|epcc|

Fitting Large-Scale Gaussian Mixtures With Accelerated Gradient Descent

Nestor Sanchez Guadarrama

August 2018

MSc in High Performance Computing With Data Science

The University of Edinburgh

Year of Presentation: 2018

Abstract

Gaussian Mixtures Models (GMMs) are amongst the most popular clustering algorithms in Machine Learning due to their simplicity and interpretability, endowing the fitted data with a probabilistic model. However, most implementations use Expectation-Maximisation (EM), which requires multiple passes through the data to converge, making them expensive to use in applications for which high data availability and velocity are important. As a more scalable alternative, in this work we build on the results of [Hosseini and Sra (2017)] to implement accelerated stochastic gradient descent (SGD) algorithms for GMMs. Through empirical results, we show that they outperform standard SGD in convergence speed; moreover, we also show that they substantially outperform full-batch optimisation algorithms in wallclock time while using a fraction of hardware resources and producing results of similar quality. Gradient-based optimisation makes the model a natural candidate for data stream clustering; we develop an intuitive understanding of the step size parameter as a forgetfulness rate for such applications and analyse its throughput capacity for distributed processing. Finally, we show that conjugate prior regularisation, although preventing covariance singularities, can have a negative effect on the model's quality in mini-batch optimisation; we propose a logarithmic barrier regularisation that also prevents covariance singlarities while having only a small effect on the optimisation result.

A substantial amount of effort was devoted to developing a high-quality software implementation robust enough to be used as an out of the box tool for any scientific application or production environment.

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgements

I would like to express my gratitude to Dr. Amy Krause for her guidance throughout this project and also to all of the EPCC staff for their help and dedication throughout the course.

1 Introduction

A Gaussian Mixture Models (GMM) is a clustering algorithm that segments the data and provides it with a simple, concise probabilistic description from which many statistical properties can be inferred and interpreted; it is in part thanks to this that it has been successfully applied to many problems in different scientific areas, such as Astronomy[Lee et al. (2012)], Chemestry[Spainhour et al. (2014)], Biology[Maretić and Lacković (2014)] and Acoustics[Zhuang et al. (2009)], to name a few. Due to its popularity, there are already many implementations in different software packages and programming languages with which a researcher can easily process small and middle-sized static datasets with up to, say, a few hundred thousands of observations. However, most of those implementations use the EM algorithm, and as the dataset size grows they become increasingly expensive to use, because many passes through the dats are needed for convergence; this can have a considerable impact in the project's budget if it is necessary to devote significant computational resources to fitting such models. Moreover, batch algorithms such as EM are not a appropriate to use for streaming applications, where data is produced continuously and can have non-stationary distributions.

In this work, we address the scalability issue of GMMs by implementing accelerated gradient descent algorithms (SGD) to fit the models; gradient descent algorithms have already been proven to produce extremely scalable and robust Machine Learning models in other areas, such as Deep Learning[Amari (1993)] by using mini-batches of data to perform stochastic optimisation, allowing such models to be efficiently trained with billions of examples; furthermore, the accelerated variants of said algorithms have additional desirable theoretical properties[Goh (2017), Bottou et al. (2016)] and generally achieve faster convergence[Sutskever et al. (2013), Kingma and Ba (2014)]. The main problem when trying to use stochastic gradient descent (SGD) for fitting GMMs is the symmetry and positive-definiteness constraints on the covariance matrices, which cannot be enforced by SGD alone; constrained gradient-based optimisation have also been tried before, but was heavily outperformed by EM[Sra and Hosseini (2013)]. However, in [Hosseini and Sra (2017)], a reformulation for the GMM optimisation problem was derived in which it is possible to use Riemannian Optimisation techniques to perform unconstrained gradient-based optimisation on the manifold of symmetric positive deffinite (SPD) matrices to fit the models. This has two immediate advantages: firstly, it allows large-scale mixture modelling by using stochastic optimisation as described above, and secondly, it provides a very natural way of handling streaming data, making it straightforward to use GMMs in data stream clustering applications.

We test the behaviour of our algorithms and compare them to standard SGD as proposed on [Hosseini and Sra (2017)] under many different conditions, varying data separability, number of components, acceleration parameters and step size hyperparameters, and we produce results that show that standard SGD is consistently and significantly outperformed in convergence speed under practically all tested conditions. We also compare sequential accelerated SGD to an EM-optimised, 16-core cluster Spark GMM, finding substantial reductions in wallclock time for relatively large datasets while getting

models of similar quality.

One of the difficulties that may arise when fitting GMMs is so-called *covariance singularities*, that occurs when a component's mean estimate collapses with a data point, opening the possibility of an unbounded growth in log-likelihood by shrinking the corresponding covariance matrix to zero, creating a single-point cluster component. This problem is usually prevented by adding some kind of regularisation to the log-likelihood function, and one of the most popular choices to do this on a GMM model is placing a conjugate prior distribution on the parameters. However, we produce empirical results that suggests that for mini-batch optimisation, the prior's weight in each iteration may be large enough to affect the model's quality considerably; this is why we propose a regularisation by logarithmic barrier, which prevents covariance singularities while requiring less memory and CPU usage and having a much smaller impact on the model's quality. We also test the program's throughput for streaming applications and develop an intuitive interpretation of the step size as the model's forgetfullness rate.

The second goal of the project was creating a robust, open-source implementation that can be used as an out-of-the-box tool for any suitable clustering problem. We do this using `scala` and `Spark`, a popular open-source framework for distributed computing which already implements many Machine Learning models, including EM-fitted GMMs, which we will use to compare our results. Special care was taking to ensure that the resulting program is well documented, easy to use and easily extensible.

The report is organised as follows: in the second section, we explain the necessary theory and concepts to understand the fundamentals, methodology and results; in the third section we briefly review the problem reformulation derived in [Hosseini and Sra (2017)] and that forms the foundation of this project; in the fourth one, the software implementation is explained and some of the design choices are reviewed; the fifth section contains the results of the tests we ran, where we show how our program behaves under different circumstances and parameter values. We also compare it to the current GMM implementation in `Spark`; finally, the sixth sections contains the work's conclusions.

2 Technical Background

In this section we talk briefly about the concepts that one needs to know to understand the problem formulation on the next section.

2.1 Gaussian Mixtures

In a Gaussian Mixture Model (GMM), we assume that observed vectors $x \in \mathbb{R}^m$ are generated by one of K different underlying Gaussian distributions, although we don't know which particular distribution a point proceeds from, or the distributions' parameters.

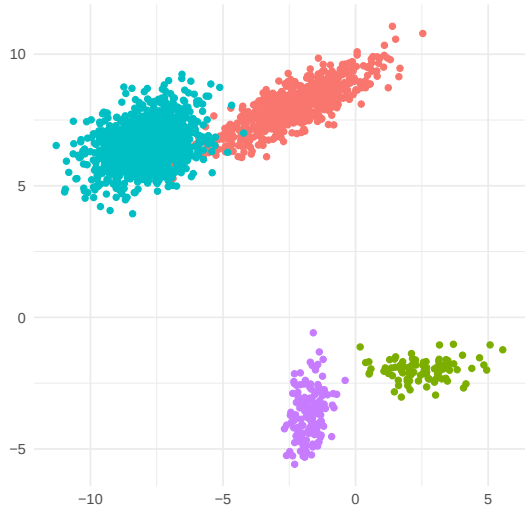


Figure 1: Example GMM data in $\mathbb{R}^2$ with 4 Gaussian components

Formally, the probability density of a vector x under this model is given by

$$p(x) = \sum_{j=1}^{K} \pi_j \mathcal{N}(x|\mu_j, \Sigma_j)$$

$$\sum_{j=1}^{K} \pi_j = 1, \quad 0 \leq \pi_j \leq 1 \quad \forall j = 1, ..., k$$

$$\mathcal{N}(x|\mu, \Sigma) = \frac{1}{\sqrt{(2\pi)^m |\Sigma_j|}} \exp\left(-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)\right)$$

where π_j is the expected proportion of the data that is generated by the j-th component, whose mean and covariance matrix are μ_j and Σ_j, respectively.

Usually, this model is treated as a *latent variable model*; as such, a set of unobserved variables determines what we observe, and said variables are estimated as well, though

indirectly, through the observed data. Using the notation from [Bishop (2006)], let z_i be a K-dimensional binary random vector that represents the particular Gaussian component x_i comes from, this is, $z_{im} = 1$ if and only if x_i was generated by the m-th Gaussian; the vector z_i entries are the latent variables for this model; note that $\pi_m = P(z_m = 1)$. The purpose of the model is to estimate the parameters $\{\pi_j\}$, $\{\mu_j\}$, $\{\Sigma_j\}$ of such distributions, thereby giving us a description of the different subpopulations that are present in the data.

2.1.1 EM algorithm for Gaussian Mixtures

As a latent variable model, GMMs are usually fitted using *Expectation-Maximization* (EM), a very general algorithm (see [Bishop (2006)]) for latent variable models which in this case is similar to the K-means algorithm, so much as to be considered a "soft" version of it, since assignments of points to clusters are not binary but probabilistic. Before we show the EM algorithm, let us assume we have n data points, let K be the number of Gaussian components, and define

$$\gamma(z_{ik}) = P(z_{ik} = 1|x_i) = \frac{\pi_j \mathcal{N}(x_i|\mu_j, \Sigma_j)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(x_i|\mu_j, \Sigma_j)} \tag{1}$$

This is, $\gamma(z_{ik})$ is the probability that an observed point x_i proceeds from component k. The EM algorithm improves the model iteratively by maximising the model's log-likelihood, which is given by

$$L(X; \{\mu_j\}, \{\Sigma_j\}, \{\pi_j\}) = \ln p(X|\pi, \mu, \Sigma) = \sum_{i=1}^{n} \ln \left(\sum_{j=1}^{K} \pi_j \mathcal{N}(x_i|\mu_j, \Sigma_j) \right) \tag{2}$$

The EM algorithm for GMMs can be summarised as follows:

Algorithm 1: Expectation-Maximization for Gaussian Mixtures

Data: points $\{x_1, ..., x_n\}$, number of clusters K
Result: Estimated parameters $\{\hat{\pi}_j\}$, $\{\hat{\mu}_j\}$, $\{\hat{\Sigma}_j\}$

Initialize parameters $\{\hat{\pi}_j\}$, $\{\hat{\mu}_j\}$, $\{\hat{\Sigma}_j\}$ and evaluate $L(X)$;
while *not converged* **do**
  Calculate $\gamma(z_{ij})$ with eq. 1 $\forall\ i = 1, ..., n,\ j = 1, ..., K$; // Expectation step
  $N_j \leftarrow \sum_{i=1}^{n} \gamma(z_{ij}), j = 1, ..., K$
  $\hat{\mu}_j \leftarrow \frac{1}{N_j} \sum_{i=1}^{n} \gamma(z_{ij}) x_i, j = 1, ..., K$; // Maximization step begin here
  $\hat{\Sigma}_j \leftarrow \frac{1}{N_j} \sum_{i=1}^{n} \gamma(z_{ij})(x_i - \hat{\mu}_j)(x_i - \hat{\mu}_j)^T, j = 1, ..., K$
  $\hat{\pi}_j \leftarrow \frac{N_j}{n}, j = 1, ..., K$
  evaluate $L(X)$ with eq. 2 and check for convergence
return $\{\hat{\pi}_j\}$, $\{\hat{\mu}_j\}$, $\{\hat{\Sigma}_j\}$

A more in-depth analysis of such models can be found at [Bishop (2006)].

2.1.2 Model Regularisation

A known issue that arises sometimes when fitting GMMs is *covariance singularity*, which happens when one of the component means equals one of the data points; in this case, the loglikelihood can grow unbounded as the covariance matrix is shrinked to singularity, effectively creating a single-point cluster with zero variance. This can be fixed adding a regularisation term to the loss function (i.e., the log-likelihood) that removes such singularities; in this way we optimise instead a function of the form $L(X;\theta) + r(\theta)$, where θ represents the model parameters.

We can do this placing prior distributions over the parameters, and conjugate priors are a popular choice to do this because they allow the posterior distributions to be analytically tractable. For a GMM model these are given by

$$
\begin{aligned}
p(\pi_1, ..., \pi_K) \sim &\text{Dirichlet}(\alpha) \propto \prod_{j=1}^{K} \pi_j^{\alpha-1} \\
p(\mu_j|\lambda, \Sigma_j) \sim &\text{Normal}(\lambda, \Sigma_j) \propto \frac{1}{|\Sigma_j|}\text{exp}(-\frac{1}{2}(\mu_j - \lambda)^T\Sigma_j^{-1}(\mu_j - \lambda)) \\
p(\Sigma_j|\Lambda, \nu) \sim &\text{Inverse-Wishart}(\Lambda, \nu) \propto |\Sigma_j|^{-\frac{1}{2}(\nu+d+1)}\text{exp}(\text{tr}(\Lambda\Sigma_j^{-1})
\end{aligned}
$$

After we choose suitable hyperparameters $\alpha, \lambda, \Lambda, \nu$, we then proceed to fit the model maximizing $L(X) + r(\theta)$ where $r(\theta)$ is the prior's log-likelihood; this is equivalent to performing Bayesian estimation and using the mode of the resulting posterior distributions as the estimated parameters. Those are sometimes called *maximum a posteriori* (MAP) estimates.

2.2 Numerical Optimization

Once we reformulate the GMM optimisation problem in the context of Riemannian Optimisation (we will talk about this in subsection 2.3.1), we will be able to use simple Stochastic Gradient Descent (SGD) to fit them, and this is convinient for two main reasons: the first one is that this allows large-scale mixture modelling, making it feasible to fit models with very large datasets, and the second one is that it has the added benefit of allowing us to plug in any of the different acceleration techniques for SGD that have been developed and used in the last few years and that have been used in Machine Learning and particularly Deep Learning applications with very good results. We present below a review of the basic SGD and also of three well-known acceleration

techniques: Momentum, Nesterov's correction and ADAM, which uses momentum and also provides a dynamic parameterwise step size.

For the particular problem of GMM we aim to maximise the log-likelihood function, so technically we want to perform gradient *ascent* instead of descent; however, in the literature there is the convention of talking about descent, as in SGD; for the sake of consistency we will follow that convention, only noting that we actually perform gradient ascent in our implementation.

2.2.1 Stochastic Gradient Descent

SGD is a very simple optimisation algorithm that find local extrema of differentiable functions. In the context of Machine Learning, we usually want to find a local minima with respect to the parameters θ for a loss function and a given training dataset. Let $L(X;\theta)$ be a differentiable loss function parametrized by θ and $X = \{x_1, ...x_n\}$ a set of data points, then stochastic gradient descent can be performed as follows:

Algorithm 2: Stochastig Gradient Descent

Data: points $\{x_1, ..., x_n\}$, batch size k, step size α_0
Result: estimated local minimum $\hat{\theta}^* \in \mathbb{R}^n$

set initial guess θ_0, $t = 0$
while *not converged* **do**
    select random sample $X_s = \{x_{s1}, ..., x_{sk}\}$
    $\theta_{t+1} \leftarrow \theta_t - \alpha_t \nabla L(X_s; \theta_t)$; // α_t can shrink or stay constant
    $t \leftarrow t + 1$
return θ_t

We can also replace $\nabla_\theta L(X_s;\theta)$ for any other descent direction; some of them use the gradient as the basic direction but add a correction term or some other modification, and they can sometimes perform much better than typical SGD. We will call these *accelerated gradient descent methods*, and will review some of them below. SGD has become very popular in the last few years because of it simplicity and its ability to deal with massive datasets, making the optimisation problem agnostic to the data volume. On the one hand, it is a first order method, which means it does not need to store Hessian matrices, and on the other, there are only a few quite intuitive parameters that need to be chosen: when $k = n$ we have batch gradient descent; when $k = 1$ we have online learning, and when k is in between we have mini-batch stochastic gradient descent; the case of α is also intuitive but it has more complicated consequences: it is the learning rate and controls how large each step is, but a large α, while making the descent faster at first, could also cause instability.

To see why, note that for the mini-batch and online case we are effectively using a noisy estimate $\nabla L(X_s;\theta)$ of the true descent direction $\nabla L(X;\theta)$ as the descent direction; our estimate is a random variable and its bias and variance determine the behaviour of the optimisation procedure; In the particular case of non-convex functions like the GMM

loss function and under relatively mild assumptions over $L(X;\theta)$ and its gradient, it can be proven [Bottou et al. (2016)] that there is an interplay between the step size α_t, the convergence speed, and the convergence accuracy, namely, that as α_t becomes smaller, so does the average norm of the gradients (this is, the procedure spends more time closer and closer to the local optimum), but it also reduces the speed at which the gradient reaches its limiting distribution[Bottou et al. (2016)]. The presence of noise in the estimates prevent us from reaching the true optimum and we can only get asymptotically close by shrinking the step size, but this in turn means it will take more time, so a good step size is one that finds a compromise between those two notions.

Results for diminishing step size sequences also exists and it can also be shown that they have been show to have some nice theoretical properties [Bottou et al. (2016)].

2.2.2 Stochastic Gradient Descent With Momentum

One of the most popular modifications to simple SGD is adding a *momentum* to the descent direction:

Algorithm 3: Gradient Descent With Momentum

Data: initial parameter θ_0, momentum $\beta \in (0,1)$, step size α_0
Result: estimated local minimum $\hat{\theta}^* \in \mathbb{R}^n$

set initial guess θ_0, $t = 0$
$g_0 \leftarrow 0$
while *not converged* **do**
    $g_{t+1} \leftarrow \beta g_t + \nabla f(\theta_t)$
    $\theta_{t+1} \leftarrow \theta_t - \alpha g_{t+1}$
    $t \leftarrow t + 1$
return θ_t

Intuitively, this means that the descent procedure will behave like a ball going down a hill, catching inertia as it moves, and this is why it is sometimes also called the *heavyball* method; this smooths the path to the solution, boosting consistent descent directions and shrinking noisy ones, and helps avoid some of the problems normal SGD run into when dealing with poorly scaled functions; it can also be shown to give a quadratic speedup for some functions, and although it has nice theoretical properties [Loizou and Richtárik (2017)], in practice its convergence is not guaranteed even for convex functions [Goh (2017)]; that being said, it usually gives good resultswhen properly tuned[Sutskever et al. (2013)].

2.2.3 Nesterov Accelerated Gradient

Another acceleration method that has become very popular in recent years is *Nesterov Accelerated Gradient*, which behaves similarly to gradient descent with momentum, but is not the same.

This descent algorithm have been shown to have very good convergence properties, in the sense that for some convex, smooth functions, its convergence rate with respect to the number of iterations t is $\mathcal{O}(t^{-2})$, which is faster than the one achievable with simple SGD, which is $\mathcal{O}(t^{-1})$. In fact, it can be proven that quadratic convergence is the fastest possible rate for first-order methods, so in that sense, Nesterov's descent is optimal [Beck and Teboulle (2009)].

Algorithm 4: Nesterov Accelerated Gradient direction

Data: initial parameter θ_0, momentum $\gamma \in (0, 1)$, step size α_0
Result: estimated local minimum $\hat{\theta}^* \in \mathbb{R}^n$

1 set initial guess θ_0, $t = 0$
2 $y_0 \leftarrow \theta_0$
3 **while** *not converged* **do**
4 $\quad y_{t+1} \leftarrow \theta_t - \alpha \nabla f(\theta_t)$
5 $\quad \theta_{t+1} \leftarrow \theta_t + \gamma(y_{t+1} - y_t)$
6 $\quad t \leftarrow t + 1$
7 **return** θ_t

2.2.4 ADAM and ADAMAX

Another couple methods that add a correction direction to basic gradient descent and that additionally provide dynamic stepsizes for each parameter individually are ADAM and ADAMAX (from *Adaptive Moment Estimation*) [Kingma and Ba (2014)]. ADAM was first proposed for deep learning applications, but it can be plugged in in any SGD procedure as the descent direction; its distinctive characteristic is providing an adaptive step size for each parameter, depending of a notion of signal-to-noise ratio of the estimated gradient history.

Intuitively, it computes exponential moving averages of the first and second moments of the descent direction (recall that in SGD, such directions are noisy estimates of the true steepest descent direction, so they are essentially random variables), and in each iteration the step size is calculated as a function to signal-to-noise ratio of such averages. Both moving averages `m` and `v` for the first and second moment respectively are initialised as 0.

In the pseudocode below, ε is a constant added to avoid division by zero, and the lines 4 and 5 are meant to remove the bias on the averages introduced by initialising `m` and `v` as zero. The authors show that once the descent direction is computed, the step size α_t of the optimisation procedure acts effectively as an upper bound on the magnitude of the dynamic step sizes.

A tweak of ADAM proposed in the same paper is ADAMAX, where instead of using an

L^2 norm when rescaling the individual parameter gradients, the use the infinity norm.

Algorithm 5: ADAM

Data: decay rates $\beta_1, \beta_2 \in [0, 1)$, function f, sample X_s, $\varepsilon > 0$
Result: estimated local minimum $\hat{\theta}^* \in \mathbb{R}^n$

$t \leftarrow 0$
$m \leftarrow 0$
$v \leftarrow 0$
while *not converged* **do**
  $t \leftarrow t + 1$
  $g \leftarrow \nabla_\theta f(X_s; \theta)$
  $m \leftarrow \beta_1 m + (1 - \beta_1) g$
  $v \leftarrow \beta_2 v + (1 - \beta_2) g^2$; // Here g^2 means element-wise square
  $m \leftarrow m / (1 - \beta_1)^t$
  $v \leftarrow v / (1 - \beta_2)^t$
  $\theta_{t+1} \leftarrow \theta_t - \alpha m / (\sqrt{v} + \varepsilon)$
return θ_t

Algorithm 6: ADAMAX

Data: decay rates $\beta_1, \beta_2 \in [0, 1)$, function f, sample X_s, $\varepsilon > 0$
Result: estimated local minimum $\hat{\theta}^* \in \mathbb{R}^n$

$t \leftarrow 0$
$m \leftarrow 0$
$v \leftarrow 0$
while *not converged* **do**
  $t \leftarrow t + 1$
  $g \leftarrow \nabla_\theta f(X_s; \theta)$
  $m \leftarrow \beta_1 m + (1 - \beta_1) g$
  $v \leftarrow \max(\beta_2 v, |g|)$
  $\theta_{t+1} \leftarrow \theta_t - \alpha m / ((1 - \beta_1^t) v)$
return θ_t

2.3 Riemannian Optimisation

The main problem with using gradient-based methods like SGD to fit a GMM is that the covariance matrices need to remain symmetric positive deffinite (SPD) throughout the optimisation process, and such algorithms cannot enforce this. Also, as is pointed out in [Hosseini and Sra (2017)], gradient-based methods for constrained problems, such as interior points methods, have been tried before for closely related problems, but they turned out to be much slower than EM-like algorithms [Sra and Hosseini (2013)].

Fortunately, it turns out that the set of SPD matrices form a mathematical structure called manifold(which is a structure that locally resembles a Euclidean space), and this

means that we can formulate the problem in the context of Riemannian Optimisation (RO), a branch of optimisation that deals with finding optima when the constraint sets are manifolds, like in this case. Doing this will in turn allow us to successfully use gradient descent methods to fit GMMs.

2.3.1 Manifold of SPD Matrices

There is already a great deal of literature dealing specifically with matrix manifolds (see for example [Absil et al. (2007)]), and in this work we focus on using only first-order methods on the manifold of positive definite (SPD) matrices $\mathbb{P}$; the results we show next are all taken from [Hosseini and Sra (2017)] .

Assume we have a differentiable function $f : \mathbb{P} \mapsto \mathbb{R}$ that we want to optimise. Each element of the manifold is a SPD matrix X, and each one of them has a tangent space T_X(which is an approximating vector space at a given point in the manifold, and represent the same concept as a tangent plane in Euclidean space) isomorphic to the set of symmetric matrices; we also endow the space with the following Riemannian metric

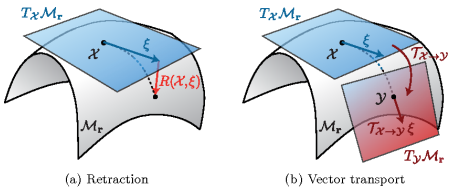
$$g_X(A, B) = tr(\Sigma^{-1} A \Sigma^{-1} B) \tag{3}$$

where A, B are elements in the tangent space of X. This is effectively the dot product between two vectors on the tangent space of X, T_X.

Performing SGD on a manifold involves two steps:

1. Find a descent direction in the tangent space of the current parameter value
2. Take a step along it and then project the resulting update back to the manifold through an operation called *retraction*

Let $X \in \mathbb{P}$, $A \in T_X$. The Riemannian metric implies that the gradient of $f(X)$ on T_X is then given by



(a) Retraction (b) Vector transport

Figure 2: Basic steps of Riemannian optimisation [Kressner et al. (2014)]

$$\nabla f(X) = \frac{1}{2} X(\nabla_E f(X) + [\nabla_E f(X)]^T)X \tag{4}$$

where $\nabla_E f(X)$ is the Euclidean gradient of $f(X)$, i.e., the usual gradient. There are many alternatives for the retraction, one of which the authors suggest is the exponential retraction

$$R_X(A) = X \exp(X^{-1} A) \tag{5}$$

A much cheaper alternative from a computational point of view is the *Euclidean retraction*

$$R_X(A) = X + A \tag{6}$$

The following pseudocode summarises this process

Algorithm 7: SGD over the manifold of SPD matrices

Data: points $\{x_1, ..., x_n\}$, batch size k, step size α_0
Result: estimated local minimum $\hat{\Theta}^* \in \mathbb{R}^n$

set initial guess Θ_0, $t = 0$
while *not converged* **do**
    select random sample $X_s = \{x_{s1}, ..., x_{sk}\}$
    Calculate descent direction $g(X_s; \Theta_t)$ using eq. 4
    $\Theta_{t+1} \leftarrow R_{\Theta_t}(\alpha_t g_t)$ using eq. 5 or 6 ; // α_t can shrink or stay constant
    check convergence with $\|\Theta_{t+1} - \Theta_t\|$
    $t \leftarrow t + 1$
return Θ_t

An important concept in Manifold optimisation is that of *geodesic convexity* or *g-convexity*. First we need a few definitions. A *geodesic* function between two points $X_1, X_2 \in \mathbb{P}$ is defined as

$$\gamma_{X1,X2} : [0, 1] \mapsto \mathbb{P}, \quad \gamma_{X1,X2}(0) = X_1, \quad \gamma_{X1,X2}(1) = X_2$$

A *geodesically convex* set $\mathcal{A}$ in $\mathbb{P}$ is one in which any two points X_1, X_2 have a minimising geodesic (a geodesic that contains the shortest path as measured by the Riemannian metric) completely contained in $\mathcal{A}$.

Formally, a function $f : \mathcal{A} \mapsto \mathbb{R}$ is said to be g-concave if it satisfies

$$f(\gamma_{X1,X2}(t)) \leq (1 - t)f(X_1) + tf(X_2), \quad t \in [0, 1], X_1, X_2 \in \mathbb{P}^d \tag{7}$$

the negative of a g-convex function is called g-concave. Since we intent to maximise the model's log-likelihood we are mostly interested in g-concavity, and it is important because a g-concave function have additional desirable properties, and can result in better performance[Hosseini and Sra (2017)].

3 Problem Formulation

In the following section we review the results from [Hosseini and Sra (2017)] needed to derive the reformulation we base our algorithms on. The main problem with the classical formulation of GMM when seen from the point of view of Riemannian Optimisation is that the loss function in eq. 2 is not g-concave (a form of concavity in manifolds; see section 2.3.1), which may slow down convergence. The main purpose of the reformulation is deriving a g-concave loss function that preserves the solution of the original formulation.

3.1 G-Concave Loss Function

Let $x_1, ..., x_n$ be a set of observations from a GMM with K components and define the following transformations for the data and parameters

$$y_i = (x_i \;\; 1)^T \qquad \forall i = 1, ..., n \tag{8}$$

$$S_j = \begin{bmatrix} \Sigma_j + s\mu_j\mu_j^T & s\mu_j \\ s\mu_j^T & s \end{bmatrix} \in \mathbb{R}^{(d+1)\times(d+1)}, \quad s > 0 \qquad \forall j = 1, ..., K \tag{9}$$

Let us also define the following formula

$$q_{\mathcal{N}}(y_i; S_j) = \sqrt{2\pi}\exp\left(\frac{1}{2}\right)\mathcal{N}(y_i; 0, S_j) \tag{10}$$

It turns out that replacing $\mathcal{N}(x_i|\mu_j, \Sigma_j)$ by equation 10 in the log-likelihood loss (equation 2) makes the loss function g-concave on the manifold of $(d+1) \times (d+1)$ SPD matrices; furthermore, equation 10 coincides with $\mathcal{N}(x_i|\mu_j, \Sigma_j)$ when $s = 1$, so if the modified loss function's solution has its maximum at $s^* = 1$, we will also recover the true model's log-likelihood, and hence the original problem's solution. We will see below that this is indeed the case.

The second problem of equation 2 from a gradient descent perspective is the constraint over the weights, since they have to remain in a simplex set (i.e., $\sum_{j=1}^{K} \pi_j = 1, \pi_j \geq 0 \;\; \forall$
$; j = 1, ...K$). To fix this, let us define the following mapping between the weights and some real-valued variables $\omega_j, \quad j = 1, ..., K$

$$\omega_j = \ln\left(\frac{\pi_j}{\pi_K}\right) \;\; \forall \;\; j = 1, ..., K-1 \tag{11}$$

$$\omega_k = 0 \tag{12}$$

Then, the new g-concave loss function is

$$\underset{\{S_j\},\{\omega_j\}}{\text{argmax}}\,\hat{\mathcal{L}}(Y;\{S_j\},\{\omega_j\}) = \underset{\{S_j\},\{\omega_j\}}{\text{argmax}}\frac{1}{n}\sum_{i=1}^{n}\ln\left(\sum_{j=1}^{K}\frac{\exp(\omega_j)}{\sum_{m=1}^{K}\exp(\omega_m)}q_{\mathcal{N}}(y_i;S_j)\right) \tag{13}$$

Where we have used the fact that the weights $\{\pi_j\}$ are related to $\{\omega_j\}$ by the Softmax function

$$\pi_j = \frac{\exp(\omega_j)}{\sum_{j=1}^{K}\exp(\omega_j)}, \quad j = 1,...K \tag{14}$$

The following lemma, which is proven in A, states that from the solution of this reformulation we can straightforwardly recover the solution of the original problem.

Lemma 3.1. *Let $X = \{x_1,...x_n\}$ and let $\{\mu_j^*\}$, $\{\Sigma_j^*\}$, $\{\pi_j^*\}$ be the maximisers of $L(X,\{\mu_j\},\{\Sigma_j\},\{\pi_j\})$ in eq. 2. Then the maximisers $\{S_j^*\}$, $\{\omega_j^*\}$ of $\hat{\mathcal{L}}(Y;\{S_j\},\{\omega_j\})$ are such that*

$$S_j^* = \begin{bmatrix}\Sigma_j^* + \mu_j^*\mu_j^{*T} & \mu_j^* \\ \mu_j^{*T} & 1\end{bmatrix} \quad \forall \;\; j = 1,...,K \tag{15}$$

$$\omega_j^* = ln\left(\frac{\pi_j^*}{\pi_K^*}\right)\forall \;\; j = 1,...,K-1 \tag{16}$$

$$\omega_k^* = 0 \tag{17}$$

Let $\mathbb{P}^d$ be the manifold of $d \times d$ SPD matrices. By using equation 8 to wrap both Gaussian parameters in a single matrix, our new formulation is an optimisation problem on $\left(\prod_{i=1}^{K}\mathbb{P}^{d+1}\times\mathbb{R}^{K-1}\right)$, since we have K $(d+1)\times(d+1)$ matrices and $K-1$ free weight parameters.

3.1.1 Conjugate Prior Regularisation

As was mentioned in section 2.1.2, to prevent the problem of covariance singularity, we can add a regularisation term to the loss function; In [Hosseini and Sra (2017)] expressions for a conjugate prior under the new formulation are derived using equations 8 - 16. The prior distributions are

$$
\begin{aligned}
p(\pi_1, ..., \pi_K) &\sim \text{Dirichlet}(\zeta) \\
p(\Sigma_j|\Lambda, \nu) &\sim \text{Inverse-Wishart}(\Lambda, \nu) && \forall\ j = 1, ...K \\
p(\mu_j|\lambda, \Sigma_j) &\sim \text{Normal}(0, (\nu + d + 1)^{-1}\Sigma_j) && \forall\ j = 1, ...K \\
p(s^{-1}) &\sim \text{Gamma}\left(\frac{\nu + d + 1}{2}, \frac{\nu + d + 1}{2} + 1\right)
\end{aligned}
$$

where d is the data dimensionality. In [Hosseini and Sra (2017)] it is shown that the corresponding regularisation term can be expresed as

$$\eta(S_j, \Psi) = -\frac{1}{2}(\nu + d + 1)\log\det(S_j) - \frac{1}{2}\text{tr}(\Psi S_j^{-1}) \tag{18}$$

where

$$\Psi = \begin{bmatrix} \Lambda + \kappa\lambda\lambda^T & \kappa\lambda \\ \kappa\lambda^T & \kappa \end{bmatrix}, \kappa = \nu + d + 2 \tag{19}$$

Using equations 11 and 14, the weight regularisation term becomes

$$\psi(\{\omega_j\}) = \ln p(\pi_1, ..., \pi_K) = \alpha \sum_{j=1}^{K} \omega_j - K \ln\left(\sum_{k=1}^{K} \exp(\omega_k)\right) \tag{20}$$

The new optimisation problem is

$$\underset{\{S_j\},\{\omega_j\}}{\text{argmax}}\, \hat{\mathcal{L}}(Y; \{S_j\}, \{\omega_j\}) + \psi(\{\omega_j\}) + \sum_{j=1}^{K} \eta(S_j, \Psi) \tag{21}$$

The authors go on to show that eq. 21 is also g-concave and preserve the solution of the original formulation regularised by a conjugate prior.

3.1.2 Logarithmic Barrier Regularisation

A disadvantage of using a conjugate prior distribution, is that we need to store an additional matrix and do a $\mathcal{O}(d^2)$ operation to compute the trace of a matrix product on every iteration. A more efficient alternative to keep the covariances from becoming singular may be using a logarithmic barrier on $\det(S_j)$.

In appendix B.1 we show that the following loss function

$$\underset{\{S_j\},\{\omega_j\}}{\text{argmax}}\, \hat{\mathcal{L}}(Y;\{S_j\},\{\omega_j\}) + \sum_{j=1}^{K} (\ln\det(S_j) - s) \tag{22}$$

prevents covariance singularities and preserves the solution of adding a logarithmic barrier on $\det(\Sigma_j)$ to the original log-likelihood loss (eq. 2). Unlike a conjugate prior, this regularisation only penalises the matrices Σ_j, so the weights and mean vectors are unaffected.

Logarithmic barriers are used in constrained optimisation problems as a way to include the constraints directly in the objective function (see [Nocedal and Wright (2006)]), and usually the barrier term is shrunk to zero as more iterations go by to recover the problem's original solution. Here we are using it to avoid covariance singularities (see B.2) for as long as the model is updated, so we are not able to recover the exact solution the the original, unregularised problem, although this difference is not significant, as we will see in the next section.

This regularisation does not need to store additional matrices, so is more efficient in terms of memory than a conjugate prior, as well as in CPU time: since we calculate the determinant of the updated matrices S_j in every iteration to evaluate the updated probability densities, evaluating this regularisation term is effectively free.

3.1.3 Loss Function Gradients

Once we have defined the loss function of our model, we need to find the gradients with respect to S_j and ω_j, $\quad j = 1, ..., K$, to fit it. Using eq. 4 we can derive the loss function gradients over the manifold of SPD matrices; for a given S_j and ω_j define

$$w_i = \frac{\pi_i q_{\mathcal{N}}(y_i; S_j)}{\sum_{j=1}^{K} \pi_i q_{\mathcal{N}}(y_i; S_j)} \tag{23}$$

Term	$\nabla_{S_j}\hat{\mathcal{L}}(Y;\{S_j\},\{\omega_j\})$	$\nabla_{\omega_j}\hat{\mathcal{L}}(Y;\{S_j\},\{\omega_j\})$
Loss function	$\frac{1}{2n}\sum_{i=1}^{n} w_i \left(y_i y_i^T - S_j\right)$	$\left(\frac{1}{n}\sum_{i=1}^{n} w_j\right) - \pi_j$
Conjugate prior	$\frac{1}{2n}\left(\Psi - \nu S_j\right)$	$\frac{\zeta}{n}\left(1 - K\pi_j\right)$
Logarithmic barrier	$\frac{1}{2n}\left(S_j - s^2 \hat{\mu}_j \hat{\mu}_j{}^T\right), \hat{\mu}_j = [\mu_j\ 1]$	0

Table 1: Expressions for the loss function gradients

Note that w_i is analogous to eq. 1 in the new formulation. The following table summarises the gradients of S_j under the proposed loss function and its possible modifications:

4 Program Design

In this section we make a detailed explanation of the choices we have made for the implementation of the model; the purpose of the implementation was twofold: first, making the program suitable for sequential and distributed processing and for streaming data, and secondly, writing a decoupled program that makes maintainability and extensibility as easy as possible, particularly when adding/tuning optimisation options such as algorithms or regularisers. A significant amount of time and effort was invested in making a robust, properly documented, high-quality software project that is easy to use or extend by anyone interested in it, and to make it as similar as possible to existing Spark objects and methods.

The implementation is written in `scala 2.11.8` and is designed to run on Apache Spark (https://spark.apache.org/), an open-source distributed computing platform that runs on commodity hardware; it is a very popular framework in so-called big data applications and has a big user base, which ensures that the resulting program will have high availability. From the many programming languages supported by Spark, `scala` has the most functionalities available (it is Spark's main development language), in particular for Spark's `streaming` library. A sequential implementation is also available and can be run outside Spark; it takes Breeze's `DenseVectors` as input.

The package's full documentation is in

https://nestorsag.github.io/streaming-gmm/index.html#package

and the source code can be found in

https://github.com/nestorSag/streaming-gmm

4.1 Main Dependencies

Some of the libraries that the program uses are:

- `MLlib` (https://spark.apache.org/mllib/) This Spark library contains many Machine Learning related classes that served as inspiration for the program, such as `MultivariateGaussian` and `GaussianMixture`. This was done to avoid reinventing the wheel but also to preserve some compatibility with how classes and methods are generally operated in Spark, which would be useful for experienced Spark users.
- `Breeze` (https://github.com/scalanlp/breeze) is a numerical processing library that was used to perform all of the heavy numeric routines, specially the Linear Algebra ones. It is also used by Spark's `MLlib` internally

- `Log4j` (https://logging.apache.org/log4j/2.x/) was used as the main logging library. Logging in general is not used frequently accross the program, only for a few important warnings and to log the different parameter values through time when needed.
- `ScalaTest` (http://www.scalatest.org/) was used for unit testing.

4.2 Program Structure

The program is divided in 3 main parts. Below is a rough description of them.

4.2.1 `components`

This subpackage contains the basic units of the program: classes that wrap the updatable parameters (the weights vector and Gaussian parameters) and basic utilities (array wrappers) to perform accelerated gradient descent. The most visible classes are

- `UpdatableWeights`, which wraps the weigths array and its accelerated gradient utilities; it also checks that the weights fulfill the model's constraint at each update.
- `UpdatableGaussianComponent`,which wraps a `MultivariateGaussian` instance together with objects necessary for the descent updates. It implements many different constructors, including one that takes a Spark's `MultivariateGaussian`.

The project's `MultivariateGaussian` class is heavily based on Spark's. Initially the idea was to simply extend the Spark's version, but the access modifiers on `MLlib`'s code, and on some other Spark classes, made it very difficult to work that way, and it was decided that it was simpler to redo a few classes replicating the functionality.

4.2.2 `optim`

This subpackage contains all classes related to the actual optimisation procedure's iterations: classes that contain the logic to transform the weights to and from the simplex set, regularisation terms and optimisation algorithms. Its functionality is divided in 3 different kinds of objects, each of them having at least two implementations:

- `Optimizer` wraps all the optimisation logic and utilities, including regularisation, weight mapping, hyperparameter setters and getters and so on. Its most distinctive functionality is calculating the actual descent direction; at the moment it has 5 different implementations:
  - `GradientAscent`

  - `MomentumGradientAscent`
  - `NesterovGradientAscent`
  - `ADAM`
  - `ADAMAX`
- `Regularizer` represent regularising terms added to the loss functions and contains logic for its evaluation and for getting its gradients; at the moment it has two implementations:
  - `ConjugatePrior`
  - `LogBarrier`[1]
- `WeightsTransformation` represents different possible mapping between the model's weights and real parameters that can be optimised with descent methods. At the moment it has two implementations
  - `SoftmaxWeightsTransformation`, as in eq. 14
  - `RatioWeightsTransformation`

`RatioWeightsTransformation` was added as an experimental alternative to see if faster convergence was possible using different transformations. The transformation used instead is

$$z_i = \frac{\pi_i}{\pi_K} \tag{24}$$

$$\pi_i = \frac{z_i}{\sum_{j=1}^{K} z_i} \tag{25}$$

the new variables, z_i, are constrained to be positive.

As a side note, abstracting away the loss function as a class in itself was also considered but it was decided it was not worth it, since it would make the program a bit less understandable at some parts and because there is no need for it since, to the best of our knowledge, there is not a single alternative to log-likelihood as the objective function in a Gaussian Mixture Model. That being said, in the current state of the program it could be done with relatively little effort.

4.2.3 `model`

Contains the `GradientGaussianMixture` class, which is the main class and entry point in our program. It also contains the `MetricAggregator` class that reduce the data in the workers to compute gradients and the data log-likelihood.

[1] Even though it is not technically a regulariser, in this case a log barrier can be used to fulfil the same goal as the conjugate prior regularisation, which is avoiding covariance singularities.

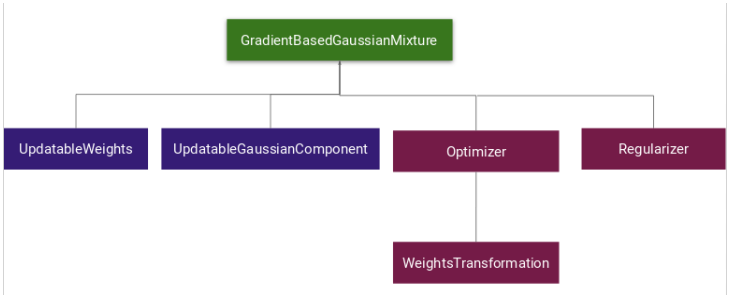


Figure 3: Class embeding structure coloured by functionality

4.3 Data Processing

The program is aimed at applications with high data availability and relatively low dimensionality. As a reference point, we are thinking in streaming applications whose data dimensionality do not exceed a few hundreds; in any case, due to the curse of dimensionality performing clustering in extremely high-dimensional data is not recommended, and it is generally a good idea to perform some kind of dimensionality or preprocessing reduction first.

4.3.1 Distributed Processing Patterns

The processing pattern is composed by two parts: in the first one, the distributed data passes through the current model (which has to be sent from the driver to the workers) and the gradients for all of the parameters are computed and aggregated for each data point. Finally, the resulting gradients are sent to the driver. In the second stage, gradients and Gaussian components are paired up and distributed among the workers to be updated; this is because, unlike most of the gradient-based models out there, this one involves inverting the covariance matrices each time it is updated, which can be expensive computationally, and this is why the model's components are sent to the workers to be updated in parallel. The complete process is illustrated in the following diagram.

Inverting a matrix has a complexity of order $\mathcal{O}(d^3)$, where d is the matrix dimension. The complexity of computing the gradient (see table 1) for a single data point is of order $\mathcal{O}(d^2)$, so in computational terms, the additional matrix inversion step for a GMM with K components is equivalent to processing $\mathcal{O}(Kd)$ more data points per iteration, which for dimensionalities of up a few hundred can be still negligible in applications with high data availability; those are the uses cases that most interests us.

Overall, this process follows the same distributing patterns as `MLlib`'s `GaussianMixture`, although the calculations for each data point are different.

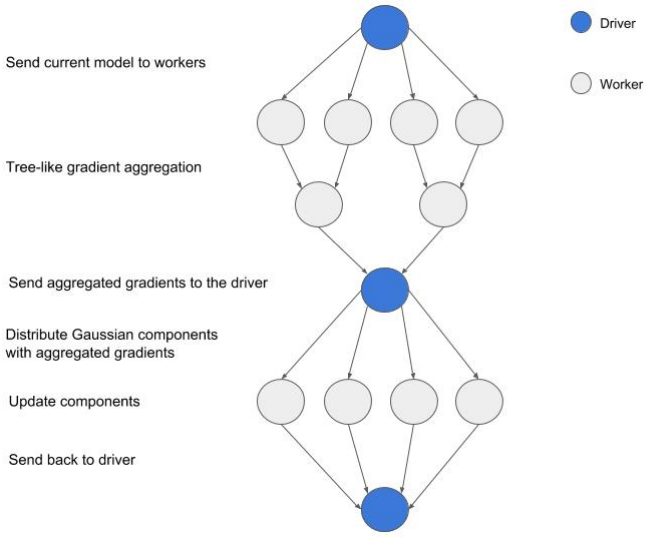


Figure 4: Data processing pattern

4.3.2 Sequential Processing Patterns

The program can also process Breeze's `DenseVector` data in a sequential manner on a single machine, outside Spark. The only difference between the sequential and distributed processing logic is that for the former the data is shuffled and partitioned in batches each time a new epoch is started. After that, the data processing patterns are analogous, besides doing sequentially what would otherwise be done by many different worker machines.

4.3.3 Memory Requirements

For data points in $\mathbb{R}^d$ and K clusters, the memory requirements of a GMM is of order $\mathcal{O}(Kd^2)$ because for each cluster it needs to store the corresponding covariance matrix. For our program, each `UpdatableGaussianComponent` stores at least two $d \times d$ matrices and one $(d+1) \times (d+1)$ matrix at any time: The covariance matrix, its inverse's square root, and the parameter matrix as defined in 9; although it is not strictly necessary to permanently store the last one, being just a rearrangement of the mean and covariance parameters, this saves computation time). In addition to this, accelerated gradient descent procedures (like ADAM) can use up to two more $(d+1) \times (d+1)$ matrices to store historic information about past gradients. With all this in mind, the maximum memory usage happens at the master machine just after the per-point gradients (which is, one more matrix per cluster) have been aggregated and sent to it. At this moment, the driver would need to hold a bit more than $6Kd^2$ `Double` values. This is fine when the dimensionality is not high, but would take almost 6MB of memory per cluster with a 1000-dimensional data set.

The cheapest scenario memory-wise, would be to use simple Gradient Ascent, and in

this case we would have a bit over $4Kd^2$ `Double` values in the driver at its maximum. As a reference, Spark's `GaussianMixture` holds $3Kd^2$ values at the driver at its maximum.

4.4 Other Design Decisions

4.4.1 Updating the Weights

In our program, updating the weights is a separate process and is all done in the driver; this is because updating them in parallel together wit their corresponding Gaussian distribution would have make the code less clear, because of the dependence of the transformations to and from the simplex set on the last weight π_K (see equations 16 and 24), which is rather asymmetric and makes the code more cumbersome. Besides, once the aggregated weights gradients are sent to the driver, updating the weights vector is extremely cheap, so this does not affect performance.

4.4.2 Numerical Stability

The program should not suffer from numerical instability in the Gaussian parameters since the form of the gradients ensures that all values are on the same scale, and they don't involve any nonlinear transformation that can add instability. The weights on the other hand involve an exponential function when projecting the auxiliary variables ω_j to the simplex set through the softmax function, which can become $\pm\infty$ if some ω_j exceeds a few hundreds (or minus a few hundreds); for this reason, the trait `WeightsTransformation` implements a method `bound` which is designed to ensure that the auxiliary variables remain in a set that prevents this.

More precisely, let $\omega_1, ..., \omega_K$ be the up-to-date values for the auxiliary variables. The softmax function is invariant under addition, so we can add some amount c to each ω_j without changing the corresponding weights; we are looking to find a $c \in \mathbb{R}$ that prevents underflow or overflow, and this can be summarised as

$$c* = \text{argmin}_c \ \ \max_j\{|\omega_j + c|\} = -\frac{1}{2}\left(\max_j\{\omega_j\} + \min_j\{\omega_j\}\right)$$

There are times when even this won't prevent under/overflow from happening simply because the estimated weights are extremely unbalanced; to address this problem, the `bound` method also trims the values of ω_j to (forcibly) prevent that from happening; summarising, the weights update for the softmax transformation looks like the following

$$\begin{aligned} \omega'_j &= \omega_j + c* \ \ \forall \ \ j = 1, ..., K \\ \pi_j &= \min\{\max\{\omega'_j, m\}, M\} \ \ \forall \ \ j = 1, ..., K \end{aligned}$$

For some bounds $m < 0 < M$ that depend on the machine's epsilon and that are calculated at runtime.

In the case of the alternative weight transformation given by eq. 24, the reasoning is the same except that $\omega'_j = c * \omega_j$ and the values m, M are calculated differently, since we do not deal with the exponential function anymore.

Numerical underflows also occurred when working with high-dimensional data due to the Gaussian components' probability density values; as the data dimensionality grows, it is more likely to have points for which $\exp\left(-\frac{1}{2}x^T\Sigma^{-1}x\right)$ causes an underflow, specially when the data is not normalised (this is, it is not zero-mean, unit variance data). For this reason, all such calculations where bounded to be between scala's minimum and maximum positive values.

4.4.3 Initialising the Model With K-Means

Using the result of a fitted K-Means model is a popular way to initialise a GMM, and is also the way to go on [Hosseini and Sra (2017)]; the authors run 30 different K-Means models and initialise the GMM using the best results, with respect to the K-means algorithm's cost function. Our program can be initialised using K-Means models with a small data sample; the size of the sample, number of iterations per K-means model and number of models to try can be specified by the user through input parameters. Care was taken to avoid corner cases such as K-Means clusters with no data (that would cause the corresponding initial covariance matrix to be singular), so the initialisation is not exactly the original K-Means solution, but is close enough. There is, of course, the option of specifying the weights and Gaussian components entirely to initialise the model instead.

4.4.4 Randomised Unit Tests

Many of the unit tests need initial parameter values (this is, initial matrices and vectors) to actually test the program's methods. Such values were not fixed and instead are generated randomly each time the tests are run (of course taking care that covariance matrices are indeed symmetric positive deffinite and so on), in an effort to check the correctness of the methods at the broadest possible level.

4.5 Possible Improvements

- **Distributed Matrix Inversion** It is possible that for relatively high dimensionalities, we can improve performance by inverting the covariance matrices in parallel; just as Spark's GMM, ours use a Singular Value Decomposition (SVD) to invert it because it is a numerical stable procedure, but there are ways of parallelising an SVD and it may be worth exploring that path.

- **Taking Advantage of Matrix Symmetry** All matrices in the program are symmetric, which means we can take advantage of this by creating a new Matrix object that stores only the upper-half and diagonal values; at the moment there is no such thing in `Breeze`, and it would halve the memory usage of the program and speedup calculations.
- **Dynamic Number of Gaussian Components** At the moment the number of Gaussian components remains fixed through the lifetime of the model, but for some streaming applications this may not be the case. It would be desirable to develop a way to dynamically update the number of components.

4.6 Usage

Assume that we have some `data` in the form of an `RDD` of Spark's vectors and we want a model with K components. We can initialise it using a K-Means model with

```
import com.github.gradientgmm._

val model = GradientGaussianMixture.
            init(data, k = K)
```

We can also initialise it with a K-Means model and do gradient-based optimisation with a single instruction (the default optimisation algorithm is simple Gradient Ascent)

```
val model = GradientGaussianMixture.
            fit(data, k = K)
```

We can also use a different optimisation algorithm in the form of a new Optimizer object. All of the optimisation hyperparameters such as the learning rate (step size), its shrinkage rate, the regularisation term and so on are specified using the optimisation algorithm through setters. If we want to use, for instance, Nesterov Ascent with log-barrier regularisation, we do

```
import com.github.gradientgmm.optim._


val optim = new NesterovGradientAscent()
                .setGamma(0.5) //Nesterov correction
                .setLearningRate(0.9)
                .halveStepSizeEvery(50)
                .setMaxIter(200)
                .setRegularizer(new LogBarrier())

val model = GradientGaussianMixture.
            fit(data,k,optim)
```

We can instead create a model from a predefined array of `weights` and `gaussians` and a predefined optimiser `optim` like

```
val model = GradientGaussianMixture(
    weights,
    gaussians,
    optim)
```

For an existing model, we can update its parameters with `step()` like

```
model.step(data)
```

The sequential version of the program works in exactly the same way, plugging an `Array` of Breeze's `DenseVectors` instead of an `RDD` of Spark's `Vectors`, the only difference being that K-means initialisation is not possible for the sequential version, because this is done with Spark's `KMeans` class. For a more thorough explanation of the usage, see the home page of the project's repository.

5 Results

The following section illustrates the program's results and the testing methodology and design choices.

All the tests were performed in a separate `scala` project that has the main program as a dependency. This project can be found in

https://github.com/nestorSag/streaming-gmm-test

In addition to the dependencies and configuration settings described in 4, this project uses `R`[2] to parse the tests' output and generate the figures.

5.1 Test Design Choices

- **Input parameters**: These are set in `textConf.conf`, a text file inside the `resources` folder from the `sbt` project. The input parameters include data size, dimensionality and number of Gaussian components, and also the optimisers' hyperparameters related to the step size and acceleration.
- **Output**: The tests are designed to print the relevant output to the log file `reports.log` at root level, from where they can be parsed with the `R` functions inside the folder `logsParser`.
- **Reproducibility**: All randomness in the program is seeded so that simulations can be replicated exactly; this includes the data generation and mini-batch selection at every iteration of the main program.

5.2 Data

The project contains a `DataGenerator` and `CSeparatedDataGenerator` classes which produces synthetic data generated from a 'true' GMM model. they take as input the number of components K, the desired data dimensionality, and the data separability in the case of the later (see section 5.2.1), and generates a random GMM model from which data can be simulated.

The parameters are determined in the following way:

- **Weights**: Vector of `Uniform(0,1)` random variables, normalised so that it sum up to 1.
- **Covariance matrix**: Outer products of random vectors are added to an initial identity matrix; this ensures that the resulting matrix is in fact SPD. The vectors have all the same random offset for a given mixture component, in order to simulate some amount of covariance.

[2] https://www.r-project.org/about.html

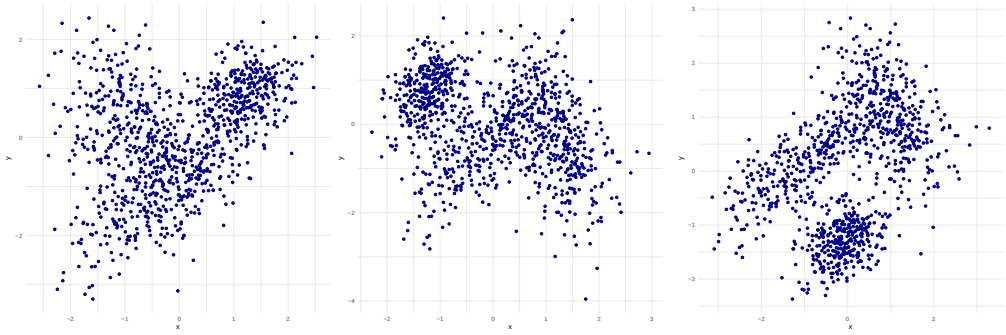
- **Means**: they are set as random uniform vectors in a box centered around zero, and then adjusted to fulfil c-separation criteria (see below).

5.2.1 C-separation

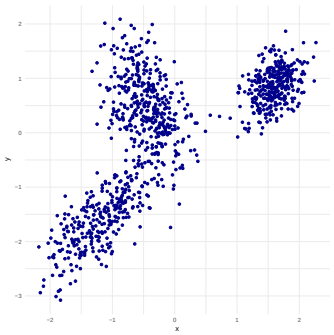
The performance of GMM-fitting algorithms is known to depends heavily on how well separated the real mixture components are [Ma et al. (2000)]. To take this into account, for the tuning and performance tests we have generated the synthetic datasets characterising them by their *c-separability* [Verbeek et al. (2003), Dasgupta (1999)]. For a $c \in \mathbb{R}^+$, a GMM is c-separable if

$$\forall \ i \neq j : ||\mu_j - \mu_i|| \geq c * \sqrt{\max\{\text{trace}\left(\Sigma_j\right), \text{trace}\left(\Sigma_i\right)\}}$$

The larger c is, the better separated the components are; this allow us to easily generate arbitrarily 'complicated' data to test our algorithms and their different parameters.



(a) c = 1 (b) c = 1.4 (c) c = 1.7



(d) c = 2

Figure 5: Example of c-separable GMM data

The data was subsequently standardised for the purpose of numerical stability (always checking that the c-separability condition still held); non-standardised data have he risk of producing numerical overflow or underflow in high dimensions because of the Multivariate Gaussian density evaluations.

5.3 Behaviour Tests

To observe the qualitative differences in the results induced by different choices of descent direction, regularisation term and weight mapper, low-dimensional synthetic data was generated to plot the trajectories of the parameter estimators throughout the optimisation procedure. The goal is to see how the trajectories towards the solution behave as the iterations go by.

The following tables summarise the data parameters used for these tests. The c-separability metric was not used for these tests.

Parameter	Value
Data size	1,000
Components	3
Dimension	2

Table 2: Data hyperparameters

Class	Parameter	Value
`Optimizer`	`learningRate`	0.9
	`halveStepEvery`	10
	`minLearningRate`	0.1
`GradientGaussianMixure`	`maxIter`	50
	`batchSize`	25
`MomentumGradientAscent`	`beta`	0.55
`NesterovGradientAscent`	`gamma`	0.55
`ADAM`	`beta1`	0.5
	`beta2`	0.5
`ADAMAX`	`beta1`	0.5
	`beta2`	0.5
`LogBarrier`	`scale`	1.0
`ConjugatePrior`	`normalMean`	empirical mean
	`iwMean`	empirical covariance
	`Df`	3
	`dirichletParam`	1/3
GMMTester	seed	99

Table 3: Optimisation hyperparameters

The initial parameters were set as follows:

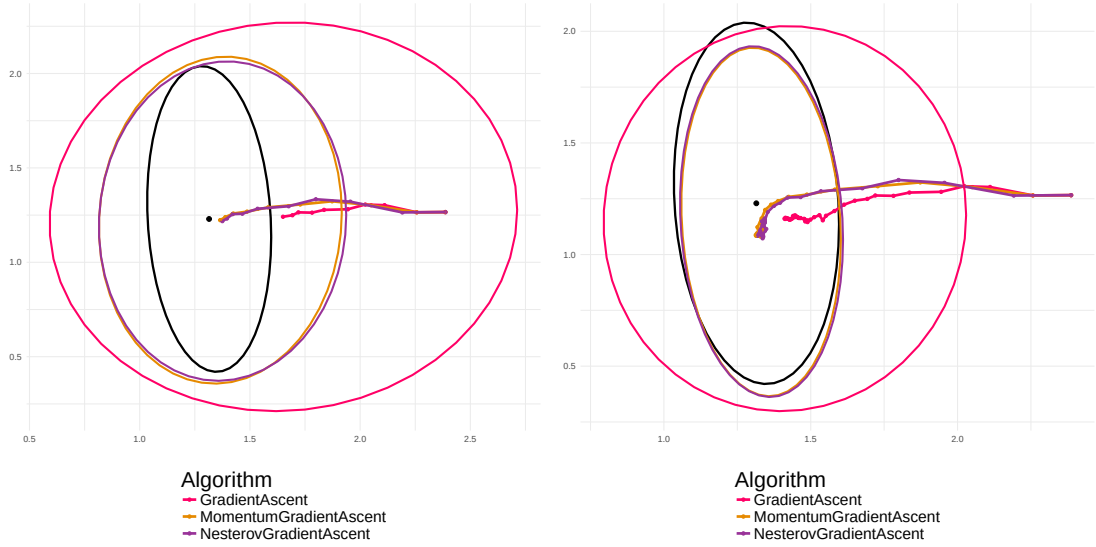
- **weights**: set to $\frac{1}{3}$
- **Covariance matrices**: set to identity matrices
- **means**: Set to data mean plus a step toward the respective component's true mean

The objective was to initialise the model far enough to observe clearly the difference in the trajectories' behaviour while avoiding getting stuck in spurious minima; high sensibility to initial parameters is a well-known problem for the EM algorithm applied to GMMs and it is also present for gradient-based algorithms. Normally we would initialise the models using a preliminary K-Means model, but in $\mathbb{R}^2$ that would not leave much space to analyse the paths toward the final parameters .

To represent the covariance matrix estimates in the following plots, they are depicted as the induced Gaussian distribution's contour lines; intuitively, these lines indicate the shape of the component's corresponding data. In the following subsection, the black contour lines and mean correspond to the true component's parameters; ideally, the algorithms' trajectories should eventually overlap with those.

50 iterations of 25 samples make 1.25 *epochs*, so in computational terms, this test is equivalent to a bit more than a single iteration of the EM algorithm.

5.3.1 Algorithms



(a) After 10 iterations

(b) After 50 iterations

Figure 6: Trajectories of accelerated descent estimators.

In figure 6 we can see the results of running the optimisation procedure with different choices of optimisation directions. What we see is the path of the estimates towards the true parameters of one of the model's components; we did not include the three components to avoid cluttering the figures.

The most important thing to note is that both of the accelerated algorithms shown here achieve better results than simple SGD, and this was almost always true for all the behaviour test we ran; the accelerated trajectories exhibit the inertia that is typical of them in other problems, and this makes them follow a curvier, smoother path towards

the solution than SGD, but also make them converge faster most of the time. After 10 iterations the accelerated algorithms have almost reached the solution, while SGD is roughly half way there for both the mean and variance estimates. After 50 iterations the estimates of the accelerated algorithms have stabilised close to the true parameters, while the quality of the SGD estimates is roughly the same as those obtained by the accelerated algorithms after just 10 iterations.

Hyperparameters related to the inertia in both Nesterov's and Momentum descent need to be given some thought; too much inertia would make the estimates go past the true parameters and wander around. We found that momentum's `beta` and Nesterov's `gamma` of between 0.5 and 0.6 resulted in the best performance; more than that often resulted in somewhat long, entangled trajectories.

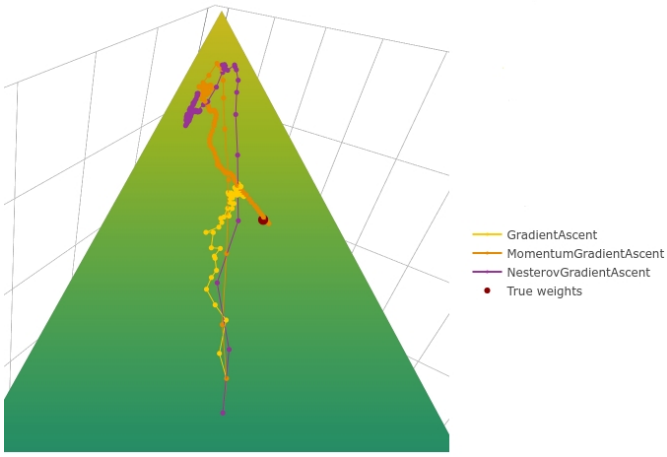


Figure 7: Weight estimation with excess inertia

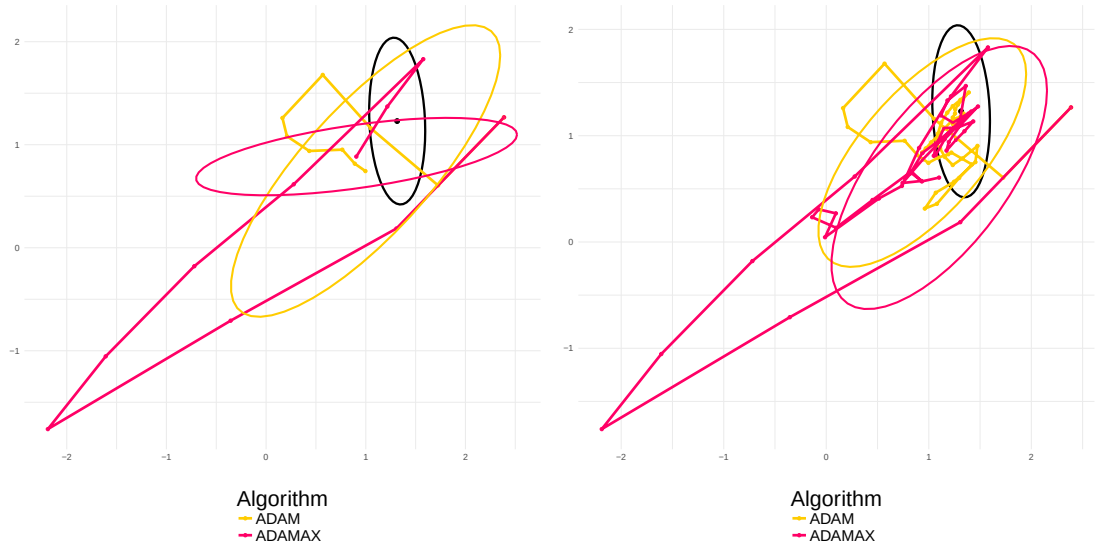
An example of the effect of too much inertia can be best observed in the weights' estimates in figure 7, where we have set momentum's `beta` = 0.8 and Nesterov's `gamma` = 0.8; the green surface represents the weight simplex. In this case, simple SGD gets closer to the true solution and stays around there for the rest of the iterations, while both accelerated algorithms go past it; Nesterov's algorithm do not gets close at all, and momentum SGD reach the true solution after a long detour.

From the four algorithms we have developed for the program, ADAM and ADAMAX have consistently been the worst-performing ones (see figure 9); despite our best efforts to tune its parameters, we obtain highly unstable results as the one shown below, both for the Gaussian and weights estimators.

Their behaviour was characterised by very large step sizes at the beginning, not always towards the true parameters. The same behaviour was observed for different configurations of `beta1`, `beta2`, and step size related hyperparameters.

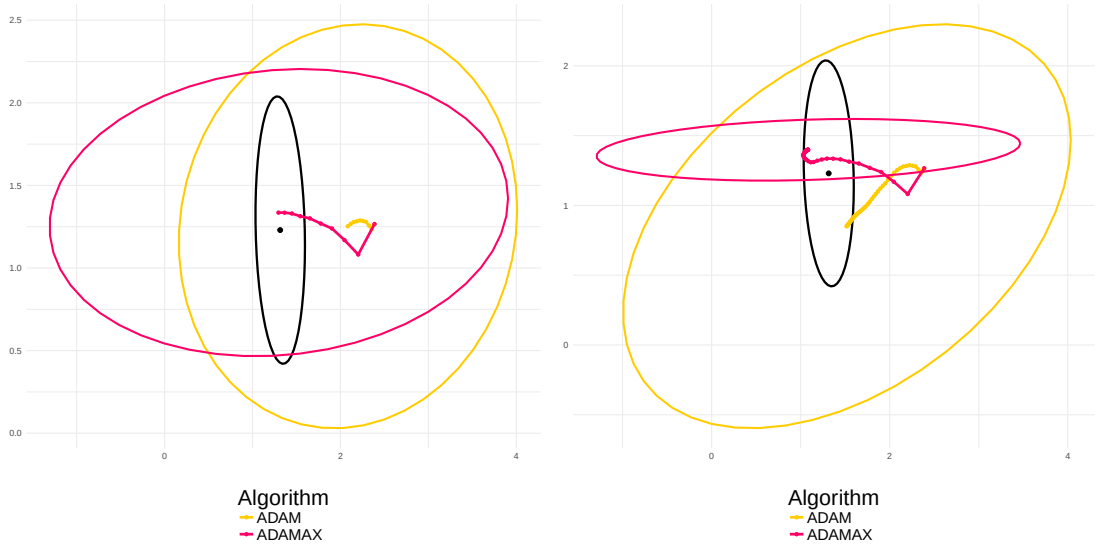
We found that part of the reason for this instability is that the bias-correcting term that ADAM uses internally can cause very large step sizes at the beginning (specially for a starting step size of 0.9 as in our experiments) which affect the covariance estimates,

causing them to become significatively inflated or elongated; this begins a chain effect, making the other estimates (weights and means) to be pulled more strongly by points farther away, and all estimates are destabilised as a result. A possible fix for this is to set the initial step size to a small value, and in fact in [Kingma and Ba (2014)] an initial value of 10^{-3} is suggested; this makes it more stable, however they still perform poorly; moreover, this puts ADAM and ADAMAX at a disadvantage against the other descent algorithms, since they perform well with much larger initial step sizes.



(a) 10 iterations (b) 50 iterations

Figure 8: Trajectories of ADAM, initial step size of 0.9



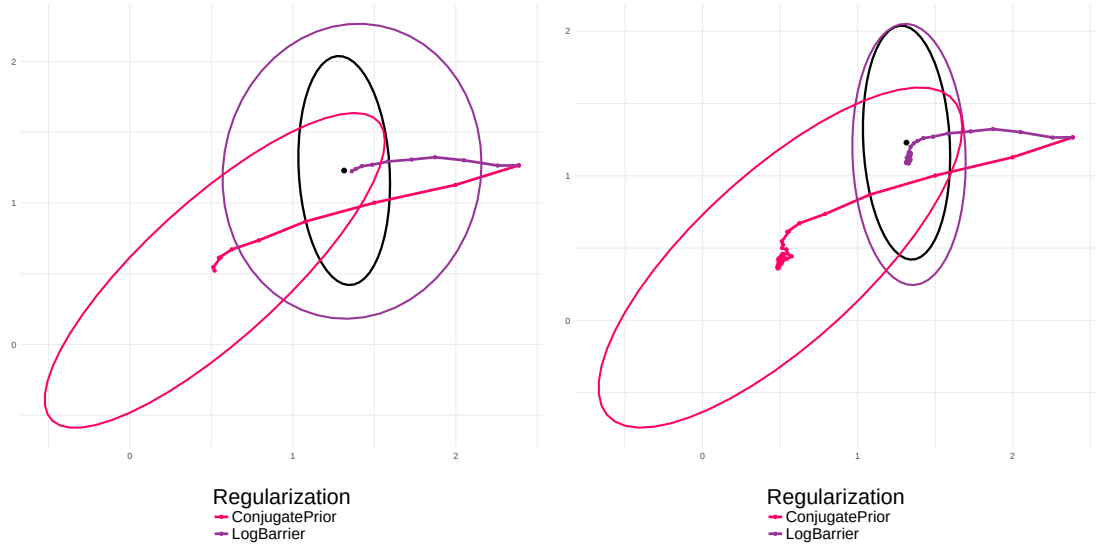
(a) 10 iterations (b) 50 iterations

Figure 9: Trajectories of ADAM, initial step size of 0.05

5.3.2 Regularisation

Figure 6 suggests that accelerated methods perform at least as good as simple SGD, and for this reason we have used momentum SGD for all the following behaviour tests.

We were certainly expecting slower convergence for a model with a conjugate prior because it effectively attracts the model a bit toward the prior parameters, but the effect on the final model's quality was more significant than we anticipated. The size of this effect is directly related to the batch size: with a batch of size 25, the prior distribution weighs enough to prevent the estimates from reaching the true parameter values (see left plot, figure 10) and in the case of the mean estimate in particular, even pull it to the prior parameter value, which is at the origin; this in turn causes the model's components to try to reach points from other components, so their variance become elongated. With a batch size of 100 (figure 11) this effect becomes weaker, although still significant: the estimates stop short of the true component's mean and covariance. The weights estimates do not converge to the true solution either, in part because of analogous reasons and in part because errors in the Gaussian estimates propagate to the weights estimates.



(a) After 10 iterations

(b) After 50 iterations

Figure 10: Regularised models for a batch size of 25

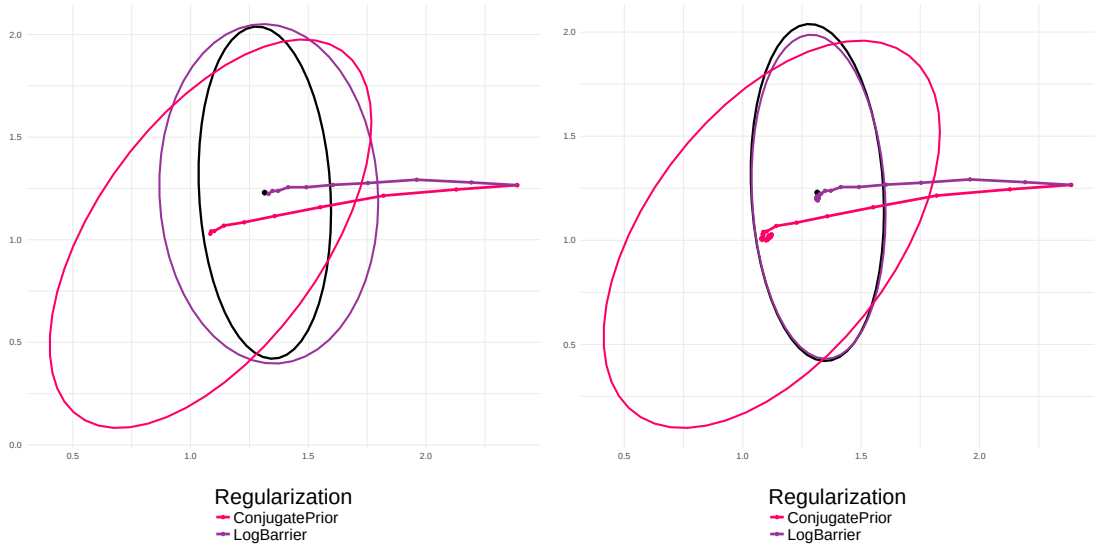
It is also possible to tune the prior distribution's parameters to atenuate this effect; in [Hosseini and Sra (2017)] they use a κ parameter close to zero and seem to obtain good results for such regularisers (even though technically speaking they are not prior distributions anymore, because the parameter range restrictions are not fulfilled); we tried such an approach but the prior's effects were still visible.

For models with logarithmic barrier on the other hand, they consistently showed similar performance than unregularised models both in trajectories and final estimates, which

is a good thing in terms of model quality. It can be observed in the left plot of figure 10 that covariances are overestimated when compared to an unregularised model under the same conditions (see figure 6, left plot), and this was always true in our experiments; this is to be expected: we are penalising 'thin' covariance matrices to avoid singularities. We found this overestimation small enough so as not to be a problem.

A model with log barrier regularisation follows practically the same path towards the true weights and means since the barrier penalises neither.

Overall the quality of a model with logarithmic barrier remains virtually unchanged, so we can argue that for mini-batch gradient-based GMMs, a logarithmic barrier carries less risk of degrading the model quality while also avoiding covariance singularities.



(a) After 10 iterations

(b) After 50 iterations

Figure 11: Regularised models for a batch size of 100

5.3.3 Weights Mappers

Recall that in [Hosseini and Sra (2017)], they used a softmax function to recover the weights from a set of auxiliary real-valued variables (see equations 11 and 14) and that we proposed an alternative transformation (equation 24), hopping to speed up the convergence of the weights estimators. We have found that as the data's true weights gets closer to the simplex borders (this is, as the weights get more unbalanced), the estimates given by the ratio transformation in equation 24 become unstable, as can be seen in figure 12, where it goes from one side of the simplex to the other in just a few iterations. Even when the weights are not too unbalanced, for almost all the simulations we ran the softmax transformations showed faster convergence.

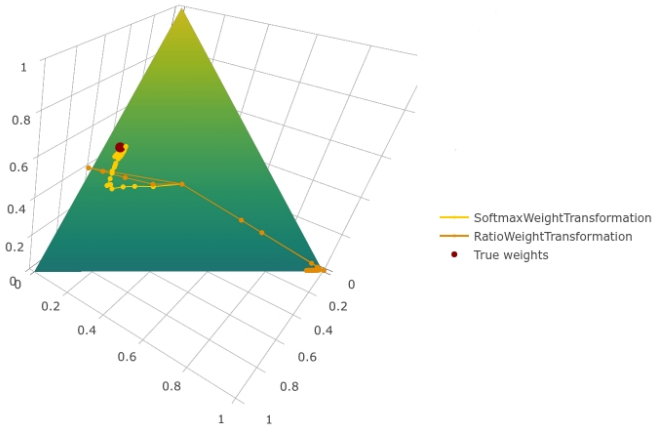


Figure 12: Weight estimation under different weight mappings.

5.3.4 Summary

We found that initialising the step size with a value close to 1 and shrinking it up to a minimum allowed value (say, 10^{-2} or 10^{-3}), as was done in [Hosseini and Sra (2017)], results in the fastest performance.

We also found that for most of the simulations we ran, Momentum's and Nesterov's SGD reached the solution much faster than simple SGD (up to five times faster in the example of figure 6). Their only drawback is having to tune an additional parameter, β and γ, respectively. In our experiments, both algorithms achieves the best behaviour when said parameters are between 0.5 and 0.6.

ADAM performed consistently poorly in all our experiments. We found out that part of the problem is that ADAM's bias-correcting mechanism induces big step sizes at the beginning which can cause the covariances to very rapidly inflate or become elongated, and this in turn makes the estimates to be pulled more strongly by far-away points, destabilising them. A possible fix is setting a shorter initial step size, but then the convergence will be very slow. Overall, the performance of ADAM and ADAMAX was markedly worse than that of the other algorithms.

We also found that a conjugate prior regulariser can degrade the final model's quality considerably for small batch sizes, pulling the estimates towards the prior parameters. This effect becomes weaker as the batch size increases, although its effects were still clearly visible for batches of size 100 in our experiments. We found that placing a logarithmic barrier to avoid covariance singularities as suggested in 3.1.2 works much better than placing a conjugate prior, leaving the model's trajectories and final estimates virtually unchanged; the overestimation of covariances that it induced was minimal and did not represent a problem. For big data problems, where batch sizes can be many thousands or more, both regularisers would probably give very similar estimates, however, we would still favour using the logarithmic barrier regulariser because it is also more efficient in terms of memory and computation: unlike a conjugate prior it does not

need to store additional prior parameter matrices, and the evaluation of the regularisation term and its gradient is also cheaper. We would only recommend using a conjugate prior regulariser when it is intended as a true prior distribution that carries background knowledge. Unfortunately, specifying individual prior parameters for each component is not possible in the current implementation, which can somehow discourage said use case.

The alternative weights transformation of equation 24 turned out to be too unstable for relatively unbalanced weights. We concluded that it is better to always use the `SoftmaxWeightTransformation` class, which implements equations 14 and 11. A possible fix for ratio transformation would be using $\pi_{\max}$ at every iteraition instead of simply π_K in equation 24, but this has not been tried.

5.4 Parameter Tuning

The purpose of the following tests was to determine the optimal parameter setting under different conditions and in general to observe how the change in some of the program's parameter affect performance. All models were initialised using the results of a K-means model fitted with a small data sample

The parameters that were tried (unless explicitly said otherwise) in these tests are below

Parameter	Value	Parameter	Value
Data size	100,000	`learningRate`	0.99
Components	4	`halveStepSizeEvery`	30
Dimension	20	`minLearningRate`	0.01
Separability	0.2, 0.5, 1	`batchSize`	500
Validation data	1,000	`maxIter`	100

Table 4: Data and optimizer hyperparameters

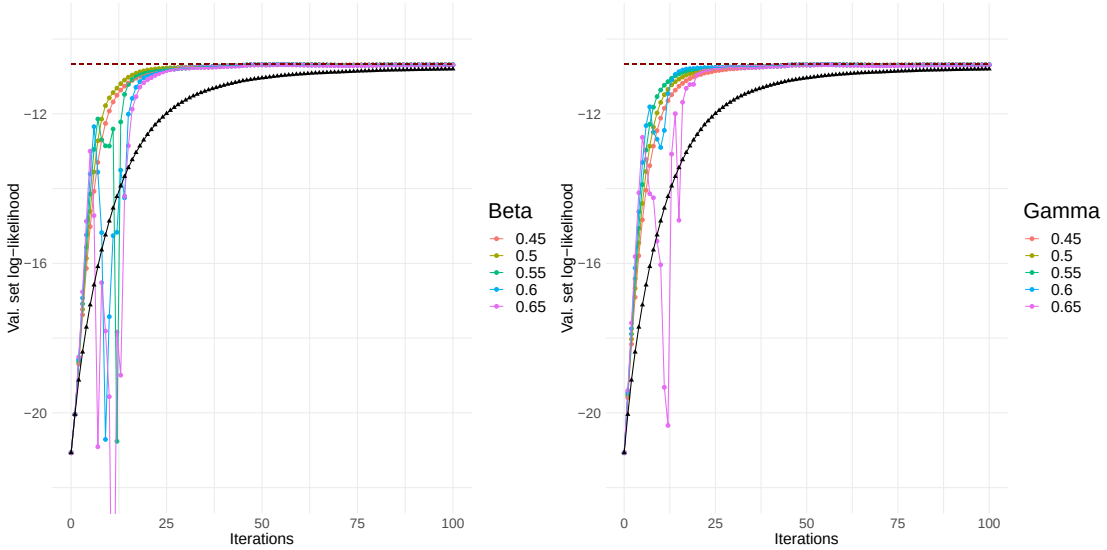
To measure the algorithms' performance we tracked the improvement in the data log-likelihood versus number of iterations. The log-likelihood was calculated using a separate evaluation set.

The tests were ran in a local `Spark` cluster using four i7-4500U 1.80GHz cores.

5.4.1 Acceleration And Data Separability

We tried different values of the inertia-related parameters, which are `beta` for Momentum SGD and `gamma` for Nesterov SGD with data that had different degrees of separability. For the results shown in figure 13 we used $c = 1$, which produced relatively separated data, unlikely to have heavily overlapping components. In all of the

following figures, the red dotted line represents the true log-likelihood; The black line trend represents standard SGD, which is intended as a baseline for the comparison of accelerated algorithms.

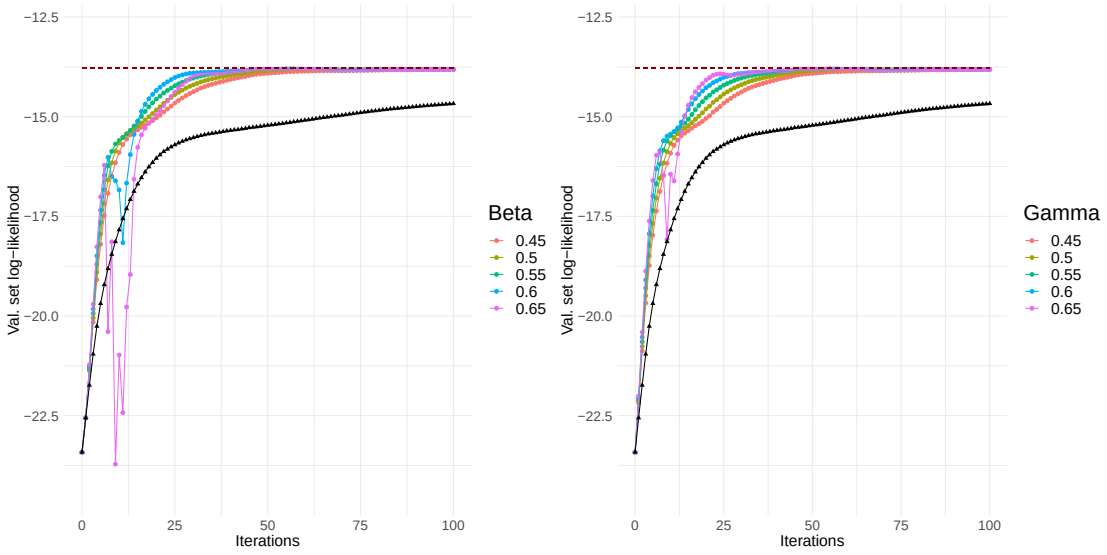


(a) Momentum SGD (b) Nesterov SGD

Figure 13: Acceleration comparison, data separability c = 1 (high)

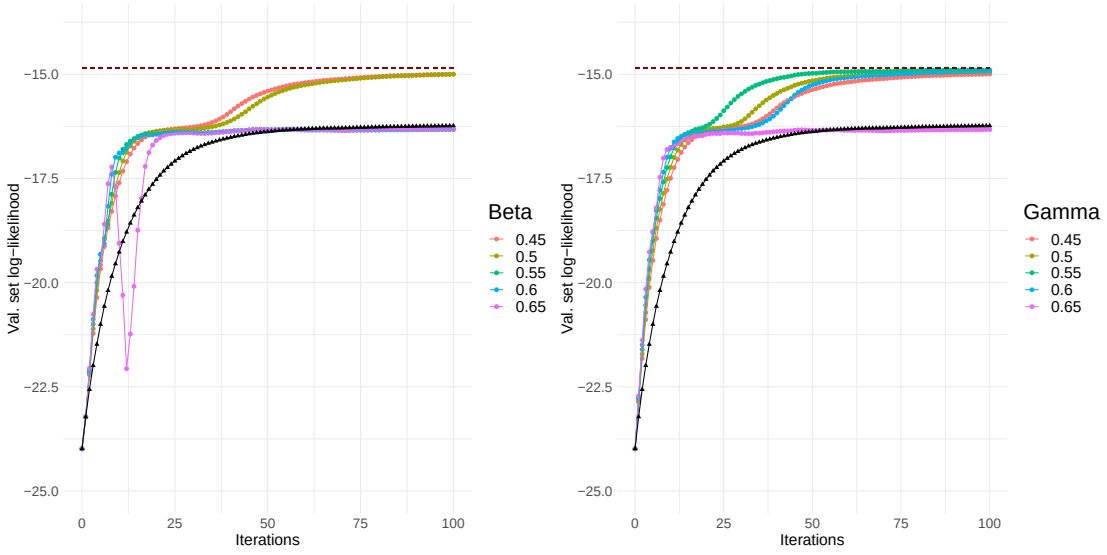
We can see in figure 13 that for well-separated components, all levels of inertia induced better performance in both accelerated algorithms than standard SGD. We also found, just as in the behaviour tests, that a value of between 0.5 and 0.6 gives the best performance; higher values make the algorithms unstable, while lower ones result in slightly slower convergence.

We can clearly see the effect that poorly separated data have on convergence in figure 14, where we see, specially for SGD, that the algorithms converge much more slowly after a few iterations. Accelerated algorithms still manage to recover the true parameters after comparatively few iterations; we hypothesise that the deceleration and subsequent acceleration in log-likelihood improvement of accelerated algorithms around iteration 20 is due to getting close to a spurious local maximum; it would suggest that acceleration is beneficial for avoiding some of such spurious optima, in which other algorithms, like EM, often get caught.



(a) Momentum SGD (b) Nesterov SGD

Figure 14: Acceleration comparison, data separability c = 0.5 (medium)



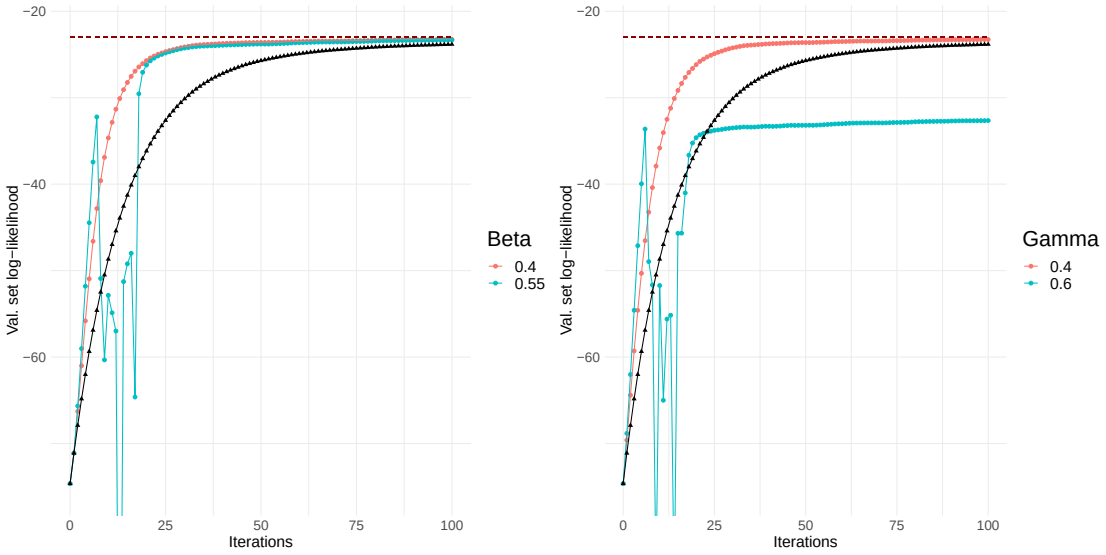
(a) Momentum SGD (b) Nesterov SGD

Figure 15: Acceleration comparison, data separability c = 0.2 (low)

Finally, in the case of data with very low separation, as in figure 15 the convergence is slower still than in figure 14; it takes approximately four times more iterations for the best-performing algorithms to get as close to the true log-likelihood value as before. Standard SGD together with some of the accelerated algorithms with higher inertia level seem to get caught in a spurious optima; judging by the log-likelihood trajectories the remaining algorithms seem to get close to the same saddle point, but they manage to break free from it after a few iterations; this suggest that only the 'right' level of inertia would help the algorithm to escape such spurious solutions.

5.4.2 Acceleration And Dimensionality

Hyperparameter values found to be good in $\mathbb{R}^{20}$ may induce instability when dealing with data in an 80-dimensional space, as shown below



(a) Momentum SGD (b) Nesterov SGD

Figure 16: Acceleration comparison, 80-dimensional data, c = 1

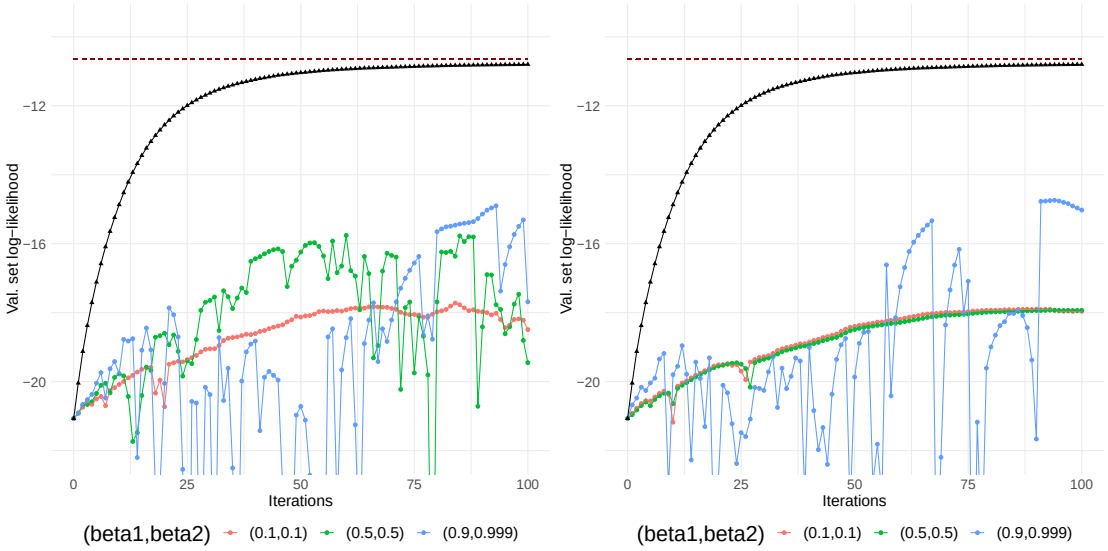
As we can see, we get mixed results; optimal parameters for 20-dimensional data makes the algorithm unstable in an 80-dimensional space; although it can still converge to the true log-likelihood value, it can also perform poorly. This might suggest that a given value for `beta` or `gamma` implies a higher level of inertia as the problem's dimensionality grows.

5.4.3 ADAM(AX) Parameters

In the case of ADAM and ADAMAX, they again performed poorly compared with all other algorithms. Using an initial step size of 0.9 as in the other trials produced extremely unstable results, so the ones shown in figure 20 were obtained using an initial stepsize of 0.01. Although for most trials in said figure there is an upward trend in log-likelihood, the results are still unstable and oscilate wildly, except for `beta1` and `beta2` values of 0.1, and even then the convergence is extremely slow.

The `beta1` and `beta2` parameters suggested originaly in [Kingma and Ba (2014)] correspond to the blue line, which does not give particularly good results. ADAM and ADAMAX were first intended for Deep Learning applications, and it is possible that the optimisation problems of such applications are qualitatively different from the one from GMM. Furthermore, the main appeal of ADAM is that it provides a dynamic step size

for each parameter individually, which can be a big advantage in Deep Learning problems because the surface of the loss function in such problems tends to be very irregular, and this makes the results depend heavily on how the step size behaves throughout the optimisation [Nar and Shankar Sastry (2018)].



(a) ADAMAX (b) ADAM

Figure 17: ADAM and ADAMAX, data separability c = 1

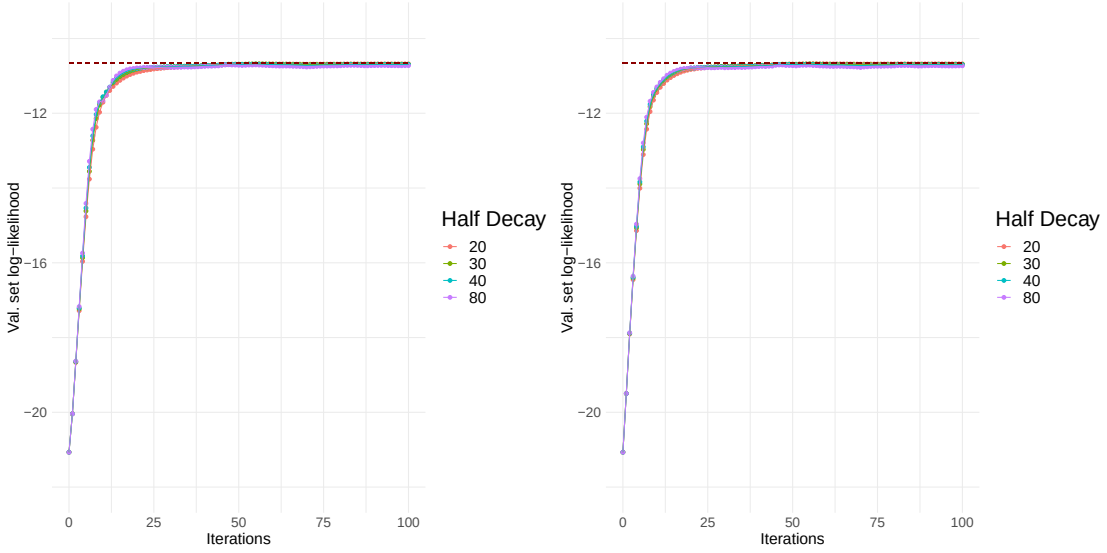
For GMMs, however, we have found that the stepsize is somewhat less important in the resulting model's quality. There is a relatively wide set of step size behaviours that will produce approximately the same result. This fact, coupled with the instability induced by the gradient's initial moment estimates on the covariance matrices, make ADAM less appealing for fitting GMMs.

5.4.4 Step Size Shrinkage Rate

Our experiments agree with those of [Hosseini and Sra (2017)] in the sense that the best strategy for the step size seems to be starting with a large one, say, 0.9 or 0.99, and shrinking it to a small size as iterations go by; the minimum allowed step size controls how close we can get to the true solution, since mini-batch optimisation will always inccur in some approximation error unless the step size tends to zero; we have not found so far a significant different in results when changing this value between 10^{-3} and 10^{-1}. However, the shrinkage rate may be important for the algorithms' overall performance, and for this reason we also ran tests trying different values; The rates are chosen indirectly, selecting a *half decay* instead, which denotes the number of iterations that have to pass to make the learning rate half its current value; this parameter results much more intuitive than selecting a decay rate between 0 and 1.

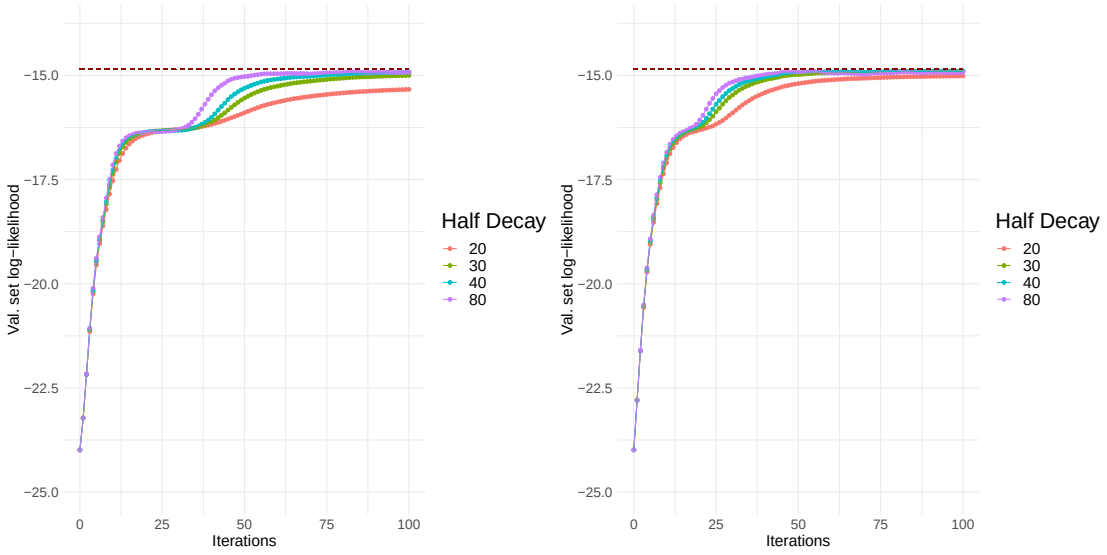
Taking into account our results so far about the best-performing algorithms, we ran the

tests using a Momentum and Nesterov SGD with `beta` and `gamma` parameters of 0.55 and 0.6 respectively.



(a) Momentum SGD (b) Nesterov's SGD

Figure 18: Step size shrinkage rate comparison, data separability c = 1



(a) Momentum SGD (b) Nesterov's SGD

Figure 19: Step size shrinkage rate comparison, data separability c = 0.2

In the case of relatively well-separated data (figure 17 the shrinkage rate does not affect performance significantly; however, for data with low separation, the effect on convergence is stronger. The slower half decay turns out to give the faster performance for both algorithms. This might suggest that the model works well for relatively big step

sizes, and it suggests trying large half decay rates to get the fastest convergence. However, when the step size stays large for too many iterations, the acceleration terms can make it oscilate wildly.

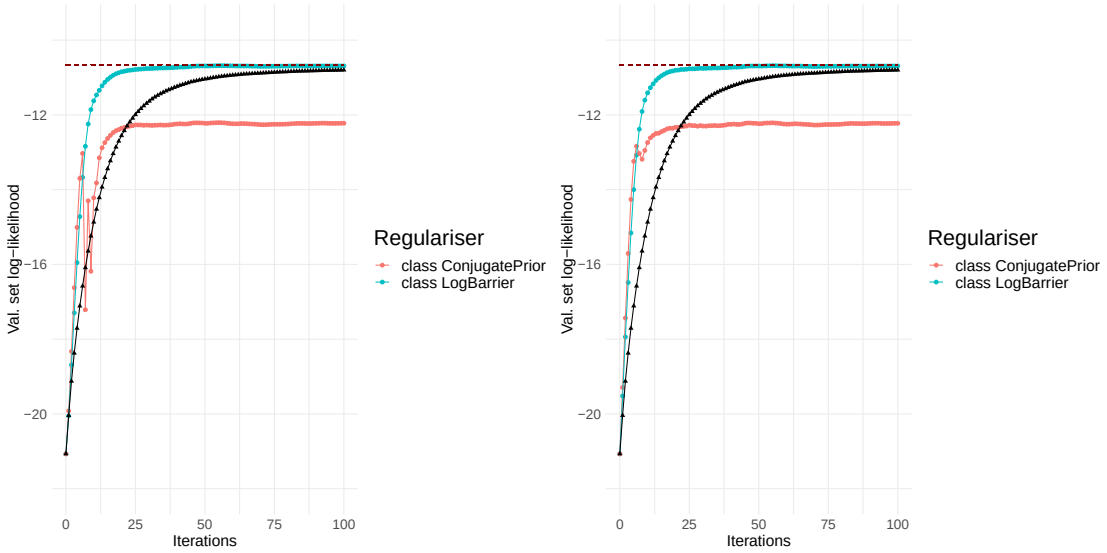
5.4.5 Regularisation And Model Quality

To test the effect of different regularisation terms in a more realistic setting than a 2-dimensional problem as in section 5.3.2, we tried adding regularisation to a problem with characteristics described in table 4, and a separacion of c = 1.

Parameter	Value
`df`	20
Prior means	zero vectors
Prior covariances	Identity matrices
Prior weight	0.25

Table 5: Conjugate prior parameters

Recall that the data is normalised so that it has zero mean and unit variance, so the parameters in table 5 are effectively the empirical mean and axis-wise variance; without any further information, this is the most information we can incorporate to the prior. `df` must be at leas 19 if we weant the prior to be a true density function, and is set to 20.



(a) Momentum SGD (b) Nesterov's SGD

Figure 20: Comparison of regularisation terms, data separability c = 1

Just as in the behaviour tests, we see that the prior affect the performance considerably. It is possible that such a strong pull is due to κ in equation 19, which grows with the problem dimensionality if we want the prior to be a true distribution function; we could

use a pseudo-distribution, setting κ to a value close to zero, as in [Hosseini and Sra (2017)]], and see if the log-likelihood improves. In any case, we can confirm that for more realistic settings the logarithmic barrier regularisation leaves the model's quality virtually intact.

5.4.6 Summary

We have found empirically that good values for the `beta` and `gamma` hypermarameters for Momentum and Nesterov's SGD respectively are between 0.5 and 0.6, and this seems to be robust to changes in data separation; parameters in this range are more likely to converge to the true log-likelihood and even seem to help getting out of saddle points (figure 13). However, this must be adjusted to the data dimensionality; as the later grows the values of these parameters must be lowered to avoid making the algorithms unstable. For all tested parameter values, Nesterov's and Momentum SGD achieve consistently faster convergence than simple SGD; in regard to ADAM and ADAMAX, although they seem to improve the log-likelihood, they perform very poorly even compared to standard SGD, which may point to qualitative difference between its intended use cases in Deep Learning and GMM models. Furthermore, in many situations the fitted GMM models seem to be more robust to small changes in step size behaviour, making the main advantage of ADAM and ADAMAX less appealing and discouraging its further use.

We have also found that for well-separated data, the step size decay rate does not affect performance significantly. However, for poorly separated data, it affects convergence speed, and our experiments suggest setting a large half decay, making the step size to decrease slowly to achieve fast convergence.

Finally, we have found evidence that for realistic settings, the difference between the effects of both regularisation terms on the model still hold; logarithmic barrier regularisation have a much smaller footprint in the final model's quality, if any, than a conjugate prior.

5.5 Performance Tests

In the following subsection, we review our findings running the algorithms under different conditions, measuring its performance in terms of time and scalability.

5.5.1 Sequential Mini-Batch Optimisation

The following tests were run on an AWS instance using regular `scala` functions, without `Spark` or any multi-thread functionality. To compare the results, MLLib's `GaussianMixture` algorithm was also run distributing the data on 16 cores across

two workers; experiments with both algorithms used the same data and initial parameters.

To get a concise measure of the algorithms' performance, let us define the *progress* of an algorithm in iteration t as

$$\text{progress} = \frac{LL_t - LL^*}{LL_0 - LL^*}$$

where LL_t is the log-likelihood value as measured on a validation set on iteration t, and LL^* is the true model's log-likelihood. At the beginning of the optimisation, the progress would be 0% and we aim to reach 100%. This gives us an easy way to see how close the algorithm is to the desired solution regardless of the actual log-likelihood values. The hyperparamters used for these tests are below.

Parameter	**Value**
Components (k)	{4,8,12}
Dimensionality (dim)	{30,50}
separability (C)	{0.2,1}
size	4,000,000 (915MB)

Table 6: Data hyperparameters

Parameter	**Value**
Batch size	500
Step size	0.99
Halve step every	30
Min step size	0.001
max iterations	500

Table 7: Descent hyperparameters

In the following tables, we will refer to Momentum SGD as MSGD and to Nesterov SGD as NSGD; as was said above, LL_0 means the initial log-likelihood value while LL^* is the true model's log-likelihood, they are shown just as a reference.

C	dim	k	MSGD		EM		LL_0	LL^*
			Time(s)	Progress	Time(s)	Progress		
0.20	50	4	14.40	97.44%	109.34	36.51%	-55.97	-28.50
0.20	50	8	13.39	82.10%	184.14	20.14%	-52.68	-26.77
0.20	50	12	35.28	89.79%	327.39	0.56%	-49.94	-25.50
1.00	30	4	1.42	98.57%	192.23	99.99%	-32.08	-15.51
1.00	30	8	1.20	85.03%	334.87	99.99%	-28.02	-12.18
1.00	30	12	11.35	92.27%	618.96	99.98%	-26.11	-11.15
0.20	30	4	4.38	98.34%	211.70	99.75%	-35.50	-20.20
0.20	30	8	4.93	94.59%	547.70	99.68%	-33.39	-19.28
0.20	30	12	8.51	92.67%	1116.44	98.86%	-33.27	-20.28

Table 8: MSGD and EM time and progress comparison

Table 8 fully illustrates the advantage of using stochastic optimisation: the elapsed time is independent of the data size, scaling well to large datasets. For 50-dimensional data our program outperforms Spark's completely both in time and progress, and for the rest of the tests the results are fairly close in terms of progress while using a tenth to a hundredth of the EM's time budget and much less computational resources; our program also processed much less than a single epoch. The poor performance of EM on the 50-dimensional data tests was likely due to particularly poor spurious optima; our program managed to avoid them, and we hypothesise that acceleration terms help the algorithms escape some of the spurious optima in which EM might still get stuck.

Such huge savings in time budget can be used to compensate some of the additional estimation error incurred in by mini-batch optimisation; the following table shows the results of identical experiments, this time running MSGD a second time over the results of the first run and using a batch size of 2,000 for the second run.

C	dim	k	MSGD		EM		LL_0	LL^*
			Time(s)	Progress	Time(s)	Progress		
0.20	50	4	21.47	99.39%	109.34	36.51%	-55.97	-28.50
0.20	50	8	16.16	84.22%	184.14	20.14%	-52.68	-26.77
0.20	50	12	62.32	92.24%	327.39	0.56%	-49.94	-25.50
1.00	30	4	3.89	99.63%	192.23	99.99%	-32.08	-15.51
1.00	30	8	40.83	89.29%	334.87	99.99%	-28.02	-12.18
1.00	30	12	13.90	93.67%	618.96	99.98%	-26.11	-11.15
0.20	30	4	5.82	99.64%	211.70	99.75%	-35.50	-20.20
0.20	30	8	19.20	96.27%	547.70	99.68%	-33.39	-19.28
0.20	30	12	47.74	97.74%	1116.44	98.86%	-33.27	-20.28

Table 9: Comparison of EM vs two consecutive runs of MSGD

Table 9 shows that our program can get near-optimal estimators for a fraction of time and computation, compared to EM; in many Machine Learning applications, the additional error incurred in by using mini-batch optimisation would not be a proble, since

is often the case that achieving the absolute best estimators is not necessary or desirable because on the one hand, it can be very expensive, and on the other, parameter estimation is not the most important source of error and after a certain point, higher accuracy in the parameter estimators does not make any difference. This may be also true for many clustering applications. Hence, for such applications mini-batch-optimised GMMs provide a much faster, cheaper alternative for doing large-scale Gaussian Mixtures.

MSGD and NSGD behave similarly but not identically; moreover, neither is completely dominated by the other and each will give better results for some problems. For real applications where we do not know the maximum possible log-likelihood, it is advisable to try both algorithms under many configurations in order to get the best possible result. Below the results of the same tests using NSGD are shown, which can be compared with table 8.

C	dim	k	MSGD		EM		LL_0	LL^*
			Time(s)	Progress	Time(s)	Progress		
0.20	50	4	16.21	97.14%	109.34	36.51%	-55.97	-28.50
0.20	50	8	26.45	87.43%	184.14	20.14%	-52.68	-26.77
0.20	50	12	7.51	82.69 %	327.39	0.56%	-49.94	-25.50
1.00	30	4	1.09	97.90%	192.23	99.99%	-32.08	-15.51
1.00	30	8	14.35	88.09%	334.87	99.99%	-28.02	-12.18
1.00	30	12	17.85	91.83 %	618.96	99.98%	-26.11	-11.15
0.20	30	4	1.20	98.37%	211.70	99.75%	-35.50	-20.20
0.20	30	8	11.56	94.73%	547.70	99.68%	-33.39	-19.28
0.20	30	12	2.99	90.43%	1116.44	98.86%	-33.27	-20.28

Table 10: NSGD and EM time and progress comparison

The results of standard SGD were similar in terms of time, but were outperformed in model's quality by both MSGD and NSGD. They are shown on the appendix C.

5.5.2 Stochastic Distributed Optimisation

The distributed tests were all done on an Amazon Web Services (AWS) computer cluster with different configuration types that are specified below. The reason of this was to try the program under conditions more closely resembling real application deployments and to save configuration/debugging time.

The results were disappointing. SGD (and by extension, its accelerated versions) is hard to parallelize efficiently; we are doing what is often called synchronous SGD[Arnold (2016)] (we cannot do otherwise because of Spark's synchronisation barriers), where we alternate between gradient computation by the workers and parameter updating at the master, who then sends the new model to the workers (see figure 4) and so on; this version relies heavily on the hardware being used[Arnold (2016)], and for commodity

hardware such as the one in an AWS cluster it is difficult to get gains in performance, because the synchronisation and communication overhead makes any mini-batch step costly, regardless of the batch size. On top of this, Spark is currently not optimised for mini-batch processing. The way this is done in the sequential version of the code and in other frameworks, is shuffling the data at the beginning of each epoch, and then processing ordered batches of it until it runs out, then shuffling it again and so on. Spark currently does not have this functionality, and the most straightforward fix is taking a random data sample at each iteration, but this requires scanning the whole datsaset, which affects performance heavily; we also tried pre-partitioning the data set in batches of the required size previous to the optimisation using MLLib's `RDDFunctions`, and in this way processing a single pre-built batch in each worker at each iteration. In theory, this should have made the processing time independent from the dataset size, solving the problem for the most part (such preprocessing is expensive in itself), but after doing this the performance got worse. We do not have an explanation for this.

All this discourages the use of mini-batch optimisation on distributed data, and suggest to use instead full-batch optimisation, which automatically negates the scalability advantages of SGD. In any case, we present the speedup results for the batch version of our algorithms when processing 10 epochs.

Spec	**Value**	**Spec**	**Value**
workers	8	Data size	4,000,000
cores/worker	8	Clusters	4
memory/worker	15 GB	Dimensions	30
Clock Speed	2.3 GHz	Separability	1

Table 11: Workers and data characteristics

Cores	SGD	Momentum SGD	Nesterov SGD
4	237.82	248.99	244.32
8	180.21	181.65	184.90
16	146.65	147.61	149.03
32	131.45	132.26	127.38
64	123.24	120.19	123.28

Table 12: Times of speedup tests (s)

Figure 21 show that the algorithm's speedup is poor; it improves as the data dimensionality grows, but it is uncommon to to use standard GMMs for very high-dimensional datasets, and in general it is not recommended to perform clustering on such data without special algorithms or preprocessing[Steinbach et al. (2004)].

We note that the program took between 145 and 150 seconds to complete 10 full-batch iterations using 16 cores, depending on the particular descent direction; in the last section, we saw that for data with the exact same characteristics, Spark's GMM class took 192.23 seconds; we did not record the number of iterations elapsed in that case so we

cannot directly compare the timings, but judging by the results in subsection 5.4 in which often ten iterations were enough to get close to the solution, we would argue that in full-batch mode both algorithms give roughly the same result for a given time budget; moreover, since our program follows the distributed data processing pattern of Spark's `GaussianMixture` class, their speedup should be similar too.

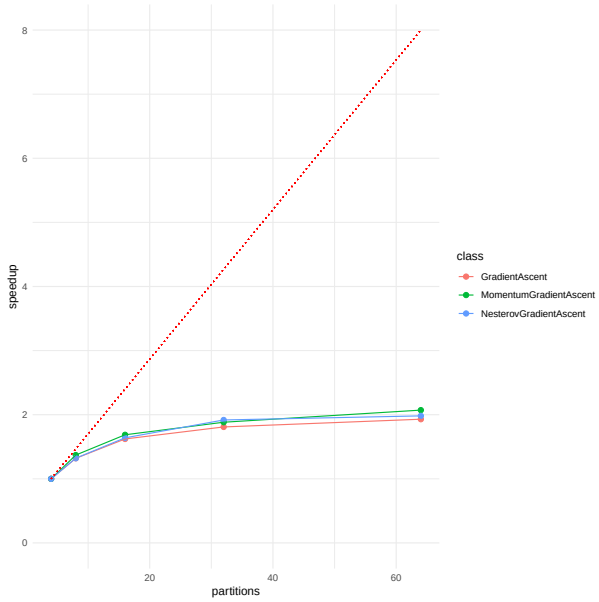


Figure 21: Full-batch gradient descent speedup

5.6 Streaming Data

Gradient-based GMMs can handle streaming data just as well as batch data since it treats data streams as a series of mini-batches; moreover, gradient-based optimisation is also appropriate for moving target optimisation, which can occur in data streams when the data distribution changes with time; the step size parameter provides an intuitive way of controlling the rate at which the model forgets previous data.

5.6.1 Forgetfulness Rate

Forgetfulness rate is an important concept in online learning and data stream problems that can be useful when we suspect that the data is non-stationary. It helps us control the importance that the model gives to past and current data, and should be related to how fast the data distribution is changing. When we talk about forgetfulness rate r we mean that we want current parameter information to contribute just $100(1-r)\%$ to the model state after processing the next batch, so, for instance, we can make the current parameter values fade at a 10% or a 5% rate with each new batch. It turns out that for the formulation we are using, the step size has a remarkably simple connection to

the forgetfulness rate; such connection is summarised in the following lemma, which is proven in appendix D:

Lemma 5.1. *Let $0 \leq r \leq 1$. For a step size of $\alpha = 2r$, the contribution of former Gaussian parameter values to the current state of the model decays at a rate of $1 - r$ per iteration.*

so if we want, for instance, a forgetfulness rate of 5%, $\alpha = 0.1$ will do. Alternatively we can instead talk about *half life*, the number of iterations that have to go by so that the current parameter values contribute to only 50% of the model's state. The conversion from r to half life h and viceversa is simple and is given by

$$r = 1 - 2^{-\frac{1}{h}}$$

The above results apply only to the Gaussian parameters; the non-linearity of the weight updates makes it difficult to make a similar analysis. However, for $r \leq \frac{1}{2}$, the forgetfullness rate of the weights is bounded below by r; in any case, being able to control the forgetfulness of the model for the location and spread parameters may be convenient.

Such an analysis is harder to do for accelerated algorithms because the acceleration terms induce additional information from past iterations in the updates, making it more difficult to determine the actual decay rate for former parameter values; however, the result for standard SGD can be taken as a guide for such algorithms.

5.6.2 Streaming Performance

We will assume a time window of one second, and test how much data the program can process in that window with different numbers of machines; by processing the data we mean updating the model and subsequently labelling the data batch with the updated model, as this would most likely be the the case in a data stream pipeline. We also assume that we will do one iteration per data batch. Up to 4 AWS instances were used for the following tests, and their technical specifications are listed in table 11.

Machines	samples	dim	Time (s)
4	220,000	30	0.97
3	180,000	30	0.97
2	165,000	30	0.99
1	135,000	30	0.91
4	140,000	50	0.99
3	123,000	50	0.98
2	110,000	50	0.98
1	95,000	50	0.93

Table 13: Single iteration times (1 machine = 8 cores)

Although the throughput is high, the time overhead of communication and synchronisation is also high; the first machine can process 135,000 points per second, while

additional 8-core machines add less than 30,000 to the processing capacity on average, in the case of 30-dimensional vectors . If subsampling is still required from the data stream, this would be more efficiently done in a stage previous to model updating, because of the issues we discussed in section 5.5.2.

6 Conclusions

In this work, we have shown that accelerated gradient algorithms outperform SGD consistently on a varied set of conditions when fitting GMMs (section 5.3); we have also shown that they are much faster and cheaper to use -in terms of hardware - than full-batch algorithms such as EM, giving results of comparable quality, thus providing a way to scale mixture modelling to millions of data points cheaply and efficiently (section 5.5.1). Performing mini-batch optimisation makes conjugate prior regularisation to degrade the final model's quality (section 5.4.5); we have proposed a logarithmic barrier regulariser that fulfils the same function, namely, avoiding covariance singularities, while being less expensive in terms of memory and CPU time and preserving the model's quality. Finally, we have derived an intuitive interpretation of the step size in standard SGD as a forgetfulness rate in streaming settings that can also be used as a guide value for accelerated algorithms, and proved that in such settings our program can process hundreds of thousands of data points per second (section 5.6).

Having derived a reformulation that makes it possible to efficiently fit GMMs with gradient-based methods, the results of [Hosseini and Sra (2017)] have opened the door to the huge number of breakthroughs that have been made for SGD-optimised models in fields like Deep Learning and High Performance Computing. Promising extensions to this work include developing asynchronous versions of distributed SGD in frameworks that are friendly to such approaches, such as TensorFlow. Another one would be developing a way to update the number of components dynamically in streaming settings. Finally, there is a large number of other descent directions that have been developed for Deep Learning and other Machine Learning models, each one with a comparative advantage, and it might be worth trying them for fitting GMMs.

A Proof of Lemma 3.1

Proof. Using the definition of S_j and $q_{\mathcal{N}}(y_i; S_j)$ and S_J in eq. 9 we have that for any $\{S_j\}, \{\omega_j\}$

$$\hat{\mathcal{L}}(Y; \{S_j\}, \{\omega_j\}) = \frac{1}{n} \sum_{i=1}^{n} \ln \left(\sum_{j=1}^{K} \pi_j q_{\mathcal{N}}(y_i; S_j) \right) =$$
$$\frac{1}{n} \sum_{i=1}^{n} \ln \left(\frac{1}{\sqrt{s}} \exp \left(\frac{1}{2} \left(1 - \frac{1}{s} \right) \right) \sum_{j=1}^{K} \pi_j \mathcal{N}(y_i; \mu_j, \Sigma_j) \right)$$

This can be separated as $\hat{\mathcal{L}}(Y; \{S_j\}, \{\omega_j\}) = f(s) + \frac{1}{n} L(X; \{\mu_j\}, \{\Sigma_j\}, \{\pi_j\})$ where $L(\cdot)$ is the log-likelihood function of the data (see equation 2) and $f(s)$ has its maximum at $s^* = 1$, therefore the result holds.

□

B Logarithmic Barrier

B.1 Formulation

To show that equation 22 is equivalent to adding a logarithmic barrier on $\det(\Sigma_j)$ to the log-likelihood loss, let us note that

$$\hat{\mathcal{L}}(Y; \{S_j\}, \{\omega_j\}) + \sum_{j=1}^{K} (\ln \det(S_j) - s) =$$
$$\hat{\mathcal{L}}(Y; \{S_j\}, \{\omega_j\}) + \sum_{j=1}^{K} (\ln \det(\Sigma_j) + \ln s - s)$$

As we saw above in A, $\hat{\mathcal{L}}(Y; \{S_j\}, \{\omega_j\}) = f(s) + \frac{1}{n} L(X, \{\mu_j\}, \{\Sigma_j\}, \{\pi_j\})$ where $L(\cdot)$ is the log-likelihood function of the data, so

$$\hat{\mathcal{L}}(Y; \{S_j\}, \{\omega_j\}) + \sum_{j=1}^{K} (\ln \det(\Sigma_j) + \ln s - s) =$$
$$f(s) + \frac{1}{n} L(X; \{\mu_j\}, \{\Sigma_j\}, \{\pi_j\}) + \sum_{j=1}^{K} (\ln \det(\Sigma_j) + \ln s - s) =$$

Rearranging, we have that

$$\hat{\mathcal{L}}(Y;\{S_j\},\{\omega_j\}) + \sum_{j=1}^{K}(\ln\det(S_j) - s) =$$
$$\frac{1}{n}L(X;\{\mu_j\},\{\Sigma_j\},\{\pi_j\}) + \sum_{j=1}^{K}(\ln\det(\Sigma_j)) + g(s)$$

Where $g(s) = f(s) + K(\ln s - s)$, which has its maximum at $s^* = 1$, so 22 preserves the maximum of a log-barrier-regularised log-likelihood loss.

B.2 Preventing Covariance Singularities

From table 1 and equation 6 we can see that the general form of the update for S_j and a single data point y_i (see eq. 9) under SGD is given by

$$S_j \leftarrow (1-\lambda) + \lambda y_i y_i^T$$
$$\lambda = \frac{1}{2}\alpha w_i$$

where α is the step size and w_i is defined as in equation 23.

Set $s = s^* = 1$, then using equations 8 and 9 we can derive the expression for the updates of Σ_j, which are

$$\Sigma_j \leftarrow (1-\lambda)\left(\Sigma_j + \lambda\,(x_i-\mu_j)\,(x_i-\mu_j)^T\right)$$

For a covariance singularity to exist, μ_j have to coincide with x_i (or at least $\mu_j \approx x_i$), so the updates of Σ_j becomes

$$\Sigma_j \leftarrow (1-\lambda)\Sigma_j$$

and the covariance matrix shrinks exponentially fast to zero.Adding a logarithmic barrier term produces instead an update of the form

$$\Sigma_j \leftarrow (1-\lambda)\left(\Sigma_j + \lambda\,(x_i-\mu_j)\,(x_i-\mu_j)^T\right) + \frac{1}{2}\alpha\Sigma_j$$

So in the case of a mean collapse, the update takes the form

$$\Sigma_j \leftarrow \left(1 + \frac{1}{2}\alpha(1 - w_i)\right)\Sigma_j$$

Since $w_i \leq 1$, this implies that $1 + \frac{1}{2}\alpha(1 - w_i) \geq 1$, preventing the shrinkage of Σ_j altogether.

C SGD Time Results

C	dim	k	MSGD		EM		LL_0	LL^*
			Time(s)	Progress	Time(s)	Progress		
0.20	50	4	14.44	98.92%	109.34	36.51%	-55.97	-28.50
0.20	50	8	25.36	83.65%	184.14	20.14%	-52.68	-26.77
0.20	50	12	38.01	85.52 %	327.39	0.56%	-49.94	-25.50
1.00	30	4	6.48	84.71%	192.23	99.99%	-32.08	-15.51
1.00	30	8	13.77	89.00%	334.87	99.99%	-28.02	-12.18
1.00	30	12	9.16	91.04 %	618.96	99.98%	-26.11	-11.15
0.20	30	4	6.10	87.80%	211.70	99.75%	-35.50	-20.20
0.20	30	8	2.45	92.50%	547.70	99.68%	-33.39	-19.28
0.20	30	12	13.53	92.86%	1116.44	98.86%	-33.27	-20.28

Table 14: Standard SGD and EM time and progress comparison

D Proof of Lemma 5.1

Proof. From the gradient expressions in table 1 and equations 6 and 9, we can derive expressions for the mean, and covariance updates. Assume $\alpha \leq 2$ and let

$$\begin{aligned}
\lambda_j =& \frac{\alpha}{2}\frac{1}{n}\sum_{i=1}^{n} w_{ij} \\
m_j =& \frac{\alpha}{2}\frac{1}{n}\sum_{i=1}^{n} w_{ij}x_i \\
V_j =& \frac{\alpha}{2}\frac{1}{n}\sum_{i=1}^{n} w_{ij}(x_i - \mu_j^{(t)})(x_i - \mu_j^{(t)})^T \\
U_j =& \left(\frac{\alpha}{2}\frac{1}{n}\sum_{i=1}^{n} w_{ij}(x_i - \mu_j^{(t)})\right)\left(\frac{\alpha}{2}\frac{1}{n}\sum_{i=1}^{n} w_{ij}(x_i - \mu_j^{(t)})\right)^T
\end{aligned}$$

where w_{ij} is the probability that x_i belongs to the current j-th component (see equation 23). Then the parameter updates at each iteration are given by

$$\mu_j^{(t+1)} \leftarrow (1-\lambda_j)\mu_j^{(t)} + m_j$$
$$\Sigma_j^{(t+1)} \leftarrow (1-\lambda_j)\Sigma_j^{(t)} + V_j - U_j$$

Both updates can be expressed as convex combinations between the current parameters and data-dependent statistics [3]. The coefficient corresponding to the current parameters in such combinations is λ_j, hence, we can choose α so that $1-\lambda_j = 1-r$. This give us the equation

$$\frac{1}{k}\sum_{j=1}^{K}(1-\lambda_j) = \frac{1}{k}\sum_{j=1}^{K}(1-\frac{1}{2n}\alpha\sum_{i=1}^{n} w_{ij}) = 1-r$$

Since $\sum_{i=1}^{n}\sum_{j=1}^{K} w_{ij} = n$, this reduce to $\alpha = 2r$, and the proof is complete. □

[3] $V_j - U_j = \frac{\alpha}{2n}\sum_{i=1}^{n} w_{ij}(x_i - \mu_j^{(t)})\left((x_i - \mu_j^{(t)}) - \frac{\alpha}{2n}\sum_{k=1}^{n} w_{kj}(x_k - \mu_j^{(t)})\right)^T$

References

Absil, P.-A., Mahony, R., and Sepulchre, R. (2007). *Optimization Algorithms on Matrix Manifolds*. Princeton University Press, Princeton, NJ, USA.

Amari, S.-i. (1993). Backpropagation and stochastic gradient descent method. 5:185–196.

Arnold, S. (2016). An introduction to distributed deep learning.

Beck, A. and Teboulle, M. (2009). A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM J. Img. Sci.*, 2(1):183–202.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg.

Bottou, L., Curtis, F. E., and Nocedal, J. (2016). Optimization methods for large-scale machine learning. cite arxiv:1606.04838.

Dasgupta, S. (1999). Learning mixtures of gaussians. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 634–, Washington, DC, USA. IEEE Computer Society.

Feldman, D., Faulkner, M., and Krause, A. (2011). Scalable training of mixture models via coresets. In Shawe-Taylor, J., Zemel, R. S., Bartlett, P. L., Pereira, F., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 24*, pages 2142–2150. Curran Associates, Inc.

Goh, G. (2017). Why momentum really works. *Distill*.

Hosseini, R. and Sra, S. (2017). An Alternative to EM for Gaussian Mixture Models: Batch and Stochastic Riemannian Optimization. *ArXiv e-prints*.

Kingma, D. P. and Ba, J. (2014). Adam: A Method for Stochastic Optimization. *ArXiv e-prints*.

Kressner, D., Steinlechner, M., and Vandereycken, B. (2014). Low-rank tensor completion by Riemannian optimization. *BIT Numer. Math.*, 54(2):447–468.

Lee, K. J., Guillemot, L., Yue, Y. L., Kramer, M., and Champion, D. J. (2012). Application of the Gaussian mixture model in pulsar astronomy - pulsar classification and candidates ranking for the Fermi 2FGL catalogue. 424:2832–2840.

Loizou, N. and Richtárik, P. (2017). Linearly convergent stochastic heavy ball method for minimizing generalization error. *ArXiv e-prints*.

Ma, J., Xu, L., and Jordan, M. I. (2000). Asymptotic convergence rate of the em algorithm for gaussian mixtures. *Neural Computation*, 12(12):2881–2907.

Maretić, I. S. and Lacković, I. (2014). Application of gaussian mixture models with expectation maximization in bacterial colonies image segmentation for automated

counting and identification. In Roa Romero, L. M., editor, *XIII Mediterranean Conference on Medical and Biological Engineering and Computing 2013*, pages 388–391, Cham. Springer International Publishing.

Nar, K. and Shankar Sastry, S. (2018). Step Size Matters in Deep Learning. *ArXiv e-prints*.

Nocedal, J. and Wright, S. J. (2006). *Numerical Optimization*. Springer, New York, 2nd edition.

Raghunathan, A., Krishnaswamy, R., and Jain, P. (2017). Learning Mixture of Gaussians with Streaming Data. *ArXiv e-prints*.

Spainhour, J. C. G., Janech, M. G., Schwacke, J. H., Velez, J. C. Q., and Ramakrishnan, V. (2014). The application of gaussian mixture models for signal quantification in maldi-tof mass spectrometry of peptides. *PLOS ONE*, 9(11):1–10.

Sra, S. and Hosseini, R. (2013). Geometric optimisation on positive definite matrices for elliptically contoured distributions. In *Advances in Neural Information Processing Systems 26*, pages 2562–2570.

Steinbach, M., Ertöz, L., and Kumar, V. (2004). *The Challenges of Clustering High Dimensional Data*, pages 273–309. Springer Berlin Heidelberg, Berlin, Heidelberg.

Sutskever, I., Martens, J., Dahl, G., and Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pages III–1139–III–1147. JMLR.org.

Verbeek, J. J., Vlassis, N., and Kröse, B. (2003). Efficient greedy learning of gaussian mixture models. *Neural Comput.*, 15(2):469–485.

Zhuang, X., Huang, J., Potamianos, G., and Hasegawa-Johnson, M. (2009). Acoustic fall detection using gaussian mixture models and gmm supervectors. In *2009 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 69–72.