`blowtorch`: an `R` package for "on-line" constrained optimization

Lucia Petito, Steven Pollack

1 Introduction

For numerical optimization, methods that use a function's gradient (and/or Hessian matrix), "gradient methods", are often easy to implement, or come pre-implemented in many programming languages. However, a problem with these methods (or at least with all the methods that come implemented in `R`) is that they are designed to search for global optima, and have no means of respecting constraints should the problem necessitate them. This drawback is a non-issue if we are able to calculate gradients and Hessians of our objective and constraint functions. Indeed, this paper will demonstrate how we can use Lagrange multipliers to turn the original problem into unconstrained minimization, and thereby use the basic steepest descent algorithm to perform "on-line" constrained optimization.

Recall that for an objective function, $f : \mathbb{R}^n \to \mathbb{R}$, and equality constraints, $G : \mathbb{R}^n \to \mathbb{R}^m$, $G(x) = (g_1(x), \dots, g_m(x))$, the standard method used to maximize f, subject to G, is via the method of Lagrange multipliers: finding $x^* \in \mathbb{R}^n$ such that

$$x^* = \operatorname{argmax}_{\substack{x\in\mathbb{R}^n: \\ G(x)=0}} f(x)$$

is equalivalent to setting $\Lambda(x, \lambda) = f(x) - \langle \lambda, G(x) \rangle$, solving $\nabla\Lambda = 0$, and checking the critical points to find which are maximizers. In fact, it is well known that the *saddle points* of Λ are the optima of f. Thus, in solving $\nabla\Lambda = 0$, the extreme values of Λ are not of interest: just finding the values which optimize Λ is incorrect. Instead, consider the function, $h : \mathbb{R}^n \times \mathbb{R}^m \to [0, \infty)$, $h(x, \lambda) = \frac{1}{2} \|\nabla\Lambda(x, \lambda)\|_2^2$.

Since $\|\cdot\|_2^2$ is a valid metric on $\mathbb{R}^m$, any value that minimizes h must also be a critical point of Λ (and vice versa), hence the set of minimizers for h are all candidate extreme values for f. An immediate consequence is that numerically minimizing h, and numerically approximating roots of h are equivalent. Thus, to (numerically) optimize f it suffices to pick whichever is least computationally intensive. In particular, we can use (stochastic) gradient descent to approximate h's roots.

To minimize h via SGD, we need to calculate ∇h. After some tedious vector calculus (the full derivation is included in the appendix), we can express ∇h as:

$$\nabla h = \begin{pmatrix} \left(\mathcal{H}(f) - \sum_{k=1}^m \lambda_k \mathcal{H}(g_k)\right)(\nabla_x f - D_x[G]\lambda) + D_x[G]G^T \\ -D_x[G]^T(\nabla_x f - D_x[G]\lambda) \end{pmatrix} \tag{1}$$

where $\mathcal{H}(f)$ is the matrix of mixed partials of f, so that $\mathcal{H}(f)_{ij} = \partial_{x_i x_j} f$, and $D_x[G] = \begin{pmatrix} \nabla_x g_1 \mid \cdots \mid \nabla_x g_m \end{pmatrix}$ is an $n \times m$ matrix whose j^{th} column is the gradient of the j^{th} constraint.

The formula for ∇h in equation 1 makes it clear that to transform our original problem from constrained optimization to unconstrained optimization we will need $\nabla_x f$, $\mathcal{H}(f)$, and for each i: g_i, $\nabla_x g_i$, and $\mathcal{H}(g_i)$. Indeed, we have made an `R` package, "`blowtorch`", which given these objects, assembles ∇h according to equation 1, and searches through the (x, λ) space via (stochastic) gradient descent.

The remainder of this paper is dedicated to examining the behaviour of `blowtorch` in the (artificial) context of mixture probability estimation. Section 2 will explain the two experiments run – one to determine if `blowtorch` is performing correctly, and another to see how we may use `blowtorch` in conjunction with other methods for more refined estimates. Section 3 will present our results. Finally, Section 4 will provide a discussion of said results, with some concluding remarks.

2 Methods

Both experiments revolved around the (toy) situation where we have bivariate data from a mixture model where we believe we know the mixture components F_i, however we want to estimate the mixture proportions, π. In particular, we created an artificial data set of 2000 bivariate observations following the hierarchial model (figure 2 shows the distribution of this data in the x, y-plane):

$$\begin{aligned} \pi &\sim \mathrm{Dir}(\alpha = (2,2,1)) \\ Z_i \,|\, \pi &\sim \mathrm{Mult}(n = 1, \theta = \pi) \qquad i = 1, 2, \ldots, 2000 \\ (X, Y) \,|\, Z_i &\sim \mathcal{N}(\mu = MZ_i, \Sigma) \qquad i = 1, 2, \ldots, 2000 \end{aligned}$$

where π, M, and Σ took values:

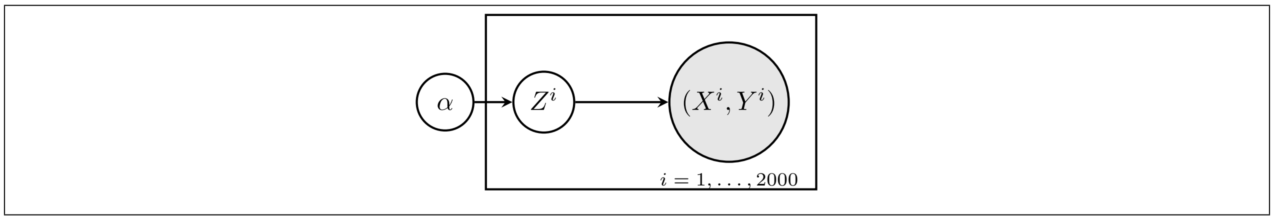

Figure 1: Plate Model for 2000 observations of bivariate data

```
print(pi <- mixtureModelParameters$mixture_probs)

# [1] 0.1252 0.6196 0.2551

print(M <- mixtureModelParameters$Mu)

#      [,1] [,2] [,3]
# [1,]   10  -10    0
# [2,]    1    0    0

print(Sigma <- mixtureModelParameters$Sigma)

#         [,1]    [,2]
# [1,]  0.8058 -0.5159
# [2,] -0.5159  2.1740
```

The log-likelihood of our data, as a function of π, is

$$\ell(\pi; \mathcal{D}) = \sum_{n=1}^{2000} \log \left(\sum_{i=1}^{3} f_i(x_n)\pi_i \right)$$

where f_i is the density associated to a $\mathcal{N}(\mu = Me_i, \Sigma)$ random-vector, and is visualized in Figure 3. Note that the surface's maxium, $\hat{\pi}$, happens at a different position than the true parameter value, π.

2.1 Experiment 1: solo-viability

To answer the question as to whether `blowtorch` can stand, by itself, as a means towards esimating π, we investigated how the (projected) path of a run of `blowtorch` depended on the size of the mini-batch used at each step. The batch sizes used where $1, 50, 100, 200$, and 500 and `blowtorch` was run until h descended below 10^{-3}, or the number of steps taken exceeded some amount that was batch-size dependent. For each particular batch size, `blowtorch` was run with various learning rates, and the largest rate that didn't send our objective function to infinity was selected. See table 1 for the complete experiment parameters.

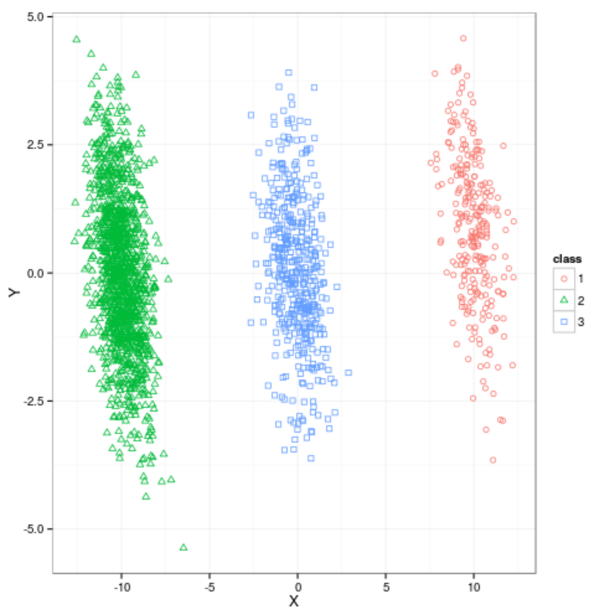

Figure 2: Scatterplot of the bivariate, mixture model, data generated according the hierarchial model scheme presented in Section 2.

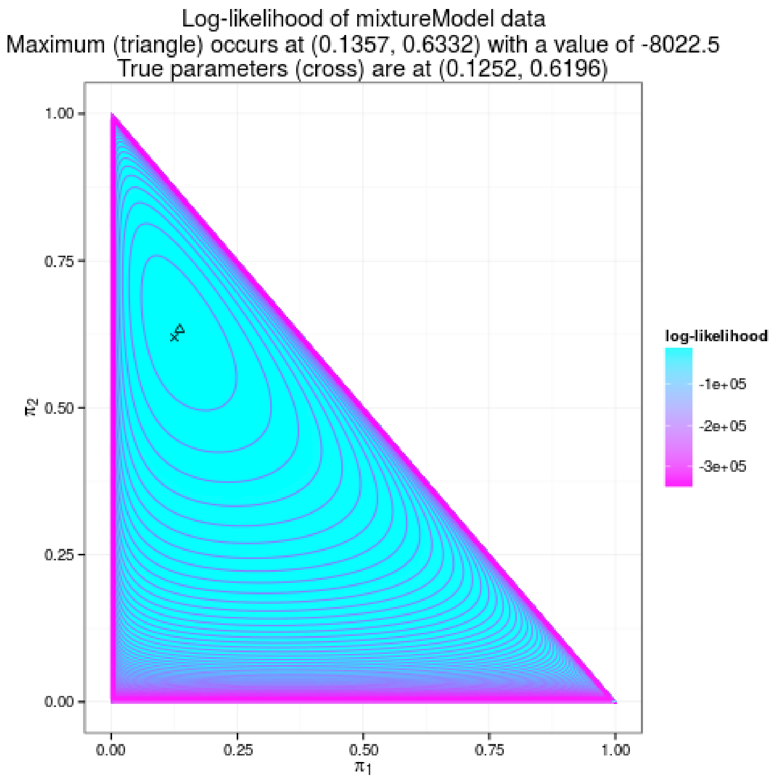

Figure 3: log-likelihood for simulated data. Note that the maximum likelihood estimate, $\hat{\pi}$, – located at the triangle – is slightly off from the true parameter, π – located at the cross.

Batch size	Learning rate	Maximum iterations	Convergence tolerance
1	0.004	20000	0.001
50	0.030	4000	0.001
100	0.030	2000	0.001
200	0.050	1000	0.001
500	0.020	500	0.001

Table 1: Batch sizes, learning rates, and stopping criterion for the various trials in experiment 1.

2.2 Experiment 2: `blowtorch` as an initial step

To answer the question as to what gains `blowtorch` may confer when used as an "initial step" for more robost optimizers, we repeated the following experiment 1,000 times:

1. Generate a random starting point, π_0 according to a Dirichlet distribution with parameter $\alpha = (4, 4, 4)$ and generate a random set of weights, $\lambda_1, \lambda_2 \sim \mathcal{N}(0, 1)$.
2. Starting at $\theta_0 = (\pi_0, \lambda_1, \lambda_2)$:
   (a) Run `blowtorch` for either 80 iterations, or until $h < 10^{-3}$, with a batch size of 50 observations, and a learning rate of 0.025.
   (b) Run `optim()` – the `R` optmization routine – with conjugate gradient (CG), and BFGS algorithms.
3. If `blowtorch` successively completes in step 2(a), take its final values as initial values for runs of CG and BFGS in step 2(b)

The run-times and returned values for steps 2 and 3 are presented in the following section.

3 Results

3.1 Experiment 1

Figures 4 and 5 show the (projected) paths through the (π_1, π_2)-plane. Both figures maintain the convention that the final point's size should be inversely proportion to the value of h at the point. Hence, the larger a point, the closer h is to zero, and therefore the "closer" the point is to being an acceptable solution to our original optimization problem. It stands to point out that these are the (π_1, π_2) paths of a particular run of `blowtorch`. For no point in either plot are we guaranteed that $\pi_1 + \pi_2 + \pi_3 = 1$; we must rely on the size of h at any iteration to give use a sense of how closely our estimates are adhering to our imposed constraints.

In the context of our toy mixture model setting, the effect of batch size on the SGD path is quite clear: as batch sizes increases the path becomes less noisy, and we seem to be able to get closer to the true parameters with the same number of passes through the data (epochs). This relationship is vividly exhibited in figure 5 where the SGD path for the trivial batch spends almost all of its time outside of the feasible region. Increasing the batch size to 50 points allows the path to return to the feasible region and converge upon a reasonable approximation to the parameter.

Note that all SGD paths but the one corresponding to the trivial batch were able to bring h below the convergence tolerance, 0.001.

3.2 Experiment 2

Figures 6 and 7 contain violin plots for the run time distributions of BFGS and CG, respectively, when initialized at a random value, and the run time distributions for "SGD then BFGS" and "SGD then CG", respectively. I.e., the latter distributions measure the total time it took to run `blowtorch` and then BFGS (or CG) using `blowtorch`'s final values. The mean and median run times for all of the methods used in Experiment 2 are available in table 2. Note that while the median run time for BFGS is lower than "SGD then BFGS" the top-5 worst run times for BFGS are significantly higher than the top-5 worst run times for

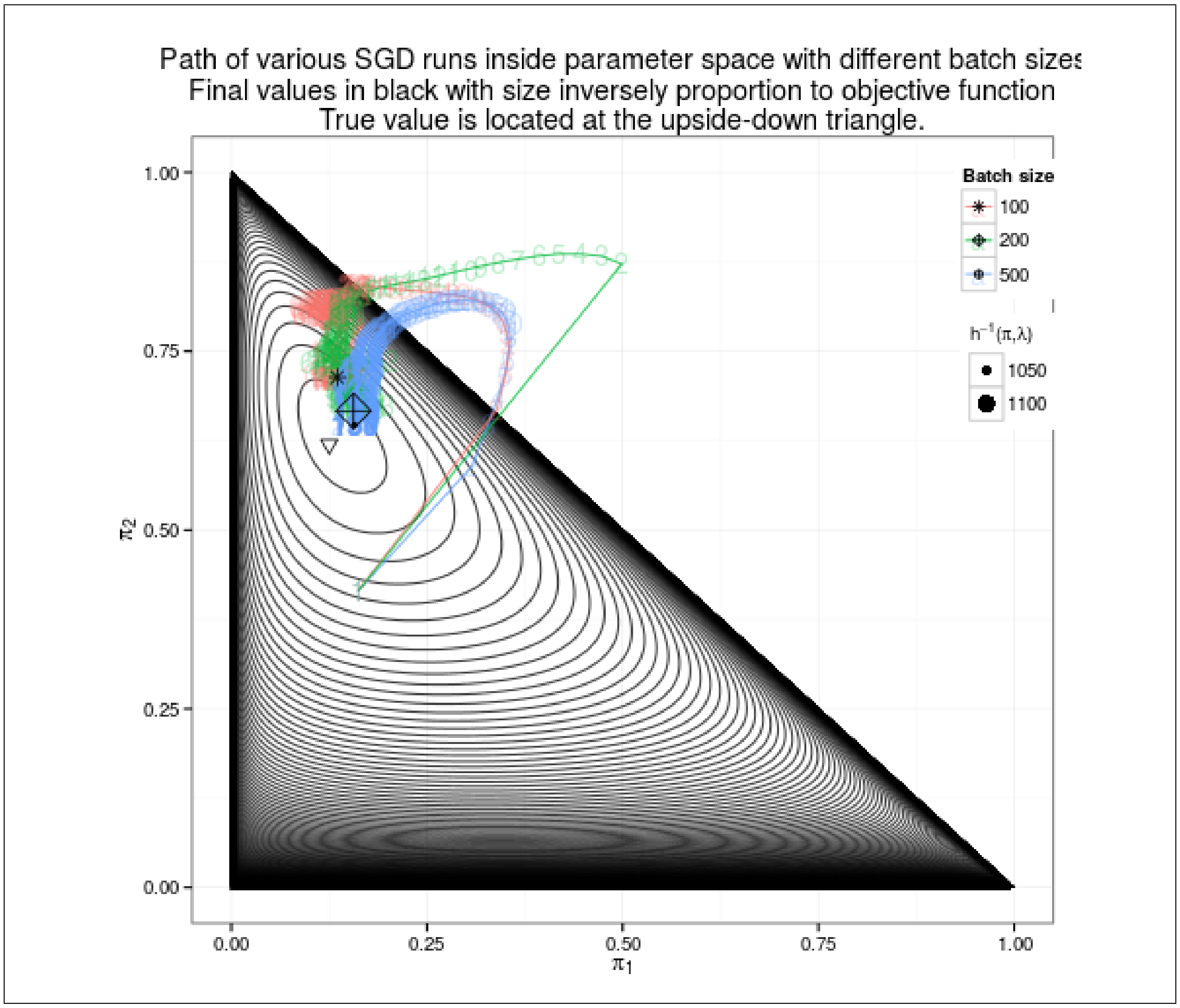

Figure 4: Path of `blowtorch` for batch sizes 100 (red, asterisk), 200 (green, diamond), and 500 (blue, circle). Contours represent changes in the log-likelihood by 50 points.

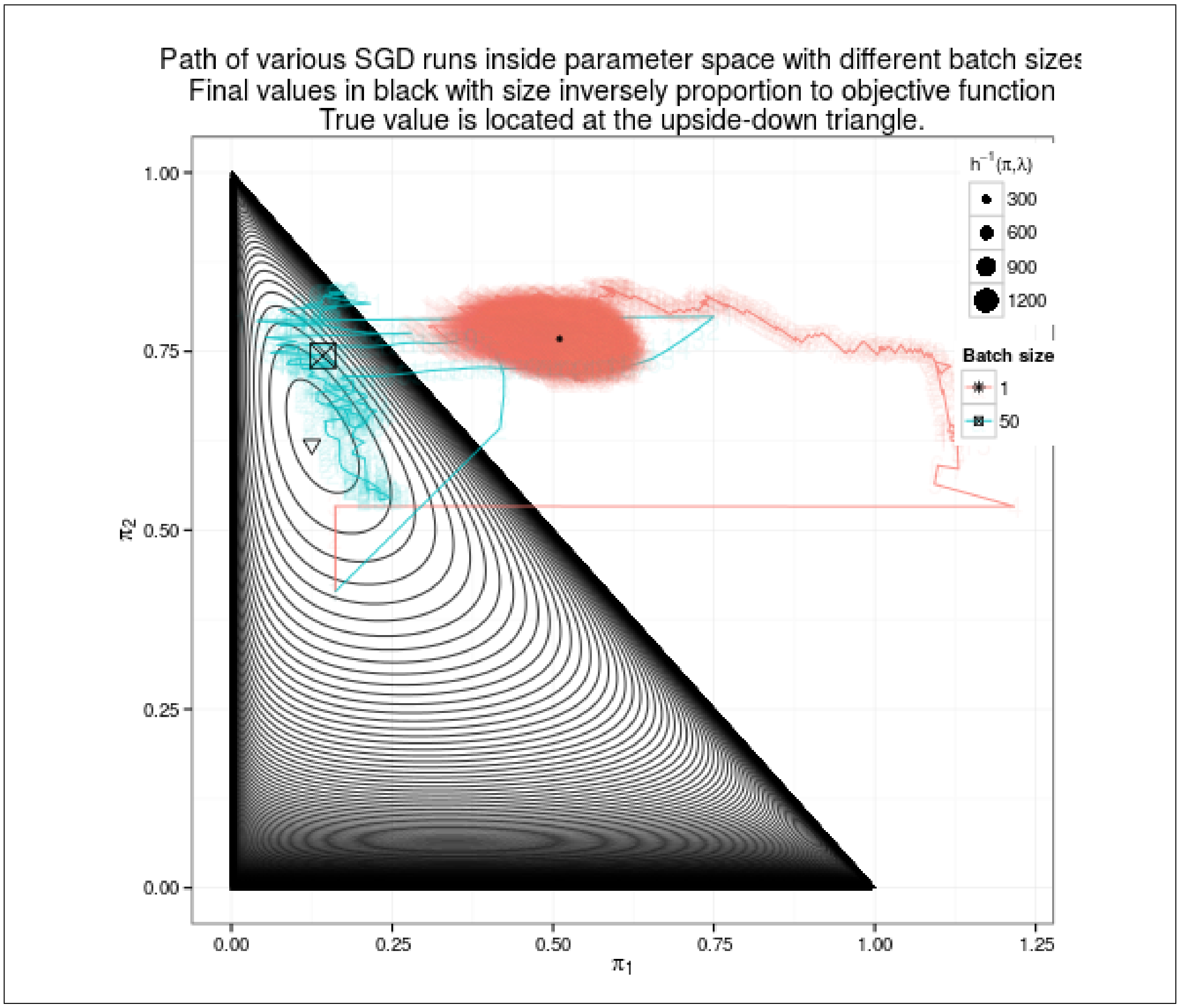

Figure 5: Path of `blowtorch` for batch sizes 1 (red, asterisk) and 50 (green, square). Notice that these paths are noisier than those of the larger batch sizes. Contours represent changes in the log-likelihood by 50 points.

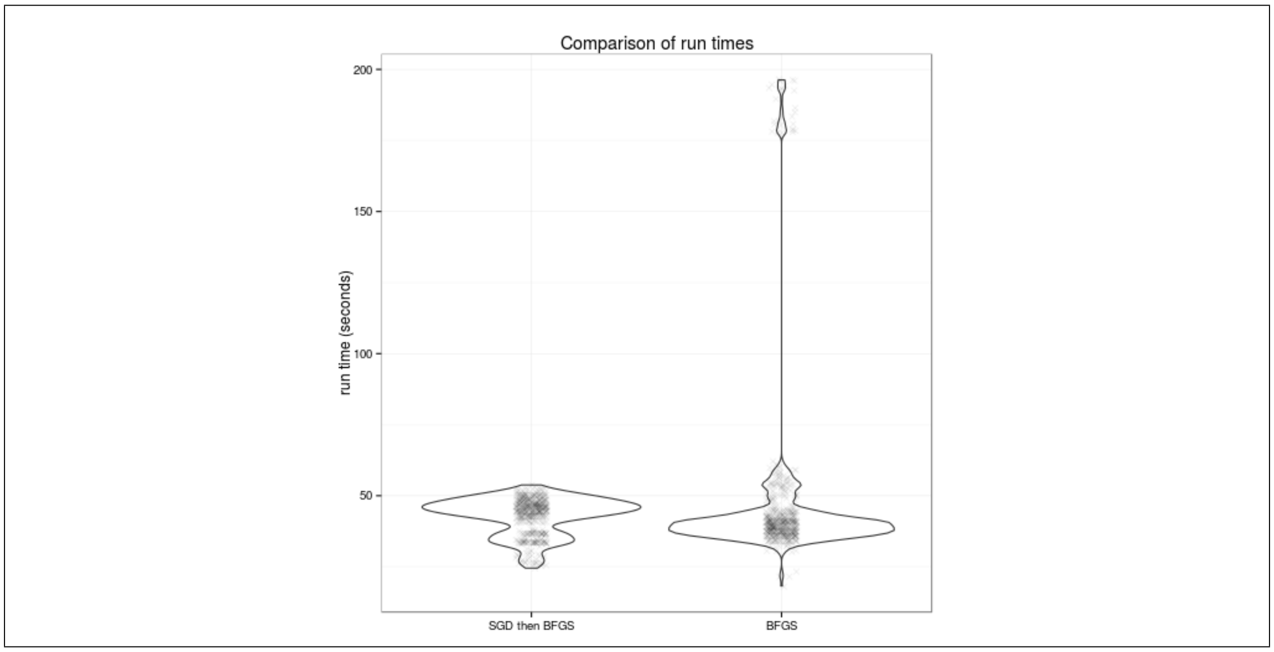

Figure 6: Run time comparison for instances when BFGS started from a `blowtorch` final value as opposed to the run time had BGFS used the same initial value as `blowtorch`.

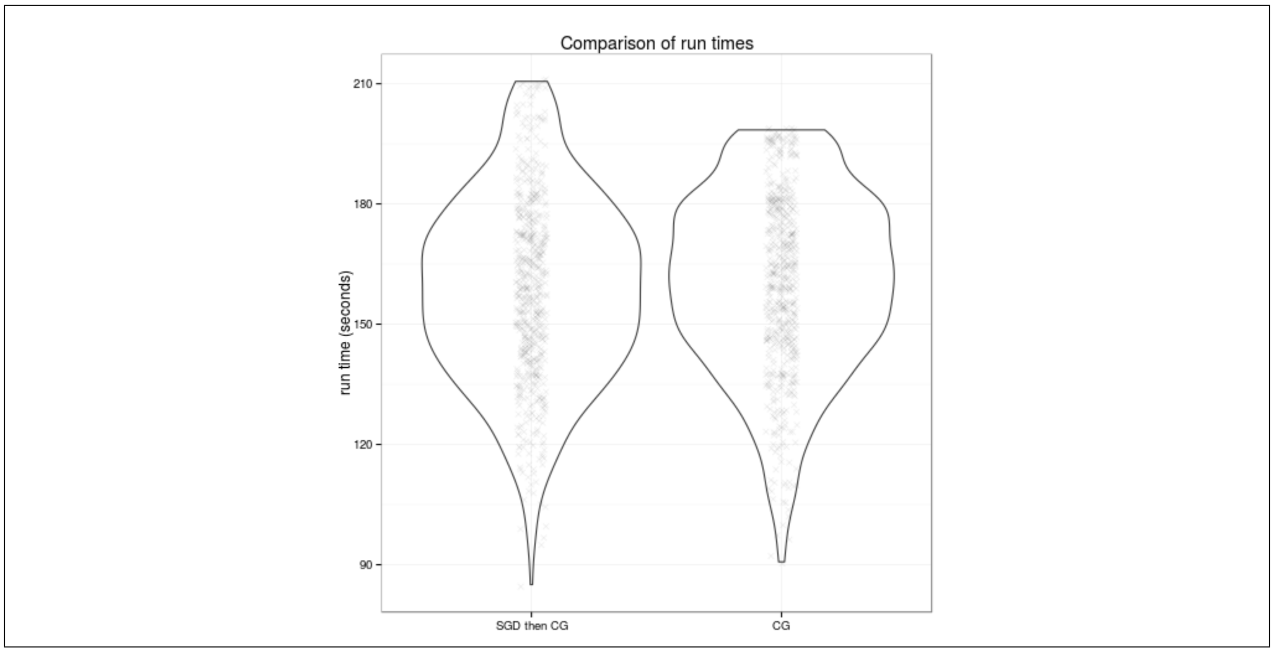

Figure 7: Run time comparison for instances when CG started from a `blowtorch` final value as opposed to the run time had CG used the same initial value as `blowtorch`.

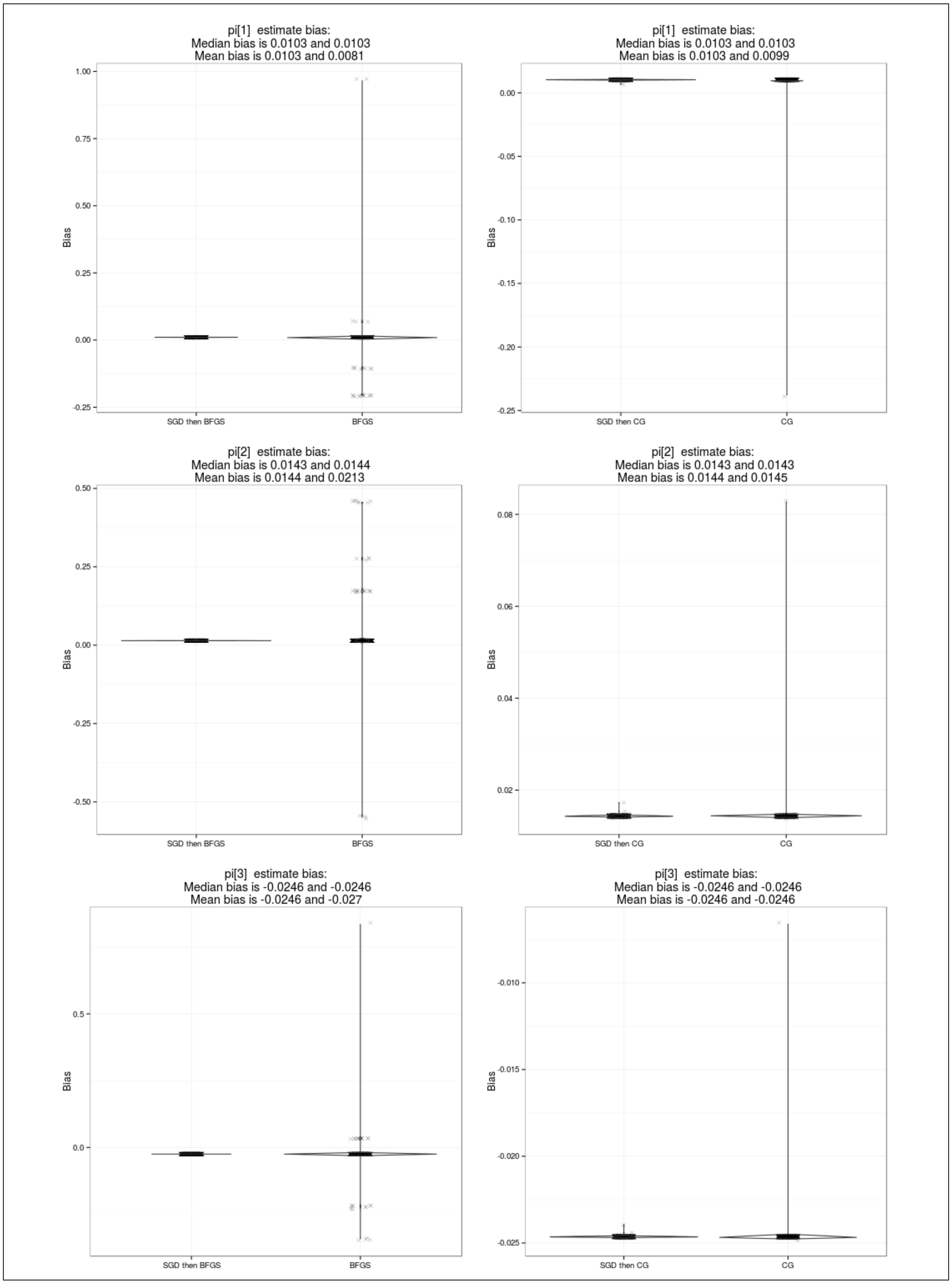

Figure 8: The bias for the final estimates of π given `blowtorch` found "initial values", as well as random initial values.

"SGD then BFGS". Although the worst-case run times for "SGD then CG" were higher than CG's, the mean, median, and best-case run times for "SGD then CG" are all lower than CG's.

Method	Mean run time	Median run time
BFGS	47.00	40.10
BFGS (warm start)	30.59	32.33
SGD then BFGS	42.92	44.50
blowtorch	12.33	12.33
CG	160.09	161.60
CG (warm start)	146.31	146.43
SGD then CG	158.63	158.83

Table 2: Mean and Median run times for the various methods in Experiment 2.

Figure 8 contains violin plots for the distribution of $\hat{\pi}_i - \pi_i$ for each method, BFGS, "SGD then BFGS", CG and "SGD then CG". Not surprisingly, everywhere `blowtorch` was employed, the support of the distribution shrank.

4 Discussion

As the theory suggests, and figures 4 and 5 indicate, "on-line" learning of parameters *with constraints* can be done within an SGD framework. Indeed, for our simulated scenario, we needed to do non-trivial "batch" learning; however, in real world applications where the data set is not 2,000 but 2,000,000,000 observations, a batch-size of 50 (or 100), as opposed to a full batch, should not be so computationally burdensome as to make this caveat a true issue.

Perhaps not as astonishing are the results of experiment 2, which aimed to investigate what speed and accuracy gains could be had when using `blowtorch` as a means to achieving a more educated initial value for more robust optimization methods like BFGS or CG. For the cost of around 12 seconds (on average), an application of `blowtorch` greatly reduces the variance in the final values returned by BFGS and CG. Even more surprising, though, is the fact that the time spent having `blowtorch` search through the parameter space does not (significantly) add to overall run-time: the mean run times for "SGD then BFGS" and "SGD then CG" are below their respective counter-parts.

These results lead to a few questions, however. First, if it's possible to run a method like BFGS or CG, why bother with a two-step procedure like "SGD then BFGS" or "SGD then CG"? Second, how should we interpret the results of experiment 2, in the context of big data?

Towards the first question, it should be said: if the data in question does not qualify as "big" (i.e., it can fit in memory), there is most likely no reason to use a two-step like "SGD then BFGS". In fact, there is most likely no reason to use a generic optimization routine like BFGS of CG. If the data can fit in memory, there is no reason to not use a dedicated constrained optimization package like `Rsolnp`.

Towards the second question, with the answer to the first in mind: we shouldn't extrapolate too far beyond this example. If the data is truly "big", a single iteration of BFGS or CG would probably be very computationally costly. If the data is not "big", we advocate for using an appropriate software package for constrained optimization. Hence, in either situation, we would not suggest running BFGS or CG, and thus we emphasize that this is a contrived example to demonstrate that `blowtorch` *can* be used to search inside the parameter space for rough approximations to our true parameters. Not that it necessarily *should* be used.

Though it stands to be investigated further, the decision tree in figure 9 succinctly expresses our conclusions, based on the results of experiments 1 and 2. In particular, we believe that unless a rough approximation is acceptable (or unless the data is sufficiently large), you should employ a constrained optimizer such as `Rsolnp`. In our mixture model example, `Rsolnp` found the maximum likelihood estimates in approximately 0.98 seconds, using the full data.

A A friendly, self-contained derivation of ∇h

Suppose we want to optimize a function $f : \mathbb{R}^n \to \mathbb{R}$, according to a set of constraints, $g_1(x) = 0, \dots, g_m(x) = 0$, $g_i : \mathbb{R}^n \to \mathbb{R}$, $i = 1, 2, \dots, m$. Compactly, we'll write $G(x) = (g_1(x), \dots, g_m(x))$, and consider our problem with Lagrange multipliers; That is, instead of optimizing f subject to $G(x) = 0$, we will find the critical points of the Lagrangian, $\Lambda(\theta)$,

$$\Lambda(\theta) = f(x) - G(x)\lambda$$

where $\theta = (x, \lambda) \in \mathbb{R}^{n+m}$, and and λ is an $m \times 1$ column vector with real-valued entries.

Now, if f was to be optimized with no constraints on the domain of f, we could use stochastic gradient ascent (SGA), or descent for that matter, to search for extreme values. However, the constraints, and consequently the need to work with a higher dimensional object make an immediate application of SGA inappropriate. Indeed, we are not interested in optimal points for Λ, but in points where $\nabla_\theta \Lambda = 0$, so called "critical points". While critical points do correspond to extreme values, they also correspond to saddle points (in one dimension, these are also called "inflection points"). Thus, the set of critical points for Λ may be a proper superset of the set of points where Λ is extreme and in using SGA to only find extreme points, there's always a chance that we converge upon coordinates that do not optimize our original problem.

A sensible solution to this hurdle is to, instead, consider the function $h : \mathbb{R}^{n+m} \to \mathbb{R}$,

$$h(\theta) = -\frac{1}{2} \left\| \nabla_\theta \Lambda(\theta) \right\|^2$$

and perform SGA on $h(\theta)$. Since $h(\theta) \leq 0$ for all θ, and $h(\theta) = 0$ precisely when $\nabla_\theta \Lambda = 0$, we have a convenient means of checking if an SGA scheme has converged upon a maximum and therefore optimal coordinates from our original optimization problem. However, to perform SGA on h, we need $\nabla_\theta h$. As the following section demonstrates, the calculus needed to find this value isn't profound, just unfortunately tedious.

Calculating $\nabla_\theta h$: Before getting too deep in the vector-calulus, let's agree on some notation. First, for $x = (x_1, \dots, x_n)$, we'll write $\partial_{x_i} \triangleq \partial/\partial_{x_i}$, and

$$D_x = \begin{pmatrix} \partial_{x_1} \\ \vdots \\ \partial_{x_n} \end{pmatrix}$$

So that for a real-valued $f(x)$, $D_x f = (\partial_{x_1} f, \dots, \partial_{x_n} f)^T$, and for vector valued $G(x) = (g_1(x), \dots, g_m(x)) \in \mathbb{R}^m$, $D_x G$ is defined recursively as:

$$D_x G = \begin{pmatrix} D_x g_1 & | & D_x g_2 & | & \cdots & | & D_x g_m \end{pmatrix} = \begin{pmatrix} \partial_{x_1} g_1 & \partial_{x_1} g_2 & \cdots & \partial_{x_1} g_m \\ \partial_{x_2} g_1 & \partial_{x_2} g_2 & \cdots & \partial_{x_2} g_m \\ \vdots & \vdots & \ddots & \vdots \\ \partial_{x_n} g_1 & \partial_{x_n} g_2 & \cdots & \partial_{x_n} g_m \end{pmatrix}$$

Furthermore, we'll let $\mathcal{H}(f)$ denote the Hessian of f. That is, $\mathcal{H}(f)_{ij} = \partial_{x_i x_j} f$ (and if $f \in C^1(\mathbb{R})$, $\mathcal{H}(f)$ is symmetric). For G, we need something slightly more elaborate: $\mathcal{H}(G)^k_{ij} = \mathcal{H}(g_k)_{ij}$. Hence, $\mathcal{H}(G)$ is a 3-dimensional array whose last index points to the Hessian of a particular component of G. For the sake of this paper, the last index of $\mathcal{H}(G)$ is the "fastest", in that for $\lambda \in \mathbb{R}^m$, the object $\langle \lambda, \mathcal{H}(G) \rangle$ is an $n \times n$ matrix and $\langle \lambda, \mathcal{H}(G) \rangle_{ij} = \sum_{k=1}^{m} \lambda_k \mathcal{H}(G)^k_{ij}$.

Armed with these definition, let's do some calculus:

$$\begin{aligned}
\nabla_\theta \Lambda(\theta) &= D_\theta \Lambda(\theta) \\
&= (D_x \Lambda(x, \lambda), D_\lambda \Lambda(x, \lambda))^T \\
&= \begin{pmatrix} D_x f - D_x[G\lambda] \\ -G^T \end{pmatrix} \\
&= \begin{pmatrix} (D_x f - D_x[G]\lambda \\ -G^T \end{pmatrix}
\end{aligned}$$

Hence,

$$\begin{aligned}
h(\theta) &= \frac{1}{2} \left\| \nabla_\theta \Lambda(\theta) \right\|^2 \\
&= \frac{1}{2} \left(D_\theta \Lambda(\theta) \right)^T \left(D_\theta \Lambda(\theta) \right) \\
&= \frac{1}{2} \left(D_x f - D_x[G]\lambda \right)^T \left(D_x f - D_x[G]\lambda \right) + \frac{1}{2} \left(-G^T \right)^T \left(-G^T \right) \\
&= \frac{1}{2} \left(D_x f - D_x[G]\lambda \right)^T \left(D_x f - D_x[G]\lambda \right) + \frac{1}{2} G G^T
\end{aligned}$$

Now, $F \triangleq (D_x f - (D_x G)\lambda)^T$ is a $1 \times n$ (row) vector. Thus, if F_j denotes the j^{th} component of F, then

$$D_\theta \left[\frac{1}{2} F^T F \right] = D_\theta \left[\frac{1}{2} \sum_{j=1}^{n} F_j^2 \right] = \sum_{j=1}^{n} D_\theta[F_j] F_j = D_\theta[F] F^T$$

Consequently, we see that

$$D_\theta[h] = D_\theta[F] F^T + D_\theta[G] G^T$$

So our problem has reduced to calculating $D_\theta[F]$ and $D_\theta[G]$. The latter is trivial:

$$D_\theta[G] = \begin{pmatrix} D_x g_1 & D_x g_2 & \cdots & D_x g_m \\ D_\lambda g_1 & D_\lambda g_2 & \cdots & D_\lambda g_m \end{pmatrix} = \begin{pmatrix} D_x[G] \\ 0 \end{pmatrix}$$

For the former, we'll consider building up $D_\theta[F]$ column-by-column. First, let's write out F_j and consider $D_x[F_j]$ and $D_\lambda[F_j]$.

$$F_j = \partial_{x_j} f - \sum_{\ell=1}^{m} \lambda_\ell \partial_{x_j} g_\ell$$

Hence,

$$\begin{aligned}
D_x[F_j] &= D_x[\partial_{x_j} f] - D_x \left[\sum_{\ell=1}^{m} \lambda_\ell \partial_{x_j} g_\ell \right] \\
&= \begin{pmatrix} \partial_{x_1 x_j} f \\ \vdots \\ \partial_{x_n x_j} f \end{pmatrix} - \sum_{\ell=1}^{m} \lambda_\ell \begin{pmatrix} \partial_{x_1 x_j} g_\ell \\ \vdots \\ \partial_{x_n x_j} g_\ell \end{pmatrix} \\
&= \mathcal{H}(f) e_j - \sum_{k=1}^{m} \lambda_k \mathcal{H}(g_k) e_j
\end{aligned}$$

where $e_j \in \mathbb{R}^n$ is the j^{th} standard basis vector. Furthermore,

$$\begin{aligned}
D_\lambda[F_j] &= D_\lambda[\partial_{x_j} f] - D_\lambda\left[\sum_{\ell=1}^{m} \lambda_\ell \partial_{x_j} g_\ell\right] \\
&= -\sum_{\ell=1}^{m} D_\lambda[\lambda_\ell]\partial_{x_j} g_\ell \\
&= -D_x[G]^T e_j
\end{aligned}$$

Thus,

$$\begin{aligned}
D_\theta[F] &= \big(D_\theta F_1 \mid D_\theta F_2 \mid \cdots \mid D_\theta F_n\big) \\
&= \begin{pmatrix} \mathcal{H}(f)e_1 - \sum_{k=1}^m \lambda_k \mathcal{H}(g_k)e_1 & \cdots & \mathcal{H}(f)e_n - \sum_{k=1}^m \lambda_k \mathcal{H}(g_k)e_n \\ -D_x[G]^T e_1 & \cdots & -D_x[G]^T e_n \end{pmatrix} \\
&= \begin{pmatrix} \mathcal{H}(f) - \langle \lambda, \mathcal{H}(G)\rangle \\ -D_x[G]^T \end{pmatrix}
\end{aligned}$$

Putting this all together, we have

$$\begin{aligned}
D_\theta[h] &= \begin{pmatrix} \mathcal{H}(f) - \langle \lambda, \mathcal{H}(G)\rangle \\ -D_x[G]^T \end{pmatrix} (D_x f - D_x[G]\lambda) + \begin{pmatrix} D_x[G]G^T \\ 0_{m\times 1} \end{pmatrix} \\
&= \begin{pmatrix} (\mathcal{H}(f) - \langle \lambda, \mathcal{H}(G)\rangle)(D_x f - D_x[G]\lambda) + D_x[G]G^T \\ -D_x[G]^T (D_x f - D_x[G]\lambda) \end{pmatrix}
\end{aligned}$$

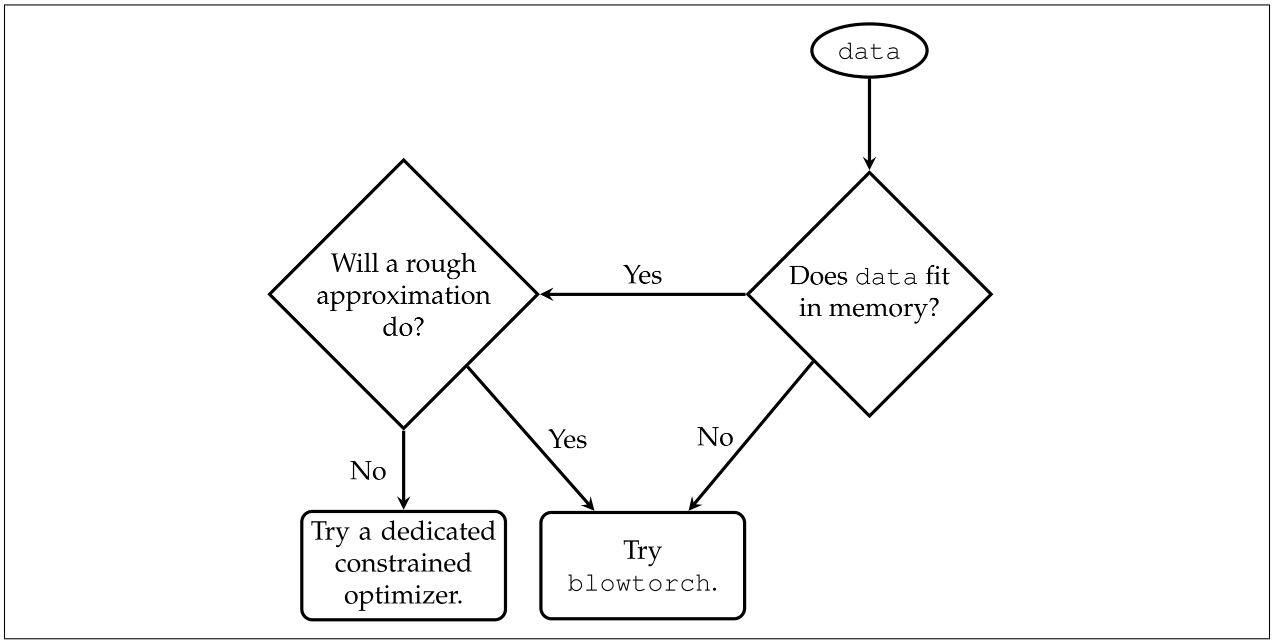
data
Does data fit in memory?
Yes
Will a rough approximation do?
No
Yes
No
Try a dedicated constrained optimizer.
Try blowtorch.

Figure 9: Decision tree