm. It is also unique in being the only CUDA implementation of EM fitting for mixture models using the inverse Gaussian distribution.

Contents

1 Introduction

We wish to fit mixture models to a large number of datasets. We assume that an appropriate model is a two-component mixture of inverse Gaussian distributions. The components of the data are not well separated. The use case is roughly 40,000 datasets of 2,000 observations each.

A natural way to estimate the mixture parameters is to use the Expectation Maximisation algorithm. However, two problems need to be overcome:

- Because of the large amount of data, most software implementations of the EM algorithm take a long time to execute
- Because the EM algorithm is sensitive to the selection of initial parameters, many attempts must be made to fit any given dataset. This makes the fitting process even slower.

One way to reduce the time needed to generate the models is to use CUDA hardware. CUDA uses graphical processing units (GPUs) found in many computers to perform general-purpose computation. The software used to perform model fitting must be customised to suit CUDA.

This report describes the design and development of `bulkem`[1], an R package which fits mixture models using CUDA hardware. Using CUDA hardware, `bulkem` can fit a large number of small datasets around thirty times faster than a conventional CPU. It can fit very large datasets around 36 times faster than a conventional CPU.

1.1 Conventions

The following variables are used throughout this report:

N: the number of elements in an array or the number of observations in a dataset

M: the number of components in the mixture model being fit

T: the number of threads being launched

D: the number of datasets

A number of performance measurements are quoted. Unless otherwise specified, those measurements were performed on an Intel i5-4460 (quad-core 3.2GHz) CPU with an NVIDIA GeForce GTX 660. The machine is running OS X Yosemite, CUDA Toolkit 6.5 and R 3.1.2.

2 Background

2.1 The inverse Gaussian distribution

The inverse Gaussian distribution is an exponential-family probability distribution with the density function:

$$f(x) = \left[\frac{\lambda}{2\pi x^3}\right]^{1/2} \exp \frac{-\lambda(x-\mu)^2}{2\mu^2 x}$$

for $x > 0$, mean $\mu > 0$ and shape $\lambda > 0$ (Seshadri 1993; p. 1).

2.2 The expectation-maximisation algorithm

The EM algorithm (Dempster et al. 1977) iteratively refines a maximum likelihood estimate in the presence of missing data. Here, we use it to fit mixture models, as described in Bilmes (1998). Two characteristics are of note:

- As it is an iterative algorithm, the execution time can be quite long. Hence our desire to find a faster way to perform model fitting.

[1] The `bulkem` source code is available at https://github.com/ihowson/bulkem

- EM is sensitive to the choice of initial parameters (Aitkin and Wilson 1980; p. 327). Therefore, we must think about how best to select them.

2.3 GPUs and CUDA

A GPU is a component of a computer that accelerates graphics operations. All modern computers and mobile devices include a GPU. GPUs can be programmed to perform general-purpose computations in the same way as a CPU; this is referred to as 'general purpose GPU computing'.

GPUs are similar to CPUs in that they run user-defined software. The key difference is that while a CPU generally has a small number of execution units (two or four are common for consumer hardware), a GPU may have thousands of execution units operating in parallel. The CPU is optimised for *serial* operations - performing a sequence of instructions as quickly as possible. The GPU is optimised for *parallel* operations where the same instructions are performed many times on different data. Each execution unit of the GPU is simple and more restricted, but there are many more of them. The peak computational output (measured in instructions per second) is far greater than for a CPU. Redesigning the problem to take advantage of this structure is the major challenge of the programmer working on a GPU computing problem.

There are two major standards for GPU computing: OpenCL and CUDA. OpenCL is supported by most GPU vendors. CUDA is only supported by NVIDIA hardware but it is a mature standard with excellent tools and documentation. We will only consider CUDA from this point on.

There are significant barriers to widespread adoption of GPU computing.

- Not everyone has access to 'large' GPU hardware. Most CPUs now ship with on-board GPUs which are sufficient for graphics, but do not provide significant performance advantages in the GPU computing context.
- The effort required to port a given algorithm to a GPU is large. Programming GPUs is significantly more difficult than CPUs
- CPUs will always be the first to get any new algorithm
- Most problems do not require a large amount of computing power. There is no incentive to speed up something that is already fast.
- Not all algorithms run faster when executed on a GPU. For an algorithm to be a good candidate for GPU execution, it must generally:
  - require many iterations (thousands), each of which is independent of the others
  - require a large amount of computation time relative to the amount of memory access
- For many problems and institutions, a cluster of general purpose PCs is a better fit. It has the advantages of running all available software, requiring minimal rework and being 'familiar' (programming a cluster is very similar to that of programming a single PC).

3 Design

3.1 The EM algorithm for mixtures of inverse Gaussian distributions

Literature review did not show any existing implementations of the EM algorithem for inverse Gaussian mixture models. Therefore, we must derive one from scratch.

As a model, the method given in Bilmes (1998; p. 3-7) is used. It shows the derivation of the EM equations for Normal mixture models.

The following variables are used:

Θ: the set of parameters estimates for the mixture model. Θ^g refers to the 'guessed' parameter set.

λ_ℓ: the shape parameter for the ℓth mixture component
μ_ℓ: the mean parameter for the ℓth mixture component
α_ℓ: the mixture weight parameter for the ℓth mixture component

3.1.1 E-step

From Bilmes (1998; p. 2), we define:

$$Q(\Theta, \Theta^{(i-1)}) = E\left[\log p(X, Y|\Theta)|X, \Theta^{(i-1)}\right]$$

For the inverse Gaussian distribution, we have the following expression for the conditional probability of mixture component ℓ given parameter guesses λ_ℓ and μ_ℓ (Wikipedia 2015a):

$$p_\ell(x|\lambda_\ell, \mu_\ell) = \left[\frac{\lambda_\ell}{2\pi x^3}\right]^{1/2} \exp \frac{-\lambda_\ell(x-\mu_\ell)^2}{2\mu^2 x} \tag{1}$$

We denote proportion of mixing components by α_ℓ in order to reduce confusion with the constant π. We have the constraint that $\sum_{\ell=1}^{M} \alpha_\ell = 1$.

The Q function is given by Bilmes (1998; p. 4):

$$Q(\Theta, \Theta^g) = \sum_{\ell=1}^{M}\sum_{i=1}^{N} \log(\alpha_\ell) p(\ell|x_i, \Theta^g) + \sum_{\ell=1}^{M}\sum_{i=1}^{N} \log(p_\ell(x_i|\theta_\ell)) p(\ell|x_i, \Theta^g) \tag{2}$$

3.1.2 M-step

On each iteration, we need to improve the parameter estimates based on the previous estimates (Bilmes 1998; p. 2):

$$\Theta^{(i)} = argmax_\Theta Q(\Theta, \Theta^{(i-1)})$$

The left-hand term of (2) is independent of the inverse Gaussian parameters and the right-hand term is independent of α. Therefore, we can reuse the result from Bilmes (1998; p. 5):

$$\alpha_\ell^{new} = \frac{1}{N}\sum_{i=1}^{N} p(\ell|x_i, \Theta^g)$$

We now need to maximise the following expression with respect to each of λ_ℓ and μ_ℓ:

$$\sum_{\ell=1}^{M}\sum_{i=1}^{N} \log(p_\ell(x_i|\theta_\ell)) p(\ell|x_i, \Theta^g) \tag{3}$$

Taking the log of (1), we get:

$$\begin{aligned}
\log p_\ell(x|\lambda_\ell, \mu_\ell) &= \log\left(\left[\frac{\lambda_\ell}{2\pi x^3}\right]^{1/2} \exp \frac{-\lambda_\ell(x-\mu_\ell)^2}{2\mu^2 x}\right) \\
&= \frac{1}{2}\log\left(\frac{\lambda_\ell}{2\pi x^3}\right) - \frac{\lambda_\ell(x-\mu_\ell)^2}{2\mu^2 x} \\
&= \frac{1}{2}\log(\lambda_\ell) - \frac{1}{2}\log(2\pi x^3) - \frac{\lambda_\ell(x-\mu_\ell)^2}{2\mu^2 x}
\end{aligned}$$

Substituting this into 3, we get:

$$\sum_{\ell=1}^{M}\sum_{i=1}^{N}(\frac{1}{2}\log(\lambda_\ell) - \frac{1}{2}\log(2\pi x_i^3) - \frac{\lambda_\ell(x_i-\mu_\ell)^2}{2\mu^2 x_i})p(\ell|x_i,\Theta^g) \tag{4}$$

We wish to maximise (4) for μ_ℓ, so we take its partial derivative with respect to μ_ℓ and set it equal to 0. Note that the first two terms inside the summations are independent of μ_ℓ, so we can ignore them for the purpose of maximisation; we seek

$$\begin{aligned} & \frac{\partial}{\partial\mu_\ell}\sum_{i=1}^{N}\left[-\frac{\lambda_\ell(x_i-\mu_\ell)^2}{2\mu^2 x_i}\right]p(\ell|x_i,\Theta^g) \\ = \quad & -\lambda\frac{\partial}{\partial\mu_\ell}\sum_{i=1}^{N}\left[\frac{(x_i-\mu_\ell)^2 p(\ell|x_i,\Theta^g)}{2\mu^2 x_i}\right] \end{aligned} \tag{5}$$

Concentrating just on the inner term of the summation, we need

$$\frac{\partial}{\partial\mu_\ell}\frac{(x_i-\mu_\ell)^2 p(\ell|x_i,\Theta^g)}{2\mu^2 x_i}$$

Apply the quotient rule:

$$\begin{aligned} = \quad & \frac{2\mu_\ell^2 x_i\left(-2x_i p(\ell|x_i,\Theta^g) + 2\mu_\ell p(\ell|x_i,\Theta^g)\right) - 4(x_i-\mu_\ell)^2 p(\ell|x_i,\Theta^g)\mu_\ell x_i}{4\mu_\ell^4 x^2} \\ = \quad & \frac{p(\ell|x_i,\Theta^g)(\mu_\ell - x_i)}{\mu_\ell^3} \end{aligned}$$

Substituting this back into (5) and setting it to zero, we get:

$$\begin{aligned} -\lambda\sum_{i=1}^{N}\frac{p(\ell|x_i,\Theta^g)(\mu_\ell-x_i)}{\mu_\ell^3} &= 0 \\ \frac{1}{\mu_\ell^3}\sum_{i=1}^{N}p(\ell|x_i,\Theta^g)(\mu_\ell-x_i) &= 0 \end{aligned}$$

This is always defined as $\mu > 0$ for the inverse Gaussian distribution.

$$\begin{aligned} \sum_{i=1}^{N}p(\ell|x_i,\Theta^g)(\mu_\ell-x_i) &= 0 \\ \mu_\ell\sum_{i=1}^{N}p(\ell|x_i,\Theta^g) - \sum_{i=1}^{N}p(\ell|x_i,\Theta^g)x_i &= 0 \\ \mu_\ell &= \frac{\sum_{i=1}^{N}p(\ell|x_i,\Theta^g)x_i}{\sum_{i=1}^{N}p(\ell|x_i,\Theta^g)} \end{aligned}$$

This is identical to the μ_ℓ maximiser for the Normal distribution (Bilmes 1998; p. 7).
In the same way, we maximise (4) for λ_ℓ:

$$
\begin{aligned}
\frac{\partial}{\partial \lambda_\ell} \sum_{i=1}^{N} (\frac{1}{2} \log(\lambda_\ell) - \frac{\lambda_\ell (x_i - \mu_\ell)^2}{2\mu_\ell^2 x_i}) p(\ell | x_i, \Theta^g) &= 0 \\
\frac{\partial}{\partial \lambda_\ell} \sum_{i=1}^{N} (\frac{1}{2} \log(\lambda_\ell) p(\ell | x_i, \Theta^g) - \frac{\partial}{\partial \lambda_\ell} \sum_{i=1}^{N} \frac{\lambda_\ell (x_i - \mu_\ell)^2 p(\ell | x_i, \Theta^g)}{2\mu_\ell^2 x_i} &= 0 \\
\sum_{i=1}^{N} \frac{p(\ell | x_i, \Theta^g)}{2\lambda_\ell} - \sum_{i=1}^{N} \frac{(x_i - \mu_\ell)^2 p(\ell | x_i, \Theta^g)}{2\mu_\ell^2 x_i} &= 0 \\
\sum_{i=1}^{N} \frac{p(\ell | x_i, \Theta^g)}{\lambda_\ell} &= \sum_{i=1}^{N} \frac{(x_i - \mu_\ell)^2 p(\ell | x_i, \Theta^g)}{\mu_\ell^2 x_i} \\
\lambda_\ell &= \frac{\sum_{i=1}^{N} p(\ell | x_i, \Theta^g)}{\sum_{i=1}^{N} \frac{(x_i - \mu_\ell)^2 p(\ell | x_i, \Theta^g)}{\mu_\ell^2 x_i}}
\end{aligned}
$$

In summary, the update equations are:

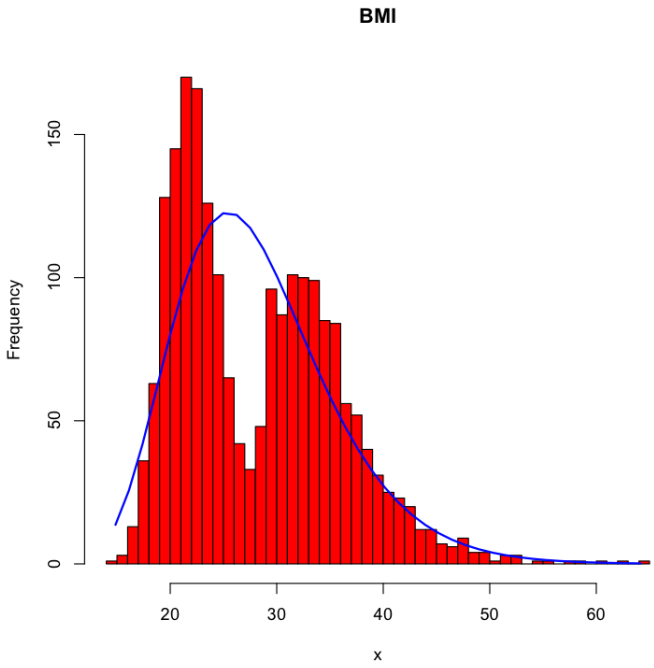
$$
\begin{aligned}
\alpha_\ell^{new} &= \frac{1}{N} \sum_{i=1}^{N} p(\ell | x_i, \Theta^g) \\
\mu_\ell^{new} &= \frac{\sum_{i=1}^{N} x_i p(\ell | x_i, \Theta^g)}{\sum_{i=1}^{N} p(\ell | x_i, \Theta^g)} \\
\lambda_\ell^{new} &= \frac{\sum_{i=1}^{N} p(\ell | x_i, \Theta^g)}{\sum_{i=1}^{N} \frac{(x_i - \mu_\ell^{old})^2 p(\ell | x_i, \Theta^g)}{(\mu_\ell^{old})^2 x_i}}
\end{aligned}
$$

We have converged when the likelihood decreases by less than ϵ, where the likelihood is given by:

$$
L(\theta; \boldsymbol{x}, \boldsymbol{z}) = \sum_{n=1}^{N} \log \left(\sum_{m=1}^{M} \alpha_m p_m(x) \right)
$$

3.2 Initialisation of the EM algorithm

Originally, we attempted to set initialisation parameters the same on every run (i.e. $\mu_1 = 0.99$, $\mu_2 = 1.01$, $\lambda_1 = 1$, $\lambda_2 = 1$, $\alpha_1 = 0.5$, $\alpha_2 = 0.5$). This yielded poor quality fits on many datasets. For example, on the BMI dataset included with the `mixsmsn` package (Prates et al. 2013), the following model was generated:



This is clearly unsatisfactory as it does not adequately model the bimodal nature of the data.

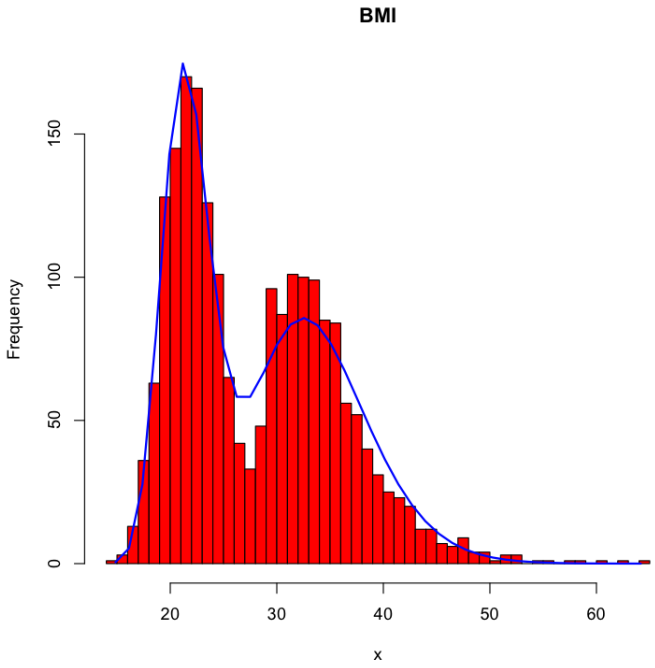
Additional research was conducted to find effective ways to initialise the EM algorithm. Many papers suggested clustering approaches such as k-means, but they require well-separated data.

We elected to use a random initialisation strategy to improve the fit, roughly following the 'alternative method' from McLachlan and Peel (2000; p. 55) or the 'subset approach' algorithm described in Schepers (2015). We chose this method as it is easy to implement and produces higher-quality fits than other methods at the expense of computation time Schepers (2015; p. 142). The algorithm proceeds as follows:

```
for each random start (e.g. 100 times):
    for each mixture component (e.g. two):
        sample p+1 observations from the dataset
        use the ML equations to estimate distribution parameters for those observations
        use those parameters as initial values for the component
    EM fit using those initial parameters and record the log-likelihood of the solution
choose the fit that achieves the highest log-likelihood
```

p is the number of parameters that characterise the distribution (2 for the inverse Gaussian distribution). We always set the mixing weight (α) components to have equal weighting (i.e. 0.5 for a two-component mixture) as we have no information on the true weighting of the components.

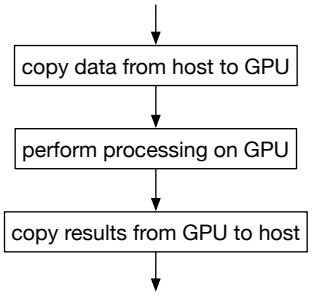
Using this algorithm on the BMI dataset, we achieve the following fit:



The new model better reflects the bimodal structure of the data.

3.3 System design

Traditionally, CUDA is used to make a single large task run very quickly. Each observation in the dataset is assigned to a separate CUDA thread; this is termed *data parallelism* (Wikipedia 2014). Larger CUDA hardware can process more observations simultaneously. Woolley (2013) states that "To get good performance ... You want to have 14K or more threads running concurrently." To maximise performance, we must make each step run as quickly as possible, even if it is wasteful of machine resources. Most tasks proceed as follows:



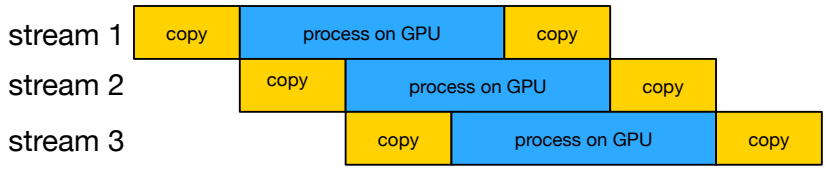
`bulkem` is intended to work efficiently on relatively small datasets with less than 14,000 observations. It expects to see a large number of datasets (thousands) and/or random starts. This suggests that we will need to have multiple tasks running simultaneously on the GPU in order to achieve good performance; *task parallelism* (Wikipedia 2015c) is more appropriate. This is a relatively uncommon usage of CUDA hardware; Tzeng et al. (2012) has a brief overview.

Recent CUDA hardware supports a feature called "streams" (Rennich 2011) which allows the GPU to perform a number of tasks simultaneously. Recent hardware can simultaneously execute up to 16 CUDA kernels while copying data back and forth between host RAM. We can use streams to:

- Overlap memory copies to and from the GPU

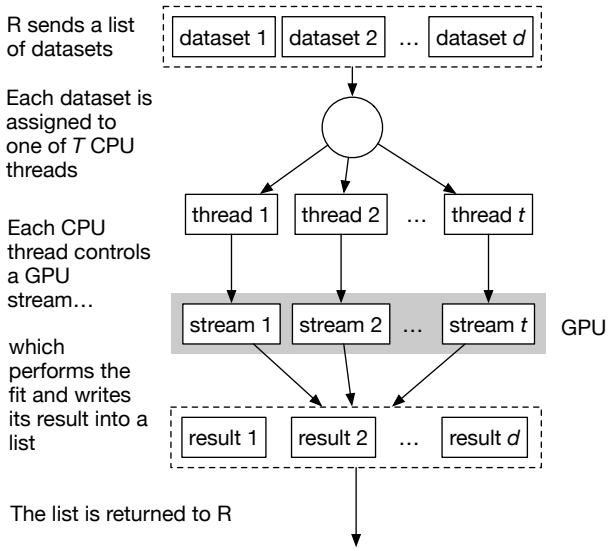
- Execute multiple kernels simultaneously. As our datasets are relatively small, this ensures that the GPU is not sitting idle.
- Use extra CPUs to queue more work for the GPU to perform, again ensuring that the GPU is kept busy.

Assuming sufficient GPU resources are available, the execution flow might look like:



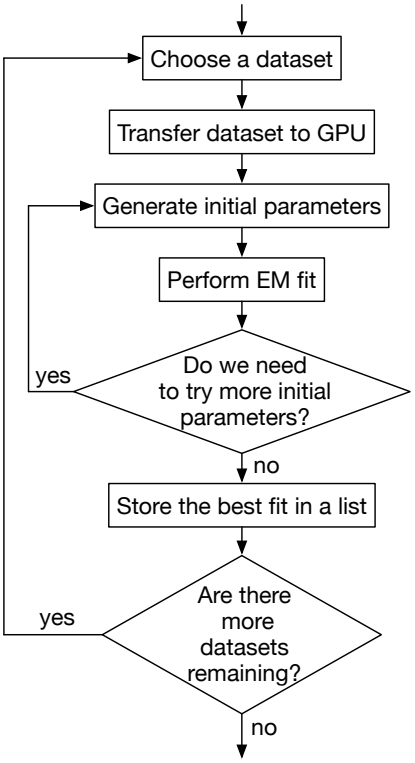
The high-level strategy for `bulkem`'s CUDA path is therefore:

- Retrieve the list of datasets from R
- Assign each dataset to a different CPU thread. Each CPU thread has an associated CUDA stream.
- Each thread generates a number of initial parameters for the dataset. It uses the GPU to execute the EM algorithm for each set of initial parameters.
- The best fit is stored in a list
- When all threads have finished (i.e. all datasets have been fit) the list is transferred back to R



3.4 CPU thread design

Each CPU thread controls a single CUDA stream. It chooses a dataset to fit, generates many sets of initial parameters and fits each using EM. The bulk of the work of EM fitting is performed on the GPU. After fitting, the best fit is selected and stored.



3.5 EM kernel design

The final kernel design is guided by the need to minimise the number of kernel launches. The reasons for this are explored in section 3.5.1.

Recall the equations that we need to evaluate to perform a single iteration of the EM algorithm:

$$\begin{aligned}
\alpha_\ell^{new} &= \frac{1}{N}\sum_{i=1}^{N} p(\ell|x_i,\Theta^g) \\
\mu_\ell^{new} &= \frac{\sum_{i=1}^{N} x_i p(\ell|x_i,\Theta^g)}{\sum_{i=1}^{N} p(\ell|x_i,\Theta^g)} \\
\lambda_\ell^{new} &= \frac{\sum_{i=1}^{N} p(\ell|x_i,\Theta^g)}{\sum_{i=1}^{N} \frac{(x_i-\mu_\ell^{old})^2 p(\ell|x_i,\Theta^g)}{(\mu_\ell^{old})^2 x_i}}
\end{aligned}$$

Each operation within the equations can be classified as either:

- performing an operation on each observation (e.g. evaluating $p(\ell|x_i,\Theta^g)$ or $(x_i-\mu_\ell^{old})^2$), or
- summing combinations of these operations (the $\sum_{i=1}^{N}$ operation appears multiple times)

Each iteration of the EM algorithm requires $2+M$ kernel launches (where M is the number of mixture components being fit). The first performs the per-observation calculations. At the end of this launch, the operands to every summation are available. This stage is referred to as `member_prob_kernel` in the source code.

The second launch performs the summation required to evaluate the current log-likelihood. This is called `lp_sum_kernel` and is described in more detail in 3.6. After this launch, we stop if convergence has been achieved.

The remaining launches perform the summations required to evaluate the new mixture parameter values, again using `lp_sum_kernel`.

The process to perform a single iteration of the EM algorithm is therefore:

1. Perform per-observation operations using `member_prob_kernel`
2. Sum over the per-observation likelihoods to calculate the solution log-likelihood using `lp_sum_kernel`
3. If we have converged (if the new log-likelihood is within ϵ of the old log-likelihood) stop the process
4. Otherwise, perform the remaining summations using `lp_sum_kernel`. The update equations can then be evaluated to generate the next iteration's initial parameter estimates.

3.5.1 Learning from failed kernel designs

It took a number of attempts to design a CUDA kernel that performed well. The following table summarises the failed attempts[2].

For reference, the pure R implementation took around 80ms to produce each fit using a single CPU core. Each fit runs on the same problem with the same initial conditions, running 100 iterations of the EM algorithm to produce a result.

Description	Time to fit	Problems
Basic CUDA C	2.3ms	Summations were implemented incorrectly, so results were incorrect
Using the Thrust API	80ms	Slow; roughly the same performance as R on CPU
Thrust with Streams	80-290ms	Multiple threads interacted poorly; often slower than before
CUB single-threaded	21ms	Not fast enough to justify use of GPU
CUB multi-threaded	100ms	`cudaFuncGetAttributes` call using a large amount of time
Modify CUB	15ms	Not fast enough to justify use of GPU
Replace CUB with single-launch sum kernel	3.5ms	Kernel launch time is now the chief bottleneck
Fuse multiple summations	2.5ms	Kernel launch time is now the chief bottleneck

Broadly, we learned the following:

- With such small datasets, kernel launch overhead takes the vast majority of the time. This is explored further below.
- Libraries such as Thrust (Hoberock and Bell 2015) and CUB (NVIDIA Corporation 2015), while making it relatively easy to develop code that runs on CUDA hardware, assume that kernel launch overhead is relatively small. They perform a lot of independent kernel launches. This makes them unsuitable in this application. We must write CUDA C/C++ by hand.
- Support for CUDA streams is still relatively new to Thrust. A lot of operations - particularly memory copies - are performed without streams in mind, which costs performance.
- Both CUB and Thrust run poorly in multithreaded environments such as this one. The different threads interact, costing performance[3].
- Performing a summation in CUB or Thrust launches many kernels

[2]Source code for these can be browsed at https://github.com/ihowson/CUDA-Task-Pipeline

[3]Profiling revealed a significant bottleneck in CUB when used in multithreaded applications. This bottleneck has been patched in https://github.com/ihowson/cub/commit/0c90360c9b9c397398a646d689ddd980aa5da811

3.5.2 Kernel launch time

Kernel launch time is the time that it takes the CPU to launch a kernel on the GPU. Boyer gives measurements showing that calling a CPU function takes about 3.3ns, but launching an asynchronous CUDA kernel takes between 3.0 and 3.9μs - a thousand times longer. Lee et al. (2010) notes that "For GPUs, we found that global inter-thread synchronization is very costly, because it involves a kernel termination and new kernel call overhead from the host."

This implies that:

- the time that the GPU spends executing the kernel must be greater than the time taken to launch the kernel or the GPU will be idle for some time
- if the CPU can execute the task in less time than the kernel takes to launch, we do not benefit from using the GPU at all

Using CUDA streams or CPU threads does not impact this restriction. If many threads are trying to launch a kernel at once, they enter a queue. Only one CPU-GPU operation (a memory copy or kernel launch) can be started at any time, even though many can be in-progress simultaneously.

With small datasets, the kernels do not take long to execute. Kernel launch time then becomes the main determinant of performance. The only way we can reduce launch time is to minimise the number of kernel launches.

3.6 Fused single launch sum kernel

The algorithm requires $2 + M$ summations per iteration. Using the standard CUB or Thrust libraries, two kernel launches are required per summation, giving a total of nine kernel launches for each iteration on a two-component mixture.

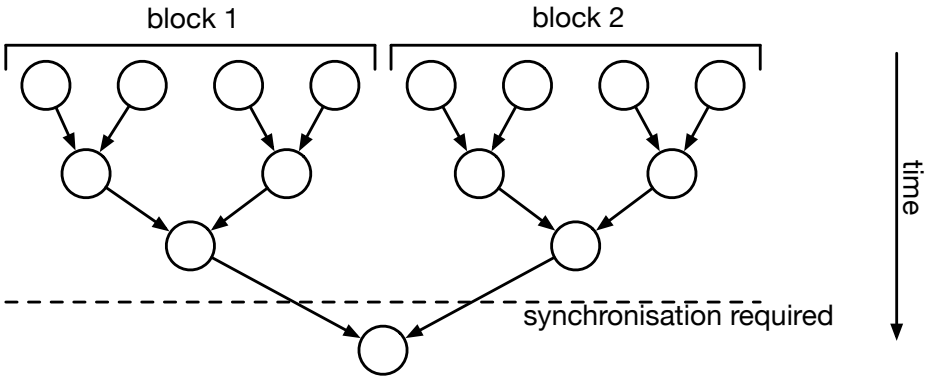
To reduce the number of kernel launches required, a new kernel was developed with two important features: it can perform *multiple summations* with a *single kernel launch.*

3.6.1 Sum reductions

Most sum kernels use a reduction tree, demonstrated on page 3 of Harris (2010). Rather than having a single thread step through each item and keeping a running total, the many cores of CUDA hardware are used. A large number of threads are launched, proportional to the number of items to be summed[4]. Each thread sums two adjacent items, a task which can be performed extremely quickly. Then, the adjacent items of *those* summations are summed, and so on until the last two items are summed. The effect of this is that the summation is performed in roughly $O(\frac{N}{T}\log(N))$ time. The traditional 'running sum' algorithm operates in $O(N)$ time; far slower on a GPU where T is large (hundreds or thousands).

For 2 million items, the kernel might be launched across 1 million threads. No existing GPU hardware has this many hardware execution units available, so the launch is split into blocks. Each block runs on the hardware in turn. The blocks are not resident in the GPU at the same time and cannot synchronise with each other. Therefore, multiple kernel launches are needed to achieve synchronisation (Harris 2010; p. 4).

[4]This is a simplified explanation; the curious reader is encouraged to read Harris (2010)

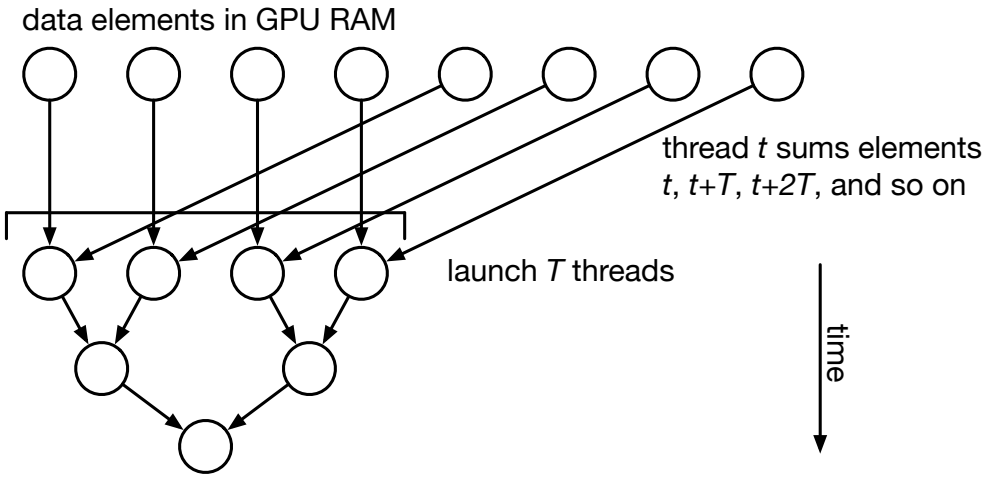


The downside to this algorithm is that each stage must run completely before the next starts, and there is no way to do this except for waiting for the kernel launch to complete. This implies that many kernel launches are required.

The new kernel solves this problem by adding a stage before the reduction tree. If T threads are launched, the tth thread calculates

$$\sum_{i=0}^{\lceil N/T \rceil} \begin{cases} x_{t+iT} & \text{for } t + iT < N \\ 0 & \text{otherwise} \end{cases}$$

In other words, it sums every Tth element into a T-wide array. Then, the sum reduction can proceed as per normal. This is slower than the traditional algorithm, but it has the key advantage that global synchronisation is not required. The entire summation can execute with a single kernel launch.



T can be selected to be any number smaller than the maximum thread block size supported by the CUDA hardware. It is critical that all threads are executed simultaneously (i.e. the kernel must be launched with a grid size of 1).

3.6.2 Kernel fusion

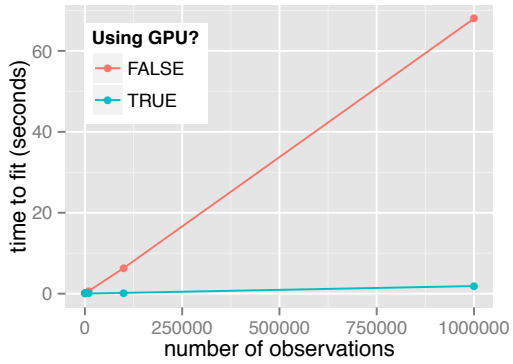
The M-step of EM requires three summations across N-sized arrays. These three summations can be performed in a single kernel launch by providing the sum kernel with the details of all three arrays in a single launch, rather than performing three launches. This technique is called Vectored I/O in other contexts (Wikipedia 2015d).

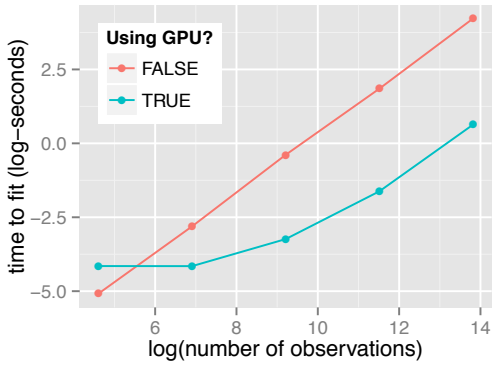
4 Results

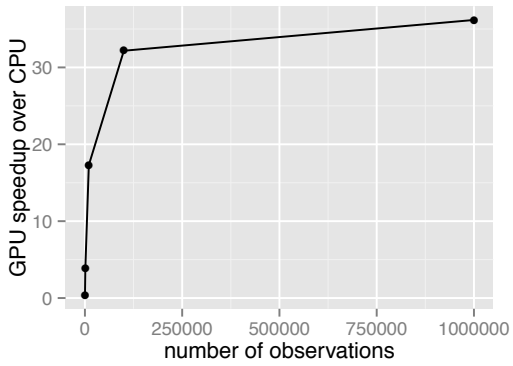
4.1 Time to fit a single dataset

To test performance across varying dataset sizes, we sample from a two-component inverse Gaussian mixture model with known parameters. Only a single dataset is fit.

Dataset size	CPU time (seconds)	GPU time (seconds)	GPU speedup
100	0.00620	0.01576	0.39
1,000	0.06032	0.01572	3.84
10,000	0.67876	0.03924	17.30
100,000	6.35048	0.19740	32.17
1,000,000	67.98868	1.87952	36.17





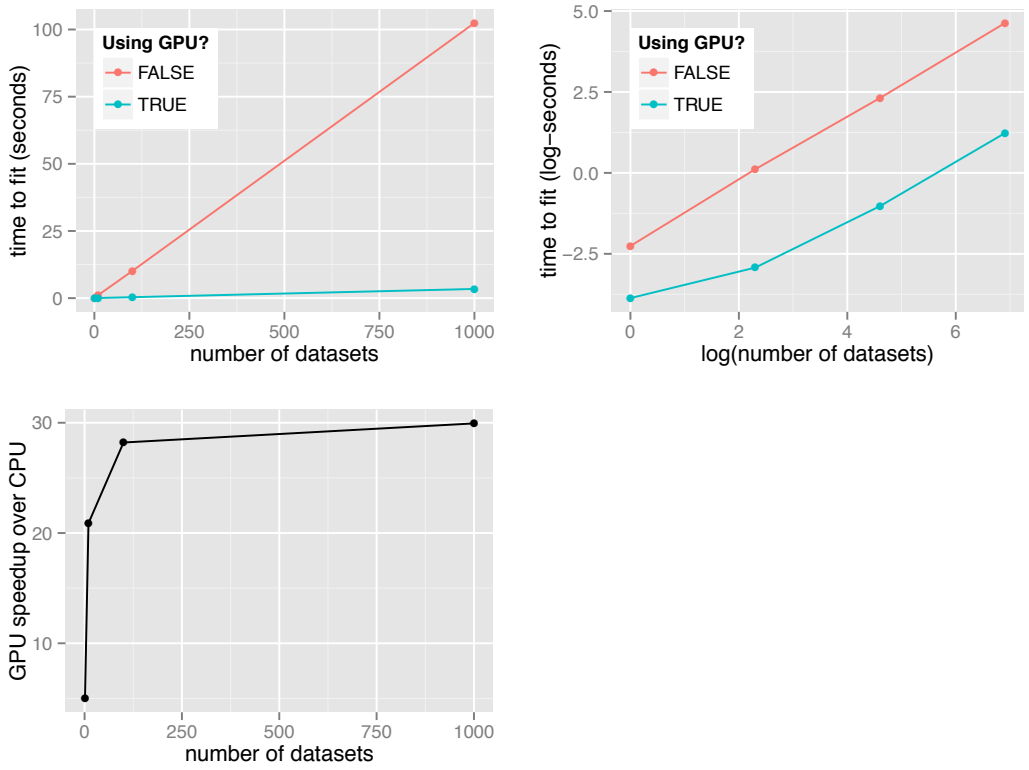


On the test hardware, we see that the GPU is slower for small dataset sizes (100 samples) but outperforms the CPU for larger dataset sizes. For datasets with 1 million samples, the GPU runs around 36 times faster than the CPU.

4.2 Time to fit many datasets

In this case, the dataset size is held constant (2000 samples) and we fit many datasets simultaneously, generating them in the same way as for the single dataset case.

Number of datasets	CPU time (seconds)	GPU time (seconds)	GPU speedup
1	0.10452	0.02092	5.00
10	1.12048	0.05364	20.89
100	10.12788	0.35904	28.21
1000	102.40840	3.42036	29.94



We see similar results - the ratio of GPU-CPU performance increases as the number of datasets increases. When 1000 datasets of 2000 samples are being fit simultaneously, the GPU runs around 30 times as fast as the CPU.

4.3 Multiple datasets on EC2

Comparing performance of CPUs vs. GPUs is somewhat unsound; there is no obvious way to say "this CPU is equivalent to this GPU". Most papers, including this one, compare performance using whatever hardware the author had available at the time (Gillespie 2011). No effort was made to optimise the CPU implementation, while significant time was spent optimising the GPU implementation, an issue discussed in depth in (Lee et al. 2010).

Fortunately, services such as Amazon EC2 (Amazon Web Services) provide an alternative way to compare the CPU and GPU approaches: cost of rental. For a given price, one will be able to rent a certain amount of hardware which will perform the desired computations in an amount of time. Both CPU and GPU time can be rented. A fairer way to compare the two technologies is the *cost to perform your computation.*

A summary of the machine configurations is available at Amazon Web Services (2015). For reference, ECUs are a measure of allocated CPU capacity. The US East region was selected as it is generally the lowest priced.

For the CPU implementation, we selected a c4.large instance as they provide the best price-performance ratio at the time of writing (eight ECUs and two CPU cores at USD$0.116/hour as of 2015-06-09). t2 instances are not suitable as they provide 'burstable' CPU performance; they are not intended for long-running jobs. This machine has two CPU cores, but the R implementation will only use one. As there are no dependencies between datasets, we will assume that additional CPU cores will provide a linear speedup (that is, with appropriate software, we could obtain double the performance with double the CPU cores). The rationale for this is explored further in 5.1. Also note

that pricing for c4 instances is close to constant per CPU core and ECU allocation; the cost-to-fit ought to remain constant regardless of instance choice.

Name	Number of CPU cores	ECU allocation	Price per hour (USD)
c4.large	2	8	0.116
c4.xlarge	4	16	0.232
c4.2xlarge	8	31	0.464
c4.4xlarge	16	62	0.928
c4.8xlarge	36	132	1.856

For the GPU implementation, we chose a g2.x2large instance at USD$0.650/hour. Rephrasing this in terms of speedup ratios, the GPU implementation must achieve a $0.65/(0.116/2) = 11.2$x speedup ratio in order to break even on cost.

As before, all datasets contain 2000 randomly generated samples.

Datasets (D)	CPU time (seconds)	GPU time (seconds)	CPU cost (USD$\times 10^{-6}$)	GPU cost (USD$\times 10^{-6}$)
1	0.08912	0.01708	1.44	3.08
10	0.87360	0.06940	14.07	12.53
100	9.17784	0.63868	147.86	115.32
1000	84.44992	6.39072	1360.58	1153.88

From this, we can see that the GPU implementation is slightly more cost-effective than the CPU implementation for larger problems. The difference is not large and could probably be eliminated altogether with some optimisation work on the CPU implementation.

These prices differences may seem to be trivial (who cares about *microcents*?) but recall that use cases may include many more datasets (tens of thousands of datasets is the intended use case) and require random initialisation to achieve a good fit (100 random initialisations means 100 times as much work, and therefore cost). For 40,000 datasets and 100 random initialisations, the cost is around USD$5.44 using the CPU implementation and USD$4.62 using the GPU implementation.

For the large dataset test, we obtain the following results:

Samples (N)	CPU time (seconds)	GPU time (seconds)	CPU cost (USD$\times 10^{-6}$)	GPU cost (USD$\times 10^{-6}$)
100	0.00724	0.01616	0.12	2.92
1,000	0.05264	0.02032	0.85	3.67
10,000	0.57628	0.03264	9.28	5.89
100,000	6.03700	0.22568	97.26	40.75
1,000,000	50.11764	2.25400	807.45	406.97

For sufficiently large problems, the GPU instances can perform the model fits at roughly half the price.

5 Discussion

5.1 The linear speedup assumption

Fitting models to independent datasets is an embarrassingly parallel (Wikipedia 2015b) problem. The datasets have no dependence on each other and can be fitted separately.

This implies that, in an ideal world:

- If we have enough datasets, we can expect the performance figures quoted here to extrapolate linearly (i.e. if fitting 1000 datasets takes 10 seconds, we can expect that fitting 2000 datasets will take around 20 seconds)
- If we have more hardware to parallelise across (e.g. more CPU cores, more computers with or without GPUs) we can expect them to reduce the computation time proportionally to the amount of resources added

- Assuming that EC2 has an unlimited supply of hardware for us to rent, we can perform very large model fits in an arbitrarily small amount of time *with the same total cost*. Renting twice as much hardware will halve our model fit time, so the total expenditure remains the same.

Of course, in practice things are not so ideal:

- EC2 bills per-hour, so we cannot reduce execution time for extremely large jobs below an hour without incurring additional costs
- EC2 instances take some time to boot, so there is a cost to using large numbers of instances
- Large clusters of machines incur some overhead for communication
- Very small tasks have fixed overheads. We saw this in the GPU results, where there was almost no time difference between a 100-sample dataset and a 1000-sample dataset.

Using R, the `foreach` package (Revolution Analytics and Weston 2014) makes it easy to parallelise code across cores within a computer. The `snow` package (Tierney et al. 2013) is suitable for use across networked clusters of computers.

5.2 Future work

5.2.1 Improving GPU performance

Kernel launch time is still the limiting factor, so further reducing the number of kernel launches is the natural way to improve performance. Ideally, the entire EM algorithm (across multiple iterations) could be moved to the GPU using similar techniques to `lp_sum` to handle datasets larger than the thread block size.

On the author's hardware, compute occupancy is around 60%, so we could expect, at most, another $\frac{1}{0.6} = 67\%$ performance gain.

At that stage, GPU performance would likely be the limiting factor. The current implementation of the `lp_sum` summation is not very efficient, but it would be prudent to check for hotspots with a profiler before investing further development time.

Finally, while running the GPU software, the CPUs have significant idle time. With additional software support, one could perform additional model fits using that idle CPU capacity, improving performance-per-dollar further.

5.2.2 Improving CPU performance

The obvious way to improve performance of the CPU implementation is to take advantage of additional CPU cores. The easiest way to achieve this is with the `foreach` R package, which can run arbitrary R code across any number of CPU threads. Running the R code under a profiler ought to reveal hotspots which can guide optimisation of the R code, Finally, rewriting the R implementation in C might provide further improvement.

5.2.3 New functionality

Modifying `bulkem` to fit Normal mixture models would be fairly straightforward and very useful; Normal models are far more common than inverse Gaussian.

6 Conclusion

This report describes `bulkem`, an R package which fits inverse Gaussian mixture models using the EM algorithm. It has demonstrated that GPUs can provide significant performance and cost advantages over CPUs in this application. Unlike most GPU computing packages, `bulkem` offers significant

performance improvement even on small datasets. Directions for further improvement of the `bulkem` algorithms have been identified which might provide further improvement.

References

M Aitkin and GT Wilson. Mixture models, outliers, and the EM algorithm. *Technometrics*, 22(3): 325–331, Aug 1980.

Amazon Web Services. Amazon EC2. URL `http://aws.amazon.com/ec2/`.

Amazon Web Services. Amazon EC2 pricing, 2015. URL `http://aws.amazon.com/ec2/pricing/`.

JA Bilmes. A gentle tutorial of the EM algorithm and its application to parameter estimation for Gaussian mixture and hidden Markov models. Technical Report TR-97-021, International Computer Science Institute, 1998.

M Boyer. CUDA kernel overhead. URL `http://www.cs.virginia.edu/~mwb7w/cuda_support/kernel_overhead.html`.

AP Dempster, NM Laird, and DB Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1):1–38, 1977.

C Gillespie. Reviewing a paper that uses GPUs, July 2011. URL `https://csgillespie.wordpress.com/2011/07/12/how-to-review-a-gpu-statistics-paper/`.

M Harris. Optimizing parallel reduction in CUDA, Mar 2010. URL `http://docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pdf`.

J Hoberock and N Bell. Thrust - parallel algorithms library, Mar 2015. URL `http://thrust.github.io/`.

VW Lee, C Kim, J Chhugani, M Deisher, D Kim, AD Nguyen, N Satish, M Smelyanskiy, S Chennupaty, P Hammarlund, R Singhal, and P Dubey. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ISCA '10 Proceedings of the 37th Annual International Symposium on Computer Architecture*, pages 451–460. ACM, 2010.

G McLachlan and D Peel. *Finite mixture models*. John Wiley & Sons, Inc, 2000.

NVIDIA Corporation. CUB, Apr 2015. URL `http://nvlabs.github.io/cub/`.

MO Prates, CRB Cabral, and VH Lachos. mixsmsn: Fitting finite mixture of scale mixture of skew-normal distributions. *Journal of Statistical Software*, 54(12), August 2013.

S Rennich. CUDA C/C++ streams and concurrency, 2011. URL `http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf`.

Revolution Analytics and S Weston. foreach: foreach looping construct for R, Apr 2014. URL `http://cran.r-project.org/web/packages/foreach/index.html`.

J Schepers. Improved random-starting method for the EM algorithm for finite mixtures of regressions. *Behavior Research Methods*, 47(1):134–146, Mar 2015.

V Seshadri. *The inverse gaussian distribution: a case study in exponential families*. Oxford Science Publications, 1993.

L Tierney, AJ Rossini, N Li, and H Sevcikova. snow: simple network of workstations, Sep 2013. URL `http://cran.r-project.org/web/packages/snow/index.html`.

S Tzeng, A Patney, and JD Owens. GPU task-parallelism: primitives and applications, 2012. URL http://on-demand.gputechconf.com/gtc/2012/presentations/S0138-GPU-Task-Parallelism-Primitives-and-Apps.pdf.

Wikipedia. Data parallelism, Dec 2014. URL http://en.wikipedia.org/wiki/Data_parallelism.

Wikipedia. Inverse gaussian distribution, February 2015a. URL http://en.wikipedia.org/wiki/Inverse_Gaussian_distribution.

Wikipedia. Embarrassingly parallel, Mar 2015b. URL http://en.wikipedia.org/wiki/Embarrassingly_parallel.

Wikipedia. Task parallelism, Mar 2015c. URL http://en.wikipedia.org/wiki/Task_parallelism.

Wikipedia. Vectored I/O, February 2015d. URL http://en.wikipedia.org/wiki/Vectored_I/O.

C Woolley. GPU optimization fundamentals, 2013. URL https://www.olcf.ornl.gov/wp-content/uploads/2013/02/GPU_Opt_Fund-CW1.pdf.

Appendix: R package reference

R documentation

of 'bulkem.Rd'

June 12, 2015

---

bulkem — *Fit Finite Mixture Models to Many Datasets Using Expectation Maximisation*

Description

Fit finite mixture models composed of the inverse Gaussian distribution to a number of datasets. A random initialisation strategy helps with difficult-to-fit datasets. CUDA acceleration can be used to reduce fit time dramatically.

Usage

```
bulkem(datasets, num.components=2, max.iters=100, random.inits=1, use.gpu=TRUE,
epsilon=0.000001, verbose=FALSE)
```

Arguments

datasets	A list of datasets to fit. Each element of the list must be a vector of numbers.
num.components	The number of components in the fitted mixture models
max.iters	Maximum number of iterations of the EM algorithm
random.inits	Number of times to try to fit each dataset
use.gpu	TRUE to use an available CUDA GPU; FALSE to use the CPU
epsilon	If the difference between log-likelihood of iterations of EM is below this number, stop iterating. Higher epsilon speeds up fits but gives less accurate parameter estimates; lower epsilon gives more accurate parameter estimates but takes longer to fit.
verbose	If TRUE, dump additional debugging information to the console

Details

bulkem fits finite mixture models composed of inverse Gaussian distributed components. It has two major advantages over existing packages:

- CUDA (GPU) acceleration
- Random initialisation of parameters

CUDA acceleration is most useful if you have a large number of datasets, require many random initialisations, or both. The degree of performance improvement that you achieve depends on what hardware you are running. On the author's hardware (Intel i5-4460 and GeForce GTX 660) enabling CUDA yields a 30x speedup for large numbers of small datasets and 36x for large datasets.

Value

A list of mixEM objects. Each object describes a model that was fit by the EM algorithm. Each object in the list corresponds to a dataset in the input `datasets` list (that is, the ith object in the returned list is the model parameters for ith object in the `datasets` list.

Each mixEM object contains the following properties:

`lambda`	Vector of λ (shape) values for the model parameter estimates
`mu`	Vector of μ (shape) values for the model parameter estimates
`alpha`	Vector of α (component weight) values for the model parameter estimates
`init_lambda`	Vector of λ values used as initial parameter estimates
`init_mu`	Vector of μ values used as initial parameter estimates
`init_alpha`	Vector α values used as initial parameter estimates
`loglik`	The final log-likelihood of the model fit
`num_iterations`	The number of iterations of the EM algorithm required before convergence was achieved
`fit_success`	`TRUE` if the model was fit successfully; `FALSE` otherwise. The algorithm might fail to fit if convergence was not achieved within the allowed number of iterations or if an error was detected during fitting.

For all parameter estimates returned as vectors, the ith element of the vector corresponds to the parameter estimate for the ith mixture component (e.g. the second element of the lambda vector is the lambda estimate for the second mixture component).

Random initialisation

The EM algorithm is sensitive to the choice of initial parameter estimates. `bulkem` tries many different sets of initialisation parameters in order to find a global optimum. The initialisation strategy proceeds as follows:

1. Sample $p + 1$ observations from the dataset, where p is the number of parameters in each mixture component (so, three observations for each inverse Gaussian mixture component)
2. Generate a maximum likelihood estimate using those observations
3. Use the MLE as the initial parameter estimates

Examples

```
# Fit the BMI dataset using inverse Gaussian mixture models
# We need to try a few different initial conditions to get a good fit
library(bulkem)
library(mixsmsn)
library(statmod)
data('bmi')
x <- bmi$bmi
xlist <- list(x)
```

```
fits <- bulkem(xlist, random.inits=10)
fit <- fits[[1]]

print(paste0('fit llik: ', fit$llik, 'alpha: ', fit$alpha, ', lambda=',
fit$lambda, ', mu=', fit$mu))

# plot the density function over the histogram
# modified from http://www.statmethods.net/graphs/density.html
h <- hist(x, breaks=40, col="red", main="BMI")
xfit <- seq(min(x), max(x), length=40)
yfit <- fit$alpha[1] * dinvgauss(xfit, mean=fit$mu[1], shape=fit$lambda[1]) +
fit$alpha[2] * dinvgauss(xfit, mean=fit$mu[2], shape=fit$lambda[2])
yfit <- yfit * diff(h$mids[1:2]) * length(x)
lines(xfit, yfit, col="blue", lwd=2)


# Fit a large number of small datasets and compare how long CPU and GPU take
library(bulkem)
library(statmod)

USEGPU <- c(TRUE, FALSE)

D <- 100  # number of datasets
N <- 2000  # number of observations in each
NUM_REPEATS <- 10  # perform 10 random inits of each dataset

run <- function(xlist, use.gpu) {
    fits <- bulkem(datasets=xlist, use.gpu=use.gpu)

    f <- fits[[1]]

    print(sprintf("The first fit looks like mu=[
     alpha=[
     f$alpha[[1]], f$alpha[[2]]))
}

make.dataset <- function(x) {
    alpha <- c(0.3, 0.7)  # mixture weights

    components <- sample(1:2, prob=alpha, size=N, replace=TRUE)
    mus <- c(1, 3)
    lambdas <- c(30, 0.2)

    x <- rinvgauss(n=N, mean=mus[components], shape=lambdas[components])
    return(x)
}

# make a list of 100 datasets
xlist <- lapply(1:D, make.dataset)

for (use.gpu in USEGPU) {
    print(sprintf('--- Using GPU:
    each,
    time <- system.time(run(xlist, use.gpu=use.gpu))[3]
    print(sprintf("    Completed in
}
```

```
# On my computer, this produces the following output:

# [1] "--- Using GPU: 1. 100 datasets with 2000 observations in each, 10
#     attempts per dataset."
# Processed 100 chunks
# cuda parallel: 0.423166 seconds elapsed
# [1] "The last fit looks like mu=[2.783511 1.015576], lambda=[0.210135
#     28.845700], alpha=[0.722182 0.277818]"
# [1] "    Completed in 0.511000 seconds"
# [1] "--- Using GPU: 0. 100 datasets with 2000 observations in each, 10
#     attempts per dataset."
# [1] "The last fit looks like mu=[1.015576 2.783510], lambda=[28.845905
#     0.210135], alpha=[0.277818 0.722182]"
# [1] "    Completed in 10.214000 seconds"
```