Foundations of ML and Data Science Notes

Maruth Goyal

Spring 2021

Contents

1 Preface

These notes are based very heavily on the lectures, and slides by Dr. Rachel Ward for the course CS395T: Foundations of Machine Learning and Data Science at The University of Texas at Austin from the Spring of 2021 (aka 2020 redux). Any errors introduced are mine.

2 Introduction

3 Linear Algebra Review

3.1 Matrix Norms

This section is heavily borrowed from the great Wikipedia article [Wikipedia, 2021].

Definition 3.1 (Matrix Norm)**.** *A matrix norm* $\|\cdot\| : \mathbb{R}^{d\times n} \to \mathbb{R}$ *is any real-valued function on matrices satisfying the following properties:*

1. $\|\alpha\mathbf{A}\| = |\alpha|\|\mathbf{A}\|$
2. $\|\mathbf{A}+\mathbf{B}\| \leq \|\mathbf{A}\| + \|\mathbf{B}\|$
3. $\|\mathbf{A}\| \geq 0$
4. $\|\mathbf{A}\| = 0 \iff \mathbf{A} = \mathbf{0}$

Property 3.1 (Submultiplicative Norms)**.** *A norm* $\|\cdot\|$ *is said to be* submultiplicative *if it satisfies*

$$\|\mathbf{AB}\| \leq \|\mathbf{A}\| \cdot \|\mathbf{B}\|$$

Remark. *The notation for matrix norms is* horrible. $\|\mathbf{A}\|_2$ *can mean the* 2*-operator norm, or the entry-wise* 2*-norm, or the Schatten* 2*-norm (Frobenius norm). Always make sure you know exactly which norm the authors mean, and if possible be explicit about it if/when you use the notation if not immedeately clear from context.*

3.1.1 Vector-induced (Operator) norms

Given any matrix $\mathbf{A} \in \mathbb{R}^{d\times n}$, and any vector norm $\|\cdot\| : \mathbb{R}^d \to \mathbb{R}$, we define the induced matrix norm to be

$$\|\mathbf{A}\| = \sup\left\{\|\mathbf{Ax}\| \mid \mathbf{x} \in \mathbb{R}^n, \|\mathbf{x}\| = 1\right\}$$

Special Cases

1. When the norm is $\|\cdot\|_1$, i.e. the 1-norm for vectors, we have

   $$\|\mathbf{A}\|_1 = \max_{1\leq j\leq n} \|\mathbf{A}_{\bullet,j}\|_1$$

   i.e., the largest 1-norm among the columns.

2. When the norm is $\|\cdot\|_2$, i.e. the 2-norm for vectors, we have

   $$\|\mathbf{A}\|_2 = \sigma_{\max}(\mathbf{A})$$

   i.e., the largest singular value of $\mathbf{A}$. For square matrices, equivalently this is the largest eigenvalue $\lambda_{\max}(\mathbf{A})$. Also known as the *Spectral Norm*.

Properties

1. Any operator norm is *submultiplicative*.

3.1.2 Entry-wise norms

Another way to use vector norms to create matrix norms is to simply flatten an $n \times d$ matrix into a vector of length nd, and then use a vector norm $\|\cdot\| : \mathbb{R}^{dn} \to \mathbb{R}$. Thus the entry-wise norm under the p-vector norm is

$$\|A\|_p = \left(\sum_{i,j} |a_{ij}|^p\right)^{1/p}$$

Special Cases

1. When $p = 2$, this is the Frobenius norm $\|\cdot\|_F$.
2. When $p = \infty$ this is the ℓ_∞ norm, i.e., simply the maximum value in the matrix.

3.1.3 Schatten p-norms

The Schatten p-norms are defined in terms of the singular values of the matrix $\mathbf{A}$. In particular, if $\{\sigma_i\}_{i\in[r]}$ are the singular values for a matrix $\mathbf{A}$, then the Schatten p-norm is

$$\|\mathbf{A}\|_p = \left(\sum_{i\in[r]} \sigma_i^p\right)^{1/p}$$

Special Cases

1. When $p = 1$, this is known as the *Nuclear Norm*.
2. When $p = 2$, this is again equivalent to the *Frobenius Norm* $\|\cdot\|_F$.

Properties

1. All Schatten norms are *submultiplicative*
2. The Schatten norms are invariant under unitary operations. i.e., $\|U\mathbf{A}V\|_p = \|\mathbf{A}\|_p$ where U, V are unitaries.

3.2 Frobenius norm

The Frobenius norm is a 2-norm as both a Schatten 2-norm, and an entry-wise norm. i.e.,

$$\|\mathbf{A}\|_F = \sqrt{\sum_{i\in[r]} \sigma_i^2} = \sqrt{\operatorname{tr}\left(\mathbf{A}^\top\mathbf{A}\right)} = \sqrt{\sum_{i,j} |a_{i,j}|^2}$$

It is also in fact the norm induced by the *Frobenius Inner Product*

Definition 3.2 (Frobenius Inner Product)**.** *Given* $\mathbf{A}, \mathbf{B}$ *the Frobenius inner product* $\langle \mathbf{A}, \mathbf{B}\rangle$ *is defined as*

$$\langle \mathbf{A}, \mathbf{B}\rangle_F = \operatorname{tr}\left(\mathbf{A}^\top\mathbf{B}\right)$$

Thus we also get the following equality for the Frobenius norm

Property 3.2. $\|A + B\|_F^2 = \|A\|_F^2 + \|B\|_F^2 + 2\langle A, B\rangle_F$

Property 3.3. $\|\mathbf{A}^\top\mathbf{A}\|_F = \|\mathbf{A}\mathbf{A}^\top\|_F \leq \|\mathbf{A}\|_F^2$

Of course, the norm also enjoys the properites of unitary invariance, and submultiplicity of Schatten norms.

3.3 Spectral Decomposition of Semi Positive Definite Matrices

3.4 Singular Value Decomposition

3.4.1 Principal Components Analysis

3.4.2 Power Method

3.4.3 QR Decomposition

4 Probability Review

4.1 The Gaussian Distribution

4.2 χ^2 Distribution

4.3 Concentration Inequalities

4.3.1 Markov's Inequality

4.3.2 Chebyshev's Inequality

4.3.3 Law of Large Numbers

4.3.4 Master Tail Bound

4.3.5 Chernoff Bounds

4.4 Sampling from Distributions

5 Geometry of High Dimensions

5.1 Properties of the Unit Ball

5.1.1 Surface Area of the Unit Ball

5.1.2 Volume of the Unit Ball

6 Curse and Blessings of High Dimensional Geometry

6.1 The Curse

There are many well-studied ways of interpolating functions in low dimensions. For instance, linear interpolation, polynomial interpolation, spline interpolation, etc. from numerical analysis (see subsection 11.2). However, these methods break down in high dimensions. This is because of the *curse of dimensionality*.

Curse 6.1 (Curse of Dimensionality)**.** *The number of samples $f(x_j)$ needed to meaningfully approximate a function $f : [0,1]^d \to \mathbb{R}$ increases* **exponentially** *in d.*

One way to see this is to see how many samples we would need to meaningfully approximate our domain. For instance, suppose our domain is the unit cube in d dimensions, $[0,1]^d$. Let's say we want to approximate the domain by splitting it up into cubes of side $1/4$. i.e., we will divide the unit cube into cubes of side $1/4$ and then just sample one point from each of those cubes as a reasonably approximation of the unit cube. Well, that's still $\mathbf{4^d}$ points!

Theorem 6.1. *Approximating the unit cube with samples from cubes of side $1/\epsilon$, requires $\mathcal{O}(\epsilon^d)$ samples. i.e., for any $\epsilon > 0$ we need to take $(1/\epsilon)^d$ samples $x_j \in [0,1]^d$ to form a grid resolution $\|\mathbf{x}_j - \mathbf{x}_k\| = \epsilon$ with respect to any norm $\|\cdot\| = \|\cdot\|_p$.*

Thus, in order to approximate functions in high-dimensions, we must be more clever. In particular, we need to exploit *strong additional information* about the function. Assumptions such as smoothness (i.e., existence of derivatives) are generally not strong enough.

6.2 The Unit Ball

As an example of how assumptions on our function might help us, we shall study functions which lie on the unit ball in d dimensions. i.e., we have $f : \mathbb{B}^d \to \mathbb{R}$ where $\mathbb{B}^d = \{x \in \mathbb{R}^d : \|x\|_2^2 \leq 1\}$. As cute and symmetrical as this domain seems, it has its own set of problems. For instance, observe that in d dimensions there are 2^d orthants. Thus, even to just sample points on the surface of the ball in the direction of each of these orthants we need 2^d points, and even then we get a grid resolution of at most $\sqrt{2}$.

But what if we know something more about our function, and so can use fewer samples? How do we just generate samples from the unit ball?

6.2.1 Sampling from the Unit Ball

With computers, it is common and standard to have access to a psuedorandom generator (PRNG) which produces a stream of numbers indistinguishable to most common programs from a uniform distribution $\mathcal{U}[0,1)$. Thus, given access to a randomness oracle to sample from $\mathcal{U}[0,1]$, how do we sample from $\mathbb{B}^d$?

Method 1: Sample from $[-1,1]^d$ One way to do this is to sample points from $\mathcal{U}[-1,1]^d$, and keep only those which are inside $\mathbb{B}^d$. This sounds great, and can be implemented in a single line of Python (though, what can't these days?), but there's an issue: in higher dimensions, we will end up keeping almost none of the points we sample. This happens because in high dimensions the ratio of the volume of the ball to the volume of the cube vanishes exponentially fast.

Theorem 6.2 (Volume of Unit Ball)**.** *The volume $V(d)$ of the unit ball $\mathbb{B}^d$ in d-dimensions is given as*

$$V(d) = \frac{\pi^{d/2}}{\frac{d}{2}\Gamma\left(\frac{d}{2}\right)}$$

Observe that since $\Gamma\left(\frac{d}{2}\right) \sim \left(\frac{d}{2}\right)!$, we have that $V(d)$ decays exponentially in d. Moreover, the ratio of this volume to the unit ball would be given by

$$\frac{\pi^{d/2}}{\frac{d}{2}\Gamma\left(\frac{d}{2}\right)} \cdot 2^{-d}$$

which decays at least as fast. Thus, for even relatively small d this method becomes infeasible very quickly.

Curse 6.2 (Volume Ratios in High Dimensions)**.** *The volume of the unit sphere decays to* 0 *exponentially fast in d. Moreover, the ratio of the volume of the unit sphere to that of the unit cube decays exponentially fast in d.*

Surely there is another way?

Method 2: When in doubt, Gaussian While the name of this paragraph is a bit of a giveaway, we shall nonetheless motivate why it's a good idea to call upon the mighty Gaussian. First, recall the density of the Gaussian (see subsection 4.1:

Definition 6.1 (Gaussian Distribution)**.** *The univariate Gaussian distribution with mean μ and variance σ^2, written as $\mathcal{N}(\mu, \sigma^2)$, is defined as the distribution with density*

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right)$$

Definition 6.2 (Multivariate Gaussian)**.** *The multivariate d-dimensional Gaussian distribution with mean $\mu \in \mathbb{R}^d$, and symnmetric positive semidefinite covariance matrix $\Sigma \in \mathbb{R}^{d\times d}$, written as $\mathcal{N}(\mu, \Sigma)$, is defined as the distribution with density*

$$p(x) = \frac{1}{(2\pi)^{d/2}\sqrt{|\Sigma|}} \exp\left(-\frac{1}{2} \cdot (x-\mu)^\top \Sigma^{-1}(x-\mu)\right)$$

The key thing to observe is, if we look at the multivariate Gaussian $\mathcal{N}(\mathbf{0}, \mathbf{I})$ then the density becomes

$$p(x) = \frac{1}{(2\pi)^{d/2}} \exp\left(\frac{-(x_1^2 + x_2^2 + \cdots + x_d^2)}{2}\right)$$

That is, the distribution is **spherically symmetric**. In other words, the density is the same for all vectors $x \in \mathbb{R}^d$ which lie on the surface of the same sphere. In particular, this means that sampling from this distribution amounts to sampling a random direction on the unit sphere (up to rescaling). You know what's even better? This is precisely the join density obtained by sampling d i.i.d points from $\mathcal{N}(0, 1)$.

Theorem 6.3. *Sampling $\frac{x}{\|x\|}$ where $\mathbf{x} \sim \mathcal{N}(0,1)^d$ is equivalent to sampling a point uniformly at random on the **surface** of the sphere.*

Are we done? Nope! We can now generate points uniformly at random on the **surface** of the unit sphere, but we want to generate points uniformly at random **inside** the unit sphere. i.e., the surface and everything inside. Intuitively, once we have random points on the sphere, we should be able to simply rescale appropriately to get a point inside right? The answer is yes, but with some care.

Suppose we simply sample a radius uniformly at random $r \sim \mathcal{U}[0, 1]$ and rescale our point to be $r \cdot \frac{\mathbf{x}}{\|\mathbf{x}\|}$. Does this work? Unfortunately, no. The reason is this way we will end up sampling **way** too many points close to the origin. Think about it this way: there are more points on the surface of the sphere of radius 1 than the sphere of radius 0.1. Thus, it makes sense to scale the probability of sampling a point at some radius with the number of points on the surface of the sphere of that radius. i.e., make it scale as the surface area of the sphere of radius r.

Theorem 6.4. *In d dimensions, the surface area of the sphere of radius r is given by*

$$S(r;d) = \frac{2\pi^{d/2}}{\Gamma\left(\frac{d}{2}\right)} \cdot r^{d-1}$$

Thus, we see that $S(r;d)$ scales as r^{d-1}. A quick integral shows that with appropriate normalization constants, the distribution we are looking for has density

$$p(r) = dr^{d-1}$$

Thus we have the following final result:

Theorem 6.5. *Sampling $r \cdot \frac{\mathbf{x}}{\|\mathbf{x}\|}$ where $r \sim dr^{d-1}$ and $\mathbf{x} \sim \mathcal{N}(0,1)^d$ is equivalent to sampling a point uniformly at random from $\mathbb{B}^d$.*

Note that we can sample from both $\mathcal{N}(0,1)$ and dr^{d-1} with access to only $\mathcal{U}[0,1]$ (see subsection 4.4).

A "blessing" of dimensionality Until now we have mentioned multiple curses of high dimensionality. But surely it isn't all cursed? Well, if it's an reassurance one of the blessings of high dimensionality is that we "know" how random i.i.d vectors behave in high dimensions. In particular, recall the following from subsubsection 4.3.4

Theorem. *For $X \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_d)$ the expected length of X is $\sqrt{d}$.*

Proof.

$$\mathbb{E}\left[\|X\|^2\right] = \mathbb{E}\left[\|X_1\|^2 + \cdots + \|X_d{}^2\|\right] = d\mathbb{E}\left[\|X_1\|^2\right] = d$$

□

Theorem (Gaussian Annulus Theorem)**.** *For a d-dimensional spherical Gaussian with unit variances, for any $\beta \leq \sqrt{d}$, all but at most $3\exp\left(-c\beta^2\right)$ of the probability mass lies within the annulus $\sqrt{d} - \beta \leq \|\mathbf{x}\| \leq \sqrt{d} + \beta$, where $c > 0$ is a fixed positive constant.*

These two theorems put together tell us that in high dimensions, if we sample random Gaussian vectors, then with **extremely** high probability these vectors will lie approximately on the sphere of radius $\sqrt{d}$. Thus, in high dimensions we have an increasingly confident sense of how these i.i.d Gaussian vectors behave.

7 Random Projections, Johnson-Lindenstrauss Lemma

8 Convex Functions

9 Gradient Descent

If you have spent more than an hour (or in some cases, a second) with someone involved in Computer Science (or increasingly, STEM in general) you have probably heard of gradient descent. The elevator pitch of the method is essentially "start with a guess, and then move in the direction of steepest decrease". Recall from multivariable calculus (or physics) that the negative gradient is the direction of steepest descent, hence the name. In this section we will first try to somewhat motivate the utility of gradient descent (as opposed to other optimization methods), and then take a gander at what it looks like in different settings. In particular, we will look at the various variants of GD, and analyze their performance in different settings.

9.1 Least Squares problems

9.1.1 Motivation and Setup

A simple way to build any sort of predictive model is to conjecture some form of a "feature basis", and then try to find the best fit of the data to this basis. For instance, we might want to find the best fit cubic polynomial, or combination of sine/cosine waves, or even just a best-fit linear model. In each case, it's a matter of finding the coefficients $\langle c_1, \ldots, c_k \rangle$ which results in our resultant function's evaluation on the training points match the expected output.

More formally, we may define a matrix $\mathbf{A} \in \mathbb{R}^{n \times d}$ such that

$$\mathbf{A}_{ij} = \varphi_i \left(x_i^{(j)} \right)$$

where $\langle \varphi_1, \ldots, \varphi_d \rangle$ defines our feature basis. Then, our task is to solve

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

where $\mathbf{x}$ is our vector of coefficients. We will generally consider the case where this system is overdetermined. i.e., where we have more training samples than coeffecients ($n > d$). In a practical setting, it is more feasible to instead look for an approximate solution $\mathbf{x}$ simply satisfying $\mathbf{A}\mathbf{x} \approx \mathbf{b}$.

One way to do it is to find $\mathbf{x}$ which minimizes some measure of "badness". We shall now refer to this measure as a "loss function", denoted as $\mathcal{L}$. Thus, we want to find

$$\min_{\mathbf{x} \in \mathbb{R}^d} \mathcal{L}(\mathbf{A}, \mathbf{x}, \mathbf{b})$$

It remains to determine what $\mathcal{L}$ is. Intuitively, $\mathcal{L}$ should certainly involve the error $\mathbf{A}\mathbf{x} - \mathbf{b}$. We could pick

$$\mathcal{L}(\mathbf{A}, \mathbf{x}, \mathbf{b}) = \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_0$$

where $\|u\|_0 = |\text{supp}(u)|$ is the number of non-zero entries in a vector. Intuitively, this is trying to find the "simplest" solution, as a proxy for generalization (along the lines of Occam's razor). However, this is a combinatorially hard problem, and thus perhaps not a great idea. What about

$$\mathcal{L}(\mathbf{A}, \mathbf{x}, \mathbf{b}) = \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_1$$

Well, minimizing the 1-norm is at some level a proxy for minimizing the 0-norm so it's a good idea in that sense. However, it is not differentiable everywhere. Differentiability is in general a good property to have, as it makes the problem amenable to various optimization techniques. What about the 2-norm?

$$\mathcal{L}(\mathbf{A}, \mathbf{x}, \mathbf{b}) = \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2$$

You might have noticed I sneaked in a square in there. As it turns out, the 2-norm is somehow the Goldilocks zone of norms (see Scott Aaronson's lecture notes on Quantum Information Science [Aaronson, 2018] for a similar observation in that realm). This loss function is twice-differentiable everywhere, and *convex* (see section 8). As we will see soon, this means our problem has a unique solution, and moreover has a nice closed form.

9.1.2 Solving Least Squares

The problem we want to solve can now be written out as

$$\min_x \|\mathbf{A}\mathbf{x}-\mathbf{b}\|_2^2 = \min_x \sum_{i=1}^n \left(\langle \mathbf{A}_{i,\bullet}, \mathbf{x}\rangle - b_i\right)^2 = \min_{x_1,\ldots,x_d} \sum_{i=1}^n \left(\sum_{j=1}^d \mathbf{A}_{ij}\mathbf{x}_j - b_i\right)^2$$

We let our loss function be $\mathcal{L}(x) = \frac{1}{2}\|\mathbf{A}\mathbf{x}-\mathbf{b}\|_2^2$. Then, to minimize we solve for the stationary points of $\mathcal{L}$. However, since $\mathcal{L}$ is convex, this will also be the global minimizer!

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \mathbf{x}_j} = \frac{\partial}{\partial \mathbf{x}_j}\frac{1}{2}\sum_{i=1}^n \left(\langle \mathbf{A}_{i,\bullet}, \mathbf{x}\rangle - b_i\right)^2 &= \sum_{i=1}^n \left(\langle \mathbf{A}_{i,\bullet}, \mathbf{x}\rangle - b_i\right)\frac{\partial}{\partial \mathbf{x}_j}\left(\langle \mathbf{A}_{i,\bullet}, \mathbf{x}\rangle - b_i\right)\\ &= \sum_{i=1}^n \mathbf{A}_{ij}\left(\langle \mathbf{A}_{i,\bullet}, \mathbf{x}\rangle - b_i\right) = \left\langle \left(\mathbf{A}_{1j},\ldots,\mathbf{A}_{nj}\right), \mathbf{A}\mathbf{x}-\mathbf{b}\right\rangle\\ &= \langle \mathbf{A}_{\bullet,j}, \mathbf{A}\mathbf{x}-\mathbf{b}\rangle\end{aligned}$$

Thus, we have that the gradient $\nabla\mathcal{L}$ satisfies

$$\nabla\mathcal{L} := \left(\frac{\partial \mathcal{L}}{\partial \mathbf{x}_j}\right)_{j=1}^d = \mathbf{A}^\top(\mathbf{A}\mathbf{x}-\mathbf{b}) \tag{1}$$

Thus, solving for the stationary points,

$$\begin{aligned}&\nabla\mathcal{L} = 0\\ \iff \mathbf{A}^\top(\mathbf{A}\mathbf{x}-\mathbf{b}) = 0 &\iff \mathbf{A}^\top\mathbf{A}\mathbf{x} = \mathbf{A}^\top\mathbf{b}\\ &\iff \boxed{\mathbf{x} = (\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top\mathbf{b}}\end{aligned}$$

The term $(\mathbf{A}\mathbf{A}^\top)^{-1}\mathbf{A}^\top$ is also known as the *Moore-Penrose psuedoinverse.*

Definition 9.1 (Moore-Penrose psuedoinverse)**.** *For a full-rank overdetermined matrix $\mathbf{A} \in \mathbb{R}^{n\times d}$, the matrix $\mathbf{A}^\dagger := (\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top \in \mathbb{R}^{d\times n}$ is call the (Moore-Penrose) psuedoinverse of $\mathbf{A}$.*

1. *$\mathbf{A}^\dagger$ is a* left *inverse of $\mathbf{A}$. That is, $\mathbf{A}^\dagger\mathbf{A} = \mathbf{I}_{d\times d}$.*
2. *The psuedoinverse $x^* = \mathbf{A}^\dagger$ is the solution to the linear least squares problem.*

Properties of the Psuedoinverse The psuedoinverse also has the following interesting properties:

Property 9.1. *$\mathbf{A}\mathbf{A}^\dagger \in \mathbb{R}^{n\times n}$ is an* orthogonal projection matrix *onto the subspace spanned by the columns of $\mathbf{A}$. Similarly $(\mathbf{A}\mathbf{A}^\dagger - \mathbf{I})$ is an orthogonal projection onto the complementary space.*

Property 9.2. *The vector $\mathbf{A}\mathbf{x}^* = \mathbf{A}\mathbf{A}^\dagger\mathbf{b} \in \mathbb{R}^n$ minimizing the least squares loss is the projection of $\mathbf{b}$ onto the subspace $\mathbf{A}\mathbf{A}^\dagger$ spanned by the columns of $\mathbf{A}$.*

9.1.3 Least Squares Example

A simple instance of least squares is finding the best linear fit to a data set. i.e., find the best line $mx + b$ minimizing the sum of squared *vertical* offsets

$$\min_{m,b} \sum_{i=1}^n \left(mx_i + b - y_i\right)^2$$

In matrix form,

$$\min_{m,b} \|\mathbf{A} \begin{pmatrix} m & b \end{pmatrix}^\top - \mathbf{y}\|_2^2$$

where

$$\mathbf{A} = \begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \\ x_n & 1 \end{pmatrix}$$

The least squares solution can then be calculated as follows:

$$\begin{pmatrix} m^* \\ b^* \end{pmatrix} = (\mathbf{A}^\top \mathbf{A})^{-1} \mathbf{A}^\top \mathbf{y}$$

$$m^* = \frac{\sum_{i=1}^n x_i y_i}{\sum_{i=1}^n x_i^2} \quad b^* = \frac{1}{n} \sum_{i=1}^n y_i$$

9.1.4 Efficiency of Solving Least Squares

Solving the least squares problem using the above psuedoinverse method is quite inefficient, since just to compute a single vector $\mathbf{x}^*$ we need to invert a large matrix which can be a very expensive operation. While it is true that as we saw above the unique solution $\mathbf{x}^*$ can be obtained by solving the linear system

$$\mathbf{A}^\top \mathbf{A} \mathbf{x}^* = \mathbf{A}^\top \mathbf{b}$$

with highly specialized linear solvers, we are more interested in general iterative solvers that extend beyond simple linear models. This is where **gradient descent** comes in.

9.1.5 Solving Least Squares with Gradient Descent

Gradient Descent is an extremely general, and simple first-order optimization method. The key intuition/insight is that in order to minimize a function, we keep moving in the direction of the negative gradient. The reader may recall that this is the direction of steepest decrease.

Algorithm 1 Gradient Descent Psuedocode

```
procedure GradientDescent(f, η)
    Initialize x_0 ∈_R ℝ^d
    while not converged do
        x_{t+1} = x_t − η∇f(x_t)
    return x_{t+1}
```

Now, specifically for the case of linear least squares, we recall Equation 1

$$\nabla \mathcal{L} = \mathbf{A}^\top (\mathbf{A}\mathbf{x} - \mathbf{b})$$

Thus, the least squares gradient descent update becomes

9.1.6 Convergence of Gradient Descent

Although we have this seemingly wonderful iterative algorithm, it is imperative that we also establish bounds on how long it will take for it to converge given some convergence criterion. If, for instance, it takes $\mathcal{O}(2^n)$ rounds, that's kind of useless. Well, thankfully as it turns out GD isn't useless, and in fact we can prove the following convergence guarantee:

Algorithm 2 Linear Least Squares Gradient Descent

procedure LINEARLEASTSQUARESGD($\mathbf{A}, \mathbf{b}, \eta$)
  Initialize $x_0 \in_R \mathbb{R}^d$
  while not converged **do**
    $x_{t+1} = x_t - \eta \mathbf{A}^\top (\mathbf{A}\mathbf{x} - \mathbf{b})$
  return x_{t+1}

Theorem 9.1. *For the linear least squares problem, the gradient descent algorithm can meet the following convergence criterion in $\mathcal{O}(\log \epsilon)$ iterations, for any $\epsilon > 0$:*

$$\|\mathbf{x}_{t+1} - \mathbf{x}^*\|_2 \leq \epsilon \cdot \|\mathbf{x}_0 - \mathbf{x}^*\|_2$$

Proof. We will proceed by first bounding $\|\mathbf{x}_{t+1} - \mathbf{x}^*\|$, and then proving our choice of η implies a convergence with the desired rate. Observe

$$\begin{aligned}
\mathbf{x}_{t+1} - \mathbf{x}^* &= \left(\mathbf{x}_t - \eta \mathbf{A}^\top (\mathbf{A}\mathbf{x}_t - \mathbf{b})\right) - \mathbf{x}^* && \text{(GD update)} \\
&= \left(\mathbf{x}_t - \eta \mathbf{A}^\top (\mathbf{A}\mathbf{x}_t - \mathbf{A}\mathbf{x}^*)\right) - \mathbf{x}^* && (\mathbf{A}\mathbf{x}^* = \mathbf{b} \text{ by assumption}) \\
&= \mathbf{x}_t - \mathbf{x}^* - \eta \mathbf{A}^\top \mathbf{A} (\mathbf{x}_t - \mathbf{x}^*) \\
&= (\mathbf{I} - \eta \mathbf{A}^\top \mathbf{A})(\mathbf{x}_t - \mathbf{x}^*)
\end{aligned}$$

We will now use the following property of the Frobenius norm

Property 9.3. *The Frobenius norm $\|\cdot\|_2$ on matrices is submultiplicative. i.e.,*

$$\|\mathbf{A}\mathbf{B}\|_2 \leq \|\mathbf{A}\|_2 \|\mathbf{B}\|_2$$

Thus, we now obtain the following bound

$$\|\mathbf{x}_{t+1} - \mathbf{x}^*\|_2 \leq \|\mathbf{I} - \eta \mathbf{A}^\top \mathbf{A}\|_2 \cdot \|\mathbf{x}_t - \mathbf{x}^*\|_2$$

In order for convergence, we need this to be a decreasing series. In particular, we need $\|\mathbf{I} - \eta \mathbf{A}\mathbf{A}^\top\|_2 < 1$. We now attempt to find η so that this is satisfied.

We first make the following observation

Lemma 9.1.1. *For any matrix $\mathbf{A} \in \mathbb{R}^{n \times d}$, $\|\mathbf{A}\|_2 \leq \sqrt{\min(n, d)} \cdot \lambda_{\max}(\mathbf{A})$.*

Proof.

$$\|\mathbf{A}\|_2 = \sqrt{\sum_{i=1}^{\min(n,d)} \sigma_i^2} \leq \sqrt{\min(n, d) \cdot \sigma_1^2} = \sqrt{\min(n, d)} \cdot \sigma_1$$

□

Thus, we will now bound the largest eigenvalue of $(\mathbf{I} - \eta \mathbf{A}^\top \mathbf{A})$ and then appropriately pick η so we get a contraction. In particular, suppose $\mathbf{A}^\top \mathbf{A}$ has eigendecomposition $\mathbf{A}^\top \mathbf{A} = \mathbf{U} \Lambda \mathbf{U}^\top$. Then, $\mathbf{I} - \eta \mathbf{A}^\top \mathbf{A}$ has eigendecomposition $\mathbf{U}(\mathbf{I} - \eta \Lambda)\mathbf{U}^\top$. In particular the eigenvalues satisfy

$$1 - \eta \lambda_{\max} \leq \cdots \leq 1 - \eta \lambda_{\min}$$

Where $\lambda_{\min}, \lambda_{\max}$ are the smallest, and largest eigenvalues of $\mathbf{A}^\top \mathbf{A}$ respectively. Thus, since $n > d$, we have that

$$\|\mathbf{I} - \eta \mathbf{A}^\top \mathbf{A}\|_2 \leq \sqrt{d}(1 - \eta \lambda_{\min})$$

. In order for this to be a contraction, we need

$$\begin{aligned} &\sqrt{d}(1-\eta\lambda_{\min}) < 1 \\ \iff &\eta > \frac{1}{\lambda_{\min}}\left(1-\frac{1}{\sqrt{d}}\right) \end{aligned}$$

Thus, in particular $\eta = \frac{1}{\lambda_{\min}}\left(1-\frac{1}{\alpha\sqrt{d}}\right)$ for some $\alpha > 1$ satisfies our requirement. However, we can get a tighter bound if we have the following:

$$\begin{aligned} &\left(1-\frac{1}{\sqrt{d}}\right) \leq \frac{\lambda_{\min}}{\lambda_{\max}} \\ \iff &\kappa(\mathbf{A}^\top\mathbf{A}) \leq \left(1-\frac{1}{\sqrt{d}}\right)^{-1} \end{aligned}$$

If this condition is satisfied, we can get away with $\eta = 1/\lambda_{\max}$. In either case, we get the following:

$$\begin{aligned} \|\mathbf{x}_{t+1}-\mathbf{x}^*\|_2 &\leq (1-\eta\lambda_{\min})\cdot\sqrt{d}\cdot\|\mathbf{x}_t-\mathbf{x}^*\|_2 \\ &\leq \left((1-\eta\lambda_{\min})\cdot\sqrt{d}\right)^t\cdot\|\mathbf{x}_0-\mathbf{x}^*\|_2 \end{aligned}$$

Thus we can meet the following convergence criterion for any $\epsilon > 0$ within $\mathcal{O}(\log\epsilon)$ iterations. The optimal rate is obtained when the aforementioned condition is satisfied, and we may pick $\eta = 1/\lambda_{\max}$.

$$\|\mathbf{x}_{t+1}-\mathbf{x}^*\|_2 \leq \epsilon\cdot\|\mathbf{x}_0-\mathbf{x}^*\|_2$$

□

9.2 Convergence of GD for Smooth Functions

In Machine Learning in general, while model families come in different shapes and sizes – from simple best-fit feature bases, to SVMs, to deep neural networks – it is still a common occurence that the loss function $\mathcal{L}$ has a special property: smoothness.

Definition 9.2. *A differnetiable function f is called **smooth** iff it has a Lipschitz continuous gradient. i.e., there exists L such that*

$$\|\nabla f(y)-\nabla f(x)\|_2 \leq L\|y-x\|_2$$

Thus, herein we shall equivalently assume that $\nabla\mathcal{L}$ is L-Lipschitz for some L. To study the convergence of GD, we will attempt to bound $\|\nabla\mathcal{L}(w_k)\|^2$. In particular, we will show the following:

Theorem 9.2. *Fix $\eta = 1/L$. Then, we will achieve stopping criterion $\|\nabla\mathcal{L}(w_t)\|^2 \leq \epsilon$ in at most*

$$t = \frac{2L[\mathcal{L}(w_0)-\mathcal{L}^*]}{\epsilon}$$

where $\mathcal{L}$ is assumed to be smooth, and L is the Lipschitz constant for $\nabla\mathcal{L}$.

In order to prove things about $\|\nabla\mathcal{L}\|^2$, it's a good idea to first get a bound for $\mathcal{L}(w)$ in terms of ∇L. We will utilize the following variant of the multivariate Taylor expansion:

Theorem 9.3. *For a C^2 function*

$$f(y) \leq f(x) + \nabla f(x)^\top(y-x) + \frac{1}{2}(y-x)^\top\nabla^2 f(x^*)(y-x)$$

However, we also want to utilize our smoothness assumption on $\mathcal{L}$. We will make use of the following lemma:

Lemma 9.3.1. *For C^2 functions, Lipschitz continuity of the gradient is equivalent to*

$$\nabla^2 f(w) \preccurlyeq L \cdot \mathbf{I}_{d\times d}$$

where $A \preccurlyeq B$ iff $B - A$ is positive semidefinite. Equivalently, we have $v^\top \nabla^2 f(w) v \leq L\|v\|_2^2 \, \forall v, w$

We will now apply Lemma 9.3.1 to Theorem 9.3 to obtain the following "Descent Lemma":

Lemma 9.3.2 (Descent Lemma)**.** *For smooth $\mathcal{L}$, it is the case that*

$$\mathcal{L}(y) \leq \mathcal{L}(x) + \nabla\mathcal{L}(x)^\top (y - x) + \frac{L}{2}\|y - x\|^2$$

The Descent Lemma also holds for C^1 functions. This lemma gives us a quadratic upper bound function for f, touching at $f(x)$. To get a sense of what kind of learning rate/step size we should use, it might be a good idea to look at the minimizer for this upper bound. A simple calculation shows that the minimizer is

$$y - x - \frac{1}{L}\nabla\mathcal{L}(x)$$

i.e., one gradient step away from x, with learning rate $\frac{1}{L}$. Ok, so now we will pick a constant step-size $\alpha = 1/L$ as our starting point. Using this, we will be able to prove the following theorem:

Theorem 9.4. *Suppose that $\mathcal{L}$ is smooth, and we pick a step-size $\alpha = \frac{1}{L}$. Then, the following holds:*

$$\min_{k\in[t]} \|\nabla\mathcal{L}(w_k)\|^2 \leq \frac{2[\mathcal{L}(w_0) - \mathcal{L}^*]}{t}$$

Proof. Suppose we pick $\alpha = 1/L$. Then, our gradient update is the following:

$$w_{t+1} = w_t - \frac{1}{L}\nabla\mathcal{L}(w_t)$$

Substituting this into the "Descent Lemma" from 9.3.1 we get

$$\begin{aligned}
\mathcal{L}(w_{t+1}) &\leq \mathcal{L}(w_t) + \nabla\mathcal{L}(w_t)^\top \left(-\frac{1}{L}\cdot\nabla\mathcal{L}(w_t)\right) + \frac{L}{2}\|\frac{1}{L}\nabla\mathcal{L}(w_t)\|^2 \\
&= \mathcal{L}(w_t) - \frac{1}{L}\|\nabla\mathcal{L}(w_t)\|^2 + \frac{1}{2L}\|\nabla\mathcal{L}(w_t)\|^2 \\
&= \mathcal{L}(w_t) - \frac{1}{2L}\|\nabla\mathcal{L}(w_t)\|^2 \\
\mathcal{L}(w_{t+1}) &\leq \mathcal{L}(w_t) - \frac{1}{2L}\|\nabla\mathcal{L}(w_t)\|^2
\end{aligned}$$

We may rearrange this last result to get

$$\|\nabla\mathcal{L}(w_t)\|^2 \leq 2L\left[\mathcal{L}(w_t) - \mathcal{L}(w_{t+1})\right]$$

Finally, we sum up the squared norms of these gradients to get

$$\sum_{k=1}^{t}\|\nabla L(w_t)\|^2 \leq 2L\sum_{k=1}^{t}\left[\mathcal{L}(w_k) - \mathcal{L}(w_{k+1})\right] \leq 2L\left[\mathcal{L}(w_0) - \mathcal{L}(w_{t+1})\right] \leq 2L\left[\mathcal{L}(w_0) - \mathcal{L}^*\right]$$

where $\mathcal{L}^*$ is the minimum value of $\mathcal{L}$. Thus, we derive our final result

$$\min_{k\in[t]} \|\nabla\mathcal{L}(w_k)\|^2 \leq \frac{2[\mathcal{L}(w_0) - \mathcal{L}^*]}{t}$$

Picking α in practice It's not really a good idea to compute L in order to use $\alpha = 1/L$ in practice, since computing L is usually very expensive to compute. Moreover, this is a very small step-size as it is a **worst**-case bound on curvature. Generally, it is preferred to use line-search techniques.

For instance, we may decrease α by binary search until the following *Armijo condition* is met:

$$f(w_t - \alpha \nabla f(w_t)) \leq f(w_t) - \alpha\gamma \|\nabla f(w_t)\|^2 \quad \gamma \in (0, 1/2]$$

□

9.2.1 Concluding Notes

The cost of computing a gradient scales with the number of variables d. Thus, if we are working with n training samples and computing a gradient, by linearity of the gradient it'll take us $\mathcal{O}(nd)$ time. Above, we showed that for smooth functions GD has a guaranteed $O(1/t)$ convergence rate. i.e., if we want to converge to an ϵ-approximation of a stationary point, $O(1/\epsilon)$ iterations suffice. But then, including gradient computations that's at least $\mathcal{O}(nd/\epsilon)$ time (huge!). This, however, (thankfully?) is at least in part an artifact of this being a very loose bound. Gradient descent is **much** faster in practice. However, this does not mean GD is finding **global** minimizers efficiently – NP-hard functions still exist.

In order to find global optimums of non-convex smooth functions, the fastest possible algorithms require $\mathcal{O}(1/\epsilon^d)$ iterations in the worst case. Smoothness is too "local" or "weak" of a model to overcome the curse of dimensionality.

> *"The biggest open question in neural network theory over the past 15 years: how is it possible that gradient descent tends to converge to global minimizers of (highly nonlinear, highly nonconvex) neural network loss functions in practice, as if such optimization problems were not much more complex than linear regression? What is the underlying structure of neural networks as implemented in practice that allows us to overcome the curse of dimensionality?" – R. Ward*

Convex functions, however, are able to overcome the curse. In particular, $\mathcal{O}(1/\epsilon)$ iterations of GD will do the trick of bringing us ϵ-close to a **global** optimum, since with a convex function a stationary point is a global minimizer. There are even algorithms which can achieve $\mathcal{O}(1/\sqrt{\epsilon})$.

Nesterov's accelerated gradient method This method achieves the aforementioned $\mathcal{O}(1/\sqrt{\epsilon}$ bound. Instead of a simple constant-sized step, this method uses the previous gradient as well. In particular, it makes the following update:

$$w_{t+1} = v_t - \alpha \nabla f(v_t)$$
$$v_{t+1} = w_t + \beta_t (w_{t+1} - w_t)$$

This method with $\alpha_t = 1/L$ and $\beta_t = \frac{t-1}{t+2}$ achieves an error of $\mathcal{O}(1/t^2)$ after t iterations.

9.3 Convergence of GD for Convex Functions, Polyak-Lojasiewic Inequality

While smoothness alone may not have been enough to overcome the curse of high dimensionality, all hope is not lost. Notice that for some "nice" functions, for instance overdetermined linear regression, gradient descent has a much faster convergence rate. In particular, it converges **expoenentially** fast. A general property underlying many such "nice" functions is the *Polyak-Lojasiewic (PL) Inequality.*

Definition 9.3 (Polyak-Lojasiewic Inequality)**.** *A function $f : \mathbb{R}^d \to \mathbb{R}$ satisfies the PL inequality iff*

$$\frac{1}{2}\|\nabla f(w)\|^2 \geq \mu (f(w) - f^*)$$

Example. *For the least squares regression problem, $\mu = \lambda_{\min}(X^\top X)$, and*

$$\|X^\top (Xw - y)\|^2 \geq \lambda_{\min}(X^\top X)\|Xw - y\|^2$$

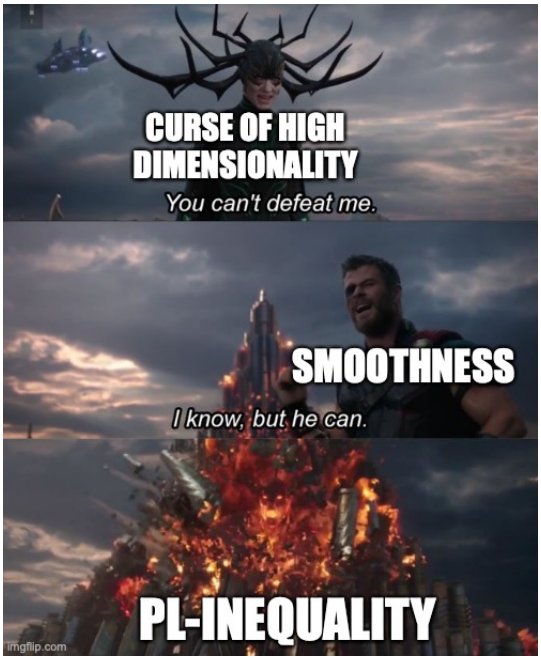

Figure 1: A quality meme

9.3.1 GD with Smooth Functions satisfying PL-inequality

Now, using this PL-inequality, we will show that we can establish the following convergence rate:

Theorem 9.5. *For a smooth function $f : \mathbb{R}^d \to \mathbb{R}$ which also satisfies the PL-inequality, for any $\epsilon > 0$ the following convergence criterion can be met in $\mathcal{O}(\log(1/\epsilon))$ iterations:*

$$f(w_t) - f^* \leq \epsilon$$

Proof. Assuming smoothness of our function we were able to derive the following bound in the proof of Theorem 9.4

$$f\left(w_{t+1}\right) \leq f\left(w_t\right) - \frac{1}{2L}\|\nabla f\left(w_t\right)\|^2$$

Under the PL-inequality (9.3) we have

$$-\|\nabla f(w_t)\|^2 \leq -2\mu\left(f(w_t) - f^*\right)$$

and thus,

$$\begin{aligned} f(w_{t+1}) &\leq f(w_t) - \frac{\mu}{L}\left(f(w_t) - f^*\right) \\ f(w_{t+1}) - f^* &\leq \left(f(w_t) - f^*\right) - \frac{\mu}{L}\left(f(w_t) - f^*\right) \\ f(w_{t+1}) - f^* \leq \left(1 - \frac{\mu}{L}\right)\left(f(w_t) - f^*\right)) &\leq \left(1 - \frac{\mu}{L}\right)^t \left(f(w_0) - f^*\right) \end{aligned}$$

Finally, observe that

$$1 + x \leq e^x$$

Thus

$$f(w_{t+1}) - f^* \leq \exp\left(-\frac{\mu}{L} \cdot t\right) [f(w_0) - f^*]$$

Hence, we have $f(w_t) - f^* \leq \epsilon$ for any t satisfying

$$t \geq \frac{L}{\mu} \log\left(f(w_0) - f^*)/\epsilon\right) = \mathcal{O}(\log(1/\epsilon))$$

□

9.3.2 Properties of the PL-inequality

The PL-inequality is general enough that it is satisfied by many standard convex models. However, it has the additional benefit of being satisfied even by some non-convex functions like $w^2 + 3\sin^2(w)$. It is also satisfied for many classes of *overparameterized* neural networks [Liu et al., 2020], and is the reason for good empirical performance of GD in optimizing neural networks.

9.3.3 Strong Convexity and PL-inequality

Strong convexity, as the name may imply, is a stronger variant of convexity which induces some nice properties.

Definition 9.4 (Strong Convexity)**.** *A function is μ-strongly convex if the following function is convex:*

$$f(w) - \frac{mu}{2}\|w\|^2$$

If f is C^2, then equivalently

$$\nabla^2 f(w) \succcurlyeq \mu I$$

Property 9.4. *If $f : \mathbb{R}^d \to \mathbb{R}$ is a μ-strongly convex function, then*

1. *A* unique *minimizer exists*
2. *if f is additionally C^1, then it satisfies the PL inequality.*

If we have a convex loss function f, then adding ℓ_2-regularization makes it strongly convex. This can improve the convergence rate from $\mathcal{O}(1/t)$ to exponential. This motivates the weight-decay regularization technique used for neural networks.

9.4 Stochastic Gradient Descent

Until now, we have been talking about gradient descent while somewhat closing our eyes towards the reality of computational cost. Unfortunately, for large datasets computing the gradient across all of our training samples is quite intractable. The cost of each gradient descent update scales linearly with both (1) the number of variables, and (2) the number of training samples. Thus, in order to make the problem tractable we instead consider a randomized version of GD, *Stochastic Gradient Descent* (SGD).

SGD performs pretty much the same update as vanilla gradient descent with the modification that instead of computing the gradient across all training samples, we will simply compute the gradient with respect to a *randomly* chosen sample. In practice, the random choice will be without replacement ,however, to simplify mathematical analysis we will always assume it is with replacement.

Algorithm 3 Stochastic Gradient Descent Psuedocode

procedure StochasticGradientDescent(f, η)
  Initialize $x_0 \in_R \mathbb{R}^d$
  while not converged **do**
    pick $j_t \in_R [n]$
    $x_{t+1} = x_t - \eta \nabla f_{j_t}(x_t)$
  return x_{t+1}

The reader may constrast Algorithm 1 with Algorithm 3. Note that here we assume the loss function f is of the form

$$f(w_t) = \frac{1}{n} \sum_{i=1}^{n} f_i(w)$$

Where f_i is the loss with respect to the i^{th} training example. Observe with SGD we shave off a factor of n in the time spent on each gradient update. Moreover, the SGD vector also enjoys the following nice property:

Property 9.5. *The stocastic gradient vector is an* unbiased *estimate of the full gradient. i.e.,*

$$\mathbb{E}\left[\nabla f_{j_t}(w)\right] = \frac{1}{n} \sum_{j=1}^{n} \nabla f_j(w) = \nabla f(w)$$

When implementing SGD, x_0 is generally sampled as an i.i.d Gaussian vector. While SGD sounds great right now, it does have (at least) two challenges:

1. It takes n iterations before the model has "seen" all the training data.
2. The individual updates may be more noisy.

Thus, as with many things in life, a solution which is often used in practice is **Mini-Batching**. In mini-batching, instead of just selecting 1 random component of the loss function, we pick a batch size B, and pick a random batch of B components, and compute the average gradient with respect to the batch. Thus, the update becomes

Algorithm 4 Mini-Batch Stochastic Gradient Descent Psuedocode

procedure MinibatchStochasticGradientDescent(f, η, B)
  Initialize $x_0 \in_R \mathbb{R}^d$
  while not converged **do**
    pick $S \subset [n]$ where $|S| = B$
    $x_{t+1} = x_t - \eta \frac{1}{B} \sum_{j \in S} \nabla f_j(x_t)$
  return x_{t+1}

Observe that by linearity of expectation this is still an unbiased estimate of the full gradient vector. However, we have now reduced the variance. In particular, observe

$$\text{Var}\left(\frac{1}{B} \sum_{j \in S} \nabla f_j(x_t)\right) = \sum_{j \in S} \text{Var}\left(\frac{1}{B} \nabla f_j(x_t)\right) = B \cdot \frac{1}{B^2} \text{Var}\left(f_1(x_t)\right) = \frac{1}{B} \text{Var}\left(f_1(x_t)\right)$$

It is advisable to pick a reasonable batch-size based on available hardware. For instance, it may be a good idea to make sure your batch fits in your GPU memory, or perhaps isn't too large with respect to the bandwidth/throughput of the memory bus between CPU and GPU memory etc. Ok, that's all great, but what about convergence? With minibatching we seem to have gotten something which is on expectation the full gradient, and with *lower* variance than just SGD, but does that mean the same guarantees from prior sections carry over? Not quite.

Theorem 9.6. *Suppose we are optimizing a loss function* $f(w) = \frac{1}{n}\sum_{j=1}^{n} f_j(w)$ *where the following are satisfied:*

1. *Each* f_j *is L-Lipschitz smooth (*$\|\nabla f_j(u) - \nabla f_j(v)\|_2 \leq L\|u - v\|_2$*)*
2. *Each* f_j *is convex*
3. *The total loss* f *is* μ*-strongly convex*
4. *At the minimizing solution* w_* *the "component noise" is* $\sigma^2 = \frac{1}{n}\sum_{j=1}^{n}\|\nabla f_j(w_*)\|^2$

Then, assuming step-size $\alpha \leq \frac{1}{L}$ *we have*

$$\mathbb{E}\left[\|w_t - w_*\|_2^2\right] \leq \left(1 - 2\alpha\mu\left(1 - \alpha L\right)^t\right)\|w_0 - w_*\|_2^2 + \alpha\frac{\sigma^2}{\mu\left(1 - \alpha L\right)}$$

10 Kernel Methods

10.1 Linear Classification for Binary Data

Suppose we are given training data $\{(x_i, y_i)\}_{i\in[n]}$ where $x_j \in \mathbb{R}^d$ and $y_j \in \{+1, -1\}$. Suppose further that the data is linearly separable.

Definition 10.1 (Linearly Separable Data)**.** *A dataset is said to be linearly separable if there exists an* **affine** *map*

$$\langle \mathbf{w}, \mathbf{x} \rangle - b$$

which separates the data. i.e.,

$$\hat{y_i} = \text{sign}\left(\langle \mathbf{w}, \mathbf{x} \rangle - b\right) = y_j \quad \forall j$$

Recall that $\mathbf{w}$ is a vector which is normal to the plane. $\mathbf{b}$ is roughly the intercept indicating how much the hyperplane is shifted in the direction of the normal $\mathbf{w}$. For instance, $\langle e_1, \mathbf{x} \rangle = b$ in $\mathbb{R}^3$ is simply the $y-z$ plane shifted b in the $\mathbf{x}$-direction. The goal is to have all data points in one class to be on one side, and the others on the other. For linearly separable data, a "best" lienar classifier is often the one which *maximizes the margin.* i.e., maximizes the minimial distance between any point $\mathbf{x}_j$ and the hyperplane $\langle \mathbf{w}, \mathbf{x} \rangle - b$. Chosing the max-margin classifier is a good heuristic for generalization.

For linearly separable data, there exist two parallel hyperplanes

$$\begin{aligned} \langle \mathbf{w}, \mathbf{x} \rangle - b &= 1 \\ \langle \mathbf{w}, \mathbf{x} \rangle - b &= -1 \end{aligned}$$

such that

$$\begin{aligned} \langle \mathbf{w}, \mathbf{x} \rangle_j - b &\geq 1 \quad y_j = 1 \\ \langle \mathbf{w}, \mathbf{x}_j \rangle - b &\leq -1 \quad y_j = -1 \end{aligned}$$

Thus, we want to maximize the distance between these two parallel hyperplanes subject to these constraints. We define the distance from a point u to the hyperplane $H = \{\mathbf{x} : \langle \mathbf{w}, \mathbf{x} \rangle = b\}$ as

$$d(\mathbf{u}, H) = \min_x \|\mathbf{u} - \mathbf{x}\|_2 \quad \text{s.t. } \langle \mathbf{w}, \mathbf{x} \rangle = b$$

We will minimize this by the method of Lagrange multipliers. In particular, we define

$$\begin{aligned} d'(x; u) = \frac{1}{2}\|x - u\|_2^2 \quad \nabla d'(x; u) &= \mathbf{x} - \mathbf{u} \\ \nabla \langle \mathbf{w}, \mathbf{x} \rangle &= \mathbf{w} \end{aligned}$$

Thus we must simply solve

$$\begin{aligned} \mathbf{x} - \mathbf{u} - \lambda \mathbf{w} &= 0 \\ \langle \mathbf{w}, \mathbf{x} \rangle &= b \end{aligned}$$

Setting $x = \mathbf{u} + \lambda \mathbf{w}$ and substituting into the second equation we get

$$\begin{aligned} \langle \mathbf{w}, \mathbf{u} + \lambda \mathbf{w} \rangle = \langle \mathbf{w}, \mathbf{u} \rangle + \lambda \|w\|_2^2 &= b \\ \implies \lambda &= \frac{b - \langle \mathbf{w}, \mathbf{u} \rangle}{\|\mathbf{w}\|_2^2} \end{aligned}$$

Finally, substituting back we obtain

$$\mathbf{x} = \mathbf{u} + \frac{b - \langle \mathbf{w}, \mathbf{u} \rangle}{\|\mathbf{w}\|_2} \hat{w} \quad d(u, H) = \frac{b - \langle \mathbf{w}, \mathbf{u} \rangle}{\|\mathbf{w}\|_2}$$

Picking $\mathbf{u} = \left(0, 0, \ldots, 0, \frac{1+b}{\mathbf{w}_d}\right)$ on the line $\langle \mathbf{w}, \mathbf{x} \rangle - b = 1$ we get $d(\mathbf{u}, H) = \frac{1}{\|\mathbf{w}\|_2}$ Thus, the distance between these two parallel hyperplanes is $\frac{2}{\|\mathbf{w}\|_2}$. Maximizing this quantity is equivalent to minimizing $\|\mathbf{w}\|_2^2$, which is a nice convex optimization problem.

10.1.1 Support Vector Machine (SVM) Classification

If we look at the above constraints, and multiply both sides by y_i, then we may observe that we get the following optimization problem:

$$\min_{\mathbf{w}, b} \|\mathbf{w}\|_2^2$$
$$\text{s.t.} \quad y_j \cdot (\langle \mathbf{w}, \mathbf{x}_j \rangle - b) \geq 1$$

This is the Support Vector Machine (SVM) classification problem, developed at AT&T Bell Labs by Vapnik et al. [Boser et al., 1992] in the 1990s. They became famous at the time for beating neural networks with hand-designed features on handwriting recognition on the MNIST dataset.

Property 10.1. *Unlike neural networks, SVM is highly robust to outlier data points.*

Optimizing an SVM is really solving a constrained *Quadratic Program* (QP)

Definition 10.2. *A quadratic program optimizing over some parameter* $\mathbf{z}$ *is defined as*

$$\max_{\mathbf{z}} \frac{1}{2} \mathbf{z}^\top \mathbf{A} \mathbf{z} - \mathbf{d}^\top \mathbf{z}$$
$$s.t. \quad \mathbf{B}\mathbf{z} \leq e$$

Now, the quadratic program stated above is in "primal" form. However, similar to linear programming, in quadratic programming it is easier to solve the "dual" program. In particular, for SVM the dual program is

$$\max_{\alpha \in \mathbb{R}^n} \left(\sum_{i=1}^{n} \alpha_i \right) - \frac{1}{2} \sum_{(i,j) \in [n] \times [n]} (y_i y_j) \cdot (\alpha_i \alpha_j) \cdot \langle \mathbf{x}_i, \mathbf{x}_j \rangle$$
$$\text{s.t.} \quad \alpha_i \geq 0 \text{ and } \sum_{i=1}^{n} y_i \alpha_i = 0$$

If $\alpha \in \mathbb{R}^n$ is the solution to this program, then the solution to the primal program $\mathbf{w}$ can be written as

$$\mathbf{w} = \sum (\alpha_i y_i) \cdot \mathbf{x}_i$$

Important to note that $\alpha_i > 0$ when $\mathbf{x}_i$ lies on the boundary, or when it's a *support vector*.

10.2 Kernel Trick for Non-Linearly Separable Data

Often in real-world problems, data is structured, and separable, but perhaps not necessarily linearly separable. When this situation arises, of course we cannot directly apply something like SVM to it. However, we may apply a *kernel trick* by *lifting* our data to a higher dimension wherein it does exhibit linear separability and apply SVM in the higher dimension. The linear boundary in the higher dimension may correspond to a very non-linear seperator in the normal dimension.

Definition 10.3. *A kernel $K : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ can be any legal definition of an inner product*

$$K(\mathbf{x}, \mathbf{z}) := \langle \varphi(\mathbf{x}), \varphi(\mathbf{z}) \rangle \quad \varphi : \mathbb{R}^d \to \mathbb{R}^D \; D >> d$$

Definition 10.4 (Kernel Trick)**.** *The idea of the kernel trick is the following:*

1. *Rewrite the algorithm (if possible) such that it only uses the data points $\mathbf{x}_i$ in the form of inner products $\langle \mathbf{x}_i, \mathbf{x}_j \rangle$*
2. *Replace all inner products $\langle \mathbf{x}_i, \mathbf{x}_j \rangle$ with kernels $K(\mathbf{x}_i, \mathbf{x}_j)$*

Example (Polynomial Kernel)**.** *An intuitive example of a kernel is the polynomial kernel*

$$K_r(\mathbf{x}, \mathbf{z}) := (\langle \mathbf{x}, \mathbf{z} \rangle)^r$$

For $r = 2, d = 2$

$$\varphi : \mathbb{R}^2 \to \mathbb{R}^3 : (x_1, x_2) \to \varphi(\mathbf{x}) = (x_1^2, x_2^2, \sqrt{2} x_1 x_2)$$

For the polynomial kernel of degree r, $K_r(\mathbf{x}, \mathbf{z})$ lifts into a dimension which grows at least exponentially in r. However, since we are never explicitly computing φ, we just need to compute r multiplications.

While the above Polynomial kernel is a nice toy example, it is not really used in practice. Instead, it is common to use kernels such as the Gaussian kernel which is an inner product in an infinite-dimensional space.

Definition 10.5 (Gaussian Kernel)**.** *The Gaussian Kernel is defined as*

$$K(\mathbf{x}, \mathbf{z}) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{z}\|_2^2}{2\sigma^2} \right)$$

Since the Gaussian kernel operates in a "radial basis", it is able to provide linear separators for almost any data set. However, it often separates way too well and thus overfitting. There is also the tanh kernel, which when used with an SVM corresponds exactly to a 2-layer neural-network.

Definition 10.6 (tanh Kernel)**.** *The* tanh *kernel is defined as:*

$$K(\mathbf{x}, \mathbf{z}) = \tanh\left(\alpha \mathbf{x}^\top \mathbf{z} + c \right)$$

10.2.1 Kernel SVM and other Generalizations

Observe that the dual program for SVM is formulated in terms of just the inner products of the data points, $\langle \mathbf{x}_i, \mathbf{x}_j \rangle$. Thus, we may apply the kernel trick, resulting in the "kernel SVM" program:

Definition 10.7 (Kernel SVM)**.** *The generalization of SVM substituting kernels for inner products is defined by the parameters obtained from the following program*

$$\max_{\alpha \in \mathbb{R}^n} \left(\sum_{i=1}^n \alpha_i \right) - \frac{1}{2} \sum_{(i,j) \in [n] \times [n]} (y_i y_j) \cdot (\alpha_i \alpha_j) \cdot k(\mathbf{x}_i, \mathbf{x}_j)$$

$$s.t. \quad \alpha_i \geq 0 \text{ and } \sum_{i=1}^n y_i \alpha_i = 0$$

As mentioned earlier this can let us find separating hyperplanes for non linearly seperable data by mapping into higher dimensions. However, for certain choices of Kernels the model becomes prone to overfitting. Thus, it is advisable to add a regularization term to prevent this. We may even want to explicitly allow some misclassification as a proxy for generalization. This results in the following general SVM formulation:

Definition 10.8 (Generalized SVM)**.** *The variant of SVM allowing for some classification error for each example, ξ_j, and a regularization parameter λ upon $\mathbf{w}$ is described by the following program:*

$$\min_{\mathbf{w},b,\xi} \ \lambda\|\mathbf{w}\|_2^2 + \frac{1}{n}\sum_{j=1}^{n}\xi_j$$
$$s.t.\ y_i \cdot (\langle \mathbf{w}, \mathbf{x}_j \rangle - b) \geq 1 - \xi_j,\ \xi_j \geq 0,\ \forall j$$

11 Appendix

11.1 Math Glossary for lowly CS majors

Definition 11.1 (Orthogonal Projection Matrix)**.** *A matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ is an orthogonal projection matrix if $\mathbf{A}^2 = \mathbf{A}$ and $\mathbf{A}^\top = \mathbf{A}$.*

Definition 11.2 (Gamma Function)**.** *The Gamma Function $\Gamma(x) : \mathbb{R} \to \mathbb{R}$ is defined as*

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} \mathrm{d}t$$

Importantly, it satisfies $\Gamma(x) = (x-1)!$ for $x \in \mathbb{Z}^+$. In general, it satisfies $\Gamma(x) = x \cdot \Gamma(x-1)$.

Definition 11.3 (Hyperplane)**.** *In d dimensions, a hyperplane is a subspace of dimension $d-1$.*

Definition 11.4 (Half-space)**.** *A half space is either of the two parts that a hyperplane divides a space into.*

Example (half-space)**.** *In 2 dimensions, consider any line in the plane. Then the parts on either side of the line define half-spaces.*

Definition 11.5 (Orthant)**.** *An orthant in d dimensions is defined to be the intersection of d mutually orthogonal half-spaces. The halfspaces induced by the standard bases for $\mathbb{R}^d$ induce 2^d orthants (for $d = 2$, quadrants).*

Example (Orthants)**.** *The upper right quadrant in 2 dimensions is the intersection of the half-spaces $\{(x, y) \in \mathbb{R}^2 \mid x \geq 0\}$ and $\{(x, y) \in \mathbb{R}^2 \mid y \geq 0\}$.*

Definition 11.6 (Smoothness)**.** *A function $f : A \to B$ is said to be smooth if it is infinitely-differentiable everywhere.*

Definition 11.7 (Latent)**.** *A variable which is inferred, but not observed is considered to be a latent variable.*

Example (Latent Variable)**.** *Properties such as time taken to add to cart, time taken to check out cart, hit rate of recommendations etc are all measurable, but "consumer satisfaction" scores are inferred from these, not observed.*

11.2 Classical Interpolation Methods

11.2.1 Linear Interpolation

11.2.2 Polynomial Interpolation

11.2.3 Spine Interpolation

11.3 Lagrange Multipliers

12 References

References

[Aaronson, 2018] Aaronson, S. (2018). Introduction to quantum information science lecture notes.

[Boser et al., 1992] Boser, B. E., Guyon, I. M., and Vapnik, V. N. (1992). A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152.

[Liu et al., 2020] Liu, C., Zhu, L., and Belkin, M. (2020). Toward a theory of optimization for over-parameterized systems of non-linear equations: the lessons of deep learning. *arXiv preprint arXiv:2003.00307*.

[Wikipedia, 2021] Wikipedia (2021). Matrix norm — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Matrix%20norm&oldid=1008374867`. [Online; accessed 11-March-2021].