STAT 535

11/8/22

Lecture 12

ELECTION DAY!

- Training

- HW 4 - NN training
  → HW 5
  extra credit
- Q2 on Thu beginning of class

Lecture IV: Training predictors, Part I

Marina Meilă
mmp@stat.washington.edu

Department of Statistics
University of Washington

October, 2022

Analytic optimization ✔

Optimization generalities ← • • • •
- Optimization glossary
- Descent methods zoo

② what makes opt. problem hard

Descent methods
- The steepest descent method ✔
- Line minimization algorithms
- The Newton-Raphson method
- Examples: Logistic regression and Backpropagation ← ①

+ Examples

Reading HTF Ch.: Lasso 3.1,2,4, Logistic regression 4.4, Neural networks 11, Murphy Ch.: Ridge regression (including numerics) 7.5, Descent methods 8.3.2,3,5, Neural networks 16.5.1–4, Autoencoders 28, Bach Ch.: 5.

For more mathematical background, look e.g. at "Numerical recipes" chapter 10 or, for really advanced treatment Nocedal and Wrigth (Ch 3, 6).

Regularized Linear Regression

- L2 regularized (linear) LS regression (**Ridge regression**)

$$J(f;\mathcal{D}) = \sum_{i=1}^{n}(y^i - \beta^T x_i)^2 + \lambda||\beta||^2 \quad (5)$$

Solution $\hat{\beta} = (\mathbf{X}^T\mathbf{X} + \lambda I_n)^{-1}\mathbf{X}^T Y$ Exercise Derive it.

- L1 regularized (linear) regression (**LASSO**)

$$J(f;\mathcal{D}) = \sum_{i=1}^{n}(y^i - \beta^T x_i)^2 + \lambda||\beta||_1 \quad (6)$$

By contrast with the previos problem(s), this one does not have an analytic solution.

Descent methods

Many unconstrained optimization methods for finding a local minimum are of the form:

$$x^{k+1} = x^k + \eta^k d^k \tag{13}$$

where $d^k \in \mathbb{R}^d$ represents an **(unnormalized) direction** and $\eta^k > 0$ is a scalar called the **step size**.

Direction choice

- gradient based $d^k = -D^k \nabla f(x^k)$ with $D^k \in \mathbb{R}^{n \times n}$
  - steepest descent $D^k = I$
  - stochastic gradient (more about it later)
  - Newton-Raphson $D^k = \nabla^2 f(x^k)^{-1}$
  - conjugate gradient – implicity multistep rescaling of the axes "equivalent" to $D^k = \nabla^2 f(x^k)^{-1}$
  - quasi-Newton – implicit multistep approximation of $D^k = \nabla^2 f(x^k)^{-1}$
- non-gradient based
  - coordinate descent $d^k =$ one of the basis vectors in $\mathbb{R}^d$

Step size choice

- line minimization $\eta^k = \min_\eta f(x^k + \eta d^k)$
- Armijo rule (also called Backtracking) = search but not minimization
- constant step size $\eta^k = s$
- diminishing step size $\eta^k \to 0$; $\sum_k \eta^k = \infty$

Steepest descent

Algorithm STEEPEST-DESCENT

Input x^0 initial point ← random, or sometimes several random init

For $k = 0, 1, \ldots$

1. calculate $d^k = \nabla f(x^k)$
2. find η^k by line minimization ✓ η fixed OK but not optimal
3. $x^{k+1} \rightarrow x^k - \eta^k d^k$

until stopping condition satisfied

Output x^{k+1}

NEVER INVERT A MATRIX and other rules of the numerically savvy

. . . unless the inverse matrix is the last result you need from your computations. You may not think of it much, but data in computers has finite precision, and all our results have numerical errors. Here are a few simple rules to get the same result faster and often with a smaller numerical error

- To calculate $z = A^{-1}b$ call a linear system solver for $Az = b$. This is twice as fast for general A.
- If A is symmetric (in a linear system, eigenvalue problem, etc), tell your solver. You save time (another factor of 3) and the results are much more accurate.
- (An advanced one) Solve all large linear systems iteratively.
- Computing the product $ABCd$ where A, B, C are matrices and d is a vector: $((AB)C)d$ takes $2n^3 + n^2$ operations, $A(B(Cd))$ takes $3n^2$.
- Adding small numbers to large numbers: in a computer, $1. + \varepsilon = 1.$ if $\varepsilon < 1e - 16$ or so. Use the log-sum-exp trick (Murphy) or don't waste computer time doing it (more examples of the former later).
- Ask me about: numerical precision vs convergence precision vs statistical precision

Example: Logistic regression

+1 → 1
−1 → 0

Training = Estimating the parameters by Max Likelihood

Problem setup

$y \in \pm 1 \Rightarrow y_* \in \{0, 1\}$

- Denote $y_* = (1 - y)/2 \in \{0, 1\}$
- The likelihood of a data point is $P_{Y|X}(y|x) = \frac{e^{y_* f(x)}}{1+e^{f(x)}}$
- The log-likelihood is $l(\beta; x, y) = y_* f(x) - \ln(1 + e^{f(x)})$
- Log-likelihood of the data set $\mathcal{D}$

$$l(\beta; \mathcal{D}) = \frac{1}{n} \sum_{i=1}^{n} l(\beta; (x^i, y^i)) \tag{21}$$

- Define the loss function

$$L_{logl}(\beta) = -l(\beta) \tag{22}$$

- and the optmization criterion

$$J(\beta) = \hat{L}_{logl} = \frac{1}{n} \sum_{i=1}^{n} -l(\beta; x^i, y^i) \tag{23}$$

$\Leftrightarrow$ max log $\ell(\beta)$

Minimizing J is maximizing $l(\beta; \mathcal{D})$

Calculating the gradient

- $\frac{\partial l}{\partial f} = y_* - \frac{1}{1+e^f}$

  This is a scalar, and $\operatorname{sgn} \frac{\partial l}{\partial f} = y$
- We have also $\frac{\partial f(x)}{\partial \beta} = x$
- Now, the gradient of l w.r.t the parameter vector β is

$$\frac{\partial l}{\partial \beta} = \frac{\partial l}{\partial f}\frac{\partial f}{\partial \beta} = \left(y_* - \frac{1}{1+e^{-f(x)}}\right)x \tag{24}$$

Interpretation: The infinitezimal change of β to increase log-likelihood for a single data point is along the direction of x, with the sign of y Exercise Prove that (23) has a unique local optimum.

Calculating the gradient

- $\frac{\partial l}{\partial f} = y_* - \frac{1}{1+e^f}$
  This is a scalar, and $\operatorname{sgn} \frac{\partial l}{\partial f} = y$
- We have also $\frac{\partial f(x)}{\partial \beta} = x$
- Now, the gradient of l w.r.t the parameter vector β is

$$\frac{\partial l}{\partial \beta} = \frac{\partial l}{\partial f}\frac{\partial f}{\partial \beta} = \left(y_* - \frac{1}{1+e^{-f(x)}}\right) x \qquad (24)$$

$\ell = -\text{Loss}$

Interpretation: The infinitezimal change of β to increase log-likelihood for a single data point is along the direction of x, with the sign of y Exercise Prove that (23) has a unique local optimum.

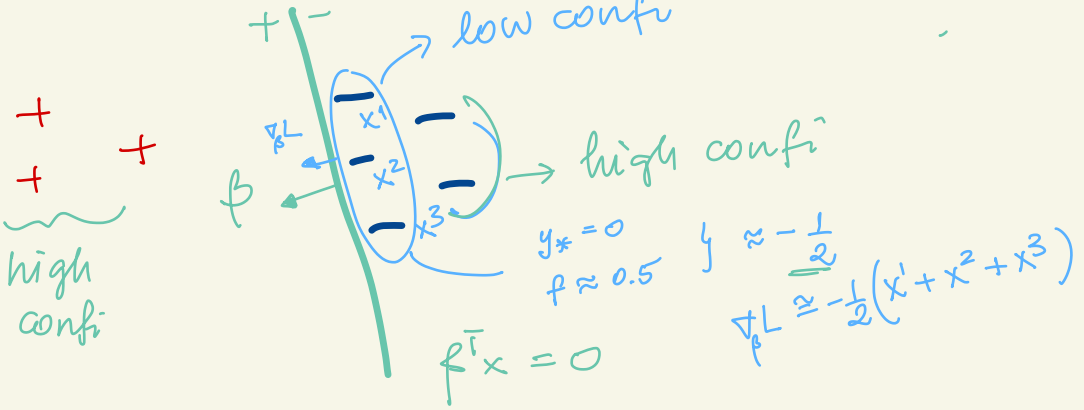
Algorithm STEEPEST-DESCENT FOR LOGISTIC REGRESSION

Input $\beta^0 \in \mathbb{R}^d$ initial point
For $k = 0, 1, \ldots$

1. calculate $d^k = \frac{1}{n}\sum_{i=1}^n \left(y_*^i - \frac{1}{1+e^{-f(x^i)}}\right) x^i \equiv \nabla_\beta \ell$
2. find η^k by line minimization
3. $\beta^{k+1} \to \beta^k + \eta^k d^k$

until stopping condition satisfied
Output β^{k+1}

scalar $\in [0,1] \Leftrightarrow$ weight for x^i
$\in \mathbb{R}^d$
$\Rightarrow$ lin combi of x^i for which weight $\neq 0$ w.h. confidence
or η constant
≈ 0 when f correct
≈ 1 — " — f wrong — " —



See next page

Rem $\beta^0 = 0$

$$\beta^k = \eta\left[\nabla_{\beta^0} L + \nabla_{\beta^1} L + \dots \nabla_{\beta^{k-1}} L\right] = \text{linear}(x^{1:n})$$

$\Downarrow$

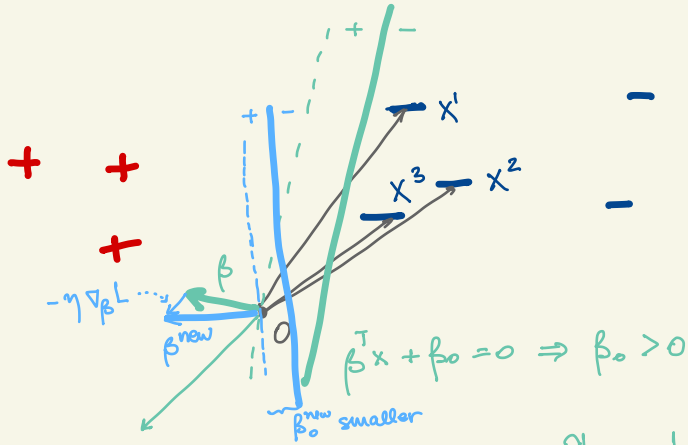
$\beta^k \in \text{span}\{x^{1:n}\}$

- L has only global opt $\Rightarrow$ no random restarts needed
- Ex: $x^{new} = x^{new}_{\mathcal{D}} \oplus x^{new}_{\perp}$ ($\perp$ on $\text{span}\{x^{1:n}\}$)

$x^{new}_{\mathcal{D}} \in \text{span}\{x^{1:n}\}$

$$\text{span}\{v^{1:k}\} = \left\{\sum_{j=1}^{k} \alpha_j v^j,\ \alpha_{1:k} \in \mathbb{R}\right\}$$

$f(x^{new}) = ?$ and how does it depend on $\mathcal{D}$?



$\beta^T x + \beta_0 = 0 \Rightarrow \beta_0 > 0$

$\nabla_\beta L \approx \frac{1}{2}(x^1 + x^2 + x^3)$

weights depend on β

$\frac{\partial L}{\partial \beta_0} = -\frac{1}{2}(1+1+1) = -\frac{3}{2}$ reduce threshold

Example: Backpropagation

- The **Backpropagation** algorithm is steepest descent for neural networks
- Consider a two layer neural network

$$f(x) = \sum_{j=1}^{m} \beta_j z_j = \sum_{j=1}^{m} \beta_j \phi(\sum_{k=1}^{n} w_{kj} x_k) \tag{25}$$

The parameters are β and $W = [w_{kj}]_{j=1:m,k=1:n}$

- Let the loss be L_{LS} the Least Squares loss, $J(\beta, W) = \hat{L}_{LS}(\beta, W)$

Derivation of the gradient Exercise Derive this

$$\frac{\partial J}{\partial \beta_j} = \frac{1}{n} \sum_i \frac{\partial (y^i - f(x^i))^2}{\partial \beta_j} = \frac{1}{n} \sum_i 2(y^i - f(x^i)) z_j(x^i) \tag{26}$$

$$\frac{\partial J}{\partial w_{kj}} = \frac{1}{n} \sum_i \frac{\partial L_{LS}(y^i, f(x^i))}{\partial z_j(x^i)} \frac{\partial z_j(x^i)}{\partial w_{kj}} = \frac{1}{n} \sum_i \left(2\beta_k (y^i - f(x^i))\right) \underbrace{z_j(x^i)(1 - z_j(x^i))}_{\phi'} x_k^i$$

$$= \frac{\beta_k}{n} \sum_i x_k^i (y^i - f(x^i)) \nabla(\text{logistic regressor}) \tag{27}$$

In the above we used the identity $\phi' = \phi(1 - \phi)$ Exercise Prove it

Local optima!!

Computational savings

- when $f(x^i)$ is computed, $z_j(x^i)$ are too; they should be "cached" and re-used
- the derivative of ϕ is easily obtained from the ϕ value
- Exercise The above gradient formulas can be easilty written in matrix-vector form

Backpropagation extends recursively to multi-layer networks. Exercise Derive it. Exercise Calculate the gradient for the 2 layer neural network with logistic output.

Practical properties of backpropagation

Ex $x^{1:n}$ have non-zero mean. Good/Bad for Logistic R The same

random restarts!!

- Unlike in logistic regression, J has many local optima even for two layers and simple problems.
- Hence, initialization is important, and there are no general rules for a good initialization. Even if the neural network works well, we do not know if we are at the optimum.
- Saturation If $\tilde{z}_j = w_j^T x$ is large in magnitude, then $z_j = \phi(\tilde{z}_j)$ is near 0 or 1. In either case, $\phi'(\tilde{z}_j) = z_j(1 - z_j) \approx 0$. We say that that this sigmoid is **saturated**; z_j will be virtually insensitive to changes in w_j[2]
  To avoid saturation at the beginning of the training, one initializes W with "small" (w.r.t $\max ||x^i||$, random values. Exercise Why random and not exactly 0?
- To speed up training, it is useful to standardize the input data[3] $x^{1:N}$ as a preprocessing step. Exercise Note that theoretically shifting and rescaling the data should NOT have any effect.
- J can have **plateaus**, i.e. regions where $\nabla J \approx 0$ but that do not contain a local minimum. Exercise What can cause plateaus? Exercise And what is bad about them?

Solution: Heavy Ball

- In conclusion, training neural networks by backpropagation is an art: requires experience with the algorithm, careful tuning, repeated restarts, and a long time.

How to initialize: $\beta = 0$, $W \neq 0$? ? $W \sim$ iid. uniform $[a, -a]$

large $|w_{jk}| \Rightarrow$ w.h.p $|\sum w_{jk} x^i_k|$ or large, $\nabla_{w_{jk}} \approx 0$

a small ~~large~~ →

[2] or to changes in previous layers, if this is a multilayer network.

[3] Standardization should NOT include the dummy coordinate $x^0 \equiv 1$.

Training autoencoders

simple

- The **autoencoder** is a neural network with one (or more) hidden layers, whose output y is identical with the input x.
- Let $x \in \mathbb{R}^d$, denote by $z \in \mathbb{R}^m$ the variables in the hidden layer, and by $\tilde{x} \in \mathbb{R}^d$ the output variables. Then,

representation

$$x_{1:n} \xrightarrow{W} z_{1:m} \xrightarrow{\tilde{W}} \tilde{x}_{1:n}$$

more hidden layer in real life

$$z_j = \phi(w_j^T x), j = 1:m \qquad \tilde{x}_k = \phi(\tilde{w}_k^T z), k = 1:n \tag{28}$$

where $W = [w_{ij}]_{i=1:n, j=1:m}$ and $\tilde{W} = [\tilde{w}_{kj}]_{j=1:n, k=1:m}$ are the parameters (or weights) to be learned.
Note that this is a neural network with multiple outputs

- The "labels" are the inputs $x_{1:n}$, and the cost is the least squares cost.

$$L(x, \tilde{x}) = ||x - \tilde{x}||^2 = \sum_{i=1}^{n} (x_i - \tilde{x}_i)^2 \tag{29}$$

If the variables x are binary, then the output layer will have a sigmoid, and the cost will be the logistic regression cost L_{logl}

- The training proceeds by backpropagation.

What is an autoencoder good for?

- Note that if $m \geq n$, we could set $W = \alpha I, \tilde{W} = \frac{1}{\alpha} I$; then z could be a (scaled) copy of x, and no training would be necessary to reproduce the input. Exercise Why do we need α at all?
  Thus, interesting autoencoders set $m < n$. If x can be reconstructed well from z, then we have succeeded to compress x, and we have learned in the process a set of descriptors, or a representation for x.
  If we want to have $m > n$, then we must use a sparsity inducing regularization (e.g L_1) to obtain an interesting representation.
- Autoencoders are the winning ingredient in Deep neural networks, and are a general method automatically find features for prediction.
  If the real problem is to predict another variable y from x, one can do as follows:
  1. Train an autoencoder for x, learn $W, \tilde{W}$.
  2. "remove" the top layer $\tilde{W}, \tilde{x}$, i.e discard all but W
  3. Construct a predictor $\hat{y} = f(z)$, with $z = \phi(Wx)$, where W are the autoencoder weights.

  Optional Do backpropagation to fine tune W.

  If f is linear or logistic, we obtain a two layer neural net $x \rightarrow z \rightarrow y$.
- The above can be applied recursively. Given $z^{1:N}$ the representations of the inputs $x^{1:N}$, one can now train an autoencoder $z_{1:m} \rightarrow u_{1:p} \rightarrow \tilde{z}_{1:m}$, perhaps with $p < m$.
  The interpretation is that $u_{1:p}$ are representing x at a higher level of abstraction than z.
- By using autoencoders, one can train multilayer neural networks, i.e. deep networks, while avoiding the plateaus that plague backpropagation. Hence "deep learning" is training with autoencoders.

Ex: Ridge Regression

$f(x) = \beta^T x$

$$J(\beta) = \frac{1}{n}\sum_{i=1}^{n}(y^i - \beta^T x^i)^2 + \lambda\|\beta\|^2$$

$$\nabla_\beta = \underbrace{\left[\nabla_\beta(\beta^T x^i - y^i)\right]}_{x^i \in \mathbb{R}^d} \cdot \underbrace{2(\beta^T x^i - y^i)}_{\mathbb{R}}$$

$$\nabla_\beta\left(\|\beta\|^2\right) = \nabla_\beta\, \beta^T\beta = 2\beta$$

$x, \beta \in \mathbb{R}^d$

$$\nabla_\beta J = \frac{-2}{n}\sum_{i=1}^{n}\underbrace{\left[(y^i - \beta^T x^i)\cdot x^i\right]}_{\text{span}\{x^{1:n}\}} + 2\lambda\underbrace{\beta}_{\text{linear in } \beta}$$

$$\beta^{k+1} \leftarrow \beta^k - \eta^k \nabla_\beta J = \beta^k\underbrace{(1 - 2\lambda\eta^k)}_{\epsilon(0,1),\ \text{small}} + \frac{2}{n}\sum_{i=1}^{n}[\cdots]$$

$\downarrow$ shrinkage

How to evaluate an optimization method? [Optional]

- Does it converge to a minimum?

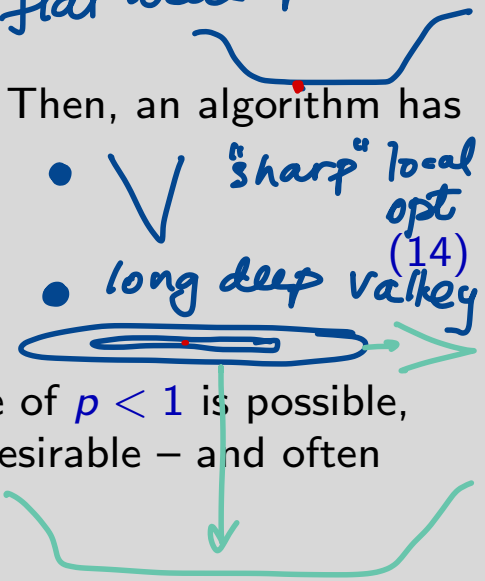
Good

BAD: Diverge, Slow convergence

As we shall see, all the methods described here converge to a minimum, but some of the require the function f to have additional "good" properties. A method which converges for larger classes of f's is more **robust**.

- How fast?

Difficult: - plateaus
- flat local opt
- "sharp" local opt
- long deep valley



Answer is usually in terms of **rates of convergence**
Let $e^k = x^k - x^*$ or $e^k = f(x^k) - f(x^*)$ denote the "error" at step k. Then, an algorithm has a **rate of convergence of order** p if

$$||e^{k+1}|| \leq \beta(||e^k||)^p \quad \text{for some } 0 < \beta < 1 \tag{14}$$

In the above, $p > 0$ but not necessarily an integer.
Most common cases are $p = 1$ (**linear**[1]) and $p = 2$ (quadratic). A rate of $p < 1$ is possible, and relevant for machine learning (see Part II). Superlinear scales are desirable – and often achievable.
Some modern machine learning algorithms have **sublinear** rates, e.g

$$||e^k|| \leq \frac{\beta}{k}||e^0|| \tag{15}$$

This is considered a slow convergence rate in classical optimization. Exercise Why may we like this rates in statistics/machine learning?

- Practical issues: Is it easy to implement or tune? Available software?

[1] The use of the term "linear" here is inconsistent with its use in e.g complexity theory. If an optimization algorithm is *linear*, that means that the error decreases **exponentially** with k, as $||e^k|| \leq \beta^k||e^0||$.

Computational complexity – theory and practice

In optimization problems, there are various ways of expressing the computational complexity of an algorithm:

- number of flops (floating point operations) per iteration, usually as a function of n the dimension of the problem
- *number of function* or gradient evaluations per iteration
- *number of iterations*; this latter quantity is given implicitly, by the rate of convergence.
- *memory requirements*
- With the increased complexity and variation of computer systems, the above mentioned number of operations is becoming obsolete. Algorithms are increasingly judged by other, system-related qualities, like: type of memory access (do they access memory in blocks or randomly), cache misses, etc. These criteria are beyond the scope of this course, but what you need to remember is that the texbook properties on an algorithm alone do not always predict its performance on the system you are going to run it. You may need to experiment with parameters and with algorithms to determine which algorithm is better suited for your data and system.