CSE 546 HW #notes

Sam Kowash

October 23, 2018

(1) LASSO

Selects for sparse predictor w. We may want this either for efficiency or human-legibility. How do we do this? Well, greedy approach adds features one at a time based on improvement in test error, but that's pretty hacky. How do we know when to stop? How do we avoid just including a billion features?

Looking for sparse results is a type of regularization; we want to penalize feature overselection. This motivates the lasso objective:

$$\hat{w}_{lasso} = \arg\min_{w} \sum_{i=1}^{n} \left(y_i - x_i^T w\right)^2 + \lambda \|w\|_1. \tag{1.1}$$

This punishes big vectors w, which is what we want. Fact: for any $\lambda \geqslant 0$ for which $\hat{w}_r$ finds the minimum, there exists $\nu \geqslant 0$ such that

$$\hat{w}_r = \arg\min_{w} \sum_{i=1}^{n} \left(y_i - x_i^T w\right)^2 \text{ subject to } r(w) \leqslant \nu. \tag{1.2}$$

That is, regularized regression problems can always be reframed as constrained optimization.

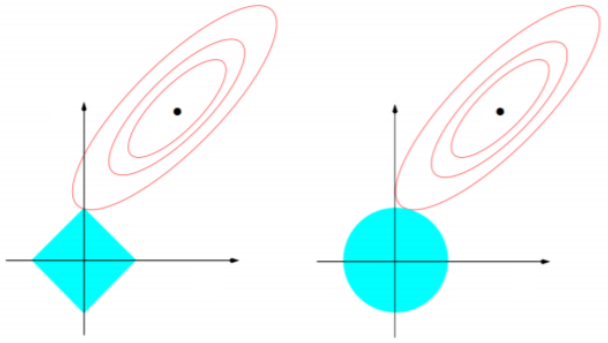

Figure 1.1: Lasso on left, ridge on right; lasso prefers solutions along coordinate axes (i.e. sparse)

If we incorporate an offset,

$$\hat{w}, \hat{b} = \arg\min_{w,b} \sum_{i=1}^{n} \left(y_i - (x_i^T w + b)\right)^2 + \lambda \|w\|_1, \tag{1.3}$$

but joint optimization is a pain, so let's prefer to de-mean our data. This is still actually kind of tricky minimization; the 1-norm isn't differentiable at the origin and this complicates things. Do by coordinate descent, minimize one direction at a time. This is guaranteed to approach the optimum for lasso (which is nice), but how do we pick our order of coordinate descent? Options,

- Random each time
- Round robin
- Try to pick "important" coordinates (biases us).

Let's see an example. Take $j \in \{1, \ldots, d\}$.

$$\sum_{i=1}^{n} \left(y_i - x_i^T w\right)^2 + \lambda \|w\|_1 = \sum_{i=1}^{n} \left(y_i - \sum_{k=1}^{d} x_{i,k} w_k\right)^2 + \lambda \sum_{k=1}^{d} |w_k| \tag{1.4}$$

$$= \sum_{i=1}^{n} \left(\left(y_i - \sum_{k \neq j} x_{i,k} w_k\right) - x_{i,j} w_j\right)^2 + \lambda \sum_{k \neq j} |w_k| + \lambda |w_j|. \tag{1.5}$$

So set $\hat{w}_k = 0$ for $k \in \{1, \ldots, d\}$ and loop over points:

$$r_i^{(j)} = \sum_{k \neq j} x_{i,j} \hat{w}_k \tag{1.6}$$

$$\hat{w}_j = \arg\min_{w_j} \sum_{i=1}^{n} \left(r_i^{(j)} - x_{i,j} w_j\right)^2 + \lambda |w_j|. \tag{1.7}$$

Pulling out one 1-d problem at a time! Works b/c lasso objective is *separable*. Except... this isn't actually a lot better. Hard to optimize because of pointy bit. Need to extend some concept of derivative and convexity. Traditional definition for fn is that lines b/t points on $f(x)$ lie above $f(x)$ (epigraph is convex). We need a different one here:

$$f(y) \geqslant f(x) + \nabla f(x)^T (y - x) \quad \forall x, y \tag{1.8}$$

This amounts to saying that there's a "supporting hyperplane" touching the epigraph at x such that the epigraph lies entirely on one side of the plane. If the function is differentiable at x, this is going to be the tangent plane. If not, we may have *many* options, and these are called *subgradients* (denoted ∂_{w_j} for subgradient set at w_j). We call differentiable functions extremized at x where $\nabla f(x) = 0$, and similarly we will call other functions extremized when $f(x)$ admits 0 as a subgradient at x.

OK, so how do we actually take subgradients and set them to zero? Consider example,

$$\partial_{w_j} \left(\sum_{i=1} \left(r_i^{(j)} - x_{i,j} w_j\right)^2 + \lambda |w_j|\right) = \begin{cases} a_j w_j - c_j - \lambda & \text{if } w_j < 0 \\ [-c_j - \lambda, -c_j + \lambda] & \text{if } w_j = 0 \\ a_j w_j - c_j + \lambda & \text{if } w_j > 0 \end{cases}, \tag{1.9}$$

where

$$a_j = \left(\sum_{i=1}^{n} x_{i,j}^2\right), \qquad c_j = 2 \left(\sum_{i=1}^{n} r_i^{(j)} x_{i,j}\right). \tag{1.10}$$

This tells us how to do our minimization (look for regime that contains zero as subgrad):

$$\hat{w}_j = \begin{cases} \frac{c_j + \lambda}{a_j} & \text{if } c_j < -\lambda \\ 0 & \text{if } |c_j| \leqslant \lambda \\ \frac{c_j - \lambda}{a_j} & \text{if } c_j > \lambda \end{cases} \tag{1.11}$$

where, recall,

$$a_j = \sum_{i=1}^{n} x_{i,j}^2 \qquad\qquad c_j = 2\sum_{i=1}^{n}\left(y_i - \sum_{k\neq j} x_{i,k}w_k\right)x_{i,j}. \tag{1.12}$$

This central flattening behavior provides "soft thresholding"; can make predictor entries identically zero depending on strength of regularization.

(2) Classification problems

Different from regression! Same principles though. Need a loss function; what is? Let's start with binary classification, want to learn $f : X \to Y$ where X contains features, $Y \in \{0, 1\}$ is target class. Natural loss is 0/1 function: $\mathbf{1}\{f(X) \neq Y\}$. Expected loss is then

$$\mathbb{E}_{XY}\left[\mathbf{1}\{f(X) \neq Y\}\right] = \mathbb{E}_X\left[\mathbb{E}_{Y|X}\left[\mathbf{1}\{f(x) \neq Y\} \mid X = x\right]\right] \tag{2.1}$$

$$= \mathbb{E}_X\Big\{\mathbf{1}\{f(x) = 1\}\mathbb{P}(Y = 0 \mid X = x) + \mathbf{1}\{f(x) = 0\}\mathbb{P}(Y = 1 \mid X = x)\Big\}. \tag{2.2}$$

Supposing we know $P(Y \mid X)$, the Bayes optimal classifier is

$$f(x) = \arg\max_y \mathbb{P}(Y = y \mid X = x). \tag{2.3}$$

How do we actually estimate $\mathbb{P}(Y \mid X)$? Well, can't do linear, and we're out of tricks now. Need a new trick; some function that goes from $\mathbb{R}^d$ to $[0, 1]$ (called link function). A nice option is the sigmoid/logistic curve:

$$\mathbb{P}(Y = 0 \mid X, W) = \frac{1}{1 + \exp\left[w_0 + \sum_i w_i X_i\right]}. \tag{2.4}$$

Still a sort of linear model in terms of what we do to our data variables. Note that w_0 applies a horizontal shift, and the w_i "stretch" the curve. Note a nice property for binary classification, which is that

$$\frac{\mathbb{P}(Y = 1 \mid w, X)}{\mathbb{P}(Y = 0 \mid w, X)} = \exp\left[w_0 + w^T X\right]. \tag{2.5}$$

Reasonable to make our rule to classify as 1 if ratio is greater than 1, classify as 0 if ratio is less than 1. Equivalent result,

$$\ln \frac{\mathbb{P}(Y = 1 \mid w, X)}{\mathbb{P}(Y = 0 \mid w, X)} = w_0 + \sum_i w_i X_i \to \begin{cases} < 0 \implies \text{classify as } 0 \\ > 0 \implies \text{classify as } 1 \end{cases}. \tag{2.6}$$

Alternative formulation (conditional likelihood): say $y \in \{-1, 1\}$ instead of $\{0, 1\}$ so we can then write

$$\mathbb{P}(Y = y \mid x, w) = \frac{1}{1 + \exp\left(-yw^T x\right)} \tag{2.7}$$

and find the MLE:

$$\hat{w}_{\text{MLE}} = \arg\max_w \prod_{i=1}^{n} \mathbb{P}(y_i \mid x_i, w) \tag{2.8}$$

$$\hat{w}_{\text{MLE}} = \arg\min_w \sum_{i=1}^{n} \underbrace{\log\left(1 + \exp\left(-y_i x_i^T w\right)\right)}_{\sigma(y_i x_i^T w)} \tag{2.9}$$

Call the objective $J(w)$ (logistic loss): what the hell does it look like? Is it convex? (Yes.) How do we minimize it? Note that for an argument z, $\sigma(z) \sim |z|$ as $z \to -\infty$ and $\sigma(z) \to 0$ as $z \to +\infty$. In between, does some nonsense. So $\sigma(z)$ is easy to understand, but what about $J(w)$? Generally, no closed form in terms of w, can't just take derivative and set to zero. It turns out, moreover, that if we happen to have a w that correctly classifies every point (by dumb luck; linearly separable data set), then $y_i x_i^T w$ is strictly positive and so for $t > 0$, $y_i x_i^T (tw) > y_i x_i^T w$ and we will always prefer to infinitely extend our w along our "good" direction to continue reducing loss. This breaks the classifier. Sigmoid classifiers become step functions. We *need* to regularize this conditional log likelihood objective.

(3) Interlude: Gradient descent

Recall that our general problem is to be handed i.i.d. data $\{(x_i, y_i)\}_{i=1}^n$ with $x_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$, a model with some parameters w, and some loss function depending on w to be calculated on our data, then optimize w with respect to that loss. The computational meat is in optimization. We turn it into a problem that looks like $X^T X w = X^T Y$, but how do we actually do that? Lots of different ways! Many algorithms! Some depend on properties of matrices, sparseness, etc. Let's look at one in particular, which is gradient descent.

Say I have some function $f(x)$ where $x \in \mathbb{R}^d$. Taylor tells us that

$$f(x + \delta) = f(x) + \nabla f(x) \cdot \delta + \cdots, \tag{3.1}$$

so at any point we can find the gradient and take a step in the opposite direction, which must decrease f. Ex:

$$f(w) = \frac{1}{2}\|Xw - y\|_2^2 \tag{3.2}$$

$$\nabla f(w) = X^T(Xw - y) = X^T(Xw - y) = X^T X w - X^T y, \tag{3.3}$$

so if we are at some w_t our next step should be

$$w_{t+1} = w_t - \eta X^T (X w_t - y). \tag{3.4}$$

Examine recursion behavior, find (for optimum w_*) that

$$(w_{t+1} - w_*) = \left(I - \eta X^T X\right)^{t+1} (w_0 - w_*), \tag{3.5}$$

so can bound rate of approach to optimum. (Shocker: need η small.)

We can do one better by going one step further down the Taylor series,

$$f(y) \approx f(x) + f'(x)(y - x) + \frac{1}{2} f''(x)(y - x)^2 \tag{3.6}$$

make our next step the minimum

$$\tilde{y} = \frac{f'(x)}{f''(x)} \tag{3.7}$$

(4) Perceptron

Recall binary classification: we want to learn $f : X \to Y$ with loss $\ell(f(x), y) = \mathbf{1}\{f(x) \neq y\}$. Expected loss

$$\mathbb{E}_{XY}\left[\mathbf{1}\{f(X) \neq Y\}\right] = \mathbb{E}_X\left[\mathbb{E}_{Y|X}\left[\mathbf{1}\{f(x) \neq Y\} \mid X = x\right]\right] \tag{4.1}$$

$$\mathbb{E}_{Y|X}\left[\mathbf{1}\{f(x) \neq Y\} \mid X = x\right] = 1 - P\left(Y = f(x) \mod X = x\right), \tag{4.2}$$

so obviously want Bayes classifier

$$f(x) = \arg\max_y \mathbb{P}\left(Y = y \mid X = x\right), \tag{4.3}$$

and often take a logistic model like

$$P(Y = y \mid x, w) = \frac{1}{1 + \exp(-yw^T x)}. \tag{4.4}$$

Buuuuuut it's really hard to know if that model's good. Often probably isn't. Can we do this without a model? Certainly it's not hard to do it with our eyeballs. Can we give our computer eyeballs? Yup!

Say we're classifying to $y \in \{-1, 1\}$ and take a linear model where we predict $\operatorname{sgn}(w^T x + b)$. Start with $w_0 = 0, b_0 = 0$, get a data point x_k, predict y_k, check, and if we were wrong, update

$$\begin{bmatrix} w_{k+1} \\ b_{k+1} \end{bmatrix} = \begin{bmatrix} w_k \\ b_k \end{bmatrix} + y_k \begin{bmatrix} x_k \\ 1 \end{bmatrix}. \tag{4.5}$$

Basically we rotate/shift our classification line until we get everything right. Guaranteed to converge if we have linearly separable data! If γ is the width of the widest margin (hyperplanar slab separating classes) and feature vector norms are bounded by $\|x_k\| \leqslant R$, then there is a theorem (Block, Novikoff): if we have a sequence of examples (x_t, y_t), then the number of mistakes made by the perceptron is bounded by R^2/γ^2 for *any* such sequence. Convergence is guaranteed and *fast*.

This is pretty dank, but... we can't really do much with it. Most things aren't linearly separable (or even separable at all!), and even a single point in the way of separability can cause infinite cycling and destroy convergence. So why do we care at all? Well, it's a totally different way of thinking about learning! Previously we wrote down a model and then developed an algorithm that optimizes it. Here we did exactly the opposite: wrote down an algorithm and analyzed it. Is there an equivalent modeling problem? What is the perceptron optimizing? Does it have a definable loss function? This all leads to support vector machines (SVMs).

(5) Support vector machines

The perceptron model admitted many possible classifiers for a given data set, and it liked them all just as well, but we are not so generous. Some seem to skirt very close to the edges of our classes,

which feels dangerous, and we'd be more comfortable with a boundary as far from our known data as possible, considering that our data may suffer perturbations on future collections.

How can we pick the "safest" classifier? Consider some hyperplane boundary $x^T w + b = 0$ that we've trained, and the closest point x_0 to it. How close is it? Let

$$\tilde{x}_0 = \underset{z \in \{x : x^T w + b = 0\}}{\arg\min} \|z - x_0\|_2^2 \tag{5.1}$$

be the closest point on the hyperplane to x_0. Then

$$\|x_0 - \tilde{x}_0\|_2 = \left\| \frac{w^T}{\|w\|_2} (x_0 - \tilde{x}_0) \right\|, \tag{5.2}$$

but we know that $w^T \tilde{x}_0 = -b$, so

$$\|x_0 - \tilde{x}_0\|_2 = \frac{|w^T x_0 + b|}{\|w\|_2}. \tag{5.3}$$

So if we now call γ the distance from the classification boundary to a parallel hyperplane on either side, our problem is to maximize γ over w and b subject to

$$\frac{y_i (x_i^T w + b)}{\|w\|_2} \geqslant \gamma \qquad \forall i \tag{5.4}$$

where the sign y_i makes sure we're paying attention to points on both sides of the boundary correctly. This problem ends up being equivalent, however, to the minimization of $\|w\|_2^2$ subject to

$$y_i \left(x_i^T w + b \right) \geqslant 1 \qquad \forall i, \tag{5.5}$$

which gives us both our optimal hyperplane and a measure of our separability for free.

However, this still breaks for non-separable data! To make this useful, we need to permit some misclassification or a smaller margin! We implement this as minimizing w subject to

$$y_i (x_i^T w + b) \geqslant 1 - \xi_i \tag{5.6}$$

$$\xi_i \geqslant 0 \tag{5.7}$$

$$\sum_j \xi_j < \nu. \tag{5.8}$$

We now have a knob to trade off between ν and $\|w\|_2$. The ξ_i are called support vectors (samples on the margin).

Constrained optimization sucks, but as we found with lasso, we have some tricks. Specifically, for any $\nu \geqslant 0$, there is a $\lambda \geqslant 0$ such that we can just as well minimize

$$\sum_{i=1}^{n} \max\{0, 1 - y_i (b + x_i^T w)\} + \lambda \|w\|_2^2. \tag{5.9}$$

We're back in machine learning land, now! We have a loss function

$$\ell_i(w) = \max\{0, 1 - y_i x_i^T w\} \tag{5.10}$$

which we call the *hinge loss* and can optimize by any of the methods we've talked about before. Fact: this is exactly the loss being optimized by the perceptron!

$$\nabla_w \ell_i(w, b) = \begin{cases} -y_i x_i + \frac{2\lambda}{n} w & \text{if } 1 - y_i (b + x^T w) > 0 \\ \frac{2\lambda}{n} w & \text{otherwise} \end{cases} \tag{5.11}$$