Computational Skills for Biostatistics I: Lecture 7

Amy Willis, Biostatistics, UW

15 May, 2019

Algorithms

This lecture is intended to give a very fast and broad overview of some common algorithms in statistics

- Matrix inversion
- Newton-Raphson
- Gradient descent
- Linear programs
- Monte Carlo integration
- Markov Chain Monte Carlo
- Metropolis Hastings
- Hamiltonian Monte Carlo

Every single one of these topics could be a quarter-long course

Thanks

Many thanks to Mauricio Sadinle, Dan Kowal, Noah Simon, Mike Betancourt, Mark Schmidt, and many others for resources used to make these slides

How do we compare algorithms

- Arithmetic complexity: number of arithmetic operations it takes to run an algorithm

Computational complexity

Example: How many arithmetic operations does it take to multiply two $n \times n$ matrices: $C = AB$

$$c_{ij} = \begin{bmatrix} a_{i1}, a_{i1}, \ldots, a_{in} \end{bmatrix} \begin{bmatrix} b_{1j} \\ b_{2j} \\ \vdots \\ b_{nj} \end{bmatrix} = \sum_{h=1}^{n} a_{ih} b_{hj}$$

Computational complexity

Example: How many arithmetic operations does it take to multiply two $n \times n$ matrices: $C = AB$

For each (i, j) we do n multiplications and $n - 1$ sums

We need $n^2(2n - 1)$ arithmetic operations (fewer operations if we know the structure!)

Big O notation

Goal: to describe asymptotic behaviour of a function

How do the computational requirements of the algorithm grow as a function of the input?

Big O notation

Example. Say your algorithm takes $f(n)$ arithmetic operations, e.g.

$$f(n) = \log(n) + 6n + 4n^2$$

Big O notation

Example. Say your algorithm takes $f(n)$ arithmetic operations, e.g.

$$f(n) = \log(n) + 6n + 4n^2$$

As n grows, the dominant term in the sum is n^2 (even the constant becomes irrelevant when $n \to \infty$)

The fastest growing term determines the order of $f(n)$

Big O notation

Definition. We say $f(n) = O(g(n))$ if and only if there exists a constant $M > 0$ and a real number n_0 such that

$$|f(n)| \le M|g(n)| \quad \text{for all} \quad n \ge n_0.$$

Big O notation

Where n is the size of a matrix:

- (Naive) matrix multiplication: $O(n^3)$
- (Naive) matrix inversion: $O(n^3)$

We can also talk about different inputs, e.g., magnitude of inputs in bits

Matrix inversion

Matrix inversion is extremely common in statistics

What is your favourite example?

Matrix inversion

We can write the least squares estimate as

$$\hat{\beta} = (X^T X)^{-1} X^T Y$$

or as the solution to

$$(X^T X)\beta = X^T Y$$

Which is faster?

General problem

- A : $n \times n$ full-rank matrix
- b : $n \times 1$ vector
- $z^* = A^{-1}b$: solution of system of linear equations $Az = b$

This **does not mean** that you should compute A^{-1} and then multiply by b

Matrix inversion

Fact: solving systems of equations is often faster (and never slower) than finding inverses

- If you can, avoid computing inverses!
- Instead, **just solve the system of linear equations** $Az = b$

Note: A^{-1} is the solution to $AZ = I_n$

If you can, avoid computing inverses!

Both solving $Az = b$ and computing A^{-1} by Gaussian elimination have arithmetic complexity of $O(n^3)$

While they have the same asymptotic complexity, solving $Az = b$ is much faster!

- **The big O notation can be misleading!**

Note: there exist faster algorithms than $O(n^3)$ for computing inverses but many are not practical for usual values of n

Practical implementation in R

```r
solve( t(X) %*% X ) # find inverse
solve( t(X) %*% X, t(X) %*% Y ) # solve (X^TX)\beta=X^TY
```

Regression example

```
X <- replicate(2000, rnorm(10000)); Y <- rnorm(10000)
system.time( b1 <- solve( t(X) %*% X ) %*% t(X) %*% Y )
```

```
##    user  system elapsed
##  56.735   0.446  59.781
```

```
system.time( b2 <- solve( t(X) %*% X,
                          t(X) %*% Y ) )
```

```
##    user  system elapsed
##  24.585   0.161  25.129
```

```
sum((b1 - b2)^2)
```

```
## [1] 1.09786e-29
```

Regression example

```
ops <- function(h){ Xh <- X[,1:h]
 t1 <- system.time( solve( t(Xh) %*% Xh ) %*%
                      t(Xh) %*% Y )[3]
 t2 <- system.time( solve( t(Xh) %*% Xh,
                           t(Xh) %*% Y ) )[3]
 data.frame("inverse" = t1, "solve" = t2)
}
ps <- seq(from = 50, to = 300, by = 50)
times <- sapply(ps, ops)
colnames(times) <- paste("p =", ps)
```

Regression example

```r
times
```

```
##          p = 50 p = 100 p = 150 p = 200 p = 250 p = 300
## inverse 0.031  0.123   0.275   0.502   0.704   1.159
## solve   0.014  0.086   0.193   0.275   0.362   0.607
```

About twice as fast to solve a linear system!

Matrix inversion

Conclusions

- avoid if possible
- solve systems instead
- used closed form solution for special cases (e.g., diagonal, block)

Algorithms

- ~~Matrix inversion~~
- Newton-Raphson
- Gradient descent
- Linear programs
- Monte Carlo integration
- Markov Chain Monte Carlo
- Metropolis Hastings
- Hamiltonian Monte Carlo

Newton Raphson

Goal: Solve $f(x) = 0$

Idea: Taylor expand around x_n:

$$f(x) \approx f(x_n) + f'(x_n)(x - x_n)$$

then solve for $f(x) = 0$:

$$x = x_n - \frac{f(x_n)}{f'(x_n)}$$

Newton Raphson

Goal: Solve $f(x) = 0$

Algorithm: start at reasonable guess x_0, then set

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Pros: easy, converges quickly

Cons: needs smooth functions and first derivatives

Newton Raphson: finding stationary points

Goal: Solve $f'(x) = 0$

Algorithm: start at reasonable guess x_0, then set

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

or for multivariate functions,

$$x_{n+1} = x_n - [\nabla^2 f|_{x_n}]^{-1} \nabla f|_{x_n}$$

Newton Rapson: finding stationary points

Goal: Solve $f'(x) = 0$

Common modification:

$$x_{n+1} = x_n - \gamma[\nabla^2 f|_{x_n}]^{-1}[\nabla f|_{x_n}]$$

for $\gamma \in (0, 1)$

- Pros: more stable
- Cons: slower

Gradient descent

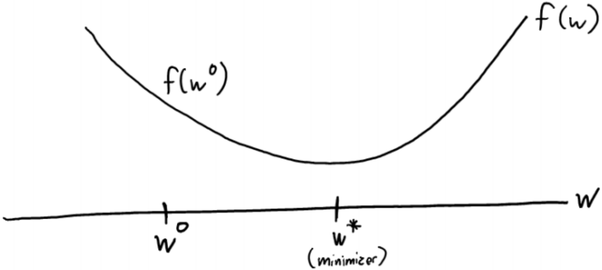
Goal: minimise $f(x)$

Algorithm: move in the steepest direction

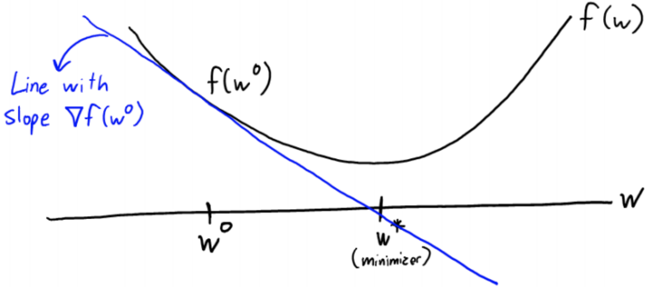
$$x_{n+1} = x_n - \gamma \nabla f(x_n)$$

for small γ

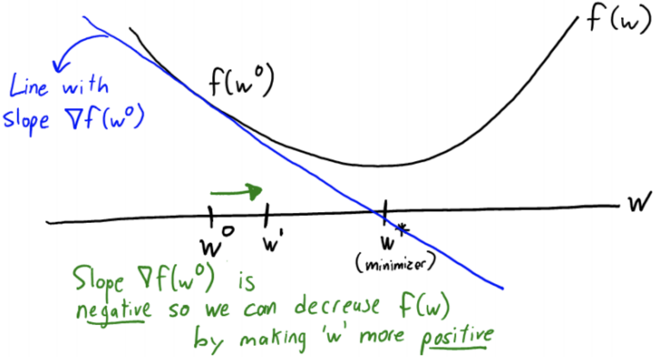
Gradient descent



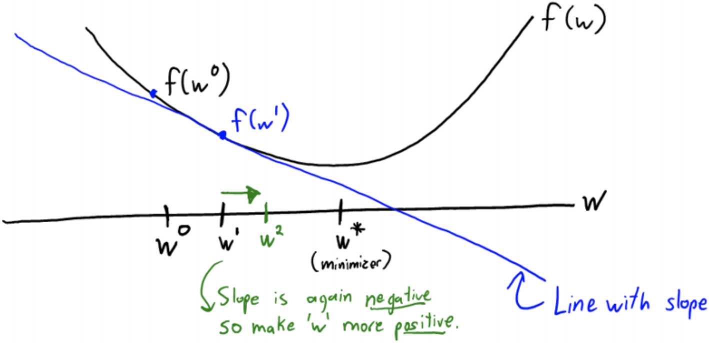
Gradient descent



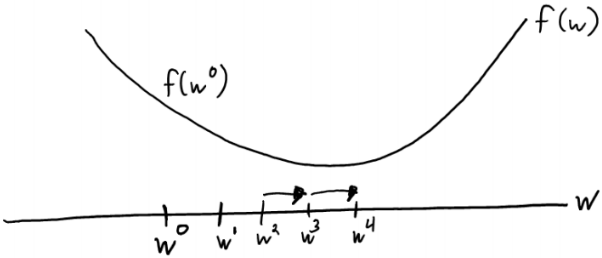
Gradient descent



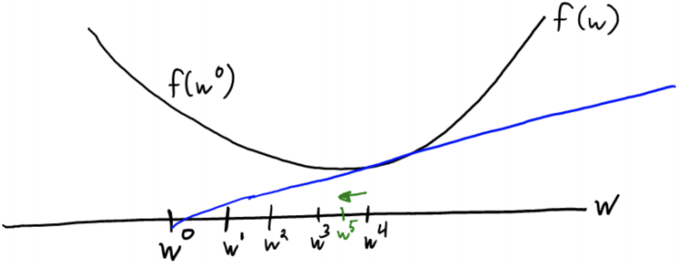
Gradient descent



Gradient descent



Gradient descent



Now the slope $\nabla f(w^4)$ is positive
so we move in the negative direction.

Gradient descent

Also called steepest descent

- Pros
  - Don't use second derivatives (curvature)
  - Don't need to take an inverse
  - Guarantees exist for convex functions
- Cons
  - Theoretically not as fast

Linear programming

Goal: Solve a linear objective function with linear constraints

Minimise $c^T x$ subject to $Ax \leq b$ and $x \geq 0$

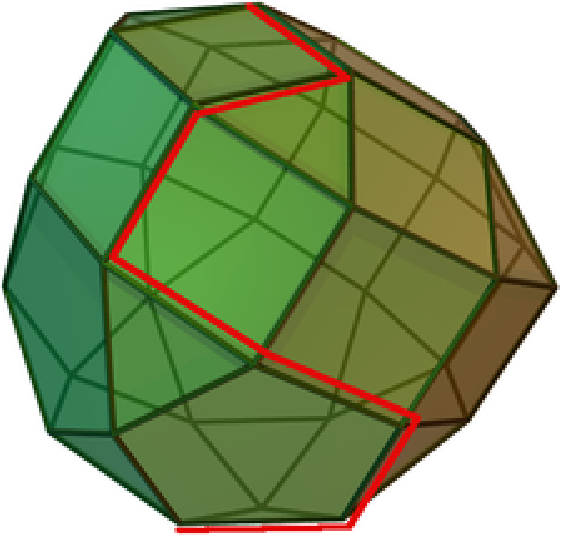
Related: Linear integer programming

Linear programming

Minimise $c^T x \quad$ subject to $Ax \leq b \quad$ and $x \geq 0$

- Solution exists on boundary of convex polytope
- Common algorithm: Simplex algorithm
- Exponential at worst case but typically outperforms provably better algorithms in practice

Simplex Algorithm

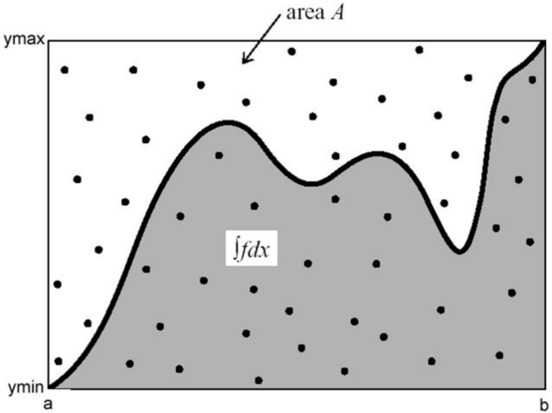


Algorithms

- ~~Matrix inversion~~
- ~~Newton-Raphson~~
- ~~Gradient descent~~
- ~~Linear programs~~
- Monte Carlo integration
- Markov Chain Monte Carlo
- Metropolis Hastings
- Hamiltonian Monte Carlo

Monte Carlo

Monte Carlo typically refers to a numerical integration procedure



Monte Carlo

Monte Carlo methods are extremely common in Bayesian statistics where it is difficult to obtain the posterior distribution in closed form

That is, we can't even maximise the posterior distribution because we can't even write it down!

Monte Carlo

However, we don't often care about finding the posterior distribution – typically we care about functionals

- moments
- quantiles

Many of these can be written as integrals!

Monte Carlo

Problem: Find $\int_{\mathbb{R}^k} g(x)f(x)dx$

Setting: it is easy to simulate from $f(x)$

Algorithm: Draw $X_1, \ldots, X_n \overset{iid}{\sim} f$ and estimate $\int_{\mathbb{R}^k} g(x)f(x)dx$ by $\frac{1}{N}\sum_{i=1}^{N} g(X_i)$

By the law of large numbers, this is a consistent estimator

Markov chain Monte Carlo

Challenge: It's typically hard to get an iid sample from f

Proposal: Create a Markov chain with stationary distribution f

Markov Chain: $f(X_{t+1}|X_0, \ldots, X_t) = f(X_{t+1}|X_t)$

How do we create such a Markov chain?

Metropolis Hastings MCMC

Problem: Generate a Markov chain with stationary distribution f

Metropolis Hastings MCMC

Algorithm:

- Propose $Y \sim q(\cdot|X_t)$, where q is a proposal distribution
- Calculate acceptance probability

$$\alpha(X_t, Y) = min\left\{1, \frac{f(Y)q(X_t|Y)}{f(X_t)q(Y|X_t)}\right\}$$

- Let $X_{t+1} = Y$ with probability α and X_t otherwise.

This algorithm generates a Markov chain $X_0, X_1, X_2, \ldots$ with stationary distribution f

Choosing proposal distributions

There are many options for choosing the proposal distribution q.

A common choice is $q(\cdot|X) = \mathcal{N}(X, \sigma^2)$.

Problem with MH-MCMC

Problem: α is typically low

- Waste a lot of time generating proposals that are not accepted
- Problem exacerbated in high dimensional settings

Hamiltonian Monte Carlo

Problem: Generate a Markov chain with stationary distribution f

Algorithm/approach

- Complicated!
- Use Hamiltonian dynamics to more efficiently sample from f

Amazing: STAN

Coming up

- Homework 7: due next *Wednesday*