STA2311: Advanced Computational Methods for Statistics I

Class 2: Classical Optimization Methods

Radu Craiu Robert Zimmerman

University of Toronto

September 19, 2023

Section 1

Introduction

Optimization Methods

- Optimization methods are used for maximizing (or minimizing) a function
- For smooth multivariate functions, this can be achieved by solving a system of non-linear equations
  - Or linear, if you're lucky!
- Many methods were developed for specific applications
- We will focus on fairly robust methods, although their efficiency can vary

Notation

- Consider a pdf/pmf $f(\boldsymbol{x} \mid \boldsymbol{\theta})$, where $\boldsymbol{x} \in \mathbb{R}^d$ and $\boldsymbol{\theta} \in \mathbb{R}^p$, which generates a sample of data $\tilde{\boldsymbol{x}}_n := \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$
- We want to maximize (in $\boldsymbol{\theta}$) the likelihood

$$L(\boldsymbol{\theta} \mid \tilde{\boldsymbol{x}}_n) = \prod_{i=1}^{n} f(\boldsymbol{x}_i \mid \boldsymbol{\theta})$$

  which is equivalent to maximizing the log-likelihood

$$\ell(\boldsymbol{\theta} \mid \tilde{\boldsymbol{x}}_n) = \sum_{i=1}^{n} \log(f(\boldsymbol{x}_i \mid \boldsymbol{\theta}))$$

  - Maximizing $\ell(\boldsymbol{\theta} \mid \tilde{\boldsymbol{x}}_n)$ is almost always easier!
- The maximizer is among the solutions of

$$\frac{\partial \ell(\boldsymbol{\theta} \mid \tilde{\boldsymbol{x}}_n)}{\partial \theta_i} = 0, \quad 1 \leq i \leq d$$

Section 2

Fixed Point Methods

Fixed Point Iteration

- A point $\boldsymbol{\theta}^*$ is a *fixed point* of a function h iff $h(\boldsymbol{\theta}^*) = \boldsymbol{\theta}^*$
- A *fixed point iteration* seeks to approximate the fixed points of h using the following steps:
    1. Initialize the process at $\boldsymbol{\theta}_0$
    2. Make the updates $\boldsymbol{\theta}_{t+1} = h(\boldsymbol{\theta}_t)$ for $t \geq 1$
    3. Stop when $\frac{||\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t||}{||\boldsymbol{\theta}_t||} < \epsilon$ where ϵ is a small user-defined threshold (say $\epsilon \approx 10^{-6}$)
- When is h guaranteed to have a fixed point?

Fixed Point Solutions

Theorem

Let $\boldsymbol{h} : \mathbb{R}^d \to \mathbb{R}^d$*. Suppose any of the following conditions hold:*

1. *h satisfies the Lipschitz condition* $||\boldsymbol{h}(\boldsymbol{\theta}) - \boldsymbol{h}(\boldsymbol{\theta}')|| \leq C \cdot ||\boldsymbol{\theta} - \boldsymbol{\theta}'||$ *for some constant* $C \in (0, 1)$ *and for all* $\boldsymbol{\theta}, \boldsymbol{\theta}' \in \mathbb{R}^d$
2. $\boldsymbol{h} : K \to K$ *is continuous and* $K \in \mathbb{R}^d$ *is compact*
3. $d = 1$*, h is differentiable, and* $||h'(\theta)|| < 1$ *for all* $\theta \in \mathbb{R}$

Then a solution exists to $\boldsymbol{h}(\boldsymbol{\theta}) = \boldsymbol{\theta}$*.*

Example: Existence of a Fixed Point Solution

```r
norm <- function(v) {sqrt(sum(v^2))}

h <- function(th) {c(sin(th[1]), cos(th[2]))}

th <- c(0.5, 0.5)

err <- Inf

while (err > 10e-6) {
  th_new <- h(th)
  err <- norm(th_new - th)/norm(th)
  print(th_new)
  th <- th_new
}

th
h(th)
```

Section 3

Newton-Raphson Methods

Univariate Newton-Raphson

- Let $g : \mathbb{R} \to \mathbb{R}$ be twice continuously differentiable such that $g'(\theta) \neq 0$ whenever $g(\theta) = 0$
- The Newton-Raphson (NR) algorithm approximates a root of g using the following steps:
  1. Initialize the process at θ_0
  2. Make the updates $\theta_{t+1} = \theta_t - \frac{g(\theta_t)}{g'(\theta_t)}$ for $t \geq 1$
  3. Stop when $\frac{||\theta_{t+1} - \theta_t||}{||\theta_t||} < \epsilon$ where ϵ is a small user-defined threshold

Newton-Raphson: A Quick Derivation

- Why should this work?
- Suppose that θ^* is a root of g
- By Taylor's theorem,

$$0 = g(\theta^*) = g(\theta_t) + (\theta^* - \theta_t)g'(\theta_t) + \frac{(\theta^* - \theta_t)^2}{2!}g''(\tilde{\theta}_t) \qquad (1)$$

  for some $\tilde{\theta}_t$ between θ_t and θ^*

- If θ_t is close to θ^*, then $(\theta^* - \theta_t)^2$ is small and the last term in (1) is (hopefully) negligible
- So

$$\theta^* \approx \theta_t - \frac{g(\theta_t)}{g'(\theta_t)}$$

Newton-Raphson: Convergence Order

- Let $\epsilon_t := \theta_t - \theta^*$ be the error of the t'th approximation
- An optimization method for finding θ^* has *convergence order* $\beta > 0$ if $\lim_{t\to\infty} \epsilon_t = 0$ and
$$\lim_{t\to\infty} \frac{|\epsilon_{t+1}|}{|\epsilon_t|^\beta} = c$$
for some $c > 0$
- What is the convergence order of NR (if it exists at all)?
- From (1), we get that

$$\overbrace{(\theta^* - \theta_t)^2}^{\epsilon_t^2} \frac{g(\tilde{\theta}_t)}{2g'(\theta_t)} = \overbrace{\theta_t - \frac{g(\theta_t)}{g'(\theta_t)}}^{\theta_{t+1}} - \theta^*$$
$$\implies \left| \frac{g''(\tilde{\theta}_t)}{2g'(\theta_t)} \right| = \frac{|\epsilon_{t+1}|}{|\epsilon_t|^2} \qquad (2)$$

Newton-Raphson: Convergence Order (Continued)

- Using (2), one can rigorously show that NR has a convergence order of 2 in the proximity of θ^*
- That is, the convergence order is quadratic
- Moreover, if g is steep in an interval around θ^*, then g' will be large in that interval and the convergence will be even faster
- But the algorithm is not guaranteed to find its way into that interval
    - More on that later

When Derivatives Are Unavailable. . .

- When the derivative of g cannot be computed, we may approximate $g'(\theta_t)$ by a finite difference:

$$g'(\theta_t) \approx \frac{g(\theta_t) - g(\theta_{t-1})}{\theta_t - \theta_{t-1}}$$

- Then the modified NR process becomes
  1. Initialize the process at θ_0, θ_1
  2. Make the updates $\theta_{t+1} = \theta_t - \frac{g(\theta_t) - g(\theta_{t-1})}{\theta_t - \theta_{t-1}}$ for $t \geq 1$
  3. Stop when $\frac{||\theta_{t+1} - \theta_t||}{||\theta_t||} < \epsilon$ where ϵ is a small user-defined threshold

Multivariate Newton-Raphson

- Suppose $\boldsymbol{g} = (g_1, \ldots, g_d)^\top : \mathbb{R}^d \to \mathbb{R}^d$ and assume we want to solve $g_i(\boldsymbol{\theta}) = 0$ for $1 \leq i \leq d$
- Define the Jacobian matrix $\boldsymbol{J}_{\boldsymbol{g}}(\boldsymbol{\theta}) \in \mathbb{R}^{d \times d}$ with $[\boldsymbol{J}_{\boldsymbol{g}}(\boldsymbol{\theta})]_{i,j} = \frac{\partial g_i(\theta)}{\partial \theta_j}$ and assume that $\boldsymbol{J}_{\boldsymbol{g}}(\boldsymbol{\theta})$ is invertible when $\boldsymbol{g}(\boldsymbol{\theta}) = 0$
- The multivariate NR algorithm approximates a root of g using the following steps:
  1. Initialize the process at $\boldsymbol{\theta}_0$
  2. Make the updates $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - [\boldsymbol{J}_{\boldsymbol{g}}(\boldsymbol{\theta}_t)]^{-1} \boldsymbol{g}(\boldsymbol{\theta}_t)$ for $t \geq 1$
  3. Stop when $\frac{||\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t||}{||\boldsymbol{\theta}_t||} < \epsilon$ where ϵ is a small user-defined threshold

Newton-Raphson: General Remarks

- NR may fail if the initial value $\boldsymbol{\theta}_0$ is far from the solution $\boldsymbol{\theta}^*$
- There may be more than one solution to $\boldsymbol{g}(\boldsymbol{\theta}) = \mathbf{0}$
- It is generally a good idea to run multiple NR algorithms, each initialized at different values widely spread out across $\mathrm{Dom}(\boldsymbol{g})$

Connections to Statistical Inference

- When $g : \mathbb{R}^d \to \mathbb{R}$ is the derivative of the log-likelihood $g(\theta) = \frac{\partial \ell(\boldsymbol{\theta} | \tilde{\mathbf{x}}_n)}{\partial \theta}$, the Jacobian $\boldsymbol{J}_g(\boldsymbol{\theta}^*)$ is the observed Fisher information:

$$[\mathcal{J}_n(\boldsymbol{\theta}^*)]_{i,j} := \left(\frac{\partial^2 \ell(\boldsymbol{\theta} \mid \tilde{\mathbf{x}}_n)}{\partial \theta_i \partial \theta_j} \right) \Bigg|_{\boldsymbol{\theta} = \boldsymbol{\theta}^*}$$

- The *Fisher scoring* algorithm is obtained when we replace the observed Fisher information with the (expected) Fisher information

$$[\mathcal{I}_n(\boldsymbol{\theta}^*)]_{i,j} = \mathbb{E}_\theta \left[\left(\frac{\partial^2 \ell(\boldsymbol{\theta} \mid \tilde{\mathbf{X}}_n)}{\partial \theta_i \partial \theta_j} \right) \right]$$

  in the NR algorithm

Newton-Raphson: Example 1

- Consider an iid sample $\{1, 1, 1, 1, 1, 1, 2, 2, 2, 3\}$ from

$$f(y \mid \theta) = \frac{\theta^y}{-y \cdot \log(1-\theta)}, \quad y \in \mathbb{N}^*, \quad \theta \in (0,1)$$

- Compute the MLE of θ using both NR and Fisher scoring
- How do the methods compare to one another?

Newton-Raphson: Example 2

- Consider the four blood types A, B, O, and AB
- We know that...
  - Type A blood is given by alleles *aa, ao*, and *oa*
  - Type B blood is given by alleles *bb, bo*, and *ob*
  - Type AB blood is given by alleles *ab* and *ba*
  - Type O blood is given by allele *oo*
- Given counts of people with these four blood types, $n_A, \dots, n_{AB}$ obtained from a sample of size n, we would like to estimate the frequency of the three alleles *a, b*, and *o* in the population

Newton-Raphson: Example 3

- Consider the mining town data in the table below concerning the number of children per family in a sample of 4075 families living in a mining town

No. children	0	1	2	3	4	5	6
No. families	3,062	587	284	103	33	4	2

- Assume the samples are collected from a mixture of two subpopulations:
- One subpopulation consists of families without children, and its proportion of the total population is $\xi \in (0, 1)$
- The other subpopulation consists of families with any number of children, and is well-modelled by a Poisson(λ) distribution

Newton-Raphson: Example 3 (Continued)

- Given this model structure, the likelihood function of $\boldsymbol{\theta} = (\lambda, \xi)$ is

$$L(\boldsymbol{\theta} \mid n_0, \ldots, n_6) = \left[\xi + (1-\xi)e^{-\lambda}\right]^{n_0} \cdot \prod_{j=1}^{6} \left[(1-\xi) \cdot \frac{e^{-\lambda}\lambda^j}{j!}\right]^{n_j}$$

- Compute the MLE of $\boldsymbol{\theta}$ using both NR and Fisher scoring

Newton-Raphson: Example 3 (Implementation)

```
n0 <- 3062
n <- c(587, 284, 103, 33, 4, 2)

expit <- function(x) {1/(1+exp(-x))}

ll <- function(lambda, eta) {
  xi <- expit(eta)
  n0*log(xi + (1-xi)*exp(-lambda)) + log(1-xi) -1013*lambda + 1628*log(lamb
}

dll.dlambda <- function(lambda, eta) {
  (1628 + exp(eta+lambda)*(1628-1013*lambda) - 4075*lambda)/( lambda*(1+exp
}

dll.deta <- function(lambda, eta) {
  -exp(eta)*(3063-3062*exp(lambda) + exp(eta+lambda))/( (1+exp(eta))*(1+exp
}
```

Newton-Raphson: Example 3 (Implementation Cont'd)

```
d2ll.dlambda2 <- function(lambda, eta) {
  -2*(814 + 814*exp(2*(eta+lambda)) + exp(eta+lambda)*(1628-1531*lambda^2)
}

d2ll.deta2 <- function(lambda, eta) {
  exp(eta)*(-3063 + 3062*exp(lambda) - 2*exp(eta+lambda) - 3063*exp(2*(eta+
}

d2ll.detadlambda <- function(lambda, eta) {
  3062*exp(eta+lambda)/(1+exp(eta+lambda))^2
}
```

Newton-Raphson: Example 3 (Implementation Cont'd)

```
norm <- function(x) {sqrt(sum(x^2))}

eps <- 10e-5
delta <- 1
lambda.old <- 0.5
xi.old <- 0.9
eta.old <- log(xi.old/(1-xi.old))

while (delta > eps) {
  print(c(lambda.old, xi.old, ll(lambda.old, eta.old)))
  dl.old <- c(dll.dlambda(lambda.old, eta.old), dll.deta(lambda.old, eta.ol
  Jl.old <- matrix(c(d2ll.dlambda2(lambda.old, eta.old), d2ll.detadlambda(l
                       d2ll.detadlambda(lambda.old, eta.old), d2ll.deta2(lamb
                     nrow=2, byrow=T)
  w <- c(lambda.old, eta.old) - solve(Jl.old)%*%dl.old
  lambda.new <- w[1]
  eta.new <- w[2]
  delta <- norm(w - c(lambda.old, eta.old))/norm(c(lambda.old, eta.old))
  lambda.old <- lambda.new
  eta.old <- eta.new
  xi.old <- expit(eta.old)}
```

Gauss-Newton

- Let $g(\boldsymbol{\theta}) = \sum_{i=1}^{n}(y_i - f_i(\boldsymbol{\theta}))^2$, where each $f_i : \mathbb{R}^d \to \mathbb{R}$ is differentiable and $\boldsymbol{\theta} \in \mathbb{R}^d$
- Suppose we want to find

$$\boldsymbol{\theta}^* = \operatorname*{argmin}_{\boldsymbol{\theta}} g(\boldsymbol{\theta}) \tag{3}$$

- If $f_i(\boldsymbol{\theta}) = \boldsymbol{X}_i^\top \boldsymbol{\theta}$ (i.e., each f_i is *linear*), then (3) is uniquely solved by the well-known *least-squares estimate* $\boldsymbol{\theta}^* = (\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{y}$
- Here $\boldsymbol{X} = [\boldsymbol{X}_1 \cdots \boldsymbol{X}_n]^\top$ and $\boldsymbol{y} = (y_1, \ldots, y_n)^\top$
- The *Gauss-Newton algorithm* is an iterative procedure that solves (3) using local linear approximations of f_i

Gauss-Newton: A Quick Derivation

- Suppose $\boldsymbol{\theta}^*$ is the (unknown) solution to (3)
- Use a Taylor series expansion of each f_i at $\boldsymbol{\theta}$ close to $\boldsymbol{\theta}^*$, say $\boldsymbol{\theta} = \boldsymbol{\theta}^* + \boldsymbol{u}^*$ with a small $\boldsymbol{u}^*$.
- Taylor expansion:

$$f_i(\boldsymbol{\theta}^*) = f_i(\boldsymbol{\theta} - \boldsymbol{u}^*) \approx f_i(\boldsymbol{\theta}) + \nabla f_i(\boldsymbol{\theta})^\top \boldsymbol{u}^* \tag{4}$$

- For a fixed θ, set $h_{\boldsymbol{\theta}}(\boldsymbol{u}) = \sum_{i=1}^n (y_i - f_i(\boldsymbol{\theta} - \boldsymbol{u}))^2$, so that $\operatorname*{argmin}_{\boldsymbol{u}} h_{\boldsymbol{\theta}}(\boldsymbol{u}) = \boldsymbol{\theta} - \boldsymbol{\theta}^* = \boldsymbol{u}^*$
- Plugging $\boldsymbol{u}^*$ into (4) we see that it is also the minimizer of $\sum_{i=1}^n (y_i - f_i(\boldsymbol{\theta}) - \nabla f_i(\boldsymbol{\theta})^\top \boldsymbol{u})^2$
- And the latter is just the least-squares sum $\sum_{i=1}^n (\tilde{y}_i - \tilde{\boldsymbol{X}}_i^\top \boldsymbol{u})^2$ with $\tilde{y}_i = y_i - f_i(\boldsymbol{\theta})$ and $\tilde{\boldsymbol{X}}_i = \nabla f_i(\boldsymbol{\theta})$

Gauss-Newton

- The previous derivation suggests the following iterative procedure:

  1. Initialize the process at $\boldsymbol{\theta}_0$
  2. Set
  $$\boldsymbol{A}_t^\top = (\nabla f_1(\boldsymbol{\theta}_t), \ldots, \nabla f_n(\boldsymbol{\theta}_t)) \in \mathbb{R}^{d \times n}$$
  where $\nabla f_i(\boldsymbol{\theta}) = \left(\frac{\partial f_i}{\partial \theta_1}, \ldots, \frac{\partial f_i}{\partial \theta_d}\right)^\top$, and
  $$\boldsymbol{Z}_t = (y_1 - f_1(\boldsymbol{\theta}_t), \ldots, y_n - f_n(\boldsymbol{\theta}_t))^\top \in \mathbb{R}^n$$
  and
  $$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + (\boldsymbol{A}_t^\top \boldsymbol{A}_t)^{-1} \boldsymbol{A}_t^\top \boldsymbol{Z}_t, \quad t \geq 0$$
  3. Stop when $\frac{||\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t||}{||\boldsymbol{\theta}_t||} < \epsilon$ where ϵ is a small user-defined threshold

Section 4

Gradient Descent

The Setup

- Our goal here is to minimize a differentiable function $g : \mathbb{R}^d \to \mathbb{R}$
- By "differentiable", we mean that the gradient ∇g exists
    - But we will relax this assumption later
- Recall from multivariate calculus that the gradient $\nabla g(\boldsymbol{\theta})$ is the vector at which g increases the fastest at the point $\boldsymbol{\theta}$
    - So $-\nabla g(\boldsymbol{\theta})$ gives the direction in which g has the "steepest descent" at $\boldsymbol{\theta}$
- Equivalently, $\nabla g(\boldsymbol{\theta}) \cdot \boldsymbol{u}$ gives the directional derivative of g along the vector $\boldsymbol{u} \in \mathbb{R}^d$ at $\boldsymbol{\theta}$
- That is,

$$\nabla g(\boldsymbol{\theta}) \cdot \boldsymbol{u} = \lim_{h \to 0} \frac{g(\boldsymbol{\theta} + h\boldsymbol{u}) - g(\boldsymbol{\theta})}{h}$$

The Motivation

- Idea: if we want to find our way down the surface of g, take a step in the steepest downward direction from where we currently stand
- So if we stand at $\boldsymbol{\theta}_t$, we then take a step to

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \nabla g(\boldsymbol{\theta}_t)$$

- We may want to take smaller or larger steps in the same direction, so choose

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - h\nabla g(\boldsymbol{\theta}_t)$$

  for some pre-chosen $h > 0$
- With small enough step sizes, we will always have $g(\boldsymbol{\theta}_{t+1}) \leq g(\boldsymbol{\theta}_t)$
  - But too small, and we'll move very slowly...

The Algorithm

- The *gradient descent* algorithm is
  1. Initialize the process at $\boldsymbol{\theta}_0$ and choose a pre-specified step size $h > 0$
  2. Make the updates $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - h\nabla g(\boldsymbol{\theta}_t)$ for $t \geq 0$
  3. Stop when $||\nabla g(\boldsymbol{\theta}_{t+1})|| < \epsilon$ where ϵ is a small user-defined threshold

The Step Size Matters!

- For a toy example, consider minimizing $g(\theta) = \theta^4$ with $\theta_0 = 1$ and step sizes $h \in \{0.0025, 0.025, 0.25\}$

```
g <- function(x) {x^4}
grad.g <- function(x) {4*x^3}

h.1 <- 0.0025
t.1 <- 0

th <- 1

while (abs(grad.g(th)) > 1e-6) {
  t.1 <- t.1 + 1
  th <- th - h.1*grad.g(th)
}

cat("theta* = ", th, "; iterations: ", t.1, sep="")
```

```
## theta* = 0.006299604; iterations: 1259864
```

The Step Size Matters! (Continued)

- Now $h = 0.025$:

```r
h.2 <- 0.025
t.2 <- 0

th <- 1

while (abs(grad.g(th)) > 1e-6) {
  t.2 <- t.2 + 1
  th <- th - h.2*grad.g(th)
}

cat("theta* = ", th, "; iterations: ", t.2, sep="")
```

```
## theta* = 0.006299591; iterations: 125980
```

The Step Size Matters! (Continued)

- And finally $h = 0.25$:

```r
h.3 <- 0.25
t.3 <- 0

th <- 1

while (abs(grad.g(th)) > 1e-6) {
  t.3 <- t.3 + 1
  th <- th - h.3*grad.g(th)
}

cat("theta* = ", th, "; iterations: ", t.3, sep="")
```

```
## theta* = 0; iterations: 1
```

- We seem to be improving as h gets larger
- What happens if we try $h = 0.5$?

Guarantees for GD

Theorem

Suppose g is convex and ∇g is L-Lipschitz. Let $\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, g(\boldsymbol{\theta})$ and let $\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \ldots$ be the sequence of GD outputs. If $h \leq 1/L$, then

$$g(\boldsymbol{\theta}_t) - g(\boldsymbol{\theta}^*) \leq \frac{||\boldsymbol{\theta}^* - \boldsymbol{\theta}_0||^2}{2th}.$$

- If g is convex then any local minimum is a global minimum
- Unfortunately, we rarely have the luxury of dealing with convex functions
  - At least Lipschitz gradients are fairly common in statistics and machine learning
- If g is nonconvex, then GD can easily find its way into a local mode (and get stuck there)

Adaptive Step Sizes

- The choice of h is generally not an easy one to make, especially when the conditions in the theorem above cannot be verified
- At the cost of extra computation, we can make the step size adaptive
- That is, we make the updates $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - h_t \nabla g(\boldsymbol{\theta}_t)$, where we choose the h_t's in a principled way
- In principle, the "best" choice is $h_t = \operatorname*{argmin}_h g(\boldsymbol{\theta}_t - h \nabla g(\boldsymbol{\theta}_t))$
  - This is called *exact line search*, but is usually difficult/impractical

Backtracking Line Search

- A basic variation is *backtracking line search*, where we use the idea above to find a "good enough" step size without doing too much work:
  1. Initialize the process at $\boldsymbol{\theta}_0$ and choose a "large" $\alpha > 0$ and $\gamma \in (0, 1)$
  2. For each $t \geq 0$, choose a decreasing sequence $\alpha \geq \alpha_{t,1} > \alpha_{t,2} > \dots$ until
  $$g(\boldsymbol{\theta}_t - \alpha_{t,k}\nabla g(\boldsymbol{\theta}_t)) \leq g(\boldsymbol{\theta}_t) - \alpha_{t,k}\gamma||\nabla g(\boldsymbol{\theta}_t)||^2$$
  3. Stop when $||\nabla g(\boldsymbol{\theta}_{t+1})|| < \epsilon$ where ϵ is a small user-defined threshold
- One common choice of sequence: choose $\alpha \in (0, 1)$ and $\alpha_{t,k} = \alpha^k$
- But there are more clever methods out there

Example: Logistic Regression

```
set.seed(2311)
expit <- function(x) {1/(1+exp(-x))}
logit <- function(p) {log(p/(1-p))}
norm <- function(x) {sqrt(sum(x^2))}

n <- 1000

X1 <- rnorm(n=n)
X2 <- rbinom(n=n, size=1, prob=0.2)
X3 <- rpois(n=n, lambda=0.7)
X <- cbind(1, X1, X2, X3)

y <- rbinom(n=n, size=1, prob=expit(0.4 + 0.7*X1 + 3*X2 - X3))

g <- function(theta) {
  -sum(y*log(expit(apply(X, 1, function(x) x%*%theta)) +
      (1-y)*log(1-expit(apply(X, 1, function(x) x%*%theta))))}

grad.g <- function(theta) {
  t(X) %*% (expit(apply(X, 1, function(x) x%*%theta)) - y)}
```

Example: Logistic Regression (Continued)

```r
eps <- 1e-5
gam <- 0.001

th <- rep(1, 4)

while (norm(grad.g(th)) > eps) {
  alp <- 0.05

  while(g(th - alp*grad.g(th)) > g(th) - alp*gam*norm(grad.g(th))^2) {
    alp <- alp*0.05
  }

  th <- th - alp*grad.g(th)
}

th.GD <- as.vector(th)
th.NR <- as.vector(glm(y ~ ., family = binomial(link="logit"),
                       data=data.frame(y, X1, X2, X3))$coefficients)
```

Including Constraints: Projected Gradient Descent

- In statistical contexts, we often need to perform *constrained* optimization
- The vanilla GD algorithm puts no constraints on the iterates $\boldsymbol{\theta}_t$, but this is easily modified
- The *projected gradient descent* algorithm adds an intermediate step in which each $\boldsymbol{\theta}^{(t)}$ is projected onto some *closed* constraint set C
- More specifically, we replace Step 2 in the basic GD algorithm by
  2. For $t \geq 0$, make the updates $\boldsymbol{\theta}_{t+1/2} = \boldsymbol{\theta}_t - h\nabla g(\boldsymbol{\theta}_t)$ and $\boldsymbol{\theta}_t \in \underset{\boldsymbol{\eta} \in C}{\operatorname{argmin}} ||\boldsymbol{\eta} - \boldsymbol{\theta}_{t+1/2}||$
- When C is convex, the argmin is unique (so "$\in$" can be replaced by "$=$")
- Modifications such as adaptive step sizes can still be applied

Projected Gradient Descent: Example

- Consider again finding the MLE of θ given an iid sample $\{1, 1, 1, 1, 1, 1, 2, 2, 2, 3\}$ from

$$f(y \mid \theta) = \frac{\theta^y}{-y \cdot \log(1-\theta)}, \quad y \in \mathbb{N}^*, \quad \theta \in (0,1)$$

- The constraint set here is $(0, 1)$, which is open
- However, if we're confident that θ_{MLE} isn't too close 0 or 1, we can take $C = [\epsilon, 1-\epsilon]$ for some very small $\epsilon > 0$
- It is not hard to show that

$$\operatorname*{argmin}_{\eta \in [\epsilon, 1-\epsilon]} ||\eta - \theta_{t+1/2}|| = \min\{\max\{\theta_{t+1/2}, \epsilon\}, 1-\epsilon\}$$

Projected Gradient Descent: Example (Continued)

```r
y <- c(1,1,1,1,1,1,2,2,2,3)

ll <- function(theta) {sum(y*log(theta) - log(-y*log(1-theta)))}
grad.ll <- function(theta) {sum(y/theta + 1/((1-theta)*log(1-theta)))}

th <- 0.9

h <- 0.005

while (abs(grad.ll(th)) > 1e-6) {
  thp5 <- th + h*grad.ll(th)
  th <- min(max(thp5, 0 + .Machine$double.eps), 1 - .Machine$double.eps)
}
```

Subgradient Methods

- Sometimes our objective function g may not be differentiable
- If g is convex, then it still has at least one *subgradient* $\boldsymbol{v}_0$ at any point $\boldsymbol{\theta}_0$ satisfying
$$g(\boldsymbol{\theta}) - g(\boldsymbol{\theta}_0) \geq \boldsymbol{v}_0 \cdot (\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$
- The basic *subgradient method* is
  1. Initialize the process at $\boldsymbol{\theta}_0$ and choose a pre-specified step size $h > 0$
  2. Make the updates $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - h\boldsymbol{v}_t$ for $t \geq 0$, where $\boldsymbol{v}_t$ is any subgradient of h at $\boldsymbol{\theta}_t$
  3. Keep track of the best iterate so far by setting $h_t^{\text{best}} = \min\{h_{t-1}^{\text{best}}, h(\boldsymbol{\theta}_t)\}$
  4. Stop when...?!
- Although subgradient methods also have convergence guarantees, they have no universally agreed upon stopping criteria

Extensions

- There are *many* (hundreds?) of extensions and variations of the basic GD algorithm and the subgradient method [Ruder, 2016]
- We will examine SGD (*stochastic gradient descent*) in Class 4

References I

Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.