University of Vermont
CMPE 5990 (Fall 🌬️ 🎃 🌬️ 2025)

Machine Learning for Engineering Sciences

Notes and Homeworks*

β-Version 1.42: October 16, 2025

We all stand on the shoulders of giants: mathematical formulations in this document are borrowed extensively from [1, 2, 3, 4]. This is a "**living document**" which will be updated throughout the semester. **Assignments**: If there is a ✔ next to the assignment in the following list, then all problems have been assigned. If there is an ✗, then the assignment **has not yet been finalized**.

- There are `2 HW0 problems` assigned ✔
  - **Due:** Th, Sept 4, 11PM
- There are `2 HW1 problems` assigned ✔
  - **Due:** Th, Sept 11, 11PM
- There are `6 HW2 problems` assigned ✔
  - **Due:** Thu, Sept 18, 11PM
- There are `5 HW3 problems` assigned ✔
  - **Due:** Thu, Sept 25, 11PM
- There are `5 HW4 problems` assigned ✔
  - **Due:** Thu, Oct 2, 11PM
- There are `5 HW5 problems` assigned ✔
  - **Due:** Thu, Oct 9, 11PM
- There are `6 HW6 problems` assigned ✔
  - **Due:** Thu, Oct 23, 11PM
- There are `2 HW7 problems` assigned ✔
  - **Due:** Thu, Oct 30, 11PM
- There are `6 HW8 problems` assigned ✔
  - **Due:** Thu, Nov 6, 11PM
- There are `5 HW9 problems` assigned ✔
  - **Due:** Sat, Nov 13, 11PM
- There are `5 HW10 problems` assigned ✔
  - **Due:** Thu, Nov 20, 11PM
- There are `0 HW11 problems` assigned ✔
  - **Due:** Thu, Dec 4, 11PM

*This document was written by **Sam Chevalier**. Credit to **Yoonki Hong** and **Eren Tekeler** for proof reading and correctness verification. Credit to **Safwan Wshah** for general inspiration and some figures.

Contents

1 Introduction and Background

1.1 A Brief History of... AI

The histories of Machine Learning (ML) and Artificial Intelligence (AI) are closely tied to the histories of **computation**, mathematical completeness, and algorithmic complexity. This story may be told in many ways, but here is Sam's version (strongly influenced by [5, 6, 7, 8, 9]).

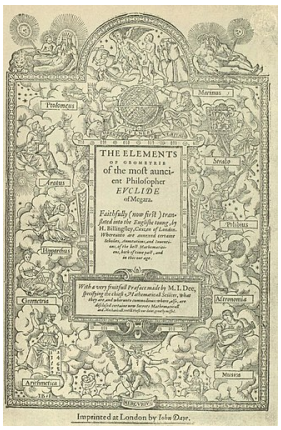


Figure 1: *Elements.*

We begin 2300 years ago, with the Greek mathematician **Euclid**. His famous treatise on Geometry, the ***Elements***, is one of the most reproduced books of all time. In this treatise, Euclid offers five famous postulates:

1. A straight line can join any two points.
2. Any straight line can be extended indefinitely.
3. Given any straight line, a circle can be drawn having the segment as radius and one endpoint as center.
4. All Right Angles are congruent.
5. **If two lines are drawn which intersect a third in such a way that the sum of the inner angles on one side is less than two Right Angles, then the two lines inevitably must intersect each other on that side if extended far enough.** ⋆

This set of postulates forms the foundations of an **Axiomatic System**; in such a system, the initial set of true axioms are used to logically deduce other, more complex, yet less obvious, truths (we call them **theorems**). While Euclid's first four postulates are rock solid, the **fifth postulate** has a problem. For starters, it is less self-evident than the first four; even Euclid himself tried to avoid using the 5^{th} postulate in his proofs. Despite its squishiness, it still "feels" true, and generations mathematicians after Euclid attempted to **rigorously prove** the fifth postulate from the preceding four. All tried, and failed. By 1763, there were at least 28 erroneous proofs published in the literature.

Farkas Bolyai, a famous Hungarian mathematician, spent many years of his life attempting to prove the parallel line conjecture. When he learned that his son, **János Bolyai**, began to also work on the topic, his father passionately tried to dissuade him:

> "You must not attempt this approach to parallels. I know this way to its very end. I have traversed this bottomless night, which extinguished all light and joy of my life. I entreat you, leave the science of parallels alone... I though I would sacrifice myself for the sake of the truth. I was ready to become a martyr who would remove the flaw from geometry and return it purified to mankind. I accomplished monstrous, enormous labors; my creations are far better than those of others and yet I have not achieved complete satisfaction. For here it is true that *si paullum a summo discessit, vergit ad imum* (*if it's failed to make the grade, even by a smidgen, it might as well be the worst*). I turned back when I saw that no man can reach the bottom of this night. I turned back unconsoled, pitying myself and mankind. I have traveled past all reefs of this infernal Dead Sea and have always come back with broken mast and torn sail. The ruin of my disposition and my fall date back to this time. I thoughtlessly risked my life and happiness–*aut Caesar aut nihil* (*either Caesar or nothing*)."

Very soon after, though, János, **Nikolai Ivanovich Lobachevsky**, and even **Carl Friedrich**

Gauss all simultaneously* discovered **non-Euclidean geometry** (e.g., hyperbolic geometry, elliptic geometry, Riemannian geometry). In many of these new geometries, Euclid's fifth postulate simply isn't true; since reality is not inherently Euclidean, it is no surprise that the fifth postulate gave mathematicians so much trouble.

As non-Euclidean geometry was shaking the foundations of mathematics, there was another emerging problem. In 1891, German mathematician and set theory founder **Georg Cantor** published his **diagonalization proof**, where he showed that there are more real numbers between 0 and 1, than there are natural numbers. Since both are infinitely large, we are left with an odd conclusion: *not all infinities are the same size*. Of course, this result defies intuition.

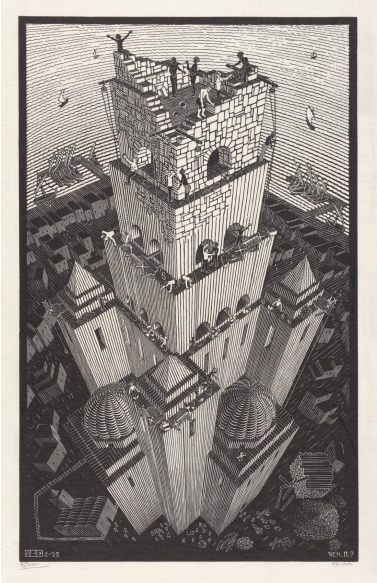


Figure 2: Tower of Babel by M. C. Escher [7].

With these unsettling upheavals, the end of the 19$^{\text{th}}$ century witnessed a fracturing of the mathematical community. On one side, the **intuitionists** (lead by Brouwer) rejected the paradoxes emerging from set theory. On the other, the **formalists** believed that axiomatic rigor, grounded in set theory, would rescue the drowning field of pure math.

Lead by the famous German mathematician **David Hilbert**, the formalists embraced set theory and generally believed that a sure-footed and properly executed axiomatic system of formalized logic and structure could reveal all mathematical truths. To the intuitionists, Hilbert declared, "**No on shall expel us from the paradise that Cantor created.**" Many were skeptical of this approach, including **Emil du Bois-Reymond**. He believed there are certain "transcendent" truths which are **beyond** what humans can know; he employed the Latin phrase "`ignoramus et ignorabimus`", meaning "we do not know and will not know". In response, David Hilbert famously said, in 1930, "Wir müssen wissen – wir werden wissen." (In English: "**We must know – we will know**"; these words are now on his grave). Hilbert's famous "Foundations of Geometry" treatise, published in 1902, set out to do just this.

In 1901, **Bertrand Russell** found a **paradox** at the heart set theory. Some sets are members of themselves (i.e., the set of all sets). Some sets are not (i.e., the set of all UVM students). "Russell's paradox" considers $\mathcal{R}$: "the set of all sets that don't contain themselves." Does $\mathcal{R}$ contain itself? Either answer is wrong, leading to an obvious **paradox**. Practically, we can think of a small village with a barber. This barber will cut the hair of every person who doesn't cut their own hair. So, who cuts the barber's hair...?

The heart of the problem is related to the concept of **self-reference**, especially recursive self-reference. **Douglas Hofstadter** refers to the complexity emerging from such recursive self-referential systems as a consequence of "strange loops" (see Gödel, Escher, Bach [7]). To overcome these challenges, the formalists wanted a stronger axiomatic system which rooted out self-reference altogether. In 1913, Bertrand Russell and Alfred North Whitehead published the third volume of their monumental, and massive (2000 pages!), *Principia Mathematica* [10]. After 379 pages, they were able to finally prove that "1+1=2", which is an "occasionally useful" result, they remarked [5]. In *Principia Mathematica*, they introduce the "**theory of types**", a hierarchy of sets which completely disallows self-reference (i.e., only a higher-order set can refer to a lower-order

*Farkas Bolyai on the budding co-discoveries of non-Euclidean geometry: "When the time is ripe for certain things, these things appear is different places in the manner of violets coming to light in early spring." [7]

set).

Pleased by this progress, at the 1928 International Congress of Mathematicians, David Hilbert posed three problems that must be answered to ensure Math was was a truly bulletproof fortress:

1. Is Math **complete**? [`Can you prove every true statement?`]
2. Is Math **consistent**? [`Is it free of contradictions?`]
3. Is Math **decidable**? [`Is there a step-by-step algorithm which can determine if a theorem or statement follows from the axioms?`] This is also known as the Entscheidungsproblem, which is German for the "decision problem".

In 1931, a young logician named **Kurt Gödel** published *On Formally Undecidable Propositions of Principia Mathematica and Related Systems I* [11]. It contains two Earth shattering theorems:

I. In any sufficiently powerful formal system (i.e., conventional Math), there are statements which can **neither be proved nor disproved**.

II. Any sufficiently strong formal system **cannot prove its own consistency**.

And with that, mathematics was show to be **incomplete**: there are truths that cannot be proven. Furthermore, the consistency of math cannot be shown by math itself; we need a more powerful system; but that more powerful system can only be shown to be consistent by an *even* more power system, and so on. While mathematical completeness was shown to be a pipe dream, ***Entscheidungsproblem*** problem was still up for debate. If math was decidable, then at the very least, the things that can be proven true, can be done so with algorithmic ease (once the algorithm is discovered, or course).

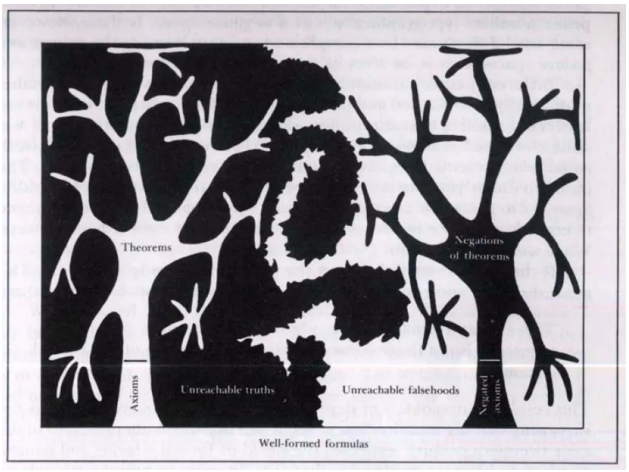


Figure 3: A gorgeous depiction, by Hofstadter [7], of provable and unprovable theorem space.

Alan Turing then went a step farther. To do so, Turing famously introduced the concept of a **Turing machine**, and he showed that this machine can compute anything that is computable. While this sounds like a trivial statement to you and me, (non-human) computers did not exist yet, so these concepts were highly abstract. Famously, Turning then formulated the **halting problem** as an equivalent version of the **decidability** problem: the Turing machine can compute anything, so if we can a priori predict if it will **halt** when executing a program, then we can solve any decidability problem. As an example, the famously unsolved **Riemann hypothesis** asks: "Do all nontrivial zeros of the Riemann zeta function line on the $0.5+j\omega$ line?" To solve this problem, we can just design a program which searches for a solution to the zeta function *off* the $0.5 + j\omega$ line; if this problem halts, then we can solve the **Riemann hypothesis**, since we know it has found a solution. A similar approach could be used to solve other problems, like the **twin prime conjecture** [6].

In 1936, in *On computable numbers, with an application to the Entscheidungsproblem* [12], Alan Turing showed that the **halting problem cannot be solved**. As demonstrated in proof-by-contradiction by Christopher Strachey, let's assume `halts(f)` is a function which **can** predict if

program `f` will halt. Let's now run a new program, `g`:

$$\texttt{g} = \begin{cases} \text{loop}, & \texttt{halts}(\texttt{g}) = \text{true} \\ \text{halt}, & \texttt{halts}(\texttt{g}) = \text{false}. \end{cases} \tag{1.1}$$

This, of course, leads to a logical **contradiction**, since `g` will neither halt, nor endlessly loop. The function `halts(f)`, therefore, cannot exist, because it leads to impossible situations. Thus, mathematics is **fundamentally undecidable**. Sometimes, to know if a program will halt, we just need to run it, potentially forever[†]. Once again, self-referential recursive routines brought mathematics, and in particular, its *decidability*, cowering to its knees.

To overcome the **Entscheidungsproblem**, the Turing machine was naturally born. With this invention, Alan Turing is considered the father of computer science; however, he was not the first to hypothesize about the existence of general purpose computation. That prize goes to **Ada Lovelace**. In 1834, the English polymath **Charles Babbage** proposed and designed his **Analytical Engine** (sadly, it was never actaully built). This was a loom-like mechanical device capable of being programmed with punch cards. About this device, Ada Lovelace said, "The Analytical Engine weaves algebraic patterns just as the Jacquard loom weaves flowers and leaves." Lovelace wrote the first (ever) "program" which could run on this engine (a program for computing Bernoulli numbers). More fundamentally, Lovelace noticed the power of the Analytic Engine to provide more than just calculation; such an engine could also provide **general-purpose computation**:

> "Ada saw something that Babbage in some sense failed to see. In Babbage's world his engines were bound by number...What Lovelace saw...was that number could represent entities other than quantity. So once you had a machine for manipulating numbers, if those numbers represented other things, letters, musical notes, then the machine could manipulate symbols of which number was one instance, according to rules. It is this fundamental transition from a machine which is a number cruncher to a machine for manipulating symbols according to rules that is the fundamental transition from calculation to computation—to **general-purpose computation**—and looking back from the present high ground of modern computing, if we are looking and sifting history for that transition, then that transition was made explicitly by Ada in [1843]." [13]

As formal mathematics crumbled and new theories and methods of computation were emerging, a young (12 years old, to be exact) prodigy named **Walter Pitts** was reading Bertrand Russell's *Principia Mathematica* in the public libraries of Detroit [10]; he even managed to find several "problems" in the work, which he pointed out to Bertrand Russell in a letter. At the age of 25, Walter Pitts teamed up with **Warren McCulloch**, a neuropsychologist and cybernetician. Both Pitts and McCulloch were strongly influenced by Russell's attempt to use axiomatic logic to derive truths, and by Turing's new universal Turing machine, which could compute any computable function. "McCulloch became convinced that the brain was just such a machine—one which uses logic encoded in neural networks to compute. Neurons, he thought, could be linked together by

[†]"What had he (John von Neumann) seen? What had the computer shown him? Had he experienced a similar revelation to mine? There was no way for me to know without running his code. Because that is a basic computational truth that very few people are aware of, and that Turing proved mathematically: there is simply no form of knowing what a particular string of code will do unless you run it. You cannot know by looking at it. Even the simplest programs can lead to fabulous complexity. And the opposite is also true: you can erect a sprawling, many-leveled tower of ciphers that produce nothing but sterility, a barren unchanging landscape where no water will ever fall. So I will go to my grave with that knowledge withheld from me, and it tortures my curiosity." – Benjamin Labatut, speaking on behalf of Nils Aall Barricelli in *The Maniac*.

the rules of logic to build more complex chains of thought, in the same way that the *Principia* linked chains of propositions to build complex mathematics." [9]. McCulloch and Pitts laid out these ideas in their seminal work, *A Logical Calculus of Ideas Immanent in Nervous Activity* [14]. This was the first work to model the artificial neural network, and they showed that this model could theoretically compute all logical functions.

In 1945, **John von Neumann** published "First Draft of a Report on the EDVAC," which proposed the first general purpose binary computing machine; without any rewiring, this machine could operate as a universal Turing machine. "To accomplish this, von Neumann suggested modeling the computer after Pitts and McCulloch's neural networks. In place of neurons, he suggested vacuum tubes, which would serve as logic gates, and by stringing them together exactly as Pitts and McCulloch had discovered, you could carry out any computation." [9] The only paper referenced in this report was the McCulloch and Pitts' *Logical Calculus* [14].

And the rest is history. Modern Neural Networks architectures have unleashed the unbridled **power** of Machine Learning, yielding some of humanity's first truly **Artificially Intelligent** learning systems. The history of Machine Learning, as we have now seen, is closely linked to the history of computation; and computation primarily emerged as a **theoretical tool** for analyzing the crumbling formal mathematical systems of the 1800s.

There are other interesting facets to explore, of course, but we'll end our story here. Despite passing through two "AI winters" (i.e., 1974–1980 and 1987–2000, where progress and interest in the field of AI seriously slumped), Artificial Intelligence today is booming, and its story is clearly not over. AI is not a lifeless technology that came to us from beyond the void; its very roots are inextricably fused with humanity's struggle to calculate, to compute, to learn, and to ***know***. These roots also highlight how **people** have carried these complex ideas forward from generation to generation. So, what part will **you** play?

1.2 And Now: A Big, Annoying Notation Table

In this document, there will be a lot of math, so let's getting our notation all set. Matrices are generally represented by uppercase letters, e.g., A, B, C, bold lowercase letters generally represent vectors, e.g., $\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{c}$, and un-bold lowercase letters generally represent scalars, e.g., a, b, c.

Symbol(s)	Description
A, B, C, ... X, Y, Z	Matrices, Tensors, or Random Variables (depends on context) $M \in \mathbb{R}^{m \times n}$ is clearly a matrix $T \in \mathbb{R}^{m \times n \times p}$ is clearly a tensor $X \sim \mathcal{N}(\mu, \sigma^2)$ is clearly a random variables
a, b, c, ... x, y, z	Scalars
$\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{c}$, ... $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{z}$	Vectors
$\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, ... $\mathcal{X}$, $\mathcal{Y}$, $\mathcal{Z}$	Sets, in most cases. Some **exceptions**: $\mathcal{N}$ is a normal distribution $\mathcal{L}$ is a loss function, or a Lagrangian
$\mathcal{S} \times \mathcal{T}$	Cartesian product: denotes the set of elements $\{s \in \mathcal{S}, t \in \mathcal{T}\}$
i, j, k	Usually used for indices
M_{ij}	Element at the i^{th} row and j^{th} column of matrix M
$\{\cdot\}^T$	Transposition operator
$\{\cdot\}^H$	Conjugate, or Hermitian, transpose
$\mathbb{R}$	Set of real values
$\mathbb{R}^n$	Set of real-valued vectors of length n (column vector)
$\mathbb{R}^{m \times n}$	Set of real-valued matrices of dimension $m \times n$
$\mathbb{Z}$	Set of integers
$\mathbb{C}$	Set of complex values
$\in$	"belongs to the set"
$\exists$	"there exists"
$\subseteq$	"is a subset of", e.g., $A \subseteq B$
$\subset$	"is a **proper** subset of", e.g., $A \subset B$
$\forall$	"for any", or "for all"
: or $\mid$	"such that", like $x \mid f(x) \geq 0$
$\triangleq$	"is defined to be"
$\equiv$	"identical to", like, $f(x) \equiv 0$
$\Rightarrow$	"implies", like, $f(x) = 0 \quad \Rightarrow \quad g(x) \cdot f(x) = 0$
$\setminus$	"excluding", like $\mathcal{A} \setminus \mathcal{B}$, which means, $\mathcal{A}$ without the elements of $\mathcal{B}$
$\mathcal{S} = \{...\}$	Used to defined a set via "**set-builder notation**"
$\mathcal{D} = \{(\boldsymbol{x}_n, \boldsymbol{y}_n) \mid n = ...\}$	Set of labeled training data
$\hat{\boldsymbol{y}}_n$	Predicted labels
I	Identity matrix
$\hat{\boldsymbol{e}}_i$	Unit/"one hot" vector: all zeros, but position i is 1
$\text{diag}\{\cdot\}$	Diagonalization operator, mapping a vector to a matrix
$\det\{\cdot\}$	Determinant operator
$\text{E}\{\cdot\}$	Expected value operator
μ	Mean ($\mu_x = \text{E}[x]$)
σ^2	Variance ($\mu_x = \text{E}[(x - \mu_x)^2]$)
∇	Gradient operator: $\nabla_x f(x)$ means take the partials $f(x)$ wrt x **Note:** physicists might bristle at this definition

$\|\boldsymbol{x}\|_p$	p norm of vector x
$\|\boldsymbol{x}\|$	Generally, euclidean magnitude: $\|\boldsymbol{x}\| \triangleq \|\boldsymbol{x}\|_2$
$\bar{\boldsymbol{x}}$	Normalized ($\|\bar{\boldsymbol{x}}\| = 1$) version of vector $\boldsymbol{x}$, where $\bar{\boldsymbol{x}} = \boldsymbol{x}/\|\boldsymbol{x}\|$
$\otimes$	Kronecker product
$\lambda\{\cdot\}$	Eigenvalue operator
$\sum_{i=1}^{n}$	Summation, from $i = 1$ to n
$\text{svd}\{\cdot\}$	Singular Value Decomposition operator

Table 1: Commonly Used Notation

⋆ Example 1: Translating math to English I

Here are some examples of how we use this table to "speak math":

(a) $\mathcal{X} = \{x \subseteq \mathbb{R} \,|\, f(x) \geq 0\}$. This means, "$\mathcal{X}$ is the set of all x, which are a subset of the reals, such that $f(x) \geq 0$".

(b) $f(x, y) : \mathcal{X} \times \mathcal{Y} \to \mathbb{Z}^{m \times n}$. This means, "The function f which maps pairs of things from the sets $\mathcal{X}$ and $\mathcal{Y}$ to an $m \times n$ matrix of integers."

Your turn.

⋆ Homework 0, Problem 1: Translating math to English II

Translate the following mathematical statements into English words.

(a) $\mathbb{Z} \subset \mathbb{R} \subset \mathbb{C}$

(b) $\forall y \in \mathcal{Y}, \exists x : f(x, y) \geq 0$

(c) $\mathcal{M} = \{M \in \mathbb{C}^{m \times n} \,|\, M_{ij} \geq 0, M_{ij} = M_{ji}\}$

(d) $\mathcal{Y} = \{y \,|\, y \notin \mathcal{Y}\}$ ...🤯 [a]

[a] This is Bertrand Russell's paradox, and it's actually at the heart of Gödel's Incompleteness Theorems. Practically, we can think of a small village with a barber. This barber will cut the hair of every person who doesn't cut their own hair. So, who cuts the barber's hair...?

Solution.

(not posted yet)

1.3 Linear Algebra Review

Matrices. $A \in \mathbb{R}^{m \times n}$ is called a matrix, or a 2D array. Higher dimensional arrays (i.e., arrays with dimensions > 2 are called **tensors**: $A \in \mathbb{R}^{m \times n \times p}$. RGB images, for examples, are often represented as 3D tensors, where dimensions are associated with channels, pixel height, and pixel width. Following are several useful matrix properties:

- If $A = A^T$, then matrix A is symmetric.
- If $A = A^H$, then matrix A is Hermitian (i.e., conjugate symmetric)
- $(A + B)^T = A^T + B^T$, but $(AB)^T = B^T A^T$.
- $(AB)^{-1} = B^{-1}A^{-1}$, assume A, B **square** and **invertible**.

Linear Maps. A function $f : \mathcal{V} \to \mathcal{W}$ is said to be a linear map, or linear transformation, if

$$f(\alpha x + y) = \alpha f(x) + f(y),\ \forall x, y \in \mathcal{V}. \tag{1.2}$$

In other words, "linear transformations preserve the operations of vector addition and scalar multiplication." This is called superposition.

Vector Norms. The following norms are useful to understand. Assume $\boldsymbol{x} \in \mathbb{R}^n$

- p-norm: $\|\boldsymbol{x}\|_p = (\sum_{i=1}^n |x_i|^p)^{\frac{1}{p}}$, $p \geq 1$.
- ∞-norm: The infinity norm returns the largest element in a set: $\|\boldsymbol{x}\|_\infty = \max_i\{|x_i|\}$. This can also be computed by applying limit calculus to the equation $y = (x_1^\infty + x_2^\infty + \cdots + x_n^\infty)^{\frac{1}{\infty}}$.

- 2-norm: $\|\boldsymbol{x}\|_2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$. This is the Euclidean or "Pythagorean" magnitude of a vector, and it can also be computed via $\sqrt{\boldsymbol{x}^T\boldsymbol{x}}$.
- 1-norm: This is the Manhattan norm, and it measures the sum of the absolute values: $\|\boldsymbol{x}\|_1 = |x_1| + |x_2| + \cdots + |x_n|$
- 0-norm: The 0-norm is **not a real norm**, and it is highly nonconvex, but it simply counts the number of nonzero elements in a vector:

$$\|\boldsymbol{x}\|_0 = \sum_{i=1}^{n} b_i, \quad b_i = \begin{cases} 1, & x_i \neq 0 \\ 0, & x_i = 0. \end{cases} \tag{1.3}$$

Hyperplanes. A hyperplane is just an $n-1$ dimensional linear surface in an n dimensional space (e.g., a line in 2D, a plane in 3D, etc).

Hyperplane Orthogonality

When a hyperplane is described by the equation $\boldsymbol{w}^T\boldsymbol{x} + w_0 = 0$, the vector $\boldsymbol{w}$ is **orthogonal** to the hyperplane.

Proof. Take two distinct points on the hyperplane as $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$. The difference vector $\boldsymbol{d} = \boldsymbol{x}_1 - \boldsymbol{x}_2$ must lie on the hyperplane. We may test the orthogonality of $\boldsymbol{d}$ and $\boldsymbol{w}$:

$$\begin{aligned} \boldsymbol{w}^T\boldsymbol{d} &= \boldsymbol{w}^T(\boldsymbol{x}_1 - \boldsymbol{x}_2) && \text{(1.4a)} \\ &= \boldsymbol{w}^T\boldsymbol{x}_1 - \boldsymbol{w}^T\boldsymbol{x}_2 && \text{(1.4b)} \\ &= w_0 - w_0 && \text{(1.4c)} \\ &= 0. && \text{(1.4d)} \end{aligned}$$

Therefore, $\boldsymbol{w}$ is perpendicular to the hyperplane, since $\boldsymbol{d} \perp \boldsymbol{w}$. □

Linear Projection. The inner product, or "dot product" of two vectors is given by

$$\boldsymbol{x} \cdot \boldsymbol{y} = \boldsymbol{x}^T\boldsymbol{y} = \sum_{i=1}^{n} x_i y_i = \|\boldsymbol{x}\| \, \|\boldsymbol{y}\| \cos(\theta), \tag{1.5}$$

where θ is the angle between the vectors $\boldsymbol{x}$ and $\boldsymbol{y}$. We may define the projection of $\boldsymbol{y}$ onto $\boldsymbol{x}$ via

$$\text{proj}_{\boldsymbol{x}}\boldsymbol{y} = \frac{\boldsymbol{x}^T\boldsymbol{y}}{\boldsymbol{x}^T\boldsymbol{x}}\boldsymbol{x} = \frac{\boldsymbol{x}^T\boldsymbol{y}}{\|\boldsymbol{x}\|_2^2}\boldsymbol{x} = \frac{\boldsymbol{x}^T\boldsymbol{y}}{\|\boldsymbol{x}\|_2 \, \|\boldsymbol{x}\|_2}\boldsymbol{x} = \left(\frac{\boldsymbol{x}^T}{\|\boldsymbol{x}\|_2}\boldsymbol{y}\right)\frac{\boldsymbol{x}}{\|\boldsymbol{x}\|_2} = \left(\bar{\boldsymbol{x}}^T\boldsymbol{y}\right)\bar{\boldsymbol{x}}. \tag{1.6}$$

This represents "the portion of $\boldsymbol{y}$ which points in the $\boldsymbol{x}$ direction". A few notes about the projection formulation:

- **Meaning.** $\text{proj}_{\boldsymbol{x}}\boldsymbol{y}$ is confusing. It just is. A good trick is to remember that $\text{proj}_{\boldsymbol{x}}y$ is a **vector** will always point in the $\boldsymbol{x}$ direction. If $\boldsymbol{y}$ is projected onto $\boldsymbol{x}$, then the result will point in the $\boldsymbol{x}$ direction. Look at the notation: in $\text{proj}_{\boldsymbol{x}}\boldsymbol{y}$, it looks like $\boldsymbol{y}$ is falling onto $\boldsymbol{x}$.
- **Invariance to scaling.** The size of $\boldsymbol{x}$ does not matter. It could be a unit vector, or an infinitely large vector. The results of the projection will not change: **only direction matters**. For example, let's take $\boldsymbol{x}$ and scale it up by a scalar α:

$$\text{proj}_{\alpha\boldsymbol{x}}\boldsymbol{y} = \frac{(\alpha\boldsymbol{x})^T\boldsymbol{y}}{(\alpha\boldsymbol{x})^T(\alpha\boldsymbol{x})}(\alpha\boldsymbol{x}) = \frac{\alpha^2}{\alpha^2}\frac{\boldsymbol{x}^T\boldsymbol{y}}{\boldsymbol{x}^T\boldsymbol{x}}\boldsymbol{x} = \frac{\boldsymbol{x}^T\boldsymbol{y}}{\boldsymbol{x}^T\boldsymbol{x}}\boldsymbol{x} = \text{proj}_{\boldsymbol{x}}\boldsymbol{y} \tag{1.7}$$

We may also define a "**rejection**", where a projection is subtracted out of the original vector:

$$\text{rej}_{\boldsymbol{x}}\boldsymbol{y} \triangleq \boldsymbol{y} - \text{proj}_{\boldsymbol{x}}\boldsymbol{y} = \boldsymbol{y} - \frac{\boldsymbol{x}^T\boldsymbol{y}}{\boldsymbol{x}^T\boldsymbol{x}}\boldsymbol{x}. \tag{1.8}$$

To summarize, we have

$$\text{proj}_{\boldsymbol{x}}\boldsymbol{y} = \frac{\boldsymbol{x}^T\boldsymbol{y}}{\boldsymbol{x}^T\boldsymbol{x}}\boldsymbol{x}, \qquad \textbf{projection}\text{: identifies the portion of } \boldsymbol{y} \text{ which "overlaps" } \boldsymbol{x}$$

$$\text{rej}_{\boldsymbol{x}}\boldsymbol{y} = \boldsymbol{y} - \frac{\boldsymbol{x}^T\boldsymbol{y}}{\boldsymbol{x}^T\boldsymbol{x}}\boldsymbol{x}, \qquad \textbf{rejection}\text{: subtracts projection to make } \boldsymbol{y} \text{ orthogonal to } \boldsymbol{x}.$$

Vector projection can be extended into higher dimension. In particular, we are often interested in projecting a vector $\boldsymbol{y}$ onto the column space of a matrix A. To do this, we first need to define the **range** (or column space) of the matrix A. Intuitively, the range is simple: it is any linear sum of the columns:

$$\text{range}(A) \triangleq \{\boldsymbol{v} \,|\, \boldsymbol{v} = A\boldsymbol{x}, \boldsymbol{x} \in \mathbb{R}^n\}: \quad \textbf{Matrix Range}. \tag{1.9}$$

Assume matrix A is non-square (i.e., A^{-1} is not defined), and in particular, "is tall and skinny": $A \in \mathbb{R}^{m\times n}, m > n$. By analogy, we can define the following matrix projection:

$$\text{proj}_{\boldsymbol{x}}\boldsymbol{y} = \boldsymbol{x}\frac{\boldsymbol{x}^T\boldsymbol{y}}{\boldsymbol{x}^T\boldsymbol{x}} \qquad \in \text{range}(\boldsymbol{x}) \tag{1.10}$$

$$\text{proj}_{A}\boldsymbol{y} = A\left(\frac{A^T\boldsymbol{y}}{A^TA}\right) = \underbrace{A\underbrace{\left(A^TA\right)^{-1}A^T}_{A^+}\boldsymbol{y}}_{\boldsymbol{x}^*} \qquad \in \text{range}(A) \tag{1.11}$$

where A^+ is **Moore–Penrose pseudo-inverse** of the matrix A[‡]. The vector $\boldsymbol{x}^*$ represents the vector which, when multiplied by A, will give us the output which (i) is optimally close to the original vector $\boldsymbol{y}$, yet still falls in the range(A) (our original goal).

We can also interpret, and derive, this result analytically by considering the problem

$$\boldsymbol{x}^* = \arg\min_{\boldsymbol{x}} \; \|\boldsymbol{y} - A\boldsymbol{x}\|_2^2 \,. \tag{1.12}$$

Note: (1.11) and (1.12) solve the same problem and yield the same solution (proof later on! Get excited!).

★★ Projecting a Point onto a Line ★★

We can consider a point in space (e.g., $(x, y)_p$, denoted by the vector $\boldsymbol{x}_p$) and a hyperplane (e.g., $y = ax + b$, denoted by $0 = \boldsymbol{w}^T\boldsymbol{x} + w_0$), where **the point does not fall directly on the hyperplane** :(What is the closest projection of this point onto this hyperplane? To solve this problem, we take some test point on the hyperplane $0 = \boldsymbol{w}^T\boldsymbol{x} + w_0$, which we call $\boldsymbol{x}_l$, where $0 = \boldsymbol{w}^T\boldsymbol{x}_l + w_0$ is satisfied (i.e., it falls on the hyperplane!). **Note:** we don't actually need this point; it's just a useful trick. Let's define $f(\boldsymbol{x})$ as the hyperplane equation:

$$f(\boldsymbol{x}) \triangleq \boldsymbol{w}^T\boldsymbol{x} + w_0, \tag{1.13}$$

where $f(\boldsymbol{x}_l) = 0$, since it falls on the hyperplane, and $f(\boldsymbol{x}_p) \neq 0$.

[‡]In this class, A will usually represent a tall-skinny data matrix, so A^+ will be directly computable.

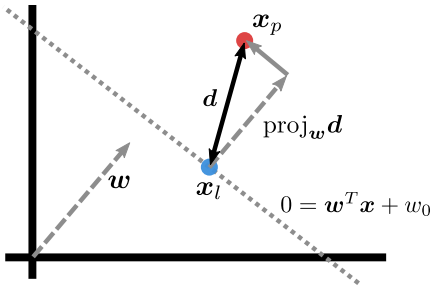


Figure 4: Projection

The distance vector $\boldsymbol{d}$ between the point on the hyperplane ($\boldsymbol{x}_l$) and the point we care about ($\boldsymbol{x}_p$) is

$$\boldsymbol{d} = \boldsymbol{x}_p - \boldsymbol{x}_l, \tag{1.14}$$

as depicted in Fig. 4. In this figure, the vector $\boldsymbol{w}$ is normal to the hyperplane. If we find the projection of $\boldsymbol{d}$ in the direction of $\boldsymbol{w}$, then we will find the distance we are seeking:

$$\begin{aligned}
\text{proj}_{\boldsymbol{w}}\boldsymbol{d}_h &= \frac{\boldsymbol{w}^T\boldsymbol{d}}{\boldsymbol{w}^T\boldsymbol{w}}\boldsymbol{w} && \text{(1.15a)}\\
&= \frac{\boldsymbol{w}^T\left(\boldsymbol{x}_p - \boldsymbol{x}_l\right)}{\boldsymbol{w}^T\boldsymbol{w}}\boldsymbol{w} && \text{(1.15b)}\\
&= \frac{\boldsymbol{w}^T\boldsymbol{x}_p + w_0}{\boldsymbol{w}^T\boldsymbol{w}}\boldsymbol{w} && \text{(1.15c)}\\
&= \frac{f(\boldsymbol{x}_p)}{\boldsymbol{w}^T\boldsymbol{w}}\boldsymbol{w}. && \text{(1.15d)}
\end{aligned}$$

The result of this projection is a vector. We can get the distance d_h associated with this vector by taking its two norm:

$$\begin{aligned}
d_h &= \sqrt{\left(\text{proj}_{\boldsymbol{w}}\boldsymbol{d}\right)^T\left(\text{proj}_{\boldsymbol{w}}\boldsymbol{d}\right)} && \text{(1.16a)}\\
&= \sqrt{\left(\frac{f(\boldsymbol{x}_p)}{\boldsymbol{w}^T\boldsymbol{w}}\boldsymbol{w}\right)^T\left(\frac{f(\boldsymbol{x}_p)}{\boldsymbol{w}^T\boldsymbol{w}}\boldsymbol{w}\right)} && \text{(1.16b)}\\
&= \sqrt{f(\boldsymbol{x}_p)^2\frac{\boldsymbol{w}^T\boldsymbol{w}}{\left(\boldsymbol{w}^T\boldsymbol{w}\right)\left(\boldsymbol{w}^T\boldsymbol{w}\right)}} && \text{(1.16c)}\\
&= \frac{|f(\boldsymbol{x}_p)|}{\sqrt{\boldsymbol{w}^T\boldsymbol{w}}} && \text{(1.16d)}\\
&= \frac{|f(\boldsymbol{x}_p)|}{\|\boldsymbol{w}\|_2}. && \text{(1.16e)}
\end{aligned}$$

Shortest Distance from a Point to Hyperplane

Given a hyperplane $0 = \boldsymbol{w}^T\boldsymbol{x} + w_0$, residual function $f(\boldsymbol{x}) \triangleq \boldsymbol{w}^T\boldsymbol{x} + w_0$, and point $\boldsymbol{x}_p$, the distance d_h of the closest orthogonal projection of this point onto the hyperplane is given by

$$d_h = \frac{|f(\boldsymbol{x}_p)|}{\|\boldsymbol{w}\|_2}. \tag{1.17}$$

If we care about the sign of this distance (as with Support Vector Machines (SVMs), where the sign of $f(\boldsymbol{x}_p)$ matters), we can drop the absolute value:

$$d_{h\pm} = \frac{f(\boldsymbol{x}_p)}{\|\boldsymbol{w}\|_2}. \tag{1.18}$$

Proof. See above. □

We will soon re-derive this result using direct optimization.

Gram–Schmidt. Given some set of vectors, projections and rejections can be used to "orthogonalize" and/or "orthonormalize" the set.

- **orthogonal**: two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ are generally orthogonal if $\boldsymbol{x}^T\boldsymbol{y} = \boldsymbol{0}$
- **orthogonalize**: to make all vectors of a set orthogonal to each other
- **orthonormalize**: to make all vectors of a set normalized to unit magnitude and orthogonal to each other

Given a set of vectors $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, ... $\boldsymbol{x}_n$, Gram–Schmidt sequentially orthogonalizes this set:

$$\boldsymbol{z}_1 = \boldsymbol{x}_1 \tag{1.19}$$
$$\boldsymbol{z}_2 = \boldsymbol{x}_2 - \text{proj}_{\boldsymbol{z}_1}\boldsymbol{x}_2 \tag{1.20}$$
$$\boldsymbol{z}_3 = \boldsymbol{x}_3 - \text{proj}_{\boldsymbol{z}_2}\boldsymbol{x}_3 - \text{proj}_{\boldsymbol{z}_1}\boldsymbol{x}_3 \tag{1.21}$$
$$\vdots$$
$$\boldsymbol{z}_i = \boldsymbol{x}_i - \sum_{j=1}^{n-1} \text{proj}_{\boldsymbol{z}_j}\boldsymbol{x}_i. \tag{1.22}$$

The set can then be orthonormalized via:

$$\bar{\boldsymbol{z}}_1 = \frac{\boldsymbol{z}_1}{\|\boldsymbol{z}_1\|},\ \bar{\boldsymbol{z}}_2 = \frac{\boldsymbol{z}_2}{\|\boldsymbol{z}_2\|},\ \cdots\ \bar{\boldsymbol{z}}_i = \frac{\boldsymbol{z}_i}{\|\boldsymbol{z}_i\|}. \tag{1.23}$$

⋆ Homework 0, Problem 2: Gram–Schmidt

Consider the following set of vectors:

$$\boldsymbol{x}_1 = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix},\ \boldsymbol{x}_2 \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix},\ \boldsymbol{x}_3 \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}. \tag{1.24}$$

(a) Via Gram–Schmidt, orthogonolize the following set of vectors, showing all steps (you

may always use Python for computing the actual math).

(b) Now, orthornormalize the set (i.e., by normalization). Again, please use Python.

(c) Now, take your three normalized vectors and build the matrix $Z = [\bar{z}_1\ \bar{z}_2\ \bar{z}_3]$. Does $Z * Z^T$ or $Z^T * Z$ (these are matrix-matrix products) produce the identity matrix? Try both!

Solution.

(not posted yet)

1.4 Optimization Review

Unconstrained Optimization. The process of optimization involves finding the "best" solution to a given problem. Sometimes, the problem will have an analytical solution; sometimes, we need to use numerical methods (tricks) to find a good solution. Consider the following problem:

$$\min_x \ f(x). \tag{1.25}$$

This means "minimize the function $f(x)$ by tuning variable x." In this case, x is called the decision variable, because the optimizer can directly decide its value. This is an example of an **unconstrained** optimization problem, because there are no constraints. Sometimes, we may also use the "arg min" notation, which means, "return the value of the **argument** which solves the problem."

$$y_1 = \min_x \ f(x) \qquad \rightarrow \textbf{returns} \text{ the optimal value of } f(x) \tag{1.26}$$

$$y_2 = \arg\min_x \ f(x). \qquad \rightarrow \textbf{returns } x \text{ at optimal value of } f(x). \tag{1.27}$$

In other words, $y_1 = f(y_2)$. When a problem is **unconstrained**, we solve it using the following three steps:

1. Take the gradient of the objective function
2. Set the gradient vector equal to 0
3. Solve this set of equations

The **gradient vector** associated with a function $f(\boldsymbol{x})$ is given/denoted by the **column vector**

$$\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = \begin{bmatrix} \frac{\partial f(\boldsymbol{x})}{\partial x_1} \\ \frac{\partial f(\boldsymbol{x})}{\partial x_2} \\ \vdots \\ \frac{\partial f(\boldsymbol{x})}{\partial x_n} \end{bmatrix} \in \mathbb{R}^{n \times 1}. \tag{1.28}$$

Since $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$, often written as just $\nabla f(\boldsymbol{x})$, is column vector, $A \nabla f(\boldsymbol{x})$ is a valid matrix-vector product (when $A \in \mathbb{R}^{m \times n}$).

⋆ Example 2: Optimization of an Unconstrained Function

We want to solve the following problem

$$\min_{x,y} (a - x)^2 + b(y - x^2)^2. \tag{1.29}$$

This is called the Rosenbrock function, and it can be very hard for numerical optimization tools to solve (in high dimensions, there are many solutions). First, we take the gradient and set it equal to 0:

$$\nabla (a - x)^2 + b(y - x^2)^2 = \begin{bmatrix} \frac{\partial f(x,y)}{\partial x} \\ \frac{\partial f(x,y)}{\partial y} \end{bmatrix} = \begin{bmatrix} -2(a - x) - 4b(y - x^2)x \\ 2b(y - x^2) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{1.30}$$

If we set $x = a$, we kill of the $a - x$ term. If we then set $y = a^2$, then we kill off both other terms! Nice. Thus, the **min** is equal to 0, and the **argmin** is equal to $(x, y) = (a, a^2)$.

Now, let's practice **unconstrained optimization** in a slightly more complicated context. In the following example, we re-derive the matrix-vector projection solution.

⋆ Example 3: Optimal Projection via Optimization

Reconsider (1.12), which seeks to find the value $\boldsymbol{x}^*$ such that the normed difference between $A\boldsymbol{x}$ and $\boldsymbol{y}$ in minimized. If A is square and invertible, the solution is clearly just $\boldsymbol{x}^* = A^{-1}\boldsymbol{y}$. However, this is generally not the case. Let's assume $A \in \mathbb{R}^{m \times n}$. Then, we need to solve

$$\boldsymbol{x}^* = \arg\min_{\boldsymbol{x}} \ \|\boldsymbol{y} - A\boldsymbol{x}\|_2^2 . \tag{1.31}$$

To begin, let's expand the 2-norm:

$$\begin{aligned} \|\boldsymbol{y} - A\boldsymbol{x}\|_2^2 &= (\boldsymbol{y} - A\boldsymbol{x})^T (\boldsymbol{y} - A\boldsymbol{x}) && \text{(1.32a)} \\ &= \boldsymbol{y}^T\boldsymbol{y} + (A\boldsymbol{x})^T (A\boldsymbol{x}) - (A\boldsymbol{x})^T\boldsymbol{y} - \boldsymbol{y}^T (A\boldsymbol{x}) && \text{(1.32b)} \\ &= \boldsymbol{y}^T\boldsymbol{y} + \boldsymbol{x}^T A^T A\boldsymbol{x} - 2\boldsymbol{y}^T A\boldsymbol{x}. && \text{(1.32c)} \end{aligned}$$

In this expansion, we have used the fact that $(A\boldsymbol{x})^T\boldsymbol{y} = \boldsymbol{y}^T(A\boldsymbol{x})$. Why is this true? The results of $\boldsymbol{y}^T(A\boldsymbol{x})$ yields a scalar, let's call it α, and we know that $\alpha = \alpha^T$ (i.e.,$1 = 1^T$). Thus:

$$\boldsymbol{y}^T(A\boldsymbol{x}) = \alpha = \alpha^T = \left(\boldsymbol{y}^T(A\boldsymbol{x})\right) = (A\boldsymbol{x})^T\boldsymbol{y} \quad \checkmark \tag{1.33}$$

Anyways, let's take the **gradient** of (1.32c):

$$\nabla_x \|\boldsymbol{y} - A\boldsymbol{x}\|_2^2 = 2A^TA\boldsymbol{x} - 2A^T\boldsymbol{y}. \tag{1.34}$$

What the heck just happened? Let's break it down:

1. $\boldsymbol{y}^T\boldsymbol{y}$ does not depend on $\boldsymbol{x}$, so $\frac{\partial}{\partial x}\boldsymbol{y}^T\boldsymbol{y} = \boldsymbol{0}$.
2. $2\boldsymbol{y}^TA\boldsymbol{x}$ is like $\boldsymbol{v}^T\boldsymbol{x}$, so its gradient is $\boldsymbol{v}$ (just stand it up). Thus, $2A^T\boldsymbol{y} = (2\boldsymbol{y}^TA)^T$.
3. The $\boldsymbol{x}^TA^TA\boldsymbol{x}$ term is quadratic, and thus, a bit trickier. To simplify, let's define $\Gamma \triangleq A^TA$. Next, we apply the **product rule** to $\boldsymbol{x}^T(\Gamma\boldsymbol{x})$, where $\boldsymbol{x}^T$ is the first function, and $(\Gamma\boldsymbol{x})$ is the second function

$$\begin{aligned}
\nabla_{\boldsymbol{x}}\boldsymbol{x}^T(\Gamma\boldsymbol{x}) &= \nabla_{\boldsymbol{x}}\boldsymbol{x}^T\underbrace{(\Gamma\boldsymbol{x})}_{\text{frozen}} + \nabla_{\boldsymbol{x}}\underbrace{\boldsymbol{x}^T}_{\text{frozen}}(\Gamma\boldsymbol{x}) && (1.35a)\\
&= \Gamma\boldsymbol{x} + \left(\boldsymbol{x}^T\Gamma\right)^T && (1.35b)\\
&= \Gamma\boldsymbol{x} + \Gamma^T\boldsymbol{x} && (1.35c)\\
&= \left(A^TA + \left(A^TA\right)^T\right)\boldsymbol{x} && (1.35d)\\
&= \left(A^TA + A^TA\right)\boldsymbol{x} && (1.35e)\\
&= 2A^TA\boldsymbol{x}\checkmark && (1.35f)
\end{aligned}$$

Now that we understand the gradient in (1.34), we set it equal to $\boldsymbol{0}$, and then we solve for $\boldsymbol{x}$:

$$\begin{aligned}
2A^TA\boldsymbol{x} - 2A^T\boldsymbol{y} &= \boldsymbol{0} && (1.36)\\
A^TA\boldsymbol{x} &= A^T\boldsymbol{y} && (1.37)\\
\boldsymbol{x}^* &= \left(A^TA\right)^{-1}A^T\boldsymbol{y}, && (1.38)
\end{aligned}$$

where A^TA is generally invertible if it is filled with non-repeating data.

Let's put the matrix gradient result in a box, because it is generally very useful:

Gradient of the Quadratic Form $\boldsymbol{x}^TM\boldsymbol{x}$

The gradient of the quadratic form $\boldsymbol{x}^TM\boldsymbol{x}$ is given by

$$\begin{aligned}
\frac{\partial}{\partial\boldsymbol{x}}\left(\boldsymbol{x}^TM\boldsymbol{x}\right) &= (M + M^T)\boldsymbol{x}: \quad \textbf{General Matrix} && (1.39)\\
&= 2M\boldsymbol{x}: \quad \textbf{Symmetric Matrix } (M = M^T) && (1.40)
\end{aligned}$$

Constrained Optimization. Optimization gets much harder once we add constraints. Con-

straints are typically represented via

$$\min_{\boldsymbol{x}} f(\boldsymbol{x}) \tag{1.41a}$$
$$\text{s.t. } g(\boldsymbol{x}) \leq 0 \tag{1.41b}$$
$$h(\boldsymbol{x}) = 0. \tag{1.41c}$$

We may formulate the **Lagrangian** $\mathcal{L}$ associated with this expression by "dualizing" the constraints:

$$\max_{\mu \geq 0, \lambda} \min_{\boldsymbol{x}} \underbrace{f(\boldsymbol{x}) + \mu^T g(\boldsymbol{x}) + \lambda^T h(\boldsymbol{x})}_{\textbf{Lagrangian}} : \quad \textbf{Lagrange Dual Problem}. \tag{1.42}$$

The optimal solution will sit at a **saddle point**, where both the minimization and maximization operators are satisfied. From a game-theoretic perspective, this saddle point is satisfied when neither opponent (i.e., the minimizer, and the maximizer) wants to deviate from the equilibrium solution[§]. Many more details can be found here [15].

Equality Constrained Optimization Solution Procedure

To solve **equality constrained** optimization problems using duality, we often use the following steps:

1. After canonicalizing the problem (so it looks like (1.41)), *formulate the Lagrangian.*
2. Fix the dual variables. Solve the "inner" minimization problem via calculus.
3. Plug the solution in for the primal variables, and then solve the "outer" maximization.
4. Plug the dual and primal variable values into the original expression.

Here is a trivial example, where we already know the solution.

⋆ Example 4: Minimization of x^2 with equality constraints

Let's solve the following problem:

$$\min x^2 \tag{1.43a}$$
$$\text{s.t. } x - 1 = 0. \tag{1.43b}$$

Clearly, the solution is $x = 1$, but let's follow the steps. Formulating the Lagrangian, we have $\mathcal{L} = x^2 + \lambda(x - 1)$. Setting the gradient with respect to x to 0 yields

$$2x + \lambda = 0. \tag{1.44}$$

[§]The Lagrange dual problem is deeply connected to the Minimax Theorem, which was first proved by John von Neumann. It is a cornerstone of both optimization and game theory. "As far as I can see, there could be no theory of games ... without that theorem ... I thought there was nothing worth publishing until the Minimax Theorem was proved."

Solving this for the primal, we get $x = -\frac{1}{2}\lambda$. Let's plug this back into the Lagrangian!

$$\mathcal{L} = \left(-\frac{1}{2}\lambda\right)^2 + \lambda(-\frac{1}{2}\lambda - 1) \tag{1.45a}$$

$$= \frac{1}{4}\lambda^2 - \frac{1}{2}\lambda^2 - \lambda \tag{1.45b}$$

$$= -\frac{1}{4}\lambda^2 - \lambda. \tag{1.45c}$$

We now take the gradient of this expression and set it to 0:

$$\frac{\partial \mathcal{L}}{\partial \lambda} = -\frac{1}{2}\lambda - 1 = 0. \tag{1.46}$$

Solving this, we have

$$-\frac{1}{2}\lambda = 1 \tag{1.47a}$$

$$\lambda = -2. \tag{1.47b}$$

If we plug the primal and dual solutions into the Lagrangian, we get the final optimal solution:

$$\mathcal{L} = -\frac{1}{4}\lambda^2 - \lambda \tag{1.48a}$$

$$= -\frac{1}{4}(4) + 2 \tag{1.48b}$$

$$= 1, \tag{1.48c}$$

where by substitution, $x^* = 1$ is the primal solution.

Now, you try with a **constrained** optimization problem!

⋆ Homework 1, Problem 1: Distance to Hyperplane via Optimization

We want to re-derive the result from (1.17) using optimization theory. To do so, we re-formulate the projection problem as an optimization problem:

$$\min_{\boldsymbol{x}} \ \|\boldsymbol{x}_p - \boldsymbol{x}\|_2^2 \tag{1.49a}$$

$$\text{s.t. } 0 = \boldsymbol{w}^T\boldsymbol{x} + w_0. \tag{1.49b}$$

This asks: "What is the point $\boldsymbol{x}$ that is both **closest** to $\boldsymbol{x}_p$ and **on** the hyperplane?" To make the problem easier, we use a 2-norm squared in the objective, but the optimal solution $\boldsymbol{x}^*$ will not change. Next, we dualize the problem:

$$d_h^2 = \max_{\lambda}\min_{\boldsymbol{x}} \ (\boldsymbol{x}_p - \boldsymbol{x})^T(\boldsymbol{x}_p - \boldsymbol{x}) + \lambda\left(\boldsymbol{w}^T\boldsymbol{x} + w_0\right) \tag{1.50a}$$

$$= \max_{\lambda}\min_{\boldsymbol{x}} \ \boldsymbol{x}_p^T\boldsymbol{x}_p + \boldsymbol{x}^T\boldsymbol{x} - 2\boldsymbol{x}_p^T\boldsymbol{x} + \lambda\left(\boldsymbol{w}^T\boldsymbol{x} + w_0\right) \tag{1.50b}$$

$$= \max_{\lambda}\min_{\boldsymbol{x}} \ \underbrace{\boldsymbol{x}_p^T\boldsymbol{x}_p + \boldsymbol{x}^T\boldsymbol{x} + \left(\lambda\boldsymbol{w}^T - 2\boldsymbol{x}_p^T\right)\boldsymbol{x} + \lambda w_0}_{\mathcal{L}(\boldsymbol{x},\lambda)} \tag{1.50c}$$

Now, follow the above steps: solve the inner minimization by setting $0 = \nabla_x\mathcal{L}(\boldsymbol{x},\lambda)$, plug the solution in, and solve the outer maximization. **Hint!** $\boldsymbol{a}^T\boldsymbol{b} = \boldsymbol{b}^T\boldsymbol{a}$.

Solution.

(not posted yet)

What about **inequality constrained** optimization problems? The 4-step solution procedure given above is generally only applicable for equality-constrained problems. **Why?** This is because

it is very hard to know if an inequality constraint $g_i(\boldsymbol{x}) \leq 0$ will be **active** (i.e., binding, i.e., $g_i(\boldsymbol{x}) = 0$) or not. To over overcome this problem, **active set methods** [16] do something like this:

1. Ignore all inequality constraints.
2. Solve the equality-constrained optimization problem.
3. Take the solution $\boldsymbol{x}^*$ and plug it into the ignored inequality constraints.
4. If an inequality constraint is violated, add it into the "active set" of constraints as an **equality** constraint.
5. Re-solve with the updated constraint set. If the dual variable μ_i of any added in/equality constraint is negative, remove the associated in/equality constraint from the active set.
6. Rinse and repeat until you have a solution $\boldsymbol{x}^*$ which (i) doesn't violate any neglected inequalities, and (ii) has all non-negative dual variables μ_i for the included inequality constraints.

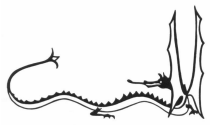
While active set methods are intuitively pleasing, they can be fairly **inefficient**. Barrier-based methods are more effective and more popular these days. We won't use either method in this class, however.

Convexity. This is **not** an optimization class, but the concept of **convexity** will come up many times when we study ML topics. Roughly speaking, an optimization problem is **convex** if a numerical optimization routine can find the globally best solution to the problem without getting "stuck" in a local nook or cranny. Convexity is generally a good thing! Yay convexity!!

Convex Functions and Convex Sets

- **Convex function.** A function $f(x)$ is convex if $f(x) \geq f(x_0) + f'(x_0)(x - x_0)$. In other words, pick on a point on the function manifold and draw a tangent line: every point on this tangent line must lie below the function itself.
- **Convex set.** A set $\mathcal{S}$ is convex if, for every $a \in \mathcal{S}$ and $b \in \mathcal{S}$, then $a\theta + (\theta - 1)b \in \mathcal{S}$, $\forall \theta \in (0, 1)$. In other words, draw a line between any two points in a set: every point on this line must also be in the set $\mathcal{S}$.
- **Convex optimization.** Generally, if we say an optimization problem is **convex**, then it means any solution to the problem will be a global solution. This is nice, because it means that gradient descent cannot get stuck in a local valley: optimization solvers will always find the best possible solution.

An example of a severely **nonconvex** loss function (i.e., the thing we try to minimize when we train a Neural Network) is given in Fig. 5. From this figure, it already should be obvious why nonconvexity is a challenge when training ML models!



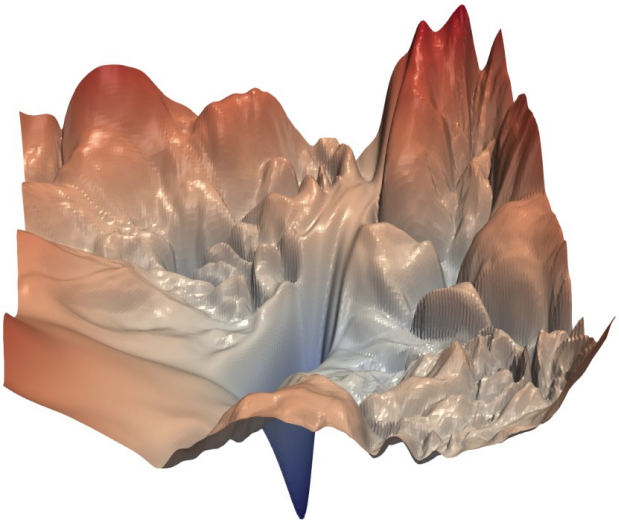


Figure 5: "The loss surfaces of ResNet-56 with skip connections." Reproduced from [17].

2 Learning Theory Basics

As in [1], we denote the set of labeled training data as

$$\mathcal{D} = \{(\boldsymbol{x}_n, \boldsymbol{y}_n) \,|\, n = 1, 2, ...N\} : \quad \textbf{training data} \tag{2.1}$$

where there are N total samples. Notably, $\boldsymbol{x}_n$ and $\boldsymbol{y}_n$ can be of different dimensions (e.g., if 10 input features in $\boldsymbol{x}_n$ map to a single classification in $\boldsymbol{y}_n$):

$$\boldsymbol{x}_n \in \mathcal{X} \subseteq \mathbb{R}^D : \quad \textbf{features} \tag{2.2}$$

$$\boldsymbol{y}_n \in \mathcal{Y} \subseteq \mathbb{R}^C : \quad \textbf{labels}. \tag{2.3}$$

In Machine Learning, we are primarily concerned with building a model $f(\cdot)$ which maps input features to predictions: $\hat{\boldsymbol{y}}_n = f(\boldsymbol{x}_n)$, where $\hat{\boldsymbol{y}}_n$ is a learned prediction. These predictions are always made in a probabilistic sense. Each dimension of $\boldsymbol{x}_n$ represents a single **feature**, and $\mathcal{X} \subseteq \mathbb{R}^D$ represent the **feature space**.

2.1 Data Generating Distributions

To formalize this, we make the assumption that $\boldsymbol{x}_n$, $\boldsymbol{y}_n$ are samples somehow drawn from a **data generating distribution** [2], which we call D. This data generating distribution is generally unknown to the user (you and me), but the set $\mathcal{D}$ can be thought of as its sampled, **and potentially noisy**, approximation. Using the tools of machine learning, our goal is to reconstruct the original data generating distribution D through some surrogate $\hat{D}$.

Given an (x_i, y_i) pair, the data generating distribution D provides a probabilistic mapping (i.e., tells the user, how probabilistic is this mapping?). As explained in [2],

> "A useful way to think about D is that it gives high probability to reasonable (x, y) pairs, and low probability to unreasonable (x, y) pairs. An (x, y) pair can be unreasonable in two ways. First, x might be an unusual input... Second, y might be an unusual rating for the paired x."

⋆ Example 5: Multivariate Uniform Data Generating Distribution

Let's assume (x, y) are scalars and D is a multivariate uniform distribution. x represents the age of toddlers (from 1 to 3, definitionally), and y represents the time of day which they were born (equally distributed over 24 hours). These factors are clearly uncorrelated. In a two-variable uniform distribution, the volume of the probability curve must be equal to 1. Thus, we may solve for the probability p_u:

$$(3)(24)p_u = 1 \tag{2.4}$$

$$p_u = \frac{1}{(3)(24)} \tag{2.5}$$

$$p_u = \frac{1}{72}. \tag{2.6}$$

The resulting uniform probability is given by

$$p(x, y) = \begin{cases} \frac{1}{72}, & x \in \{1, 2, 3\}, y \in \{1, ..., 24\} \\ 0, & \text{otherwise.} \end{cases} \tag{2.7}$$

Thus, the sampled pairs $(x = 1, y = 1)$, $(x = 2, y = 24)$, and $(x = 3, y = 7)$ are all equally

likely to occur in the sampled training set. However, $(x = 10, y = -37)$ has an occurrence probability of 0.

⋆ Homework 1, Problem 2: Multivariate Gaussian Data Generating Distribution

Assume the joint probability of two variables, x and y is given by the following multivariate Gaussian distribution:

$$p(x,y) = \frac{1}{2\pi\sigma_x\sigma_y\sqrt{1-\rho^2}} e^{\left(-\frac{1}{2(1-\rho^2)}\left[\left(\frac{x-\mu_x}{\sigma_x}\right)^2 - 2\rho\left(\frac{x-\mu_x}{\sigma_x}\right)\left(\frac{y-\mu_y}{\sigma_y}\right) + \left(\frac{y-\mu_y}{\sigma_y}\right)^2\right]\right)}, \tag{2.8}$$

where ρ is the correlation between X and Y[a]:

$$\rho = \frac{\sigma_{x,y}}{\sigma_x\sigma_y}, \ \sigma_{x,y} = \text{covariance of } x \text{ and } y. \tag{2.9}$$

(a) Assume there is no correlation: $\rho = 0$. What value pair of x and y maximizes their probability of occurring together? Why? Compute the probability density at this point.

(b) Zero correlation is a bit boring. Provide **two** approximate sketches (or plots, if you insist on using LaTex) of data points that would be generated by this distribution if (i) $\rho \approx +0.999$ and (ii) $\rho \approx -0.999$. Assume $\mu_x = \mu_y = 0$. **Hint**: No calculations are needed here.

[a] Note: X is a random variable, and x is specific value or instantiation of that random variable.

Solution.

(not posted yet)

2.2 Maximum Likelihood Estimation (MLE)

In Machine Learning, we often assume that the samples in $\mathcal{D}$ were independently sampled (i.e., **iid**) from the same data generating distribution. Using these data, we estimate a set of model parameters $\boldsymbol{\theta}$. We define the **likelihood function** as a function which computes the likelihood of the observed data, given the model parameters:

$$p(\mathcal{D}|\boldsymbol{\theta}): \quad \textbf{Likelihood of the observed data, given the model.} \tag{2.10}$$

Thus, the **maximum likelihood estimate (MLE)** is the set of model parameters which maximize the likelihood of the observed data:

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} p(\mathcal{D}|\boldsymbol{\theta}): \quad \textbf{Maximum Likelihood Estimate.} \tag{2.11}$$

Since the N samples in $\mathcal{D}$ are iid, their joint probability is just the product of the individual probabilities:

$$p(\mathcal{D}|\boldsymbol{\theta}) = \prod_{i=1}^{N} p(\boldsymbol{y}_i|\boldsymbol{x}_i, \boldsymbol{\theta}).$$

Why is this the case? Remember your probability theory: if events A and B are independent, then we say $p(A \cap B) = p(B)p(A)$. That is, the probability of both events occurring simultaneously is just equal to the product of each occurring individually. We can think of a data sample $(\boldsymbol{x}_n, \boldsymbol{y}_n)$ being drawn as an "event". Thus, the events of drawing data sample pairs $(\boldsymbol{x}_i, \boldsymbol{y}_i)$ and $(\boldsymbol{x}_j, \boldsymbol{y}_j)$ are independent.

Optimizing over a set of items multiplied together (i.e., xyz) can be numerically challenging (e.g., if we have to multiply many small probabilities together, we will run into floating point precision issues). Thus, we often choose to take the logarithm:

$$\log(zyx) = \log(x) + \log(y) + \log(z). \tag{2.12}$$

We do this for at least four reasons.

1. The log of a set of products yields a set of sums, which is easier to deal with
2. Since probabilities are non-negative, and since the logarithm is a **monotone** function (for non-negative inputs), the MLE solution $\boldsymbol{\theta}^*$ does not change when we wrap a log function around $p(\mathcal{D}|\boldsymbol{\theta})$.
3. If the probability is a Gaussian (which most are), then the log of the Gaussian yields a quadratic expression. How nice!
4. Finally, minimizing the negative log likelihood is equivalent to minimizing the **Kullback Leibler divergence** of the ***predicted*** distribution ($p(\mathcal{D}|\boldsymbol{\theta})$) and ***empirical*** distribution ($p\mathcal{D}(\boldsymbol{x}, \boldsymbol{y})$)) [1]. Essentially, this means the distributions associated with the observed data and the predicted data are optimally close/minimally divergent.

Thus, we take the log likelihood via

$$\ell(\boldsymbol{\theta}) = \log\left(p(\mathcal{D}|\boldsymbol{\theta})\right) \tag{2.13a}$$

$$= \log\left(\prod_{i=1}^{N} p(\boldsymbol{y}_i|\boldsymbol{x}_i, \boldsymbol{\theta})\right) \tag{2.13b}$$

$$= \sum_{i=1}^{N} \log\left(p(\boldsymbol{y}_i|\boldsymbol{x}_i, \boldsymbol{\theta})\right). \tag{2.13c}$$

Finally, since minimization is commonly used over maximization, we can take the **negative log likelihood**, defined as

$$\text{NLL}(\boldsymbol{\theta}) = -\sum_{i=1}^{N} \log\left(p(\boldsymbol{y}_i|\boldsymbol{x}_i, \boldsymbol{\theta})\right) : \quad \textbf{Negative Log Likelihood.} \tag{2.14}$$

The updated MLE is given by

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \; -\sum_{i=1}^{N} \log\left(p(\boldsymbol{y}_i|\boldsymbol{x}_i, \boldsymbol{\theta})\right). \tag{2.15}$$

⋆ Example 6: MLE for a Univariate Gaussian

Let us try to fit a Gaussian to a set of measurements y_1, y_2, ... y_N, which we assume came from a Gaussian with unknown mean and variance:

$$p(y|\mu, \sigma^2) = \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y_i-\mu)^2}{2\sigma^2}}. \tag{2.16}$$

Taking the NLL, we have

$$\begin{aligned}
\text{NLL}\left(p(y|\mu, \sigma^2)\right) &= -\log\left(\prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y_i-\mu)^2}{2\sigma^2}}\right) && \text{(2.17a)} \\
&= -N\log\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) - \sum_{i=1}^{N} \log\left(e^{-\frac{(y_i-\mu)^2}{2\sigma^2}}\right) && \text{(2.17b)} \\
&= \frac{N}{2}\log\left(2\pi\sigma^2\right) + \frac{1}{2\sigma^2}\sum_{i=1}^{N}(y_i-\mu)^2. && \text{(2.17c)}
\end{aligned}$$

Taking this formulation, we may minimize the NLL via:

$$\mu^*, \sigma^{2*} = \arg\min_{\mu,\sigma^2} \; \text{NLL}\left(\mu, \sigma^2\right). \tag{2.18}$$

⋆ Homework 2, Problem 1: Solving the MLE for a Univariate Gaussian

Using (2.17c), solve[a] (2.18) for the optimal values of mean and variance. Show all steps. **Hint**: as usual, take a gradient, and set it equal to 0. Then solve for what you want.

(a) $\mu^* =$

(b) $\sigma^{2*} =$

[a] The solutions are obvious; we care about understanding the steps.

Solution.

(not posted yet)

⋆ Example 7: MLE for Linear Regression

In this example, we assume a linear model is corrupted with zero-mean Gaussian noise:

$$y = \boldsymbol{x}^T \boldsymbol{w} + \epsilon,\ \epsilon \sim \mathcal{N}(0, \sigma^2). \tag{2.19}$$

In the Linear Regression problem, we wish to find the set of weights $\boldsymbol{w}$ which solve the MLE problem, thus maximizing the probability

$$p(y|\boldsymbol{x}, \boldsymbol{w}, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{\left(y - \boldsymbol{x}^T \boldsymbol{w}\right)^2}{2\sigma^2}}. \tag{2.20}$$

Taking the NLL across N data points, we have

$$\text{NLL}\left(\boldsymbol{w}\right) = \frac{N}{2} \log(2\pi\sigma^2) + \sum_{i=1}^{N} \frac{\left(y_i - \boldsymbol{x}_i^T \boldsymbol{w}\right)^2}{2\sigma^2} \tag{2.21}$$

$$\propto \sum_{i=1}^{N} \left(y_i - \boldsymbol{x}_i^T \boldsymbol{w}\right)^2, \tag{2.22}$$

where, in the second step, we removed the additive and multiplicative constants. This is a sum of squares, or least squares, formulation, whose solution we will consider later on.

We may also consider the MLE associated with a **Bernoulli distribution**. As a reminder, this distribution takes a binary input, $x \in \{0, 1\}$, and maps to a discrete probability θ. The associated

probability mass function is given by

$$p(x) = \begin{cases} \theta, & x = 1 \\ 1 - \theta, & x = 0. \end{cases} \tag{2.23}$$

This can be written more compactly via

$$p(x) = \theta^x (1 - \theta)^{1-x}. \tag{2.24}$$

⋆ Example 8: MLE for Bernoulli Distribution

Given an iid sampled set of event occurrences in $\boldsymbol{x}$, their probability is modeled by

$$p(\boldsymbol{x}|\theta) = \prod_{i=1}^{N} p(x_i|\theta). \tag{2.25}$$

Taking the NLL, we have

$$\begin{aligned} -\log\left(p(\boldsymbol{x}|\theta)\right) &= -\sum_{i=1}^{N} \log\left(p(x_i|\theta)\right) && \text{(2.26a)} \\ &= -\sum_{i=1}^{N} \log\left(\theta^{x_i}\left(1-\theta\right)^{(1-x_i)}\right) && \text{(2.26b)} \\ &= -\sum_{i=1}^{N} \log\left(\theta^{x_i}\right) + \log\left(\left(1-\theta\right)^{(1-x_i)}\right) && \text{(2.26c)} \\ &= -\sum_{i=1}^{N} x_i \log\left(\theta\right) + \left(1-x_i\right)\log\left(1-\theta\right), && \text{(2.26d)} \end{aligned}$$

just by the property of logs. Next, we note that the values of x_i are given, i.e., they are just data. We sum over this data:

$$N_1 = \sum_{i=1}^{N} x_i \tag{2.27}$$

$$N_0 = \sum_{i=1}^{N} (1 - x_i) = N - N_1. \tag{2.28}$$

Using these, we update the NLL and take its gradient:

$$\text{NLL}(p(\boldsymbol{x}|\theta)) = -N_1 \log\left(\theta\right) - N_0 \log\left(1-\theta\right) \tag{2.29}$$

$$\partial_\theta \text{NLL} = -N_1 \frac{1}{\theta} + N_0 \frac{1}{1-\theta} \equiv 0. \tag{2.30}$$

Solving for θ, we have

$$N_1 - N_1\theta = N_0\theta \tag{2.31}$$
$$N_1 = (N_0 + N_1)\,\theta \tag{2.32}$$
$$\theta = \frac{N_1}{N_0 + N_1}. \tag{2.33}$$

Thus, the MLE probability of $x = 1$ (i.e., of an event happening) has a trivial solution: just take the ratio of the number of event occurrences over the total number of observations.

2.3 Maximum A Posteriori (MAP) Estimation

MLE methods are susceptible to overfitting, and thus, generalizing poorly. To overcome this problem, we can **regularize** the model with new, useful information. Where does this information come from? Usually, it comes from some **prior** knowledge we have about, e.g., the model parameters. In order to incorporate this information into the problem, we invoke Bayes theorem:

$$p(\boldsymbol{\theta}|\mathcal{D}) = \frac{p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(\mathcal{D})} \tag{2.34}$$
$$p(\boldsymbol{\theta}) : \text{prior knowledge about the model} \tag{2.35}$$
$$p(\mathcal{D}|\boldsymbol{\theta}) : \text{likelihood of the data, given the model} \tag{2.36}$$
$$p(\boldsymbol{\theta}|\mathcal{D}) : \text{posterior distribution of the model} \tag{2.37}$$
$$p(\mathcal{D}) : \text{evidence (i.e., data).} \tag{2.38}$$

If we want to explicitly incorporate the prior information into the estimation problem, we can maximize the **posterior** distribution, which is proportional to the product of the likelihood and prior. As usual, we can minimize the negative log of the posterior to solve this problem:

$$\arg\min_{\boldsymbol{\theta}} -\log p(\boldsymbol{\theta}|\mathcal{D}) = \arg\min_{\boldsymbol{\theta}} -\log\left(p(\mathcal{D}|\boldsymbol{\theta})\right) - \log\left(p(\boldsymbol{\theta})\right). \tag{2.39}$$

This is Maximum A Posteriori (MAP) estimation.

⋆ Example 9: Gaussian Prior

Consider a regression model with unknown parameter θ. Let's assume its uncertainty can be estimated with a Gaussian prior via

$$p(\theta) = \frac{1}{\sqrt{\pi/\lambda}} e^{-\lambda(\theta-\theta_0)^2}. \tag{2.40}$$

In this case, θ_0 is the known mean, and λ is related to our uncertainty (i.e., parameter variance). Taking the negative log, we have

$$-\log(p(\theta)) = -\log\left(\frac{1}{\sqrt{\pi/\lambda}}\right) + \lambda\left(\theta - \theta_0\right)^2 \tag{2.41}$$
$$\propto \lambda\left(\theta - \theta_0\right)^2. \tag{2.42}$$

This shows up at **Tikhonov**, **l2 norm**, or **ridge** regularization (more on this later.

⋆ **Homework 2, Problem 2: Laplace Prior.**

A **Laplace** distribution is given by

$$p(x|\mu, b) = \frac{1}{2b} e^{-\frac{|x-\mu|}{b}}, \tag{2.43}$$

where b is called the Median absolute deviation (MAD). Let's now assume the prior distribution over an unknown model parameter θ is a Laplace distribution, with mean θ_0 and MAD b. What does the negative log likelihood yield? How do you describe this?

Solution.

(not posted yet)

In summary, MLE and MAP are given by

$$\boldsymbol{\theta}^*_{\text{MLE}} = \arg\min - \log(p(\mathcal{D}|\boldsymbol{\theta})) \tag{2.44}$$

$$\boldsymbol{\theta}^*_{\text{MAP}} = \arg\min - \log(p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})). \tag{2.45}$$

In a special case, the solutions are identical (i.e., when the prior has no influence).

MLE and MAP Equivalence

When the prior $p(\theta)$ is **uniform** (e.g., $p(\theta) = 1$),

$$\boldsymbol{\theta}^*_{\text{MLE}} = \boldsymbol{\theta}^*_{\text{MAP}}. \tag{2.46}$$

Proof. When the prior is uniform, the posterior function and the likelihood functions will be proportional to each other:

$$p(\boldsymbol{\theta}|\mathcal{D}) \propto p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta}) \propto p(\mathcal{D}|\boldsymbol{\theta}). \tag{2.47}$$

Thus, they will have the same negative log minimizers: $\boldsymbol{\theta}^*_{\text{MLE}} = \boldsymbol{\theta}^*_{\text{MAP}}$. □

2.4 Regression vs Classification

Within the field of supervised learning, there are generally two sorts of problems: classification problems, and regression problems.

- **Classification.** In classification problems, a model learns to choose the most probable output class from a set of discrete, mutually exclusive classes $\mathcal{Y} = \{1, 2, ..., C\}$. Generally, this set is **unordered**, so there is no reason to believe that class 1 is more similar to class 2 than it is to class 100. Since the classes are not ordered, we need to choose our loss function carefully; cross-entropy loss functions are commonly employed. If there are only two output classes, then the generally multi-class problem becomes a problem of **binary classification**.
- **Regression.** In regression problems, a model produces an output $y \in \mathbb{R}$ which is **continuous**, rather than discrete, in nature. When outputs are continuous, we can use loss functions which directly capture the notation of **distance** (e.g., Mean Square Error), since a predicted output of 2.1 is objectively closer to 2.2 than it is to 3.5. In contrast to classification, it is not necessarily possible to say if a "bird" class if closer to a "skateboard" or a "watering can".

In both classification and regression problems, we seek to learn some model $f(\boldsymbol{x}, \boldsymbol{\theta})$ which serves as a **function approximator**. The "ground truth" function it is approximating could be, e.g., a sin curve in the case of a regression model, or a binary classifier

$$\hat{y} = f_{\text{reg}}(x, \boldsymbol{\theta}),\ \hat{y} \approx \sin(x) \quad : \text{regression "ground truth"} \tag{2.48}$$

$$\hat{y} = f_{\text{cls}}(x, \boldsymbol{\theta}),\ \hat{y} \approx \begin{cases} 1, & x \in \mathcal{R}_1 \\ 0, & x \in \mathcal{R}_2 \end{cases} \quad : \text{classification "ground truth".} \tag{2.49}$$

What, though, is the "ground truth" of a classifier? As we have all learned from stressful CAPTCHA image selection tests, ground truth classification is a bit "pie in the sky". We can hypothesize the existence, however, of an optimal classifier. This is often referred to as a **Bayes Optimal Classifier** (BOC). This is a theoretical, rather than a practical, tool, and it answers the following question: given the training data, what is the most probable label $\hat{y}^*$ associated with a new input x?

$$\hat{y}^* = \arg\max_{\hat{y}} p(\hat{y}|x, \mathcal{D}) : \quad \textbf{Bayes Optimal Classifier}. \tag{2.50}$$

This differs from an MLE or MAP estimator in that it directly computes the most probable prediction $\hat{y}$ **rather than** the most probable *model* that produced the prediction. Thus, the BOC can be thought of as averaging across all possible models, weighted by their posterior probabilities.

2.5 Discriminative vs Generative Classifier Models

Classification models can also be classified as either **discriminative** or **generative**.

- **Discriminative Models.** These models learn the conditional distribution $p(y|X = \boldsymbol{x})$. In order words, given and input, they model the probability distribution over the outputs. Discriminative models do not need to waste computational effort on modeling $\boldsymbol{x}$ the input, since this is always given as an input. This is primarily what we study in this class. Discriminative modeling benefits are nicely enumerated in [1].
- **Generative Models.** These models learn the joint distribution $p(\boldsymbol{x}, y) = p(y)p(\boldsymbol{x}|y)$. Once the full joint distribution is know, it can directly generate new data pairs $(\boldsymbol{x}, y)$ for any set of targets (i.e., outputs). Generative modeling benefits are nicely enumerated in [1].

2.5.1 Naive Bayes Classifiers

The Naive Bayes Classifier is a type of classifier which makes a simple assumption: given an output classification (which is a *condition*), input features are mutually independent:

$$p(x_i|y = c, x_{j\neq i}, \boldsymbol{\theta}) = p(x_i|y = c, \boldsymbol{\theta}). \tag{2.51}$$

For example, x_1, x_2, ... x_n may represent the words in a movie review, and c may represent the classification of the review (positive or negative). If the classification is given $y = c$, then the probability of the the review containing the words "excellent" and "amazing" are independent [2]. The probability of having $y = c$ (which we denote as y_c), **given** the sequence of words $\boldsymbol{x}$, is nicely given by Bayes' rule via

$$p_{\boldsymbol{\theta}}(y_c|\boldsymbol{x}) = \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x}|y_c)p_{\boldsymbol{\theta}}(y_c)}{p_{\boldsymbol{\theta}}(\boldsymbol{x})}, \tag{2.52}$$

but we already know that the features are independent:

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}|y_c) = \prod_{i=1}^{D} p_{\boldsymbol{\theta}}(x_i|y_c). \tag{2.53}$$

Thus, by simplifying Bayes' rule, we may compute

$$\begin{aligned} p_{\boldsymbol{\theta}}(y_c|\boldsymbol{x}) &= \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x}|y_c)p_{\boldsymbol{\theta}}(y_c)}{p_{\boldsymbol{\theta}}(\boldsymbol{x})} && \text{(2.54a)} \\ &\propto p_{\boldsymbol{\theta}}(\boldsymbol{x}|y_c)p_{\boldsymbol{\theta}}(y_c) && \text{(2.54b)} \\ &\propto p_{\boldsymbol{\theta}}(y_c) \prod_{i=1}^{D} p_{\boldsymbol{\theta}}(x_i|y_c), && \text{(2.54c)} \end{aligned}$$

where the "evidence" in the denominator has been omitted. This method is **generative** in the sense that it models $p_{\boldsymbol{\theta}}(x_i|y_c)$ directly, and then uses this to compute $p_{\boldsymbol{\theta}}(y_c|\boldsymbol{x})$ (i.e., rather than predicting this directly). Before we show an example, we recall the Bernoulli distribution and the NLL solution of its MLE (see (2.33)).

⋆ **Example 10: Naive Bayes Classifier**

As in 9.3.1 from [1], assume a Naive Bayes classifier with binary input features (i.e., $x_i \in \{0, 1\}$). A **Bernoulli distribution** will model the probability of an input feature **given** the output prediction y_c. In our naive Bayes problem, the output prediction y_c is a classification, and x_i is a binary variable:

$$p_{\boldsymbol{\theta}}(x_i|y_c) = \theta_{ic}^{x_i}(1 - \theta_{ic})^{(1-x_i)} \tag{2.55}$$

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}|y_c) = \prod_{i=1}^{D} \theta_{ic}^{x_i}(1 - \theta_{ic})^{(1-x_i)}, \tag{2.56}$$

where the model parameter θ_{ic} is the probability of feature $x_i = 1$ given classification y_c, and D is the total number of features.

⋆ **Homework 3, Problem 1: Naive Bayes Classifier Probabilities (HW3!)**

In light of the previous example, lets use a Naive Bayes Classifier to predict image pixels. Let y_c represent the classification of a handwritten digit, taking a value within the set $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Let's say you are given an image matrix $X_i \in \mathbb{R}^{8\times 8}$, and an associated image classification $y_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

(a) Let $X_i^{(jk)} \in \{0, 1\}$ (1 means black, 0 mean white) denote a single pixel of grayscale image i located at position j, k. Using a Naive Bayes classifier with a Bernoulli distribution for the likelihood, what does $\theta_{jk,c}$ represent? What does $p(X_i^{jk}|y_i = c)$ mean?

(b) In an example like this, do you think the naive Bayes assumption is justifiable? Why would X_i^{jk} and $X_i^{(j+1)k}$ have no impact on each other?

(c) Using the results from the MLE Bernoulli distribution (2.33), how can you compute the MLE for $\theta_{jk,c}$? You will code this up in the coding part of the assignment. Assume you have N images X_i, $i \in \{1, 2, ..., N\}$ with classifications y_i. **Hint**: For a given image classification $y_i = c$, you want to count up the number of black pixels vs white pixels at position jk.

(d) Finally, why is the classification model "generative"?

Solution.

(not posted yet)

2.6 Entropy & Cross-Entropy

A primary goal of ML is to learn probabilistic distributions $\hat{D}$ which are optimally similar to the true distributions D used to generate the provided training data $\mathcal{D}$. The **entropy** associated with

a distribution with probability density $p(x)$ is given by

$$\mathbb{H}(p) = -\sum_x p(x)\log(p(x)) \qquad (\textbf{discrete case}) \tag{2.57}$$

$$= -\int p(x)\log(p(x))\mathrm{d}x \quad (\textbf{continuous case}), \tag{2.58}$$

where base 2 is commonly used when dealing with bits, but we will use base e, which assumes a unit of **nats**. Entropy measures the amount of *average* "surprise" in a distribution; this is maximized in **uniform distributions**, and it is minimized in δ function spikes (i.e, when all information is located at a single point).

⋆ Homework 2, Problem 3: Entropy of a Discrete Distribution

Consider the discrete distribution.

$$p(x) = \begin{cases} \frac{1}{4}, & x = 1 \\ \frac{1}{2}, & x = 2 \\ \frac{1}{4}, & x = 3. \end{cases} \tag{2.59}$$

(a) Compute the entropy associated with this distribution.

(b) Set the probabilities associated with $x = 1$ and $x = 3$ to $1/8$. Properly adjust the middle probability, and recompute the entropy. Does this increase or decrease the entropy? Why?

Solution.

(not posted yet)

What is the **intuition** behind the mathematical definition of entropy? It lies in the fact that **suspire** is an inherently additive phenomenon (i.e., when you learn two surprising facts that are uncorrelated, you are generally **twice** as surprised). The log operator maps multiplicative "probability space" into an additive "surprise space", where $-\log(1)$ maps to zero surprise, while $-\log(0)$ maps to infinite surprise. Since $-\log(p(x))$ is surprise, and $p(x)$ is the probability of the surprise, then we can compute the expected value of the surprise as $-\int p(x)\log(p(x))\mathrm{d}x$, which is exactly the definition of entropy provided in (2.58).

Cross-entropy: Given two distributions p and q, we may also compute the **cross-entropy**. This measures the amount of surprise we get when using distribution p to make predictions about distribution q. This is quantified (again, assume base e) via

$$\mathbb{H}_{\text{ce}}(p,q) = -\sum_x p(x)\log(q(x)). \tag{2.60}$$

For our purposes, we can think of $p(x)$ as the distribution of the ground-truth, while $q(x)$ is the distribution of the model we are building. Given $p(x)$, cross-entropy will be **minimized** when $q(x) = p(x)$ (this is related to **Shannon's source coding theorem** [1]). When our derived distribution matches the data generating distribution, cross-entropy is minimized.

⋆ Example 11: Cross-Entropy Minimization

Consider the known distribution $p(x)$ and the unknown distribution $q(x)$:

$$p(x) = \begin{cases} \frac{1}{3}, & x = 1 \\ \frac{2}{3}, & x = 2 \end{cases}, \quad q(x) = \begin{cases} \frac{1}{\gamma}, & x = 1 \\ 1 - \frac{1}{\gamma}, & x = 2. \end{cases} \tag{2.61}$$

We want to show that the cross-entropy will be minimized when $\gamma = 3$. We formulate the cross-entropy function as

$$\mathbb{H}_{\text{ce}}(p, q, \gamma) = -\frac{1}{3}\log\left(\frac{1}{\gamma}\right) - \frac{2}{3}\log\left(1 - \frac{1}{\gamma}\right) \tag{2.62a}$$

$$= -\frac{1}{3}\log\left(\frac{1}{\gamma}\right) - \frac{2}{3}\log\left(\frac{\gamma - 1}{\gamma}\right) \tag{2.62b}$$

$$= -\frac{1}{3}\log(1) + \frac{1}{3}\log(\gamma) - \frac{2}{3}\log(\gamma - 1) + \frac{2}{3}\log(\gamma) \tag{2.62c}$$

$$= -\frac{1}{3}\log(1) - \frac{2}{3}\log(\gamma - 1) + \log(\gamma). \tag{2.62d}$$

We take the gradient, and set it to 0:

$$\frac{\partial}{\partial\gamma}\mathbb{H}_{\text{ce}} = -\frac{2}{3}\frac{1}{\gamma - 1} + \frac{1}{\gamma} = 0. \tag{2.63}$$

Solving this expression, we have

$$\frac{2}{3}\frac{1}{\gamma - 1} = \frac{1}{\gamma} \tag{2.64a}$$

$$\frac{3}{2}(\gamma - 1) = \gamma \tag{2.64b}$$

$$\frac{3}{2}\gamma - \gamma = \frac{3}{2} \tag{2.64c}$$

$$\frac{1}{2}\gamma = \frac{3}{2} \tag{2.64d}$$

$$\gamma = 3. \tag{2.64e}$$

In classification tasks, cross-entropy becomes a key **loss function**. In particular, within classification tasks, the **binary cross-entropy loss function** is given by

$$\mathcal{L}_{\text{bce}}(y, \hat{y}) = \sum_{i=1}^{N} \left(-y_i \log(\hat{y}_i) - (1 - y_i)\log(1 - \hat{y}_i)\right) : \quad \textbf{Binary Cross-Entropy Loss}, \tag{2.65}$$

where y_i is a true label, and $\hat{y}_i$ is a prediction. As we saw in the previous example, cross entropy will be minimized when the training data matches the predictions. We will motivate this loss function more strongly when we study **logistic regression**.

2.7 Logistic and Softmax Functions

The logistic function is given by

$$\sigma(x) \triangleq \frac{1}{1 + e^{-x}} = p : \quad \textbf{Sigmoid (or Logistic) Function}. \tag{2.66}$$

This function maps any real value to between 0 and 1, where these output values are typically interpreted as probabilities. Usefully, the gradient of the sigmoid is given by

$$\frac{\partial}{\partial x}\sigma(x) = \sigma'(x) = \sigma(x)\left(1 - \sigma(x)\right). \tag{2.67}$$

⋆ **Homework 2, Problem 4: Sigmoid Derivative**

Show that $\sigma'(x) = \sigma(x)\,(1 - \sigma(x))$. **Hint**: Use $(1 + e^{-x})^{-1}$, and apply chain rule. Then, tinker. Play around with the math.

Solution.

(not posted yet)

The inverse of the sigmoid is called the **logit**. This maps the probability of an event occurring back to the event itself:

$$\text{logit}(p) \triangleq \log\left(\frac{p}{1-p}\right) = x : \quad \textbf{Logit Function}. \tag{2.68}$$

The integral of the sigmoid is given by the **softplus** function. This function starts from 0, and then climbs to ∞. This function is often used as a smoother version of a ramp function:

$$\sigma_+(x) \triangleq \log\left(1 + e^x\right) : \quad \textbf{Softplus Function}. \tag{2.69}$$

⋆ Homework 2, Problem 5: Logit and Softplus Derivation

Please show the following:

(a) Show that the inverse of the sigmoid function (2.66) yields the logit (2.68).

(b) Show that the derivative of the softplus (2.69) yields the sigmoid function (2.66).

Solution.

(not posted yet)

In **multi-class prediction problems**, we often map various class categories to their classification probabilities. In these cases, we employ the **softmax** function:

$$\mathrm{sm}\left(\boldsymbol{x}\right) = \left[\begin{array}{cccc} \frac{e^{x_1}}{\sum_j e^{x_j}} & \frac{e^{x_2}}{\sum_j e^{x_j}} & \cdots & \frac{e^{x_n}}{\sum_j e^{x_j}} \end{array} \right]^T \tag{2.70}$$

$$\mathrm{sm}_i\left(\boldsymbol{x}\right) = \frac{e^{x_i}}{\sum_j e^{x_j}}. \tag{2.71}$$

Notably, the softmax input is a vector, and all outputs (i.e., probabilities) fall between 0 and 1.

⋆ Homework 2, Problem 6: Softmax Sum

Show the sum of the softmax outputs sum to 1. Why is this necessary, from a probabilistic perspective?

Solution.

(not posted yet)

2.8 The Bias-Variance Conundrum

When we train a ML model, we are initially concerned with minimizing the loss associated with the training data (i.e., does the model make good predictions?). A simple model (with only a few parameters) might return high error, and a more complex model will probably return lower error. As this error drops, we say the **bias** of the model drops. However, what if we retrained these models on new training data? Probably, the simple model would look pretty similar (i.e., the change in model parameter would have a low variance), but the complex model would probably look very different (i.e., the change in model parameter would have a higher variance). This is because the more complex model a model is, the more capacity it has to "over-specialize", or overfit, the training data.

- **Bias:** model error, even if the model is trained on infinite training data (this goes **down** as the model gets more complex)
- **Variance:** model parameter sensitivity to new data (goes **up** as the model gets more complex). This also represents the distance between the learned model and the true model, given finite training data.

Typically, we want to operate at the point of minimizes the sum of model bias and model variance. This will yield a model which both performs well on the training data (low bias) and performs well on future unseen test data (low variance).

2.8.1 Overfitting vs Underfitting

- **Underfitting.** Models with low variance and high bias are generally "underfit", meaning they are **too simple** to capture the complexities in the training data. These models may

generalize relatively well, but have high error. **Solution:** Increase the complexities and representational power of the model.

- **Overfitting.** Models with high variance and tlower bias are generally "overfit", meaning their representational power has overspecialized on the data, essentially memorizing it. These models fit the training data super well, but they generalize poorly. **Solution:** "Regularize" the model (i.e., penalize certain types of complexity), incorporate prior information to guide the training, and utilize **cross-validation** in training.

2.8.2 Occam's Razor

What degree of model complexity is best? Let's assume we have two models which fit some training dataset equally well. **Occam's Razor** says that simpler is better; better in the sense that it will generalize more effectively to unseen data. This principle extends beyond machine learning and is generally helpful . In the context of uncertainties, the simplest explanation is usually the **most likely**. "Simple solutions generalize well."

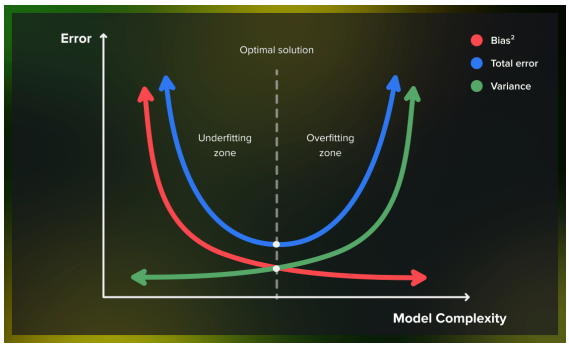


Figure 6: *Bias-Variance Trade-off.*

2.8.3 Model Regularization

Within the field of Machine Learning, regularization generally penalizes complexities (e.g., the size of parameter weights, the number of nonzero terms, the model's sensitivity to data perturbations, etc). We will see more specific regularization strategies in the coming topics. Popular regularization techniques include **Ridge** (or Tikhonov) regularization, **Lasso** regularization, **Elastic Net** regularization, and early stopping.

2.8.4 Inductive Bias

Whether we acknowledge it or not, all models make assumptions. For example,

- **Linear regression** assumes the data can be a fit with a linear model.
- **K-nearest neighbors** assumes that tightly grouped data points should have the same classification, and that all features have the same importance.
- **Weight regularization** methods assume that model parameter should be small.

Some inductive biases are helpful, while others are not. Overall, we should choose models whose inductive biases are **well matched** to the data that we are trying to learn a model for. As stated in CiML [2], inductive biases represent "what we know before the data arrives." If we know nothing, we should be careful about choosing a model with a high degree of inductive bias.

2.8.5 No Free Lunch Theorem

When faced with a range of learning tasks, it maybe tempting to assume there is "**one model to rule them all**", i.e., there is one model that will outperform all other models on all of the tasks. However, this is not true. In 1997, David Wolpert and William Macready showed that, when averaged across all possible problems (or data sets), every model performs **equally well** (i.e., **equally poorly**). This is called the No Free Lunch (NFL) Theorem. Each model has inductive biases, and when performance is averaged across these all data sets, these inductive biases help and

Algorithm 1 Early Stopping

Input: Training $\mathcal{D}^t$ and validation $\mathcal{D}^v$ data
Output: Trained model

1: Initialize: $i = 1$, $e_0 = \infty$, $e_1 = \mathcal{L}(\mathcal{D}^v)$
2: **while** $e_i < e_{i-1}$ (validation error shrinking) **do**
3: $\quad i \leftarrow i + 1$
4: $\quad$ Train the model on $\mathcal{D}^t$ for T **more** steps
5: $\quad$ Evaluate model performance on validation data: $e_i = \mathcal{L}(\mathcal{D}^v)$
6: **end while**
Return: Model from 2nd-to-last iteration

hurt the performance of the model mutually canceling ways. Given a set of data set indices $\mathcal{I}$, a performance metric $p()$, and a set of models (models m "a" and "b"), Sam's statement of the NFL theorem follows:

$$\frac{1}{|\mathcal{I}|}\sum_{i\in\mathcal{I}} p(m^{(\mathrm{b})}(\mathcal{D}_i))) \approx \frac{1}{|\mathcal{I}|}\sum_{i\in\mathcal{I}} p(m^{(\mathrm{a})}(\mathcal{D}_i))) \tag{2.72}$$

The point is this: a model that works well in one domain may perform very poorly in another. When choosing a model architecture and training program to fit some data, **domain knowledge** and **testing** (e.g., cross-validation) should always be used to select the model.

2.8.6 Data-Splitting, Early Stopping, and Cross-Validation

The final tool we present to overcome the bias-variance conundrum is a technique known as **cross-validation**. This helps us answer question like, "how long should I train my model for"? And "What degree polynomial should I use to fit the data?" Given a sampled data set $\mathcal{D}$, we typically **split** this data twice.

- The first split is the typical "80/20" split, where 80% of the data is used for training, and 20% is reserved for testing (i.e., post-processing evaluation). As stated in [2]:

  Never *ever* touch your test data!

- The second split takes the training data, and spits it further; often, the data is split into an actual training subset, and another subset known as the **validation** set. The validation set is used to monitor the performance of the model as it trains on the training data.

After these two splits, the full dataset is now partitioned into something like "70/10/20" = "training/validation/test" buckets, but these numbers can certainly vary [2]. In order to apply **early stopping**, we can use Alg. 1, where the function $\mathcal{L}(\mathcal{D}^v)$ assesses model performance/error on validation data $\mathcal{D}^v$.

Once validation error starts to climb, this is an indication that we are starting to **overfit** the data. Early stopping can be used to determine, either, **how much** we should train a model, or **how strongly** we should regularize a model (more on this later).

Cross-validation: While this simple "70/10/20" split + early stopping can work well, there are several drawbacks. First, we "lose" 10% of our data just to validate. At the same time, only using 10% of the data to validate is a pretty small percentage.

Algorithm 2 Cross Validation (for Length of Model Training)

Input: Data split into K folds: (2.73)
Output: Trained model

1: Initialize: $j = 1$, $e_0 = \infty$, $e_1 = \frac{\infty}{2}$
2: **while** $e_j < e_{j-1}$ (validation error shrinking) **do**
3: $j \leftarrow j + 1$
4: **for** $i \in \{1, 2, ..., K\}$ **do**
5: Train model i on $\mathcal{D}^t \backslash \mathcal{D}_i^t$ for T **more** steps
6: Validate on $\mathcal{D}_i^t$
7: $e_j \leftarrow e_j + \mathcal{L}(\mathcal{D}_i^t)$
8: **end for**
9: $e_j \leftarrow e_j / K$ (take the average across the K folds)
10: **end while**
Return: Model from 2nd-to-last iteration

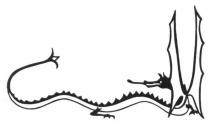
To overcome these problems, we can use a **cross-validation** procedure, where small subsets of the training data are sequentially used for training, then validation, then training, etc. To apply this, we split the training data into K "folds" ($K = 10$ is very common):

$$\mathcal{D}^t = \{\mathcal{D}_1^t, \mathcal{D}_2^t, ...\mathcal{D}_K^t\}. \tag{2.73}$$

Next, we train on $K - 1$ of these data sets, and validate on the one left out. The overall procedure is given in Alg. 2. In this algorithm, line 9 takes an average error across all training evaluations:

$$e = \frac{1}{K} \sum_{i=1}^{K} \mathcal{L}_{\mathcal{D}^t \backslash \mathcal{D}_i^t} \left(\mathcal{D}_i^t \right) : \quad \textbf{Cross Validation Error} \tag{2.74}$$

where $\mathcal{L}_{\mathcal{D}^t \backslash \mathcal{D}_i^t} \left(\mathcal{D}_i^t \right)$ means "the loss function for a model trained on all data, except $\mathcal{D}_i^t$, but evaluated on $\mathcal{D}_i^t$".



3 Unsupervised Learning

Imagine someone comes to you and dumps a bag of data on your desk. "Is this data organized?" Nope. "Is it labeled into correct classes?" Nope. "Well, do I even know what I am looking for?" Nope. Welcome, friends, to the ~~Adams Administration!~~* world of unsupervised learning! As a human being, armed with some basic machine learning tools, you might feel a natural urge to predict one set of features from another set. However, we are getting ahead of ourselves.

In unsupervised learning, we consider a set of sampled data that look like this:

$$\mathcal{D} = \{\boldsymbol{x}_i \,|\, i = 1, 2, ..., N\}: \quad \textbf{Unsupervised Learning Dataset}. \tag{3.1}$$

Where is the labeled ouput predictions? There are none.

3.1 K-Means Clustering

The first unsupervised approach we consider is a clustering approach called **K-Means Clustering**. Consider some i^{th} data point $\boldsymbol{x}_i \in \mathbb{R}^D$, where D is the number of features. This data point is generally multidimensional (e.g., $\boldsymbol{x}_7 = (3.2, 4.4, -6.1)$ in three dimensional coordinate space when $D = 3$). With K-Means clustering, our goal is to find a set of K centers $\boldsymbol{\mu}_k \in \mathbb{R}^D$, such that the total distance from the data points to their assigned cluster centers is minimized. In order to formulate this problem, we introduce selection variable z_i which maps a given data point $\boldsymbol{x}_i$ to its associated center $\boldsymbol{\mu}_k$. For example, assume $\boldsymbol{x}_1$ is associated with cluster 13, and $\boldsymbol{x}_2$ is associated with cluster 6, then

$$i = 1 \rightarrow z_1 = 13 \rightarrow \mu_{z_1} = \mu_{13}: \qquad \text{"data point } i = 1 \text{ is assigned to center } k = 13\text{"} \tag{3.2}$$

$$i = 2 \rightarrow z_2 = 6 \rightarrow \mu_{z_2} = \mu_6 \qquad \text{"data point } i = 2 \text{ is assigned to center } k = 6\text{"} \tag{3.3}$$

Using this notation, we may define a loss function $J_{\text{KM}}(\mu, z)$. This loss function computes the K-Means quality of fit via

$$J_{\text{KM}}(\mu, z) = \sum_{i=1}^{N} \|\boldsymbol{x}_i - \boldsymbol{\mu}_{z_i}\|_2^2 \,. \tag{3.4}$$

That is, it loops over each data point, measures its distance from its assigned center, and sums these distances up. On the surface, we have two "knobs" to tune. First, we may tune the **location** of the centers $\boldsymbol{\mu}_k$, and second, we may tune which center each data point is assigned to via z_i. By tuning the centers and assignments, we may find better or worse K-Means models.

However... given a **set of assignments**, we may actually directly compute the **optimal** center definitions via

$$\boldsymbol{\mu}_k = \frac{1}{N_k} \sum_{i\,:\,z_i = k} \boldsymbol{x}_i. \tag{3.5}$$

Under the sum, "$i : z_i = k$" means: "sum over all indices i, such that $z_i = k$", meaning, only sum data points which are assigned to a given cluster. In other words, (3.5) says that the center is just the average of all associated data points.

⋆ Homework 3, Problem 2: Cluster Center Minimization Solution

*Hamilton joke.

In this problem, we have the following remark (simplified to the scalar case):

Remark 1. *Given a collection of scalar data points $x_1, x_2, x_3 \ldots$ the value of μ which solves*

$$\mu^* = \arg\min_{\mu} \sum_{i=1}^{n} (x_i - \mu)^2$$

is given by $\mu^* = \frac{1}{n}\sum_{i=1}^{n} x_i$.

Proof. **Show this result for HW! Hint:** Take the gradient and set it equal to 0. □

Solution.

(not posted yet)

Thus, for a given set of cluster assignments z_i, we may directly compute the cluster means. Thus, we can compute the K-Means loss function directly from cluster assignments via

$$J_{\mathrm{KM}}(z) = \sum_{i=1}^{N} \left\| \boldsymbol{x}_i - \left(\frac{1}{N_k} \sum_{j\,:\,z_j = z_i} \boldsymbol{x}_j \right) \right\|_2^2 . \tag{3.6}$$

The variables associated with the minimization of this function are discrete integers, meaning this is a very hard problem to solve to global optimality (non-convex, NP-hard). We can use a heuristic algorithm, given in Alg. 3, which starts with some set of centroids, locates their closest points, and then iteratively update the centroids.

Algorithm 3 K-Means Clustering Algorithm

Input: N data points $\boldsymbol{x}_1$, $\boldsymbol{x}_2$... $\boldsymbol{x}_N$
Input: Number of clusters K
Output: N cluster assignments z_1, z_2... z_N

Initialize centroid values $\boldsymbol{\mu}_1$, $\boldsymbol{\mu}_2$... $\boldsymbol{\mu}_K$
Optional: Initialize centroids with K-Means++
while Convergence criteria not met **do**
  for $i \in \{1, 2, ..., N\}$ **do**
    Assign x_i to closest centroid: $z_i = \arg\min_{j \in \{1...K\}} \|\boldsymbol{x}_i - \boldsymbol{\mu}_j\|_2^2$
  end for
  Update all centroids, based on assignments, via (3.5)
  Optional: Compute K-Means loss: (3.4)
end while

⋆ Homework 3, Problem 3: Clustering Algorithm Convergence

In the coding part of this assignment, you will code up Alg. 3 from scratch. Then, you will test three different **convergence criteria** which you described here. For each of the following convergence criteria, describe, using words and variables and equations (if needed), how you would implement each of the following convergence criteria:

(a) Total number of loop iterations

(b) Percent change in loss function

(c) An additional metric that you design (the more creative the better).

Solution.

(not posted yet)

⋆ Homework 3, Problem 4: K-Means Computational Complexity

In this problem, please derive or compute the claim that Alg. 3 runs in $\mathcal{O}(TNK)$, where T is the number of while loop iterations, N is the number of points, and K is the number of centroids.

Solution.

(not posted yet)

3.1.1 Selection of K

One key challenge of the K-Means algorithm is choosing the number of clusters (i.e., big K). A helpful heuristic is the **Silhouette Coefficient**; this measures "how similar an object is to its own cluster (cohesion) compared to other clusters (separation)" [18]. The silhouette coefficient associated with a single data-point i, $\text{sc}(i)$, is given by

$$\text{sc}(i) = \frac{b_i - a_i}{\max\{a_i, b_i\}} : \qquad \textbf{silhouette coefficient} \tag{3.7}$$

$$a_i = \|\boldsymbol{x}_i - \boldsymbol{\mu}_{z_i}\|_2 : \qquad \text{mean distance to other in-cluster points} \tag{3.8}$$

$$b_i = \left\|\boldsymbol{x}_i - \boldsymbol{\mu}'\right\|_2 : \qquad \text{mean distance to other points in next-closest cluster,} \tag{3.9}$$

where k is the centroid index associated with data point i, and $\boldsymbol{\mu}'$ is the **next-closest** centroid.

3.1.2 Computational Complexity of K-Means

Within the theoretical computer science domain, **computational complexity** refers to the number of steps an algorithm requires in order to finish. If an algorithm, with, say, N inputs is $\mathcal{O}(N)$, then we say it takes **on the order of** N steps to solve. Furthermore, say α is some positive constant: curiously, we have that $\mathcal{O}(\alpha \cdot N) = \mathcal{O}(N)$, since we care about dominant orders of magnitude. Assume Alg. 3 iterates T times, with K centroids and N data points. The computational complexity of Alg. 3 is approximately $\mathcal{O}(TNK)$.

3.1.3 K-Means++

In order to improve K-Means performance, K-Means++ performs a smarter initialization. The key idea is to start with a set of centroids μ_k which more optimally "cover" the full dataset, according

to the following steps.

1. We first randomly pick a staring point $\mu_1 = x_n$.
2. Next, we compute the distance from this point to all other points: $D(x_i) = \|x_i - \mu_1\|$. The next centroid is then selected. The probability of a point x_m being selected as the next centroid is proportional to its normalized distance from the first centroid: $p(\mu_2 = x_i) = \frac{D(x_i)}{\sum_{j=1}^{N} D(x_j)}$.
3. As more centroids are added, this process is generalized. Probabilities are computed based on the minimum distance to all previously-selected centroids.

$$p(\mu_t = x_i) = \frac{D_{t-1}(x_i)}{\sum_{j=1}^{N} D_{t-1}(x_j)}, \tag{3.10}$$

where

$$D_{t-1}(x_i) = \min_{k \in \{1 \ldots t-1\}} \|x_i - \mu_k\| . \tag{3.11}$$

4. Using these initializations, continue with standard K-Means.

⋆ Homework 3, Problem 5: K-Means++ Centroid Selection

Let's say you are given a set of positive distances: D_1, D_2, and D_3. Using these, (3.10) computes the **probability** that each associated point x_1, x_2, and x_3, will be selected as the next centroid. You are also given a random number generator, `random.uniform()`, which spits out a value r between 0 and 1. Write a function $f(r, D)$ which takes r and the three distances, and returns the **correct next centroid**. **Hint**: chop up the uniform probability domain! **Another hint:** Here is how your function should look:

$$f(r, D) = \begin{cases} x_1, & r \in \ldots \\ x_2, & r \in \ldots \\ x_3, & r \in \ldots \end{cases} \tag{3.12}$$

Solution.

(not posted yet)

3.2 Principal Component Analysis

Principal Component Analysis (PCA) is a fundamentally important unsupervised learning approach which shows up in many different contexts. It is used, primarily, for dimensionality reduction and for encoding data sets into smaller "latent variable" (i.e., hidden, mathematical variable) dimensions. To understand its basics, we need to first understand the Singular Value Decomposition (SVD).

3.2.1 Singular Value Decomposition

Singular Value Decomposition (SVD) takes a matrix, $A \in \mathbb{R}^{m \times n}$, and decomposes it, similar to how you can decompose a matrix using eigenvalue decomposition. However, matrix A need not be square to apply the SVD. The SVD decomposition is given by

$$A = U\Sigma V^T : \quad \textbf{Singular Value Decomposition} \tag{3.13a}$$

$$U \in \mathbb{R}^{m \times m}, \quad U^T U = UU^T = I_m \quad \text{Left Singular Vectors} \tag{3.13b}$$

$$\Sigma \in \mathbb{R}^{m \times n} \quad \text{Singular Values Matrix} \tag{3.13c}$$

$$V \in \mathbb{R}^{n \times n}, \quad V^T V = VV^T = I_n \quad \text{Right Singular Vectors.} \tag{3.13d}$$

Matrix Σ has generally positive, but always nonnegative, values on the diagonals (starting at (1,1)). These are called **singular values**, and they are very similar to eigenvalues[†]. When $m \neq n$, we note that the matrix (3.13c) is **not a square matrix**, and there will be rows or columns of all zeros. To simplify computation, it is common to define an "economy" version of the SVD:

[†] $\sigma(A) = \lambda(A)$ when A is a normal matrix: $A^T A = AA^T$.

Given the matrix $A \in \mathbb{R}^{m\times n}$, we set $t \triangleq \min(m,n)$. Then, the Economy SVD is given by

$$A = U\Sigma V^T \qquad \textbf{Economy SVD} \tag{3.14a}$$
$$U \in \mathbb{R}^{m\times t}, \quad U^TU = I \qquad \text{Left Singular Vectors} \tag{3.14b}$$
$$\Sigma \in \mathbb{R}^{t\times t} \qquad \text{Singular Values Matrix} \tag{3.14c}$$
$$V \in \mathbb{R}^{n\times t}, \quad V^TV = I \qquad \text{Right Singular Vectors.} \tag{3.14d}$$

The economy version of the SVD is easier to compute and generally much more useful, so it is what we will use most often. When we say SVD, we always mean the **economy SVD**. If we take the SVD of a matrix A, we can decompose the matrix into a sum of n **rank-1** outer products[‡]:

$$A = U\Sigma V^T = \begin{bmatrix} \boldsymbol{u}_1 & \boldsymbol{u}_2 & \cdots & \boldsymbol{u}_m \end{bmatrix} \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \end{bmatrix} \begin{bmatrix} \boldsymbol{v}_1^T \\ \boldsymbol{v}_2^T \\ \vdots \\ \boldsymbol{v}_n^T \end{bmatrix} \tag{3.15a}$$
$$= \begin{bmatrix} \boldsymbol{u}_1 & \boldsymbol{u}_2 & \cdots & \boldsymbol{u}_m \end{bmatrix} \begin{bmatrix} \sigma_1\boldsymbol{v}_1^T \\ \sigma_2\boldsymbol{v}_2^T \\ \vdots \\ \sigma_n\boldsymbol{v}_n^T \end{bmatrix} \tag{3.15b}$$
$$= \sigma_1\boldsymbol{u}_1\boldsymbol{v}_1^T + \sigma_2\boldsymbol{u}_2\boldsymbol{v}_2^T + \cdots + \sigma_n\boldsymbol{u}_n\boldsymbol{v}_n^T \tag{3.15c}$$
$$= \sum_{i=1}^{n} \sigma_i\boldsymbol{u}_i\boldsymbol{v}_i^T. \tag{3.15d}$$

Importantly, the singular values σ_1, σ_2, ..., σ_n are ordered by magnitude:

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n, \tag{3.16}$$

where σ_1 has the most "energy", and σ_n has the least energy. The associated singular vectors, $\boldsymbol{u}_1$,..., $\boldsymbol{u}_t$ and $\boldsymbol{v}_1$,..., $\boldsymbol{v}_t$ form orthonormal bases. That is

$$\boldsymbol{u}_i^T\boldsymbol{u}_j = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases} \quad \text{(left singular vecs)}, \qquad \boldsymbol{v}_i^T\boldsymbol{v}_j = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases} \quad \text{(right singular vecs)}.$$

[‡]An **inner product** of two vectors $\boldsymbol{x}^T\boldsymbol{y}$ yields a scalar. An *outer product* yields a matrix (!): $\boldsymbol{x}\boldsymbol{y}^T$. This matrix is *rank-1*, meaning every column is just a scaled version of the first one.

The SVD and Unitary Matrices

In the standard SVD (3.13), the left and right singular vector matrices are both **square** and **unitary** (or **orthogonal**), meaning the transpose is equal to the inverse:

$$U^T U = UU^T = I_m \tag{3.17}$$
$$V^T V = VV^T = I_n. \tag{3.18}$$

In the economy SVD, we have to **BE CAREFUL**: one of these matrices will **not** be square, and thus, will not be orthogonal. In the **economy** SVD, both matrices will still satisfy

$$U^T U = I \quad \text{(economy svd)} \tag{3.19}$$
$$V^T V = I \quad \text{(economy svd)}. \tag{3.20}$$

However, only one of them will be square and orthogonal, depending on (i) how the data is stacked (rows vs columns – more on this later), and (ii) how many samples vs features there are in the data:

$$t = \min(m, n) \rightarrow \begin{cases} t = m, & UU^T = I,\ VV^T \neq I \\ t = n, & UU^T \neq I,\ VV^T = I \end{cases} \quad \text{(economy svd)}. \tag{3.21}$$

SVD Interpretation: We can think of the SVD as a **recipe** for a matrix. However, let's not think of a matrix. Let's think of **data**. In particular, let the column vector $\boldsymbol{x}$ represent some set of features (maybe, the movie preferences of a single person across many different categories or movie rankings). Let's collect many of these profiles, for you, for me, for Bob, for n people, and then we stack them up together into a matrix:

$$A = [\boldsymbol{x}_{\text{you}}\ \boldsymbol{x}_{\text{me}}\ \boldsymbol{x}_{\text{Bob}}\ \cdots\ \boldsymbol{x}_{\boldsymbol{n}}]$$

If we apply and SVD to this data matrix,

- U is the general **shape** of the columns, with $\boldsymbol{u}_1$ being the dominant shape
- σ is like the **energy** which each of these shapes have. Dominant shapes have more energy!
- V is the **ratio** which you combine these scaled shapes to reproduce the original data. This is like a recipe!

The SVD has many uses, but one of its uses is to find the **dominant shapes**, and their associated energies, for a given set of data (i.e, a matrix). Given the orthonormality of U and V, we can use the SVD as a numerically robust way to solve overdetermined linear systems.

⋆ **Homework 4, Problem 1: Linear System Solution Using the SVD**

Recall our solution to the optimal project problem (also known as a linear least squares, as we will see in later topics) (1.31):

$$\boldsymbol{x}^* = \left(A^T A\right)^{-1} A^T \boldsymbol{y}. \tag{3.22}$$

Please compute the solution for $\boldsymbol{x}^*$ in terms of the economy SVD, assuming $m > n$. There should only be four terms in the solution, including $\boldsymbol{y}$. How can you describe this solution? Do you like it? Why is the $m > n$ assumption needed? **Hint**: Take the definition from (3.14a),

and “plug it in” to (3.22). Then simplify.

Solution.

(not posted yet)

SVD vs PCA: Data Stacking Conventions

Important Note! The interpretation of the SVD and the PCA depend strongly on how the data is structured (i.e., is a data observation vector a row vector, or a column vector?).

1. **SVD data.** This data is usually organized into **columns**:

$$X_{\text{svd}} = \begin{bmatrix} | & | & & | \\ \boldsymbol{x}_1 & \boldsymbol{x}_2 & \cdots & \boldsymbol{x}_n \\ | & | & & | \end{bmatrix}. \tag{3.23}$$

2. **PCA data.** This data is usually organized into **rows**:

$$X_{\text{pca}} = \begin{bmatrix} - & \boldsymbol{x}_1^T & - \\ - & \boldsymbol{x}_2^T & - \\ & \vdots & \\ - & \boldsymbol{x}_n^T & - \end{bmatrix}. \tag{3.24}$$

⋆ **Homework 4, Problem 2: SVD Transpose**

Assume X_{svd} and X_{pca} are the same data matrices (just, transposed). Take the (economy) SVD of both. What is the relationship between the singular values of these data matrices? How about the singular vectors?

Solution.

(not posted yet)

3.2.2 Low Rank Approximation and the Eckart–Young Theorem

The **Eckart–Young** (or, more properly, Eckart-Young-Mirsky) Theorem is amazing. It answers the following question: what is the best low-rank (i.e., rank k) approximation for a given matrix? There is actually a solution to this question. To understand this question, let's define A_k as a **low rank approximation** of A:

$$A = \sum_{i=1}^{n} \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^T \tag{3.25}$$

$$A_k = \sum_{i=1}^{k} \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^T, \quad k \leq n. \tag{3.26}$$

We observe that A_k is **rank** k (meaning its **nullspace** is $n-k$). We can see this by considering, e.g., a rank-2 matrix:

$$A_2 = \sigma_1 \underbrace{\boldsymbol{u}_1 \boldsymbol{v}_1^T}_{\text{rank 1}} + \sigma_2 \underbrace{\boldsymbol{u}_2 \boldsymbol{v}_2^T}_{\text{rank 1}}. \tag{3.27}$$

Since A_2 is written as the sum of two rank 1 matrices, and since these two matrices have orthogonal **spans**[§], it is necessarily of rank 2.

We must also define a **matrix norm**. This is not the same as a vector norm.. A matrix norm considers the matrix as a **mapping**, and it asks, **what is the largest possible mapping this matrix can provide?** We can measure a mapping by taking a matrix-vector product, and then taking a vector norm: $\max_{\boldsymbol{x}} \|A\boldsymbol{x}\|_p$. But wait.. this can explode, if we just choose larger and larger values of $\boldsymbol{x}$! So, let's normalize the solution based on the **size** of the chosen vector:

$$\|A\|_p \triangleq \max_{\boldsymbol{x}\neq 0} \frac{\|A\boldsymbol{x}\|_p}{\|\boldsymbol{x}\|_p} : \quad \textbf{Matrix Norm.} \tag{3.28}$$

The matrix 2 norm, for example can be defined as

$$\|A\|_2 = \max_{\boldsymbol{x}\neq 0} \frac{\|A\boldsymbol{x}\|_2}{\|\boldsymbol{x}\|_2} \tag{3.29a}$$

$$= \max_{\|\boldsymbol{x}\|_2=1} \|A\boldsymbol{x}\|_2 \tag{3.29b}$$

$$= \sigma_{\max}(A)\,. \tag{3.29c}$$

We will not derive this, but it's true (just choose $\boldsymbol{x}$ so coincide with the largest singular vector). So, if we know the largest singular **value** of a matrix, we know its two norm! Using this machinery, we can now present Eckart–Young.

Remark 2 (Eckart–Young Theorem)**.** *Let $A \in \mathbb{R}^{m\times n}$ and let A_k be its low-rank approximation via (3.26). For any matrix $B \in \mathbb{R}^{m\times n}$ of rank k,*

$$\|A - A_k\|_2 \leq \|A - B\|_2\,. \tag{3.30}$$

In other words, A_k is the best possible low-rank (rank k) approximation of A.

We present the proof below, similar to the one presented in [4], which uses contradiction to prove that a better approximation **cannot** exit. This proof uses the concept the **nullspace** (or **kernel**) of a matrix. As a reminder the nullspace is just the space of vectors which satisfy $\mathbf{0} = A\boldsymbol{x}$.

$$\mathcal{N}(A) = \{\boldsymbol{x} \,|\, A\boldsymbol{x} = \mathbf{0},\, \boldsymbol{x} \neq \mathbf{0}\} : \quad \textbf{Matrix Nullspace}. \tag{3.31}$$

The dimension of the nullspace, $\dim(\mathcal{N}(A))$, is just the largest number of **linearly independent** vectors which satisfy $A\boldsymbol{x} = \mathbf{0}$.

[§]We know this because the columns of these rank 1 matrices are orthogonal to each other. Consider this: $\left(\boldsymbol{u}_1\boldsymbol{v}_1^T\right)^T\left(\boldsymbol{u}_2\boldsymbol{v}_2^T\right) = \boldsymbol{v}_1\boldsymbol{u}_1^T\boldsymbol{u}_2\boldsymbol{v}_2^T = \boldsymbol{v}_1 0\boldsymbol{v}_2^T = \mathbf{0}$.

Proof (Eckart–Young Theorem). **First**, we consider the matrix 2-norm of the difference between A and its low-rank approximation A_k:

$$\|A - A_k\|_2 = \left\| \sum_{i=1}^{n} \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^T - \sum_{i=1}^{k} \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^T \right\|_2 \tag{3.32a}$$

$$= \left\| \sum_{i=k+1}^{n} \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^T \right\|_2 \tag{3.32b}$$

$$= \sigma_{k+1}, \tag{3.32c}$$

since σ_{k+1} is the largest remaining singular value. **Second**, since there are $k+1$ orthonormal singular vectors $\boldsymbol{v}_i$ associated with the first $k+1$ singular values, we know there is a $k+1$ dimensional subspace where

$$\|A\boldsymbol{x}\|_2 \geq \sigma_{k+1}, \quad \|\boldsymbol{x}\|_2 = 1. \tag{3.33}$$

We can show this by setting $\boldsymbol{x} = \boldsymbol{v}_i$:

$$\|A\boldsymbol{x}\|_2 = \left\|U\Sigma V^T \boldsymbol{v}_i\right\|_2 \tag{3.34a}$$

$$= \|U\Sigma \hat{e}_i\|_2 \tag{3.34b}$$

$$= \|U\sigma_i \hat{e}_i\|_2 \tag{3.34c}$$

$$= \sigma_i \|\boldsymbol{u}_i\|_2 \tag{3.34d}$$

$$= \sigma_i, \tag{3.34e}$$

where $\hat{e}_i$ is a unit vector with a 1 at position i. **Third**, and finally, we now ask: can there exist some matrix B, $\text{rank}(B) = k$, such that $\|A - B\|_2 < \sigma_{k+1}$? In other words, can there be some matrix B which is an even better approximation of A than A_k? Let's assume there is. We know that $B \in \mathbb{R}^{m \times n}$ (i.e., same size as A), and $\text{rank}(B) = k$. Therefore, the **nullspace** of B, $\mathcal{N}(B)$, has dimension of $n - k$. Let's pick a vector from this nullspace, and multiply it by the matrix $A - B$:

$$\|(A - B)\boldsymbol{x}\|_2 = \|A\boldsymbol{x} - B\boldsymbol{x}\|_2 = \|A\boldsymbol{x}\|_2, \quad \boldsymbol{x} \in \mathcal{N}(B). \tag{3.35}$$

Using the **Cauchy–Schwarz inequality**, we **also** have

$$\|(A - B)\boldsymbol{x}\|_2 \leq \|(A - B)\|_2 \|\boldsymbol{x}\|_2 \tag{3.36a}$$

$$< \sigma_{k+1} \|\boldsymbol{x}\|_2. \tag{3.36b}$$

Putting these together by equating the RHS of (3.35) and (3.36b), we have $\|A\boldsymbol{x}\|_2 < \sigma_{k+1} \|\boldsymbol{x}\|_2, \boldsymbol{x} \in \mathcal{N}(B)$.

$$\|A\boldsymbol{x}\|_2 : \begin{cases} < \sigma_{k+1}, & \boldsymbol{x} \in \mathcal{N}(B) \qquad (n-k \text{ dimensions}) \\ \geq \sigma_{k+1}, & \boldsymbol{x} \in \text{span}(V_{K+1}) \qquad (k+1 \text{ dimensions}). \end{cases} \tag{3.37}$$

If we add these dimensions up, we have an $n - k + k + 1 = n + 1$ dimensional space. This violates the **rank-nullity theorem** (i.e., a rank+nullspace $= n$), so B cannot exist (i.e., it cannot be a better rank-k approximation of A than A_k). □

⋆ **Homework 4, Problem 3: Best Low-Rank Approximation**

You are given matrix A and its SVD.

(a) Using the singular vectors and singular values, write down the best rank-3 approximation of this matrix. $A_3 = ?$

(b) Let's assume the singular values capture the "energy" associated with the data modes/shapes. How much "energy", as a percentage of the full energy, does this rank-3 approximation capture. **Hint**: think.

Solution.

(not posted yet)

3.2.3 Principal Component Analysis & the SVD

Principal Component Analysis (PCA) is very similar in spirit to the SVD: it seeks to take some high dimensional dataset and project it down onto lower dimensions. Typically, the PCA deals with the de-centered **covariance matrix** associated with some dataset. Recall the definitions of variance and covariance (for scalar random variables):

$$\sigma^2 = E\left[(x - x_0)^2\right] \tag{3.38}$$

$$\sigma_{xy} = E\left[(x - x_0)(y - y_0)\right] \tag{3.39}$$

In the same spirit, the sampled covariance matrix, Γ, is constructed via

$$\Gamma = \frac{1}{n-1}(X - X_0)^T (X - X_0), \tag{3.40}$$

where $\Gamma_{i,j}$ the represents the covariance of features i and j, based on all data samples. The PCA is then used to identify **directions of maximum variance**, which are called the principal components of the data, as depicted in Fig. 7.

In large datasets, some dimensions are very important, but other dimensions are very unimportant (and potentially, just noise). PCA helps us discover which dimensions are most important, and it also offers a very useful **encoding/decoding** algorithm. When we apply dimensionality reduction to some data, we do two central things:

1. First, we **encode** the data into some latent space: $\boldsymbol{z} = V^T\boldsymbol{x}$, where $\boldsymbol{x}$ is some data, V^T is the PCA transformation, and $\boldsymbol{z}$ is a compressed, "latent" representation of the data.
2. Second, we **drop** some of the data. There is no free lunch folks: if we want to compress data, in general, we need to lose something. Our assumption, though, is that the thing we drop is not important enough to keep.

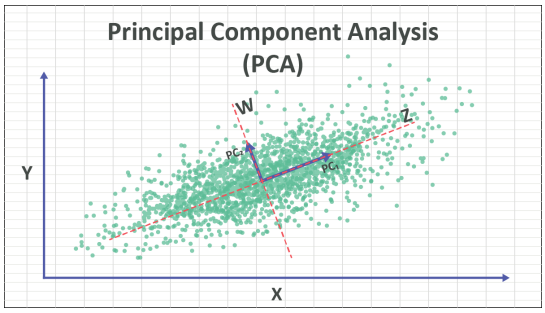


Figure 7: Principal Components (NUMXL)

After we **encode** and **reduce** the data, we can then **decode** the data. Decoding applies a reverse transformation. Note: this is not a *lossless* decoding. Through encoding, something was lost that we can never, ever get back (unless our transformation is fully invertible, but that defeats the point of dimensionality reduction). From the encoding/decoding perspective, we can define

$$\boldsymbol{z} = V^T\boldsymbol{x}: \quad \textbf{encode} \tag{3.41}$$

$$\hat{\boldsymbol{x}} = VV^T\boldsymbol{x}: \quad \textbf{decode}, \tag{3.42}$$

We then ask, what is the best V which minimizes the encoding/decoding, or ***reconstruction***, error:

$$\min_{V\in\mathbb{R}^{m\times k}} \left\|X - VV^TX\right\|. \tag{3.43}$$

In this problem, X is a data matrix, and $V \in \mathbb{R}^{m\times k}$, where k is the dimension of the compressed latent variable $\boldsymbol{z}$. We can notice a few things about this formulation. First, the columns of V should align with the directions of maximal variance of the data (think Fig. 7). Second, these columns should be orthogonal to each other. Why? If they are not orthogonal, then they won't be maximally aligned with the directions of maximal variance. Thus, we may embed this assumption via

$$\min_{V\in\mathbb{R}^{m\times k},\, V^TV=I} \left\|X - VV^TX\right\|. \tag{3.44}$$

It turns out that the **solution** to (3.44) coincides with taking the first k singular vectors (left or right, they are the same) from the SVD of the covariance matrix (3.40).

⋆ **Homework 4, Problem 4: SVD of a Covariance Matrix**

Show that the left and right singular vectors of the covariance matrix Γ are the same. **Hint**: define the SVD of $X - X_0$ first, then plug this in to the definition of Γ.

Solution.

(not posted yet)

Actually computing the PCA: Despite all this **talk** of covariance matrices, we don't have to actually compute the covariance matrix[¶], nor its SVD, to apply the PCA. This is because the right singular vectors of the covariance matrix, and the right singular vectors of the de-centered data matrix are **the same**. Why? (See previous homework problem.) In contrast to a direct application of the SVD to a data matrix, however, the PCA operates on the de-centered data matrix associated with this data. To **compute** the PCA of a matrix using the SVD, we can take the following steps:

1. First, in contrast to the SVD convention, stack your data in vertical rows via (3.24):

$$X = \begin{bmatrix} \boldsymbol{x}_1^T \\ \vdots \\ \boldsymbol{x}_n^T \end{bmatrix}. \tag{3.45}$$

2. Next, de-center the data (i.e., subtract off the mean **of every feature**):

$$\bar{X} = X - X_0 = X - \mathbf{1}\left(\frac{\mathbf{1}^T X}{n}\right). \tag{3.46}$$

3. Next, take the SVD: $\bar{X} = U\Sigma V^T$

[¶]Actually, if you have a really big, diverse dataset, building the covariance matrix can be a very bad idea, because you tend to lose numerical precision, but it depends.

4. The **principal components** are given by the columns of V (keep as many as you like)
5. Data encodings/projections are given by $\bar{X}V$.
6. Data decodings/reconstructions are given by $\bar{X}VV^T$
7. Given a **new** sample (i.e., data not used to build the PCA transformation), subtract the sample mean ($\mathbf{1}^T X/n$), and ***then*** compute $\hat{\boldsymbol{x}}^T_{\text{new}} = \boldsymbol{x}^T_{\text{new}} VV^T$, which is the reconstructed sample.

Once we run PCA and we compute a the principal component vectors $\boldsymbol{v}_1$, $\boldsymbol{v}_2$, ..., $\boldsymbol{v}_k$, the reconstruction error associated with data vector $\boldsymbol{x}$ is given by

$$e = \left\|\boldsymbol{x} - VV^T\boldsymbol{x}\right\|_2 : \quad \textbf{PCA reconstruction error (data = column vector)} \tag{3.47a}$$

$$= \left\|\boldsymbol{x}^T - \boldsymbol{x}^T VV^T\right\|_2 : \quad \textbf{PCA reconstruction error (data = row vector).} \tag{3.47b}$$

⋆ Example 12: PCA with 1 Principal Component

Let's say we are given a zero-mean data matrix, X, and we compute its SVD. Let's now assume $\sigma_1 >> \sigma_2$, so we only want to keep 1 principal component, and throw out the rest. Thus, $\boldsymbol{v}_1$ is the direction of the only principal component we keep. Now, we are given a de-centered sample, $\boldsymbol{x}_s$. Its reconstruction is given by

$$z_s = \boldsymbol{v}_1^T \boldsymbol{x}_s \tag{3.48a}$$

$$\hat{\boldsymbol{x}}_s = z_s \boldsymbol{v}_1 \tag{3.48b}$$

$$= \left(\boldsymbol{v}_1^T \boldsymbol{x}_s\right) \boldsymbol{v}_1. \tag{3.48c}$$

Question: when is the **reconstruction error** zero? Remember, this is the error between $\boldsymbol{x}_s$ and $\hat{\boldsymbol{x}}_s$ and is given by

$$e = \left\|\boldsymbol{x}_s - \boldsymbol{v}_1^T \boldsymbol{x}_s \boldsymbol{v}_1\right\|_2 . \tag{3.49}$$

Let us now set $\boldsymbol{x}_s$ as a vector which points in the same direction as $\boldsymbol{v}_1$. We can think of it as some scaled version of $\boldsymbol{v}_1$, i.e., $\boldsymbol{x}_s = \alpha \boldsymbol{v}_1$. Let's compute the error:

$$e = \left\|\boldsymbol{x}_s - \boldsymbol{v}_1^T \boldsymbol{x}_s \boldsymbol{v}_1\right\|_2 \tag{3.50a}$$

$$= \left\|\alpha \boldsymbol{v}_1 - \boldsymbol{v}_1^T \left(\alpha \boldsymbol{v}_1\right) \boldsymbol{v}_1\right\|_2 \tag{3.50b}$$

$$= \left\|\alpha \boldsymbol{v}_1 - \alpha \boldsymbol{v}_1\right\|_2 \tag{3.50c}$$

$$= 0, \tag{3.50d}$$

since $\boldsymbol{v}_1^T \boldsymbol{v}_1 = 1$.

There are two interesting conclusions from this example:

- First, when a data vector **exactly** points in the direction of a principal component, the reconstruction error is 0 (i.e., the encoding and decoding steps are **lossless**).
- Second, the process of reconstruction is exactly the **rejection** of one vector from another, which is rooted in **projection**. Remember projection? See (1.6) for a refresher. Consider this:

$$\text{proj}_{\boldsymbol{v}_1} \boldsymbol{x}_s = \frac{\boldsymbol{v}_1^T \boldsymbol{x}_s}{\boldsymbol{v}_1^T \boldsymbol{v}_1} \boldsymbol{v}_1 = \boldsymbol{v}_1^T \boldsymbol{x}_s \boldsymbol{v}_1, \tag{3.51}$$

since $\boldsymbol{v}_1^T \boldsymbol{v}_1 = 1$. Therefore, **PCA reconstruction is just the vector project of some data, $\boldsymbol{x}$, onto a principal component $\boldsymbol{v}$.**

Viewing PCA as Vector Projections

Given a de-centered data vector $\boldsymbol{x}$, we may view the application of PCA as the projection of $\boldsymbol{x}$ onto k principal component direction vectors:

$$\begin{aligned}\hat{\boldsymbol{x}} = VV^T\boldsymbol{x} = V\begin{bmatrix}\boldsymbol{v}_1^T\boldsymbol{x}\\ \boldsymbol{v}_2^T\boldsymbol{x}\\ \vdots\\ \boldsymbol{v}_k^T\boldsymbol{x}\end{bmatrix}\\ = \boldsymbol{v}_1\boldsymbol{v}_1^T\boldsymbol{x} + \boldsymbol{v}_2\boldsymbol{v}_2^T\boldsymbol{x} + \cdots + \boldsymbol{v}_k\boldsymbol{v}_k^T\boldsymbol{x}\\ = \text{proj}_{\boldsymbol{v}_1}\boldsymbol{x} + \text{proj}_{\boldsymbol{v}_2}\boldsymbol{x} + \cdots + \text{proj}_{\boldsymbol{v}_k}\boldsymbol{x}\end{aligned}$$

A quick note: It can be *confusing* that sometimes we write the PCA reconstruction as XVV^T, with stacks of data **rows** (3.24), and sometimes, we write it as $VV^T\boldsymbol{x}$, with a column vector of data. However, one is just the transpose of the other, and they are **otherwise the same**. To see this, just take the transpose:

$$\begin{aligned}\left(XVV^T\right)^T &= VV^TX^T &(3.52a)\\ &= VV^T\boldsymbol{x} &(3.52b)\end{aligned}$$

In the previous example, we showed that when a data vector points in the same direction as a **principal component**, the PCA reconstruction error is zero. **Nice!** In the following HW problem, we will generalize this result to higher dimensions. This problem will use the concept of a **span**. Remember, the span of a set of vectors is just the reachable space of linear combinations of those vectors:

$$\text{span}\{\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_k\} = \left\{\sum_{i=1}^{k}\beta_i\boldsymbol{x}_i \,|\, \beta_i \in \mathbb{R}\right\} \tag{3.53}$$

⋆ Homework 4, Problem 5: PCA Reconstruction Error (Higher Dimensions

In this problem, we want to show that the reconstruction error associated with a PCA projection goes to 0 if the incoming data vector lies in the span of a given set of principal components.

Remark 3. *Given a de-centered data sample $\boldsymbol{x}$ and k principal component vectors $\boldsymbol{v}_1$, $\boldsymbol{v}_2$, ..., $\boldsymbol{v}_k$, the PCA reconstruction error is zero if*

$$\boldsymbol{x} \in \text{span}\{\boldsymbol{v}_1, \boldsymbol{v}_2, ..., \boldsymbol{v}_k\}. \tag{3.54}$$

Proof. ***Show this proof for HW!*** **Hint 1:** Remember that the principal component vectors are orthonormal. **Hint 2:** Assume $\boldsymbol{x} \in \text{span}\{\boldsymbol{v}_1, \boldsymbol{v}_2, ..., \boldsymbol{v}_k\}$, and then write $\boldsymbol{x} = \sum_{i=1}^{k}\alpha_i\boldsymbol{v}_i$; now, compute the reconstruction error. □

Solution.

(not posted yet)

3.2.4 PCA Applications: the Netflix Completion Prize, Eigenfaces, Compressed Optimization, and Clustering

In this subsection, we briefly review three, slightly nonstandard examples of how PCA can be useful in practice. The fourth one (clustering) is more standard.

1. PCA Example: Netflix Completion Prize: Years ago, Netflix hosted a series of competitions where competitors were asked to "**fill in**" missing data from a data matrix. What was this missing data? Movie rankings! That is, Netflix wanted to know, if you liked Lord of the Rings, and The Matrix (both of which you saw and ranked highly), how would you rank Jaws (which your account has not seen)? The solution to this problem involves a techniques called low rank matrix completion, which we will not study in this class, but the main idea is that there are a few universal movie ranking profiles (i.e., stereotypical archetypes) which can "explain" most preference combinations.

An interesting related problem is this: let's say you have a data matrix of **fully completed** reviews, and you run PCA. Can you use the result to impute missing values from other users? Yes, you can! First, take the PCA of the data matrix (note: we assume there is **no missing data** in this data matrix):

$$V \leftarrow \text{pca}(X_{\text{movies}}). \tag{3.55}$$

Next, take the new data sample $\boldsymbol{x}$, and separate it into the parts of the data that are known ($\boldsymbol{x}_*$) and unknown/missing ($\boldsymbol{x}_?$):

$$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{x}_* \\ \boldsymbol{x}_? \end{bmatrix}. \tag{3.56}$$

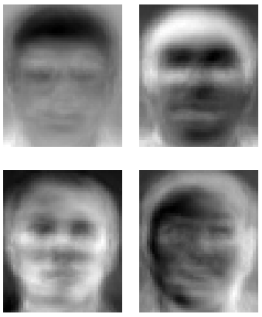


Figure 9: Eigenfaces

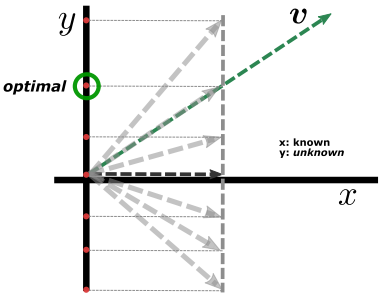


Figure 8: Optimal Imputation

Next, pose the following optimization problem: what is the unknown data $\boldsymbol{x}_?$ which allows the full vector $\boldsymbol{x}$ to project onto the principal components with minimal reconstruction error?

$$\min_{\boldsymbol{x}_?} \left\| \boldsymbol{x} - VV^T \boldsymbol{x} \right\|_2^2 \tag{3.57}$$

This projection is depicted in two dimensions in Fig. (8), where x is known, and y is unknown. The optimal imputed value of y will align the data vector with the given principal component $\boldsymbol{v}$. While this is trivial in two dimensions, it is a nontrivial task in higher dimensions. Details are provided in the Appendix (see (10.1)-(10.6)), but the solution to (3.57) can be computed directly. Defining matrix M via

$$\begin{bmatrix} M_1 & M_2 \\ M_2^T & M_3 \end{bmatrix} = M \triangleq VV^T, \tag{3.58}$$

which is partitioned such that the blocks align with the given and missing data via

$$\text{partition:} \quad \begin{bmatrix} M_1 & M_2 \\ M_2^T & M_3 \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_* \\ \boldsymbol{x}_? \end{bmatrix}, \tag{3.59}$$

the solution to the problem of optimal missing data imputation is given as follows.

Optimal Missing Data Imputation

Given a set of principal components V, and the block decomposition VV^T in (3.58), the **optimal** solution of (3.57) is given by

$$\boldsymbol{x}_? = (I - M_3)^{-1} M_2^T \boldsymbol{x}_*. \tag{3.60}$$

2. PCA Example: Eigenfaces: The "Eigenface" method decomposes a series of face images into their principal statistical components. These eigenvectors, or "eigenfaces", can then be used to (1) generate new images from the linear eigen basis, or (2) perform facial recognition. In order to derive a set of eigenfaces, we can first take a series of images, flatten them, and place them into a data matrix:

$$\boldsymbol{x}_i \leftarrow \text{flatten}\left(X_{\text{face},i}\right), \quad X_{\text{face},i} \in \mathbb{R}^{m \times m}, \boldsymbol{x}_i \in \mathbb{R}^{m^2} \tag{3.61}$$

$$X \leftarrow \begin{bmatrix} \boldsymbol{x}_1 & \boldsymbol{x}_2 & \cdots & \boldsymbol{x}_n \end{bmatrix} \tag{3.62}$$

Applying the SVD to this matrix, $U\Sigma V^T = X - X_0$, we can get the first two eigenfaces by unflattening the first two left singular vectors:

$$U_{\text{eigenface},1} \leftarrow \text{unflatten}\left(U_1\right), \quad U_1 \in \mathbb{R}^{m^2} \tag{3.63}$$

$$U_{\text{eigenface},2} \leftarrow \text{unflatten}\left(U_2\right), \quad U_2 \in \mathbb{R}^{m^2}. \tag{3.64}$$

3. PCA Example: Compressed Optimization: This application is much more specialized, but it's worth mentioning. Let's say you have an engineering system with some state, $\boldsymbol{x} \in \mathbb{R}^m$. In **power system engineering**, this might represent a vector of nodal voltages. Now, take a massive number of these states, collected over time through e.g., sensor data, or simulation study, and run a PCA:

$$V \leftarrow \text{pca}(X_{\text{voltages}}). \tag{3.65}$$

By running a PCA on this data, you are asking, is there a low dimensional space where this data usually lives? Now, take the first K singular vectors from this PCA, where $K << m$, and where m is the dimension of the **original** state. Defining V_K as a matrix with the first K singular vectors, our assumption is that $\boldsymbol{z} = V_K^T \boldsymbol{x}$ is a pretty good latent representation of the state/voltage vector:

$$\boldsymbol{z} = V_K^T \boldsymbol{x} \tag{3.66}$$

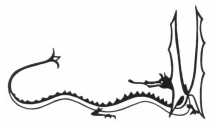
$$\boldsymbol{x} \approx V_K \boldsymbol{z}. \tag{3.67}$$

Now, we can take an optimization, or simulation, problem we care about, and replace $\boldsymbol{x}$ with $V_K \boldsymbol{z}$ (note, you may also have to re-center the data, but this isn't shown):

$$\begin{array}{rl} \min_{\boldsymbol{x}} & c^T \boldsymbol{x} \\ \text{s.t.} & A\boldsymbol{x} \leq b \end{array} \quad \underset{\text{reduce!}}{\rightarrow} \quad \begin{array}{rl} \min_{\boldsymbol{z}} & (c^T V_K)\boldsymbol{z} \\ \text{s.t.} & (AV_K)\boldsymbol{z} \leq b. \end{array} \tag{3.68}$$

This updated optimization problem now has k decision variables instead of m, so it can be easier to solve, and feasibility in the original problem is maintained (if the compressed version is feasible). However, in the resulting solution, $\boldsymbol{x}$ is constrained to live with the **span** of the columns of V_K, which can lead to suboptimal solutions. More details on this sort of approach can be found here [19].

4. PCA Example: Clustering: We can also user PCA to solve a **clustering problem** (or at least, to make it much easier). The process is this: run PCA on a given dataset. Next, project the data into the direction of the principal components. Now, in this latent space, cluster data with e.g., K-Means.



4 Linear Regression

When you need to predict a numerical value, or some set of numerical values, based on a series of known features, linear regression is ***always, always, always*** the first modeling tool you should reach for*. Linear regression is a **model** which estimates the **relationship** between x (an independent input) and y (a dependent output) through a linear prediction model. Remember: a model is *linear* if and only if it satisfies the superposition principle (1.2). In contrast to classification models, which predict discrete classifications, regression models produce **continuous results**. Why should you start with a linear regression model?

- Linear Regression models are, often, "good enough"
- Linear Regression models are easy to train (you can analytically construct the globally optimal regression model with very little effort)
- You quickly get a sense for "how nonlinear" the system which produced your data actually is.

4.1 A Gentle Introduction

Consider a model which estimates the income y of an individual. Income is positively correlated with many predictor variables (class *regressors*), including age, years of education, and occupation (which we assume can be quantified). Given some training data set $\mathcal{D}$, we may wish to build a **linear regression model**:

$$\hat{y} = \boldsymbol{w}^T \boldsymbol{x} \tag{4.1}$$

$$= \underbrace{w_0 x_0}_{x_0=1} + w_1 x_1 + w_2 x_2 + \cdots + w_k x_k : \quad \textbf{Linear Regression Model} \tag{4.2}$$

where $\hat{y}$ is an output prediction, and $w_0 x_0$ represents our bias term. Stacking N inputs and outputs in matrix X and vector $\boldsymbol{y}$ via

$$X = \begin{bmatrix} \boldsymbol{x}_1^T \\ \boldsymbol{x}_2^T \\ \vdots \\ \boldsymbol{x}_N^T \end{bmatrix}, \ \hat{\boldsymbol{y}} = \begin{bmatrix} \hat{y}_1 \\ \vdots \\ \hat{y}_N \end{bmatrix} \tag{4.3}$$

we may write the predictive model via

$$\hat{\boldsymbol{y}} = X\boldsymbol{w} \tag{4.4}$$

Our goal is to choose the trainable parameters $\boldsymbol{w}$ such that some loss function associated with this predictive model is minimized. When it comes to linear regression, mean square error is a commonly applied loss function (we will see why in the next subsection):

$$\text{MSE}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{N} (\boldsymbol{y} - \hat{\boldsymbol{y}})^T (\boldsymbol{y} - \hat{\boldsymbol{y}}) \tag{4.5}$$

$$= \frac{1}{N} \sum_{i=1}^{N} \|y_i - \hat{y}_i\|_2^2 : \quad \textbf{Mean Square Error}. \tag{4.6}$$

*The only exception here might be when you are trying to directly model a system of analytical equations which are **definitely known to be nonlinear**: e.g., $y = \sin(x)$. However, linear regression is still a helpful benchmark, if nothing else, and its effectiveness might surprise you!

4.2 Least Squares "and All His Friends"

There are many other ways to **motivate** linear regression. From a *probabilistic perspective*, we may recall an example from Topic 2, where measurements from a linear system were corrupted with Gaussian noise via (2.19):

$$y = \boldsymbol{x}^T \boldsymbol{w} + \epsilon \tag{4.7a}$$

$$= \underbrace{w_0 x_0}_{x_0=1} + w_1 x_1 + w_2 x_2 + \cdots + w_m x_m + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2). \tag{4.7b}$$

For convenience, we have append a leading 1 to the set of input features in order to easily capture a model bias term, w_0. The likelihood of some data, given this set of model parameters, is

$$p(y|\boldsymbol{x}, \boldsymbol{w}, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{\left(y - \boldsymbol{x}^T \boldsymbol{w}\right)^2}{2\sigma^2}} : \textbf{ Likelihood Function for Linear Regression.} \tag{4.8}$$

This is called **multiple linear regression**, since there are multiple features mapping to a single ouput prediction. Assuming we have N iid samples collected from this model, the joint likelihood is familiarly given by the product of the individual observation likelihoods:

$$p(y|\boldsymbol{x}, \boldsymbol{w}, \sigma^2) = \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{\left(y_i - \boldsymbol{x}_i^T \boldsymbol{w}\right)^2}{2\sigma^2}}. \tag{4.9}$$

When a model has multiple outputs, we call this **multivariate linear regression**, and the associated likelihood function is

$$p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{w}, \sigma^2) = \prod_{j=1}^{J} \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\sigma_j^2}} e^{-\frac{\left(y_{j,i} - \boldsymbol{x}_{j,i}^T \boldsymbol{w}_j\right)^2}{2\sigma_j^2}}. \tag{4.10}$$

The following box summarizes the different types of linear regression models.

Linear Regression Model Names

Linear regression models have different names, depending on the number of model inputs and outputs:

$$y = w_0 + w_1 x_1 : \quad \textbf{Simple Linear Regression} \tag{4.11}$$

$$y = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_m x_m : \quad \textbf{Multiple Linear Regression} \tag{4.12}$$

$$\boldsymbol{y} = \begin{bmatrix} \boldsymbol{w}_1^T \\ \vdots \\ \boldsymbol{w}_n^T \end{bmatrix} \boldsymbol{x} : \quad \textbf{Multivariate Linear Regression.} \tag{4.13}$$

To **optimize** a multiple linear regression likelihood function, like the one in (4.9), means to find the set of model parameters which map to the highest probability of the observed data. As usual, we take the **Negative Log Likelihood**. Recall from (2.21) that the NLL applied (4.9) will yield

$$\text{NLL}\left(p(y|\boldsymbol{x}, \boldsymbol{w}, \sigma^2)\right) = \frac{N}{2} \log(2\pi\sigma^2) + \frac{1}{2\sigma^2} \sum_{i=1}^{N} \left(y_i - \boldsymbol{x}_i^T \boldsymbol{w}\right)^2. \tag{4.14}$$

When we minimize this model, we typically only solve for the model parameters that we care about: $\boldsymbol{w}$. The variance parameter isn't useful for prediction making. In many cases, this parameter is known apriori, anyways (i.e., noise strength is known). Thus, minimizing the negative log likelihood corresponds to solving a problem we call **Ordinary Least Squares** (OLS):

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} \sum_{i=1}^{N} \left(y_i - \boldsymbol{x}_i^T \boldsymbol{w}\right)^2 : \quad \textbf{Ordinary Least Squares}. \tag{4.15}$$

We may rewrite this in more familiar terms using a data matrix X and the ℓ_2 norm:

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} \|\boldsymbol{y} - X\boldsymbol{w}\|_2^2, \qquad X = \begin{bmatrix} \boldsymbol{x}_1^T \\ \boldsymbol{x}_2^T \\ \vdots \\ \boldsymbol{x}_N^T \end{bmatrix}. \tag{4.16}$$

4.3 Weighted Least Squares

So far, we have assumed the variance of each observed data point is **identical**. For example, if a single sensor collects all of the observations, then we can assume that all of the variances are identical. However, in some cases, the variances will **not be equal** (different sensors!). Let us assume the extreme case, where each data point has a separate, but known, variance. Taking the NLL, the **updated minimization problem** will take the form

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} \frac{1}{2} \sum_{i=1}^{N} \frac{\left(y_i - \boldsymbol{x}_i^T \boldsymbol{w}\right)^2}{\sigma_i^2}. \tag{4.17}$$

We can define a useful matrix, Ω, which puts all of the inverted standard deviations on the diagonal:

$$\Omega = \begin{bmatrix} \frac{1}{\sigma_1} & 0 & \cdots & 0 \\ 0 & \frac{1}{\sigma_2} & & \\ \vdots & & \ddots & \\ 0 & & & \frac{1}{\sigma_N} \end{bmatrix} : \quad \textbf{Weight Matrix}. \tag{4.18}$$

Using this matrix, we update weighted minimization problem (4.17):

$$\begin{aligned} \sum_{i=1}^{N} \frac{\left(y_i - \boldsymbol{x}_i^T \boldsymbol{w}\right)^2}{\sigma_i^2} &= \sum_{i=1}^{N} \Omega_{ii}^2 \left(y_i - \boldsymbol{x}_i^T \boldsymbol{w}\right)^2 && \text{(4.19a)} \\ &= \left(\boldsymbol{y} - X\boldsymbol{w}\right)^T \Omega^2 \left(\boldsymbol{y} - X\boldsymbol{w}\right) && \text{(4.19b)} \\ &= \left(\Omega\left(\boldsymbol{y} - X\boldsymbol{w}\right)\right)^T \left(\Omega\left(\boldsymbol{y} - X\boldsymbol{w}\right)\right) && \text{(4.19c)} \\ &= \|\boldsymbol{y} - X\boldsymbol{w}\|_\Omega^2, && \text{(4.19d)} \end{aligned}$$

where this final norm notation is called a "weighted norm":

$$\|\boldsymbol{x}\|_W^2 \triangleq \|W\boldsymbol{x}\|^2 = \left(W\boldsymbol{x}\right)^T \left(W\boldsymbol{x}\right) : \quad \textbf{Weighted Norm}. \tag{4.20}$$

The Weighted Least Squares (WLS) problem weights each data sample proportional to its inverse noise. Samples with **high noise** (bad data) are down-weighted, while samples with **low noise** (good data) are up-weighted.

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} \|\boldsymbol{y} - X\boldsymbol{w}\|_\Omega^2 : \quad \textbf{Weighted Least Squares}. \tag{4.21}$$

Many observation problems (e.g., state estimation) solve a WLS-type problem.

⋆ Homework 5, Problem 1: Weighted Least Squares Solution

If the solution to OLS is $\boldsymbol{w}^* = (X^T X)^{-1} X^T \boldsymbol{y}$, *by analogy*, what is the solution to WLS problem (4.21)? **Hint**: Distribute the weight matrix in the WLS problem, and then think about what the solution must look like.

Solution.

(not posted yet)

4.4 Linear Systems: Square, Overdetermined, and Underdetermined

We have several times seen the solution to the least squares problem (4.15): $\boldsymbol{w}^* = (X^T X)^{-1} X^T \boldsymbol{y}$. However, this solution make a key assumption: $X^T X$ is invertible! This is generally the case when we have a glut of data (that is, we have more data than features). However, it is not always the case. We consider three key situations

Linear System Classification

We can classify a linear system based on the **dimensionality** of the data matrix:

- **Square** System: $X \in \mathbb{R}^{m \times n}$, $m = n$: (data and features have same dimension)
- **Overdetermined** System: $X \in \mathbb{R}^{m \times n}$, $m > n$: (more data than features)
- **Underdetermined** System: $X \in \mathbb{R}^{m \times n}$, $m < n$: (more features than data)

Case 1: Square System. When the data matrix $X \in \mathbb{R}^{n \times n}$, we say the system is "**square**", and we can compute the optimal set of linear regression parameters via

$$\boldsymbol{w}^* = X^{-1}\boldsymbol{y}: \quad \textbf{Square System Parameter Solution}. \tag{4.22}$$

This is commonly solved via LU decomposition [1] of the data matrix.

Case 2: Overdetermined System. In this case, which is the most common one in data science and ML problem, there is "too much" data. Thus, the given set of features will not be able to "explain" all of it, and there will be some error. The resulting solution is given by

$$X^T X\boldsymbol{w}^* = X\boldsymbol{y}: \quad \textbf{Normal Equations} \tag{4.23}$$

$$\boldsymbol{w}^* = (X^T X)^{-1} X^T \boldsymbol{y}: \quad \textbf{Overdetermined System Parameter Solution}. \tag{4.24}$$

Numerically, this solution can computed via SVD or QR decomposition (next subsection).

Case 3: Undetermined System. In some situations, an observation of a system may have many features, but there are only a small number of data samples (i.e., there is not enough data!). In these situations, we actually have an **infinite** number of explanations for the data. Why? Think of is this way: take a single output observation with two input features. Which one "explains" the data best? There is no way of knowing – we need more data! We explore this more in the following problem.

⋆ Homework 5, Problem 2: Underdetermined Linear System: Infinite Solutions

Consider a single output observation, $y = 10$, and a single input feature observation: $x_1 = 2.5$. We seek to build a linear regression model $\hat{y} = w_1 x_1 + w_0$ which explains this data. Thus, we need to solve

$$\{w_0, w_1\} = \underset{\{w_0, w_1\}}{\arg\min} \|y - (w_1 x + w_0)\|_2^2 . \tag{4.25}$$

(a) What is the smallest possible objective value?

(b) Find a numerical solution for w_0 and w_1 which minimizes the objective.

(c) Find another numerical solution which minimizes the objective.

(d) In fact, since this is an underdetermined system, there are an infinite number of solutions which can explain the data! Find them all. Write them as a set. **Hint**: Set the residual to its minimal values, and then write one parameter in terms of the other.

Solution.

(not posted yet)

Since there are an infinite number of solutions in the underdetermined case, **which one do we pick?** Based on Occam's Razor, it can make sense to choose the one with the smallest model parameters; we call this the **least norm** solution.

⋆ Example 13: Least Norm Solution for Underdetermined Linear Systems

The "least norm" solution will minimize the size of the model parameters ($\|\boldsymbol{w}\|$). We may formulate a least norm problem via

$$\min_{\boldsymbol{w}} \ \|\boldsymbol{w}\|_2^2 \tag{4.26a}$$

$$\text{s.t. } \boldsymbol{y} = X\boldsymbol{w}. \tag{4.26b}$$

Why is the equality constraint present? Since the problem is underdetermined, we know we can drive error to 0 and fully explain the data. To solve this **constrained optimization** problem, we use the set of steps we learned back in Topic 1. First, we formulate the Lagrangian $\mathcal{L} = \boldsymbol{w}^T\boldsymbol{w} + \boldsymbol{\lambda}^T(\boldsymbol{y} - X\boldsymbol{w})$ and build the dual:

$$\max_{\boldsymbol{\lambda}} \min_{\boldsymbol{w}} \|\boldsymbol{w}\|_2^2 + \boldsymbol{\lambda}^T(\boldsymbol{y} - X\boldsymbol{w}). \tag{4.27}$$

Next, we minimize over $\boldsymbol{w}$, by setting the gradient to 0:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = 2\boldsymbol{w} - X^T\boldsymbol{\lambda} = \boldsymbol{0}. \tag{4.28}$$

Thus, $\boldsymbol{w} = \frac{1}{2}X^T\boldsymbol{\lambda}$. Plugging this in, we have a maximization problem given by

$$\max_{\boldsymbol{\lambda}} \ \frac{1}{4}\boldsymbol{\lambda}^T XX^T\boldsymbol{\lambda} + \boldsymbol{\lambda}^T\left(\boldsymbol{y} - \frac{1}{2}XX^T\boldsymbol{\lambda}\right). \tag{4.29}$$

Once again, taking a gradient and setting to 0, we have

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\lambda}} = \boldsymbol{y} - \frac{1}{2}XX^T\boldsymbol{\lambda} = \mathbf{0}, \tag{4.30}$$

yielding $\boldsymbol{y} = \frac{1}{2}XX^T\boldsymbol{\lambda}$. Solving for the dual variable vector, we have

$$\boldsymbol{\lambda} = 2\left(XX^T\right)^{-1}\boldsymbol{y}. \tag{4.31}$$

Note! We have taken the inverse of XX^T. Can we do this? Generally, in an underdetermined system, yes, we can! Since it collapses a "short, fat" matrix into a short square matrix, which will generally be invertible. We may now combine the results:

$$\begin{aligned}
\boldsymbol{w} &= \frac{1}{2}X^T\boldsymbol{\lambda} && \text{(4.32a)}\\
&= \frac{1}{2}X^T 2\left(XX^T\right)^{-1}\boldsymbol{y} && \text{(4.32b)}\\
&= X^T\left(XX^T\right)^{-1}\boldsymbol{y}. && \text{(4.32c)}
\end{aligned}$$

Thus, we have an analytical solution for the least norm solution! This solution has employed the so-called right pseudo inverse.

Pseudo Inverse Family!

Given a linear regression "training" problem

$$\min_{\boldsymbol{w}} \ \|\boldsymbol{y} - X\boldsymbol{w}\|, \tag{4.33}$$

the solution strategy, once again, depends on the **dimensionality** of the data matrix X.

- **Standard Inverse**: $\boldsymbol{w} = X^{-1}\boldsymbol{y}$: uniquely eliminates residual error
  ➔ suitable for square systems
- **Left Pseudo Inverse**: $\boldsymbol{w} = \left(X^TX\right)^{-1}X^T\boldsymbol{y}$ minimizes residual
  ➔ suitable for overdetermined systems
- **Right Pseudo Inverse**: $\boldsymbol{w} = X^T\left(XX^T\right)^{-1}\boldsymbol{y}$ eliminates residual error while minimizing the norm of the model parameters
  ➔ suitable for underdetermined systems

4.5 Analytically Solving Least Squares

As shown in the previous homework problem, the SVD can be used to solve a least squares problem. We again consider the problem $\min_{\boldsymbol{w}} \ \|\boldsymbol{y} - X\boldsymbol{w}\|$. Next, we write the following, *kind of weird*,

equation:

$$X\boldsymbol{w} = \boldsymbol{y}: \qquad \text{☞ Impossible.} \tag{4.34}$$

As we have shown, in many cases, this is an impossible equation. It is like writing

$$\alpha \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \end{bmatrix}: \quad \text{No scalar can solve this.}$$

There is no α which satisfies the equality. The expression, therefore, can be interpreted as, "find the parameters $\boldsymbol{w}$ which, as closely as possible, satisfy this expression." Using the SVD of $X \in \mathbb{R}^{m\times n}$, $m > n$, we have

$$U\Sigma V^T\boldsymbol{w} = \boldsymbol{y} \tag{4.35}$$

$$\boldsymbol{w} = V\Sigma^{-1}U^T\boldsymbol{y}: \quad \textbf{SVD Least Squares Solution}. \tag{4.36}$$

This solution is valid when the problem is overdetermined, and thus, $VV^T = I$. Before we move on, let's reconsider our "impossible equation". Plugging in our SVD solution for the parameters,

$$X\boldsymbol{w} = \boldsymbol{y} \tag{4.37a}$$

$$\left(U\Sigma V^T\right)\left(V\Sigma^{-1}U^T\boldsymbol{y}\right) = \boldsymbol{y} \tag{4.37b}$$

$$UU^T\boldsymbol{y} \neq \boldsymbol{y}, \quad \text{🤯} \tag{4.37c}$$

since UU^T is not the identity matrix in the economy SVD with $m > n$. Equality sure ain't what is used to be.

4.5.1 QR Decomposition

There is another, slightly cheaper factorization which is also commonly used to solve, either, poorly condition square systems of equations, or overdetermined least squares systems. This factorization is called the **QR decomposition** (QR does not stand for anything). Vanilla QR is rooted in Gram Schmidt: it takes a matrix, and it orthonormalizes the columns (this is matrix Q). We then write the original matrix columns as a linear sum of the columns of Q (this is matrix R, which is an upper right triangular matrix).

Given the matrix $A \in \mathbb{R}^{m\times n}$, where $m > n$ (more data than features), the reduced, or thin, QR factorization is given by

$$A = QR \qquad \textbf{Reduced QR Factorization} \tag{4.38a}$$

$$Q \in \mathbb{R}^{m\times n}, \quad Q^TQ = I \qquad \text{Orthonormalized Columns} \tag{4.38b}$$

$$R \in \mathbb{R}^{n\times n} \qquad \text{Upper Right Triangular Matrix} \tag{4.38c}$$

Using the properties of the QR decomposition, we may re-solve our impossible equation:

$$QR\boldsymbol{w} = \boldsymbol{y} \tag{4.39a}$$

$$Q^TQR\boldsymbol{w} = Q^T\boldsymbol{y} \tag{4.39b}$$

$$R\boldsymbol{w} = Q^T\boldsymbol{y} \tag{4.39c}$$

$$\boldsymbol{w} = R^{-1}Q^T\boldsymbol{y}: \quad \textbf{QR Least Squares Solution}. \tag{4.39d}$$

Since matrix R is an upper triangular matrix, $R^{-1}\boldsymbol{x}$ can be computed **very efficiently** via the process of back-substitution. This is briefly reviewed in the Appendix (Sec. 10).

To **construct** the QR factorization, we first apply a normalized Gram-Schmidt procedure to the columns of the data matrix X:

$$\boldsymbol{z}_1 = \boldsymbol{x}_1 \qquad \rightarrow \boldsymbol{q}_1 = \boldsymbol{z}_1 / \left\|\boldsymbol{z}_1\right\|_2 \tag{4.40}$$

$$\boldsymbol{z}_2 = \boldsymbol{x}_2 - \text{proj}_{\boldsymbol{z}_1}\boldsymbol{x}_2 \qquad \rightarrow \boldsymbol{q}_2 = \boldsymbol{z}_2 / \left\|\boldsymbol{z}_2\right\|_2 \tag{4.41}$$

$$\boldsymbol{z}_3 = \boldsymbol{x}_3 - \text{proj}_{\boldsymbol{z}_2}\boldsymbol{x}_3 - \text{proj}_{\boldsymbol{z}_1}\boldsymbol{x}_3 \qquad \rightarrow \boldsymbol{q}_3 = \boldsymbol{z}_3 / \left\|\boldsymbol{z}_3\right\|_2 \tag{4.42}$$

$$\vdots$$

$$\boldsymbol{z}_m = \boldsymbol{x}_m - \sum_{j=1}^{m-1} \text{proj}_{\boldsymbol{z}_j}\boldsymbol{x}_i \qquad \rightarrow \boldsymbol{q}_m = \boldsymbol{z}_m / \left\|\boldsymbol{z}_m\right\|_2 . \tag{4.43}$$

Matrix Q is then constructed via

$$Q = \begin{bmatrix} \boldsymbol{q}_1 & \boldsymbol{q}_2 & \cdots & \boldsymbol{q}_n \end{bmatrix} . \tag{4.44}$$

To build the matrix R, which has the structure

$$R = \begin{bmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,n} \\ 0 & r_{2,2} & & r_{2,n} \\ \vdots & & \ddots & \cdots \\ 0 & 0 & & r_{n,n} \end{bmatrix} , \tag{4.45}$$

we ask, "**what linear sum of Q columns will reconstruct the data matrix columns?**" We can see this explicitly by writing out the QR product

$$X = QR = \begin{bmatrix} \boldsymbol{q}_1 & \boldsymbol{q}_2 & \cdots & \boldsymbol{q}_n \end{bmatrix} \begin{bmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,n} \\ 0 & r_{2,2} & & r_{2,n} \\ \vdots & & \ddots & \cdots \\ 0 & 0 & & r_{n,n} \end{bmatrix} \tag{4.46a}$$

$$= \begin{bmatrix} (r_{1,1}\boldsymbol{q}_1) & (r_{1,2}\boldsymbol{q}_1 + r_{2,2}\boldsymbol{q}_2) & \cdots & \sum_{i=1}^{n} r_{i,n}\boldsymbol{q}_i \end{bmatrix} \tag{4.46b}$$

$$= \begin{bmatrix} \boldsymbol{x}_1 & \boldsymbol{x}_2 & \cdots & \boldsymbol{x}_n \end{bmatrix} . \tag{4.46c}$$

In this formulation, we see that $r_{1,1}\boldsymbol{q}_1 = \boldsymbol{x}_1$. To then solve for the numerical value of $r_{1,1}$, we can simply take the dot product of both sides with $\boldsymbol{q}_1$ (which, remember, is normalized):

$$r_{1,1}\boldsymbol{q}_1 = \boldsymbol{x}_1 \tag{4.47}$$

$$\boldsymbol{q}_1^T (r_{1,1}\boldsymbol{q}_1) = \boldsymbol{q}_1^T (\boldsymbol{x}_1) \tag{4.48}$$

$$r_{1,1} = \boldsymbol{q}_1^T \boldsymbol{x}_1 \tag{4.49}$$

To get $\boldsymbol{x}_1$, for example, all we need to do is scale up $\boldsymbol{q}_1$ by the dot product of $\boldsymbol{q}_1$ and $\boldsymbol{x}_1$:

$$\boldsymbol{x}_1 = \underbrace{\left(\boldsymbol{q}_1^T \boldsymbol{x}_1\right)}_{r_{1,1}} \boldsymbol{q}_1 . \tag{4.50}$$

Similarly, we can reconstruct $\boldsymbol{x}_2$ by summing its projections onto $\boldsymbol{q}_1$ and $\boldsymbol{q}_2$:

$$\boldsymbol{x}_2 = \underbrace{\left(\boldsymbol{q}_1^T \boldsymbol{x}_2\right)}_{r_{1,2}} \boldsymbol{q}_1 + \underbrace{\left(\boldsymbol{q}_2^T \boldsymbol{x}_2\right)}_{r_{2,2}} \boldsymbol{q}_2 . \tag{4.51}$$

Generalizing this process, matrix R takes the form

$$R = \begin{bmatrix} \boldsymbol{q}_1^T \boldsymbol{x}_1 & \boldsymbol{q}_1^T \boldsymbol{x}_2 & \cdots & \boldsymbol{q}_1^T \boldsymbol{x}_n \\ 0 & \boldsymbol{q}_2^T \boldsymbol{x}_2 & & \boldsymbol{q}_2^T \boldsymbol{x}_n \\ \vdots & & \ddots & \cdots \\ 0 & 0 & & \boldsymbol{q}_n^T \boldsymbol{x}_n \end{bmatrix}. \tag{4.52}$$

Rather than computing these entries one by one, we can compute the matrix R directly via

$$QR = X \tag{4.53}$$

$$R = Q^T X. \tag{4.54}$$

⋆ Homework 5, Problem 3: QR Decomposition Practice

We want to use QR decomposition to solve the overdetermined system

$$\min_{\boldsymbol{w}} \|\boldsymbol{y} - X\boldsymbol{w}\|, \; X = \begin{bmatrix} 1 & 12 \\ 6 & -5 \\ 8 & 9 \end{bmatrix}, \; \boldsymbol{y} = \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix}. \tag{4.55}$$

(You may do the following math in python, but write down what we ask to see, and show the backward substitution solution steps).

(a) First, compute the matrix $Q = [\boldsymbol{q}_1 \; \boldsymbol{q}_2]$ by applying Gram-Schmidt to the columns of the data matrix X. **Hint**: $\boldsymbol{q}_1 = \boldsymbol{x}_1 / \|\boldsymbol{x}_1\|_2$. For $\boldsymbol{q}_2$, reject $\boldsymbol{q}_1$ from $\boldsymbol{x}_2$, and then normalize. Now, build matrix R, where $R = Q^T X$. Write down the matrix values.

(b) Compute $Q^T \boldsymbol{y}$ from (4.39c). Now, by hand, solve the system of equations

$$\underbrace{R}_{\checkmark} \underbrace{\boldsymbol{w}}_{??} = \underbrace{Q^T \boldsymbol{y}}_{\checkmark}. \tag{4.56}$$

This represent two equations and two unknowns. Use backward substitution to solve this system: start with the last equation, plug in the solution, and then solve the first equation. Write down your answers for w_1 and w_2.

Solution.

(not posted yet)

4.6 *Regularized* Linear Regression

Regularization is powerful tool. As previously discussed, it introduces an inductive bias into your model, "steering" the training (or optimization) towards solutions with certain, hopefully desirable, characteristics. In this subsection, we review three popular penalty-based regularization methods: Ridge, Lasso, and Elastic Net[†].

4.6.1 Ridge Regression (L2 Norm Regularization)

Statistically, ridge regression poses a standard MLE least squares problem, but it adds a 0-mean **Gaussian prior** on the weights of the model (the 0-mean assumption can be relaxed, when helpful). When this prior is added, the MLE problem becomes a MAP problem. We saw an example of a Gaussian prior way back in (2.40). Assuming we have M model parameters, and each are uncorrelated, the Gaussian prior can we written as

$$p(\boldsymbol{w}) = \prod_{i=1}^{M} \frac{1}{\sqrt{2\pi\sigma_i^2}} e^{-\frac{w_i^2}{2\sigma_i^2}} : \quad \textbf{Gaussian Prior on Model Parameters.} \tag{4.57}$$

[†]These methods are simultaneously referred to as **regression** methods, and as **regularization** methods. Both conventions are fine. I prefer to call them regularization methods; it feels more specific.

Taking the NLL[‡], we have

$$\text{NLL}(p(\boldsymbol{w})) = c + \sum \frac{w_i^2}{2\sigma_i^2} \tag{4.58a}$$

$$= c + \sum \lambda_i w_i^2, \quad \lambda_i \triangleq \frac{1}{2\sigma_i^2}, \tag{4.58b}$$

where c is a constant. When we add this as a **prior** for the MLE of the least square problem, we get (several steps not shown):

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} \|\boldsymbol{y} - X\boldsymbol{w}\|_2^2 + \sum \lambda_i w_i^2 \tag{4.59a}$$

$$\arg\min_{\boldsymbol{w}} \|\boldsymbol{y} - X\boldsymbol{w}\|_2^2 + \|\boldsymbol{w}\|_\Lambda^2 : \quad \textbf{Ridge Regression} \tag{4.59b}$$

where $\|\boldsymbol{w}\|_\Lambda^2$ is a "weighted norm", and Λ is diagonal weight matrix:

$$\Lambda = \begin{bmatrix} \sqrt{\lambda_1} & 0 & \cdots & 0 \\ 0 & \sqrt{\lambda_2} & & \\ \vdots & & \ddots & 0 \\ 0 & & 0 & \sqrt{\lambda_M} \end{bmatrix}. \tag{4.60}$$

Of course, if all models weights are equal $\lambda_1 = \lambda_2 = \cdots = \lambda_M$, the problem simplifies to

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} \|\boldsymbol{y} - X\boldsymbol{w}\|_2^2 + \lambda \|\boldsymbol{w}\|_2^2, \tag{4.61}$$

which is a more common statement of Ridge Regression. Due to the presence of the L2 norm penalty on the weights, this is also called **L2 norm regulation**. What does it do? Simply put, it applies downward pressure on the size of the weights to make them smaller. Of course, changing weights from this optimal (MLE) values will incur some extra training loss: your model won't fit the training data quite as effectively! However, when problems are **poorly conditioned** (i.e., sensitive to data) or underdetermined (i.e., there isn't enough data), L2 norm regularization **speeds up** and **robustifies** training, and it can also help the model generalize more effectively. There may be other real-world benefits too: what if model coefficients correspond to energy curtailment in a power grid, or job layoffs in a large corporation? L2 norm regularization help push down the large coefficient values and e.g., "spread" the pain of job losses across many departments.

⋆ Homework 5, Problem 4: Ridge Regression Solution

Ridge Regression in (4.61) has a "closed-form" solution given by

$$\boldsymbol{w}^* = \left(X^T X + \lambda I\right)^{-1} X^T \boldsymbol{y} : \quad \textbf{Ridge Regression Solution}. \tag{4.62}$$

Using unconstrained optimization, show how to derive this solution. **Hint**: Take the gradient of (4.61), written as $(\boldsymbol{y} - X\boldsymbol{w})^T (\boldsymbol{y} - X\boldsymbol{w}) + \lambda \boldsymbol{w}^T \boldsymbol{w}$, and set it equal to 0.

Solution.

[‡]NLL refers specifically to the negative log of a likelihood function; however, allowing slight abuse of notation, we can use it to also mean "take the negative log of this distribution".

(not posted yet)

4.6.2 Lasso Regression (L1 Norm Regularization)

Pushing parameter values to be "small" isn't always enough: sometimes, we want a majority of parameter values to be **exactly 0**! Why would we want this? Why would we want to identify a sparse set of model parameters?

- **Example 1:** You are building a model whose coefficients represent the people infected with Covid back in February, 2020
- **Example 2:** You are building a model whose coefficients represent the suspected epicenter(s) of an earthquake
- **Example 3:** You are building a model whose coefficients represent the sources of a forced oscillations in an electric power grid.

To achieve **sparsity** in model parameters, one idea is to use the L0 "norm" from (1.3). As a reminder, $\|\boldsymbol{x}\|_0$ simply counts the number of nonzero elements in the vector $\boldsymbol{x}$. Thus, if we want to allow for only n nonzero model parameters, we could pose the following regression training problem:

$$\min_{\boldsymbol{w}} \ \|\boldsymbol{y} - X\boldsymbol{w}\|_2^2 \tag{4.63a}$$

$$\text{s.t. } \|\boldsymbol{w}\|_0 \leq n. \tag{4.63b}$$

The problem with (4.63) is that the L0 "norm" is nonconvex, and further more, it is generally non differentiable. Thus, getting a **good** solution, much less the ***best*** solution, for (4.63), is very hard. Instead, we modify (4.63) in two key ways: first, we take the tightest "convex relaxation" of (4.63b), which turns out to be an L1 norm, and second, we relax the constraint into a penalty,

which we "turn up" to drive subsets of model parameters to 0. **Both** of the changes are reflected in the following regression formulation:

$$\min_{\boldsymbol{w}} \ \|\boldsymbol{y} - X\boldsymbol{w}\|_2^2 + \lambda t : \quad \textbf{Lasso } \text{🤠} \tag{4.64a}$$

$$\text{s.t. } \|\boldsymbol{w}\|_1 \leq t. \tag{4.64b}$$

We call this formulation the **least absolute shrinkage and selection operator**, or **lasso**. Statistically, lasso regression poses a standard MLE least squares problem, but it adds a 0-mean **Laplace prior** on the weights of the model (the 0-mean assumption can be relaxed, when helpful). When this prior is added, the MLE problem becomes a MAP problem. We saw an example of a Laplace prior way back in (2.43). Assuming we have M model parameters, and each are uncorrelated, the Laplace prior can we written as

$$p_L(\boldsymbol{w}) = \prod_{i=1}^{M} \frac{1}{2b_i} e^{-\frac{|w_i|}{b_i}} : \quad \textbf{Laplace Prior on Model Parameters.} \tag{4.65}$$

Taking the NLL,

$$\text{NLL}(p_L(\boldsymbol{w})) = c + \sum_{i=1}^{M} \frac{|w_i|}{b_i} \tag{4.66a}$$

$$= c + \sum_{i=1}^{M} \lambda_i \left|w_i\right|, \ \lambda_i \triangleq \frac{1}{b_i}, \tag{4.66b}$$

where c is a constant. When we add this as a prior for the MLE of the least square problem, we get (several steps not shown):

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} \|\boldsymbol{y} - X\boldsymbol{w}\|_2^2 + \sum \lambda_i |w_i| \tag{4.67}$$

$$\arg\min_{\boldsymbol{w}} \|\boldsymbol{y} - X\boldsymbol{w}\|_2^2 + \lambda \|\boldsymbol{w}\|_1 , : \quad \textbf{Lasso Regression} \tag{4.68}$$

which has assumed all model weights are equal (this is common in lasso). We note that (4.68) and (4.64) are **equivalent**. Why? Well, the minimizer in (4.64) is going to push "down" as much as it can on the slack variable t, until $t = \|\boldsymbol{w}\|_1$. Thus, we can just replace t with $\|\boldsymbol{w}\|_1$. This is called the "epigraph trick" [15], but it's beyond the scope of this class.

Solving Lasso: Ridge Regression has a known, and rather beautiful, closed-form solution: (4.62). Lasso, however, lacks such a solution 😞. Very sad. This lack of solution is driven by the fact that $|x|$ is inherently non-differential at $x = 0$. To have a deeper understanding of why (i) lasso does not have a closed for solution, and (ii) lasso is so good at driving a large set of model parameters to 0, we need to understand **multi-objective optimization**, and we need to understand **sub-gradients**.

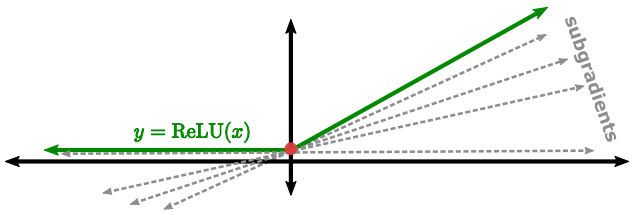


Figure 10: A subset of the **subgradients** of the ReLU activation function ($\text{ReLU}(x) \triangleq \max(x, 0)$).

Multi-Objective Optimization

An optimization problem with **competing** terms in the objective is referred to as a **Multi-Objective Optimization** problem. For example:

$$\arg\min_{\boldsymbol{x}} f(\boldsymbol{x}) + p(\boldsymbol{x}). \tag{4.69}$$

At optimality, $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) + \nabla_{\boldsymbol{x}} p(\boldsymbol{x}) \equiv \mathbf{0}$ (first order optimality condition). Thus, at optimality,

$$\nabla f(\boldsymbol{x}) = -\nabla p(\boldsymbol{x}). \tag{4.70}$$

The multi-objective optimization condition (4.70) can help us understand regularization: when you solve $\arg\min_{\boldsymbol{x}} \|\boldsymbol{y} - X\boldsymbol{w}\|_2^2$, and then you plug this solution into ridge regression, the gradients will not be balanced!

$$\underbrace{\|\boldsymbol{y} - X\boldsymbol{w}\|_2^2}_{\nabla_{\boldsymbol{w}}=0} + \underbrace{\lambda \|\boldsymbol{w}\|}_{\nabla_{\boldsymbol{w}}=2\lambda\boldsymbol{w}}, \qquad 0 \neq -2\lambda\boldsymbol{w}. \tag{4.71}$$

The optimizer will then push **down** on the model coefficients, to make the penalty function gradient **smaller**, and the loss gradient **larger** (but negative). Eventually, these two meet in the middle at the solution (**??**), where is (4.70) satisfied. To apply this analysis to lasso, we need to define the **subgradient**.

Subgradients of a Non-Differentiable Function

When the gradient of a (convex) function $f(x)$ is discontinuous at some point x_0, we may define the **subgradients** as the set of all gradients which lower-bound function (see Fig. 10):

$$\mathcal{S} = \{g \,|\, f(x_0 + \Delta x) \geq f(x_0) + g\Delta x, \ \forall \Delta x\}. \tag{4.72}$$

This definition is given for a **convex function**, but that technical condition isn't too restrictive. In the end, a sub-gradient is just a tangent line which **lower bounds** a function at a point of non-differentiability.

⋆ Example 14: Subgradient of the Absolute Value Operator

What is the **subgradient** of the absolute value operator $|\cdot|$ at $x = 0$? Well, any gradient

between -1 and $+1$ (inclusive) will "lower bound" the function:

$$\frac{\partial |x|}{\partial x} = \begin{cases} -1, & x < 0 \\ +1, & x > 0 \\ \mathcal{S} = [-1, +1], & x = 0: \quad \textbf{subgradients!} \end{cases} \tag{4.73}$$

⋆ Homework 5, Problem 5: Subgradient of the ReLU

What is the subgradinent of the ReLU operator at $x = 0$?

Solution.

(not posted yet)

Level Sets: In Fig. 12, we plot the **level sets** associated with two regression problems (well, identical regression problems, but regularized with different norms). A **level set** is the set of all input values (e.g., x, y) which map a function $f(\cdot)$ to some constant output c:

$$\mathcal{C} = \{x, y \,|\, f(x, y) = c\}: \quad \textbf{Level Set} \tag{4.74}$$

Level sets can be likened to the **contour elevation lines** drawn on a topological map, as shown in Fig. 11. Each line represents a constant elevation: when you traverse a line, the elevation does not change! There are two intuitive points that follow:

1. The "loss function" value will not change as you traverse along the level set (i.e., the **directional derivative** of the loss function, in the direction of the level set, is 0). As a reminder, the directional derivative computes the component of a gradient which points in some direction $\boldsymbol{v}$:

$$\boldsymbol{v}^T \nabla_{\boldsymbol{x}} f(\boldsymbol{x}): \quad \textbf{Directional Derivative.} \tag{4.75}$$

2. The **steepest descent direction** will be normal (i.e., perpendicular) to the level sets. This will be shown in the following example.

⋆ Example 15: Level Set of a Multivariate Quadratic

Consider the level set of the **circular paraboloid** $x^2 + y^2 = z$ with $z \equiv 1$. In this example, we want to show that if you take the directional derivative of the level set, you get 0. In other words, the gradient of steepest descent is **perpendicular** to the level set.

The gradient of this function at some point on the level set is given by

$$\nabla f = \begin{bmatrix} 2x \\ 2y \end{bmatrix} = \begin{bmatrix} 2x \\ 2\sqrt{1-x^2} \end{bmatrix} = \boldsymbol{g}, \tag{4.76}$$

since $y^2 = 1 - x^2$ must be satisfied on the level set.

The level set ***tangent*** (i.e., the line that points in the direction of the level set) is given by

$$x^2 + y^2 = 1 \tag{4.77}$$

$$y = \sqrt{1-x^2} \tag{4.78}$$

$$\frac{\partial y}{\partial x} = \frac{1}{2}\left(1-x^2\right)^{-\frac{1}{2}}(-2x) \tag{4.79}$$

$$= \frac{-x}{\sqrt{1-x^2}} \tag{4.80}$$

$$\partial y = \left(\frac{-x}{\sqrt{1-x^2}}\right)\partial x: \quad \textbf{Level set tangent}. \tag{4.81}$$

We put this tangent into vector form via

$$\boldsymbol{t} = \begin{bmatrix} \partial x \\ \partial y \end{bmatrix} = \begin{bmatrix} \partial x \\ \frac{-x}{\sqrt{1-x^2}}\partial x \end{bmatrix} = \begin{bmatrix} 1 \\ \frac{-x}{\sqrt{1-x^2}} \end{bmatrix},$$

by just setting the partial equal to 1. We may now take the **directional derivative** along this level set (in the direction of $\boldsymbol{t}$):

$$\boldsymbol{t}^T \nabla f = \begin{bmatrix} 1 \\ \frac{-x}{\sqrt{1-x^2}} \end{bmatrix}^T \begin{bmatrix} 2x \\ 2\sqrt{1-x^2} \end{bmatrix} \tag{4.82a}$$

$$= 2x - 2x\frac{\sqrt{1-x^2}}{\sqrt{1-x^2}} \tag{4.82b}$$

$$= 0. \tag{4.82c}$$

Thus, as predicted, directional derivative of the function along the level set is 0: there is no change in the loss function in this direction!

So, how does lasso induce model parameter shrinkage? When we regularize a least squares problem with an **L1 norm penalty** on the weights, the gradient of the loss function begins to grow in magnitude (but negatively) until it intersects with the gradient of the L1 norm. Typically, this intersection will happen at a subgradient, as depicted in 12. In short, the subgradeint

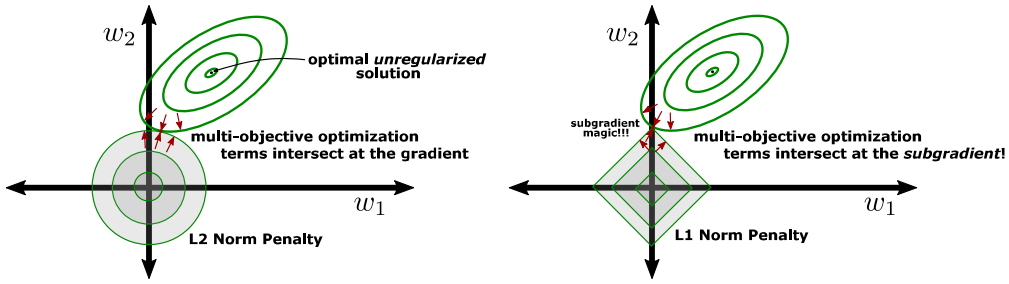


Figure 12: Depiction of why ridge regression parameters tend to be dense, while lasso parameters tend to be sparse. The left plot show ridge regression, where the multi-objective optimization problem finds a stationary solution when the steepest gradients (normal to the level sets) are opposite and equal, à la (4.70). The right plots shows lasso regression. In this case, the steepest gradients are opposite and equal at a corner point of the L1 norm, where the gradient is undefined and is thus replaces by a family a subgradients One of these subgradients is equal and opposite to the gradient of the loss function term, and thus a solution is found.

of the absolute value operator is **diverse** at $x = 0$, so it provides many opportunities for intersection with a corresponding loss term in a multi-objective optimization problem.

Actually Solving Lasso: Sadly, lasso does not have a closed form solution, so iterative numerical techniques are typically employed. Such techniques include **projected gradient descent**, **proximal gradient descent**, **coordinate descent**, **least angle regression** [1], and other more traditional methods, like **simplex** and **barrier** methods (these methods, however, do not scale as effectively).

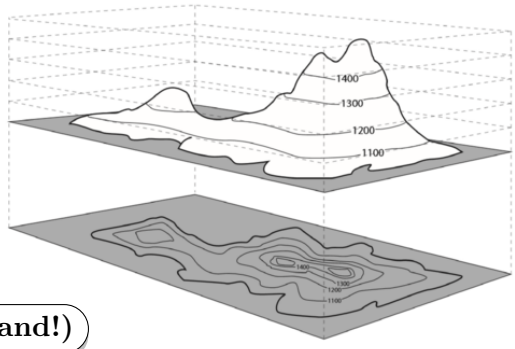


Figure 11: Topological Map Contour Lines

⋆ Homework 6, Problem 1: Solving Lasso (by hand!)

In this problem, we want to demonstrate how "**shrinkage**" can drive a model parameter to 0. Consider the L1-norm regularized loss function

$$f(\boldsymbol{w}) = (w_1 - 1)^2 + (w_2 - 10)^2 + \lambda\left(|w_1| + |w_2|\right). \tag{4.83}$$

This, of course, is **lasso**.

(a) Set $\lambda = 6$ and solve lasso. As usual, take the gradient, and set it to 0. What are the values of w_1 and w_2?

(b) Based on your solution from the first part, what is the value of λ which will drive both parameters to 0? This might take a little bit **puzzling** to think through. That's ok.

Solution.

(not posted yet)

4.6.3 Elastic Net Regression (L2 *and* L1 Norm Regularization)

Elastic Net Regularization combines L1 and L2 norm regularization via

$$\boldsymbol{w}^* \arg\min_{\boldsymbol{w}} \|\boldsymbol{y} - X\boldsymbol{w}\|_2^2 + \lambda_1 \|\boldsymbol{w}\|_1 + \lambda_2 \|\boldsymbol{w}\|_2 : \quad \textbf{Elastic Net Regression}. \tag{4.84}$$

As with lasso, there is no closed form solution for an elastic net regression problem. In general, the elastic net offers a nice **middle ground** between ridge and lasso. Lasso can be brittle (especially in underdetermined problems, or when training samples are highly correlated [3]), and ridge can unfairly penalize large parameters. Thus, elastic net can offer a good trade-off between these alternatives.

4.7 Gradient-Based Solutions for Least Squares

In the final subsection of this topic, we investigate how we can use **gradient-based methods** to solve least squares problems. This will help serve as a gentle introduction to a technique which is used to optimize the largest models ever conceived of (i.e., LLMs, whose trainings are guided by gradients descent steps).

If you are lost on a mountain, and you need to get to the bottom, what should you do? Well, try taking a step **downhill**. If you keep moving downhill, eventually, you will get to the bottom. Which direction will get you there these fastest? Typically, you want to move in the **steepest descent** direction (this is the direction that a ball will naturally roll, since the effective pull of gravity will be strongest). At its core, **gradient descent** solves an optimization problem like $\min_x \ f(x)$ by taking a step "downhill":

$$x \leftarrow x - \eta \nabla_x : \quad \textbf{Gradient Descent}. \tag{4.85}$$

Similarly, **gradient ascent** takes an uphill step to solve a maximization problem (i.e., $\max_x \ f(x)$):

$$x \leftarrow x + \eta \nabla_x : \quad \textbf{Gradient Ascent}. \tag{4.86}$$

Both gradient routines incorporate a step size parameter, η. Properly setting and updating this parameter as the gradient routines iterate has a profound effect on **convergence speed** (i.e., how long it takes from the gradient routine to solve the min or max problem). We will see several methods for adaptively updating these step sizes.

Remembering these formulas: Gradient ascent **adds** the scaled gradient to the state x, while gradient descent **subtracts** the gradient. This is easy enough to remember. More fundamentally, however, a gradient tells us how a variable and its output function are **positively correlated**: if you move one, how will this other change? Thus, a gradients says: "if I increase this variables, how much will my function increase?" Remember the definition of a derivative:

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}. \tag{4.87}$$

Analytical Motivation: We can also motivate gradient descent (and ascent) by viewing the gradient descent step as the solution to a linearized and regularized version of the original problem. To show this, we take a first order Taylor series approximation of the function $f(x)$ (scalar function of one variable, x):

$$f(x) \approx f(x_0) + \left.\frac{\partial f}{\partial x}\right|_{x_0} (x - x_0). \tag{4.88}$$

In the following HW problem, we minimize a regularized version of this function.

⋆ Homework 6, Problem 2: Minimization of a Linearized, Regularized Function

Minimize the following linear function, which has been regularized with an L2 norm penalty:

$$\min_x \ f(x_0) + \left.\frac{\partial f}{\partial x}\right|_{x_0} (x - x_0) + \frac{1}{2\eta} (x - x_0)^2 . \tag{4.89}$$

What is the solution for x? How does this relate to the gradient descent update rule?

Solution.

(not posted yet)

Least Squares Gradient: The actual gradient of a least squares loss function (i.e., the loss function that emerges when we build a linear regression model) is computed via

$$\frac{\partial}{\partial \boldsymbol{w}} \left((\boldsymbol{y} - X\boldsymbol{w})^T (\boldsymbol{y} - X\boldsymbol{w}) \right) = \frac{\partial}{\partial \boldsymbol{w}} \left(\boldsymbol{y}^T \boldsymbol{y} + \boldsymbol{w}^T X^T X \boldsymbol{w} - 2\boldsymbol{y}^T X \boldsymbol{w} \right) \tag{4.90a}$$

$$= 2X^T X \boldsymbol{w} - 2X^T \boldsymbol{y}. \tag{4.90b}$$

4.7.1 Feature Normalization

Before plugging (4.90b) into a gradient descent routine, it is common to first perform a feature normalization step. When a set of features arrive, they will all potentially have their own units (different feature means, ranges, variances, etc). In order to normalize the features, it is common to (1) **subtract the feature mean** from the inputs and outputs, and (2) **normalize by the standard deviation** of the data (or the range, if you have good reason). Why do we do this? The main reason is this: gradient descent tends to optimize normalized models much more quickly. If you don't believe this, just go try it. By pre-normalizing, we are doing some of the work "for" the gradient optimizer. It just so happens that this work is easy for us, and hard for the optimizer, so it's a good match.

For a simplified example, we consider a model with a single input and a single output: $y = f(x)$.

Given vectors of inputs $\boldsymbol{x}$ and outputs $\boldsymbol{y}$, we first compute the means and variances:

$$\mu_x = \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \textbf{(input mean)} \tag{4.91}$$

$$\sigma_x^2 = \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_x)^2 \qquad \textbf{(input variance)} \tag{4.92}$$

$$\mu_y = \frac{1}{m}\sum_{i=1}^{m} y_i \qquad \textbf{(output mean)} \tag{4.93}$$

$$\sigma_y^2 = \frac{1}{m}\sum_{i=1}^{m} (y_i - \mu_y)^2 \qquad \textbf{(output variance)}. \tag{4.94}$$

Input and output data are then normalized via

$$\hat{\boldsymbol{x}} = \frac{\boldsymbol{x} - \mu_x}{\sigma_x}, \quad \hat{\boldsymbol{y}} = \frac{\boldsymbol{y} - \mu_y}{\sigma_y}, \tag{4.95}$$

and a model is trained on the normalized data:

$$\min_f \ \|\hat{\boldsymbol{y}} - f(\hat{\boldsymbol{x}})\|_2^2 . \tag{4.96}$$

We don't actually specify the model in (4.96), because the process of feature normalization (also called **batch normalization** in future topics) is highly general. When we go to train a neural network model, we will use a similar normalization process. Once the model is trained and we want to **use or deploy** it, we must ensure to (i) normalize all future inputs, and then (ii) de-normalize the output:

$$\hat{y} = f\left(\frac{x_{\text{new}} - \mu_x}{\sigma_x}\right) \qquad \textbf{(normalize future inputs)} \tag{4.97}$$

$$y_{\text{new}} = \sigma_y \hat{y} + \mu_y \qquad \textbf{(de-normalize outputs)} \tag{4.98}$$

While this example has been shared for a single-input single-output system, the process **generalizes** for any system with m output features and n inputs features. For **each feature**, we define the mean and variance, and the we normalize according to (4.95).

⚠ If you are building a linear regression model, and you normalize the inputs and outputs, you **do not need** to include a bias term in the regression model: it will be zero!

$$\hat{y} = \underbrace{w_0}_{\textbf{bias coefficient not needed when features are normalized!}} + w_1 x_1 + w_2 x_2 + \cdots + w_k x_k. \tag{4.99}$$

Mean Square Error: The loss function in (4.96) represent the **Sum of Squared Error** (SSE). In order to ensure that gradient step size isn't too large, it can be helpful to instead use **Mean Square Error**, which normalizes the sum of the squared errors by the number of data points, N. For Ordinary Least Squares (OLS), the MSE is given by

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{N}\|\boldsymbol{y} - X\boldsymbol{w}\|_2^2 : \quad \textbf{Mean Square Error (MSE) for OLS}. \tag{4.100}$$

Of course, this can (and should) also be combined with feature normalization.

Algorithm 4 Full Batch Gradient Descent

Input: Input features X and output features $\boldsymbol{y}$

while Not Converged **do**
  Compute the full-batch gradient: $\nabla_{\boldsymbol{w}} = 2X^T(X\boldsymbol{w} - \boldsymbol{y})$
  Take a gradient step: $\boldsymbol{w} = \boldsymbol{w} - \eta\nabla_{\boldsymbol{w}}$
  Compute the loss: $\mathcal{L} = \|\boldsymbol{y} - X\boldsymbol{w}\|_2^2$
end while

Algorithm 5 Stochastic Gradient Descent

Input: Input features X, output features $\boldsymbol{y}$, number of data samples N

while Not Converged **do**
  Reset the set $\mathcal{S} = \{1, 2, ..., N\}$
  for $i = 1, 2, ..., N$ **do**
    Remove a random element, j, from S
    Compute the single-sample (j) gradient: $\nabla_{\boldsymbol{w}} = 2\boldsymbol{x}_j(\boldsymbol{x}_j^T\boldsymbol{w} - y_j)$
    Take a gradient step: $\boldsymbol{w} = \boldsymbol{w} - \eta\nabla_{\boldsymbol{w}}$
  end for (*end of training epoch*)
  Compute the loss: $\mathcal{L} = \|\boldsymbol{y} - X\boldsymbol{w}\|_2^2$
end while

4.7.2 Accelerating Gradient Computations with Batching

When we compute the gradient in (4.90b), we are computing the gradient of the loss function with the **full data matrix**. This can be very slow when the data set is big (especially when we start solving neural network training problems). Furthermore, we don't necessarily need all of this gradient information to train a really good model. Taking the gradient of the loss function with respect to the full dataset is called **Batch** (or **Full Batch**) **Gradient Descent**. If instead, we take the gradient with respect to a random data sample (i.e., one randomly pulled from the full dataset) and perform a parameter update step, we call this **Stochastic Gradient Descent** (SGD). Finally, if we break the data into "mini-batches" and take a parameter update step based on the gradients of each one of these batches, we call this **Mini-Batch Gradient Descent**. These routines are reviewed in Algs. 4, 5, and 6.

Gradient Computation Strategies

⋆ **Batch Gradient Descent:** The gradient of the **full** dataset is used at each parameter update. **Accurate** ✔, but **slow** ✗

⋆ **Stochastic Gradient Descent:** The gradient of a single, randomly chosen data sample is used at each parameter update. **Fast** and **doesn't get stuck** in local minima ✔, but **erratic convergence** ✗

⋆ **Mini-Batch Gradient Descent:** The gradient of a rotating group of data samples is used at each parameter update. Falls somewhere between BGD and SGD (✔/✗)

Algorithm 6 Mini-Batch Gradient Descent

Input: Input features X and output features $\boldsymbol{y}$, number of batches B

Evenly Sort data into batches $\{X_1, \boldsymbol{y}_1\}, \{X_2, \boldsymbol{y}_2\}, \cdots, \{X_B, \boldsymbol{y}_B\}$
while Not Converged **do**
  for $i = 1, 2, ..., B$ **do**
    Compute the mini-batch gradient: $\nabla_{\boldsymbol{w}} = 2X_i^T(X_i\boldsymbol{w} - \boldsymbol{y}_i)$
    Take a gradient step: $\boldsymbol{w} = \boldsymbol{w} - \eta\nabla_{\boldsymbol{w}}$
  end for (*end of training epoch*)
  Compute the loss: $\mathcal{L} = \|\boldsymbol{y} - X\boldsymbol{w}\|_2^2$
end while

4.7.3 Learning Rate Decay

In order to help gradient descent routines to converge (especially Stochastic Gradient Descent and Mini-Batch), it is common to use a **learning rate decay** scheme. A commonly used learning rate update rule simply multiplies the learning rate by a decay factor at each gradient descent iteration:

$$\eta \leftarrow 0.99\eta: \quad \textbf{Learning Rate Decay}. \tag{4.101}$$

Exponential decay: With this update rule, η will exhibit **exponential decay** from its starting value of η_0. Why is this? Consider the first order differential equation $\dot{\eta} = -0.01\eta$. The **solution** for this differential equation is given by $\eta = e^{-0.01t}\eta_0$. You can check that this is a valid solution by "plugging it in" to the differential equation:

$$\dot{\eta} = -0.01\eta \quad \rightarrow \quad \eta = e^{-0.01t}\eta_0 \tag{4.102a}$$

$$\frac{d}{dt}\left(e^{-0.01t}\eta_0\right) = -0.01\left(e^{-0.01t}\eta_0\right) \tag{4.102b}$$

$$-0.01e^{-0.01t}\eta_0 = -0.01e^{-0.01t}\eta_0 \quad 4 \tag{4.102c}$$

Now, let us discretize the differential equation:

$$\dot{\eta} = -0.01\eta \tag{4.103a}$$

$$\frac{\eta_t - \eta_{t-1}}{\Delta t} = -0.01\eta_{t-1} \tag{4.103b}$$

$$\eta_t = \eta_{t-1} - \Delta t \cdot 0.01\eta_{t-1} \quad \rightarrow \quad \Delta t = 1 \tag{4.103c}$$

$$\eta_t = (1 - 0.01)\,\eta_{t-1} \tag{4.103d}$$

$$\eta_t = 0.99\eta_{t-1}, \tag{4.103e}$$

where we set $\Delta t = 1$, without loss of generality. Thus, we have shown that a difference equation (4.103e) maps back to a continuous time differential equation whose solution is a **decaying exponential**. Nice! We can also think of η as a variable which "**exponentially forgets**" its own initial conditions η_0. This will be useful later on.

4.7.4 Faster Gradient-Based Optimization Routines

The vanilla gradient-descent routine has **no memory** of past gradients, and it makes **no prediction** of future gradients. It simply looks at the current gradient, and it takes a step. There have been many proposed modifications which improve the effectiveness of first order gradient routines. We summarize five of these approaches below, with further details provided in [3].

Gradient Routine 1: Momentum The vanilla gradient-descent routine has no memory of past gradients. The most basic way to improve a gradient based routine is to give the parameter updates **momentum**. That is, allow the optimizer to speed up or slow down based on the past history of the gradients:

$$\textbf{Momentum:} \quad \begin{cases} \boldsymbol{m} \leftarrow \beta \boldsymbol{m} - \eta \nabla_{\boldsymbol{w}} f(\boldsymbol{w}) \\ \boldsymbol{w} \leftarrow \boldsymbol{w} + \boldsymbol{m} \end{cases} \tag{4.104}$$

where $\boldsymbol{m}$ represents the **momentum** of the parameters, and β is a "friction" parameter which falls between 0 (high friction) and 1 (low friction). $\beta = 0.9$ is typical. When friction is is high, the momentum is easily lost. The first equation represent an "exponential moving average" (where past gradient information is forgetting exponentially fast). When there is friction present, Momentum gradient descent will will reach a "terminal velocity" (i.e., it won't just accelerate forever, if it is going down an infinitely long ramp).

⋆ Homework 6, Problem 3: Terminal Momentum Velocity

We can also think of β as a **memory** parameter: when β approaches 0, the memory of past gradients is quickly lost. In this problem, we are going to explore the forgetting factor (also called, **exponential forgetting**) associated with momentum updates.

(a) Assume a constant gradient γ which does not change, such that $\boldsymbol{m}_i = \beta \boldsymbol{m}_{i-1} - \gamma$. Write $\boldsymbol{m}_i$ as a function of $\boldsymbol{m}_{i-2}$, then $\boldsymbol{m}_{i-3}$, then $\boldsymbol{m}_{i-n}$.

(b) Show that, after many iterations, $\boldsymbol{m} = -\frac{1}{1-\beta}\gamma$. This is called the momentum's **terminal velocity**. **Hint**: for $|\beta| < 1$, the power series expansion of $1/(1-\beta)$ is given by $1 + \beta + \beta^2 + \beta^3 + ...$

Solution.

(not posted yet)

Gradient Routine 2: Nesterov Accelerated Gradient Nesterov Accelerated Gradient (NAG) also uses momentum, but it uses the **future gradient**, rather than the **current gradient**, to update the momentum variable:

$$\textbf{Nesterov Accelerated Gradient:} \quad \begin{cases} \boldsymbol{m} \leftarrow \beta \boldsymbol{m} - \eta \nabla_{\boldsymbol{w}} f(\boldsymbol{w} + \beta \boldsymbol{m}) \\ \boldsymbol{w} \leftarrow \boldsymbol{w} + \boldsymbol{m} \end{cases} \tag{4.105}$$

Thus, NAG knows something about the future. Once again, as $\beta \to 0$, NAG returns to standard gradient descent. NAG tends to be faster than momentum.

Gradient Routine 3: AdaGrad When a learning problem has **steep gradients** mixed with **shallow gradients**, decaying the steep gradients will help "point" the parameter updates towards the global optimum. Adaptive Gradient (i.e., AdaGrad) will do just this[§]:

$$\textbf{AdaGrad:} \quad \begin{cases} \boldsymbol{s} \leftarrow \boldsymbol{s} + \nabla_{\boldsymbol{w}}^2 \\ \boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \frac{\nabla_{\boldsymbol{w}}}{\sqrt{\boldsymbol{s} + \epsilon}} \end{cases} \tag{4.106}$$

The variable $\boldsymbol{s}$ keeps a running tally of the squared gradients. Subtly, AdaGrad will penalize gradient terms that are decaying most rapidly, and **slow** the associated parameter updates down. This is called *adapting* the learning rates. AdaGrad generally stops too early, since small gradients get scaled down to 0 eventually.

⋆ Example 16: **AdaGrad Initialization**

[§]Please excuse the abuse of notation in (4.106); $\nabla_{\boldsymbol{w}}^2$, e.g., means element-wise squaring of all vector entries.

Assume we are taking the very first step with AdaGrad, and $\boldsymbol{s}$ has been initialized to the zero vector. What are the very first descent directions? Let's compute them! Applying the update rule (4.106), we have

$$\nabla_{\boldsymbol{w}}^2 \leftarrow 0 + \nabla_{\boldsymbol{w}}^2 \tag{4.107}$$

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \frac{\nabla_{\boldsymbol{w}}}{\sqrt{\nabla_{\boldsymbol{w}}^2 + \epsilon}}. \tag{4.108}$$

The parameter ϵ is very small, so we can ignore its effect. Thus, our descent directions are given by

$$\frac{\nabla_{\boldsymbol{w}}}{\sqrt{\nabla_{\boldsymbol{w}}^2 + \epsilon}} \approx \frac{\nabla_{\boldsymbol{w}}}{\sqrt{\nabla_{\boldsymbol{w}}^2}} = \operatorname{sign}\left(\nabla_{\boldsymbol{w}}\right), \tag{4.109}$$

since $x/\sqrt{x^2} = x/|x| = \operatorname{sign}(x)$. Thus, our very first gradients steps are all in the +1 and -1 directions. This seems **odd**, since it basically ignores the relative **magnitudes** of the various gradients, but this is a feature, not a bug, of adaptive gradient routines: aggressive descent normalizations.

⋆ Homework 6, Problem 4: Understanding Adaptive Learning Rates

Gradients in AdaGrad are scaled according to

$$\text{gradient scale} = \frac{g}{\sqrt{\sum_i g_i^2}}. \tag{4.110}$$

Consider two features.

(i) The first feature had an initial gradient of 100, and second gradient of 10.

(ii) The second feature had an initial gradient of 15, and a second gradient of 10.

For both of these features, compute the gradient scale (4.110). Even though they both have the same gradient presently, which one is scaled down more? Why?

Solution.

(not posted yet)

⋆ Example 17: AdaGrad's BIG Problem

Now, assume a set of features are on an infinitely long, flat hill with constant gradient vector $\boldsymbol{g}$. We compute the AdaGrad step sizes as $n \to \infty$. We first consider the updates to $\boldsymbol{s}$:

$$\boldsymbol{g}^2 \leftarrow \boldsymbol{0} + \boldsymbol{g}^2 \tag{4.111a}$$
$$2\boldsymbol{g}^2 \leftarrow \boldsymbol{g}^2 + \boldsymbol{g}^2 \tag{4.111b}$$
$$3\boldsymbol{g}^2 \leftarrow 2\boldsymbol{g}^2 + \boldsymbol{g}^2 \tag{4.111c}$$
$$4\boldsymbol{g}^2 \leftarrow 3\boldsymbol{g}^2 + \boldsymbol{g}^2 \tag{4.111d}$$
$$\vdots \tag{4.111e}$$
$$n\boldsymbol{g}^2 \leftarrow (n-1)\boldsymbol{g}^2 + \boldsymbol{g}^2 \tag{4.111f}$$
$$\tag{4.111g}$$

Now, we consider the update to the parameter vector after n updates to $\boldsymbol{s}$:

$$\boldsymbol{w} = \boldsymbol{w} - \eta \frac{\boldsymbol{g}}{\sqrt{n\boldsymbol{g}^2}} \tag{4.112a}$$
$$= \boldsymbol{w} - \eta \frac{1}{\sqrt{n}} \text{sign}(\boldsymbol{g}). \tag{4.112b}$$

Thus, as $n \to \infty$, the parameter step size shrinks to 0: $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \cdot 0 \cdot \text{sign}(\boldsymbol{g})$. This highlights

the **problem** with AdaGrad: over time, small gradients get normalized to 0, as previous large gradients dominate the denominator.

Gradient Routine 4: RMSProp The RMSProp algorithm "fixes" AdaGrad by applying exponential forgetting (i.e, decay) to the sum of the squared gradients:

$$\textbf{RMSProp:} \quad \begin{cases} \boldsymbol{s} \leftarrow \beta \boldsymbol{s} + (1-\beta)\nabla_{\boldsymbol{w}}^2 \\ \boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \frac{\nabla_{\boldsymbol{w}}}{\sqrt{\boldsymbol{s}+\epsilon}} \end{cases} \tag{4.113}$$

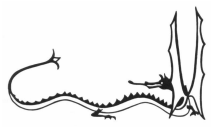
Typically, $\beta = 0.9$ is used. Based on the structure of (4.113), β has two interpretations: it is an exponential forgetting factor, and it also balances the memory of the past squared gradients, vs the inclusion of new squared gradient contributions.

Gradient Routine 5: Adam Finally, Adam combines features from RMSProp and Momentum. That is, it keeps tracks of the parameter momentum, *and* it uses adaptive learning rates. Adam stands for adaptive moment estimation, and it is the dominant gradient-based optimization routine used across much of Machine Learning. Its paper is the second-most cited paper in all of machine learning, racking up hundreds of thousands of citations [20]. It is also the optimization routine trusted to train modern Large Language Models[¶]. Its formulation is given by

$$\textbf{Adam:} \quad \begin{cases} \boldsymbol{m} \leftarrow \beta_1 \boldsymbol{m} - (1-\beta_1)\nabla_{\boldsymbol{w}} & \textbf{(Thanks Momentum!)} \\ \boldsymbol{v} \leftarrow \beta_2 \boldsymbol{v} + (1-\beta_2)\nabla_{\boldsymbol{w}}^2 & \textbf{(Thanks RMSProp!)} \\ \hat{\boldsymbol{m}} \leftarrow \frac{\boldsymbol{m}}{1-\beta_1^t} & \textbf{(Adam Contribution)} \\ \hat{\boldsymbol{v}} \leftarrow \frac{\boldsymbol{v}}{1-\beta_2^t} & \textbf{(Adam Contribution)} \\ \boldsymbol{w} \leftarrow \boldsymbol{w} + \eta \frac{\hat{\boldsymbol{m}}}{\sqrt{\hat{\boldsymbol{v}}+\epsilon}} & \textbf{(Thanks AdaGrad!)} \end{cases} \tag{4.114}$$

where $\beta_1 = 0.9$, and $\beta_2 = 0.999$ are commonly used. The third and fourth steps are de-biasing steps, where t is the iteration number (since the $\boldsymbol{m}$ and $\boldsymbol{s}$ are initialized at 0, these steps help boost their values up when t is small – as t grows in size, these steps have no effect). **AdaMax** (Adam with L-infinity norm, rather than L2-norm, tracking in step 2) and **Nadam** (Adam + NAG) are popular alternatives.

For an **excellent visual overview** of various gradient-based routines, check out this great Medium post by Lili Jiang. The associated github repo is here.



[¶]See page 43 of [21], which specifies the training routines for GPT3. Note: they also use "cosine decay" in place of, e.g., an exponential step decay from (4.101).

5 Nonlinear Regression

Some models are truly nonlinear. For example, consider the gravitational law:

$$f_{ij} = G\frac{m_i m_j}{r_{ij}^2} : \quad \textbf{Gravity!} \tag{5.1}$$

In other words, gravitational force is proportional to the product of masses divided by their distance square. This model has three input features: two masses and a distances. Given these three features, do you think you could ever learn a linear regression model to **accurately predict** gravitational force? The answer is **no**: the *best* model that a linear regression model can provide is a flat hyperplane. Clearly, (5.1) is not a flat hyperplane.

Here is an idea, though: what if we first **encode** the nonlinearity into a function, $g(m_i, m_j, r_{ij})$? Let define

$$g_{ij} = g(m_i, m_j, r_{ij}) \triangleq \frac{m_i m_j}{r^2} \tag{5.2}$$

where g_{ij} is a new variable which $g(m_i, m_j, r_{ij})$ spits out. Let's assume we have access to this new variable when we train out model. In doing so, we have "lifted" our problem (i.e., we have turned a problem of three variables into a problem of four variables, thus **lifting** our problem into a higher dimensional space). By properly lifting the problem, it has actually become linear again! Now, if we train a linear regression model, it will probably look like the following

$$f_{ij} \approx [G]\, g_{ij} + [0]m_i + [0]m_j + [0]r_{ij} + 0, \tag{5.3}$$

where feature coefficients are in parenthesis. We call a nonlinear function like (5.2) a **kernel**, and we say that (5.2) *kernelizes* our problem.

Kernelization and Lifting

⋆ **Lifting:** If a nonlinear function or transformation produces a new feature, or a new *set* of features, we say the associated model has been **lifted**.

⋆ **Kerelization:** We call the process of passing a feature, or a set of features, through a nonlinear function or transformation **kernelization**.

Take these definitions with a *grain of salt*: both terms can have slightly different meanings, depending on the context.

⋆ **Homework 6, Problem 5: Kernelizing A Quadratic Equation**

Given the initial position x_0, velocity v_0, and acceleration a_0 of a projectile, the equation of its current position is a quadratic function of time:

$$x(t) = x_0 + v_0 t + \frac{1}{2} a_0 t^2. \tag{5.4}$$

Let's say you want to build a model of the position without knowing this physics, and your only input is t (assume x_0, v_0, and a_0 are constants that your learn). Kernelize this function (you will need to use something like (5.2)). What are the effective linear regression model coefficients? Note: there can be more than one way to do this.

Solution.

(not posted yet)

5.1 Polynomial Regression

Often times, we **suspect** there is a nonlinear relationship between a set of inputs and outputs, but we don't know which kernel functions to use! In these cases, we can apply a series of polynomial kernel functions, and apply all of them. For single input, we define

$$\phi(x) = [1, x, x^2, ..., x^D]^T, \tag{5.5}$$

for a polynomial regression model of degree D [1]. In multiple dimensions (i.e., where we want to kernelize the higher order bilinear terms (i.e., $x_1 x_2$ is called a bilinear product), we may compute

$$\phi(x_1, x_2) = [1, x_1, x_1^2, x_2, x_2^2, x_1 x_2]^T, \quad D = 2. \tag{5.6}$$

Tools like scikit-learn's PolynomialFeatures will generate these polynomial features for you. Clearly, the dimensionality of the new feature set quickly explodes if you simultaneously kernelize all features in a dataset (with even a moderate degree. In fact,

$$\textbf{number of features: } \frac{(N+D)!}{D!N!}, \qquad D = \text{degree}, \; N = \text{original feature count},$$

which grows combinatorially* [3]. Once these features have now been kernelized, we can train a linear regression model on them! We call this process polynomial regression:

$$\min_{\boldsymbol{w}} \left\| \boldsymbol{y} - \boldsymbol{w}^T \phi(X) \right\|_2^2 : \quad \textbf{"Polynomial" Regression}. \tag{5.7}$$

We note that polynomial regression is a bit of a misnomer, since we are actually solving a linear regression problem in the end. It is important to **heavily regularize** polynomial regression so that we don't overfit.

*__Combinatorial growth__ makes __exponential growth__ look like a cute little chipmunk. Always beware of combinatorial explosion!

Algorithm 7 Polynomial Regression

Input: Input features X and output features $\boldsymbol{y}$; kernelization degree D
Output: Trained model

1: Suitably Kernelize the data via $\phi(X)$
2: Solve a linear regression problem (5.7)
Return: Regression model parameters $\boldsymbol{w}^*$

⋆ Homework 6, Problem 6: Polynomial Regression Regularization

Just like linear regression, we can **regularize** the training of polynomial regression models.

(a) What happens to **training error** as D (i.e., the kernelization degree) grows in size?

(b) Why do you think regularization of polynomial regression is so important (e.g., as compared to linear regression)?

Solution.

(not posted yet)

5.2 Sparse Identification of Nonlinear Dynamics (SINDy)

Regularization is a hugely important aspect of polynomial regression. In this section, we discuss a generalized version of polynomial regression which uses **lasso regularization** to **sparsely** select the most likely kernel functions out of a large candidate set.

Proposed by Steve Brunton[†] et al. [22], **Sparse Identification of Nonlinear Dynamics** (SINDy) applies a regularized regression procedure to identify the nonlinear functions which produce a set of dynamical **time series** data. This is not a class on nonlinear dynamics, but SINDy can be used for identifying the underlying models which produce many types of time series sequence data. The ultimate goal here is **parsimony**: start with a large candidate set of nonlinear kernel functions (we also call these **basis** functions), and then only select a very small subset which, hopefully, map back to a real-world "physics-based" model. The main motivation here is to **re-discover** the analytical structure of physics-based systems; for example, we might want to re-discover

- Navier-Stokes (fluids)
- Maxwell's equations (electromagnetics)
- Celestial mechanics (gravitational forces)
- Schrödinger equation (quantum!).

5.2.1 Nonlinear Dynamics and Time Series Data

We now offer a **very quick** primer on dynamical systems and time series data. Consider the set of ordinary differential equations (ODEs)

$$\dot{\boldsymbol{x}} = f(\boldsymbol{x}): \quad \textbf{Ordinary Differential Equations}, \tag{5.8}$$

which mean, "take a state vector, pass it through a set of functions f, and this will give you how the state is changing." This derivative information is then used to update the state vector as it evolves along some trajectory. Some examples of nonlinear dynamical systems are given below:

Cubic Oscillator:	**Hopf bifurcation:**	**Electric Power system:**
$\dot{x} = -0.1x^3 + 2y^3$	$\dot{\mu} = 0$	$\dot{\delta}_i = \omega_i$
$\dot{y} = -2x^3 - 0.1y^3$	$\dot{x} = \mu x + y - x(x^2 + y^2)$	$\dot{\omega}_i = \hat{p}_i - \hat{d}_i\omega_i - \sum \hat{B}_{ij} \sin(\delta_i - \delta_j)$
	$\dot{y} = \mu y - x - y(x^2 + y^2)$	$\forall i \in \mathcal{I}.$

Generally, there is no closed-form solution for most nonlinear differential equations, so we use numerical approximations. One of the simplest approximations is called **Forward Euler** Integration. Taking a **forward difference** approximation of the time derivative, we have

$$\frac{\boldsymbol{x}_{i+1} - \boldsymbol{x}_i}{\Delta t} = f(\boldsymbol{x}_i) \tag{5.9}$$

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \Delta t \cdot f(\boldsymbol{x}_i): \quad \textbf{Forward Euler Integration}. \tag{5.10}$$

From this integration scheme, time series data naturally emerges:

$$\boldsymbol{x}_1 = \boldsymbol{x}_0 + \Delta t \cdot f(\boldsymbol{x}_0) \tag{5.11a}$$

$$\boldsymbol{x}_2 = \boldsymbol{x}_1 + \Delta t \cdot f(\boldsymbol{x}_1) \tag{5.11b}$$

$$\boldsymbol{x}_3 = \boldsymbol{x}_2 + \Delta t \cdot f(\boldsymbol{x}_2) \tag{5.11c}$$

$$\vdots \tag{5.11d}$$

⋆ **Homework 7, Problem 1: Forward Euler Applied to an Exponential**

[†]As a brief aside, the first SINDy paper (and many papers by Brunton) is a wonderful example of high-quality research. No Theorems, no fancy math, just good ideas, explained clearly, with good visuals.

Consider the exponential decay ODE

$$\dot{x} = -x. \tag{5.12}$$

(a) Write down the forward Euler mapping, which maps x_i to x_{i+1}, for this system.

(b) Starting at $x_0 = 10$ with $\Delta t = 0.1$, compute x_1, x_2, and x_3. Are these increasing or decreasing?

Solution.

(not posted yet)

5.2.2 SINDy Regression

Given a set of time series measurements and gradients, our goal is recover the dynamical equations which produced said measurements. As an example, we may consider the dynamical equation $\dot{\boldsymbol{x}} = -\boldsymbol{x}^2$, where the dynamics are unknown (i.e., we only have $\dot{\boldsymbol{x}}$ and $\boldsymbol{x}$. Next, we build a regression model, where four candidate kernels are chosen (note: not all of these kernel functions are polynomial – SINDy considers a larger swath of potential basis functions):

$$\dot{\boldsymbol{x}} = [w_1]\,\boldsymbol{x} + [w_2]\,\boldsymbol{x}^2 + [w_3]\sin(\boldsymbol{x}) + [w_4]\cos(\boldsymbol{x}). \tag{5.13}$$

Given enough data, a regression solver should identify the following coefficients:

$$\dot{\boldsymbol{x}} = [0]\,\boldsymbol{x} + [-1]\,\boldsymbol{x}^2 + [0]\sin(\boldsymbol{x}) + [0]\cos(\boldsymbol{x}). \tag{5.14}$$

We see that this solution is **sparse**, in the sense that it was able to identify a single kernel/basis function to explain the output observations. The other basis functions were driven to 0. Given a

set of state measurements, though, how do we compute the gradients $\dot{\boldsymbol{x}}$? The rate of change of a variable is usually not a directly observable measurement. To estimate this rate-of-change[‡], we can use a finite difference method, like forward Euler: $\dot{x}(t_1) \approx \frac{x(t_1)-x(t_0)}{\Delta t}$.

Higher Dimensions: To extend this problem to higher dimensions (i.e., a system with n states and m time steps), we define the data matrix X and its approximated gradient $\dot{X}$.

$$X = \begin{bmatrix} x_1(t_0) & x_2(t_0) & \cdots & x_n(t_0) \\ x_1(t_1) & x_2(t_1) & \cdots & x_n(t_1) \\ \vdots & \vdots & \ddots & \vdots \\ x_1(t_{m-1}) & x_2(t_{m-1}) & \cdots & x_n(t_{m-1}) \end{bmatrix} \tag{5.15}$$

$$\dot{X} \approx \begin{bmatrix} \frac{x_1(t_1)-x_1(t_0)}{\Delta t} & \frac{x_2(t_1)-x_2(t_0)}{\Delta t} & \cdots & \frac{x_n(t_1)-x_n(t_0)}{\Delta t} \\ \frac{x_1(t_2)-x_1(t_1)}{\Delta t} & \frac{x_2(t_2)-x_2(t_1)}{\Delta t} & \cdots & \frac{x_n(t_2)-x_n(t_1)}{\Delta t} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{x_1(t_m)-x_1(t_{m-1})}{\Delta t} & \frac{x_2(t_m)-x_2(t_{m-1})}{\Delta t} & \cdots & \frac{x_n(t_m)-x_n(t_{m-1})}{\Delta t} \end{bmatrix}. \tag{5.16}$$

Next, we kernelize the states with a series of pre-chosen basis functions. We kernelize this matrix using the function $\Phi(\cdot)$. For **example**, we can kernelizes two states (x_1 and x_2) with degree two polynomial terms via

$$\Phi(X) = \begin{bmatrix} 1 & x_1(t_0) & x_2(t_0) & x_1^2(t_0) & x_2^2(t_0) & x_1(t_0)x_2(t_0) \\ 1 & x_1(t_1) & x_2(t_1) & x_1^2(t_1) & x_2^2(t_1) & x_1(t_1)x_2(t_1) \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_1(t_{m-1}) & x_2(t_{m-1}) & x_1^2(t_{m-1}) & x_2^2(t_{m-1}) & x_1(t_{m-1})x_2(t_{m-1}) \end{bmatrix} : \quad \text{(example)}. \tag{5.17}$$

Next, with slight abuse of notation, we pose a **lasso-regularized** regression problem:

$$\min_W \left\| \dot{X} - \Phi(X) W \right\|_2^2 + \lambda \|W\|_1 : \quad \textbf{“SINDy” Regression}, \tag{5.18}$$

where the coefficient matrix W is generally given by

$$W = \begin{bmatrix} \boldsymbol{w}_1 & \boldsymbol{w_2} & \cdots & \boldsymbol{w}_n \end{bmatrix}. \tag{5.19}$$

In this matrix, $\boldsymbol{w}_1$ are the model coefficients associated with state $x_1(t)$, etc. Of course, (5.18) can be broken up, so that the model parameters associated with each state are solved for independently:

$$\min_{\boldsymbol{w}_i} \left\| \dot{X}_i - \Phi(X) \boldsymbol{w}_i \right\|_2^2 + \lambda \|\boldsymbol{w}_i\|_1, \forall i \in \{1, 2, ..., n\}. \tag{5.20}$$

This, of course, is a regularized linear regression problem, and it can be solved as a standard lasso problem (i.e, despite the specific application, there is nothing inherently different about (5.20) and the canonical lasso problem (4.68). SINDy is nicely depicted in Fig. 13, where the sparse model coefficients of a Lorenz attractor are recovered. In this example, we note that SINDy is able to recover the **exact model parameters** associated with the Lorenz attractor; see [22] for more details.

⋆ Homework 7, Problem 2: SINDy and Overfitting

[‡]The original SINDy paper uses a method called Total Variation Regularization Derivative (TVRD).

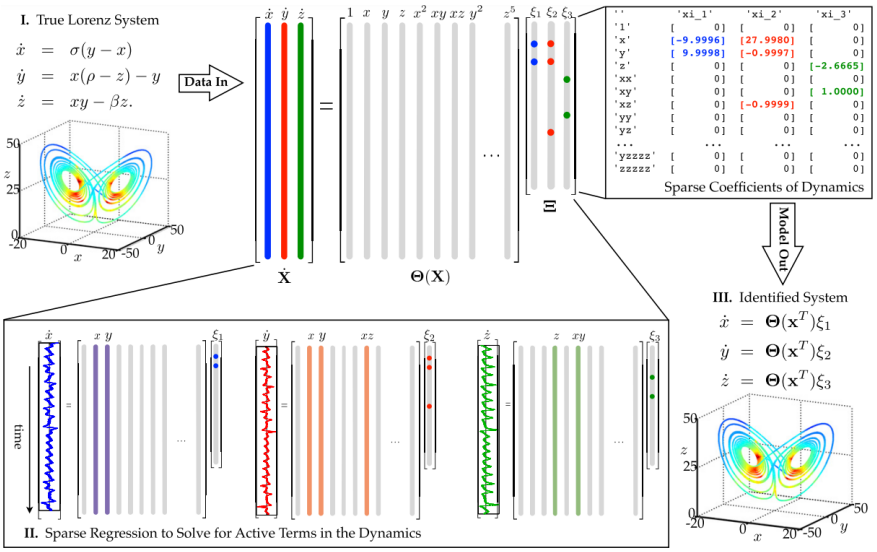
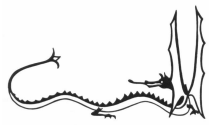


Figure 13: Visual depiction of SINDy, from the original paper [22].

Assume you have some simulated data $\boldsymbol{x}(t)$. With a regression tool like SINDy, we are hoping to **exactly** reconstruct a given time series data sequence. See, for example, the two images in Fig. 13, where the "True Lorenz System" dynamics are almost exactly replicated by the "Identified System". Therefore, is **overfitting** still a risk? You can't fit data any more perfectly than an "exact fit", so **why regularize** with lasso? Provide some thoughts.

Solution.

(not posted yet)



6 Classification Methods

In this section, we pivot from **regression models** (i.e., models which predict continuous numerical values) to **classification models** (i.e., models which predict the discrete classification of a sample based on a set of input features). There are many classification tools, but this section with focus on three of the **most foundational** methods: K-Nearest Neighbors (KNNs), Logistic Regression, and Support Vector Machines (SVMs).

6.1 K-Nearest Neighbors

So far, to make predictions, we have focused on building, and then training, "**parametric**" models. However, let's now assume I am **completely allergic to models**. Now assume I have access to a set of labeled training data, and I need to use it to **classify** an incoming data point. If I am allergic to models, what is the simplest thing I can do? The simplest thing is this:

1. Take an incoming data point
2. Find the "closest" labeled data point in the training set
3. Classify the incoming point based on its closest neighbor classification (i.e., assume it will have the same classification)

Parametric and Nonparametric Models

⋆ **Parametric:** A model is generally called *parametric* if the model size does not scale with the number of training samples (examples: linear regression, neural network).

⋆ **Nonparametric:** A model is generally called *nonparametric* if the number of model parameters scales with the number of training samples (example: K-Nearest-Neighbors, full depth decision trees). The model parameters are therefore, potentially, infinite dimensional. **Nonparametric** models are said to make fewer assumptions about the *structure* of the underlying data generating distribution, since they are more **flexible**.

The procedure of having classifying based on the classification of the nearest neighbor, however, is not **robust** to label noise nor to samples which fall right next to decision boundaries. To **robustify** this procedure, we may instead look at the classifications of the **K-nearest** neighbors. We then have these neighbors "vote" on the classification of an incoming point (where every point votes according to their own classification). Whichever classification has the most votes wins. For example, in Fig. 14, any point whose K-nearest neighbors are the seven points in the box would be classified as a 0 since there are **more** 0's than 1's.

We can also think of the KNN vote as the construction of a mini Bernoulli probability distribution. To build this probability distribution, we use the **indicator function** from [1]:

$$\mathbb{I}(e) = \begin{cases} 1, & e \text{ is true} \\ 0, & e \text{ is not true.} \end{cases} \tag{6.1}$$

Using this indicator function, we say that the probability of an i^{th} incoming point being classified as c, given the labeled training data $\mathcal{D}$,

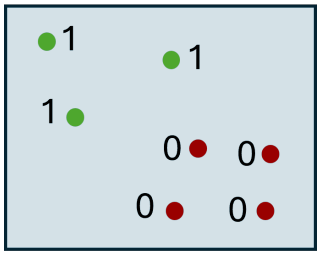


Figure 14: KNN Points

is

$$p(y_i = c \,|\, \mathcal{D}) = \frac{1}{K} \sum_{n \in \mathcal{N}_{K,i}} \mathbb{I}\left(y_n = c\right), \tag{6.2}$$

where $\mathcal{N}_{K,i}$ is the **set** of K nearest neighbors associated with incoming feature set $\boldsymbol{x}_i$. (6.2) is just a fancy way of saying: "look at the closest neighbors and sum the number of classifications into buckets. Assign the probability of the new classification based on these normalized sums."

⋆ Homework 8, Problem 1: KNN Probability of Classification

Assume Fig. 14 shows the *K-nearest neighbors* associated with an **incoming** data point.

(a) Apply (6.2) to the points in Fig. 14 to compute $p(y_i = 1)$ and $p(y_i = 0)$. Which one is higher? What do the probabilities sum to?

(b) The solution here actually corresponds to the MLE for a Bernoulli Distribution. Based on your solution from (a), construct the Bernoulli distribution from (2.24). **Hint**: in (2.24), x is the binary classification. What is θ? It is the probability of $p(y_i = 1)$.

Solution.

(not posted yet)

We summarize KNN in Alg. 8. The **training** of KNN is simple: just store all training data in memory! There is no other training involved. All of the hard work occurs at test/inference time.

6.1.1 K-Nearest Neighbors Drawbacks

We mention three key drawbacks associated with the KNN method [2].

Algorithm 8 K-Nearest Neighbors

Input: Training data $\mathcal{D}$, unclassified input data point $\boldsymbol{x}_{\text{new}}$

1: Find the distance d_i between $\boldsymbol{x}_{\text{new}}$ and each point $\boldsymbol{x}_i$ in $\mathcal{D}$.
2: Sort all distances (shortest to longest), and put into vector $\boldsymbol{d}$
3: Build a corresponding classification vector $\boldsymbol{y}$, such that y_i is the classification of the point at distance d_i
4: Take the first K elements of $\boldsymbol{y}$
5: Set y_{new} to the **most common classification** in $\{y_1, y_2, ..., y_K\}$

1. Inductive Bias. KNN simply looks for the **closest** points to a new data points. However, it has no capacity to distinguish important features from unimportant features. Thus, if you ask KNN to predict someone's height, and you given it input features of "age" and "favorite number", it will think these two features are equally important! Clearly, though, "favorite number" has no effect on/capability for influencing a person's height.

2. Feature Scaling. KNN is exclusively concerned with ***distance*** as a metric for making classification decisions. Thus, it is very sensitive to units and feature scaling. As shown in [2], for example, if the **units** of a feature that are usually expressed in mm are converted to cm, then the associated dimension can effectively **disappear** (since the data can become compressed into a very narrow channel, and all sense of distance in that dimension is lost).

3. Scalability in High Dimensions. KNN is a **lazy** method. At training time, KNN stores all training data in memory, but it doesn't do anything else! At test/inference time, it goes to work, finding the closest K points. In high dimensions, this can become a very cumbersome task. More fundamentally, though, weird things start to happen in high dimensions [2]. In particular, assume you have some randomly distributed data between the numbers 0 and 1. In order to capture 10% of this data:

- In 1 dimension, you just need to study 10% of the line to capture 10% of the data (statistically)
- In 2 dimensions, you need to capture 31.6% of the dimensions, since $(0.316 \times 0.316)/1^2 = 10\%$
- In 3 dimensions, you need to capture 46.4% of the dimensions, since $(0.464 \times 0.464 \times 0.464)/1^3 = 10\%$

This trend continues, and you end up needing to look very far away to find the "nearest neighbors" in high dimensions! Heuristic methods, like clustering, hashing, and dimensionality reduction, help to oversome these challenges.

6.2 Logistic Regression

Logistic regression is the "classification version" of linear regression. Under the hood, it trains a linear regression model, but this regression model doesn't predict continuous outputs; instead, it encodes something called a **linear decision boundary**. Nonlinear activations functions then map the distance from this decision boundary to **classification probabilities**.

6.2.1 Binary Classification

In binary classification tasks, logistic regression wraps a sigmoid (or a *logistic*) activation function around the output of a linear regression model. Given some input features $\boldsymbol{x}$, logistic regression

computes a probability:

$$z = \boldsymbol{w}^T \boldsymbol{x} : \qquad \text{(Apply Linear Regression Model)} \tag{6.3}$$

$$p = \frac{1}{1+e^{-z}} : \qquad \text{(Apply Sigmoid Activation Function).} \tag{6.4}$$

The output here, p, corresponds to the **probability**[§] (i.e., it falls between 0 and 1) of a given classification decision (by convention, is the probability of a "1" classification). Combining these equations, we have the standard logistic regression model, which predicts the probability of an output classification y, given the regression model parameters $\boldsymbol{w}$ and the input features $\boldsymbol{x}$:

$$p(y=1|\,\boldsymbol{w},\boldsymbol{x}) = \frac{1}{1+e^{-\boldsymbol{w}^T\boldsymbol{x}}} : \quad \textbf{Logistic Regression (Binary Classification).} \tag{6.5}$$

Of course, the probability of a 0 classification is just $p(y=0|\,\boldsymbol{w},\boldsymbol{x}) = 1 - p(y=1|\,\boldsymbol{w},\boldsymbol{x})$.

⋆ Homework 8, Problem 2: Probability of a "0" Classification

Consider the logistic regression model (6.5). Show that

$$p(y=0|\,\boldsymbol{w},\boldsymbol{x}) = \frac{1}{1+e^{+\boldsymbol{w}^T\boldsymbol{x}}}. \tag{6.6}$$

How is this similar/different from (6.5)? **Hint**: Use $p(y=0|\,\boldsymbol{w},\boldsymbol{x}) = 1 - p(y=1|\,\boldsymbol{w},\boldsymbol{x})$.

Solution.

[§]We may also interpret the sigmoid activation function (6.4) as the Cumulative Distribution Function (CDF) of a logistic distribution.

(not posted yet)

More generally, we can write $p(y \,|\, \boldsymbol{w}, \boldsymbol{x})$, $y \in \{0, 1\}$ via

$$p(y|\, \boldsymbol{w}, \boldsymbol{x}) = \left(\frac{1}{1 + e^{-\boldsymbol{w}^T \boldsymbol{x}}} \right)^y \left(1 - \frac{1}{1 + e^{-\boldsymbol{w}^T \boldsymbol{x}}} \right)^{1-y}. \tag{6.7}$$

Why does this work? Try plugging y in for $y \in \{0, 1\}$:

$$p(y = 0 \,|\, \boldsymbol{w}, \boldsymbol{x}) = 1 - \frac{1}{1 + e^{-\boldsymbol{w}^T \boldsymbol{x}}} \tag{6.8}$$

$$p(y = 1 \,|\, \boldsymbol{w}, \boldsymbol{x}) = \frac{1}{1 + e^{-\boldsymbol{w}^T \boldsymbol{x}}}. \tag{6.9}$$

Amazingly, we recognize (6.7) as a **Bernoulli distribution** from (2.24). As a reminder, the Bernoulli distribution equation returns the probability of an event occurring, or not, based on the outcome status y (or, in this case, classification of 1 or 0).

6.2.2 Training Logistic Regression Models

How do we train a logistic regression model? The loss function that is commonly used to train a logistical regression model is called *cross-entropy*, and it was previously seen in (2.65). Motivating this loss function is done most easily by considering the continuous Bernoulli distribution [1] defined in (6.7). Let's say we have some observed data classification y_1, y_2, ..., y_N and associated feature vectors $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, ..., $\boldsymbol{x}_N$. The likelihood function associated with this data (assuming a logistic

regression model predicts their probabilities) is given by

$$\text{likelihood function:} \quad p(\boldsymbol{y} \,|\, \boldsymbol{w}, \boldsymbol{x}) = \prod_{i=1}^{N} p(y_i \,|\, \boldsymbol{w}, \boldsymbol{x}_i) \tag{6.10a}$$

$$= \prod_{i=1}^{N} \left(\frac{1}{1 + e^{-\boldsymbol{w}^T \boldsymbol{x}_i}} \right)^{y_i} \left(1 - \frac{1}{1 + e^{-\boldsymbol{w}^T \boldsymbol{x}_i}} \right)^{1-y_i}. \tag{6.10b}$$

As we have done may times now, we want to **maximize** this likelihood function (à la MLE). This is accomplished, as usual, via application of the Negative Log Likelihood. To simplify the notation, let us define $\hat{p}_i = \frac{1}{1+e^{-\boldsymbol{w}^T \boldsymbol{x}_i}}$, which is the i^{th} prediction (i.e., probability).

$$\text{NLL}(p(\boldsymbol{y} \,|\, \boldsymbol{w}, \boldsymbol{x})) = -\log \left(\prod_{i=1}^{N} (\hat{p}_i)^{y_i} \, (1 - \hat{p}_i)^{1-y_i} \right) \tag{6.11a}$$

$$= -\sum_{i=1}^{N} \log \left(\hat{p}_i^{y_i} \right) - \sum_{i=1}^{N} \log \left((1 - \hat{p}_i)^{1-y_i} \right) \tag{6.11b}$$

$$= -\sum_{i=1}^{N} y_i \log \left(\hat{p}_i \right) - \sum_{i=1}^{N} (1 - y_i) \log \left((1 - \hat{p}_i) \right) : \quad \textbf{Cross-Entropy Loss} \tag{6.11c}$$

where, again, y_i is either a 1 or a 0, so many of the summation terms are eliminated. **To train a logistic regression model**, we minimize this cross-entropy function:

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} \quad -\sum_{i=1}^{N} y_i \log \left(\hat{p}_i \right) - \sum_{i=1}^{N} (1 - y_i) \log \left((1 - \hat{p}_i) \right) \tag{6.12a}$$

$$\text{s.t.} \;\; \hat{p}_i = \frac{1}{1 + e^{-\boldsymbol{w}^T \boldsymbol{x}_i}}. \tag{6.12b}$$

Inside the cross-entropy function, when the model picks the "wrong" classification for a given sample, the negative log terms approach $+\infty$ (i.e., the penalty terms will "**blow up**"). This is depicted in Fig. 15.

Unfortunately, there is **no closed form solution** to (6.12). This is disappointing, since linear regression has such a nice, closed form solution. **Gradient descent**, therefore, is commonly used to to train logistic regression models. In order to apply gradient descent, we can apply **backpropagation**, which is essentially the chain rule from calculus. We now briefly review chain rule. Given a set of sequential functions

$$x \to f(x) \to y \to g(y) \to z \to h(z) \to c, \tag{6.13}$$

we may "nest" these functions via $c = h(g(f(x)))$. Then, to find the sensitivity of the output, c, with respect to the input, x, we may solve for the partial derivatives of each mapping, and then multiple them all together:

$$\frac{\partial c}{\partial x} = \frac{\partial c}{\partial h} \cdot \frac{\partial h}{\partial g} \cdot \frac{\partial g}{\partial f} \cdot \frac{\partial f}{\partial x}. \tag{6.14}$$

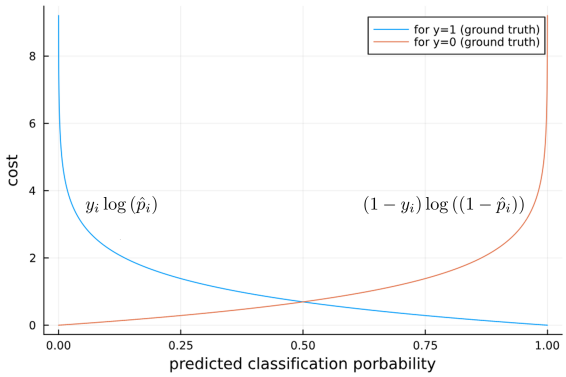


Figure 15: Cross-entropy explosion.

We now apply this same procedure in order to compute the gradient of the objective function in (6.12). First, we **split** the function into three nested parts:

1. Cross-entropy function: $f(\hat{p}) = -y_i \log(\hat{p}) - (1-y_i)\log((1-\hat{p}))$.
2. Sigmoid function: $\hat{p}(z) = \frac{1}{1+e^{-z}} = \sigma$
3. Linear regression: $z(\boldsymbol{w}) = \boldsymbol{w}^T \boldsymbol{x}_i$.

The gradient we are looking for is $\frac{\partial}{\partial \boldsymbol{w}} f(\hat{p}(z(\boldsymbol{w})))$. We compute this by noting that

$$\frac{\partial f}{\partial \hat{p}} = -y_i \frac{1}{\hat{p}} + (1-y_i)\frac{1}{1-\hat{p}} \tag{6.15}$$

$$\frac{\partial \hat{p}}{\partial z} = \sigma(1-\sigma) \tag{6.16}$$

$$\frac{\partial z}{\partial \boldsymbol{w}} = \boldsymbol{x}_i. \tag{6.17}$$

Stringing these three together, and noting that $\sigma = \hat{p} = \frac{1}{1+e^{-\boldsymbol{w}^T \boldsymbol{x}_i}}$, we have

$$\frac{\partial f}{\partial \hat{p}}\frac{\partial \hat{p}}{\partial z}\frac{\partial z}{\partial \boldsymbol{w}} = \sum_{i=1}^{N} \underbrace{\left(-y_i \frac{1}{\sigma} + (1-y_i)\frac{1}{1-\sigma}\right)}_{\frac{\partial f}{\partial \hat{p}}} \underbrace{\sigma(1-\sigma)}_{\frac{\partial \hat{p}}{\partial z}} \underbrace{\boldsymbol{x}_i}_{\frac{\partial z}{\partial \boldsymbol{w}}} \tag{6.18a}$$

$$= \sum_{i=1}^{N} \left(-y_i \frac{\sigma(1-\sigma)}{\sigma} + (1-y_i)\frac{\sigma(1-\sigma)}{1-\sigma}\right)\boldsymbol{x}_i \tag{6.18b}$$

$$= \sum_{i=1}^{N} \left(-y_i(1-\sigma) + (1-y_i)\,\sigma\right)\boldsymbol{x}_i \tag{6.18c}$$

$$= \sum_{i=1}^{N} (\sigma - y_i)\,\boldsymbol{x}_i. \tag{6.18d}$$

Thus, we have

$$\nabla_{\boldsymbol{w}} = \sum_{i=1}^{N} \left(\frac{1}{1+e^{-\boldsymbol{w}^T \boldsymbol{x}_i}} - y_i\right)\boldsymbol{x}_i : \quad \textbf{Gradient of Logistic Regression Loss}. \tag{6.19}$$

This gradient can be used to numerically optimize (6.12) via gradient descent.

6.2.3 Probabilistic Decision Boundaries

Once we have a training logistic regression model, we may use it to find **decision boundaries associated with specific probabilities**. For example, we may find the hyperplane associated with $p = 0.25$, meaning any point falling "below" this hyperplane will have a classification probability of < 0.25, while any point falling "above" this hyperplane will have a classification probability of > 0.25. To find this hyperplane, (i) take a trained logistic regression model, (ii) set it equal to some probability $0 < p < 1$, and then (iii) solve for the the coefficients. **Remember**: hyperplanes generally take the form "$\boldsymbol{w}^T \boldsymbol{x} + b = 0$".

⋆ **Homework 8, Problem 3: General Expression for Hyperplane of Probability**

Given a **trained** logistic regression model,

$$p = \frac{1}{1 + e^{-\boldsymbol{w}^T \boldsymbol{x}}}, \tag{6.20}$$

find the hyperplane associated with some probability $p^\star$. What is the "$+b$" term?

Solution.

(not posted yet)

6.2.4 Nonlinear Decision Boundaries

Just like linear regression, logistic regression may perform **poorly** when there is a true, nonlinear relationship between the features and the outputs. In these cases, we say that data is **not linearly separable**. To overcome this challenge, we may kernelize the data with, e.g., a polynomial kernel. For example: $\phi(x_1, x_2) = [1, x_1, x_1^2, x_2, x_2^2, x_1 x_2]^T, \quad D = 2$. Next, we plug $\phi(\boldsymbol{x})$ into the otherwise-standard logistic regression training problem:

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} \quad -\sum_{i=1}^{N} y_i \log\left(\hat{p}_i\right) - \sum_{i=1}^{N} (1 - y_i) \log\left((1 - \hat{p}_i)\right) \tag{6.21a}$$

$$\text{s.t. } \hat{p}_i = \frac{1}{1 + e^{-\boldsymbol{w}^T \phi(\boldsymbol{x}_i)}}, \tag{6.21b}$$

where the dimension of $\boldsymbol{w}$ will **expand** to meet the dimension $\phi(\boldsymbol{x})$. This optimization problem can be solved by any solver which can solve a **standard** logistic regression problem.

6.2.5 Multiclass Classification

In multi-class classification, there can be K **potential output classifications**. To predict the classification probabilities, we then employ the **softmax function** (2.70), which is a *generalization* of the logistic (or sigmoid) function in higher dimensions:

$$\text{sm}(\boldsymbol{z}) = \begin{bmatrix} \frac{e^{z_1}}{\sum_j e^{z_j}} \rightarrow \text{Probability of classification 1} \\ \frac{e^{z_2}}{\sum_j e^{z_j}} \rightarrow \text{Probability of classification 2} \\ \vdots \\ \frac{e^{z_K}}{\sum_j e^{z_j}} \rightarrow \text{Probability of classification } K \end{bmatrix}, \tag{6.22}$$

where $\boldsymbol{z}$ is an input vector. Where does this input vector come from? It is the output of a series of **linear prediction models** (like linear regression models) which we build *inside* of the multiclass logistic function. The structure of the multiclass logistic function follows:

$$\boldsymbol{x} \rightarrow \begin{cases} \boldsymbol{w}_1^T\boldsymbol{x} \rightarrow z_1 \rightarrow \frac{e^{z_1}}{\sum_j e^{z_j}} \rightarrow \text{class 1 probability: } \hat{p}^{(1)} \\ \boldsymbol{w}_2^T\boldsymbol{x} \rightarrow z_2 \rightarrow \frac{e^{z_2}}{\sum_j e^{z_j}} \rightarrow \text{class 2 probability: } \hat{p}^{(2)} \\ \vdots \\ \boldsymbol{w}_K^T\boldsymbol{x} \rightarrow z_K \rightarrow \frac{e^{z_K}}{\sum_j e^{z_j}} \rightarrow \text{class K probability: } \hat{p}^{(K)} \end{cases} : \textbf{Multiclass Logistic} \tag{6.23}$$

The standard cross-entropy loss function in (6.11c) can only accommodate binary classifications (1 or 0). In order to apply cross-entropy loss in the context of multiclass classification, we first split the data K times, as demonstrated in Fig. 17. For each split, we introduce a binary variable $y_i^{(k)} \in \{0, 1\}$, which says: "Does sample i belong to classification k?" Since we do this N times, there will be a total of $N \times K$ of these new variables. We then use these new binary indicators to pose the following multi-class cross-entropy loss function:

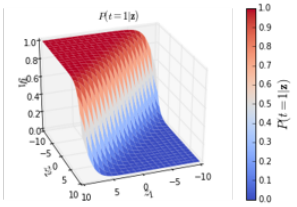


Figure 16: Softmax.

$$\mathcal{L}_{\text{mc}} = \sum_{i=1}^{N}\sum_{k=1}^{K} -y_i^{(k)} \log\left(\hat{p}_i^{(k)}\right) : \quad \textbf{Multiclass Cross-Entropy Loss} \tag{6.24}$$

where N is the number of training samples, K is the number of potential output classifications.

⚠ Notationally, we use $y_i^{(k)}$ as a **binary variable** indicating class membership in a multiclass setting, but we use y_i, as in (6.12), in a binary classification setting. These are equivalent.

⋆ Homework 8, Problem 4: Binary vs Multi-Class Cross-Entropy

Show that, for $K = 2$ (i.e., binary classification), the multiclass cross entropy function of (6.24) is equivalent to the binary cross-entropy function in (6.11c). **Hint**: What should the two probabilities sum to? What should the two classification binaries sum to?

Solution.

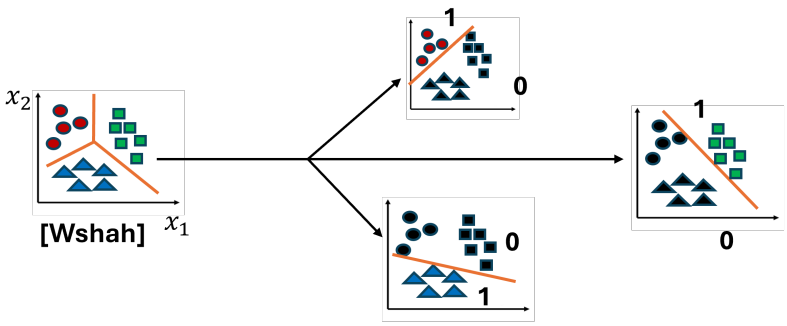


Figure 17: Multiclass classification data splits. Call the **set** of red circle samples $\mathcal{R}$. The red circles in the first (i.e., upper-most) split will have $y_i^{(k=1)} = 1$, $i \in \mathcal{R}$ since they belong to this class ($k = 1$). In the other two splits, $y_i^{(k=2)} = 0$, and $y_i^{(k=3)} = 0$, $i \in \mathcal{R}$.

(not posted yet)

6.3 Support Vector Machines

Support Vector Machines (SVMs) are an essential machine learning tool for finding "**separating hyperplanes**" between sets of labeled training data. We consider a set of training data given as

$$\mathcal{D} = \{(\boldsymbol{x}_i, y_i) \,|\, y_i \in \{+1, -1\}, i = 1, 2, ..., N\}, \tag{6.25}$$

where output labels are designated as $+1$ or -1. We will parameterize the SVM hyperplanes using a vector of slopes $\boldsymbol{w}$ and a bias parameter w_0 **Note:** in this section, the bias term is not included in the vector of slopes $\boldsymbol{w}$. Future test points will be classified depending on which side of the hyperplane they fall. Unlike its cousin logistic regression, SVM is **not a probabilistic/statistical**

classifier [1]. It is, instead, a deterministic "widest margin" classifier. SVMs have a beautiful history, as nicely summarized by Patrick Winston **in this clip** (skip to 46 minutes).

6.3.1 Hard Margin Classifier

A large margin, also called a "hard margin", classifier works by choosing a separating hyperplane which introduces the "largest margin" between points of different classifications. In particular, our classification prediction will be given by

$$\hat{y}_i = \begin{cases} +1, & \boldsymbol{w}^T \boldsymbol{x}_i + w_0 \geq 0 \\ -1, & \boldsymbol{w}^T \boldsymbol{x}_i + w_0 < 0, \end{cases} \tag{6.26}$$

where $\boldsymbol{x}_i$ are the features associated with some new test point, and $\hat{y}_i$ is its predicted classification. We refer to $\boldsymbol{w}^T \boldsymbol{x} + w_0$ as a linear decision rule. (6.26) can be stated more succinctly via the "sign" function, which maps positive and negative values to +1 and -1, respectively:

$$\hat{y} = \text{sign}(\boldsymbol{w}^T \boldsymbol{x} + w_0). \tag{6.27}$$

In SVM, our goal is to find $\boldsymbol{w}$ and w_0 such that the **margin** between the two classification sets (training data) is **maximized**. To do this, we first re-define

$$f(\boldsymbol{x}) \triangleq \boldsymbol{w}^T \boldsymbol{x} + w_0. \tag{6.28}$$

According to (1.18), $\frac{f(\boldsymbol{x})}{\|\boldsymbol{w}\|_2}$ gives us the signed distance between some input point $\boldsymbol{x}$ and the separating hyperplane. Additionally, given y_i, we want $y_i(\boldsymbol{w}^T \boldsymbol{x}_i + w_0) > 0$, $\forall i$ (since positive times positive, and negative times negative, both yield a positive). Thus, we may pose the following optimization:

$$\max_{\boldsymbol{w}, w_0} \min_i \; y_i \left(\frac{\boldsymbol{w}^T \boldsymbol{x}_i + w_0}{\|\boldsymbol{w}\|_2} \right), \tag{6.29}$$

where $\min_i$ looks for the smallest item in the set by looping over i. (6.29) says, "Find the values of $\boldsymbol{w}, w_0$ such that the **minimum** distance to the separating hyperplane is **maximized**." Why minimum? Well, we care most about the points closest to the decision boundary, since these are the "support vectors". Next, we pull the denominator out, since it does not depend on i:

$$\max_{\boldsymbol{w}, w_0} \frac{1}{\|\boldsymbol{w}\|_2} \min_i \; y_i \left(\boldsymbol{w}^T \boldsymbol{x}_i + w_0 \right) \tag{6.30}$$

Rescaling. It is common to rescale this problem into a more standard format[¶]. To do so, we note that we can arbitrarily scale the hyperplane equation so that its smallest output (i.e., where a data-point is closest to the boundary) is "scaled up" to 1:

$$\frac{1}{\epsilon} \left(\boldsymbol{w}^T \boldsymbol{x}_i + w_0 = \epsilon \right) \quad \rightarrow \quad \left(\frac{1}{\epsilon} \boldsymbol{w} \right)^T \boldsymbol{x}_i + \frac{1}{\epsilon} w_0 = 1. \tag{6.31}$$

[¶]This rescaling does not **normalize** the data in the problem; you have to do that directly, if want to. Instead, it puts the problem into a standard format which is easier to manipulate.

If this is the case, we know that $\min_i \; y_i \left(\boldsymbol{w}^T \boldsymbol{x}_i + w_0\right) = 1$ will be satisfied, and we can repose the problem with this constraint added explicitly:

$$\max_{\boldsymbol{w}, w_0} \frac{1}{\|\boldsymbol{w}\|_2} \tag{6.32a}$$

$$\text{s.t. } y_i \left(\boldsymbol{w}^T \boldsymbol{x}_i + w_0\right) \geq 1, \; \forall i \in \{1...N\}. \tag{6.32b}$$

Don't worry: the scaling ϵ terms are applied to all model coefficients and effectively cancel out[‖]. Finally, maximizing $\frac{1}{\|\boldsymbol{w}\|_2}$ will have the same **maximizer** (i.e., solution) as the **minimizer** for $\frac{1}{2}\|\boldsymbol{w}\|_2^2$. Thus, we update the problem:

$$\min_{\boldsymbol{w}, w_0} \frac{1}{2}\|\boldsymbol{w}\|_2^2 : \quad \textbf{Large Margin SVM Training Problem} \tag{6.33a}$$

$$\text{s.t. } y_i \left(\boldsymbol{w}^T \boldsymbol{x}_i + w_0\right) \geq 1. \tag{6.33b}$$

This is a **linearly constrained quadratic program**. Furthermore, it is **convex**, so modern optimization tools can solve (6.33) to it global optimum in a generally efficient manner. Once this model is constructed, it can be **deployed** to classify new points via (6.26).

Minimizing Slopes in the SVM

The vector $\boldsymbol{w}$ represents the **slope** terms of the SVM decision boundary. Why do we minimize these slopes? Mathematically, the **margin** between the points and the decision boundary is proportional to $1/\|\boldsymbol{w}\|_2$, so we **maximize** this while **maintaining separation** between the classes.

There is one hiccup, though: (6.33) is not necessarily **feasible** (i.e., solvable), since the optimizer may not necessarily be able to find a hyperplane which satisfies the given constraints; it will only be feasible when the data is linearly separable. Furthermore, even if a data set is linearly separable, the hard margin classifiers can be very **sensitive to outliers**. A single outlier can cause very strange support vectors to be chosen.

6.3.2 Soft Margin Classifier

The soft margin classifier gives the constraints a little bit of "slack" to be violated; however, this slack is **penalized**. We apply this nonnegative slack, $\xi \geq 0$, to the constraint in (6.33) via $y_i \left(\boldsymbol{w}^T \boldsymbol{x}_i + w_0\right) \geq 1 - \xi_i$. Thus, if $y_i \left(\boldsymbol{w}^T \boldsymbol{x}_i + w_0\right)$ needs to grow **smaller** than 1, it is "covered" by the slack. *But*, the optimizer "pays a price" for this in the objective function:

$$\min_{\boldsymbol{w}, w_0} \frac{1}{2}\|\boldsymbol{w}\|_2^2 + c\sum_i \xi_i : \quad \textbf{Soft Margin SVM Training Problem} \tag{6.34}$$

$$\text{s.t. } \xi_i \geq 0 \tag{6.35}$$

$$y_i \left(\boldsymbol{w}^T \boldsymbol{x}_i + w_0\right) \geq 1 - \xi_i, \quad \forall i. \tag{6.36}$$

[‖] Just pose the following problem, and see that ϵ cancels:

$$\max_{\boldsymbol{w}, w_0} \frac{1}{\left\|\frac{1}{\epsilon}\boldsymbol{w}\right\|_2} \min_i \; y_i \left(\frac{1}{\epsilon}\boldsymbol{w}^T \boldsymbol{x}_i + \frac{1}{\epsilon} w_0\right).$$

In this formulation, the slack terms are summed together and then multiplied by c, a penalty constant. The smaller this constant, the more the SVM will "tolerate" outliers. Of course, the optimizer will try its best to drive the slack terms to 0 (since it tries to minimize penalties, but slack can't go lower than 0).

⋆ Homework 8, Problem 5: Penalty Term Limit

Consider the penalized, soft margin SVM training problem (6.34).

(a) Drive $c \to \infty$. Is the resulting problem similar to (6.33)? Why?

(b) How should you choose c? Suggest a procedure.

Solution.

(not posted yet)

6.3.3 The Dual SVM

In this subsection, we are going to take the dual of the SVM problem, which will allow us to elegantly "kernelize" the problem, as we shall see. We now return to the hard margin SVM (the following analysis is **applicable** to soft margin SVM too). The SVM training problem in (6.33) is posed as a **constrained optimization problem**. We call this a **primal** problem (as opposed to a **dual** problem). To solve this constrained optimization problem, we can take the steps outlined after we formed the Lagrangian way back in (1.42). First, we build the Lagrangian by "**dualizing**" the inequality constraints:

$$\max_{\alpha \geq 0} \min_{\boldsymbol{w}, w_0} \frac{1}{2} \|\boldsymbol{w}\|_2^2 + \sum_{i=1}^{N} \alpha_i \left(1 - y_i \left(\boldsymbol{w}^T \boldsymbol{x}_i + w_0\right)\right). \tag{6.37}$$

Here, we have used α as a dual variable instead of μ in order to be consistent with the SVM literature. Simplifying the objective function, we have

$$\mathcal{L} = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + \sum_{i=1}^{N}\alpha_i - \sum_{i=1}^{N}\alpha_i y_i \boldsymbol{w}^T\boldsymbol{x}_i - \sum_{i=1}^{N}\alpha_i y_i w_0. \tag{6.38}$$

Next, we apply a first order optimality condition to solve the inner minimization problem, which we leave as a homework problem.

⋆ Homework 8, Problem 6: Dual SVM Formulation

Show that the **dual** form of primal SVM problem is given by

$$\max_{\alpha \geq 0} \sum_{i=1}^{N}\alpha_i - \frac{1}{2}\sum_{j=1}^{N}\sum_{i=1}^{N}\alpha_i y_i \alpha_j y_j \boldsymbol{x}_i^T\boldsymbol{x}_j \tag{6.39a}$$

$$\text{s.t.} \sum_{i=1}^{N}\alpha_i y_i = 0, \ \forall i. \tag{6.39b}$$

Hint: Take the gradient of (6.38) with respect to the primal variables, set it to 0, and solve. Then, plug in. Make sure to keep, and use, the $\frac{\partial \mathcal{L}}{\partial w_0} = 0$ optimality condition! **Second hint**: $\sum_i x_i \sum_j y_j = \sum_i \sum_j x_i y_j$, by properties of FOIL.

Solution.

(not posted yet)

Unfortunately, we need to stop here. There is **no closed-form solution** of the maximization problem in (6.39). But that's ok, because we have another trick up our sleeves: **the kernel trick**.

A quick note on α: Before moving on, we quickly note that typically, almost all of the α terms are 0. Why? Well, if we apply the so-called **KKT conditions** (which are slightly *outside* the scope of this class), we have something called a "**complementary slackness**" condition. This tell us that, at optimality, the solution to (6.37) satisfies

$$\alpha_i \left(1 - y_i \left(\boldsymbol{w}^T \boldsymbol{x}_i + w_0\right)\right) = 0, \quad \forall i. \tag{6.40}$$

But we also know that $y_i \left(\boldsymbol{w}^T \boldsymbol{x}_i + w_0\right) = 1$ is only satisfied at the **support vectors**. Therefore, at all nonsupport vectors, $y_i \left(\boldsymbol{w}^T \boldsymbol{x}_i + w_0\right) > 1$. By (6.40), the corresponding α_i must satisfy $\alpha_i = 0$. Thus, α_i at all nonsupport vectors must be zero.

6.3.4 The Kernel Trick

There is *much* written on the so-called **kernel trick** that we will now introduce. However, one take on it is this: sometimes, there are hard ways to compute something, and there easier, but *non-obvious*, ways to compute something. Take the easier route, when you can! Following is an analogous example from linear algebra.

Kernel Trick Analogy

Consider the following **matrix-matrix-vector product**:

$$y = ABx, \tag{6.41}$$

where A and B are large known matrices, and x is a known vector. How should we **compute** y? If we move left-to-right, we will compute a matrix-matrix product ($C = AB$), followed by a matrix-vector product $y = Cx$. Numerically, this is expensive (matrix-matrix products are $\mathcal{O}(n^3)$)! Instead, we can compute $y = A(Bx)$, which is just two matrix-vector products.

- Hard way: $y = (AB)x$
- Easier way: $y = A(Bx)$

This "easier way" is analogous to the "kernel trick". It is cheaper to compute, numerically, but it **computes the same thing**, in the end. **Note:** the kernel trick does not exploit matrix multiplication; this example is just an *analogy* of computing something you care about more easily through clever manipulation.

SVM is powerful tool, but if we want to apply it in the context of **nonlinearity**, we need to *kernelize* the data somehow. When we do this in high dimensions with lots of data, SVM becomes very hard to solve. **For example**, if we apply a degree $D = 2$ polynomial kernelization to a problem with three input features, we get 10 features we need to train over:

$$\phi(x_1, x_2, x_3) = [1, x_1, x_1^2, x_2, x_2^2, x_3, x_3^2, x_1x_2, x_1x_3, x_2x_3]^T, \quad D = 2. \tag{6.42}$$

Feature space explosion is a real challenge! However, let's look very closely at the **dual SVM problem objective function**:

$$\sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{j=1}^{N} \sum_{i=1}^{N} \alpha_i \alpha_j y_i y_j \underbrace{\boldsymbol{x}_i^T \boldsymbol{x}_j}_{\text{features}}. \tag{6.43}$$

If we change the **features** that we train on, nothing about this objective function changes, **except** for the last term, where we take a dot product of the features. This term, essentially, captures the **similarity** of two features. If two feature vectors are **orthogonal** to each other, it means the vector are very dissimilar, so $\boldsymbol{x}_i \perp \boldsymbol{x}_j \rightarrow \boldsymbol{x}_i^T \boldsymbol{x}_j = 0$

We may replace this term, therefore, with something called a positive definite **Mercer Kernel function**, which also capture some **generalized notional** of kernelized **similarity**. We define the Mercer Kernel as something which takes a **dot product** of two transformed vectors $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$:

$$\mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}_j) \triangleq \phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}_i) : \quad \textbf{Mercer Kernel}, \tag{6.44}$$

where the transformation is given by $\phi(\boldsymbol{x}_i)$. Notably, the Mercer Kernel is *symmetric*: $\mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \mathcal{K}(\boldsymbol{x}_j, \boldsymbol{x}_i)$, since the mapping is the same for both vectors.

Kernels and kernels and kernels and ...

The word "**kernel**" is used many places in mathematics. So far, in these notes, we have used kernel to mean a **nonlinear mapping to a new, *lifted* feature space**. This is a very general process, and it can be done in many ways. In this subsection, we discuss Mercer Kernels, formally defined here [23], which are a very specific sort of kernelization shown in (6.44). This is, admittedly, confusing, because $\phi(\boldsymbol{x}_i)$ can itself be called a kernel (it's **kernels** all the way down).

So, who cares? We care for the following reason: the kernel mapping of $\phi(\boldsymbol{x})$ might lead a **very high dimensional variable space** which we want to embed in the SVM training problem:

$$\sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{j=1}^{N} \sum_{i=1}^{N} \alpha_i \alpha_j y_i y_j \underbrace{\phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}_j)}_{\text{kernelized features}} . \tag{6.45}$$

However, in 6.45, we never actually use $\phi(\boldsymbol{x}_j)$ on its own: we only need the dot product $\phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}_j)$. And it turns out, there are *faster* ways of computing $\phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}_j)$ than computing it directly, as we will see in the following example.

⋆ Example 18: $D = 2$ Polynomial Kernel for SVM

Let's say we want to train an SVM classifier, and we want to use a degree $D = 2$ polynomial kernel for an input feature space with two features: x_1 and x_2. Naively, we would just define the mapping:

$$\phi(\boldsymbol{x}) = \left[1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{2}x_2 x_1\right]^T , \tag{6.46}$$

and then we would **plug it in** to the SVM objective (the $\sqrt{2}$ terms will be explained soon). For example, given a sample $\boldsymbol{a}$ and a sample $\boldsymbol{b}$ (for notational clarity), the SVM objective would compute

$$\phi(\boldsymbol{a})^T \phi(\boldsymbol{b}) = \left[1, \sqrt{2}a_1, \sqrt{2}a_2, a_1^2, a_2^2, \sqrt{2}a_2 a_1\right]^T \left[1, \sqrt{2}b_1, \sqrt{2}b_2, b_1^2, b_2^2, \sqrt{2}b_2 b_1\right] \tag{6.47a}$$

$$= 1 + 2a_1 b_1 + 2a_2 b_2 + a_1^2 b_1^2 + a_2^2 b_2^2 + 2a_2 a_1 b_2 b_1. \tag{6.47b}$$

Instead, let's try something else: let's just compute the original dot product from the SVM,

but let's square it:

$$\begin{aligned}
\left(\boldsymbol{a}^T\boldsymbol{b}\right)^2 &= \left([1, a_1, a_2]^T\,[1, b_1, b_2]\right)^2 && \text{(6.48a)}\\
&= (1 + a_1b_1 + a_2b_2)^2 && \text{(6.48b)}\\
&= 1 + 2a_1b_1 + 2a_2b_2 + a_1b_1a_1b_1 + a_2b_2a_1b_1 + a_1b_1a_2b_2 + a_2b_2a_2b_2 && \text{(6.48c)}\\
&= \left(1 + 2a_1b_1 + 2a_2b_2 + a_1^2b_1^2 + a_2^2b_2^2 + 2a_2a_1b_2b_1\right). && \text{(6.48d)}
\end{aligned}$$

What do we see? (6.47b) and (6.48d) are the same, meaning, for this particular example,

$$\phi(\boldsymbol{a})^T\phi(\boldsymbol{b}) = \left(\boldsymbol{a}^T\boldsymbol{b}\right)^2. \tag{6.49}$$

But, what was easier to compute? $\left(\boldsymbol{a}^T\boldsymbol{b}\right)^2$ is much easier! And it gets even more advantageous to do this when $D = 3, 4, 5....$

Is the kernel trick just to save a few pennies on computation? No, it's more fundamental than that. For large polynomial degrees, kernel mappings can lead to very high dimensional features! But, there are some kernels that are literally *infinite dimensional*. For example, the Gaussian **radial basis function** (RBF) is a very commonly used kernel in SVM:

$$\mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}_j) = e^{-\gamma\|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2^2}. \tag{6.50}$$

The Taylor series expansion of (6.50) has an infinite number of terms, meaning the associated feature space is **infinite dimensional**. Following are list of commonly used Kernel functions [3]; each one is supported by sci-kit learn. Each one of these kernels, to some degree, measures **feature vector similarity**.

1. **Linear Kernel**: $\mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{x}_i^T\boldsymbol{x}_j$.
2. **Polynomial Kernel**: $\mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}_j) = (\gamma\boldsymbol{x}_i^T\boldsymbol{x}_j + r)^d$.
3. **Gaussian RBF Kernel**: $\mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}_j) = e^{-\gamma\|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2^2}$
4. **Sigmoid Kernel**: $\mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \tanh\left(\gamma\boldsymbol{x}_i^T\boldsymbol{x}_j + r\right)$.

⋆ Homework 9, Problem 1: A Simple Polynomial Kernel Function

As a classic example, let's just take the direct square of **two input features** without any added bias terms:

$$\begin{aligned}
\left(\boldsymbol{a}^T\boldsymbol{b}\right)^2 &= \left([a_1, a_2]^T\,[b_1, b_2]\right)^2 && \text{(6.51a)}\\
&= (a_1b_1 + a_2b_2)^2 && \text{(6.51b)}\\
&= a_1^2b_1^2 + 2a_2b_2a_1b_1 + a_2^2b_2^2. && \text{(6.51c)}
\end{aligned}$$

What is the kernel mapping $\phi(\cdot)$ which would yield $\phi(\boldsymbol{a})^T\phi(\boldsymbol{b}) = a_1^2b_1^2 + 2a_2b_2a_1b_1 + a_2^2b_2^2$? i.e., $\phi(\boldsymbol{a}) =?$, $\phi(\boldsymbol{b}) =?$

Solution.

(not posted yet)

6.3.5 Mapping a Kernelized SVM Back to a Classification Prediction

To make a prediction with *standard* SVM, we typically just compute

$$\textbf{SVM classification: } \operatorname{sign}\left(\boldsymbol{w}^T\boldsymbol{x} + w_0\right). \tag{6.52}$$

But, this means we need to compute $\boldsymbol{w}$ and w_0 after we have solved the dual problem. We can compute $\boldsymbol{w}$ from the dual via the stationarity condition applied to (6.38), and we can compute w_0 by taking the support vectors (where $y_i\left(\boldsymbol{w}^T\boldsymbol{x}_i + w_0\right) = 1$ is satisfied) and computing w_0 directly. Which are the support vectors? These are the points where $\alpha_i > 0$. When a constraint becomes active, its dual variable becomes nonzero, so $\alpha_i > 0$ allows you to find these activated constraints (i.e., the support vectors). Typically, you average across all support vectors to compute w_0:

$$w_0 = \frac{1}{n_s}\sum_{\forall i\,:\,\alpha_i>0}^{N}\frac{1}{y_i} - \boldsymbol{w}^T\boldsymbol{x}_i \tag{6.53}$$

where n_s is the number of support vectors, and $\frac{1}{y_i} = y_i$ (why?).

However, these steps are **only relevant for the standard, non-kernelized SVM**. When we deal with *kernelized* SVM, our classification decision is based on

$$\textbf{Kernelized SVM classification: } \operatorname{sign}\left(\boldsymbol{w}^T\phi(\boldsymbol{x}) + w_0\right), \tag{6.54}$$

where the **kernelized model parameters** are given by

$$\boldsymbol{w} = \sum_{i=1}^{N} \alpha_i y_i \phi(\boldsymbol{x}_i) \tag{6.55a}$$

$$w_0 = \frac{1}{n_s} \sum_{\forall i \,:\, \alpha_i > 0}^{N} y_i - \boldsymbol{w}^T \phi(\boldsymbol{x}_i). \tag{6.55b}$$

But... what if we have used a **super high dimensional** kernelization? Or an infinite dimensional one? Is there a simpler way to make a classification decision **without** computing $\phi(\boldsymbol{x})$ and $\boldsymbol{w}$ directly? Indeed, there is.

⋆ **Homework 9, Problem 2: Kernelized SVM Classification**

The Kernelized SVM prediction is computed via (6.54). Using (6.55), show that

$$\boldsymbol{w}^T\phi(\boldsymbol{x}) = \sum_{i=1}^{N} \alpha_i y_i \mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}), \tag{6.56}$$

and that

$$w_0 = \frac{1}{n_s} \sum_{\forall i\,:\,\alpha_i>0}^{N} \left(y_i - \sum_{j=1}^{N} \alpha_j y_j \mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}_j) \right). \tag{6.57}$$

This last one is a bit messy, but **note**: this bias term just needs to be computed one time! It does not depend on the input features $\boldsymbol{x}$.

Solution.

(not posted yet)

Using these results, we may make Kernelized SVM classification decisions via

$$\textbf{K-SVM class: } \operatorname{sign}\left(\sum_{i=1}^{N} \alpha_i y_i \mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}) + \frac{1}{n_s} \sum_{\forall i\,:\,\alpha_i>0}^{N} \left(y_i - \sum_{j=1}^{N} \alpha_j y_j \mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}_j) \right) \right). \tag{6.58}$$

6.4 Classification Accuracy Metrics

To assess the quality of a **trained** classification model, it is common to build a **Confusion Matrix**. This matrix will be construed using withheld test data (i.e., data that the model **has not seen**

previously). Its typical structure is shown in Fig. 18.

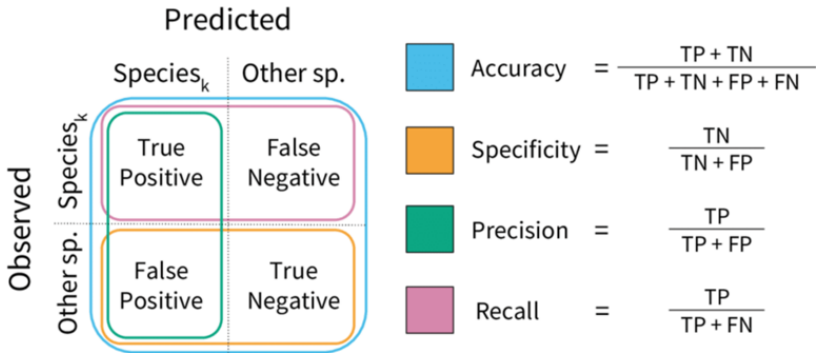


Figure 18: Confusion Matrix and Associated Scores [LinkedIn].

Along with overall classifier **accuracy**, there are many, many other classification metrics that have been invented. Two key ones are **precision** (i.e., accuracy of the positive predictions) and **recall** (i.e., ratio of positive instances that are correctly predicted).

⋆ Homework 9, Problem 3: Gaming Precision and Recall

Let's say you are build a classifier that predicts if a hand written digit is a 5 or not, and you want to "game" the metrics.

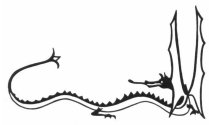
(a) How can you get a perfect **precision** score? Practically, how would you train your classifier to achieve this?

(b) How can you get a perfect **recall** score? Practically, how would you train your classifier to achieve this?

Hint: use the definitions provided in Fig. 18

Solution.

(not posted yet)



7 Trees and Forests

In this section, we consider **decision trees**, **ensemble methods**, **random forests**, **bagging** and **boosting** methods, and **XGBoost**. The notes in the section are fairly sparse, compared to other sections; readers are referred to chapters 6 and 7 of [3] for additional details.

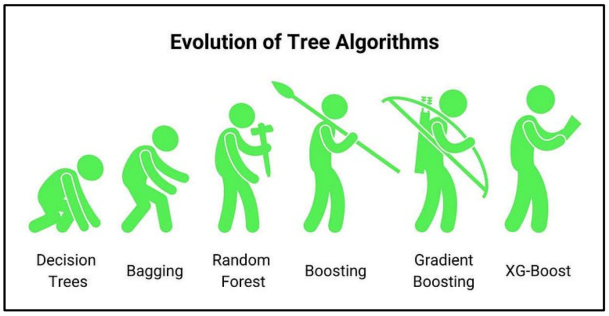


Figure 19: Evolution of tree algorithms. [AIML].

7.1 Decision Trees (DTs)

Decision trees are a ***non-parametric*** supervised learning tool (meaning, their size generally **scales** with the number of input data samples that they train on). Generally, if left **unregularized**, decision trees will grow and grow and grow, until they fit a given data set with 100% accuracy. To make classification or regression decisions, decision inspect a set of feature, and then they "split" on features which maximize the **purity** of the resulting subsets of data. **Entropy** and **Gini index** are two common metrics which measure the purity of subset of data.

Given a set of N data samples, we define p_k as the probability of selecting an item with classification k randomly as

$$p_k = \frac{N_k}{N}, \tag{7.1}$$

where N_k is the number of items in the subset with classification k. The Gini index is defined via

$$G_i = 1 - \sum_k p_k^2 : \quad \textbf{Gini Index.} \tag{7.2}$$

Entropy, meanwhile, is defined via

$$\mathbb{H} = -\sum_k p_k \log_2(p_k) : \quad \textbf{Entropy.} \tag{7.3}$$

As explained previously, entropy measures the amount of **surprise** in a dataset, while **Gini index** measure the error rate associated with selecting the wrong item from a group of items.

⋆ Homework 9, Problem 4: Gini Index and Error Rate

You are given a set of N items with varying classifications 1, 2,..., K. You close your eyes and select one of them at random, hoping for an item with a specific classification. Show that the

error rate (i.e., the rate of not getting what you want) is equal to

$$G_i = 1 - \sum_k p_k^2. \tag{7.4}$$

Hint: To solve this problem, we must multiply the probability of classification occurrence (p_k) by the probability of misclassification ($1 - p_k$), and we need to do this for all possible classifications.

Solution.

(not posted yet)

Given the left and right hand side splits on a given features, the overall "split quality" is given as the following weighted averages:

$$\text{split quality (Gini index)} = \frac{N_L}{N} G_{i,L} + \frac{N_R}{N} G_{i,R} \tag{7.5}$$

$$\text{split quality (Entropy)} = \frac{N_L}{N} H_{i,L} + \frac{N_R}{N} H_{i,R}, \tag{7.6}$$

where N is the overall number of features on both sides, N_L is the number of features on the left side of the split, and N_R is the number of features on the right side of the split. Fig. 20 show an example of split purities: when the decision trees splits on feature 1 (humidity), the Gini index and entropy scores are both **zero**, since there is no surprise in the data (the subsets are pure).

7.1.1 DT Regularization

Decision trees are grown heuristically, one level at a time: they scan across all features, then make the best splitting decision. They do this over and over, until either (i) all of the data is **perfectly**

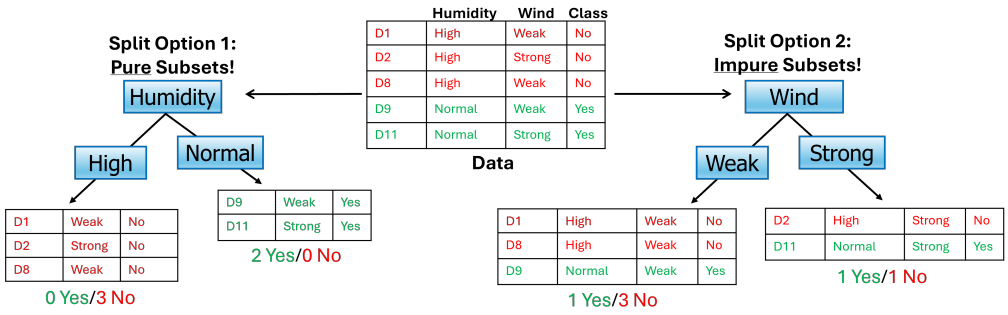


Figure 20: Decision Tree Split Purity.

fit, or (ii), regularization tells it to stop growing. Two popular regularization techniques are known as **max-depth** regularization, which controls the maximum number of tree layers, and **max-leaf** regularization, which controls the maximum number of leaf nodes.

7.1.2 DT with Continuous Features

When a feature is **continuous** in nature (e.g., wind speed), DTs will scan across all possible feature values from the associated dataset, and then choose to split on the feature value from this set (e.g., $x \leq 5.2$) that leads to **optimal subset purity**.

7.1.3 Regression Trees

DTs can also solve regression problems with two simple modifications. First, Gini index and entropy are replaced with **Mean Square Error** to measure subset purities. Second, the leaf node output predictions are made based on the average across the subsets. For example, if training elements y_1, y_2, and y_3 all flow to the same leaf node j, then the regression prediction made by that leaf node would be $w_j = (y_1 + y_2 + y_3)/3$.

7.1.4 DT Advantages and Disadvantages

According to [1], DT have the following **pros**:

- Highly interpretable
- Easily handle discrete + continuous data
- Data normalization isn't needed
- Robust to outliers
- Scale well; easy to fit big data sets
- Can handle missing data.

However, the also have various **drawbacks**:

- Training methods are greedy, so test results can be **bad**
- They can be overly sensitive to data/unstable.

7.2 Ensemble Methods

In 1906, **Sir Francis Galton** attended a fair, where he observed a contest. In this contest, fairgoers were guessing the weight of an ox. As a pompous member of elite society, he assumed the aggregation of many **uninformed weight guesses**, made by **uninformed guessers**, would result in an **uninformed aggregate guess** (i.e., the average would be way off). However, **he was wrong**: the average guess was very, very close to the true weight of the ox! This story demonstrates the "wisdom of the crowd", and it serves to motivate the idea behind **ensemble methods**. Such methods average the predictions across many models. None of these models will be correct 100% of the time. However, as long as **most** of the models are correct **most** of the time, the ensemble average achieve very good performance.

To show a specific mathematical example of this phenomena, we define a **weak classifier** as a classifier whose performance is slightly **better** than random guessing, or coin flipping.

⋆ Example 19: Aggregation of Three Weak Classifiers

We want to show that the aggregation of three **weak**, but **independent**, classifiers will actually yield a slightly stronger classifier. First, consider a single classifier, whose probability of making the correct classification is given by

$$p_1 = \theta = 0.51: \quad \text{probability of correct classification.} \tag{7.7}$$

We could, of course, use a Bernoulli distribution to define the probability of a correct classification more formally, but let's just stick with (7.7) for simplicity. Since there is only one classifier, the space of prediction combinations only has two elements: it makes a correct classification, or it makes an incorrect classification.

Now, let's consider three, *independent* classifiers:

$$p_1 = \theta \tag{7.8a}$$
$$p_2 = \theta \tag{7.8b}$$
$$p_3 = \theta. \tag{7.8c}$$

We assume these classifiers vote on a given classification decision (which is why we consider an odd number of classifiers – an even number can tie when voting happens!). We can envision the prediction combination space as a 3-dimensional box. We want to compute the "area" of the box corresponding to a correct classification vote. This area is composed of the situation where all three classifiers are correct (θ^3), and the area where two are correct and one is wrong (this can happen three times: $3(1-\theta)\theta^2$. Thus, the total probability (area) of a correct classification vote is given by

$$p_c = \underbrace{\theta^3}_{\text{all three correct}} + 3\underbrace{(1-\theta)\theta^2}_{\text{two are correct}} \tag{7.9a}$$
$$= (0.51)^3 + 3(0.49)(0.51)^2 \tag{7.9b}$$
$$= 0.514998. \tag{7.9c}$$

This isn't too much better than the individual weak classifier, but it is **slightly better**.

⋆ Homework 9, Problem 5: Aggregation of FIVE Weak Classifiers

We now want to find the probability of correct classification if **five** independent weak classifiers vote on a classification decision, where the probabilities of a correct vote are given by

$$p_1 = p_2 = p_3 = p_4 = p_5 = \theta, \quad \theta = 0.51. \tag{7.10}$$

(a) What is the probability of a correct classification, p_c, if these five classifiers vote? **Hint**: study the previous example, but extend it to the 5-dimensional case; how many ways can the five classifiers vote and still get a correct decision? Take the product of the probabilities of each of these situations.

(b) Assume you want a 90% classification accuracy based on this vote. What value should θ take? **Hint**: set $p_c = 0.9$ with θ unknown, and then use e.g., `fsolve` from `scipy.optimize`, or something similar, to solve the equation.

Solution.

(not posted yet)

Voting: Ensemble methods can use **hard voting** (majority rule) or **soft voting** (where votes are weighted by model certainty/probability). They can also combine different sorts of classification methods (e.g., SVD, DT, KNN, etc), or use **homogeneous** models (e.g., 10 DTs). Ultimately, though, the goal is to train models which have some degree of **independence**. When models make similar errors, ensemble methods work poorly; but when model make different sorts of error, they flourish, since the "**wisdom of the crowd**" can overcome the mistakes of any individual model.

7.3 Bagging

Bagging, which stands for "bootstrap aggregating" trains a series of models of parallel, where each model is trained on a random subset of the data. With bagging, the data is sampled randomly **with**

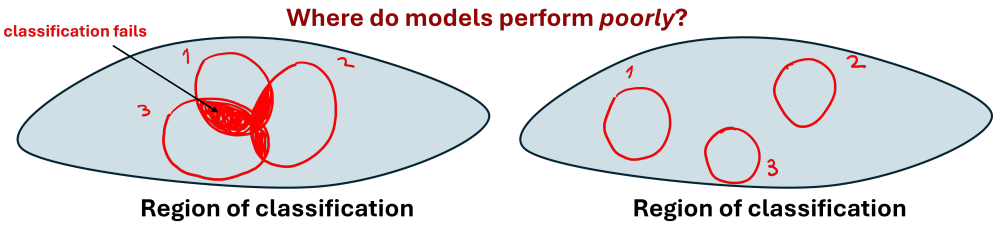


Figure 21: Example of two ensembles of models. On the left, the regions where the models **perform poorly** overlap, so voting will result in incorrect classification decisions. On the right, there is no overlap between the regions where the models perform poorly, so hard voting will always result in **correct classification decisions**.

replacement (i.e., models can potentially see the same individual data points, but it is unlikely that any two models will see the exact same random subset of data). Bagging-based methods help increase the **diversity** and **independence** of the models, and the models can be trained in parallel. These are both major **benefits** of bagging!

7.4 Random Forests

Random forests are simply an **ensemble** of **decision trees**! On top of bagging-based training, we also sprinkle in a secondary source of randomness: feature splitting. Rather than scanning across all possible feature splits, and looking for the very best feature to split on, random forest learners grab a **random subset** of the features, and then they find the best feature to split on within this random subset.

7.5 Boosting Methods

Boosting methods start with a weak learner and then **sequentially improve** the performance of the weak learner. We consider two flavors of boosting: **AdaBoost**, which improves learning performance by boosting the relative weight of misclassified samples, and Gradient Boosting, which trains sequential models on the residual error of previous models.

7.5.1 AdaBoost

In AdaBoost, we start by equally initializing the **weights** w associated with a set of training N samples: $w_i = 1/N$. These weights will tell the optimizer where to focus its attention; initially, all training samples are **equally important**. Next, we train a weak learner model $F(\boldsymbol{x})$ (e.g., decision tree) to correctly classify points based on a loss function which incorporates these weight somehow. For example, the **exponential** loss function

$$\mathcal{L} = \sum_{i=1}^{N} w_i e^{-y_i F(\boldsymbol{x}_i)}, \tag{7.11}$$

with true labels $y_i \in \{-1, 1\}$ and predicted labels $F(\boldsymbol{x}_i) \in \{-1, 1\}$, will **overemphasize** correctly classifying points with large weights. To compute these weights, we take a trained weak learner j, and then compute the weighted **error rate**:

$$r_j = \frac{\sum_{i=\text{incorrect}} w_i}{\sum_{i=\text{all}} w_i}, \; w_i = \text{sample weight}, \; r_j = \text{error rate for model } j, \tag{7.12}$$

where "i = incorrect" is the set of **incorrect classifications** made by the weak learner. Next, we compute the **model weight** associated with this weak learner model:

$$\alpha_j = \eta \frac{1 - r_j}{r_j}, \ \alpha_j = \text{model weight}, \ \eta = \text{learning rate}. \tag{7.13}$$

As $r_j \to 1$, $\alpha_j \to 0$. Next, we use this model weight to update sample weights:

$$w_i = \begin{cases} w_i & , \text{ sample } i = \text{correct} \\ w_i e^{\alpha_j} & , \text{ sample } i = \text{incorrect}. \end{cases} \tag{7.14}$$

Using these **new sample weights**, we jump back to the loss function (7.11) and train a new weak learner. Finally, once a sufficiently large number of weak learners have been trained, we deploy the model. The predicted classification, $\hat{y}$, is given by

$$\hat{y} = \text{sign}\left(\sum_{j=1}^{N} \alpha_j F_j(\boldsymbol{x}) \right), \quad F_j(\boldsymbol{x}) \in \{-1, +1\} \tag{7.15}$$

for **binary classification** problems, assuming weak learners make $\{-1, +1\}$ predictions, or

$$\hat{y} = \arg\max_k \sum_{j:\hat{y}_j = k}^{N} \alpha_j, \tag{7.16}$$

in **multiclass classification** problems, where $\hat{y}_j$ is the predicted classification of the j^{th} model. Thus, (7.16) chooses the classification k which has the largest sum over all **models weights** α_j.

⋆ Homework 10, Problem 1: AdaBoost Classification Vote

Assume we have N classification models, each with weight α_j, and each makes a classification prediction, $\hat{y}_j$, from the set of possible classifications $\{1, 2, ..., K\}$. Show or explain why (7.16) yields the correct classification vote decision.

Solution.

(not posted yet)

7.5.2 Gradient Boosting

While AdaBoost trains models by **boosting** the relative importance of incorrect samples, gradient boosting **trains** new models based on the **residual error** of previously trained models. Strong mathematical motivation is given in [1]. The general approach is this: start with weak learner, $F_0(\boldsymbol{x})$, and then **train a new weak learner**, $F(\boldsymbol{x})$, on the **residual errors** made by the first weak learner: $r = F_0(\boldsymbol{x}) - y$. Then, **update the ensemble** with the new weak learner $F(\boldsymbol{x})$:

$$\underbrace{f_m(\boldsymbol{x})}_{\text{new ensemble}} = \underbrace{f_{m-1}(\boldsymbol{x})}_{\text{old ensemble}} + \eta \underbrace{F(\boldsymbol{x})}_{\text{new weak learner}} \tag{7.17}$$

where η is a step size, or a **shrinkage factor** which scales downward the contribution of new weak learners. What does any of this have to do with **gradients** (i.e., gradient boosting)? Consider a **regression model** $F_{r0}(x)$ and its **mean square error**:

$$e = \frac{1}{2}\left(y - F_{r0}(\boldsymbol{x})\right)^2 . \tag{7.18}$$

The gradient of this error is given by $\frac{\partial e}{\partial F} = F_{r0}(x) - y$. If we train a new weak learner on this residual, and then we update the ensemble via (7.17), then we can **interpret** (7.17) as a **gradient descent step** which minimizes the overall prediction error.

7.5.3 XGBoost and LightGBM

Finally, Extreme Gradient Boosting, or **XGBoost** [24], is a **highly optimized** version of gradient boosted random forests. It is both a **learning method**, which incorporates novel regularization and accelerated learning tricks, and a **software library**. It has show dominant performance in Kaggle

ML competitions, and it's a great choice if you want excellent out-of-the-box ML performance on a highly complex dataset. Its cousin, **LightGBM**, is a faster implementation which grows trees in a slightly different way.

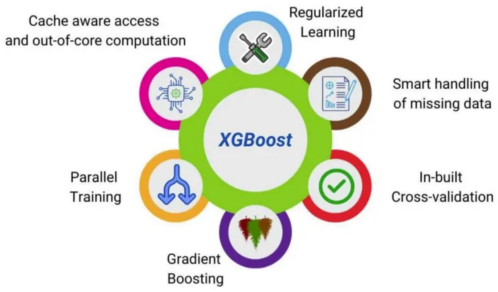


Figure 22: XGBoost Features [Medium].

8 Neural Networks

The story of Neural Networks starts in 1943, when **McCulloch and Pitts** published their paper, "**A Logical Calculus and the Ideas Immanent in Nervous Activity**". In it, they proposed using a brain-like architecture, composed of **neurons** and **synapses**, in order to execute **propositional logic**. Foundationally, neurons exhibit an "all or nothing" response; a neuron will only "fire" and send an electrical signal down its axon only when the electrical stimulus within the cell body **exceeds** a certain **threshold**. Figure (23) shows a biological model of a **neuron** and the associated **mathematical model** we use to mimic its behavior.

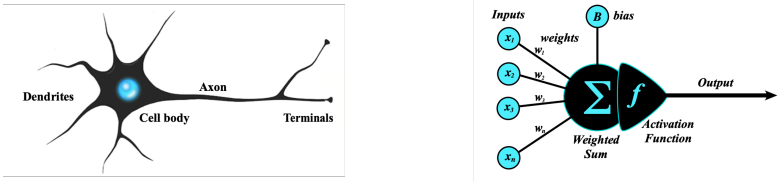


Figure 23: Quoted from MRIQs: "A biological nerve cell receives input stimuli from neighboring nerves through its **dendrites**. If the sum of these stimuli is sufficient to create **membrane depolarization** in the neuron's cell body, an electrical output signal will be transmitted down the **axon** to its terminals (which in turn may stimulate dendrites of other nerves)."

8.1 Multilayer Perceptron

A **Neural Network** refers to a broad class of learning architectures which roughly mimic the structure of the brain. The **perceptron** model, on the other hand, is a very specific NN building block which, essentially, mimics an individual neuron:

$$f_p(\boldsymbol{x}) = H(\boldsymbol{w}^T\boldsymbol{x} + w_0): \quad \textbf{Perceptron Model} \tag{8.1}$$

where $H(\cdot)$ is a Heaviside step-function:

$$H(x) = \begin{cases} 1 & x > 0 \\ 0 & x < 0 \end{cases}: \quad \textbf{Heaviside Step Function.} \tag{8.2}$$

It is noted in [1] that the perceptron model (8.1) may be considered a **non-probabilistic logistic regression** model (i.e., it just outputs a hard classification decision, rather than a probability). While the Heaviside step function is non-differentiable, perceptron models may be **trained** via a **modified gradient descent** routine, where we assume the gradient of the step function is just equal to 1:

$$w_j = w_j + \eta\,(y_i - \hat{y}_i)\,x_j: \quad \textbf{Perceptron Training Routine} \tag{8.3}$$

where y_i is a binary classification label, $\hat{y}_i$ is the prediction of the perceptron model, and η is a step size. By looping over all perceptron parameters j and all all data points i, (8.3) is famously **guaranteed to converge** [Minsky] when the training data is linearly separable [3].

⋆ **Example 20: Training Perceptrons with Modified Gradient Descent**

Assume we want to **train** a perceptron model by solving

$$\min_{\boldsymbol{w},w_0} \sum_{i=1}^{N} \frac{1}{2} \left(y_i - \hat{y}_i\right)^2 = \min_{\boldsymbol{w},w_0} \sum_{i=1}^{N} \frac{1}{2} \left(y_i - H\left(\boldsymbol{w}^T \boldsymbol{x}_i + w_0\right)\right)^2. \tag{8.4}$$

Taking the gradient of the cost function with respect to w_j, we have

$$g_j = \sum_{i=1}^{N} \left(y_i - \hat{y}_i\right)(-1)\frac{\partial H}{\partial z} x_i, \tag{8.5}$$

where $\frac{\partial H}{\partial z}$ is the gradient of the step function. If we set this to 1, we have $g_j = -\sum_{i=1}^{N} \left(y_i - \hat{y}_i\right) x_i$, which is exactly the update rule provided in (8.3) (except with a summation over all data points).

Perceptrons can perfectly classify **linearly separable** data sets, since they are essentially logistic regression models with *very sharp* probability boundaries. They can also be used to perform **logical** calculations, as we will see in the following problem.

⋆ Homework 10, Problem 2: Logical Operations with the Perceptron

Consider the following perceptron model:

$$z = H\left(w_0 + w_1 x + w_2 y\right), \tag{8.6}$$

where $H(\cdot)$ is the **Heaviside** operator, and w_0, w_1, and w_2 are tunable model parameters. x and y are the inputs. **Specify these parameters** to create a perceptron model of the following logical operators:

(a) x `and` y

(b) x `or` y

(c) x `nand` y

Hint: the solutions here are non-unique. Find something that works!

Solution.

(not posted yet)

While simple logic is permissible, more complex logic **cannot** be performed by a single layer perceptron model. For example, the logical **XOR** (exclusive or) operation **cannot** be performed. This was famously pointed out by Minsky and Papert in their book **Perceptrons: An Introduction to Computational Geometry**. In this case, we need multiple perceptrons (either stacked on top of each other, or placed sequentially). When we add multiple layers onto a perceptron model, we call it a **multilayer perceptron** (MLP):

$$f_{\text{mlp}}(\boldsymbol{x}) = H(W_N \cdots H(W_2 H(W_1 \boldsymbol{x} + w_1) + w_2) \cdots w_N) : \quad \textbf{Multilayer Perceptron.} \tag{8.7}$$

While the MLP has greatly increased **representational power** over a single perceptron, it is still non-differentiable. Furthermore, the simple update rule (8.3) proposed earlier will no longer work. To overcome these challenges, [25] in 1986 proposed two innovations: (1) replacing the Heaviside step function with a **smooth approximation** (the logistic/sigmoid function), and (*ii*) **backpropagating** the error through the NN to get the sensitivity of the training error with respect to various MLP parameters (weights and biases).

8.2 Neural Network Activation Functions

While [25] proposed replacing the Heaviside step function with **logistic function**, there are many, many **other activation functions** that are popular, some of which are listed in the following box.

Popular Activation Functions

Popular activation functions include the following:

$$\textbf{Linear}: \quad y = x = I(x) \tag{8.8a}$$

$$\textbf{Heaviside Step}: \quad y = \begin{cases} 1, & x > 0 \\ 0, & x < 0 \end{cases} \tag{8.8b}$$

$$\textbf{ReLU}: \quad y = \max(x, 0) \tag{8.8c}$$

$$\textbf{Leaky ReLU}: \quad y = \begin{cases} x, & x > 0 \\ -0.01x, & x < 0 \end{cases} \tag{8.8d}$$

$$\textbf{ELU}: \quad y = \begin{cases} x, & x \geq 0 \\ \alpha(e^x - 1), & x < 0 \end{cases} \tag{8.8e}$$

$$\textbf{Sigmoid (Logistic)}: \quad y = \frac{1}{1 + e^{-x}} = \sigma(x) \tag{8.8f}$$

$$\textbf{Tanh}: \quad y = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1. \tag{8.8g}$$

Why don't we just use "**linear**" activation functions? The following problem considers the result of just using linear activations.

⋆ Homework 10, Problem 3: "Linear" Activation Functions

Consider the **NN mapping**

$$\text{NN}(\boldsymbol{x}) = W_3 I\left(W_2 I\left(W_1 \boldsymbol{x} + b_1\right) + b_2\right) + b_3 : \mathbb{R}^n \to \mathbb{R}^1 \tag{8.9}$$

where $I(\cdot)$ is the "linear" activation function.

(a) Show that (8.9) reduces to a linear regression model $\boldsymbol{w}^T \boldsymbol{x} + w_0$.

(b) What are $\boldsymbol{w}$ and w_0?

Solution.

(not posted yet)

8.3 Backpropagation

We use **gradient descent**, or one of the many variations reviewed in sub-subsection 4.7.4, to train MLPs and Neural Networks (NNs). To compute these gradients, we use **backpropagation**, which is the **workhorse** of modern ML technology. The process of computing NN gradients generally has three steps:

(a) First, compute a **forward pass** through the NN. In a forward pass, we take a data sample, or a batch of data samples, and we pass them through the NN, computing the NN output(s) and all **intermediate** states.

(b) Second, we pass the outputs into the loss function (e.g., MSE, or Cross-Entropy), thus computing the **error**.

(c) Finally, using chain rule, we move **backwards** through the network, computing gradients of the error with respect to trainable network parameters as we go.

The **forward pass** is necessary due to the nonlinearity of activations functions: the gradient of a nonlinear function depends on *where* the function is evaluated (e.g., $\nabla x^2 = 2x$). The final step of using chain rule can be greatly aided by **decomposing** the NN mapping into a series of easily differentiable mappings, and then multiplying the results together, as was demonstrated back in (6.13)-(6.14). Next, we provide a specific example of this.

⋆ **Example 21: Gradient of Nested Function.**

We want to compute the **gradient** of the function

$$f(x) = \sin\left(e^{x^2-1}\right). \tag{8.10}$$

Applying chain rule in one shot can be very error prone. Instead, we decompose the function:

$$f = \sin(z) \tag{8.11}$$
$$z = e^y \tag{8.12}$$
$$y = x^2 - 1. \tag{8.13}$$

Next, we compute the gradient via chain rule:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial z}\frac{\partial z}{\partial y}\frac{\partial y}{\partial x} \tag{8.14a}$$
$$= \cos(z) \cdot e^y \cdot 2x. \tag{8.14b}$$

In computing this gradient's actual numerical value, we notice that we need the **values** of x, y and z, which we would compute via a forward pass **before** applying chain rule.

⋆ Homework 10, Problem 4: Backpropagation Through a 2-Layer NN

Consider the NN mapping

$$l = \left(\hat{z} - \sigma(w_2\sigma(w_1 x))\right)^2, \tag{8.15}$$

which passes an input x through a 2-layer NN ($\sigma(\cdot)$ is the **sigmoid** activation function), and into a loss function. We may decompose this mapping into a sequence of operations

$$l = z_5^2 \tag{8.16a}$$
$$z_5 = \hat{z} - z_4 \tag{8.16b}$$
$$z_4 = \sigma(z_3) \tag{8.16c}$$
$$z_3 = w_2 z_2 \tag{8.16d}$$
$$z_2 = \sigma(z_1) \tag{8.16e}$$
$$z_1 = w_1 x. \tag{8.16f}$$

Given $x = 1$, $w_1 = 2$, $w_2 = -2$, and $\hat{z} = 1$, use backpropagation, i.e., chain rule to compute the following by hand:

(a) Compute $\frac{\partial l}{\partial w_2}$.

(b) Compute $\frac{\partial l}{\partial w_1}$.

(c) Now w_1 is mistakenly **initialized** to $w_1 = 100$. Does the gradient $\frac{\partial l}{\partial w_1}$ **vanish**? How about $\frac{\partial l}{\partial w_2}$? Recompute these gradients and comment.

Hint: use the mapping provided to apply chain rule, and reuse $\frac{\partial l}{\partial w_2}$ when you compute $\frac{\partial l}{\partial w_1}$!

Solution.

(not posted yet)

AutoDiff Flavors: There are **two flavors** of NN differentiation (also know as Automatic Differentiation, or Auto-Diff): **reverse mode** differentiation, and **forward mode** differentiation. Reverse mode is generally faster when we have **many inputs and one output** (which is the case with NNs, which have a single loss function output), and forward mode is generally faster in the opposite case: when we have **many outputs and few inputs**. The next problem will explore conditions under which one is faster than the other. Generally, though, reverse-diff is faster than forward-diff for the same reason that $A\left(B\boldsymbol{x}\right)$ is faster to compute than $\left(AB\right)\boldsymbol{x}$.

Jacobians: Finally, in order to perform backpropagation, we need the concept of the **Jacobian**. A Jacobian, J, is essentially just a set of gradients stacked in separate columns. When we want to take the gradients of multiple functions f_1, ..., f_m, with respect to multiple variables x_1, ..., x_n, we get a Jacobian $J \in \mathbb{R}^{m \times n}$ of the following form (where any given column is just a gradient vector):

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}. \tag{8.17}$$

As a simple **example**, the Jacobian of $\boldsymbol{y} = A\boldsymbol{x}$, where A is a constant matrix, is given by $J = A$.

⋆ Homework 10, Problem 5: Forward Mode or Reverse Mode Differentiation?

Consider the NN mapping

$$z = W_2\sigma(W_1x + b_1) + b_2 : \mathbb{R}^m \to \mathbb{R}^n, \tag{8.18}$$

where there are m **inputs**, and n **outputs**. To compute the **backpropagation** through this NN, we separate the mapping into a series of steps:

$$z = z_3 \tag{8.19}$$
$$z_3 = W_2z_2 + b_2 \tag{8.20}$$
$$z_2 = \sigma(z_1) \tag{8.21}$$
$$z_1 = W_1x + b_1. \tag{8.22}$$

The sensitivity on z with respect to x is given by

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial z_3}\frac{\partial z_3}{\partial z_2}\frac{\partial z_2}{\partial z_1}\frac{\partial z_1}{\partial x} \tag{8.23}$$

(a) Write $\frac{\partial z}{\partial x}$ as a function of the actual gradients (i.e., plug the correct gradients and Jacobians into (8.23)). For the activation function, assume you have access to a diagonal Jacobian I_σ.

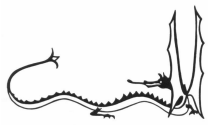
(b) Assume $m = 1$ and $n >> 1$ (one input, many outputs). Is it faster to compute (8.23) with forward mode or reverse mode? Why?

(c) Assume $m >> 1$ and $n = 1$ (many inputs, one outputs). Is it faster to compute (8.23) with forward mode or reverse mode? Why?

Solution.

> (not posted yet)

That's all folks.



9 Machine Learning Verification

9.1 Interval Bound Propagation

9.2 Neural Network Convex Relaxations

9.3 Branch and Bound

10 Appendix

10.1 Optimal PCA Data Imputation

We consider the problem

$$\min_{\boldsymbol{x}_?} \left\| \boldsymbol{x} - VV^T\boldsymbol{x} \right\|_2^2 \tag{10.1}$$

where $\boldsymbol{x}$ has two components: known data $\boldsymbol{x}_*$, and unknown data $\boldsymbol{x}_?$. We expand the objective function

$$\begin{aligned}
\left(\boldsymbol{x} - VV^T\boldsymbol{x}\right)^T \left(\boldsymbol{x} - VV^T\boldsymbol{x}\right) &= \boldsymbol{x}^T\boldsymbol{x} + \boldsymbol{x}^T \left(VV^T\right)^T VV^T\boldsymbol{x} - 2\boldsymbol{x}^T VV^T\boldsymbol{x} && \text{(10.2a)}\\
&= \boldsymbol{x}^T\boldsymbol{x} + \boldsymbol{x}^T VV^T VV^T\boldsymbol{x} - 2\boldsymbol{x}^T VV^T\boldsymbol{x} && \text{(10.2b)}\\
&= \boldsymbol{x}^T\boldsymbol{x} + \boldsymbol{x}^T VV^T\boldsymbol{x} - 2\boldsymbol{x}^T VV^T\boldsymbol{x} && \text{(10.2c)}\\
&= \boldsymbol{x}^T\boldsymbol{x} - \boldsymbol{x}^T VV^T\boldsymbol{x}. && \text{(10.2d)}
\end{aligned}$$

For simplicity, we denote VV^T as M, and we decompose this into blocks:

$$\begin{bmatrix} M_1 & M_2 \\ M_2^T & M_3 \end{bmatrix} = M \triangleq VV^T. \tag{10.3}$$

Using this block structure, we further decompose the problem the objective function:

$$\begin{aligned}
\mathcal{L} &\triangleq \boldsymbol{x}^T\boldsymbol{x} - \boldsymbol{x}^T VV^T\boldsymbol{x} && \text{(10.4a)}\\
&= \boldsymbol{x}_*^T\boldsymbol{x}_* + \boldsymbol{x}_?^T\boldsymbol{x}_? - \boldsymbol{x}_*^T M_1\boldsymbol{x}_* - \boldsymbol{x}_?^T M_3\boldsymbol{x}_? - 2\boldsymbol{x}_*^T M_2\boldsymbol{x}_? && \text{(10.4b)}
\end{aligned}$$

Next, we take its gradient, and set it to 0:

$$\begin{aligned}
0 = \frac{\partial \mathcal{L}}{\partial \boldsymbol{x}_?} &= 2\boldsymbol{x}_? - 2M_3\boldsymbol{x}_? - \left(2\boldsymbol{x}_*^T M_2\right)^T && \text{(10.5a)}\\
&= 2\left(I - M_3\right)\boldsymbol{x}_? - 2M_2^T\boldsymbol{x}_*. && \text{(10.5b)}
\end{aligned}$$

Solving this expression for $\boldsymbol{x}_?$, we have

$$\boldsymbol{x}_? = \left(I - M_3\right)^{-1} M_2^T\boldsymbol{x}_*. \tag{10.6}$$

This imputation of data will project the data vector $\boldsymbol{x}$ on the a given set of principal components.

10.2 Back-Substitution

Consider the linear system

$$\begin{bmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,n} \\ 0 & r_{2,2} & & r_{2,n} \\ \vdots & & \ddots & \cdots \\ 0 & 0 & & r_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}. \tag{10.7}$$

To solve this system, we start with the final value of x:

$$x_n = \frac{b_n}{r_{n,n}}. \tag{10.8}$$

Next, we solve the second-to-last element:

$$r_{n-1,n-1}x_{n-1} + r_{n-1,n}x_n = b_{n-1} \tag{10.9}$$

$$x_{n-1} = \frac{b_{n-1} - r_{n-1,n}x_n}{r_{n-1,n-1}}. \tag{10.10}$$

And so on.

References

[1] K. P. Murphy, *Probabilistic Machine Learning: An introduction*. MIT Press, 2022. [Online]. Available: http://probml.github.io/book1

[2] H. Daumé, *A course in machine learning*. Hal Daumé III, 2017.

[3] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. " O'Reilly Media, Inc.", 2022.

[4] M. P. Deisenroth, A. A. Faisal, and C. S. Ong, *Mathematics for machine learning*. Cambridge University Press, 2020.

[5] A. Bhattacharya, *The Man from the Future: The Visionary Ideas of John von Neumann*. WW Norton, 2023. [Online]. Available: https://books.google.com/books?id=VWyPEAAAQBAJ

[6] D. Muller. Math's fundamental flaw. Youtube. [Online]. Available: https://www.youtube.com/watch?v=HeQX2HjkcNo

[7] D. Hofstadter, *Gödel, Escher, Bach: An Eternal Golden Braid*, ser. Harvester studies in cognitive science. Penguin, 2000. [Online]. Available: https://books.google.com/books?id=grzEQgAACAAJ

[8] G. Chaitin. Gregory chaitin lecture carnegie-mellon university 2000 pt 1-8. Youtube. [Online]. Available: https://www.youtube.com/watch?v=HLPO-RTFU2o&list=PLC7C2C890974C51EC

[9] A. Gefter, "The man who tried to redeem the world with logic," January 2015. [Online]. Available: https://nautil.us/the-man-who-tried-to-redeem-the-world-with-logic-235253/

[10] A. N. Whitehead and B. Russell, *Principia mathematica to* 56*. Cambridge University Press, 1927, vol. 2.

[11] K. Gödel, "On formally undecidable propositions of principia mathematica and related systems i (1931)," in *Godel's Theorem in Focus*. Routledge, 2012, pp. 17–47.

[12] A. M. Turing *et al.*, "On computable numbers, with an application to the entscheidungsproblem," *J. of Math*, vol. 58, no. 345-363, p. 5, 1936.

[13] J. Fuegi and J. Francis, "Lovelace & babbage and the creation of the 1843 'notes'," *IEEE Annals of the History of Computing*, vol. 25, no. 4, pp. 16–26, 2003.

[14] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, pp. 115–133, 1943.

[15] S. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge university press, 2004.

[16] J. Nocedal and S. J. Wright, *Numerical optimization*. Springer, 2006.

[17] H. Li, Z. Xu, G. Taylor, C. Studer, and T. Goldstein, "Visualizing the loss landscape of neural nets," *Advances in neural information processing systems*, vol. 31, 2018.

[18] Wikipedia, "Silhouette (clustering) — Wikipedia, the free encyclopedia," http://en.wikipedia.org/w/index.php?title=Silhouette%20(clustering)&oldid=1268464861, 2025, [Online; accessed 20-January-2025].

[19] S. Chevalier, L. Schenato, and L. Daniel, "Accelerated probabilistic power flow in electrical distribution networks via model order reduction and neumann series expansion," *IEEE Transactions on Power Systems*, vol. 37, no. 3, pp. 2151–2163, 2022.

[20] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[21] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[22] S. L. Brunton, J. L. Proctor, and J. N. Kutz, "Discovering governing equations from data by sparse identification of nonlinear dynamical systems," *Proceedings of the national academy of sciences*, vol. 113, no. 15, pp. 3932–3937, 2016.

[23] B. Ghojogh, A. Ghodsi, F. Karray, and M. Crowley, "Reproducing kernel hilbert space, mercer's theorem, eigenfunctions, nystr\" om method, and use of kernels in machine learning: Tutorial and survey," *arXiv preprint arXiv:2106.08443*, 2021.

[24] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. ACM, Aug. 2016, p. 785–794. [Online]. Available: http://dx.doi.org/10.1145/2939672.2939785

[25] D. E. Rumelhart, G. E. Hinton, R. J. Williams *et al.*, "Learning internal representations by error propagation," 1985.