Machine Learning for Finance

Neal Parikh

Cornell University

Spring 2018

Optimization algorithms

Optimization algorithms

- many algorithms available for different classes of problems
- distinguish between problem formulation and optimization algorithm
- reformulating the problem may make different algorithms applicable
- specialized vs general-purpose algorithms
- we will only do a high-level survey for flavor, omitting many details

Outline

Matrix structure and algorithm complexity

cost (execution time) of solving $Ax = b$ with $A \in \mathbf{R}^{n \times n}$

- for general methods, grows as n^3
- less if A is structured (banded, sparse, Toeplitz, . . .)

flop counts

- flop (floating-point operation): one addition, subtraction, multiplication, or division of two floating-point numbers
- to estimate complexity of an algorithm: express number of flops as a (polynomial) function of the problem dimensions, and simplify by keeping only the leading terms
- not an accurate predictor of computation time on modern computers
- useful as a rough estimate of complexity

Matrix structure and algorithm complexity

vector-vector operations (x, $y \in \mathbf{R}^n$)

- inner product $x^T y$: $2n - 1$ flops (or $2n$ if n is large)
- sum $x + y$, scalar multiplication αx: n flops

matrix-vector product $y = Ax$ with $A \in \mathbf{R}^{m \times n}$

- $m(2n - 1)$ flops (or $2mn$ if n large)
- $2N$ if A is sparse with N nonzero elements
- $2p(n + m)$ if A is given as $A = UV^T$, $U \in \mathbf{R}^{m \times p}$, $V \in \mathbf{R}^{n \times p}$

matrix-matrix product $C = AB$ with $A \in \mathbf{R}^{m \times n}$, $B \in \mathbf{R}^{n \times p}$

- $mp(2n - 1)$ flops (or $2mnp$ if n large)
- less if A and/or B are sparse
- $(1/2)m(m + 1)(2n - 1) \approx m^2 n$ if $m = p$ and C symmetric

Linear equations that are easy to solve

diagonal matrices ($a_{ij} = 0$ if $i \neq j$): n flops

$$x = A^{-1}b = (b_1/a_{11}, \ldots, b_n/a_{nn})$$

lower triangular ($a_{ij} = 0$ if $j > i$): n^2 flops

$$\begin{array}{lcl} x_1 & := & b_1/a_{11} \\ x_2 & := & (b_2 - a_{21}x_1)/a_{22} \\ x_3 & := & (b_3 - a_{31}x_1 - a_{32}x_2)/a_{33} \\ & \vdots & \\ x_n & := & (b_n - a_{n1}x_1 - a_{n2}x_2 - \cdots - a_{n,n-1}x_{n-1})/a_{nn} \end{array}$$

called forward substitution

upper triangular ($a_{ij} = 0$ if $j < i$): n^2 flops via backward substitution

Linear equations that are easy to solve

orthogonal matrices: $A^{-1} = A^T$

- $2n^2$ flops to compute $x = A^T b$ for general A
- less with structure, *e.g.*, if $A = I - 2uu^T$ with $\|u\|_2 = 1$, we can compute $x = A^T b = b - 2(u^T b)u$ in $4n$ flops

permutation matrices:

$$a_{ij} = \begin{cases} 1 & j = \pi_i \\ 0 & \text{otherwise} \end{cases}$$

where $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ is a permutation of $(1, 2, \ldots, n)$

- interpretation: $Ax = (x_{\pi_1}, \ldots, x_{\pi_n})$
- satisfies $A^{-1} = A^T$, hence cost of solving $Ax = b$ is 0 flops

example:

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}, \qquad A^{-1} = A^T = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

The factor-solve method for solving $Ax = b$

- factor A as a product of simple matrices (usually 2 or 3):

$$A = A_1 A_2 \cdots A_k$$

  (A_i diagonal, upper or lower triangular, etc)

- compute $x = A^{-1}b = A_k^{-1} \cdots A_2^{-1} A_1^{-1} b$ by solving k 'easy' equations

$$A_1 x_1 = b, \qquad A_2 x_2 = x_1, \qquad \ldots, \qquad A_k x = x_{k-1}$$

cost of factorization step usually dominates cost of solve step

equations with multiple righthand sides

$$Ax_1 = b_1, \qquad Ax_2 = b_2, \qquad \ldots, \qquad Ax_m = b_m$$

cost: one factorization plus m solves

LU factorization

every nonsingular matrix A can be factored as

$$A = PLU$$

with P a permutation matrix, L lower triangular, U upper triangular

cost: $(2/3)n^3$ flops

Solving linear equations by LU factorization.

given a set of linear equations $Ax = b$, with A nonsingular.

1. *LU factorization.* Factor A as $A = PLU$ ($(2/3)n^3$ flops).
2. *Permutation.* Solve $Pz_1 = b$ (0 flops).
3. *Forward substitution.* Solve $Lz_2 = z_1$ (n^2 flops).
4. *Backward substitution.* Solve $Ux = z_2$ (n^2 flops).

---

cost: $(2/3)n^3 + 2n^2 \approx (2/3)n^3$ for large n

Cholesky factorization

every positive definite A can be factored as

$$A = LL^T$$

with L lower triangular

cost: $(1/3)n^3$ flops

Solving linear equations by Cholesky factorization.

given a set of linear equations $Ax = b$, with $A \in \mathbf{S}^n_{++}$.

1. *Cholesky factorization.* Factor A as $A = LL^T$ ($(1/3)n^3$ flops).
2. *Forward substitution.* Solve $Lz_1 = b$ (n^2 flops).
3. *Backward substitution.* Solve $L^T x = z_1$ (n^2 flops).

---

cost: $(1/3)n^3 + 2n^2 \approx (1/3)n^3$ for large n

LDL^T factorization

every nonsingular symmetric matrix A can be factored as

$$A = PLDL^T P^T$$

with P a permutation matrix, L lower triangular, D block diagonal with 1×1 or 2×2 diagonal blocks

cost: $(1/3)n^3$

- cost of solving symmetric sets of linear equations by LDL^T factorization: $(1/3)n^3 + 2n^2 \approx (1/3)n^3$ for large n
- for sparse A, can choose P to yield sparse L; cost $\ll (1/3)n^3$

Equations with structured sub-blocks

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{1}$$

- variables $x_1 \in \mathbf{R}^{n_1}$, $x_2 \in \mathbf{R}^{n_2}$; blocks $A_{ij} \in \mathbf{R}^{n_i \times n_j}$
- if A_{11} is nonsingular, can eliminate x_1: $x_1 = A_{11}^{-1}(b_1 - A_{12}x_2)$; to compute x_2, solve

$$(A_{22} - A_{21}A_{11}^{-1}A_{12})x_2 = b_2 - A_{21}A_{11}^{-1}b_1$$

Solving linear equations by block elimination.

given a nonsingular set of linear equations (1), with A_{11} nonsingular.

1. Form $A_{11}^{-1}A_{12}$ and $A_{11}^{-1}b_1$.
2. Form $S = A_{22} - A_{21}A_{11}^{-1}A_{12}$ and $\tilde{b} = b_2 - A_{21}A_{11}^{-1}b_1$.
3. Determine x_2 by solving $Sx_2 = \tilde{b}$.
4. Determine x_1 by solving $A_{11}x_1 = b_1 - A_{12}x_2$.

Structured matrix plus low rank term

$$(A + BC)x = b$$

- $A \in \mathbf{R}^{n\times n}$, $B \in \mathbf{R}^{n\times p}$, $C \in \mathbf{R}^{p\times n}$
- assume A has structure ($Ax = b$ easy to solve)

first write as

$$\begin{bmatrix} A & B \\ C & -I \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} b \\ 0 \end{bmatrix}$$

now apply block elimination: solve

$$(I + CA^{-1}B)y = CA^{-1}b,$$

then solve $Ax = b - By$

this proves the **matrix inversion lemma:** if A and $A + BC$ nonsingular,

$$(A + BC)^{-1} = A^{-1} - A^{-1}B(I + CA^{-1}B)^{-1}CA^{-1}$$

Structured matrix plus low rank term

example: A diagonal, B, C dense

- method 1: form $D = A + BC$, then solve $Dx = b$

  cost: $(2/3)n^3 + 2pn^2$

- method 2 (via matrix inversion lemma): solve

$$(I + CA^{-1}B)y = CA^{-1}b, \tag{2}$$

  then compute $x = A^{-1}b - A^{-1}By$

  total cost is dominated by (2): $2p^2n + (2/3)p^3$ (*i.e.*, linear in n)

Numerical linear algebra software

- most memory usage and computation time in optimization methods is spent on numerical linear algebra
- don't implement your own linear algebra
- BLAS
- ATLAS and optimized BLAS
- LAPACK
- vectorization

Outline

Unconstrained minimization

$$\text{minimize} \quad f(x)$$

- f convex, twice continuously differentiable (hence $\mathbf{dom}\, f$ open)
- we assume optimal value $p^\star = \inf_x f(x)$ is attained (and finite)

unconstrained minimization methods

- produce sequence of points $x^{(k)} \in \mathbf{dom}\, f$, $k = 0, 1, \ldots$ with

$$f(x^{(k)}) \to p^\star$$

- can interpret as iterative methods for solving optimality condition

$$\nabla f(x^\star) = 0$$

Descent methods

$$x^{(k+1)} = x^{(k)} + t^{(k)}\Delta x^{(k)} \quad \text{with } f(x^{(k+1)}) < f(x^{(k)})$$

- other notations: $x^+ = x + t\Delta x$, $x := x + t\Delta x$
- Δx is the *step*, or *search direction*; t is the *step size*, or *step length*
- (step size also called *learning rate* in machine learning)
- from convexity, $f(x^+) < f(x)$ implies $\nabla f(x)^T \Delta x < 0$
  (*i.e.*, Δx is a *descent direction*)

General descent method.

given a starting point $x \in \mathbf{dom}\, f$.
repeat

1. Determine a descent direction Δx.
2. *Line search.* Choose a step size $t > 0$.
3. *Update.* $x := x + t\Delta x$.

until stopping criterion is satisfied.

Line search types

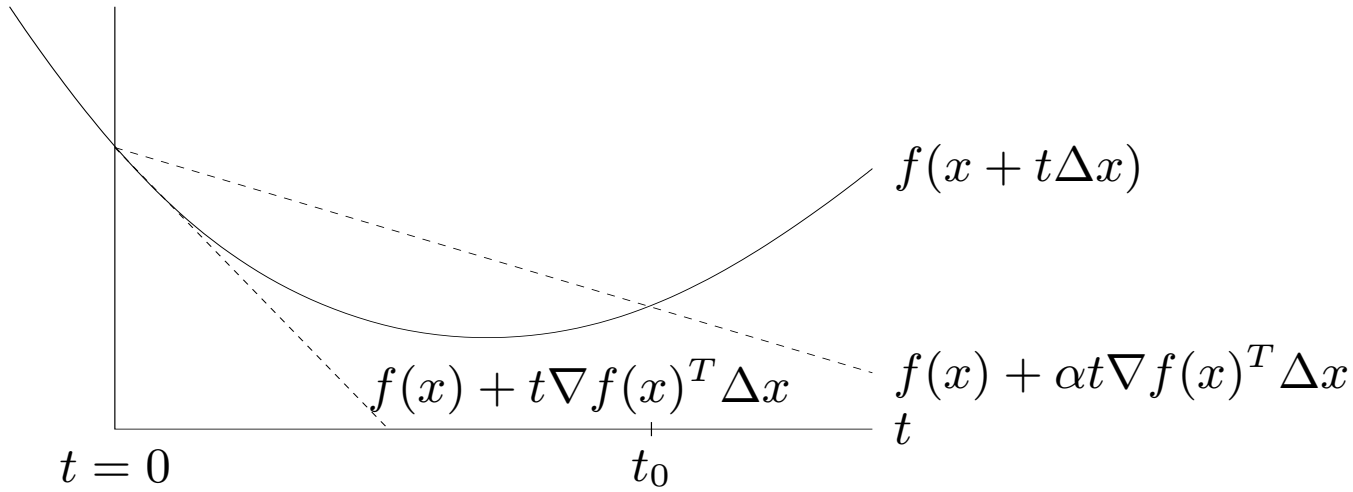
exact line search: $t = \operatorname{argmin}_{t>0} f(x + t\Delta x)$

backtracking line search (with parameters $\alpha \in (0, 1/2)$, $\beta \in (0, 1)$)

- starting at $t = 1$, repeat $t := \beta t$ until

$$f(x + t\Delta x) < f(x) + \alpha t \nabla f(x)^T \Delta x$$

- graphical interpretation: backtrack until $t \leq t_0$



Gradient descent method

general descent method with $\Delta x = -\nabla f(x)$

---

given a starting point $x \in \mathbf{dom}\, f$.
repeat

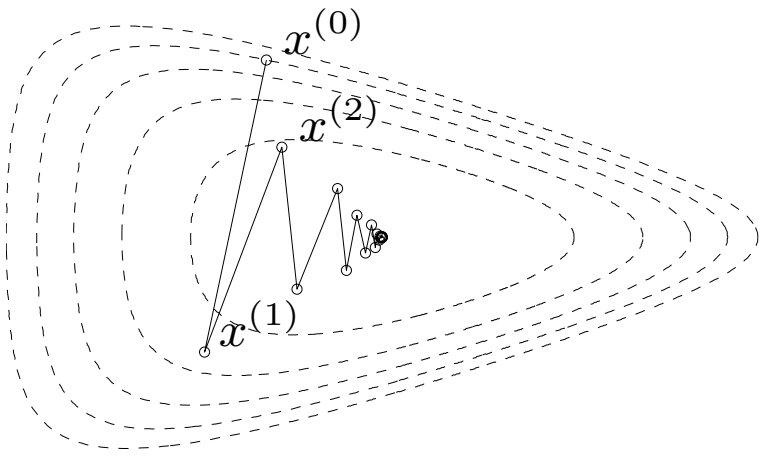
1. $\Delta x := -\nabla f(x)$.
2. *Line search.* Choose step size t via exact or backtracking line search.
3. *Update.* $x := x + t\Delta x$.

until stopping criterion is satisfied.

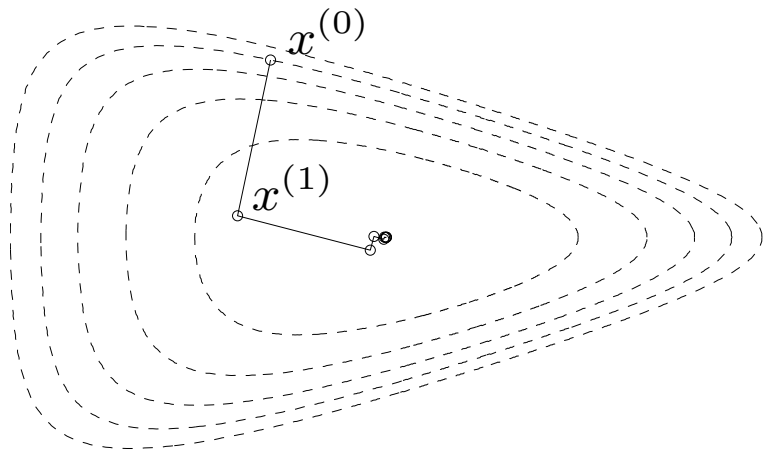
- stopping criterion usually of the form $\|\nabla f(x)\|_2 \leq \epsilon$
- very simple, but can be very slow

Gradient descent example

$$f(x_1, x_2) = e^{x_1+3x_2-0.1} + e^{x_1-3x_2-0.1} + e^{-x_1-0.1}$$



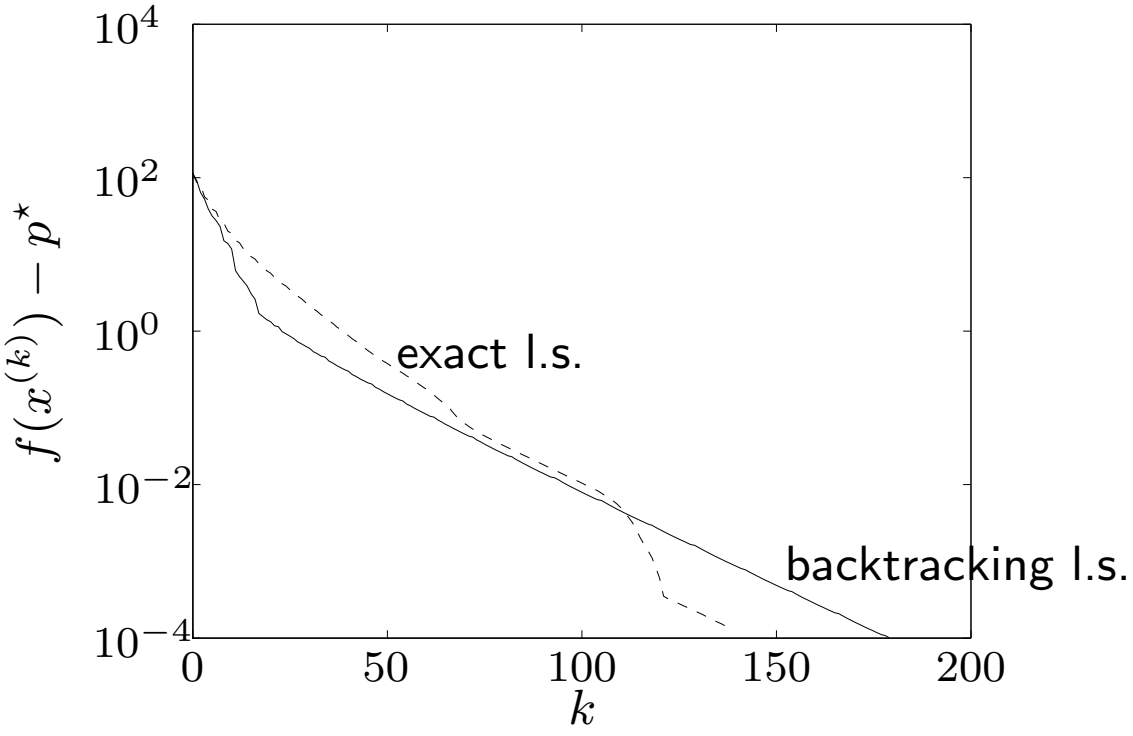
backtracking line search



exact line search

Gradient descent example in $\mathbf{R}^{100}$

$$f(x) = c^T x - \sum_{i=1}^{500} \log(b_i - a_i^T x)$$



'linear' convergence, *i.e.*, a straight line on a semilog plot

Newton step

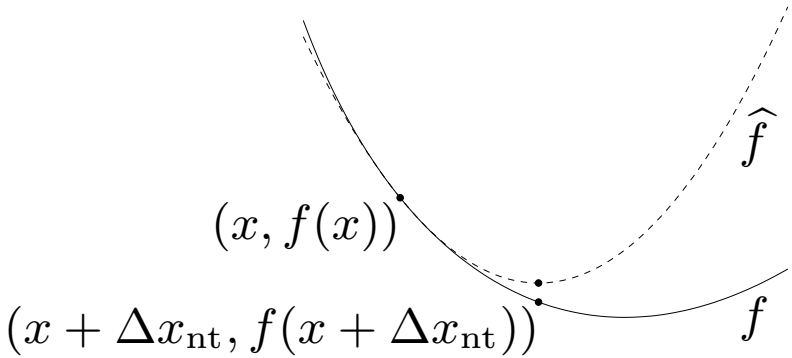
$$\Delta x_{\mathrm{nt}} = -\nabla^2 f(x)^{-1} \nabla f(x)$$

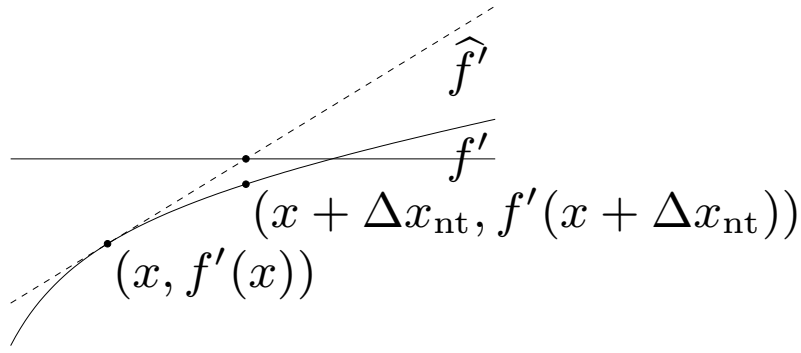
- $x + \Delta x_{\mathrm{nt}}$ minimizes second order approximation

$$\widehat{f}(x+v) = f(x) + \nabla f(x)^T v + \frac{1}{2} v^T \nabla^2 f(x) v$$

- $x + \Delta x_{\mathrm{nt}}$ solves linearized optimality condition

$$\nabla f(x+v) \approx \nabla \widehat{f}(x+v) = \nabla f(x) + \nabla^2 f(x) v = 0$$





Newton decrement

$$\lambda(x) = \left(\nabla f(x)^T \nabla^2 f(x)^{-1} \nabla f(x)\right)^{1/2}$$

a measure of the proximity of x to $x^\star$

- gives an estimate of $f(x) - p^\star$, using quadratic approximation $\widehat{f}$:

$$f(x) - \inf_y \widehat{f}(y) = \frac{1}{2}\lambda(x)^2$$

- equal to the norm of the Newton step in the quadratic Hessian norm

$$\lambda(x) = \left(\Delta x_{\rm nt}^T \nabla^2 f(x) \Delta x_{\rm nt}\right)^{1/2}$$

- directional derivative in Newton direction: $\nabla f(x)^T \Delta x_{\rm nt} = -\lambda(x)^2$
- affine invariant (unlike $\|\nabla f(x)\|_2$)

Newton's method

---

given a starting point $x \in \mathbf{dom}\, f$, tolerance $\epsilon > 0$.

repeat

1. *Compute the Newton step and decrement.*
   $$\Delta x_{\rm nt} := -\nabla^2 f(x)^{-1} \nabla f(x); \quad \lambda^2 := \nabla f(x)^T \nabla^2 f(x)^{-1} \nabla f(x).$$
2. *Stopping criterion.* **quit** if $\lambda^2/2 \leq \epsilon$.
3. *Line search.* Choose step size t by backtracking line search.
4. *Update.* $x := x + t\Delta x_{\rm nt}$.

---

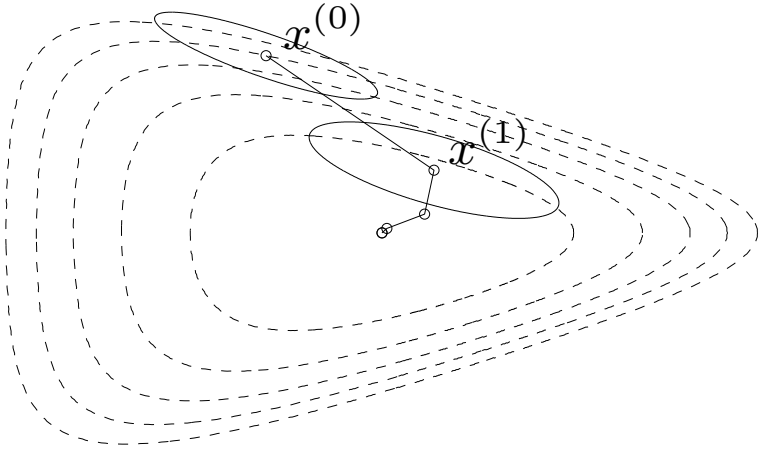
affine invariant, *i.e.*, independent of linear changes of coordinates:

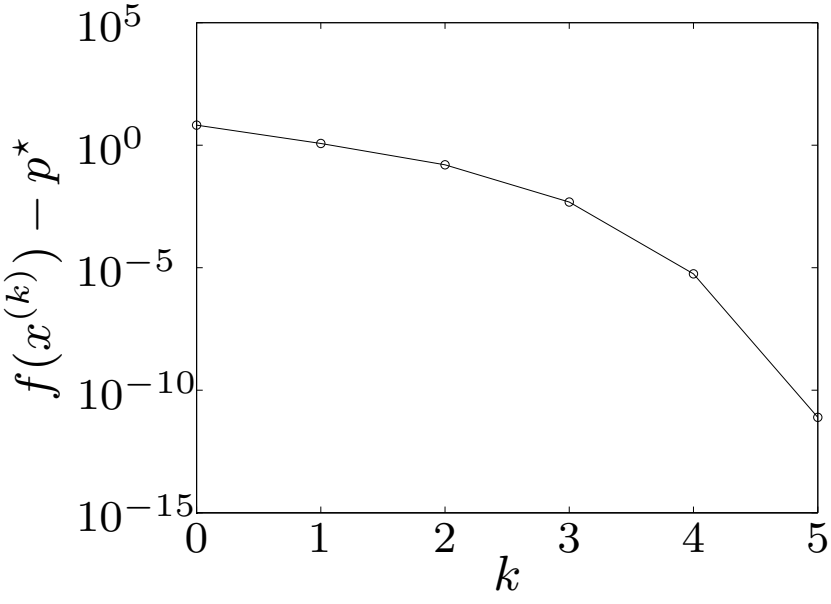
Newton iterates for $\tilde{f}(y) = f(Ty)$ with starting point $y^{(0)} = T^{-1}x^{(0)}$ are

$$y^{(k)} = T^{-1}x^{(k)}$$

Examples

example in $\mathbf{R}^2$

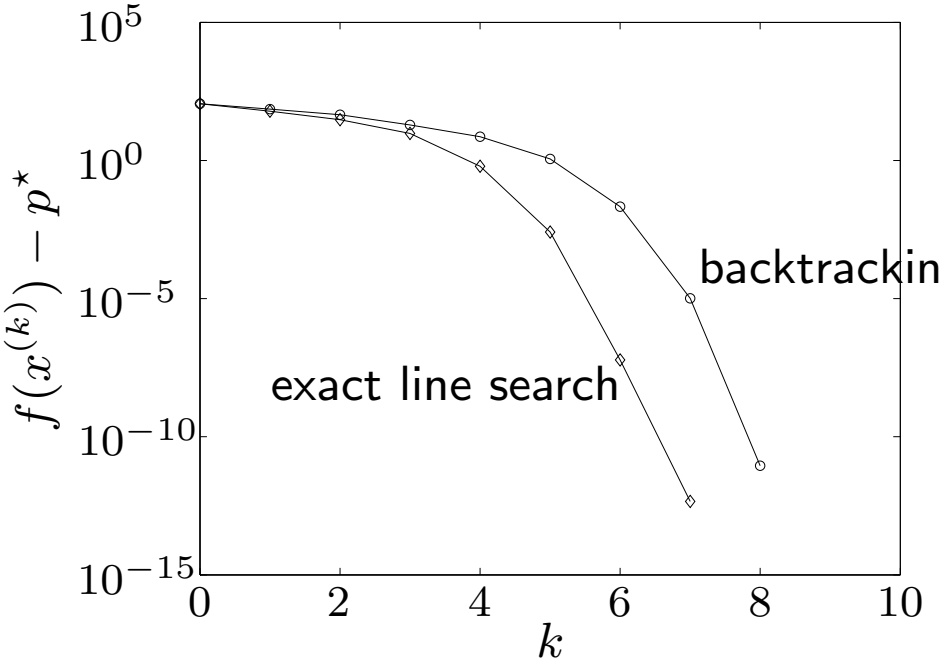


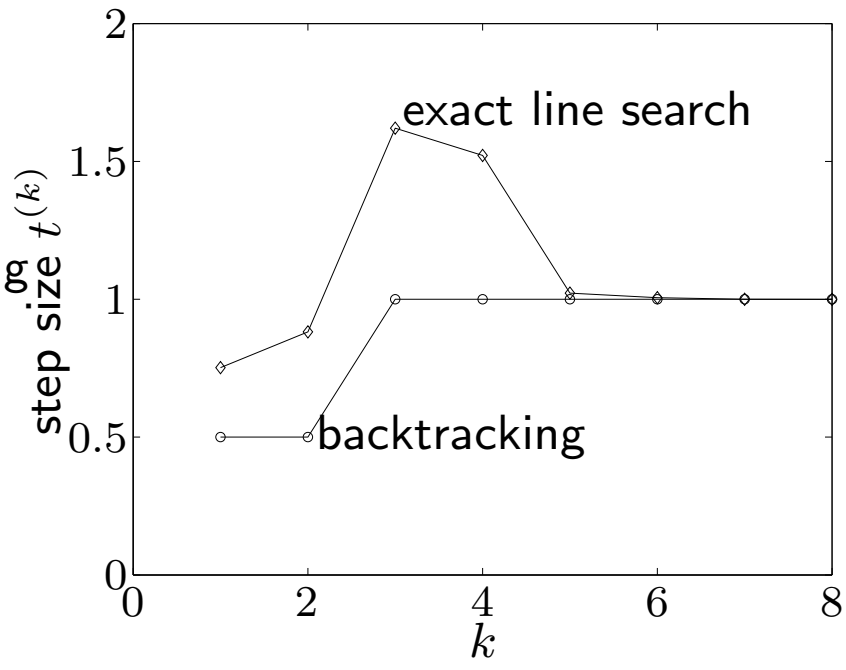


- converges in only 5 steps
- quadratic local convergence

Examples

example in $\mathbf{R}^{100}$

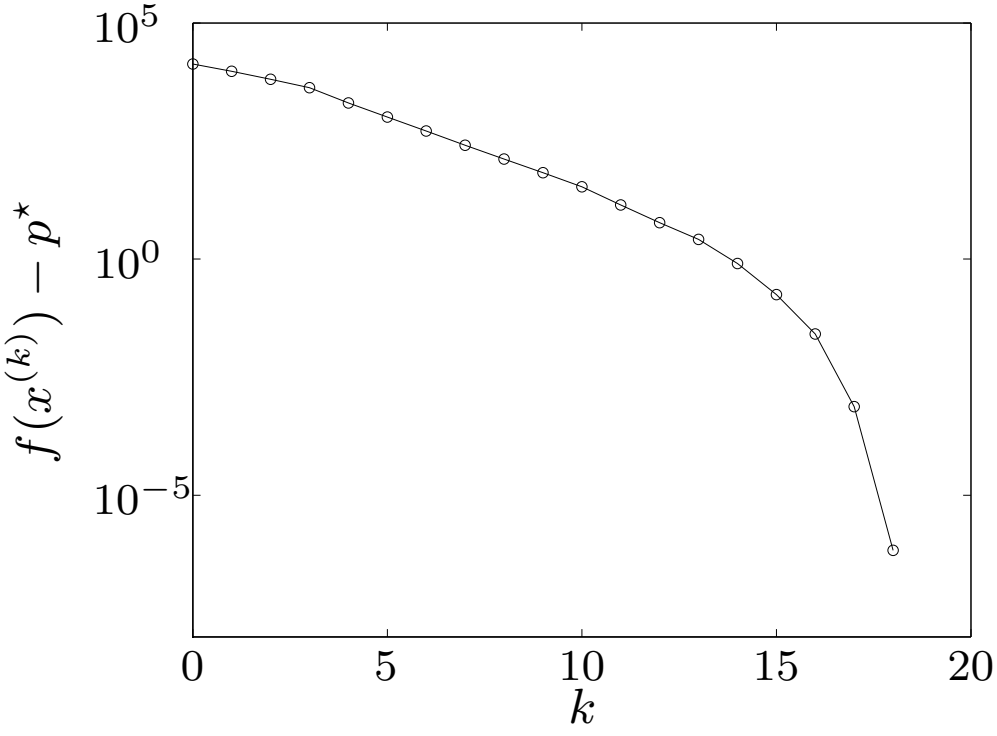




Examples

example in $\mathbf{R}^{10000}$ (with sparse a_i)

$$f(x) = -\sum_{i=1}^{10000} \log(1 - x_i^2) - \sum_{i=1}^{100000} \log(b_i - a_i^T x)$$



Outline

Optimization in machine learning

- usually interested in 'composite objective' problems of the form

$$\text{minimize} \quad g(x) + h(x)$$

  with

$$g(x) = \sum_{i=1}^{N} g_i(x), \quad h(x) = \sum_{k=1}^{K} h_k(x_k)$$

- try to exploit this (and additional) structure, taking account of
  - some or all of N, n, K may be (very) big
  - assumptions on g, h (convex? smooth?)
  - properties of problem data (storage/access? streaming/changing?)
  - generally don't care about very high accuracy solutions (why?)
- will give a few representative examples (without detailed discussion of convergence or behavior); these methods have many variations

Coordinate descent

- coordinate descent method for minimizing f

$$
\begin{aligned}
x_1^{k+1} &= \operatorname*{argmin}_{x_1} f(x_1, x_2^k, x_3^k, \ldots, x_n^k) \\
x_2^{k+1} &= \operatorname*{argmin}_{x_2} f(x_1^{k+1}, x_2, x_3^k, \ldots, x_n^k) \\
&\vdots \\
x_n^{k+1} &= \operatorname*{argmin}_{x_n} f(x_1^{k+1}, x_2^{k+1}, x_3^{k+1}, \ldots, x_n)
\end{aligned}
$$

- often take x_i to be blocks (**block coordinate descent**)
- for two blocks, called **alternating minimization**

Stochastic gradient descent

- **batch gradient descent** for additive objective is

$$x^{k+1} = x^k - \alpha \nabla f(x^k) = x^k - \alpha \nabla \sum_{i=1}^{N} f_i(x)$$

- **stochastic gradient descent** (also called **incremental** or **online**)

$$x^{k+1} = x^k - \alpha \nabla f(x^k) = x^k - \alpha \nabla f_i(x)$$

  where i iterates over $[N]$

  - batch: use all N examples each iteration
  - stochastic: use 1 example each iteration
  - mini-batch: use b examples each iteration

- natural choice for streaming data

Proximal gradient method

- given problem of minimizing $g + h$, proximal gradient method is

$$x^{k+1} := \mathbf{prox}_{\alpha^k h}(x^k - \alpha^k \nabla g(x^k))$$

  where

$$\mathbf{prox}_{\lambda f}(v) = \operatorname*{argmin}_x \left(f(x) + \frac{1}{2\lambda}\|x - v\|_2^2 \right)$$

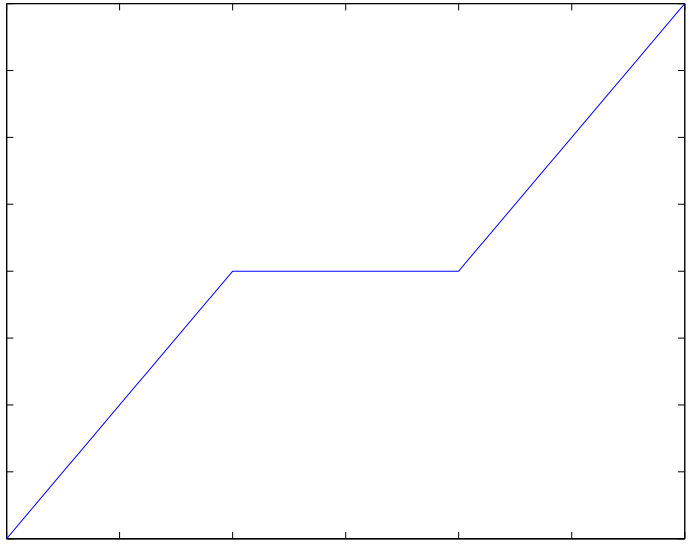
  is called the **proximal operator** of f with parameter $\lambda > 0$

- here, g is convex and smooth and h is convex

- proximal operators seem complex, but can be evaluated very efficiently for many functions that come up in machine learning and statistics problems, especially nonsmooth ones

Soft thresholding

$$\mathbf{prox}_{\lambda\|\cdot\|_1}(v) = (v-\lambda)_+ - (-v-\lambda)_+ = \begin{cases} v_i - \lambda & v_i \geq \lambda \\ 0 & |v_i| \leq \lambda \\ v_i + \lambda & v_i \leq -\lambda \end{cases}$$



Accelerated proximal gradient method

- idea: use information from previous time steps

$$\begin{aligned} y^{k+1} &:= x^k + \omega^k (x^k - x^{k-1}) \\ x^{k+1} &:= \mathbf{prox}_{\alpha^k h}(y^{k+1} - \alpha^k \nabla g(y^{k+1})) \end{aligned}$$

  where $\omega^k \in [0, 1)$ is an extrapolation parameter that must be chosen appropriately to achieve the acceleration, *e.g.*, $\omega^k = k/(k+3)$

- stated here for composite case, but acceleration often used in 'regular' gradient descent method

- note: accelerated methods are generally **not** descent methods

Sparse logistic regression