Easy Machine Learning

By Wyaaaattwho

Foreword

This is a simplified machine learning guidebook written entirely by myself. The initial idea was to find a path to learn machine learning independently, as our school doesn't offer any useful courses on the subject. Later, I realized I could write down what I'd learned as a book for my friends and others to read, which also serves as a way to review the material myself.

The book is primarily based on lectures from Stanford's CS229 and *Machine Learning* by Zhi-Hua Zhou. I removed some outdated or irrelevant content and added insights from blogs I've read. I typed this all in Notion myself, so there might be some typo and so on, please let me know if you get one.

If you do enjoy this book, please follow me on Github or stick up to my blog.

Index

Math Preparation

Mathematic preparations

Linear algebra

Norms

In our class we've already got to know commonly-used **Euclidean or ℓ_2 norm**, which is defined as

$$\|\mathbf{x}\|_2 = \sqrt{\left(\sum_{i=1}^{n} x_i^2\right)}$$

in fact , there could be so much more kind of norms.

More formally, a norm is any function $f : R^n \to R$ that satisfies 4 properties:

1. For all $x \in R^n, f(x) \geq 0$ (non-negativity).
2. $f(x) = 0$ if and only if $x = 0$ (definiteness).
3. For all $x \in R^n, t \in R, f(tx) = |t|f(x)$ (homogeneity).
4. For all $x, y \in R^n, f(x + y) \leq f(x) + f(y)$ (triangle inequality).

We have two special norms when denote p as 1 and ∞

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{n} |x_i|$$

$$\|\mathbf{x}\|_\infty = \max_{1 \le i \le n} |x_i|$$

Norms can also be defined for matrices, such as the **Frobenius norm**

$$\|\mathbf{A}\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^2} = \sqrt{tr(A^T A)}$$

Range and Nullspace of a Matrix

We've already learned *the span* in class which is

$$\text{span}(\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k\}) = \{\mathbf{v} \mid \mathbf{v} = c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \cdots + c_k\mathbf{v}_k,\ c_i \in \mathbb{R}\}$$

this actually shows all the possible vector that could be expressed as a linear combination of $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k\}$, which forms a linear space in a defined dimension.

From here, it's intuitive to find *the Range*, *(*sometimes also called the columnspace) of a matrix $A \in R^{m \times n}$, denoted $R(A)$, is the the span of the columns of A. In other words,

$$\mathcal{R}(\mathbf{A}) = \{\mathbf{y} \mid \mathbf{y} = \mathbf{A}\mathbf{x},\ \mathbf{x} \in \mathbb{R}^n, \mathbf{y} \in \mathbb{R}^m\}$$

that is actually the span of a matrix A.

Then , we define *the Projection*. The projection of a vector $y \in R^m$ onto the $span\ of \{x_1, ..., x_n\}$ (here we assume $x_i \in R^m$) is the vector $v \in span(\{x_1, ...x_n\})$, such that vis as close as possible to y, as measured by the Euclidean norm $\|v - y\|_2$. We denote the projection as $Proj(y; \{x_1, ..., x_n\})$ and can define it formally as,

$$\text{Proj}(\mathbf{y}; \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}) = \underset{\mathbf{v} \in \text{span}(\{\mathbf{x}_1, \ldots, \mathbf{x}_n\})}{\arg\min} \|\mathbf{y} - \mathbf{v}\|_2$$

Making a few technical assumptions (namely that A is full rank and that $n < m$), the projection of a vector $y \in R^m$ onto the range of A is given by,

$$Proj(y; A) = \arg\min_{v \in R(A)} \|v - y\|_2 = A(A^T A)^{-1} A^T y$$

later you will find this is actually the same as what we have in least squares estimation of parameters.

At last, we came to *the Nullspace.* The nullspace of a matrix $A \in R^{m \times n}$, denoted $N(A)$ is the set of all vectors that equal 0 when multiplied by A

$$\mathcal{N}(\mathbf{A}) = \{\mathbf{x} \mid \mathbf{A}\mathbf{x} = \mathbf{0},\ \mathbf{x} \in \mathbf{R}^n\}$$

it's of no hard to find that nullspace has something to with the range, that is

$$\{\mathbf{w} : \mathbf{w} = \mathbf{u} + \mathbf{v}, \quad \mathbf{u} \in \mathcal{R}(\mathbf{A}^T), \quad \mathbf{v} \in \mathcal{N}(\mathbf{A})\} = \mathbb{R}^n \quad \text{and} \quad \mathcal{R}(\mathbf{A}^T) \cap \mathcal{N}(\mathbf{A}) = \{0\}$$

In other words, $\mathcal{R}(A^T)$ and $\mathcal{N}(A)$ are disjoint subsets that together span the entire space of R^n. Sets of this type are called *orthogonal complements*.

The Gradient

Suppose that $f : R^{m \times n} \to R$ is a function that takes as input a matrix A of size m×n and returns a real value. Then *the gradient* of f (with respect to $A \in R^{m \times n}$) is the matrix of partial derivatives, defined as

$$\nabla_A f(A) \in \mathbb{R}^{m \times n} = \begin{bmatrix} \frac{\partial f(A)}{\partial A_{11}} & \frac{\partial f(A)}{\partial A_{12}} & \cdots & \frac{\partial f(A)}{\partial A_{1n}} \\ \frac{\partial f(A)}{\partial A_{21}} & \frac{\partial f(A)}{\partial A_{22}} & \cdots & \frac{\partial f(A)}{\partial A_{2n}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f(A)}{\partial A_{m1}} & \frac{\partial f(A)}{\partial A_{m2}} & \cdots & \frac{\partial f(A)}{\partial A_{mn}} \end{bmatrix}$$

if A is a vector, namely x, we have

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} \\ \frac{\partial f(\mathbf{x})}{\partial x_2} \\ \vdots \\ \frac{\partial f(\mathbf{x})}{\partial x_n} \end{bmatrix}$$

Then we need to talk about *the Hessian*, in present formula, we get a gradient of the same size as the input, yet what if we can get a matrix with just a vector input? Here's what we have

$$\mathbf{H} = \nabla_{\mathbf{x}}^2 f(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f(\mathbf{x})}{\partial x_1^2} & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_2 \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_2^2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_n^2} \end{bmatrix}$$

We can work out the formulas below (if A is symmetric):

$$\begin{aligned} &\nabla_x b^T x = b \\ &\nabla_x x^T A x = 2Ax \\ &\nabla_x^2 x^T A x = 2A \\ &\nabla_A |A| = (adj(A))^T = |A| A^{-T} \\ &\nabla_A log|A| = \frac{1}{|A|} \nabla_A |A| = A^{-1} \end{aligned}$$

Later we will encounter a lot of problems with least squares equations, let's say in this situation we will not be able to find a vector$x \in R^n$, such that $Ax = b$, so instead we want to find a vector x such that Ax is as

close as possible to b, as measured by the square of the Euclidean norm $||Ax - b||_2^2$, we can simply find that the best x we can find is

$$x = (A^T A)^{-1} A^T b$$

this is the same conclusion we shall see later.

At last, we ha to talk about *the Lagrangian* in matrixes. Consider the following, equality constrained optimization problem :

$$max_{x \in R^n} x^T Ax \qquad subject\ to\ ||x||_2^2 = 1$$

The Lagrangian in this case can be given by

$$L(x, \lambda) = x^T Ax - \lambda(x^T x - 1)$$

It's easy to find that the only points which can possibly maximize (or minimize) $x^T Ax$ assuming $x^T x = 1$ are *the eigenvectors* of A.

Probability Theory

In this section, we will be checking out a few names and definitions that we've already learned yet not familiar with its other forms.

First we give two properties.

$$P(A^c) = P(\Omega \setminus A) = 1 - P(A)$$
$$P(A \cup B) \leq P(A) + P(B)$$

By $P(A^c)$ we mean the complement of event A, the second formula is also known as *the union bound.*

Secondly we talk a little bit about *random variables*. **A random variable X is a function $X : \Omega \to R$.** Note that from a measure-theoretic perspective, random variables must be Borel-measurable functions. Intuitively, this restriction ensures that given a random variable and its underlying outcome space, one can implicitly define the each of the events of the event space as being sets of outcomes $\omega \in \Omega$ for which $X(\omega)$ satisfies some property (e.g., the event $\{\omega : X(\omega) \geq 3\}$).

Suppose that $X(\omega)$ is the number of heads which occur in the sequence of tosses ω. Given that only 10 coins are tossed, $X(\omega)$ can take only a finite number of values, so it is known as a *discrete random variable*. Here, the probability of the set associated with a random variable X taking on some specific value k is

$$P(X = k) := P(\{\omega : X(\omega) = k\})$$

> Note that here ω is actually an element from the Ω

And for sure we have *continuous random variables* which should be

$$P(a \leq X \leq b) := P(\{\omega : a \leq X(\omega) \leq b\})$$

In order to specify the probability measures used when dealing with random variables, it is often convenient to specify alternative functions (CDFs, PDFs, and PMFs) from which the probability measure governing an experiment immediately follows. In this section and the next two sections, we describe each of these types of functions in turn.

A *cumulative distribution function (CDF)* is a function $F_X : R \to [0, 1]$ which specifies a probability measure as,

$$F_X(x) = P(X \leq x)$$

By using this function one can calculate the probability of any event in $\mathcal{F}$, and a common figure for this function is below, this is obvious for $P(a < X \leq b) = F(b) - F(a)$

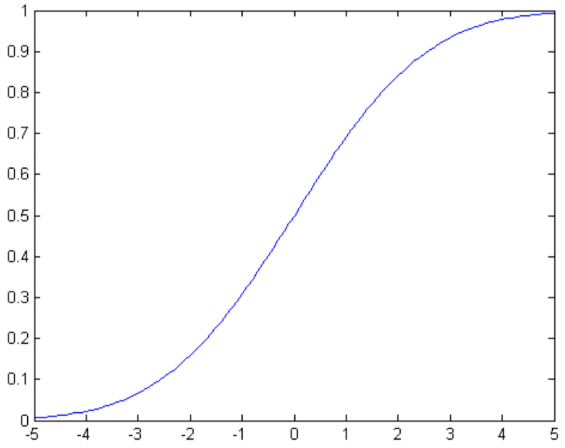


Figure 1: A cumulative distribution function (CDF).

CDF has a few simple properties we will just ignore anyway.

A *probability mass function (PMF)* is a function $p_X : \Omega \to R$ such that

$$p_X(x) = P(X = x).$$

This is way more simpler, in the case of discrete random variable, we use the notation *Val(X)* for the set of possible values that the random variable X may assume. For example, if $X(\omega)$ is a random variable indicating the number of heads out of ten tosses of coin, then $Val(X) = \{0, 1, 2, ..., 10\}$.

For some **continuous random variables**, the **cumulative distribution function $F_X(x)$ is differentiable everywhere**. In these cases, we define the *Probability Density Function (PDF)* as the derivative of the CDF, i.e.,

$$f_X(x) = \frac{dF_X(x)}{dx}$$

Of course we have the follow property for the property of differentiables

$$P(x \leq X \leq x + \Delta x) \approx f_X(x)\Delta x$$

A PDF function should always be positive, which also implies that CDF should be incremental.

And for now, we are diving into *expectation.* Suppose that X is a discrete random variable with PMF $p_X(x)$ and $g : R \to R$ is an arbitrary function. In this case, $g(X)$ can be considered a random variable, and we define the expectation or expected value of $g(X)$ as

$$E[g(X)] = \sum_{x \in \text{Val}(X)} g(x)p_X(x)$$

If X is a continuous random variable with PDF $f_X(x)$, then the expected value of $g(X)$ is defined as

$$E[g(X)] = \int_{-\infty}^{\infty} g(x)f_X(x)\,dx$$

As we all always did before, we talk about *Variance* after the expectations. The variance of a random variable X is a measure of how concentrated the distribution of a random variable X is around its mean. Formally, the variance of a random variable X is defined as

$$\text{Var}(X) = E[(X - E[X])^2]$$

which could also be written as

$$\text{Var}(X) = E[X^2] - (E[X])^2$$

Before we get something tough, let's go trough some common distribution of single variables.

- $X \sim Bernoulli(p)$ (where $0 \leq p \leq 1$): one if a coin with heads probability p comes up heads, zero otherwise.

$$p_X(x) = \begin{cases} p & \text{if } x = 1 \\ 1 - p & \text{if } x = 0 \end{cases}$$

- $X \sim Binomial(n, p)$ (where $0 \leq p \leq 1$): the number of heads in n independent flips of a coin with heads probability p.

$$p_X(x) = \binom{n}{x} p^x (1-p)^{n-x}$$

- $X \sim Geometric(p)$ (where $p > 0$): the number of flips of a coin with heads probability p until the first heads.

$$p(x) = p(1-p)^{x-1}$$

- $X \sim Poisson(\lambda)$ (where $\lambda > 0$): a probability distribution over the nonnegative integers used for modeling the frequency of rare events.

$$p_X(x) = \frac{e^{-\lambda}\lambda^x}{x!}$$

Above are all for random discrete variables, for continuous variables, we have something else.

- $X \sim Uniform(a, b)$ (where $a < b$): equal probability density to every value between a and $b$on the real line.

$$f_X(x) = \frac{1}{b-a}, \quad \text{if } a \leq x \leq b$$

- $X \sim Exponential(\lambda)$ (where $\lambda > 0$): decaying probability density over the nonnegative reals.

$$f_X(x) = \lambda e^{-\lambda x}, \quad \text{if } x \geq 0$$

- $X \sim Normal(\mu, \sigma^2)$: also known as the Gaussian distribution

$$f_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

The shape of their PDF functions and the properties are as followed

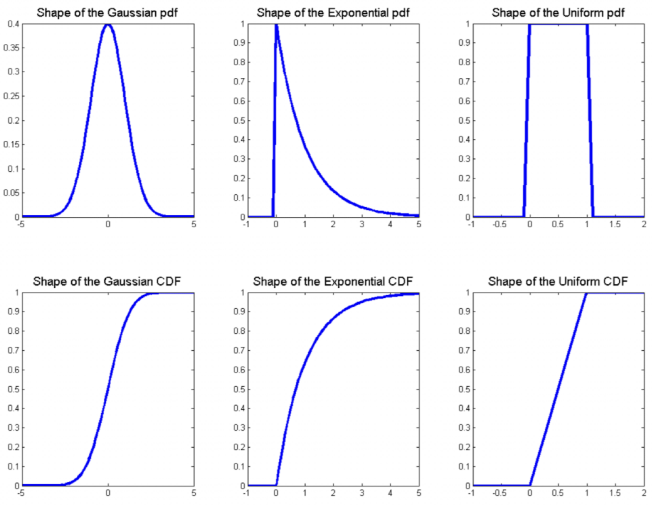


Figure 2: PDF and CDF of a couple of random variables.

Distribution	PDF or PMF	Mean	Variance
$Bernoulli(p)$	$\begin{cases} p, & \text{if } x = 1 \\ 1-p, & \text{if } x = 0. \end{cases}$	p	$p(1-p)$
$Binomial(n, p)$	$\binom{n}{k}\ p^k(1-p)^{n-k}$ for $0 \leq k \leq n$	np	npq
$Geometric(p)$	$p(1-p)^{k-1} \quad$ for $k = 1, 2, \ldots$	$\frac{1}{p}$	$\frac{1-p}{p^2}$
$Poisson(\lambda)$	$e^{-\lambda}\lambda^x/x! \quad$ for $k = 1, 2, \ldots$	λ	λ
$Uniform(a, b)$	$\frac{1}{b-a} \quad \forall x \in (a, b)$	$\frac{a+b}{2}$	$\frac{(b-a)^2}{12}$
$Gaussian(\mu, \sigma^2)$	$\frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{(x-\mu)^2}{2\sigma^2}}$	μ	σ^2
$Exponential(\lambda)$	$\lambda e^{-\lambda x} \quad x \geq 0, \lambda > 0$	$\frac{1}{\lambda}$	$\frac{1}{\lambda^2}$

And that is our complement for now, now let's get into something new.

Two Random Variables

First we will be talking about *joint and marginal distributions.* Suggesting that we have two separated variables X and Y, and we want to know about **a value assumed simultaneously** by X and Y during an experiment, we need something more than just to consider them separately. Thus, we define *joint cumulative distribution function* of X and Y

$$F_{XY}(x, y) = P(X \leq x, Y \leq y)$$

and we have the relation between joint CDF and joint distribution function described as

$$F_X(x) = \lim_{y \to \infty} F_{XY}(x, y)$$
$$F_Y(y) = \lim_{x \to \infty} F_{XY}(x, y)$$

Here, we call $F_X(x)$ and $F_Y(y)$ the *marginal cumulative distribution functions* of $F_{XY}(x, y)$.

Then we talk about *Joint and marginal probability mass functions.*

If X and Y are discrete random variables, then the joint probability mass function $p_{XY} : R \times R \to [0, 1]$ is defined by

$$p_{XY}(x, y) = P(X = x, Y = y)$$

How does the joint PMF over two variables relate to the probability mass function for each variable separately? It turns out that

$$p_X(x) = \sum_y p_{XY}(x, y)$$

In this case, we refer to $p_X(x)$ as the *marginal probability mass function* of X, and this process of forming the marginal distribution with respect to one variable by summing out the other variable is often known as "marginalization."

Thirdly, we talk about *Joint and marginal probability density functions.*

Let X and Y be two continuous random variables, then their joint CDF can be differentiable, thus we can define the *joint probability density function*

$$f_{XY}(x,y) = \frac{\partial^2 F_{XY}(x,y)}{\partial x \partial y}$$

Like in the single-dimensional case, $f_{XY}(x,y) \neq P(X=x, Y=y)$, but rather

$$\iint f_{XY}(x,y)\,dx\,dy = P((X,Y) \in A)$$

Analagous to the discrete case, we define

$$f_X(x) = \int_{-\infty}^{+\infty} f_{XY}(x,y)\,dy$$

as *marginal probability density function (or marginal density) of X.*

After all these, it's time complete everything for two variables.

Starting with *Conditional distributions,* in the discrete case, the conditional probability mass function of Y given X is simply

$$p_{Y|X}(y|x) = \frac{p_{XY}(x,y)}{p_X(x)}$$

by analogy to the discrete case, the conditional probability density of Y given $X = x$ to be

$$f_{Y|X}(y|x) = \frac{f_{XY}(x,y)}{f_X(x)}$$

An important relationship of conditional distribution and marginal distribution is the *Law of total expectation*. This result can be viewed as an extension of the *law of total probability* disucussed in Section 1.

$$E[X] = E[E[X|Y]]$$

Then the *Bayes' rule for random variables,* for discrete variables, we have

$$P_{Y|X}(y|x) = \frac{P_{XY}(x,y)}{P_X(x)} = \frac{P_{X|Y}(x|y)P_Y(y)}{\sum_{y' \in \mathrm{Val}(Y)} P_{X|Y}(x|y')P_Y(y')}$$

and for continuous variables we have

$$f_{Y|X}(y|x) = \frac{f_{XY}(x,y)}{f_X(x)} = \frac{f_{X|Y}(x|y)f_Y(y)}{\int_{-\infty}^{\infty} f_{X|Y}(x|y')f_Y(y')\,dy'}$$

Last, we discuss about the *expectation* and *covariance* of two variables, the expected value of g is defined in the following way,

$$E[g(X,Y)] = \sum_{x \in \text{Val}(X)} \sum_{y \in \text{Val}(Y)} g(x,y)\, p_{XY}(x,y)$$

$$E[g(X,Y)] = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} g(x,y)\, f_{XY}(x,y)\, dx\, dy$$

and the covariance are

$$\text{Cov}[X,Y] = E[(X - E[X])(Y - E[Y])] = E[XY] - E[X]E[Y]$$

When $Cov[X,Y] = 0$, we say that X and Y are *uncorrelated*, but this is not the same thing as stating that X and Y are independent! For example, if $X \sim Uniform(-1,1)$ and $Y = X^2$, then one can show that X and Y are uncorrelated, even though they are not independent.

We note the following properties of expectation and covariance.

- If X and Y are independent, then $Cov[X,Y] = 0$
- $Var[X+Y] = Var[X] + Var[Y] + 2Cov[X,Y]$

Multiple Random Variables

To simplify this section, we discuss continuous variables only. First we give the definition of joint CDF, joint PDF, marginal PDF and conditional PDF

$$F_{X_1,X_2,\ldots,X_n}(x_1,x_2,\ldots,x_n) = P(X_1 \le x_1, X_2 \le x_2, \ldots, X_n \le x_n)$$

$$f_{X_1,X_2,\ldots,X_n}(x_1,x_2,\ldots,x_n) = \frac{\partial^n F_{X_1,X_2,\ldots,X_n}(x_1,x_2,\ldots,x_n)}{\partial x_1 \partial x_2 \ldots \partial x_n}$$

$$f_{X_1}(x_1) = \int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} f_{X_1,X_2,\ldots,X_n}(x_1,x_2,\ldots,x_n)\, dx_2 \cdots dx_n$$

$$f_{X_1|X_2,\ldots,X_n}(x_1|x_2,\ldots,x_n) = \frac{f_{X_1,X_2,\ldots,X_n}(x_1,x_2,\ldots,x_n)}{f_{X_2,\ldots,X_n}(x_2,\ldots,x_n)}$$

To calculate the probability of an event $A \subseteq R^n$ we have,

$$P((x_1,x_2,...x_n) \in A) = \int_{(x_1,x_2,...x_n) \in A} f_{X_1,X_2,...,X_n}(x_1,x_2,...x_n) dx_1 dx_2 ... dx_n$$

And through this, we can establish what is called *the Chain rule* in probability literature.

$$
\begin{aligned}
f(x_1, x_2, \ldots, x_n) &= f(x_n|x_1, x_2, \ldots, x_{n-1}) f(x_1, x_2, \ldots, x_{n-1}) \\
&= f(x_n|x_1, x_2, \ldots, x_{n-1}) f(x_{n-1}|x_1, x_2, \ldots, x_{n-2}) f(x_1, x_2, \ldots, x_{n-2}) \\
&= \ldots \\
&= f(x_1) \prod_{i=2}^{n} f(x_i|x_1, \ldots, x_{i-1})
\end{aligned}
$$

Then we consider random vectors and their expectations and covariances.

Consider denote the given n random variables as a vector $X = [X_1 X_2 ... X_n]^T$. We call the resulting vector a random vector (more formally, a random vector is a mapping from $\Omega \rightarrow R^n$).

After that, we can now define the expectation of X as

$$
E[g(X)] = \int_{R^n} g(x_1 x_2 ... x_n) f_{X_1 X_2 ... X_N}(x_1 x_2 ... x_n) dx_1 dx_2 ... dx_n
$$

where $g(\ )$ is an arbitrary function mapping from $R^n \rightarrow R$, $\int_{R^n}$ is n consecutive integrations from $-\infty \rightarrow \infty$.

If the $g(\ )$ is an arbitrary function mapping from $R^n \rightarrow R^m$, $g(x) = \begin{bmatrix} g_1(x) \\ g_2(x) \\ \vdots \\ g_m(x) \end{bmatrix}$, then we have

$$
\mathbb{E}[g(X)] = \begin{bmatrix} \mathbb{E}[g_1(X)] \\ \mathbb{E}[g_2(X)] \\ \vdots \\ \mathbb{E}[g_m(X)] \end{bmatrix}
$$

where $g_i(\ )$ is a function mapping from $R^n \rightarrow R$.

Finally, all we have to do is to define covariance, which naturally should be

$$
\begin{aligned}
\sum &= \begin{bmatrix} \mathrm{Cov}[X_1, X_1] & \cdots & \mathrm{Cov}[X_1, X_n] \\ \vdots & \ddots & \vdots \\ \mathrm{Cov}[X_n, X_1] & \cdots & \mathrm{Cov}[X_n, X_n] \end{bmatrix} \\
&= \begin{bmatrix} \mathbb{E}[X_1^2] - \mathbb{E}[X_1]\mathbb{E}[X_1] & \cdots & \mathbb{E}[X_1 X_n] - \mathbb{E}[X_1]\mathbb{E}[X_n] \\ \vdots & \ddots & \vdots \\ \mathbb{E}[X_n X_1] - \mathbb{E}[X_n]\mathbb{E}[X_1] & \cdots & \mathbb{E}[X_n^2] - \mathbb{E}[X_n]\mathbb{E}[X_n] \end{bmatrix} \\
&= \begin{bmatrix} \mathbb{E}[X_1^2] & \cdots & \mathbb{E}[X_1 X_n] \\ \vdots & \ddots & \vdots \\ \mathbb{E}[X_n X_1] & \cdots & \mathbb{E}[X_n^2] \end{bmatrix} - \begin{bmatrix} \mathbb{E}[X_1]\mathbb{E}[X_1] & \cdots & \mathbb{E}[X_1]\mathbb{E}[X_n] \\ \vdots & \ddots & \vdots \\ \mathbb{E}[X_n]\mathbb{E}[X_1] & \cdots & \mathbb{E}[X_n]\mathbb{E}[X_n] \end{bmatrix} \\
&= \mathbb{E}[X X^T] - \mathbb{E}[X]\mathbb{E}[X]^T = \mathbb{E}[(X - \mathbb{E}[X])(X - \mathbb{E}[X])^T]
\end{aligned}
$$

We can prove that the covariance matrix of any random vector must always be symmetric positive semidefinite.

In the last part of this section, let's talk about the *Law of large numbers* and the *Central limit theorem.*

As it's known to all, we have

$$\mathbb{P}\left(\lim_{n\to\infty}\frac{1}{n}\sum_{i=1}^{n}X_i = E[X_1]\right) = 1$$

The real question here is how fast the empirical average converges to its expected value, and the Central limit theorem give a solution through a refinement of the law of the large numbers.

Let $X_1, X_2, \ldots$ be a series of iid random variables, with mean μ and variance σ^2. Then the normalized partial sum

$$\xi_n = \frac{1}{\sqrt{n}}\sum_{i=1}^{n}(\frac{X_i-\mu}{\sigma})$$

satisfy that

$$\lim_{n\to\infty} P(\xi_n \le x) = \Phi(x)$$

for any x, where Φ is CDF of the standard normal distribution.

The convergence stated in (15) is also known as *convergence in distribution* in probabilistic literature. Theorem 4.4 shows that irrespective of what distribution X follows, its normalized partial sum (or average) is always a normal distribution! So when the number of samples are large, we can approximate any distribution using a normal distribtution.

Note that both LLN(Law of Large Numbers) and CLT(Central Limit Theorem) can be extended to multi-dimensional random vectors, and generalized to weaker assumptions. The proofs of both theorems are out of the scope of this class.

The Multivariate Gaussian Distribution

One particularly important example of a probability distribution over random vectors X is called the *multivariate Gaussian or multivariate normal distribution*. A random vector $X \in R^d$ is said to have a multivariate normal (or Gaussian) distribution with mean $\mu \in R^d$ and covariance matrix $\Sigma \in S_{d++}$ (where S_{d++} refers to the space of symmetric positive definite $d \times d$ matrices)

Why is Gaussian do important

Generally speaking, Gaussian random variables are extremely useful in machine learning and statistics for two main reasons. First, they are extremely common when modeling “noise” in statistical algorithms. Quite often, noise can be considered to be the accumulation of a large number of small independent random perturbations affecting the measurement process; by the Central Limit Theorem, summations of independent random variables will tend to “look Gaussian.” Second, Gaussian random variables are convenient for many analytical manipulations, because many of the integrals involving Gaussian distributions that arise in practice have simple closed form solutions. We will encounter this later in the course.

First we draw a look at its relation with the univariate Gaussian, so we may later have a better understanding. We all know an univariate Gaussian is given by

$$f(x;\mu,\sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} exp(-\frac{1}{2\sigma^2}(x-\mu)^2)$$

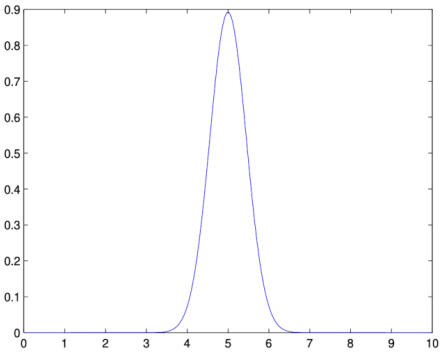
where $-\frac{1}{2\sigma^2}(x-\mu)^2$ is a quadratic function, and $\frac{1}{\sqrt{2\pi}\sigma}$ is what we view as “normalisation factor”.

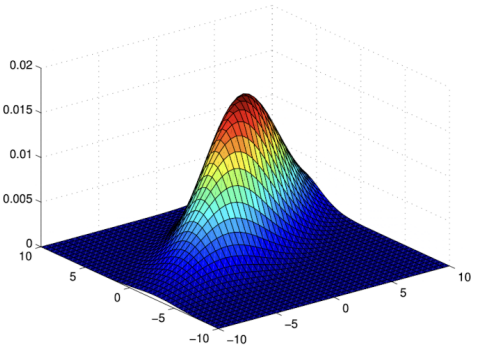
We can thus give the form of the multivariate Gaussian

$$\frac{1}{2\pi^{\frac{d}{2}}|\Sigma|^{\frac{1}{2}}} exp(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu))$$

which, as before, follows that

$$\frac{1}{2\pi^{\frac{d}{2}}|\Sigma|^{\frac{1}{2}}} \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\cdots\int_{-\infty}^{\infty} \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right) dx_1\, dx_2 \cdots dx_d = 1$$





And second lets us discuss a little about the covariance matrix. In the last section, we have

$$\Sigma = E[(X-\mu)(X-\mu)^T] = E[XX^T] - \mu\mu^T$$

In the definition of multivariate Gaussians, we required that the covariance matrix Σ be symmetric positive definite (i.e., $\Sigma \in S_{d++}$). Why does this restriction exist? First, Σ must be symmetric positive semidefinite in order for it to be a valid covariance matrix. However, in order for Σ^{-1} to exist (as required in the definition of the multivariate Gaussian density), then Σ must be invertible and hence full rank. Since any full rank symmetric positive semidefinite matrix is necessarily symmetric positive definite, it follows that Σ must be symmetric positive definite.

Now let's talk about a special kind of multivariate Gaussian which is lead by two independent variate and where the covariance matrix Σ is diagonal.

So we have

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad \mu = \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \quad \Sigma = \begin{bmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{bmatrix}$$

In this case, we can rewrite the Gaussian as

$$f(x; \mu, \Sigma) = \frac{1}{2\pi(\sigma_1^2 \cdot \sigma_2^2 - 0 \cdot 0)^{1/2}} \exp\left(-\frac{1}{2} \begin{bmatrix} x_1 - \mu_1 \\ x_2 - \mu_2 \end{bmatrix}^T \begin{bmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{bmatrix}^{-1} \begin{bmatrix} x_1 - \mu_1 \\ x_2 - \mu_2 \end{bmatrix}\right)$$

which is actually

$$= \frac{1}{\sqrt{2\pi}\sigma_1} \exp\left(-\frac{1}{2\sigma_1^2}(x_1 - \mu_1)^2\right) \cdot \frac{1}{\sqrt{2\pi}\sigma_2} \exp\left(-\frac{1}{2\sigma_2^2}(x_2 - \mu_2)^2\right)$$

it's the product of two independent univariate Gaussian. This could be generalised for real multivariate situations.

And now let's talk about isocontours, which, as we all already learned before. It's a way to understand a multivariate Gaussian conceptually is to understand the shape of its isocontours. For a function $f : R^2 \to R$, an isocontour is a set of the form.

$$\{x \in R \ : \ f(x) = c\}$$

Of course we wonder what does the isocontours of a Gaussian looks like, and take a simple example which only involves two variates (which we just discussed as an example)

If we have

$$c = \frac{1}{\sqrt{2\pi}\sigma_1} \exp\left(-\frac{1}{2\sigma_1^2}(x_1 - \mu_1)^2\right) \cdot \frac{1}{\sqrt{2\pi}\sigma_2} \exp\left(-\frac{1}{2\sigma_2^2}(x_2 - \mu_2)^2\right)$$

we can easily work out that

$$1 = \frac{(x_1 - \mu_1)^2}{2\sigma_1^2 log(\frac{1}{2\pi c \sigma_1 \sigma_2})} + \frac{(x_2 - \mu_2)^2}{2\sigma_2^2 log(\frac{1}{2\pi c \sigma_1 \sigma_2})}$$

which is obviously an ellipse

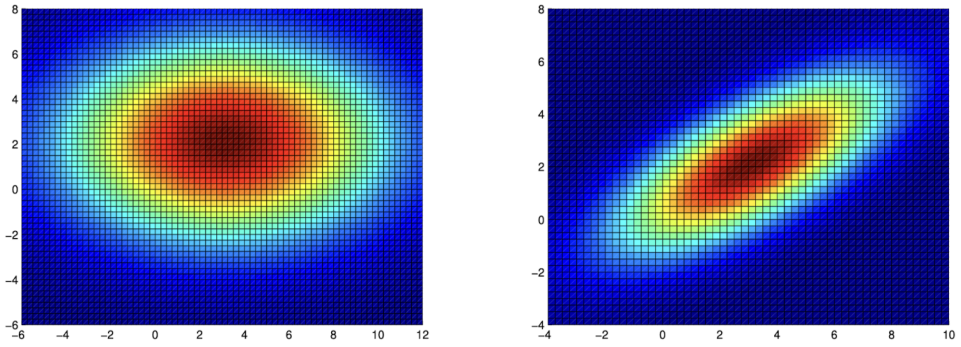


Figure 4:
The figure on the left shows a heatmap indicating values of the density function for an axis-aligned multivariate Gaussian with mean $\mu = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$ and diagonal covariance matrix $\Sigma = \begin{bmatrix} 25 & 0 \\ 0 & 9 \end{bmatrix}$. Notice that the Gaussian is centered at $(3, 2)$, and that the isocontours are all elliptically shaped with major/minor axis lengths in a 5:3 ratio. The figure on the right shows a heatmap indicating values of the density function for a non axis-aligned multivariate Gaussian with mean $\mu = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$ and covariance matrix $\Sigma = \begin{bmatrix} 10 & 5 \\ 5 & 5 \end{bmatrix}$. Here, the ellipses are again centered at $(3, 2)$, but now the major and minor axes have been rotated via a linear transformation.

Thus we could say that the μ decides the center while the Σ decides the transformation(its rotation and shape). If we define

$$r_1 = \sqrt{2\sigma_1^2 log(\frac{1}{2\pi c \sigma_1 \sigma_2})} \quad r_2 = \sqrt{2\sigma_2^2 log(\frac{1}{2\pi c \sigma_1 \sigma_2})}$$

then the former equation could be

$$1 = (\frac{x_1 - \mu_1}{r_1})^2 + (\frac{x_2 - \mu_2}{r_2})^2$$

Equation (20) should be familiar to you from high school analytic geometry: it is the equation of an *axis-aligned ellipse*, with center (μ_1, μ_2), where the x1 axis has length $2r_1$ and the x_2 axis has length $2r_2$!

We can use specially give values to draw a conclusion on how the mean and the covariance effects the level curve. The axis length r_i required to reach a fraction $1/e$ of the peak height of the Gaussian density in the i-th dimension is proportional to the standard deviation σ_i. In other words, if we move a distance $r_i = k\sigma_i$ from the mean in each dimension, we will find the point where the probability density drops to 1/e of its peak value. Here, k is a constant.

Intuitively, that is the smaller the variance of some random variable xi, the more “tightly” peaked the Gaussian distribution in that dimension, and hence the smaller the radius r_i.

> Clearly, the above derivations rely on the assumption that Σ is a diagonal matrix. However, in the non-diagonal case, it turns out that the picture is not all that different. Instead of being an axis-aligned ellipse, the isocontours turn out to be simply rotated ellipses. Furthermore, in the d-dimensional case, the level sets form geometrical structures known as *ellipsoids* in R^d.

In the last few sections, we all focused on Gaussians with a diagonal covariance. and we want to generalize our theory for all kind of Gaussians, where the covariance is not diagonal. What would that be like. This theorem describes what we want

Theorem 5.2.: Let $X \sim N(\mu, \Sigma)$ for some $\mu \in R^d$ and $\Sigma \in S_{d++}$. Then, there exists a matrix $B \in R^{d \times d}$such that if we define $Z = B^{-1}(X - \mu)$, then $Z \sim N(0, I)$.

> Note here that Z can be thought of as a collection of d independent standard normal random variables.

The theorem states that any random variable X with a multivariate Gaussian distribution can be interpreted as the result of applying a *linear transformation* ($X = BZ + \mu$) to some collection of d independent standard normal random variables (Z).

Last, we discuss about the closure properties of the Gaussian.

1. The sum of independent Gaussian random variables is Gaussian.
2. The marginal of a joint Gaussian distribution is Gaussian.
3. The conditional of a joint Gaussian distribution is Gaussian.

The formal statement of the first rule is:

> Theorem 5.3: Suppose that $y \sim N(\mu, \Sigma)$ and $z \sim N(\mu', \Sigma')$ are independent Gaussian distributed random variables,
> where $\mu, \mu' \in \mathbb{R}^d$ and $\Sigma, \Sigma' \in S_d^{++}$. Then, their sum is also Gaussian:
> $y + z \sim N(\mu + \mu', \Sigma + \Sigma')$

Note that the image of this Gaussian is not just to simply add up two bumps of the Gaussian.

The formal statement of the second rule is:

Suppose that

$$\begin{bmatrix} x_A \\ x_B \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu_A \\ \mu_B \end{bmatrix}, \begin{bmatrix} \Sigma_{AA} & \Sigma_{AB} \\ \Sigma_{BA} & \Sigma_{BB} \end{bmatrix} \right),$$

where $x_A \in \mathbb{R}^n, x_B \in \mathbb{R}^d$, and the dimensions of the mean vectors and covariance matrix subblocks are chosen to match x_A and x_B.

Then, the marginal densities:

$$p(x_A) = \int_{x_B \in \mathbb{R}^d} p(x_A, x_B; \mu, \Sigma)\, dx_B$$
$$p(x_B) = \int_{x_A \in \mathbb{R}^n} p(x_A, x_B; \mu, \Sigma)\, dx_A$$

are Gaussian

$$x_A \sim \mathcal{N}(\mu_A, \Sigma_{AA})$$
$$x_B \sim \mathcal{N}(\mu_B, \Sigma_{BB})$$

Our strategy to prove this is below

1. Write the integral form of the marginal density explicitly.
2. Rewrite the integral by partitioning the inverse covariance matrix.
3. Use a “completion-of-squares” argument to evaluate the integral over x_B.
4. Argue that the resulting density is Gaussian.

The formal statement of the thhird rule is:

Suppose that

$$\begin{bmatrix} x_A \\ x_B \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu_A \\ \mu_B \end{bmatrix}, \begin{bmatrix} \Sigma_{AA} & \Sigma_{AB} \\ \Sigma_{BA} & \Sigma_{BB} \end{bmatrix} \right),$$

where $x_A \in \mathbb{R}^n, x_B \in \mathbb{R}^d$, and the dimensions of the mean vectors and covariance matrix subblocks are chosen to match x_A and x_B.

Then, the conditional densities:

$$p(x_A \mid x_B) = \frac{p(x_A, x_B; \mu, \Sigma)}{\int_{x_A \in \mathbb{R}^n} p(x_A, x_B; \mu, \Sigma)\, dx_A}$$
$$p(x_B \mid x_A) = \frac{p(x_A, x_B; \mu, \Sigma)}{\int_{x_B \in \mathbb{R}^d} p(x_A, x_B; \mu, \Sigma)\, dx_B}$$

are also Gaussian:

$$x_A \mid x_B \sim \mathcal{N}\left(\mu_A + \Sigma_{AB}\Sigma_{BB}^{-1}(x_B - \mu_B), \Sigma_{AA} - \Sigma_{AB}\Sigma_{BB}^{-1}\Sigma_{BA}\right)$$
$$x_B \mid x_A \sim \mathcal{N}\left(\mu_B + \Sigma_{BA}\Sigma_{AA}^{-1}(x_A - \mu_A), \Sigma_{BB} - \Sigma_{BA}\Sigma_{AA}^{-1}\Sigma_{AB}\right)$$

Supervised Learning

Linear regression

Intro

In most situations, we encounter a set of data which were related, and for sure we want find the function between them and thus to make predictions. That where we need machine learning.

Right before we start, let’s take a look at some useful terms we need. To make everything sound more grounded, let suggest a real life situation, say your gpa and your pay outs. We have a data set like this

GPA	Hours in library	Hours in dorms
4.5	46	30
4.1	20	60
4.3	36	40

So we have our inputs in the shape of $x^T = [hours\ in\ lib,\ hours\ in\ dorms]$, which we denote as $x^{(i)T} = [x_1^{(i)},\ x_2^{(i)}]$. To perform supervised learning, we must decide how we're going to represent *functions/hypotheses* h in a computer. As an initial choice, let's say we decide to approximate y as a linear function of x:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2$$

Here, the θ_i's are the *parameters (also called weights)* parameterizing the space of linear functions mapping from $\mathcal{X}\ to\ \mathcal{Y}$. Later we shall drop the subscript theta in $h(x)$. To simplify our notation, we also introduce the convention of letting $x_0 = 1$ (this is the *intercept term*), so that we have

$$h(x) = \sum_{i=0}^{d} \theta_i x_i = \theta^T x$$

And intuitively, if we want to find the best theta for our model, we need to find the theta that makes each $h(x)$ as close to the y as possible, so we define our *cost function*

$$J(\theta) = \frac{1}{2} \sum_{i=1}^{n} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

If you've seen linear regression before, you may recognize this as the familiar *least-squares cost function* that gives rise to the *ordinary least squares regression model*. Anyway what we have to do is to find a way to minimize this cost function, and as you may find later, there's more than one way to do this.

LMS algorithm

LMS stands for *least mean squares,* which is a basic rule for updating our theta for the future. Intuitively, we firstly would come up with *gradient decent algorithm*, for it can't be any more obvious.

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

Here, alpha is what we call *learning rate*, which actually decides how far we move on the curve of our cost function. To further complete our formula, we need to further work out the partial derivative on the right. It's easy to find the formula could be simply denoted as followed

$$\theta_j := \theta_j + \alpha(y^{(i)} - h_\theta(x^{(i)}))x_j^{(i)}$$

And this is what we called *LMS update rule*, and is also known as the *Widrow-Hoff learning rule*. This rule has something pretty natural and intuitive, as the error defines how much change we apply to our theta.

We'd derived the LMS rule for when there was only a single training example. There are two ways to modify this method for a training set of more than one example. The first way is here

{

$$\theta_j := \theta_j + \alpha \sum_{i=1}^{n} (y^{(i)} - h\theta(x^{(i)}))x_j^{(i)} \ , \ (for\ every\ j) \qquad (1.1)$$

{

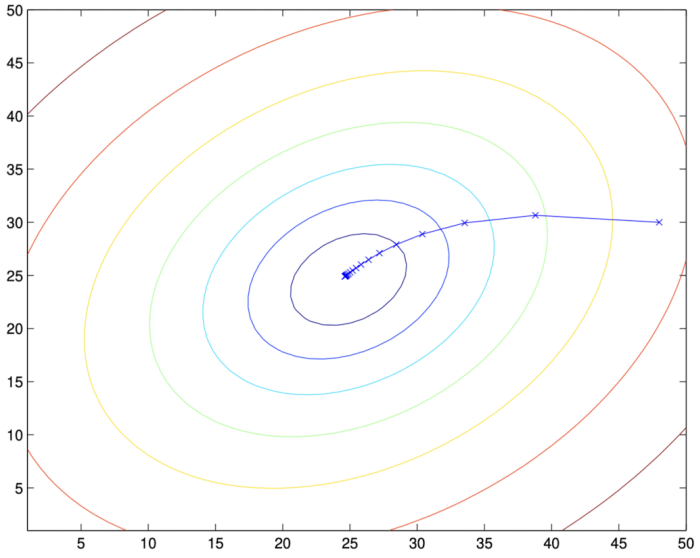
Repeat this until the theta converges. By grouping the updates of the coordinates into an update of the vector θ, we can rewrite update (1.1) in a slightly more succinct way:

$$\theta := \theta + \alpha \sum_{i=1}^{n} (y^{(i)} - h_\theta(x^{(i)}))x^{(i)}$$

We can easily verify that the quantity in the summation in the update rule above is just $\frac{\partial J(\theta)}{\partial \theta_j}$ (for the original definition of J). So, this is simply gradient descent on the original cost function J. This method looks at every example in the entire training set on every step, and is called *batch gradient descent*.

> *Note that, while gradient descent can be susceptible to local minima in general, the optimization problem we have posed here for linear regression has only one global, and no other local, optima; thus gradient descent always converges (assuming the learning rate α is not too large) to the global minimum. Indeed, J is a convex quadratic function. Here is an example of gradient descent as it is run to minimize a quadratic function.*

Here we give a graph to show how the gradient decent algorithm works



That is $ax^2 + by^2 + cxy + dx + ey + f = 0$
take $x = \begin{pmatrix} x \\ y \end{pmatrix}$, we have $x^T Ax + B^T x + C = 0$
When A is positive definite, this is an ellipse

The ellipses shown above are the contours of a quadratic function. Also shown is the trajectory taken by gradient descent.

Here is an alternative to batch gradient descent that also works very well. Consider the following algorithm:

$$\text{for } i = 1 \text{ to } n, \{$$
$$\theta_j := \theta_j + \alpha \left(y^{(i)} - h_\theta(x^{(i)}) \right) x_j^{(i)}, \quad \text{(for every } j\text{)}$$
$$\}$$

Like before, this could be denoted with using a vector to represent all

$$\theta := \theta + \alpha(y^{(i)} - h_\theta(x^{(i)}))x^{(i)}$$

In this algorithm, we repeatedly run through the training set, and each time we encounter a training example, we update the parameters according to the gradient of the error with respect to that single training example only. This algorithm is called *stochastic gradient descent* (also *incremental gradient descent*).

> Often, stochastic gradient descent gets θ “close” to the minimum much faster than batch gradient descent.

For these reasons, particularly when the training set is large, stochastic gradient descent is often preferred over batch gradient descent.

The normal equations

Gradient descent gives one way of minimizing J, and it's time to check out the another way, where we explicitly take the derivative of the J with respect to the θ_j's, and set it to zero. As we are already armed with the weapon of matrix derivatives, we can do this instantly.

Given a training set, define the *design matrix* X to be the n-by-d matrix (actually n-by-$d + 1$, if we include the intercept term) that contains the training examples' input values in its rows

$$\mathbf{X} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & x_2^{(m)} & \cdots & x_n^{(m)} \end{bmatrix} = \begin{bmatrix} - & x^{(1)} & - \\ - & x^{(2)} & - \\ & \vdots & \\ - & x^{(m)} & - \end{bmatrix}$$

Also, let ⃗y be the n-dimensional vector containing all the target values from the training set, thus we use the fact that for a vector z, we have that $z^T z = \sum z_i^2$, we can get

$$\frac{1}{2}(\mathbf{X}\boldsymbol{\theta} - \mathbf{y})^T(\mathbf{X}\boldsymbol{\theta} - \mathbf{y}) = \frac{1}{2}\sum_{i=1}^{n}(h_\theta(x^{(i)}) - y^{(i)})^2 = J(\theta)$$

By using what we've already learned, we can set its derivatives to zero, and obtain the normal equations:

$$X^T X\theta = X^T\vec{y}$$

Thus, the value of θ that minimizes $J(\theta)$ is given in closed form by the equation

$$\theta = (X^T X)^{-1} X^T\vec{y}$$

Probabilistic interpretation

In this section we try to give an explanation with a probabilistic point of view, we will give a set of probabilistic assumptions, under which least-squares regression is derived as a very natural algorithm.

First let's assume that the target variables and the inputs are related via the equation

$$y^{(i)} = \theta^T x^{(i)} + \epsilon^{(i)}$$

in which ϵ is a term which catches all unmodeled effects or random noise. And our first assumption to make is that the $\epsilon^{(i)}$ are distributed *IID (independently and identically distributed)* according to a *Gaussian distribution (also called a Normal distribution)* with mean zero and some *variance* σ^2. As we learned in high school, we write this as $\epsilon^{(i)} \sim N(0, \sigma^2)$, the density of ϵ is given by the following formula

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

which implies that

$$p(y^{(i)} \mid x^{(i)}; \boldsymbol{\theta}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y^{(i)} - \boldsymbol{\theta}^T x^{(i)})^2}{2\sigma^2}\right)$$

So when a *design matrix* X is given, the probability of the data is given by $p(\vec{y} \mid X; \theta)$, this quantity is typically viewed a function of $\vec{y}$(and perhaps X), for a fixed value of θ. When we wish to explicitly view this as a function of θ, we will instead call it the *likelihood function*:

$$L(\theta) = L(\theta; X, \vec{y}) = p(\vec{y}|X; \theta)$$

Note that by the independence assumption on the $\epsilon^{(i)}$'s (and hence also the $y^{(i)}$'s given the $x^{(i)}$'s), this can also be written as

$$L(\boldsymbol{\theta}) = \prod_{i=1}^{n} p(y^{(i)} \mid x^{(i)}; \boldsymbol{\theta}) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y^{(i)} - \boldsymbol{\theta}^T x^{(i)})^2}{2\sigma^2}\right)$$

Thus if we want to make our model as accurate as possible, what we really need is to find the θ that maximize the $L(\theta)$ (this is also called *the principal of maximum likelihood*)

After working out the function, we shall find something corresponds to our previous conclusions, what we are really trying to do is minimizing our *cost function.*

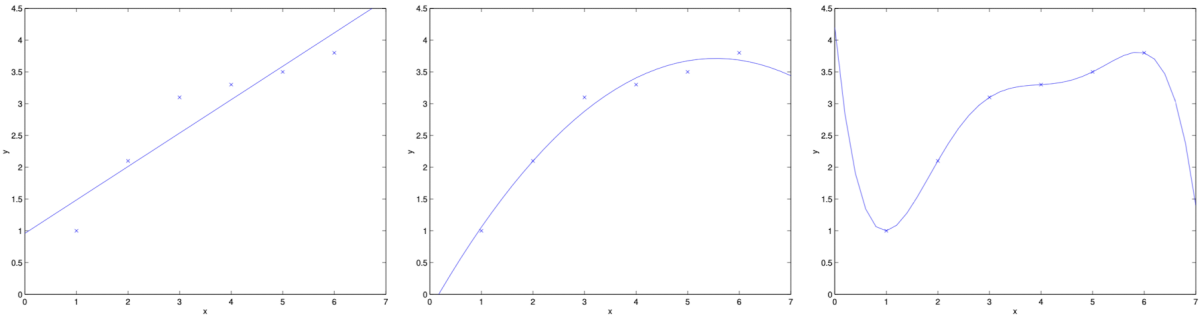
Note also that, in our previous discussion, our final choice of θ did not depend on what was σ^2, and indeed we'd have arrived at the same result even if σ^2 were unknown. We will use this fact again later, when we talk about the *exponential family* and *generalized linear models*.

Locally weighted linear regression

Firstly, to summarise this in a short, it's basically keep updating parameters(our θ) in accordance to nearby data. Before we cut into the way, we first take look at these three

different situations.

Consider the problem of predicting y from x∈R. The leftmost figure below shows the result of fitting a $y = \theta_0 + \theta_1 x$to a dataset. We see that the data doesn't really lie on straight line, and so the fit is not very good.



Instead, if we had added an extra feature x^2, and fit $y = \theta_0 + \theta_1 x + \theta_2 x^2$, then we obtain a slightly better fit to the data. And naively we would keep doing this till it fits perfectly (like in the rightmost figure!)

Without formally defining what these terms mean, we'll say the figure on the left shows an instance of *underfitting*—in which the data clearly shows structure not captured by the model —and the figure on the right is an example of *overfitting*.

As discussed previously, and as shown in the example above, the **choice of features** is important to ensuring good performance of a learning algorithm. In this section, let us briefly talk about the *locally weighted linear regression (LWR) algorithm* which, assuming there is sufficient training data, makes the choice of features less critical.

To make it clearer, we just make contrast with normal LMS ways

- In the original linear regression algorithm, to make a prediction at a query point x we would:
  - Fit θ to minimize $\sum_i (y^{(i)} - \theta^T x^{(i)})^2$
  - Output $\theta^T x$
- While in LWR algorithm, what we do is
  - Fit θ to minimize $\sum_i w^{(i)} (y^{(i)} - \theta^T x^{(i)})^2$
  - Output $\theta^T x$

Here, the $w^{(i)}$'s are non-negative valued weights. A fairly standard choice for the weights is

$$w^{(i)} = \exp\left(-\frac{(x^{(i)} - x)^2}{2\tau^2}\right)$$

Obviously θ is chosen giving a much higher "weight" to the (errors on) training examples close to the query point x, this would make our parameters be more accurate when reflecting the relation locally. The parameter τ controls how quickly the weight of a training example falls off with distance of its $x^{(i)}$ from the query point x; τ is called the *bandwidth parameter*.

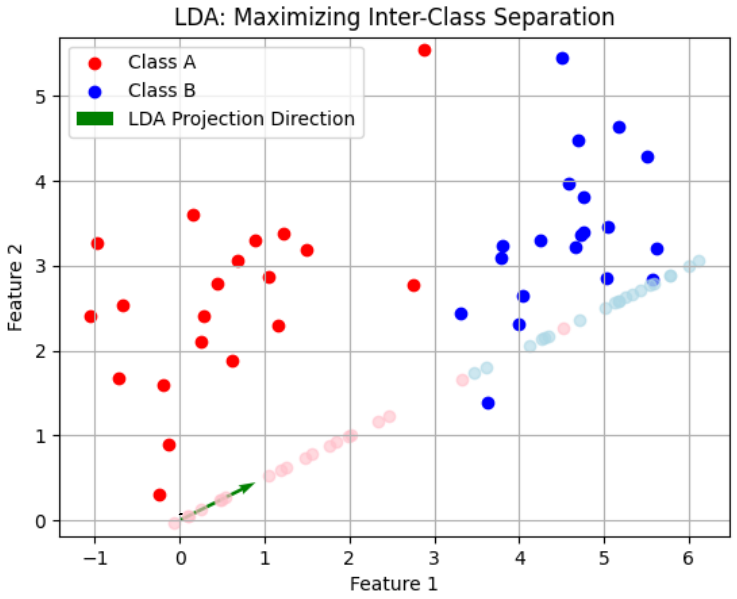
Locally weighted linear regression is the first example we're seeing of a *non-parametric algorithm*. The (unweighted) linear regression algorithm that we saw earlier is known as a *parametric learning algorithm*, because it has a fixed, finite number of parameters (the θ_i's), which are fit to the data. Once we've fit the θ_i's and stored them away, we no longer need to keep the training data around to make future predictions. In contrast, to make predictions using locally weighted linear regression, we need to keep the entire training set around.

> In summary, "non-parametric" models are characterized by their ability to grow in complexity with the size of the training data, making them highly flexible and capable of capturing intricate patterns without being confined to a fixed number of parameters.

LDA*

In this subsection, we introduce linear discriminate analysis, which is a not so common a technique that we hardly see it.

The basic idea of LDA is like PCA which we'll be talking about later in the future sections. In LDA, we try to do classification by projecting or example onto some line, as we can easily find that the points within a same group will be as close as possible and the distance of the two different groups would be as far as possible.



Consider two class denoted as $X_1$and X_2 and their mean being μ_1, μ_2 and covariance being Σ_1, Σ_2. If we set our hypothetical line as $y = w^T x$, thus we know that the center of two class's projection on the line would be $w^T \mu_1$ and $w^T \mu_2$, and the covariance of the projections are $w^T \Sigma_1 w$ and $w^T \Sigma_2 w$.

As we mentioned, we try to make two class as distant as possible and the samples in the class as dense as possible, thus we are actually doing an optimal problem which is

$$
\begin{aligned}
&arg\max_w \ ||w^T \mu_1 - w^T \mu_2||_2^2 \\
&arg\min_w \ w^T \Sigma_1 w + w^T \Sigma_2 w
\end{aligned}
$$

We can transform the problem to a simpler one:

$$
\max_w \ J(w) = \frac{||w^T \mu_1 - w^T \mu_2||_2^2}{w^T \Sigma_1 w + w^T \Sigma_2 w}
$$

Here we introduce two denotation here which are the *within-class scatter matrix*:

$$
S_w = \Sigma_1 + \Sigma_2
$$

and *between-class scatter matrix*:

$$
S_b = (\mu_1 - \mu_2)(\mu_1 - \mu_2)^T
$$

So now our problem is

–

$$\max_w \ J(w) = \frac{w^T S_b w}{w^T S_w w}$$

The $J(w)$ here is also known as the *generalized Rayleigh quotient*. Notice here the $J(w)$ is a quotient of quadratic terms of w, i.e. the length of w doesn't matter here but only the direction does—this is also very intuitive! So we can introduce constraint to form

$$\begin{aligned} \min_w \ & J(w) = -w^T S_b w \\ \text{s.t.} \quad & w^T S_w w = 1 \end{aligned}$$

By constructing the Lagrangian we get the norm solution:

$$w = S_w^{-1}(\mu_1 - \mu_2)$$

Considering the numerical stability of the solution, it is common in practice to perform *Singular Value Decomposition (SVD)* on S_w , such that $S_w = UEV^T$, where E is a real diagonal matrix with the singular values of S_w along its diagonal. Then, S_w^{-1} is obtained by $S_w^{-1} = VE^{-1}U^T$.

For a more generalised situation where we have N class with the i-th having m_i examples, we can denote

$$\begin{aligned} S_t &= S_b + S_w \\ &= \sum_{i=1}^{m} (x_i - \mu)(x_i - \mu)^T \end{aligned}$$

SWhere the μ is the mean of all examples. And accordingly we get:

$$S_w = \sum_{i=1}^{N} S_{w_i}$$

and

$$\begin{aligned} S_b &= S_t - S_w \\ &= \sum_{i=1}^{N} m_i (\mu_i - \mu)(\mu_i - \mu)^T \end{aligned}$$

A common optimization problem for this is

–

$$\max_W \frac{\text{tr}(W^T S_b W)}{\text{tr}(W^T S_w W)}$$

where $W \in \mathbb{R}^{d \times N-1}$.

The solution can be obtained by solving the following generalized eigenvalue problem: $S_B W = \lambda S_W W$. The closed-form solution for W is a matrix composed of the eigenvectors corresponding to the $N-1$ largest generalized eigenvalues of $S_w^{-1} S_b$.

Classification and logistic regression

In this session, we gonna be diving into *classification questions,* which is just like the regression problem, except that the values y we now want to predict take on only a small number of discrete values.

For now, we will focus on the *binary classification* problem in which y can take on only two values, 0 and 1. (Most of what we say here will also generalize to the multiple-class case.) 0 is also called *negative class*, 1 is also called *positive class*. Given $x^{(i)}$, the corresponding $y^{(i)}$ is also called the *label* for the training example.

Logistic regression

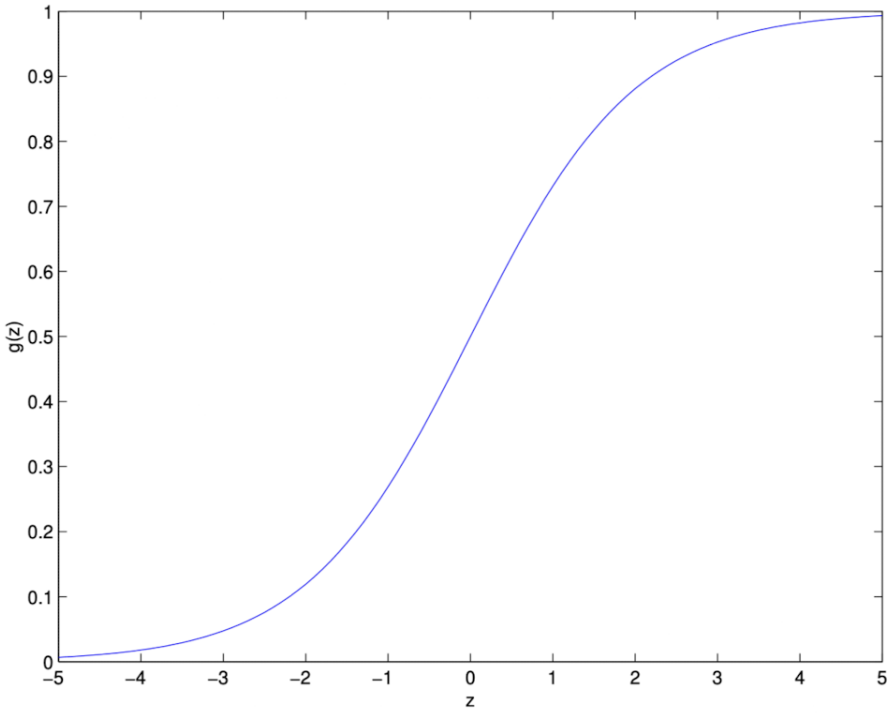
Let's take what we've already learned in linear regression, where we try to use a function we call $h_\theta(x)$ to predict. Yet in this part, it's intuitive that there is no need to include values larger than 1 or less than 0. To satisfy this request we define a new function

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}}$$

where

$$g(z) = \frac{1}{1 + e^{-z}}$$

You are familiar with this function since high school, yet you haven't really name it, given it's properties, we called it *logistic function* or the *sigmoid function*. Here's a look of it.



As before, we are keeping the convention of letting $x_0 = 1$, so that

$$\theta^T x = \theta_0 + \sum_{j=1}^{d} \theta_j^T x_j$$

For sure there are many other choice, yet when we talk about GLMs, and when we talk about generative learning algorithms we will find this is actually a natural choice.

Before moving on, here's a useful property of the derivative of the sigmoid function

$$g'(z) = g(z)(1 - g(z))$$

So, given the logistic regression model, how do we fit θ for it? Following how we saw least squares regression could be derived as the maximum likelihood estimator under a set of assumptions, let's endow our classification model with a set of probabilistic assumptions, and then fit the parameters via maximum likelihood.

First, lets assume that

$$P(y = 1 \mid x; \theta) = h_\theta(x)$$
$$P(y = 0 \mid x; \theta) = 1 - h_\theta(x)$$

For this logistic regression follows an *Bernoulli distribution*, we have can write this more compactly

$$p(y|x;\theta) = (h_\theta(x))^y(1 - h_\theta(x))^{1-y}$$

Like we did before, we define the maximum likelihood function, which is

$$L(\theta) = p(\mathbf{y} \mid X; \theta) = \prod_{i=1}^{n} p(y^{(i)} \mid x^{(i)}; \theta) = \prod_{i=1}^{n} \left(h_\theta(x^{(i)})^{y^{(i)}} (1 - h_\theta(x^{(i)}))^{1-y^{(i)}} \right)$$

Still we will be working on our log likelihood function rather than $L(\theta)$, the only thing we need to consider is how to maximize it. We decide to use a way of gradient ascent. Written in vectorial notation, our updates will therefore be given by $\theta := \theta + \alpha \nabla_\theta \ell(\theta)$. Spare the process, we jump right into our conclusion of our stochastic gradient ascent rule

$$\theta_j := \theta_j + \alpha(y^{(i)} - h_\theta(x^{(i)})x_j^{(i)}$$

If we compare this to the LMS update rule, we see that it looks identical; but this is not the same algorithm, because $h_\theta\left(x^{(i)}\right)$ is now defined as a non-linear function of $\theta^T x^{(i)}$.
Nonetheless, it's a little surprising that we end up with
the same update rule for a rather different algorithm and learning problem.
Is this coincidence, or is there a deeper reason behind this? We'll answer this when we get to GLM models.

Digression: the perceptron learning algorithm

We now digress to talk briefly about an algorithm that's of some historical interest, and that we will also return to later when we talk about learning theory. Consider modifying the logistic regression method to "force" it to
output values that are either 0 or 1 or exactly. To do so, it seems natural to change the definition of
g to be the threshold function:

$$g(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

If we then set $h_\theta(x)$ as before but using this modified definition of g, and if we use the update rule as before as well, then we have *perceptron learning algorithm*.

In the 1960s, this "perceptron" was argued to be a rough model for how individual neurons in the brain work. Given how simple the algorithm is, it will also provide a starting point for our analysis when we talk about learning theory later in this class. Note however that even though the perceptron may be cosmetically similar to the other algorithms we talked about, it is actually a very different type of algorithm than logistic regression and least squares linear regression; in particular, it is difficult to endow the perceptron's predictions with meaningful probabilistic interpretations, or derive the perceptron as a maximum likelihood estimation algorithm.

Another algorithm for maximizing $\ell(\theta)$

Let's use another way to maximize $\ell(\theta)$, considering *Newton' method*

$$\theta := \theta - \frac{f(\theta)}{f'(\theta)}$$

In maximizing the $\ell(\theta)$, we are actually in search of the point where $\ell'(\theta)$ is 0, so we can follow the following update rule

$$\theta := \theta - \frac{\ell'(\theta)}{\ell''(\theta)}$$

Something to think about: How would this change if we wanted to use Newton's method to minimize rather than maximize a function? It's actually simple, they are the same.

Lastly, in our logistic regression setting, θ is vector-valued, so we need to generalize Newton's method to this setting. The generalization of Newton's method to this multidimensional setting (also called the *Newton-Raphson method*) is given by

$$\theta := \theta - H^{-1}\nabla_\theta \ell(\theta)$$

When Newton's method is applied to maximize the logistic regression log likelihood function $\ell(\theta)$, the resulting method is also called *Fisher scoring*.

A few thing to notice here is that the Gradient is actually

$$\nabla_\theta \ell(\theta) = X^T(p - y)$$

and the Hessian matrix is

–

$$H = X^T W X$$

in which W is a dialectal matrix whose entries are $W_{ii} = p_i(1 - p_i)$

It’s just like what we find in practicing our linear regression, we tend to find that Newton’s method typically enjoys faster convergence than (batch) gradient descent, and requires many fewer iterations to get very close to the minimum. One iteration of Newton’s can, however, be more expensive than one iteration of gradient descent, since it requires finding and inverting an d-by-d Hessian.

Generalized linear models

So far, we've seen a regression example, and a classification example. In the regression example, we had $y|x;\theta \sim N(\mu, \sigma^2)$, and in the classification one, $y|x;\theta \sim \text{Bernoulli}(\phi)$, for some appropriate definitions of μ and ϕ as functions of x and θ. In this section, we will show that both of these methods are special cases of a broader family of models, called *Generalized Linear Models (GLMs)*. We will also show how other models in the GLM family can be derived and applied to other classification and regression problems.

Exponential family

To work our way up to GLMs, we will begin by defining *exponential family distributions*. We say that a class of distributions is in the exponential family if it can be written in the form of

$$p(y;\eta) = b(y)exp(\eta^T T(y) - a(\eta))$$

Here, η is called the *natural parameter (also called the canonical parameter)* of the distribution; $T(y)$ is the *sufficient statistic* (for the distributions we consider, it will often be the case that $T(y) = y$, here indicates that this could be substituted); and $a(\eta)$ is the *log*

partition function. The quantity $e^{-a(\eta)}$ essentially plays the role of a *normalization constant*, that makes sure the distribution $p(y;\eta)$ sums/integrates over y to 1.

A fixed choice of T, a and b defines a family (or set) of distributions that is parameterized by η; as we vary η, we then get different distributions within this family.

First let's see how can we get Bernoulli distribution from this general family. If we consider varying the mean ϕ of the Bernoulli, we surely can write this down

$$\begin{aligned} p(y;\phi) &= \phi^y(1-\phi)^{1-y} \\ &= \exp(y\log\phi + (1-y)\log(1-\phi)) \\ &= \exp\left(\log\left(\frac{\phi}{1-\phi}\right)y + \log(1-\phi)\right) \end{aligned}$$

compare the last entry to the exponential family, surely we can see that the natural parameter is given by $\eta = log(\phi/(1-\phi))$. Interestingly, if we invert this definition for η by solving for ϕ in terms of η, we obtain $\phi = 1/(1+e^{-\eta})$. This is the familiar sigmoid function! This will come up again when we derive logistic regression as a GLM.

We also have

$$\begin{aligned} T(y) &= y \\ a(\eta) &= -log(1-\phi) = log(1+e^{\eta}) \\ b(y) &= 1 \end{aligned}$$

Similarly, we can see that the Gaussian is also in the exponential family where

$$\begin{aligned} p(y;\mu) &= \frac{1}{\sqrt{2\pi}}\exp\left(-\frac{1}{2}(y-\mu)^2\right) \\ &= \frac{1}{\sqrt{2\pi}}\exp\left(-\frac{1}{2}y^2\right)\cdot\exp\left(\mu y - \frac{1}{2}\mu^2\right) \end{aligned}$$

where

$$
\begin{aligned}
\eta &= \mu \\
T(y) &= y \\
a(\eta) &= \frac{\mu^2}{2} = \frac{\eta^2}{2} \\
b(y) &= \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{y^2}{2}\right)
\end{aligned}
$$

There're many other distributions that are members of the exponential family: The multinomial (which we'll see later), the Poisson (for modeling count-data; also see the problem set); the gamma and the exponential (for modeling continuous, non-negative random variables, such as time-intervals); the beta and the Dirichlet (for distributions over probabilities); and many more.

Constructing GLMs

Suppose that you own a store which sells fleshlights, and for now you want to know the number y of customers arriving in your store (or number of page-views on your website) in any given hour, based on certain features x such as store promotions, recent advertising, weather, day-of-week, etc. We know that the Poisson distribution usually gives a good model for numbers of visitors, so we can apply a Generalized Linear Model (GLM) in this example.

To derive a GLM for this problem, we will make the following three assumptions about the conditional distribution of y given x and about our model:

1. $y|x;\theta \sim \text{Exponential Family}(\eta)$. I.e., given x and θ, the distribution of y follows some exponential family distribution, with parameter η.
2. Given x, our goal is to predict the expected value of $T(y)$ given x. In most of our examples, we will have $T(y) = y$, so this means we would like the prediction $h(x)$ output by our learned hypothesis h to satisfy $h(x) = E[y|x]$. (Note that this assumption is satisfied in the choices for $h_\theta(x)$ for both logistic regression and linear regression. For instance, in logistic regression, we had $h_\theta(x) = p(y = 1|x;\theta) = 0 \cdot p(y = 0|x;\theta) + 1 \cdot p(y = 1|x;\theta) = E[y|x;\theta]$.)
3. The natural parameter η and the inputs x are related linearly: $\eta = \theta^T x$. (Or, if η is vector-valued, then $\eta_i = \theta_i^T x$.)

This three assumptions are better took as "designed choices" instead of an real assumption, we will later find this recipe for making GLMs has a lot of good properties and could easily helps us with deriving both logistic regression and ordinary least squares as GLMs.

Ordinary least squares

To show that ordinary least squares is a special case of the GLM family of models, consider the setting where the target variable y (also called the *response variable* in GLM terminology) is continuous, and we model the conditional distribution of y given x as a Gaussian $N(\mu, \sigma^2)$. (Here, μ may depend x.) So, we let the $\text{Exponential Family}(\eta)$ distribution above be the Gaussian distribution. As we saw previously, in the formulation of the Gaussian as an exponential family distribution, we had $\mu = \eta$. So, we have

$$\begin{aligned} h_\theta(x) &= E[y|x;\theta] \\ &= \mu \\ &= \eta \\ &= \theta^T x. \end{aligned}$$

all this equalities are given from previous assumptions and the properties of the Gaussian.

Logistic regression

And as for the logistic regression, we have the same simple thing to do and we can have the following result

$$\begin{aligned} h_\theta(x) &= E[y|x;\theta] \\ &= \phi \\ &= \frac{1}{1+e^{-\eta}} \\ &= \frac{1}{1+e^{-\theta^T x}} \end{aligned}$$

So, this gives us hypothesis functions of the form $h_\theta(x) = 1/(1+e^{-\theta^T x})$. If you are previously wondering how we came up with the form of the logistic function $1/(1+e^{-z})$, this gives one answer: Once we assume that y conditioned on x is Bernoulli, it arises as a consequence of the definition of GLMs and exponential family distributions.

And at last let we introduce a little more terminology. The function g giving the distribution's mean as a function of the natural parameter ($g(\eta) = E[T(y);\eta]$) is called the *canonical response function*. Its inverse, g^{-1}, is called the *canonical link function*. Thus, the canonical response function for the Gaussian family is just the identify function; and the canonical response function for the Bernoulli is the logistic function.

Softmax regression

Let's look at one more example of a GLM. Consider a classification problem in which the response variable y can take on any one of k values, so $y \in \{1, 2, ..., k\}$. We will thus model it as distributed according to a multinomial distribution. Let's derive a GLM for the multinomial distribution.

To parameterize a multinomial over k possible outcomes, one could use k parameters $\phi_1, ..., \phi_k$ specifying the probability of each of the outcomes. However, these parameters would be redundant, or more formally, they would not be independent (since knowing any $k-1$ of the ϕ_i's uniquely determines the last one, as they must satisfy $\sum k_i = 1$ $\sum \phi_i = 1$). So, we will instead parameterize the multinomial with only $k-1$ parameters, $\phi_1, ..., \phi_{k-1}$, where $\phi_i = p(y = i; \phi)$, and $p(y = k; \phi) = 1 - \sum_{i=1}^{k-1} \phi_i$. For notational convenience, we will also let $\phi_k = 1 - \sum_{i=1}^{k-1} \phi_i$, but we should keep in mind that this is not a parameter, and that it is fully specified by $\phi_1, ..., \phi_{k-1}$.

And in this situation we define $T(y)$ as

$$T(1) = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad T(2) = \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad T(3) = \begin{bmatrix} 0 \\ 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \quad \cdots, \quad T(k-1) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix},$$

where $T(y)$ is not exactly the y as mentioned before but rather be a $k-1$ dimensional vector. We will write $(T(y))_i$ to denote the i-th element of the vector $T(y)$.

We introduce one more very useful piece of notation. An *indicator function* $1\{\cdot\}$ takes on a value of 1 if its argument is true, and 0 otherwise ($1\{True\} = 1, 1\{False\} = 0$). Further, we have that $E[(T(y))_i] = P(y = i) = \phi_i$.

So we try to derive it as exponential family,

$$
\begin{aligned}
& p(y;\phi) \\
& = \phi_1^{1\{y=1\}} \phi_2^{1\{y=2\}} \cdots \phi_k^{1\{y=k\}} \\
& = \phi_1^{1\{y=1\}} \phi_2^{1\{y=2\}} \cdots \phi_k^{1-\sum_{i=1}^{k-1} 1\{y=i\}} \\
& = \phi_1^{(T(y))_1} \phi_2^{(T(y))_2} \cdots \phi_k^{1-\sum_{i=1}^{k-1} (T(y))_i} \\
& = \exp\left((T(y))_1 \log(\phi_1) + (T(y))_2 \log(\phi_2) + \cdots + \left(1 - \sum_{i=1}^{k-1} (T(y))_i \right) \log(\phi_k) \right) \\
& = \exp\left((T(y))_1 \log\left(\frac{\phi_1}{\phi_k}\right) + \cdots + (T(y))_{k-1} \log\left(\frac{\phi_{k-1}}{\phi_k}\right) + \log(\phi_k) \right) \\
& = b(y) \exp\left(\eta^T T(y) - a(\eta)\right)
\end{aligned}
$$

by comparison, we can simply find that

$$
b(y) = 1
$$

$$
\eta = \begin{bmatrix} log(\phi_1/\phi_k) \\ log(\phi_2/\phi_k) \\ . \\ . \\ . \\ log(\phi_{k-1}/\phi_k) \end{bmatrix}
$$

$$
a(y) = -log(\phi_k)
$$

The link function is obviously given by

$$
\eta_i = log \frac{\phi_i}{\phi_k}
$$

To our convenience, we denote $\eta_k = log \frac{\phi_k}{\phi_k} = 0$. To invert the link function and derive the response function, we therefore have that

$$
\begin{aligned}
e^{\eta_i} &= \frac{\phi_i}{\phi_k} \quad \text{for} \quad i = 1, 2, \ldots, k-1, \\
\phi_k e^{\eta_i} &= \phi_i, \\
\phi_k \sum_{i=1}^{k} e^{\eta_i} &= \sum_{i=1}^{k} \phi_i = 1
\end{aligned}
$$

so this implies that $\phi_k = \frac{1}{\sum_{i=1}^{k} e^{\eta^i}}$, which could be substitute back into the equation, and hence we'd work our response function which is

$$\phi_i = \frac{e^{\eta_i}}{\sum_{j=1}^{k} e^{\eta^j}}$$

This function mapping from the η's to the ϕ's is called the *softmax function*.

Remember that $\eta_i = \theta_i^T x (for\ i = 1, ..., k-1)$, where $\theta_1, ..., \theta_{k-1} \in \mathbb{R}^{d+1}$ are the parameters of our model. For notational convenience, we can also define $\theta_k = 0$, so that $\eta_k = \theta_k^T x = 0$, as given previously. And our conditional diatribution of y is now

$$\begin{aligned} p(y = i|x; \theta) &= \phi_i \\ &= \frac{e^{\eta_i}}{\sum_{j=1}^{k} e^{\eta_j}} \\ &= \frac{e^{\theta_i^T x}}{\sum_{j=1}^{k} e^{\theta_j^T x}} \end{aligned}$$

This model, which applies to classification problems where $y \in \{1, ..., k\}$, is called *softmax regression*. It is a generalization of logistic regression. The result of our hypothesis will be a vector whose i-th entry will be the value of $p(y = i|x; \theta)$

$$
\begin{aligned}
h_\theta(x) &= E[T(y)|x;\theta] \\
&= E\begin{bmatrix} 1\{y=1\} \\ 1\{y=2\} \\ \vdots \\ 1\{y=k-1\} \end{bmatrix} \\
&= \begin{bmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \phi_{k-1} \end{bmatrix} \\
&= \begin{bmatrix} \frac{\exp(\theta_1^T x)}{\sum_{j=1}^k \exp(\theta_j^T x)} \\ \frac{\exp(\theta_2^T x)}{\sum_{j=1}^k \exp(\theta_j^T x)} \\ \vdots \\ \frac{\exp(\theta_{k-1}^T x)}{\sum_{j=1}^k \exp(\theta_j^T x)} \end{bmatrix}
\end{aligned}
$$

To learn our parameters, we do what we did before: write down the log likelihood function, and we could apply gradient ascent or Newton's method accordingly.

$$
\begin{aligned}
\ell(\theta) &= \sum_{i=1}^n \log p(y^{(i)}|x^{(i)};\theta) \\
&= \sum_{i=1}^n \log \prod_{l=1}^k \left(\frac{e^{\theta_l^T x^{(i)}}}{\sum_{j=1}^k e^{\theta_j^T x^{(i)}}} \right)^{1\{y^{(i)}=l\}}
\end{aligned}
$$

As you can tell easily, one good property of softmax is that it will amplify the distinctions among the scores. This occurs because of the exponential function, which magnifies larger scores and diminishes smaller ones relative to the largest score in the input vector. As a result: Large scores become significantly more dominant in the final output, which leads to sharper probabilities.

Decision trees and Ensemble learning

Weak learners vs. strong learners

Before we start our discussion here, we need to draw a quick look on what's a weak learner and what's a strong learner.

To express this to notion clearly and mathematically, we introduce $\delta, \epsilon \in (0, 1)$, and let $\gamma \in (0, \frac{1}{2})$. These will represent tolerance of uncertainty, tolerance of inaccuracy, and achievable improvement in accuracy, respectively.

Roughly speaking, we say a learning algorithm is a strong learner when for any choice of δ, ϵ, we have some $m = m_{\mathcal{H}}(\delta, \epsilon) \in N$ such that, after learning on at least m i.i.d. samples from the true distribution, we have

$$\mathbb{P}(\text{Accuracy}(f) > 1 - \epsilon) \geq 1 - \delta$$

That is to say, there's always some number of training examples that we could see such that we are guaranteed to learn a (probably, approximately) correct model.

We'll call a learning algorithm a γ-weak learner if for any choice of δ, we have some $m = m_{\mathcal{H}}(\delta) \in N$ such that, after learning on at least m i.i.d. samples from the true distribution, we have

$$\mathbb{P}(\text{Accuracy}(f) > \frac{1}{2} + \gamma) \geq 1 - \delta$$

Now we are only saying that there is always some number of training examples that we could see such that we learn a model that (probably) does a little bit better than randomly guessing on the binary classification task, which would yield an accuracy of 0.5. Note that in these examples, the Accuracy function is taken to give the expected accuracy of a model with respect to the true distribution.

Decision trees

Intro

Formally, a *decision tree* can be thought of as a mapping from some k regions of the input domain $\{R_1, R_2, ..., R_k\}$ to k corresponding predictions $\{w_1, w_2, ..., w_k\}$. We require these regions to partition the input domain, meaning that there is no intersection between any two regions, and the union of the regions recovers the entire input domain.

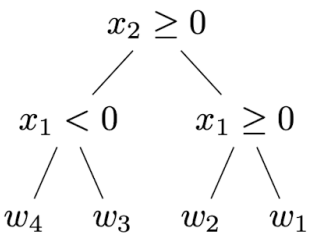
In a decision tree, we define our prediction as

$$f(x) = \sum_{j=1}^{k} w_j \mathbf{1}[\mathbf{x} \in \mathbf{R_j}],$$

where the w_j is the for most of the times calculated as the average value of the labels of the training data in region R_j

$$w_j = \frac{\sum_{i=1}^{n} y^{(i)} \mathbf{1}[\mathbf{x^{(i)}} \in \mathbf{R_j}]}{\sum_{i=1}^{n} \mathbf{1}[\mathbf{x^{(i)}} \in \mathbf{R_j}]}$$

If we visualize this with a figure of a tree, it would be like



where we follow the the right branch of a node if the condition is met, and the left one if not. We can notice here that each of our regions R_j corresponds to a single leaf (childless node) in this tree representation; indeed, this property will hold for all decision tree models we study, meaning that the number of regions k in a decision tree model is exactly the number of leaves in any corresponding tree representation.

Model fitting

Now the question has come to how to fit our model, or that is to say how do we split the dataset? Let's first take a mathematical look at the problem.

To fit the model, we seek to optimize a loss function just as we would in other supervised learning models. Here, our loss function looks like

$$L(f) = \sum_{i=1}^{n} \ell\big((y^{(i)}, f(x^{(i)})\big) = \sum_{j=1}^{k} \sum_{x^{(i)} \in R_j} \ell(y^{(i)}, w_j).$$

The first equality is a general form familiar to us from our study of other supervised learning models, while the second gives an equivalent representation using the specifics of the decision tree model.

Unfortunately, it turns out that calculating an optimal (maximally efficient) tree model to minimize this loss function is NP-complete, i.e., intractable as far as we know. Instead, we use a greedy algorithm to fit our model to the data. That is to say, we are doing this that we decide whether or not to split the node according to some condition $\mathbf{1}[\mathbf{x_j} \geq \theta]$ for some $j \in \{1, ..., d\}$ and some $\theta \in \mathbb{R}$.

$$j, \theta = \arg \max_{j,\theta} G(j, \theta)$$

where the "gain" function $G(\cdot\ ,\ \cdot)$ and assume that this function gives us some measure of the improvement that comes from doing a certain split at a given node. That means, if the split give us the biggest gain, we'll then apply this split, and then recurse at any other decision node.

If we choose not to split the node (we will discuss when we make this choice shortly), then we assign a prediction w for the region corresponding to this (leaf) node, usually according to the heuristics described in the previous section.

And how do we decide which value should be our split point? Usually, we use the average value of two adjacent features, and we have $\theta = \frac{x_j^{(i)} + x_j^{(i+1)}}{2}$, note that here all x_j's have already been sorted.

Note that to maximize the gain function, is indeed to minimize another abstract cost function $C(\cdot)$, we can thus let S be the set of datapoints associated with the node whose split we wish to evaluate. Let L and R be the datapoints associated with the left and right children of the original node after splitting according to $\mathbf{1}[\mathbf{x_j} \geq \theta]$. An intuitive representation of our gain function in terms of our cost function would be

$$G(j, \theta) = C(S) - \left(\frac{|L|}{|S|} C(L) + \frac{|R|}{|S|} C(R) \right)$$

You'll see clearly that to maximize the G is to minimize the cost function(to get a smaller cost after splitting). To fully concretize the gain function, all that remains now is to make our cost function explicit. There are many usable cost functions $C(\cdot)$, perhaps most straightforward of which would be our loss function

$$\frac{1}{|S|} \sum_{x^{(i)} \in R} (y^{(i)}, w)$$

where S is the set of datapoints associated with the node whose cost we evaluate, R is the region associated with this node, and w is the prediction for region R. But reusing the loss function isn't necessary here—one viable cost function for a regression tree would be the mean squared error

$$\frac{1}{|S|}\sum_{x^{(i)}\in R_\ell}(y^{(i)}-w)^2$$

And for the classification tree this could be the entropy

$$\sum_{c\in C}-p_c\log p_c$$

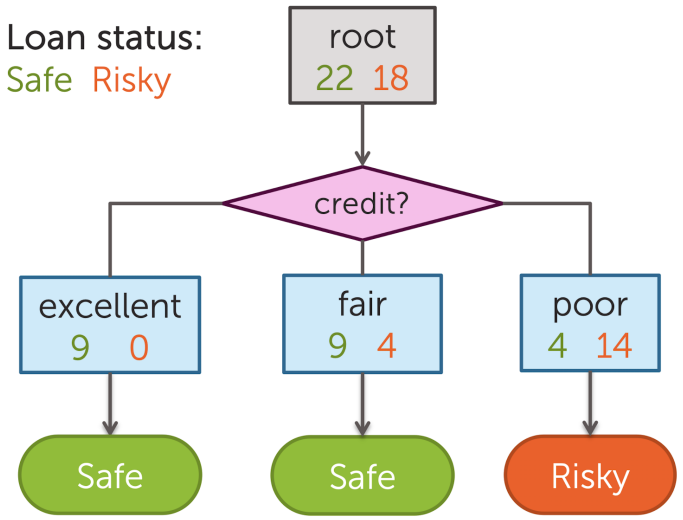
where C is the set of classes and p_c is the fraction of datapoints in S of class c.

The above are a bit more abstract process of the fitting, here we have a more concrete example to understand.

Consider in a case where we have to decide whether it is safe to approve someone a loan, and we give them labels as safe and risky according to their income, credit and so on. Let's start with all the data at first, where

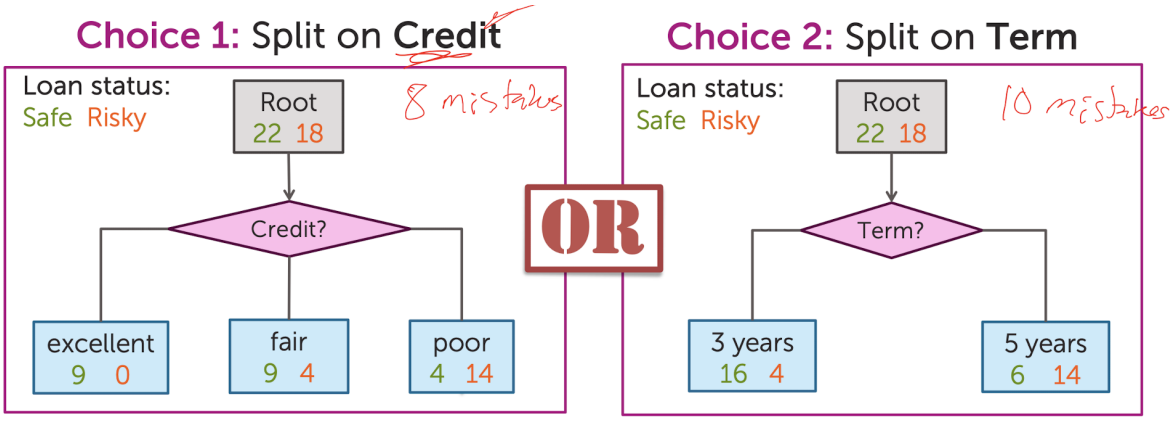
Safe	Risky
22	18

We call this our *root node*, as we did in the discrete math. And now if we split for the first time, according to their loan status, we have our first *decision stump*, which looks like



The excellent, fair, and bad label are call intermediate nodes, for each intermediate node, we set $\hat{y}$ = majority value. Of course there are a lot of ways to choose to split the

dataset, so we have to find a best way, that's where we apply our greedy algorithm.



To quantify this metric, we define the error as

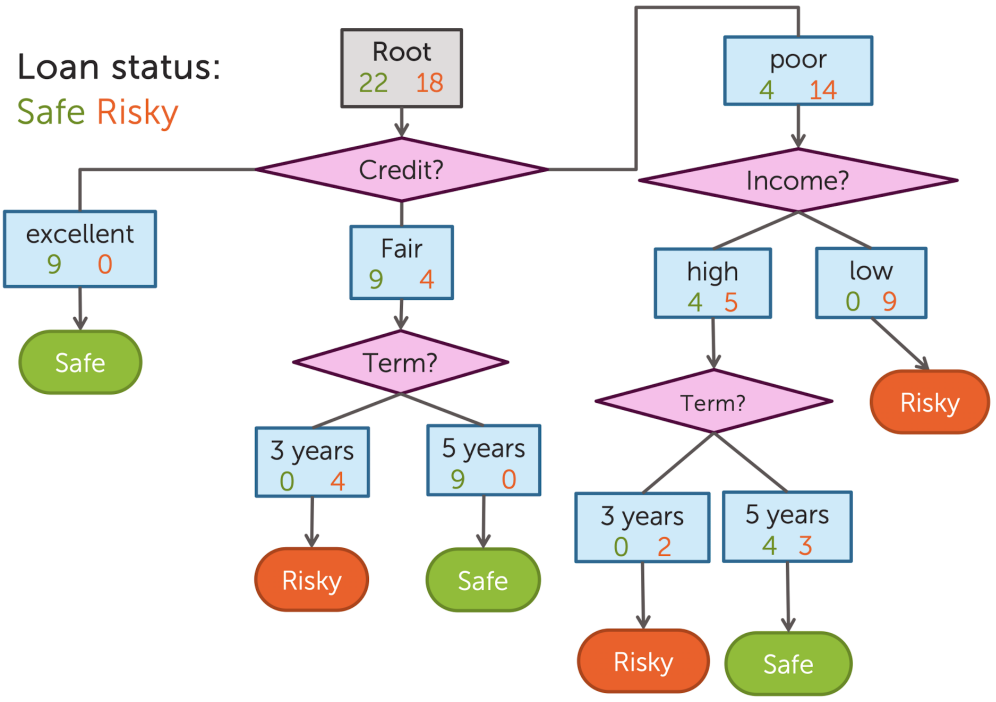
$$\text{error} = \frac{\text{mistakes}}{\text{total points}}$$

Then we can conclude our algorithm as

Feature split selection algorithm

- Given a subset of data M (a node in a tree)
- For each feature $h_i(x)$:
  1. Split data of M according to feature $h_i(x)$
  2. Compute classification error of split
- Chose feature $h^*(x)$ with lowest classification error

And let's now discuss the recursion and stopping conditions. For every tree stump we have now, as we mentioned before, if all datapoints are rightfully classified, we set it to be the leaf, and thus stop recurse on it. And for the other stumps, we do what we already did before with other splittable features. The final results should be like



To summarize, we have 3 possible stopping conditions, which are

- All data agrees on y
- Already split on all features
- Stop if no split reduces the classification error

And for continuous data we may apply *threshold split*, and note that same feature can be used multiple times for threshold splits.

Regularization

How well can a tree do on an arbitrary modeling task? In other words, if we let the greedy algorithm described above run indefinitely, what happens?

A tree of sufficient depth can perfectly fit (achieve 0 loss) on any internally-consistent training set. Furthermore, our greedy algorithm will achieve this fit eventually (can you see why?), though usually suboptimally as noted earlier. However, allowing the tree to reach such depths often invites the twin problems of overfitting to our training set and creating an undesirably large model. For the sake of regularization, heuristics are typically used for deciding not to split a node further. A few of these heuristics are limiting overall tree depth, limiting the number of leaves (distinct regions) in the model,

barring additional splits when there are too few datapoints in a node, and not splitting if the gain described above is beneath some threshold.

Another method for regularization would be to ”prune” the tree after letting it (greedily) reach a 0-loss fit. Pruning generally involves creating another function that evaluates whether the increased performance associated with a node (or set of nodes) is worth the extra model size and reduced generalization. If not, one can “prune” the node (nodes) by merging it (them) back into tree. Pruning can often be done greedily, as was our fitting algorithm, but starting from leaves instead of the root of the tree.

Summary

- Advantages of using trees include but are not limited to:
  - Highly interpretable
  - Robust to outliers
  - Robust to mix of continuous and discrete features
  - Robust to monotone transformations of input (transformations that don’t alter the output of our sorting strategy from above)
  - Can usually achieve a “decent” fit relatively quickly, even on large datasets
- Disadvantages of using trees include but are not limited to:
  - Tend to generalize poorly, even when regularized
  - Highly unstable around boundaries

Improvements via Ensemble Learning

Ensemble learning is method in machine learning that reduces model instability by averaging predictions over multiple instances of similar models. Given some individual models $F_1, ..., F_m$ (not necessarily decision trees), we can write our overall ensemble model output as

$$f(x) = \sum_{i=1}^{n} \beta_i F_i(x)$$

It’s like a vote, where we have a lot of models to vote to decide what prediction to make. Ensemble learning tends to reduce variance without significantly increasing bias,

resulting in improved performance overall. In regression, the *averaging* mentioned above can be interpreted literally, whereas in classification we can use a *plurality vote* again.

Bagging

The first ensemble method we'll introduce is bagging, which stands for "bootstrap aggregating". Bootstrapping is a general technique where we aim to simulate drawing a new sample from the true underlying distribution from which our training set is generated (since we don't have access to the true distribution outright).

Concretely, we generate a "new" dataset by sampling uniformly and with replacement from our original one. And on these different "new" datasets, we train different models (weak learners) and make predictions by averaging their results (regression) or by a plurality vote (classification).

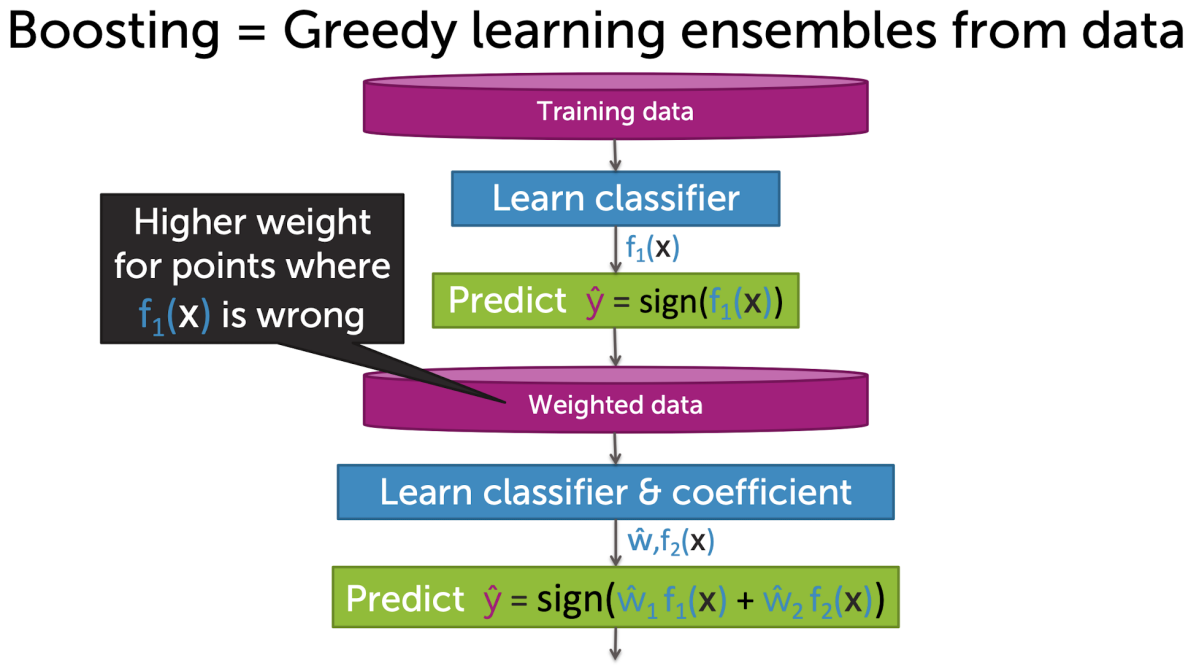
We can mathematically prove that each individual model will only (on average) see 63.2% of the data, which is disadvantageous in terms of individual models' performances on the overall dataset, but advantageous in the sense that it prevents the ensemble as a whole from fixating too much on a few datapoints, lowering the overall variance.

But is bagging sufficient for creating significant variation between the models in our ensemble? This largely depends on the underlying models themselves. If the models are sufficiently low variance (which as we've learned often implies having high bias), then bagging is likely to produce highly similar models, reducing the value of this intervention. However, if we use unstable (high variance) models, like decision trees, then we are effectively harnessing the instability of our base learner to help ensure the quality of our ensemble learning procedure.

The most common method of bagging is *the random forest algorithm*, which went a little further than the bagging itself. To further decrease the correlation between individual models, in each of our individual trees, we'll consider only a subset of features at each node-splitting step in our greedy algorithm. Often this subset will be chosen uniformly randomly at each node, where the subset size is fixed to some hyperparameter $k < d$. It turns out that random forest performs better when the base learners are the decision tree.

Boosting

Generally speaking, boosting is to train your model while focusing on the points where, you are likely to make mistakes before, and by doing so we can reduce our bias and obtain a rather well performing model. That is to say, now we have weight for each of our datapoints, and points are no longer equally valued as before.



First let's get to analyse boosting a little before we introduce concrete methods. Remember that

$$f(x) = \sum_{i=1}^{m} \beta_i F_i(x)$$

As mentioned, our overall goal with this (or any) model in the supervised setting is to minimize a loss function of the form

$$\mathcal{L}(f) = \sum_{i=1}^{n} \ell(y^{(i)}, f(x^{(i)}))$$

Now we try to train our next weak learner F_{m+1} according to the present ensemble model $f_n(x)$, thus, a strong learner can be developed from the amalgamation of several easy-to-train and easy-to-interpret weak learners, and sometimes this ensemble of weak learners is preferable to developing a single strong one.

The type of boosting we'll cover is referred to as *forward stagewise additive modeling*, and it classically takes the following form: At iteration i of our ensemble creation, we create the model F_i. To do so, we compute

$$\begin{aligned}\beta_i,\, \theta_i &= \arg\max_{\beta,\,\theta} \mathcal{L}(f_{i-1} + \beta_i F_i) \\ &= \arg\max_{\beta,\,\theta} \sum_{j=1}^{n} \ell\Big(y^{(j)}, f_{i-1}(x^{(j)}) + \beta_i F_i(x^{(j)};\,\theta)\Big)\end{aligned}$$

where θ represents the parameters of the new model F_i. We then set $F_i(x) := F_i(x;\theta_i)$ and afterwards set

$$f_i(x) := f_{i-1}(x) + \beta_i F_i(x)$$

to update our new ensemble model.

Note that here the notation $F_i(x^{(i)};\theta)$ implies that the F specific choice within a model family parameterized by θ. Yet this is not always the case when our weak learners are the decision trees.

Now, it's time for us to introduce some concrete booting methods. The first is *gradient boosting*.

In gradient boosting, instead of learning some parameters β_i, θ_i to develop $F_i(\,\cdot\,;\theta_i)$, we instead will be concerned with learning the entire function F_i itself:

$$F_i = \arg\max_{F} \mathcal{L}(f_{i-1} + F) = \arg\max_{F} \sum_{j=1}^{n} \ell\Big(y^{(j)}, f_{i-1}(x^{(j)}) + F(x^{(j)})\Big)$$

Yet the question here is that this is optimization is carried on the space of functions, which means this could be a real difficult problem to calculate! Thus, we consider to carry out this optimization in the form of (functional) gradient descent. At iteration i, let

$$f = \Big(f_{i-1}(x^{(1)}), \ldots, f_{i-1}(x^{(n)})\Big)$$

be the vector of our current function's evaluations on our datapoints. Additionally, let

$$g = \left(\frac{\partial \ell(f_{i-1}(x^{(1)}), y^{(1)})}{\partial f_{i-1}(x^{(1)})}, \ldots, \frac{\partial \ell(f_{i-1}(x^{(n)}), y^{(n)})}{\partial f_{i-1}(x^{(n)})} \right)$$

be the gradient of our loss function with respect to f. Then we see that $f := f - \alpha g$ can be thought of as an updated vector of evaluations after a single step of gradient descent.

However, this alone isn't too useful to us. We don't just seek to update f, which is our model's outputs on our training datapoints; we want to update f_{i-1} for all possible inputs x. So simply letting $f_i := f_{i-1} - \alpha g$ doesn't make sense—we'd be taking the difference between a function (which we've seen can be represented as a infinite-dimensional vector) and an n-vector. We'd need the new function F_i to be defined over all real numbers to achieve the function-level gradient descent we're after, thus we're doing a upgrade for the function which could be really useful over all real number realm.

In order to get a more general representation of g that extends to all inputs, we can fit a weak learner to our gradient step via a supervised learning task. In particular, we can write

$$F_i := \arg\min_F \sum_{j=1}^{n} \left((-g_j - F(x^{(j)}) \right)^2$$

Why it is like this? The reason are quite obvious: when the F_i are overall most close to $-g$, which is the gradient, then it's intuitive our F_i is the best to update because it is close to the gradient itself. So, we can now write

$$f_i(x) := f_{i-1}(x) + \alpha F_i(x)$$

This will accomplish our function-level gradient descent interpretation of boosting.

Note that as we iterate through this process, we'll be fitting a model to the gradient of our loss with respect to our model's current predictions, meaning that each successive model we add is tailored to weaknesses in the current ensemble model. Under this interpretation, we can think of boosting as simply adding on models that are trained according to a relabelled version of the original dataset, where the relabelling is done to improve performance on these aforementioned weaknesses.

Where do trees fit into this? Remember, in the boosting paradigm we want fast, weak learners that in the aggregate can form a powerful ensemble model. Decision trees are considered weak learners when they are highly regularized, and thus are a perfect candidate for this role. And for the next, we'll be introducing *the AdaBoost*.

The AdaBoost follows a structure like this:

1. Start with same weight for all data points: $\alpha_i = \frac{1}{N}$.
2. For $t = 1, \ldots, T$ (T weak learners)
    a. Learn $f_t(x)$ with data weights α_i. That is to say, we
    b. Compute coefficient for functions w_t.
    c. Update data weights α_i based on errors for each models and normalize it.
3. Final model predicts by: $y = \text{sign}\left(\sum_{t=1}^{T} w_t f_t(x)\right) = f(x)$

Now we only need to know how to update our weights and coefficients.

Like we mentioned before, the coefficient w_t are

$$w_t = \frac{1}{2}\ln\left(\frac{1 - \text{weighed error}(f_t)}{\text{weighed error}(f_t)}\right) = \frac{1}{2}\ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$$

And for our data weights, we have

$$\alpha_i = \frac{\alpha_i e^{(-w_t y_i f_t(x_i))}}{Z_t} = \begin{cases} \frac{\alpha_i e^{-w_t}}{Z_t} & \text{if} \quad f_t(x_i) = y_i \\ \frac{\alpha_i e^{w_t}}{Z_t} & \text{if} \quad f_t(x_i) \neq y_i \end{cases}$$

Note that here the Z_t is the normalization factor, which is $\sum_{i=1}^{N} \alpha_i$. The updating rule is intuitive, if we get the data point wrong, then we increase the weight for this point by times a scaling factor $e^{w_t} > 1$, otherwise we do the the contrary. I.e. we try to fix our mistake before and by increasing their weight to do better on the mistaken points. According to *the AdaBoost Theorem*, training error of boosted classifier $\to 0$ as $T \to \infty$, as long as we can find a classifier with a error lower than 0.5, we have this conclusion, which could also be put more formally as

$$\frac{1}{N}\sum_{i=1}^{N} \mathbf{1}[F(x_i) \neq y_i] \leq \sum_{i=1}^{N} \exp(-y_i \cdot \text{score}(x_i)) = \prod_{t=1}^{T} Z_t$$

where

$$\text{score}(x) = \sum_{t=1}^{T} w_t f_t(x), \quad F(x) = \text{sign}(\text{score}(x))$$

If we minimize $\prod_{t=1}^{T} Z_t$, we minimize our training error. We can tighten this bound greedily by choosing w_t, f_t on each iteration to minimize

$$Z_t = \sum_{i=1}^{N} \alpha_{i,t} \exp\left(-w_t y_i f_t(x_i)\right)$$

And For the last, we introduce a little about *the XGBoost*.

The XGBoost, which sees out-of-the-box usage in industry settings regularly (at time of writing). The main differences between XGBoost and gradient tree boosting are that

- We optimize the loss function with a regularization term included:

$$\mathcal{L}(f) = \sum_{i=1}^{n} \ell\left(y^{(i)}, f(x^{(i)})\right) + \gamma J + \frac{1}{2}\lambda \sum_{j=1}^{J} w_j^2$$

  J is the number of leaves, w_j is the prediction value associated with the j-th leaf in the model, and γ, λ are scalar-valued hyperparameters.

- We use a subset of features when splitting tree nodes like in random forests.
- We use a second-order approximation rather than just a gradient, i.e., the Hessian of the loss with respect to our current predictions makes an appearance in our boosting algorithm along with the gradient.

Generative learning algorithm

So far, we've mainly been talking about learning algorithms that model $p(y|x;\theta)$, the conditional distribution of y given x. Consider the other way around.

Say we want to tell the difference between a dog and a cat, so if we do this like before, we'd input all the feature we collected about this animal, denoted as X, and then map it to y, which is in the range of $\{0,1\}$. To summarise this process more humanly, we're actually tell the difference by it's feature, we know which features indicates a cat, and which means a dog.

But, we can do this in another way. If we have a picture of a cat, and a dog, then we can identify an animal by compare it to our existing picture. In a mathematical way to express, we are modeling $p(y|x;\theta)$ with $p(x|y;\theta)$.

Algorithms that try to learn $p(y|x)$ directly (such as logistic regression), or algorithms that try to learn mappings directly from the space of inputs X to the labels $\{0,1\}$, (such as the perceptron algorithm) are called *discriminative learning algorithms*. Here, we'll talk about algorithms that instead try to model $p(x|y)$ (and $p(y)$). These algorithms are called *generative learning algorithms*. After modeling $p(y)$ (called the *class priors*) and $p(x|y)$, our algorithm could be

$$p(y|x) = \frac{p(x|y) \cdot p(y)}{p(x)}$$

Actually, if were calculating $p(y|x)$ in order to make a prediction, then we don't actually need to calculate the denominator, since

$$\begin{aligned}\arg\max_y p(y|x) &= \arg\max_y \frac{p(x|y)p(y)}{p(x)} \\ &= \arg\max_y p(x|y)p(y)\end{aligned}$$

Gaussian discriminant analysis

The first GLA we're about to learn is *Gaussian discriminant analysis (GDA)*. In this model, we'll assume that $p(x|y)$ is distributed according to a multivariate normal distribution, and of course, the y itself follows a Bernoulli distribution, that could be denoted as

$$\begin{aligned} y &\sim \text{Bernoulli}(\phi) \\ x \mid y = 0 &\sim \mathcal{N}(\mu_0, \Sigma) \\ x \mid y = 1 &\sim \mathcal{N}(\mu_1, \Sigma) \end{aligned}$$

With this three basic assumptions about this model, we're now able to give this following equations

$$\begin{aligned} p(y) &= \phi^y(1-\phi)^{1-y} \\ p(x \mid y=0) &= \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu_0)^T\Sigma^{-1}(x-\mu_0)\right) \\ p(x \mid y=1) &= \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu_1)^T\Sigma^{-1}(x-\mu_1)\right) \end{aligned}$$

And we try to work out the likelihood function which is given by

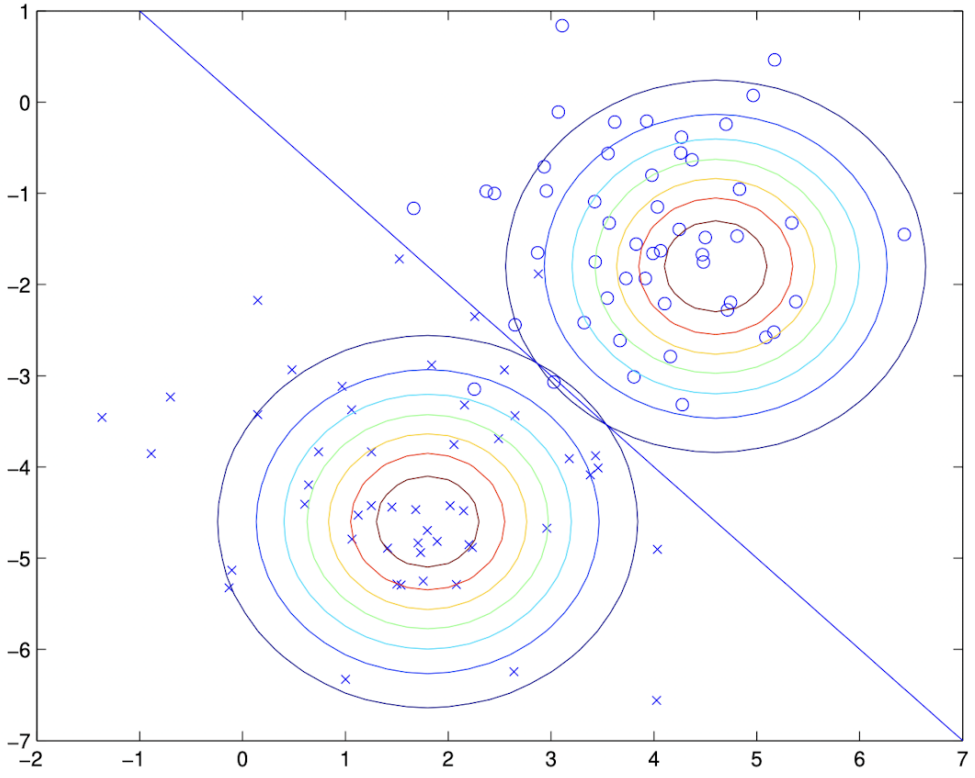
$$\begin{aligned}\ell(\phi, \mu_0, \mu_1, \Sigma) &= \log \prod_{i=1}^{n} p(x^{(i)}, y^{(i)}; \phi, \mu_0, \mu_1, \Sigma) \\ &= \log \prod_{i=1}^{n} p(x^{(i)} \mid y^{(i)}; \mu_0, \mu_1, \Sigma) p(y^{(i)}; \phi)\end{aligned}$$

In problem set 1, we solved this which results:

$$
\begin{aligned}
\phi &= \frac{\sum_{i=1}^{n} 1\{y^{(i)} = 1\}}{n} \\
\mu_0 &= \frac{\sum_{i=1}^{n} 1\{y^{(i)} = 0\} x^{(i)}}{\sum_{i=1}^{n} 1\{y^{(i)} = 0\}} \\
\mu_1 &= \frac{\sum_{i=1}^{n} 1\{y^{(i)} = 1\} x^{(i)}}{\sum_{i=1}^{n} 1\{y^{(i)} = 1\}} \\
\Sigma &= \frac{1}{n} \sum_{i=1}^{n} (x^{(i)} - \mu_{y^{(i)}})(x^{(i)} - \mu_{y^{(i)}})^T
\end{aligned}
$$

This are the optimized valued of these parameters, and we have

$$
\hat{y} = \begin{cases} 1 & \text{if } p(x \mid y = 1) \cdot \phi > p(x \mid y = 0) \cdot (1 - \phi) \\ 0 & \text{otherwise} \end{cases}
$$



Pictorially, what the algorithm is doing can be seen like this

Now let’s have a discussion about GDA and logistic regression.

The GDA model has an interesting relationship to logistic regression. If we view the quantity $p(y = 1|x; \phi, \mu_0, \mu_1, \Sigma)$ as a function of x, we’ll find that it can be expressed in the form

$$p(y=1|x;\phi,\mu_0,\mu_1,\Sigma)=\frac{1}{1+\exp(-\theta^T x)}$$

where θ is some appropriate function of $\phi, \Sigma, \mu_0, \mu_1$. 1 This is exactly the form that logistic regression—a discriminative algorithm—used to model $p(y=1|x)$.

So, which is better, or which do we prefer under certain circumstances? We just argued that if $p(x|y)$ is multivariate gaussian (with shared Σ), then $p(y|x)$ necessarily follows a logistic function. The converse, however, is not true; i.e., $p(y|x)$ being a logistic function does not imply $p(x|y)$ is multivariate gaussian. This shows that GDA makes stronger modeling assumptions about the data than does logistic regression. It turns out that when these modeling assumptions are correct, then GDA will find better fits to the data, and is a better model. Specifically, when $p(x|y)$ is indeed gaussian (with shared Σ), then GDA is *asymptotically efficient*. In this setting, GDA will be a better algorithm than logistic regression.

In contrast, by making significantly weaker assumptions, logistic regression is also more robust and less sensitive to incorrect modeling assumptions. There are many different sets of assumptions that would lead to $p(y|x)$ taking the form of a logistic function. For example, if $x|y=0 \sim \text{Poisson}(\lambda_0)$, and $x|y=1 \sim \text{Poisson}(\lambda_1)$, then $p(y|x)$ will be logistic. Logistic regression will also work well on Poisson data like this. But if we were to use GDA on such data—and fit Gaussian distributions to such non-Gaussian data—then the results will be less predictable, and GDA may (or may not) do well.

To summarize: GDA makes stronger modeling assumptions, and is more data efficient (i.e., requires less training data to learn “well”) when the modeling assumptions are correct or at least approximately correct. Logistic regression makes weaker assumptions, and is significantly more robust to deviations from modeling assumptions.

Naive bayes

In GDA, the feature vectors x were continuous, real-valued vectors. Let’s now talk about a different learning algorithm in which the x_j’s are discrete-valued.

We set up a situation where you want to tell if an email is a spam through it’s words. Classifying emails is one example of a broader set of problems called *text classification*.

We will represent an email via a feature vector whose length is equal to the number of words in the dictionary. Specifically, if an email contains the j-th word of the dictionary, then we

will set $x_j = 1$; otherwise, we let $x_j = 0$. The set of words encoded into the feature vector is called the vocabulary, so the dimension of x is equal to the size of the vocabulary.

Having chosen our feature vector, we now want to build a generative model. So, we have to model $p(x|y)$. But if we have, say, a vocabulary of 50000 words, then $x \in \{0,1\}^{50000}$ (x is a 50000-dimensional vector of 0's and 1's), and if we were to model x explicitly with a multinomial distribution over the 250000 possible outcomes, then we'd end up with a $(2^{50000} - 1)$-dimensional parameter vector. This is clearly too many parameters.

To model $p(x|y)$, we will therefore make a very strong assumption. We will assume that the x_i's are conditionally independent given y. This assumption is called the *Naive Bayes (NB) assumption*, and the resulting algorithm is called the *Naive Bayes classifier*. This doesn't mean they are independent, that $p(x_1|x_2) = p(x_1)$, but rather $p(x_1|y) = p(x_1|y, x_2)$. Now we have

$$\begin{aligned} p(x_1, \ldots, x_{50000} \mid y) &= p(x_1 \mid y)p(x_2 \mid y, x_1)p(x_3 \mid y, x_1, x_2) \cdots p(x_{50000} \mid y, x_1, \ldots) \\ &= p(x_1 \mid y)p(x_2 \mid y)p(x_3 \mid y) \cdots p(x_{50000} \mid y) \\ &= \prod_{j=1}^{d} p(x_j \mid y) \end{aligned}$$

> 📌 Instead of using a full English dictionary, we typically encode only the words that appear at least once in our training set. This approach reduces computational and storage needs and allows us to include words that might appear in emails but not in a dictionary (e.g., "EML"). We also often exclude very frequent words (like "the," "of," "and", they're called "*stop words*") as they occur in many documents and offer little value in distinguishing between spam and non-spam emails.

Our model is parameterized by $\phi_{j|y=1} = p(x_j = 1|y = 1)$, $\phi_{j|y=0} = p(x_j = 1|y = 0)$, and $\phi_y = p(y = 1)$, and as we did in problem set 1, we maximize the likelihood function at

$$\phi_{j|y=1} = \frac{\sum_{i=1}^{n} 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 1\}}{\sum_{i=1}^{n} 1\{y^{(i)} = 1\}}$$
$$\phi_{j|y=0} = \frac{\sum_{i=1}^{n} 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 0\}}{\sum_{i=1}^{n} 1\{y^{(i)} = 0\}}$$
$$\phi_y = \frac{\sum_{i=1}^{n} 1\{y^{(i)} = 1\}}{n}$$

The parameters have a very natural interpretation. For instance, $\phi_j|y = 1$ is just the fraction of the spam ($y = 1$) emails in which word j does appear. Having fit all these parameters, to make a prediction on a new example with features x, we then simply calculate with Bayes's Law.

The generalization to where x_j can take values in $\{1, 2, ..., k_j\}$ is straightforward. Here, we would simply model $p(x_j|y)$ as multinomial rather than as Bernoulli. Indeed, even if some original input attribute (say, the living area of a house, as in our earlier example) were continuous valued, it is quite common to discretize it—that is, turn it into a small set of discrete values—and apply Naive Bayes.

If we want to apply NB to continuous data, we have to use GaussianNB

Laplace smoothing

Think about this: one day you received an email from ICML, which you have never received before, and the word ICML is something you have never seen in you dataset before. And could lead to a terrible results where

$$\phi_{35000|y=1} = \frac{\sum_{i=1}^{n} 1\{x_{35000}^{(i)} = 1 \wedge y^{(i)} = 1\}}{\sum_{i=1}^{n} 1\{y^{(i)} = 1\}} = 0$$
$$\phi_{35000|y=0} = \frac{\sum_{i=1}^{n} 1\{x_{35000}^{(i)} = 1 \wedge y^{(i)} = 0\}}{\sum_{i=1}^{n} 1\{y^{(i)} = 0\}} = 0$$

Hence, when trying to decide if one of these messages containing "ICML", it calculates the class posterior probabilities, and obtains

$$
\begin{aligned}
p(y = 1 \mid x) &= \frac{\prod_{j=1}^{d} p(x_j \mid y = 1) \cdot p(y = 1)}{\prod_{j=1}^{d} p(x_j \mid y = 1) \cdot p(y = 1) + \prod_{j=1}^{d} p(x_j \mid y = 0) \cdot p(y = 0)} \\
&= \frac{0}{0}
\end{aligned}
$$

Stating the problem more broadly, it is statistically a bad idea to estimate the probability of some event to be zero just because you haven't seen it before in your finite training set.

To avoid this, we can use *Laplace smoothing*, which replaces the above estimate with

$$
\phi_j = \frac{1 + \sum_{i=1}^{n} 1\{z^{(i)} = j\}}{k + n}
$$

Here, we've added 1 to the numerator, and k to the denominator. Note that $\sum_{j=1}^{k} \phi_j = 1$ still holds (check this yourself!), which is a desirable property. Under certain (arguably quite strong) conditions, it can be shown that the Laplace smoothing actually gives the optimal estimator of the ϕ_j's.

Returning to our Naive Bayes classifier, with Laplace smoothing, we therefore obtain the following estimates of the parameters:

$$
\begin{aligned}
\phi_{j|y=1} &= \frac{2 + \sum_{i=1}^{n} 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 1\}}{1 + \sum_{i=1}^{n} 1\{y^{(i)} = 1\}} \\
\phi_{j|y=0} &= \frac{2 + \sum_{i=1}^{n} 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 0\}}{1 + \sum_{i=1}^{n} 1\{y^{(i)} = 0\}}
\end{aligned}
$$

In practice, it usually doesn't matter much whether we apply Laplace smoothing to ϕ_y or not, since we will typically have a fair fraction each of spam and non-spam messages, so ϕ_y will be a reasonable estimate of $p(y = 1)$ and will be quite far from 0 anyway.

Event models for text classification

To close off our discussion of generative learning algorithms, let's talk about one more model that is specifically for text classification. While Naive Bayes as we've presented it will work well for many classification problems, for text classification, there is a related model that does even better.

In the specific context of text classification, Naive Bayes as presented uses the what's called the *Bernoulli event model* (or sometimes *multi-variate Bernoulli event model*). In this model, we assumed that the way an email is generated is that first it is randomly determined (according to the class priors $p(y)$) whether a spammer or non-spammer will send you your

next message. Then, the person sending the email runs through the dictionary, deciding whether to include each word j in that email independently and according to the probabilities $p(x_j = 1|y) = \phi_{j|y}$. Thus, the probability of a message was given by $p(y)\prod_{j=1}^{d} p(x_j \mid y)$.

Here's a different model, called the *Multinomial event model*. To describe this model, we will use a different notation and set of features for representing emails. We let x_j denote the identity of the j-th word in the email. Thus, x_j is now an integer taking values in $\{1, ..., |V|\}$, where $|V|$ is the size of our vocabulary (dictionary). An email of d words is now represented by a vector $(x_1, x_2, ..., x_d)$ of length d; note that d can vary for different documents. For instance, if an email starts with "A NeurIPS . . . ," then $x_1 = 1$ ("a" is the first word in the dictionary), and $x_2 = 35000$ (if "neurips" is the 35000-th word in the dictionary).

In the multinomial event model, we assume that the way an email is generated is via a random process in which spam/non-spam is first determined (according to $p(y)$) as before. Then, the sender of the email writes the email by first generating x_1 from some multinomial distribution over words ($p(x_1|y)$). Next, the second word x_2 is chosen independently of x_1 but from the same multinomial distribution, and similarly for x_3, x_4, and so on, until all d words of the email have been generated. Thus, the overall probability of a message is given by $p(y)\prod_{j=1}^{d} p(x_j \mid y))$. Note that this formula looks like the one we had earlier for the probability of a message under the Bernoulli event model, but that the terms in the formula now mean very different things. In particular $x_j|y$ is now a multinomial, rather than a Bernoulli distribution.

The parameters for our new model are $\phi_y = p(y)$ as before, $\phi_{k|y=1} = p(x_j = k|y = 1)$ (for any j) and $\phi_{k|y=0} = p(x_j = k|y = 0)$. Note that we have assumed that $p(x_j|y)$ is the same for all values of j (i.e., that the distribution according to which a word is generated does not depend on its position j within the email).

Maximizing this yields the maximum likelihood estimates of the parameters:

$$\begin{aligned}
\phi_{k|y=1} &= \frac{\sum_{i=1}^{n}\sum_{j=1}^{d_i} 1\{x_j^{(i)} = k \wedge y^{(i)} = 1\}}{\sum_{i=1}^{n} 1\{y^{(i)} = 1\} \cdot d_i} \\
\phi_{k|y=0} &= \frac{\sum_{i=1}^{n}\sum_{j=1}^{d_i} 1\{x_j^{(i)} = k \wedge y^{(i)} = 0\}}{\sum_{i=1}^{n} 1\{y^{(i)} = 0\} \cdot d_i} \\
\phi_y &= \frac{\sum_{i=1}^{n} 1\{y^{(i)} = 1\}}{n}
\end{aligned}$$

we were to apply Laplace smoothing (which is needed in practice for good performance) when estimating $\phi_{k|y=0}$ and $\phi_{k|y=1}$, we add 1 to the numerators and $|V|$ to the denominators.

Kernel method

Feature maps

Think about a polynomial function: $y = \theta_3 x^3 + \theta_2 x^2 + \theta_1 x + \theta_0$, this is kind complicated for us to fit. So can we make it simple? Of course, we can define a function $\phi(x) : \mathbb{R} \to \mathbb{R}^4$ which is

$$\phi(x) = \begin{bmatrix} 1 \\ x \\ x^2 \\ x^3 \end{bmatrix} \in \mathbb{R}^4$$

So now the original function is now

$$\theta_3 x^3 + \theta_2 x^2 + \theta_1 x + \theta_0 = \theta^T \phi(x)$$

We call the original input value *the input attributes* of a problem (in this case, x). And we call the $\phi(x)$, *the features variables*. We will call ϕ a *feature map*, which maps the attributes to the features.

LMS (least mean squares) with features

We will derive the gradient descent algorithm for fitting the model $\theta^T \phi(x)$. The original batch update law was

$$\theta := \theta + \alpha \sum_{i=1}^{n} \left(y^{(i)} - \theta^T x^{(i)} \right) x^{(i)}$$

Let $\phi : \mathbb{R}^d \to \mathbb{R}^p$ be a feature map that maps attribute x (in $\mathbb{R}^d$) to the features $\phi(x)$ in $\mathbb{R}^d$, we can them replace every occurrence of $x^{(i)}$ with feature $\phi(x)$, so now the rule is

$$\theta := \theta + \alpha \sum_{i=1}^{n} \left(y^{(i)} - \theta^T \phi(x^{(i)}) \right) \phi(x^{(i)})$$

Similarly, the corresponding stochastic gradient descent update rule is

$$\theta := \theta + \alpha \left(y^{(i)} - \theta^T \phi(x^{(i)}) \right) \phi(x^{(i)})$$

LMS with the kernel trick

Let's think about this, if we have, say d input attributes, and d is a rather huge number (like 1000). Obviously, it's of no hard for us to do a simple linear regression, but what if an other multinomial function, say a $\phi(x)$ that contains every monomials of x with degree ≤ 3

$$\phi(x) = \begin{bmatrix} 1 & x_1 & x_2 & \cdots & x_1^2 & x_1 x_2 & x_1 x_3 & \cdots & x_2 x_1 & \cdots & x_3^2 & \cdots & x_2^2 & \cdots \end{bmatrix}^T$$

that could be a problem, because now we're facing a vector with over d^3 entries (there are totally $1 + d + d^2 + d^3$entries), that's too computable costly. So we try to find a better way to solve this.

We will introduce the kernel trick with which we will not need to store θ explicitly, and the runtime can be significantly improved. The main observation is that at any time, θ can be represented as a linear combination of the vectors $\phi(x^{(1)}), ..., \phi(x^{(n)})$. Assume that at any point, θ can be expressed as

$$\theta = \sum_{i=1}^{n} \beta_i \phi(x^{(i)})$$

Then we claim that in the next round, θ is still a linear combination of $\phi(x^{(1)}), ..., \phi(x^{(n)})$ because

$$
\begin{aligned}
\theta &:= \theta + \alpha \sum_{i=1}^{n} \left(y^{(i)} - \theta^T \phi(x^{(i)}) \right) \phi(x^{(i)}) \\
&= \sum_{i=1}^{n} \beta_i \phi(x^{(i)}) + \alpha \sum_{i=1}^{n} \left(y^{(i)} - \theta^T \phi(x^{(i)}) \right) \phi(x^{(i)}) \\
&= \sum_{i=1}^{n} \left(\beta_i + \alpha \left(y^{(i)} - \theta^T \phi(x^{(i)}) \right) \right) \phi(x^{(i)})
\end{aligned}
$$

Using the equation above, we see that the new β_i depends on the old one via

$$
\beta_i := \beta_i + \alpha \left(y^{(i)} - \theta^T \phi(x^{(i)}) \right)
$$

replace the θ here we have

$$
\beta_i := \beta_i + \alpha \left(y^{(i)} - \sum_{j=1}^{n} \beta_j \phi(x^{(j)})^T \phi(x^{(i)}) \right)
$$

So we can easily see that $\phi(x^{(j)})^T \phi(x^{(i)})$ here is something we can calculate before we get in to the training loop. We often rewrite $\phi(x^{(j)})^T \phi(x^{(i)})$ as $\langle \phi(x^{(j)}), \phi(x^{(i)}) \rangle$to emphasize that it's the inner product of the two feature vectors. And there's another interesting property for some of the special feature maps that is the inner product could be applied to it directly. For example, the feature map we mentioned in the first subsection, we have

$$
\begin{aligned}
\langle \phi(x), \phi(z) \rangle &= 1 + \sum_{i=1}^{d} x_i z_i + \sum_{i,j \in \{1,\dots,d\}} x_i x_j z_i z_j + \sum_{i,j,k \in \{1,\dots,d\}} x_i x_j x_k z_i z_j z_k \\
&= 1 + \langle x, z \rangle + \langle x, z \rangle^2 + \langle x, z \rangle^3
\end{aligned}
$$

As you will see, the inner products between the features $\langle \phi(x^{(i)}), \phi(x^{(j)}) \rangle$ are essential here. We define *the Kernel* corresponding to the feature map ϕ as a function that maps $\mathcal{X} \times \mathcal{X} \to \mathbb{R}$ satisfying:

$$
K(x, z) = \langle \phi(x), \phi(z) \rangle
$$

To wrap up the discussion, we write the down the final algorithm as follows:

1. Compute all the values $K(x^{(j)}, x^{(i)}) = \langle \phi(x^{(j)}), \phi(x^{(i)}) \rangle$ using equation for all $i, j \in \{1, ..., n\}$. Set $\beta := 0$.

2. Then we follow the loop

$$\forall i \in \{1, \ldots, n\}, \quad \beta_i := \beta_i + \alpha \left(y^{(i)} - \sum_{j=1}^{n} \beta_j K(x^{(i)}, x^{(j)}) \right)$$

Or in vector notation, letting K be the $n \times n$ matrix with $K_{ij} = \langle \phi(x^{(i)}), \phi(x^{(j)}) \rangle$, we have

$$\beta := \beta + \alpha(\vec{y} - K\beta)$$

And in the end, we need to check our predict law, which would be denote as

$$\theta^T \phi(x) = \sum_{i=1}^{n} \beta_i K(x^{(i)}, x)$$

Properties of kernels

We saw that the kernel function is so intrinsic so that as long as the kernel function is defined, the whole training algorithm can be written entirely in the language of the kernel without referring to the feature map, so can the prediction of a test example x. That' s tempting, because we only need to make sure there exists a map $\phi(x)$ and we don't even need to specify it.

What kinds of functions $K(\cdot, \cdot)$ can correspond to some feature map ϕ ? In other words, can we tell if there is some feature mapping ϕ so that $K(x, z) = \phi(x)^T \phi(z)$ for all x, z?

Before we get right into it, let's go through a few more things.

Suppose $x, z \in R^d$, and let's first consider the function $K(\cdot,\cdot)$ defined as: $K(x, z) = (x^T z)^2$. We can also write this as

$$\begin{aligned} K(x, z) &= \left(\sum_{i=1}^{d} x_i z_i \right) \left(\sum_{j=1}^{d} x_j z_j \right) \\ &= \sum_{i=1}^{d} \sum_{j=1}^{d} x_i x_j z_i z_j \\ &= \sum_{i,j=1}^{d} (x_i x_j)(z_i z_j) \end{aligned}$$

Thus, we see that $K(x,z) = \langle \phi(x), \phi(z) \rangle$is the kernel function that corresponds to the the feature mapping ϕ given (shown here for the case of $d = 3$) by

$$\phi(x) = \begin{bmatrix} x_1 x_1 \\ x_1 x_2 \\ x_1 x_3 \\ x_2 x_1 \\ x_2 x_2 \\ x_2 x_3 \\ x_3 x_1 \\ x_3 x_2 \\ x_3 x_3 \end{bmatrix}$$

For another related example, also consider $K(\cdot,\cdot)$ defined by

$$\begin{aligned} K(x,z) &= \left(x^T z + c\right)^2 \\ &= \left(\sum_{i=1}^{d} x_i z_i + c\right)^2 \\ &= \sum_{i=1}^{d} \sum_{j=1}^{d} (x_i x_j)(z_i z_j) + 2c \sum_{i=1}^{d} x_i z_i + c^2 \\ &= \sum_{i,j=1}^{d} (x_i x_j)(z_i z_j) + 2c \sum_{i=1}^{d} x_i z_i + c^2 \end{aligned}$$

So we can note that the corresponding ϕ is given by

$$\phi(x) = \begin{bmatrix} c \\ \sqrt{2c} x_1 \\ \sqrt{2c} x_2 \\ \sqrt{2c} x_3 \\ x_1 x_2 \\ x_1 x_2 \\ \vdots \\ x_3 x_3 \end{bmatrix}$$

c here is a parameter that controls the weight between simple x and x square.

More broadly, the kernel $K(x,z) = (x^T z + c)^k$ corresponds to a feature mapping to an $\binom{d+k}{k}$ feature space, corresponding of all monomials of the form $x_{i1}, x_{i2}, ..., x_{ik}$ that are up to order k.

Now let's talk about using Kernels as similarity metrics. Intuitively, if $\phi(x)$ and $\phi(z)$ are close together, then we might expect $K(x,z) = \langle \phi(x), \phi(z) \rangle$ to be large, other wise it's small (say x and z are orthogonal to each other). The most common case is using *Gaussian kernel,* where

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

and corresponds to an infinite dimensional feature mapping ϕ. This is a reasonable measure of x and z's similarity, and is close to 1 when x and z are close, and near 0 when x and z are far apart.

At last, we give *the Mercer Theorem,*

Theorem (Mercer)

Let $K : R^d \times R^d \to R$ be given. Then for K to be a valid (Mercer) kernel, it is necessary and sufficient that for any $x^{(1)}, ..., x^{(n)}, (n < \infty)$, the corresponding kernel matrix is symmetric positive semi-definite.

Support vector machines

Margins: the intuition

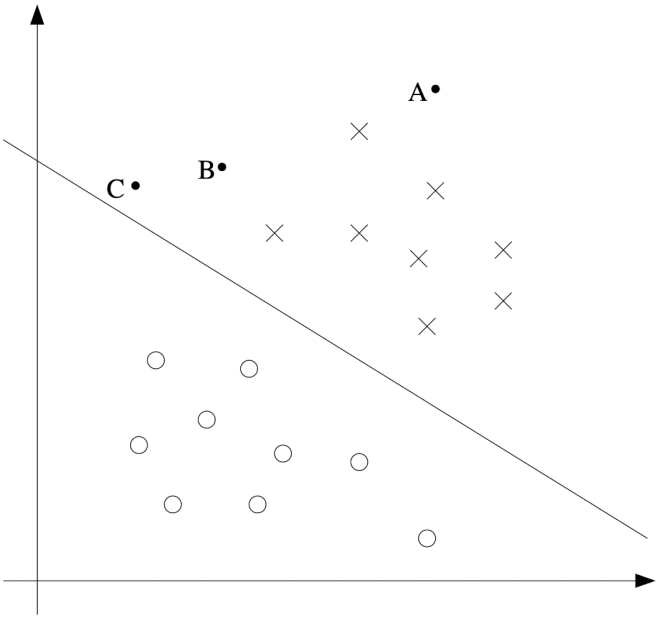
We'll start our story on SVMs by talking about margins. Our first question here is what we might think about before: How confident are we when make predictions? Take logistic regression as the example in this session. But right before we get into the topic we need to change our notations we used before. In this section, we consider $y \in \{-1,\ 1\}$ rather than $\{0, 1\},$ and also rather than parameterizing our linear classifier with the vector θ, we will use parameters w, b, and write our classifier as (obviously we drop the x_0 we had before)

$$h_{w,b}(x) = g(w^T x + b)$$

Note that here the the $g(z) = 1$ if $z \geq 0$, and $g(z) = -1$ otherwise, so our classifier will directly predict either 1 or−1 (cf. the perceptron algorithm), without first going through the intermediate step of estimating $p(y = 1)$ (which is what logistic regression does).

And now let's get back to the topic of margins. So as we know, in the logic regression, when $w^T x + b \geq 0$, we have our prediction as 1, and it's intuitive that if the $w^T x + b \gg 0$, of course we think that our prediction are more concrete, and we are for sure more confident about our prediction. This seems to be a nice goal to aim for, and we'll soon formalize this idea using the notion of functional margins.

For a different type of intuition, we have a figure to display



in which x's represent positive training examples, o's denote negative training examples

Here the decision boundary is given by the equation $w^T x + b = 0$, and is also called the *separating hyperplane*. Notice that the point A is very far from the decision boundary. If we are asked to make a prediction for the value of y at A, it seems we should be quite confident that $y = 1$ there. Conversely, the point C is very close to the decision boundary, and while it's on the side of the decision boundary on which we would predict $y = 1$, it seems likely that just a small change to the decision boundary could easily have caused out prediction to be $y = 0$. Hence, we're much more confident about our prediction at A than at C.

Again, informally we think it would be nice if, given a training set, we manage to find a decision boundary that allows us to make all correct and confident (meaning far from the decision boundary) predictions on the training examples. We'll formalize this later using the notion of geometric margins.

Functional and geometric margins

Let's formalize the notions of the functional and geometric margins. Given a training example $(x^{(i)}, y^{(i)})$, we define the *functional margin* of (w, b) with respect to the training example as

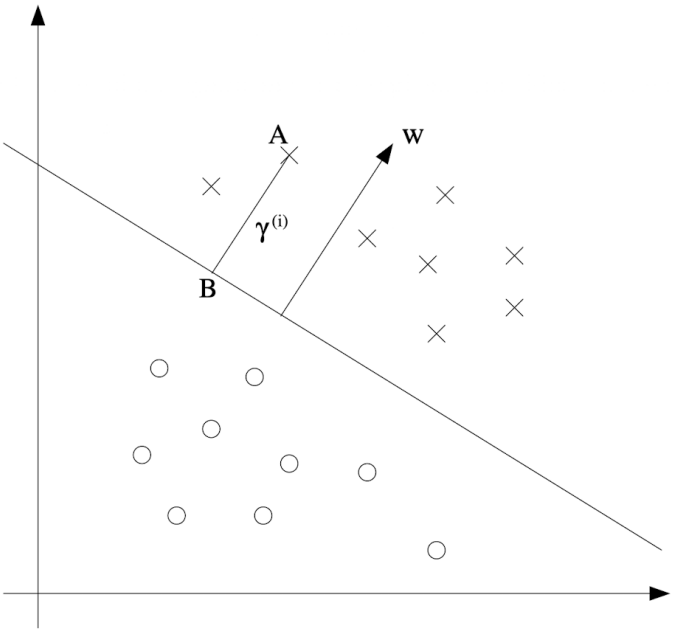
$$\widehat{\gamma}^{(i)} = y^{(i)}(w^T x^{(i)} + b)$$

This definition have some good property. Note that given $y \in \{-1,\ 1\}$, and the property of our $g(z)$, our predictions are only valid only when the functional margin are greater then 0. When $y = 1$, for sure when the $w^T x^{(i)} + b$ is a large positive number, we are confident. and the same when $y = -1$, and we find it easily that at this moment we have a rather large functional margin, and that is natural.

Yet a terrible things about functional margin is its magnitude. We will illustrate this right now. Of course, if we predict $y = 1$ for a certain data pair, but what if we multiply our w and b to 2, obviously the prediction will still be 1, and we're not statistically more confident, while our functional margin is now two times bigger! That' s confusing. Inspired by this, we decide to normalize this. Before we go, we define the functional margin of a dataset

$$\widehat{\gamma} = \underset{i=1,\ldots,n}{min}\ \widehat{\gamma}^{(i)}$$

Now let's talk about *geometric margins*.



The decision boundary corresponding to (w, b) is shown, along with the vector w. Note that w is orthogonal (at 90°) to the separating hyperplane. (You should convince yourself that this must be the case.)

Consider the point at A, which represents the input $x^{(i)}$ of some training example with label $y^{(i)} = 1$. Its distance to the decision boundary, $\gamma^{(i)}$, is given by the line segment AB. How can we find the value of $\gamma^{(i)}$? Well, $w/||w||$ is a unit-length vector pointing in the same direction as w. Since A represents $x^{(i)}$, we therefore find that the point B is given by $x^{(i)} - \gamma^{(i)} \cdot w/||w||$. But this point

lies on the decision boundary, and all points x on the decision boundary satisfy the equation $w^T x + b = 0$. Hence,

$$w^T \quad (x^{(i)} - \gamma^{(i)} \frac{w}{||w||}) \quad + b = 0$$

Solving for $\gamma^{(i)}$ yields

$$\gamma^{(i)} = \left(\frac{w}{||w||} \right)^T x^{(i)} + \frac{b}{||w||}$$

This was worked out for the case of a positive training example at A in the figure, where being on the "positive" side of the decision boundary is good. More generally, we define the geometric margin of (w, b) with respect to a training example $(x^{(i)}, y^{(i)})$ to be

$$\gamma^{(i)} = y^{(i)} \left(\left(\frac{w}{||w||} \right)^T x^{(i)} + \frac{b}{||w||} \right)$$

Note that if $||w|| = 1$, then the functional margin equals the geometric margin—this thus gives us a way of relating these two different notions of margin. Also, the geometric margin is invariant to rescaling of the parameters; i.e., if we replace w with $2w$ and b with $2b$, then the geometric margin does not change. Specifically, because of this invariance to the scaling of the parameters, when trying to fit wand b to training data, we can impose an arbitrary scaling constraint on w without changing anything important; for instance, we can demand that $||w|| = 1$ or something else.

Also we define the geometric margin of a dataset to be

$$\gamma = \underset{i=1,...,n}{min} \gamma^{(i)}$$

The optimal margin classifier

It's the natural desideratum to find a decision boundary that can separates our dataset, however to make ourselves more confident, we try to maximize our margin at the same time. How will we find the one that achieves the maximum geometric margin? We can pose the following optimization problem:

$$\begin{aligned} max_{\gamma,w,b} \quad & \gamma \\ & s.t. \quad y^{(i)}(w^T x^{(i)} + b) \geq \gamma, \quad i = 1, ..., n \\ & ||w|| = 1 \end{aligned}$$

I.e., we want to maximize γ, subject to each training example having functional margin at least γ. The $||w|| = 1$ constraint moreover ensures that the functional margin equals to the geometric margin, so we are also guaranteed that all the geometric margins are at least γ. Thus, solving this

problem will result in (w, b) with the largest possible geometric margin with respect to the training set.

But the problem here is that it's hard to solve with the constraint $||w|| = 1$, so we try to replace it with another one. Consider using the following question

$$\begin{aligned} max_{\widehat{\gamma},w,b} \quad & \frac{\widehat{\gamma}}{||w||} \\ s.t. \quad & y^{(i)}(w^T x^{(i)} + b) \geq \widehat{\gamma}, \quad i = 1, ..., n \end{aligned}$$

Here, we're going to maximize $\widehat{\gamma}/||w||$, subject to the functional margins all being at least $\widehat{\gamma}$. Since the functional margin is connected to geometric margin though the equation $\gamma = \widehat{\gamma}/||w||$, this of course is the answer we are in search of. However, $\widehat{\gamma}/||w||$ is still something unpleasant, but we already know that the functional margin can be rescaled without making any difference, We will introduce the scaling constraint that the functional margin of w, b with respect to the training set must be 1: $\gamma = 1$. So now our optimal question become

$$\begin{aligned} max_{w,b} \quad & \frac{1}{||w||^2} \\ s.t. \quad & y^{(i)}(w^T x^{(i)} + b) \geq 1, \quad i = 1, ..., n \end{aligned}$$

And this is something with a convex quadratic objective and only linear constraints. Its solution gives us the *optimal margin classifier*. This optimization problem can be solved using commercial quadratic programming (QP) code.

> While we could call the problem solved here, what we will instead do is make a digression to talk about Lagrange duality. This will lead us to our optimization problem's dual form, which will play a key role in allowing us to use kernels to get optimal margin classifiers to work efficiently in very high dimensional spaces. The dual form will also allow us to derive an efficient algorithm for solving the above optimization problem that will typically do much better than generic QP software.

Lagrange duality

This subsection is mainly about the mathematic knowledge about constrained optimization problems, which you may learn in your convex optimization class.

Let's assume a general form of a constrained optimization problem where we have an equality constraint and an inequality constraint ,which we'll refer to as the primal optimization problem later

$$\begin{aligned} min_w \quad & f(w) \\ s.t. \quad & g_i(w) \leq 0, i = 1, ..., k \\ & h_i(w) = 0, i = 1, ..., l \end{aligned}$$

To solve it, we start by defining the generalized Lagrangian

$$L(w, \alpha, \beta) = f(w) + \sum_{i=1}^{k} \alpha_i g_i(w) + \sum_{i=1}^{l} \beta_i h_i(w)$$

Consider the quantity

$$\theta_{\mathcal{P}}(w) = \max_{\alpha, \beta : \alpha_i \geq 0} L(w, \alpha, \beta)$$

Here, the "P" subscript stands for "primal." Let some w be given. If w violates any of the primal constraints (i.e., if either $g_i(w) > 0$ or $h_i(w) \neq 0$ for some i), then you should be able to verify that

$$\theta_{\mathcal{P}}(w) = \infty$$

Conversely, if the constraints are indeed satisfied for a particular value of w, then $\theta_{\mathcal{P}}(w) = f(w)$.

Thus, $\theta_{\mathcal{P}}$ takes the same value as the objective in our problem for all values of w that satisfies the primal constraints, and is positive infinity if the constraints are violated. Hence, if we consider the minimization problem

$$\min_w \theta_{\mathcal{P}}(w) = \min_w \max_{\alpha, \beta : \alpha_i \geq 0} L(w, \alpha, \beta) = \min_w f(w)$$

we see that it is the same problem (i.e., and has the same solutions as) our original, primal problem. For later use, we also define the optimal value of the objective to be $p^* = \min_w \theta_{\mathcal{P}}(w)$; we call this the value of the primal problem.

Now let's consider the dual problem of this. We denote

$$\theta_{\mathcal{D}}(\alpha, \beta) = \min_w L(w, \alpha, \beta)$$

Here, the "D" subscript stands for "dual." Note also that whereas in the definition of $\theta_{\mathcal{P}}$ we were optimizing (maximizing) with respect to α, β, here we are minimizing with respect to w. Now we can pose the dual optimization problem

$$\max_{\alpha,\beta:\alpha_i\geq 0} \theta_{\mathcal{D}}(\alpha,\beta) = \max_{\alpha,\beta:\alpha_i\geq 0} \min_{w} L(w,\alpha,\beta)$$

We also define the optimal value of the dual problem's objective to be $d^* = \max_{\alpha,\beta:\alpha_i\geq 0} \theta_{\mathcal{D}}(\alpha,\beta)$.

So how can this two value be related together, we introduce the *Lagrange duality* and we have

$$d* = \max_{\alpha,\beta:\alpha_i\geq 0} \min_{w} L(w,\alpha,\beta) \leq \min_{w} \max_{\alpha,\beta:\alpha_i\geq 0} L(w,\alpha,\beta) = p*$$

However, under certain conditions, we will have

$$d^* = p^*,$$

so that we can solve the dual problem in lieu of the primal problem. Let's see what these conditions are.

Suppose f and the g_i's are convex, and the h_i's are affine. Suppose further that the constraints g_i are (strictly) feasible; this means that there exists some w so that $g_i(w) < 0$ for all i.

Under our above assumptions, there must exist w^*, α^*, β^* so that w* is the solution to the primal problem, α^*, β^* are the solution to the dual problem, and moreover $p^* = d^* = L(w^*, \alpha^*, \beta^*)$. Moreover, w^*, α^* and β^* satisfy *the Karush-Kuhn-Tucker (KKT) conditions*, which are as follows:

$$\frac{\partial}{\partial w_i} L(w^*, \alpha^*, \beta^*) = 0, \quad i = 1, \ldots, d \quad (6.3)$$

$$\frac{\partial}{\partial \alpha_i} L(w^*, \alpha^*, \beta^*) = 0, \quad i = 1, \ldots, l \quad (6.4)$$

$$\alpha_i^* g_i(w^*) = 0, \quad i = 1, \ldots, k \quad (6.5)$$

$$g_i(w^*) \leq 0, \quad i = 1, \ldots, k \quad (6.6)$$

$$\alpha_i^* \geq 0, \quad i = 1, \ldots, k \quad (6.7)$$

We draw attention to Equation (6.5), which is called the *KKT dual complementarity condition*. Specifically, it implies that if $\alpha_i^* > 0$, then $g_i(w^*) = 0$. (I.e., the "$g_i(w) \leq 0$" constraint is active, meaning it holds with equality rather than with inequality.) Later on, this will be key for showing that the SVM has only a small number of "support vectors"; the KKT dual complementarity condition will also give us our convergence test when we talk about the SMO algorithm.

Optimal margin classifiers: the dual form

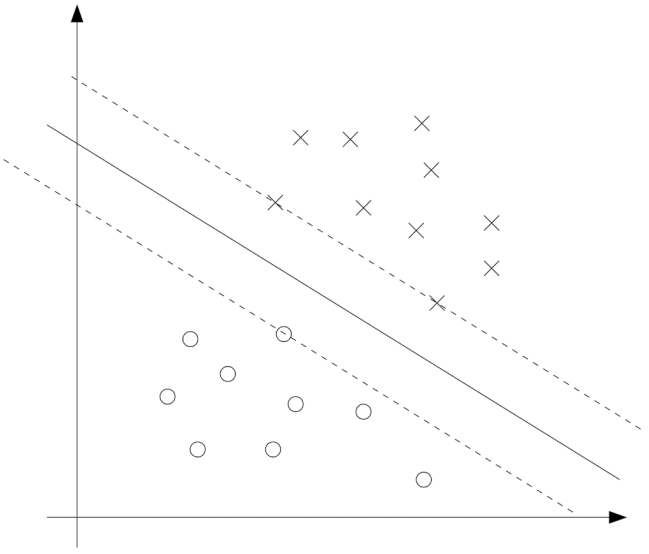
After the previous subsection, now let us draw another view on our optimal margin classifier. Previously, we posed the following (primal) optimization problem for finding the optimal margin classifier:

$$max_{w,b} \quad \frac{1}{||w||^2} \tag{6.8}$$
$$s.t. \quad y^{(i)}(w^T x^{(i)} + b) \geq 1, \quad i = 1, ..., n$$

Rewrite our constraint as

$$g_i(w) = -y^{(i)}(w^T x^{(i)} + b) + 1 \leq 0$$

We have one such constraint for each training example. Note that from the KKT dual complementarity condition, we will have $\alpha_i > 0$ only for the training examples that have functional margin exactly equal to one (i.e., the ones corresponding to constraints that hold with equality, $g_i(w) = 0$). Consider the figure below, in which a maximum margin separating hyperplane is shown by the solid line.



As we can tell from our figure, the points with the smallest margins are exactly the ones closest to the decision boundary; here, these are the three points (one negative and two positive examples) that lie on the dashed lines parallel to the decision boundary. Thus, only three of the α_i's—namely, the ones corresponding to these three training examples—will be non-zero at the optimal solution to our optimization problem. These three points are called the *support vectors* in this problem. Of course this is the case, after all only these three points will significantly effect the ultimate hyperplane.

And we can now construct our Lagrangian for our optimization problem here

$$L(w, b, \alpha) = \frac{1}{2}||w||^2 - \sum_{i=1}^{n} \alpha_i [y^{(i)}(w^T x^{(i)} + b) - 1] \tag{6.9}$$

Let's find the dual form of the problem. To do so, we need to first minimize $L(w, b, \alpha)$ with respect to w and b (for fixed α), to get $\theta_{\mathcal{D}}$, which we'll do by setting the derivatives of L with respect to w

and b to zero. We have:

$$\nabla_w L(w, b, \alpha) = w - \sum_{i=1}^{n} \alpha_i y^{(i)} x^{(i)} = 0$$

that is

$$w = \sum_{i=1}^{n} \alpha_i y^{(i)} x^{(i)} \qquad (6.10)$$

As for the derivative with respect to b, we obtain

$$\frac{\partial}{\partial b} L(w, b, \alpha) = \sum_{i=1}^{n} \alpha_i y^{(i)} = 0 \qquad (6.11)$$

If we take the definition of w in Equation (6.10) and plug that back into the Lagrangian (Equation 6.9), and simplify, we get

$$L(w, b, \alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} y^{(i)} y^{(j)} \alpha_i \alpha_j (x^{(i)})^T x^{(j)} - b \sum_{i=1}^{n} \alpha_i y^{(i)}$$

But from Equation (6.11), the last term must be zero, so we obtain

$$L(w, b, \alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} y^{(i)} y^{(j)} \alpha_i \alpha_j (x^{(i)})^T x^{(j)}$$

Recall that we got to the equation above by minimizing L with respect to w and b. Putting this together with the constraints $\alpha_i \geq 0$ (that we always had) and the constraint (6.11), we obtain the following dual optimization problem:

$$\begin{aligned} \max_{\alpha} \quad & W(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} y^{(i)} y^{(j)} \alpha_i \alpha_j \langle x^{(i)}, x^{(j)} \rangle \qquad (6.12) \\ \text{s.t.} \quad & \alpha_i \geq 0, \; i = 1, \ldots, n \\ & \sum_{i=1}^{n} \alpha_i y^{(i)} = 0 \end{aligned}$$

Note here we try to denote it in the form of the inner product of two vectors so we can apply kernel tricks later in our SMO methods.

We should also be able to verify that the conditions required for $p^* = d^*$ and the KKT conditions (Equations 6.3–6.7) to hold are indeed satisfied in this problem. Hence, we can solve the dual in lieu

of solving the primal problem. So now the optimization problem is about the α_i's, if we are able to solve this, we could then use equation 6.10 to work out the optimal w as the function of the the α, and work out the optimal value of b as well.

$$b^* = -\frac{\max_{i:y^{(i)}=-1} w^{*T} x^{(i)} + \min_{i:y^{(i)}=1} w^{*T} x^{(i)}}{2} \tag{6.13}$$

This optimal value of b is of extreme obviousness from a geometric point of view.

Before moving on, let's also take a more careful look at Equation (6.10), which gives the optimal value of w in terms of (the optimal value of) α. Suppose we've fit our model's parameters to a training set, and now wish to make a prediction at a new point input x. We would then calculate $w^T x + b$, and predict $y = 1$ if and only if this quantity is bigger than zero. But using (6.10), this quantity can also be written:

$$w^T x + b = \sum_{i=1}^{n} \alpha_i y^{(i)} x^{(i)T} x + b \tag{6.14}$$

$$= \sum_{i=1}^{n} \alpha_i y^{(i)} \langle x^{(i)}, x \rangle + b \tag{6.15}$$

And this is something dashing that depends only on the inner product between x and the points in the training set. Moreover, we saw earlier that the α_i's will all be zero except for the support vectors. Thus, many of the terms in the sum above will be zero, and we really need to find only the inner products between x and the support vectors (of which there is often only a small number) in order calculate (6.15) and make our prediction.

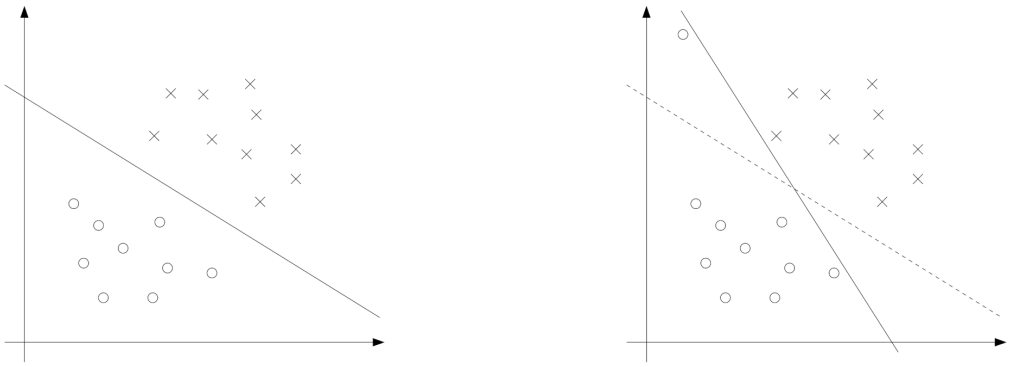
Let me make this point clear here:

In this problem, we define $g_i(w) = -y^{(i)}(w^T x^{(i)} + b) + 1 \leq 0$, and given the KKT conditions, we have $\alpha_i^* g_i(w^*) = 0$, that is to say, for any point with a margin bigger than 1 (which we set as a standard), then the $g_i(w)$ will be less than 0, thus the α_i^* of this point will be obviously 0, which indicates it share no effect on the hyperplane, or that is to say it by no means would be our support vector.

Regularization and the non-separable case

In the former subsection, we are always take care of the situation where we supposed that the margin would be at least 1, or that is to say the dataset is separable. While mapping data to a high dimensional feature space via φ does generally increase the likelihood that the data is separable, we can't guarantee that it always will be so. Also, in some cases it is not clear that finding a separating hyperplane is exactly what we'd want to do, since that might be susceptible to outliers.

For instance, the left figure below shows an optimal margin classifier, and when a single outlier is added in the upper-left region (right figure), it causes the decision boundary to make a dramatic swing, and the resulting classifier has a much smaller margin.



So we could use some regulation to avoid this, or even we could just ignore the outliers and let it be wrongly classified, we reformulate our optimization (using ℓ_1 regularization) as follows, by introducing *the slack variable* ξ:

$$\begin{aligned} \min_{\gamma,w,b} \quad & \frac{1}{2}||w||^2 + C\sum_{i=1}^{n}\xi_i \\ \text{s.t.} \quad & y^{(i)}(w^T x^{(i)} + b) \geq 1 - \xi_i, \quad i = 1, \ldots, n \\ & \xi_i \geq 0, \quad i = 1, \ldots, n. \end{aligned}$$

Here we are allowing the margin to be less than the original standard as 1, and it could be even less than 0, which indicates the misclassification! If an example has functional margin $1 - \xi_i$ (with $\xi > 0$), we would pay a cost of the objective function being increased by $C\xi_i$. The parameter C here control the balance between our fine and reward, if the C gets bigger, it means bigger fines and thus would make less mistake, which could lead to over-fitting; while when the C gets smaller, it means little fine we'll receive from each mistake and will thus make more mistake than before.

As before, we can form the Lagrangian:

$$L(w, b, \xi, \alpha, r) = \frac{1}{2}w^T w + C\sum_{i=1}^{n}\xi_i - \sum_{i=1}^{n}\alpha_i\left[y^{(i)}(w^T x^{(i)} + b) - 1 + \xi_i\right] - \sum_{i=1}^{n} r_i\xi_i$$

As before, we setting the derivatives with respect to w and b to zero as before, substituting them back in, and simplifying, we obtain the following dual form of the problem:

$$
\begin{aligned}
\max_{\alpha} \quad & W(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} y^{(i)} y^{(j)} \alpha_i \alpha_j \langle x^{(i)}, x^{(j)} \rangle \\
\text{s.t.} \quad & 0 \leq \alpha_i \leq C, \; i = 1, \ldots, n \\
& \sum_{i=1}^{n} \alpha_i y^{(i)} = 0
\end{aligned}
$$

Note that, somewhat surprisingly, in adding ℓ_1 regularization, the only change to the dual problem is that what was originally a constraint that $0 \leq \alpha_i$ has now become $0 \leq \alpha_i \leq C$. The calculation for b^* also has to be modified (Equation 6.13 is no longer valid); see the comments in the next section/Platt's paper.

Also, the KKT dual-complementarity conditions (which in the next section will be useful for testing for the convergence of the SMO algorithm) are:

$$
\begin{aligned}
\alpha_i = 0 &\implies y^{(i)}(w^T x^{(i)} + b) \geq 1 \quad (6.16) \\
\alpha_i = C &\implies y^{(i)}(w^T x^{(i)} + b) \leq 1 \quad (6.17) \\
0 < \alpha_i < C &\implies y^{(i)}(w^T x^{(i)} + b) = 1 \quad (6.18)
\end{aligned}
$$

Explanation

If $\alpha_i = 0$, indicates that it's not a support vector where $y^{(i)}(w^T x^{(i)} + b) \geq 1$

If $\alpha_i = C$, the sample is a support vector go beyond the margin. This implies that the model is penalized for classification errors on this sample, and it plays a critical role in defining the decision boundary. The constraint satisfied in this case is: $y^{(i)}(w^T x^{(i)} + b) \leq 1$

If $0 < \alpha_i < C$: The sample i is a support vector that lies exactly on the margin. This means the sample is correctly classified but is close to the decision boundary. These samples are crucial for determining the final position of the decision boundary. The constraint satisfied in this case is: $y^{(i)}(w^T x^{(i)} + b) = 1$

The SMO algorithm

After all these, we still can't find a way to optimize our α's and so on, and now we're proposing a grounded method in working out our problem.

Coordinate ascent

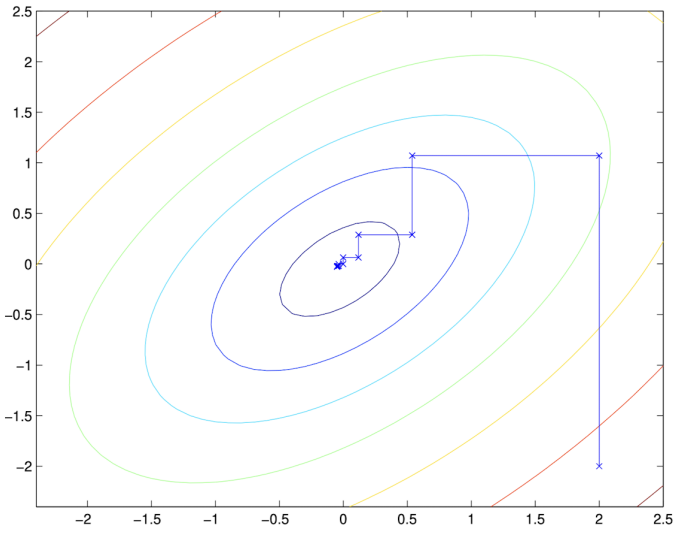
Consider trying to solve the unconstrained optimization problem

$$
\max_{\alpha} \quad W(\alpha_1, \alpha_2, ..., \alpha_n)
$$

What we can really do is try this till convergence

$$
\begin{aligned}
&\textit{Loop until convergence} : \{ \\
&\quad \text{For } i = 1, \ldots, n, \{ \\
&\qquad \alpha_i := \arg\max_{\widehat{\alpha}_i} W(\alpha_1, \ldots, \alpha_{i-1}, \widehat{\alpha}_i, \alpha_{i+1}, \ldots, \alpha_n) \\
&\qquad \} \\
&\quad \}
\end{aligned}
$$

Thus, in the innermost loop of this algorithm, we will hold all the variables except for some αi fixed, and reoptimize W with respect to just the parameter α_i. In the version of this method presented here, the inner-loop reoptimizes the variables in order $\alpha_1, \alpha_2, ..., \alpha_n, \alpha_1, \alpha_2, ...$ (A more sophisticated version might choose other orderings; for instance, we may choose the next variable to update according to which one we expect to allow us to make the largest increase in $W(\alpha)$.)



A picture of how coordinate ascent works

SMO

We close off the discussion of SVMs by sketching the derivation of the SMO algorithm.

Recall our problem here

$$\max_{\alpha} \quad W(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} y^{(i)} y^{(j)} \alpha_i \alpha_j \langle x^{(i)}, x^{(j)} \rangle \tag{6.19}$$

$$\text{s.t.} \quad 0 \leq \alpha_i \leq C, \quad i = 1, \ldots, n \tag{6.20}$$

$$\sum_{i=1}^{n} \alpha_i y^{(i)} = 0 \tag{6.19}$$

So how can we now make a progress, first we what we need to be clear about is that we can't just vary one α_i here because the constraint (6.21) ensures that

$$\alpha_1 y^{(1)} = -\sum_{i=2}^{n} \alpha_i y^{(i)}$$

Hence, α_1 is exactly determined by the other αi's, and if we were to hold $\alpha_2, \ldots, \alpha_n$ fixed, then we can't make any change to α_1 without violating the constraint (6.21) in the optimization problem. Thus, if we want to update some subject of the α_i's, we must update at least two of them simultaneously in order to keep satisfying the constraints.

So we can follow this way

Repeat till convergence {

1. Select some pair α_i and α_j to update next (using a heuristic that tries to pick the two that will allow us to make the biggest progress towards the global maximum).
2. Reoptimize $W(\alpha)$ with respect to α_i and α_j, while holding all the other α_k's $(k \neq i, j)$ fixed.

}

To test for convergence of this algorithm, we can check whether the KKT conditions (Equations 6.16-6.18) are satisfied to within some *tol*. Here, *tol* is the *convergence tolerance parameter*, and is typically set to around 0.01 to 0.001. (See the paper and pseudocode for details.)

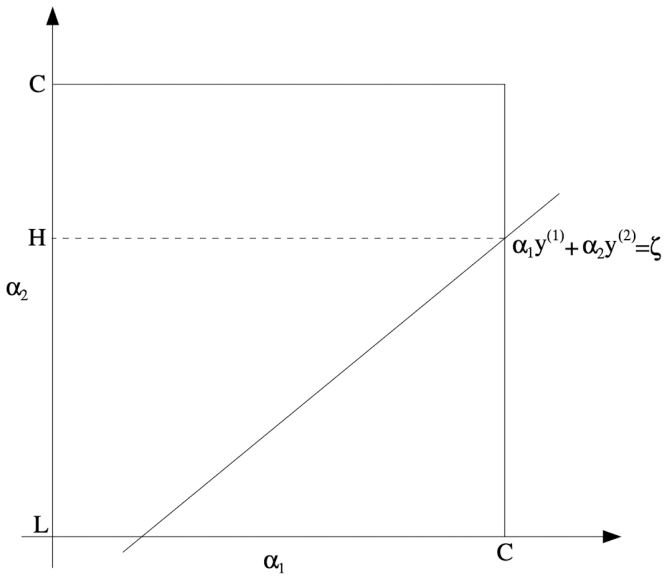
Let's now briefly sketch the main ideas for deriving the efficient update. Let's say we currently have some setting of the α_i's that satisfy the constraints (6.20-6.21), and suppose we've decided to hold $\alpha_3, \ldots, \alpha_n$ fixed, and want to reoptimize W($\alpha_1, \alpha_2, \ldots, \alpha_n$) with respect to α1 and α2 (subject to the constraints). From (6.21), we require that

$$\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = -\sum_{i=3}^{n} \alpha_i y^{(i)}$$

Since the the right hand are fixed, so we can denote it as

$$\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = \zeta \qquad (6.22)$$

We can thus picture the constraints on α_1 and α_2 as follows:



From the constraints (6.20), we know that α_1 and α_2 must lie within the box $[0, C] \times [0, C]$ shown. Note also that, from these constraints, we know $L \leq \alpha_2 \leq H$. Using Equation (6.22), we can also write α_1 as a function of α_2:

$$\alpha_1 = (\zeta - \alpha_2 y^{(2)}) y^{(1)}$$

(Check this derivation yourself; we again used the fact that $y^{(1)} \in \{-1, 1\}$ so that $(y^{(1)})^2 = 1$) Hence, the objective $W(\alpha)$ can be written

$$W(\alpha_1, \alpha_2, ..., \alpha_n) = W((\zeta - \alpha_2 y^{(2)}) y^{(1)}, \alpha_2, ..., \alpha_n).$$

Treating $\alpha_3, ..., \alpha_n$ as constants, you should be able to verify that this is just some quadratic function in α_2. I.e., this can also be expressed in the form a $\alpha_2^2 + b\alpha_2 + c$ for some appropriate a, b, and c. If we ignore the "box" constraints (6.20) (or, equivalently, that $L \leq \alpha_2 \leq H$), then we can easily maximize this quadratic function by setting its derivative to zero and solving. We'll let $\alpha_2^{new,unclipped}$ denote the resulting value of α_2. We should also be able to convince yourself that if we had instead wanted to maximize W with respect to α_2 but subject to the box constraint, then we can find the resulting value optimal simply by taking $\alpha_2^{new,unclipped}$ and "clipping" it to lie in the $[L, H]$ interval, to get

$$\alpha_2^{\text{new}} = \begin{cases} H & \text{if } \alpha_2^{\text{new,unclipped}} > H, \\ \alpha_2^{\text{new,unclipped}} & \text{if } L \leq \alpha_2^{\text{new,unclipped}} \leq H, \\ L & \text{if } \alpha_2^{\text{new,unclipped}} < L. \end{cases}$$

Then we can use α_2 to work out α_1, and then to the other α's.

There’re a couple more details that are quite easy that we can find in Platt’s paper: One is the choice of the heuristics used to select the next α_i, α_j to update; the other is how to update b as the SMO algorithm is run.

Support vector regression

Support Vector Regression (SVR) is a regression technique based on Support Vector Machines (SVM). Generally, it’s quite similar to SVM, yet with a few little difference.

The objective of SVR could be put as:

$$\begin{aligned} \min_{w,b,\xi,\xi^*} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}(\xi_i + \xi_i^*) \\ s.t. \quad y_i - w^T x_i - b \leq \epsilon + \xi_i \\ w^T x_i + b - y_i \leq \epsilon + \xi_i^* \\ \xi_i, \xi_i^* \geq 0 \end{aligned}$$

Note here the slack variable on the both side can be different, thus we have both ξ_i and ξ_i^*.

Like before, construct Lagrangian and we get the dual form:

$$\begin{aligned} &\max_{\alpha,\alpha^*} -\frac{1}{2}\sum_{i,j=1}^{n}(\alpha_i - \alpha_i^*)(\alpha_j - \alpha_j^*)K(x_i, x_j) + \sum_{i=1}^{n}(\alpha_i - \alpha_i^*)y_i - \epsilon\sum_{i=1}^{n}(\alpha_i + \alpha_i^*) \\ &s.t. \quad 0 \leq \alpha_i, \alpha_i^* \leq C \\ &\qquad \sum_{i=1}^{n}(\alpha_i - \alpha_i^*) = 0 \end{aligned}$$

$K(x_i, x_j)$ is the kernel function, α and α^* are Lagrange multipliers. Only a subset of these values will be non-zero, corresponding to the support vectors, which define the regression function.

Once the model is trained, the predicted function $f(x)$ for a new data point x can be computed as:

$$f(x) = \sum_{i=1}^{n}(\alpha_i - \alpha_i^*)K(x_i, x) + b$$

the bias term b is calculated using the support vectors to ensure predictions meet the margin constraints. Once the optimal Lagrange multipliers (α and α^*) are determined, b is computed by taking any support vector (a point on the margin boundary) and substituting it back into the equation:

$$b = y_i - \sum_{j=1}^{n}(\alpha_j - \alpha_j^*)\langle x_i, x_j \rangle$$

This ensures that the prediction $f(x) = w^T x + b$ aligns with the margin for support vectors within the ϵ-tube.

set all $\alpha_i = 0$ $b = 0$

choose two α_1, α_2 that violates KKT condition

that is to say $\begin{cases} \alpha = 0 & \text{and } y^{(i)} f(x^{(i)}) - 1 < 0 \\ \alpha = C & \text{and } y^{(i)} f(x^{(i)}) - 1 > 0 \\ 0 < \alpha < C & \text{and } y^{(i)} f(x^{(i)}) - 1 \neq 0 \end{cases}$ note that we can use $y^{(i)2} = 1$

the $y^{(i)} f(x^{(i)}) - 1 = y^{(i)} f(x^{(i)}) - y^{(i)2} = y^{(i)} \epsilon_i$ $\quad \epsilon_i = f(x^{(i)}) - y^{(i)}$

so we have $\begin{cases} \alpha = 0 & \text{and } y^{(i)} \epsilon_i < 0 \\ \alpha = C & \text{and } y^{(i)} \epsilon_i > 0 \\ 0 < \alpha < C & \text{and } y^{(i)} \epsilon_i \neq 0 \end{cases}$ would be the conditions which violate KKT.

now for α_1 and α_2 we know that $\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = -\sum_{t=3}^{m} \alpha_t y^{(t)} = \zeta$ are fixed

so $\alpha_2 = (\zeta - \alpha_1 y^{(1)}) y^{(2)}$ remember that $W(\alpha) = \sum \alpha_i - \frac{1}{2} \sum\sum y^{(i)} y^{(j)} \alpha_i \alpha_j \langle x^{(i)}, x^{(j)} \rangle$

denote $\langle x^{(i)}, x^{(j)} \rangle$ as K_{ij} ; $\sum_{i=3}^{m} y^{(i)} \alpha_i K_{ti} = V_t$

$$W(\alpha_1, \alpha_2) = \alpha_1 + \alpha_2 - \frac{1}{2}\alpha_1^2 K_{11} - \frac{1}{2}\alpha_2^2 K_{22} - \alpha_1 \alpha_2 K_{12} y^{(1)} y^{(2)} - \frac{1}{2}\alpha_1 y^{(1)} V_1 - \frac{1}{2}\alpha_2 y^{(2)} V_2 + C$$

$$\alpha_2 = y^{(2)}(\zeta - \alpha_1 y^{(1)}) \Rightarrow = \alpha_1 - y^{(1)} y^{(2)} \alpha_1 - \frac{1}{2} K_{11} \alpha_1^2 - \frac{1}{2} K_{22}(-2 y^{(1)} \zeta \alpha_1 + \alpha_1^2) - K_{12}(\zeta - \alpha_1 y^{(1)}) y^{(2)} \alpha_1 y^{(1)} y^{(2)}$$
$$- \frac{1}{2}\alpha_1 y^{(1)} V_1 - \frac{1}{2}(\zeta - \alpha_1 y^{(1)}) V_2 + C$$

remember that in constraint we have $\omega = \sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)} \Rightarrow f(x^{(j)}) = \omega x^{(j)} + b = \sum_{i=1}^{m} K_{ij} \alpha_i y^{(i)} + b$

so $f(x^{(1)}) = \sum_{i=1}^{m} \alpha_i y^{(i)} K_{1i} + b = \alpha_1 y^{(1)} K_{11} + \alpha_2 y^{(2)} K_{12} + V_1 + b$ same $f(x^{(2)}) = \alpha_1 y^{(1)} K_{12} + \alpha_2 y^{(2)} K_{22} + V_2 + b$

$$V_1 = f(x^{(1)}) - b - \alpha_1 y^{(1)} K_{11} - (\zeta - \alpha_1 y^{(1)}) K_{12} = f(x^{(1)}) - b - \zeta K_{12} + \alpha_1 y^{(1)}(K_{12} - K_{11})$$

$$V_2 = f(x^{(2)}) - b - \alpha_1 y^{(1)} K_{12} - (\zeta - \alpha_1 y^{(1)}) K_{22} = f(x^{(2)}) - b - \zeta K_{22} + \alpha_1 y^{(1)}(K_{22} - K_{12})$$

so $(1 - y^{(1)} y^{(2)} + \zeta y^{(1)} K_{22} - \zeta y^{(1)} K_{12}) \alpha_1 + \frac{1}{2}(2K_{12} - K_{11} - K_{22}) \alpha_1^2 - \frac{1}{2} y^{(1)} V_1 \alpha_1 + \frac{1}{2} y^{(1)} V_2 \alpha_1 + C$

$$\frac{\partial W}{\partial \alpha_1} = (1 - y^{(1)} y^{(2)} + \zeta y^{(1)} K_{22} - \zeta y^{(1)} K_{12}) + (2K_{12} - K_{11} - K_{22}) \alpha_1 - \frac{1}{2} y^{(1)} V_1 + \frac{1}{2} y^{(1)} V_2 = 0$$

$$(2K_{12} - K_{11} - K_{22}) \alpha_1^{new} = 1 - y^{(1)} y^{(2)} + \zeta y^{(1)} K_{22} - \zeta y^{(1)} K_{12} - \frac{1}{2} y^{(1)} (f(x^{(1)}) - f(x^{(2)}) - \zeta K_{12} + \zeta K_{22} + \alpha_1 y^{(1)} (2K_{12} - K_{11} - K_{22}))$$

$-\eta$

$$= 1 - y^{(1)} y^{(2)} - \frac{1}{2} \zeta y^{(1)} K_{12} + \frac{1}{2} \zeta y^{(1)} K_{22} - \frac{1}{2}(2K_{12} - K_{11} - K_{22}) \alpha_1 - \frac{1}{2} y^{(1)} f(x^{(1)}) + \frac{1}{2} y^{(1)} f(x^{(2)})$$

$$= 1 - y^{(1)} y^{(2)} - \frac{1}{2} K_{12} \alpha_1 - \frac{1}{2} y^{(1)} y^{(2)} K_{12} \alpha_2 + \frac{1}{2} K_{22} \alpha_1 + \frac{1}{2} y^{(1)} y^{(2)} K_{22} \alpha_2 - K_{12} \alpha_1 + \frac{1}{2} K_{11} \alpha_1 + \frac{1}{2} K_{22} \alpha_1 - \frac{1}{2} y^{(1)} f(x^{(1)}) + \frac{1}{2} y^{(1)} f(x^{(2)})$$

$\Rightarrow \alpha_1^{new} = \alpha_1 + \frac{y^{(1)}(\epsilon_2 - \epsilon_1)}{\eta}$ $\quad$ especially for b, we have

and $\alpha_2^{new} = y^{(2)}(\alpha_1 y^{(1)} + \alpha_2 y^{(2)} - \alpha_1^{new} y^{(1)})$

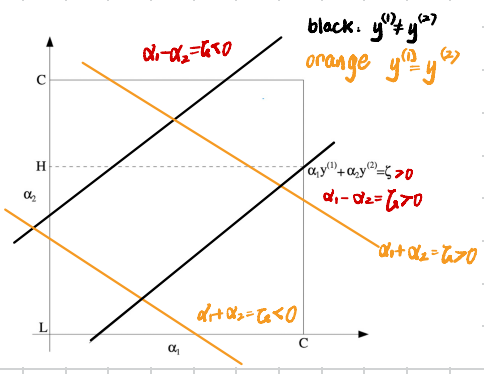
when $\alpha_1^{new} \in (0, C)$

$$b_1^{new} = b - \epsilon_1 - y^{(1)}(\alpha_1^{new} - \alpha_1) K_{11} - y^{(2)}(\alpha_2^{new} - \alpha_2) K_{12}$$

when $\alpha_2^{new} \in (0, C)$

$$b_2^{new} = b - \epsilon_2 - y^{(1)}(\alpha_1^{new} - \alpha_1) K_{12} - y^{(2)}(\alpha_2^{new} - \alpha_2) K_{22}$$

$b = \frac{b_1^{new} + b_2^{new}}{2}$ when both not or are



note that for α_1

when $y^{(1)} \neq y^{(2)}$

$L = \max(0, \underbrace{\alpha_1 - \alpha_2}_{\zeta})$

$H = \min(C, \alpha_1 - \alpha_2 + C)$

when $y^{(1)} = y^{(2)}$

$L = (0, \alpha_1 + \alpha_2 - C)$

$H = (C, \alpha_1 + \alpha_2)$

Deep Learning

Deep learning

Neural network is no doubt the most common and also the most mainstream learning algorithm in recent days, and we've already learned a lot about NN through our PyTorch tour and ISP tutorial, but we really learned a little about the mathematic theory and the precise definition about NN. So in this chapter, we gonna cover all the things we haven't met before. Yet still, this is just an overview.

Supervised learning with non-linear models

We will consider learning general family of models that are non-linear in both the parameters θ and the inputs x. The most common non-linear models are neural networks.

Cost/loss function

For this section, it suffices to think $h_\theta(x)$ as an abstract non-linear model. For simplicity, we start with the case where $y^{(i)} \in R$ and $h_\theta(x) \in R$.

As before, we define the cost/loss function as

$$J^{(i)}(\theta) = \frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2$$

this is of no difference with the the function we defined before. And we can define the loss function of the dataset as

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{n} J^{(i)}(\theta) \tag{1}$$

Optimizers (SGD)

Commonly, people use gradient descent (GD), stochastic gradient (SGD), or their variants to optimize the loss function J(θ). GD's update rule can be written as

$$\theta := \theta - \alpha \nabla_\theta J(\theta) \tag{2}$$

Here we give two different algorithms that we actually mentioned before.

Algorithm 1 Stochastic Gradient Descent

1: Hyperparameter: learning rate α, number of total iteration
n_{iter}.
2: Initialize θ randomly.
3: for i= 1 to
n_{iter} do
4: Sample j uniformly from {1,...,n}, and update θ by

$$\theta := \theta - \alpha \nabla_\theta J^{(j)}(\theta) \tag{2}$$

Oftentimes computing the gradient of B examples simultaneously for the parameter θ can be faster than computing B gradients separately due to hardware parallelization. So we give a brand new algorithm:

Algorithm 2 Mini-batch Stochastic Gradient Descent

1: Hyperparameters: learning rate α, batch size B, which is not equal to iterations n_{iter}.
2: Initialize θ randomly
3: for i= 1 to niter do
4: Sample B examples
$j_1, ..., j_B$ (without replacement) uniformly from {1,...,n}, and update θ by

$$\theta := \theta - \frac{\alpha}{B} \sum_{k=1}^{B} \nabla_\theta J^{(j_k)}(\theta) \tag{2}$$

Neural network

Neural networks refer to broad type of non-linear models/parametrizations $h_\theta(x)$ that involve combinations of matrix multiplications and other entry-wise non-linear operations. We will start small and slowly build up a neural network, step by step.

Neural network with only a single layer

Consider a model where we still try to make predictions on the housing price of California. $h_\theta(x)$ will still be our outcome as before, and as before we use a linear model, denoted as $w^T x + b$, while here we find a problem that the housing price can't be negative, which indicates that we need to drop every negative value, i.e. that there exists a "kink" in the figure. The "kink" we need here is something we can't handle simply with completely linear function, we need some simple functions to make a non-linear jump from the linear space (consider the gamma fix in mapping from CIExyz to sRGB).

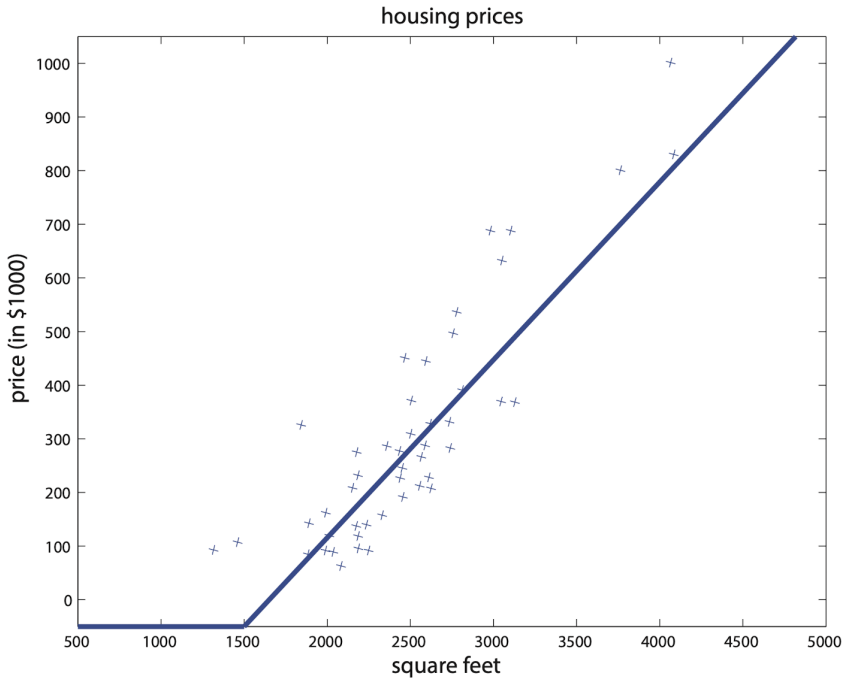


Figure 7.1: Housing prices with a “kink” in the graph.

In NN terminology, we call these non-linear functions *activation function*, some common activation functions are given here

$$\begin{aligned}\text{sigmoid}(x) &= \frac{1}{1+e^{-x}} \\ \text{ReLU}(x) &= \max\{0,\, x\} \\ \tanh &= \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}\end{aligned}$$

In this example, we can use ReLU for sure, which will give us $h_\theta(x) = \text{ReLU}(w^T x + b)$. And the term b is often referred to as the “bias”, and the vector w is referred to as the weight vector. Such a neural network has 1 layer.

Stacking neurons

Let’s suppose that we have 4 entries in our input x, where x_1 stands for size, x_2 for bedrooms, x_3 for the zip Code, x_4 for wealth, and these are all indeed factors that could effect the price. But if we dive this question a little deeper, we’ll see that these aren’t the direct factors that are pull strings, they are actually effecting some hiden and more direct factors such as school, family size and walkable. Let’s call these three hidden factors a_1, a_2, a_3, and we shall have the parameterization:

$$
\begin{aligned}
a_1 &= ReLU(\theta_1 x_1 + \theta_2 x_2 + \theta_3) \\
a_2 &= ReLU(\theta_4 x_3 + \theta_5) \\
a_3 &= ReLU(\theta_6 x_3 + \theta_7 x_4 + \theta_8)
\end{aligned}
$$

and we have

$$
h_\theta(x) = \theta_9 a_1 + \theta_{10} a_2 + \theta_{11} a_3 + \theta_{12}
$$

The only problem we have now is to learn these 12 parameters.

Two-layer fully-connected neural networks

However, what we did before, deciding which of the inputs will decide the hidden factor is not practically possible in most situations, so we need a more generic parameterization. A simple way would be to write the intermediate variable a_1 as a function of all $x_1, ..., x_4$:

$$
\begin{aligned}
a_1 &= ReLU(w_1^T x + b_1) \\
a_2 &= ReLU(w_2^T x + b_2) \\
a_3 &= ReLU(w_3^T x + b_3)
\end{aligned}
$$

That's much better! Still the last layer won't be change, and now we have a so called *"two layer fully-connected neural network"*, we're calling it fully connected because the intermediate variables are fully decided by all the original inputs.

For full generality, a two-layer fully-connected neural network with m hidden units and d dimensional input $x \in R^d$ is defined as

$$
\begin{aligned}
\forall j \in [1, ..., m], z_j = w_j^{[1]\top} x + b_j^{[1]} \ where \ \ w_j^{[1]} \in R^d, \ b_j^{[1]} \in R \\
a_j &= ReLU(z_j) \\
a &= [a_1, ..., a_m]^\top \in R^m \\
h_\theta(x) &= w^{[2]\top} a + b^{[2]} \ \ where \ \ w^{[2]} \in R^m, b^{[2]} \in R,
\end{aligned}
$$

Note that we all use column vector in the notes while all vectors or tensors are viewed as row vectors in most packages such as PyTorch or Numpy.

Vectorization

Before we introduce neural networks with more layers and more complex structures, we will simplify the expressions for neural networks with more matrix and vector notations.

We vectorize the two-layer fully-connected neural network as below. We define a weight matrix $W^{[1]}$ in $R^{m \times d}$ as the concatenation of all the vectors $w_j^{[1]}$'s in the following way:

$$W^{[1]} = \begin{bmatrix} —w_1^{[1]T}— \\ —w_2^{[1]T}— \\ \vdots \\ —w_m^{[1]T}— \end{bmatrix} \in R^{m \times d}$$

Now by the definition of matrix vector multiplication, we can write $z = [z_1, ..., z_m]^\top \in R^m$ as

$$z = W^{[1]}x + b^{[1]}$$

where $x \in R^d, b \in R^m$. Computing the activations $a \in R^m$ from $z \in R^m$ involves an element-wise non-linear application of the ReLU function, which can be computed in parallel efficiently. Define $W^{[2]} = [w^{[2]\top}] \in R^{1 \times m}$ similarly. Then, the model in equation (7.11) can be summarized as

$$\begin{aligned} a &= ReLU(W^{[1]}x + b^{[1]}) \\ h_\theta(x) &= W^{[2]}a + b^{[2]} \end{aligned}$$

The collection of W[1],b[1] is referred to as the first layer, and W[2],b[2] the second layer. The activation a is referred to as *the hidden layer*. A two-layer neural network is also called *one-hidden-layer neural network*.

Multi-layer fully-connected neural networks

For the multi-layer NN, we have the recursion

$$a^{[k]} = ReLU(W^{[k]}a^{[k-1]} + b^{[k]}), \forall k = 1, ..., r-1$$

Note that this is true for $a^{[r]} = h_\theta(x)$, but most of times, we apply a linear layer to the last layer, where

$$a^{[r]} = W^{[r]}a^{[r-1]} + b^{[r]}$$

so that negative outputs are possible and it's easier to interpret the last layer as a linear model.

Connection to the Kernel Method

Oftentimes people use domain knowledge to design the feature map φ(x) that suits the particular applications. The process of choosing the feature maps is often referred to as *feature engineering*. We can view deep learning as a way to automatically learn the right feature map (sometimes also referred to as "the representation") as follows. Suppose we denote by β the collection of the parameters in a fully-connected neural networks (equation (7.17)) except those in the last layer. Then we can abstract right a[r−1] as a function of the input x and the parameters in β : $a^{[r-1]} = \phi_\beta(x)$. Now we can write the model as

$$h_\theta(x) = W^{[r]}\phi_\beta(x) + b^{[r]}$$

When β is fixed, then $\phi_\beta(\cdot)$ can viewed as a feature map, and therefore $h_\theta(x)$ is just a linear model over the features $\phi_\beta(\cdot)$. However, we will train the neural networks, both the parameters in β and the parameters W[r],b[r] are optimized, and therefore we are not learning a linear model in the feature space, but also learning a good feature map $\phi_\beta(\cdot)$ itself so that it's possible to predict accurately with a linear model on top of the feature map.

Therefore, deep learning tends to depend less on the domain knowledge of the particular applications and requires often less feature engineering. The penultimate layer a[r] is often (informally) referred to as *the learned features* or representations in the context of deep learning.

However, oftentimes, the neural network will discover complex features which are very useful for predicting the output but may be difficult for a human to understand or interpret. This is why some people refer to neural networks as a black box, as it can be difficult to understand the features it has discovered.

Backpropagation

In this section, we introduce backpropgation or auto-differentiation, which computes the gradient of the loss $\nabla J^{(j)}(\theta)$ efficiently. We will start with an informal theorem that states that as long as a real-valued function f can be efficiently computed/evaluated by a differentiable network or circuit, then its gradient can be efficiently computed in a similar time. We will then show how to do this concretely for fully-connected neural networks.

We assume that each of the operations and functions, and their derivatives or partial derivatives can be computed in O(1) time in the computer.

Theorem 7.3.1:[backpropagation or auto-differentiation, informally sta
Suppose a differentiable circuit of size N computes a real-valued function $f : R^\ell \to R$. Then, the gradient ∇f can be computed in time O(N), by a ci

We note that the loss function $J^{(j)}(\theta)$ for j-th example can be indeed computed by a sequence of operations and functions involving additions, subtraction, multiplications, and non-linear activations. Thus the theorem suggests that we should be able to compute the $\nabla J^{(j)}(\theta)$ in a similar time to that for computing $J^{(j)}(\theta)$ itself.

Preliminary: chain rule

Define two layers where

$$g_j = g_j(\theta_1, ..., \theta_p), \forall j \in 1, \cdots, k$$
$$J = J(g_1, ..., g_k)$$

Then, by the chain rule, we have that $\forall i$,

$$\frac{\partial J}{\partial \theta_i} = \sum_{j=1}^{k} \frac{\partial J}{\partial g_j} \cdot \frac{\partial g_j}{\partial \theta_i}$$

There are actually a lot in this section, but they're basically reuse of the chain rules, and thus we're leaving them behind to save our time. The only thing worth noticing is that $v \odot w$ denote the entry-wise product of two vectors v and w of the same dimension.

Vectorization over training examples

As we discussed before, in the implementation of neural networks, we will leverage the parallelism across the multiple examples. This means that we will need to write the forward pass (the evaluation of the outputs) of the neural network and the backward pass (backpropagation) for multiple training examples in matrix notation.

The basic idea

Suppose we have three training example $x^{(1)}, x^{(2)}, x^{(3)}$, and for thiem we have

$$z^{[1](1)} = W^{[1]} x^{(1)} + b^{[1]}$$
$$z^{[1](2)} = W^{[1]} x^{(2)} + b^{[1]}$$
$$z^{[1](3)} = W^{[1]} x^{(3)} + b^{[1]}$$

Note the difference between square brackets [·], which refer to the layer number, and parenthesis (·), which refer to the training example number.

As it's turned out, all of this could be expressed in a vectorized way. Define

$$X = \begin{bmatrix} | & | & | \\ x^{(1)} & x^{(2)} & x^{(3)} \\ | & | & | \end{bmatrix} \in R^{d \times 3}$$

and

$$Z^{[1]} = \begin{bmatrix} | & | & | \\ z^{[1](1)} & z^{[1](2)} & z^{[1](3)} \\ | & | & | \end{bmatrix} = W^{[1]}X + b^{[1]}$$

You may notice that we are attempting to add $b^{[1]} \in R^{4 \times 1}$ to $W^{[1]}X \in R^{4 \times 3}$. Strictly following the rules of linear algebra, this is not allowed. In practice however, this addition is performed using broadcasting. We create
an intermediate
$\overline{b}^{[1]} \in R^{4 \times 1}$:

$$\overline{b}^{[1]} = \begin{bmatrix} | & | & | \\ b^{[1]} & b^{[1]} & b^{[1]} \\ | & | & | \end{bmatrix}$$

The matricization approach as above can easily generalize to multiple layers, with one subtlety though, as discussed below.

Complications/Subtlety in the Implementation

All the deep learning packages or implementations put the data points in the rows of a data matrix. (If the data point itself is a matrix or tensor, then the data are concentrated along the zero-th dimension.) However, most of the deep learning papers use a similar notation to these notes where the data points are treated as column vectors. There is a simple conversion to deal with the mismatch: in the implementation, all the columns become row vectors, row vectors become column vectors, all the matrices are transposed, and the orders of the matrix multiplications are flipped.

The computation for the hidden activation become

$$Z^{[1]} = XW^{[1]} + b^{[1]} \in R^{3 \times m}$$

There are so much more about deep learning and neural network could be discovered, but we won't be discussing them here, but maybe later.

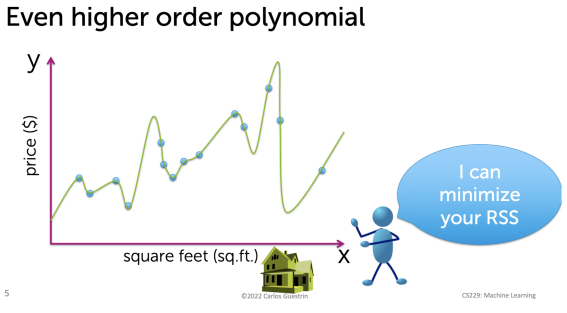
Regulation And Generalization

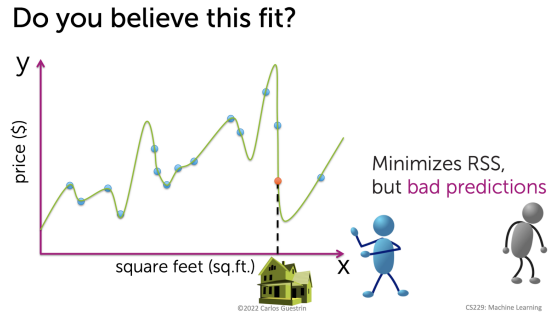
Bias-Variance Tradeoff and so more

Bias-Variance Tradeoff

> "Remember that all models are wrong; the practical question is how wrong do they have to be to not be useful." George Box, 1987.

That tells us a truth about machine learning, that sometime a perfect fit could be a terrible prediction. Just take this example





Assess the loss

So how can we **assess the loss** and make a perfect balance out of it?

Here's our first stop, *training errors*. Training error is what we encounter when training a model using a specific dataset, that is to say the error between our training data and our model predictions.

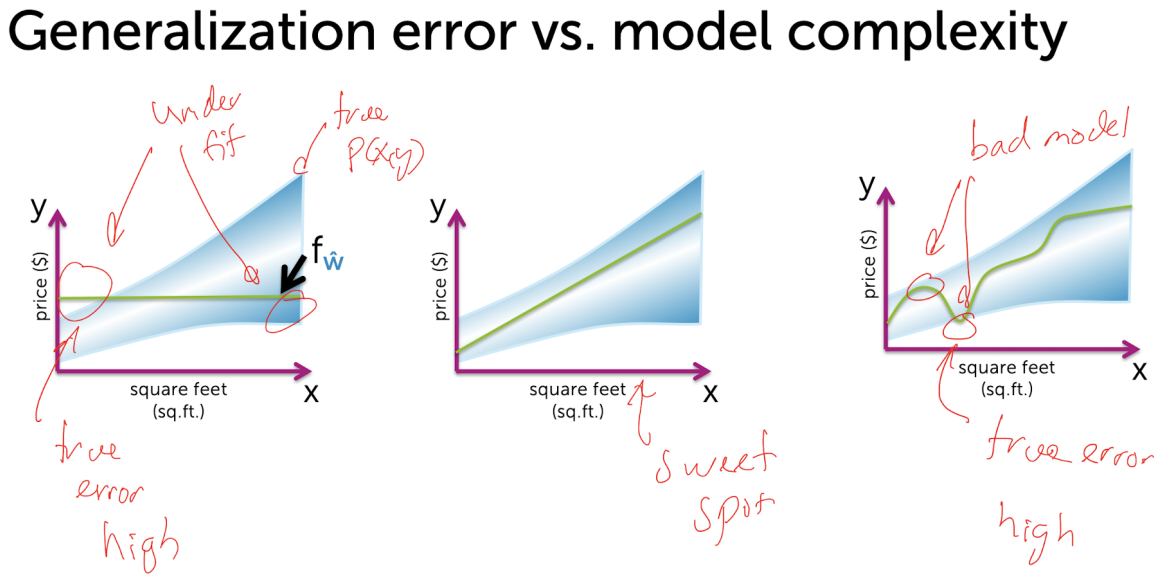
$$Training\ error(\hat{w}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - f_{\hat{w}}(x_i))^2$$

Obviously, a small training error doesn't mean a good prediction, unless you can cover every data you will encounter in the future.

And to our second stop, we have *generalization (true) errors.* Formally it is

$$generalization\ error = E_{x,y}[L(y, f_{\hat{w}}(x))]$$

In which $L(y, f_{\hat{w}}(x))$ is the loss function of the model $f_{\hat{w}}(x)$. It average over all possible (x,y) pairs weighted by how likely each is.



The last stop is *test error*, this is the easiest to understand, so we'll just skip this.

3 sources of error + the bias-variance tradeoff

With all three error we already defined, we now consider using a we to quantify our losses. We define three error to fully describe the ability of a model in prediction.

First is *bias*. Bias is defined as the subtraction between the real data and the predicted value, it describes how a model fits data on average. It could be denoted as

$$Bias(x) = f_{w(true)}(x) - f_{\bar{w}}(x)$$

where the $f_{\bar{w}}(x)$ is the averrage prediction of a range of different models $f_{\hat{w}}(x_i)$. Obviously, a high bias indicates a simple model with bad predictions.

Second is *variance*. Variance is defined as the variance of predicted values. It describe the range our models vary, and thus describes the universality of our model. It's denoted as

$$Variance(x) = \frac{1}{N}\sum_{i=1}^{N}(f_{w_i}(x) - f_{\bar{w}}(x))^2$$

The last is *noise.* Noise is something inherent, which you can't avoid. Even we have a perfect mathematic description for our data, we still encounter the noise which could come from all sorts of error such as measurement errors. It's denoted as

$$Noise(x) = \frac{1}{N}\sum_{i=1}^{N}(y_i - f_{w(true)}(x_i))$$

And now we can derive our bias-variance tradeoff from this three error.

$$\sigma^2 + [bias(f_{\hat{w}}(x_t))]^2 + var(f_{\hat{w}}(x_t))$$

Where σ is the noise that follows a Gaussian distribution.

Ridge Regression: Regulating overfitting when using many features

We can give a think on this: if we have more features included in one model, wether will the coefficient get bigger or smaller? We have an example here:

Thus we see that more features, which probably leads to overfit, also means larger coefficients. So that indicates us that we could try to regulate our coefficient to avoid overfitting.

We consider take the magnitude of our coefficient into the loss function

$$\nabla_w J(w) = \nabla_w \left[\frac{1}{2n}(Xw-y)^T(Xw-y)\right] + \nabla_w\left(\frac{\lambda}{2}w^Tw\right)$$
$$= \frac{1}{n}(X^TXw - X^Ty) + \lambda w$$
$$= \frac{1}{n}X^T(Xw-y) + \lambda w$$
$$\Rightarrow w := w - \frac{1}{n}X^T(Xw-y) - \lambda w$$

here could use a $\frac{1}{2}$ factor

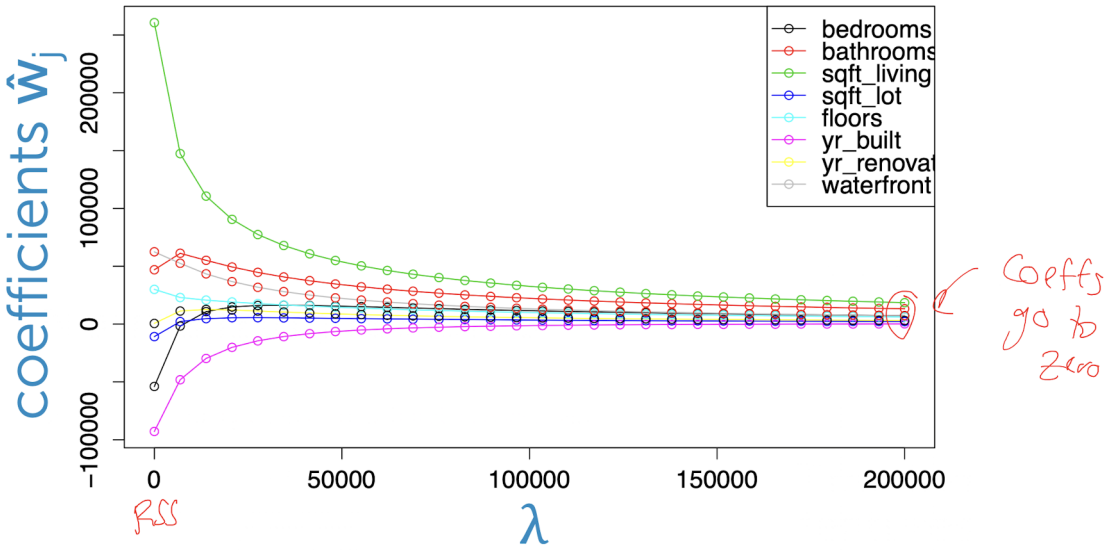
$$J(w) = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda||w||_2^2$$

here i miss the $\frac{1}{2n}$ factor

This is the motivate form of *ridge regression*, this is also called *the L_2 regularization*.

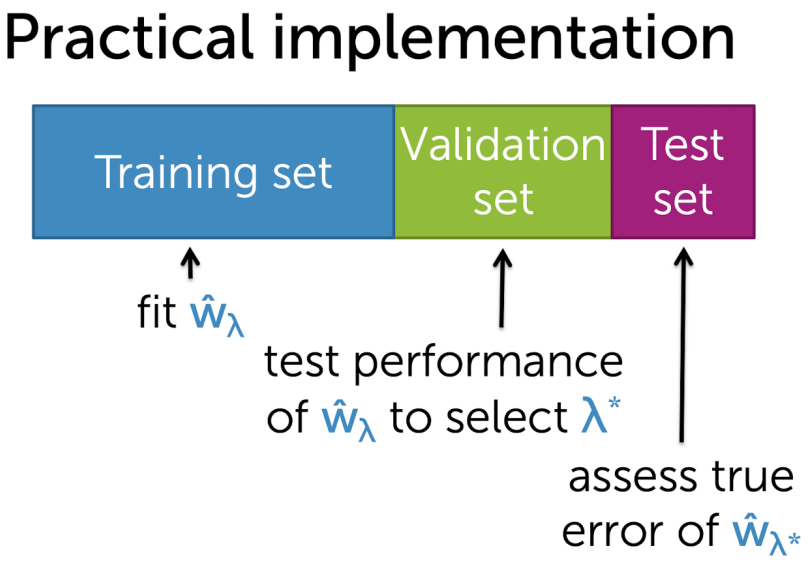
Let's decide what happens to estimated coefficients of ridge regression as tuning parameter λ is varied. If we set λ to be ∞, we can easily see that the coefficient w to be 0, that is to say every feature wasn't considered. Otherwise, if we take w to be 0, then the coefficient would be what we see in normal linear regressions (RSS), that changes noting. These conclusion allow us safely to say that when the λ is between 0 and ∞, the coefficients are something in between.

Coefficient path

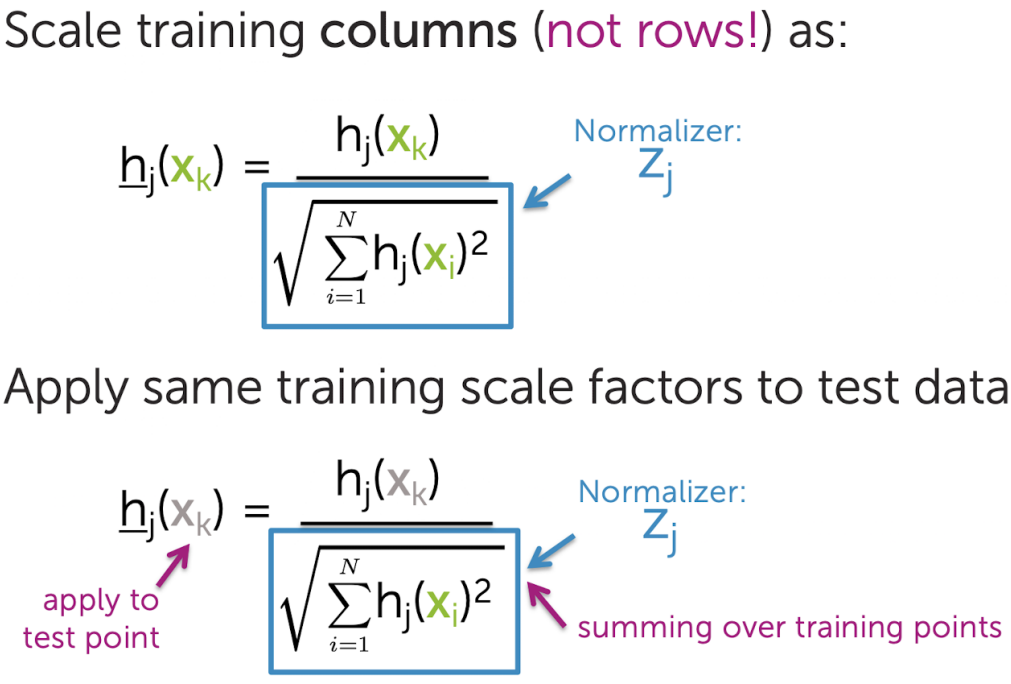


To select a proper λ, we can use a additional validation dataset to fix the issue brought by two datasets.

1. Select λ* such that ŵ minimizes error on validation set
2. Approximate true error of ŵ using test set



Sometimes we can normalize our features so that to accelerate our training under a certain conditions.



Lasso Regression: Regularization for feature selection

From what we just learned in ridge regression, we now know about something about how to regulate our coefficient to avoid overfitting. Now let's us consider when we need sparsity, or that is to say when we need to decide which data are specifically relevant.

We have few options:

1. We take every possible feature into our scope (every subset)
2. Or, we can use *greedy algorithm* to do so, that means we first take all(or none) features in, and then gradually remove(or add) them from our scope one by one

Yet both of this ways involves with incredible computational cost, we have to find a better way.

Consider this: if we take the L_1 norm of the coefficients into the loss function, we will be able to "eliminate" some of the features in our original scope.

$$J(w) = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda||w||_1$$

$\nabla_w J(w) = \nabla_w \left(\frac{1}{2n}(X\theta - y)^T(X\theta - y)\right) + \lambda\nabla_w||w||_1$

$= \frac{1}{n}X^T(X\theta - y) + \lambda \, sign(w)$

note $sign(w) = \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix}$ terrible!

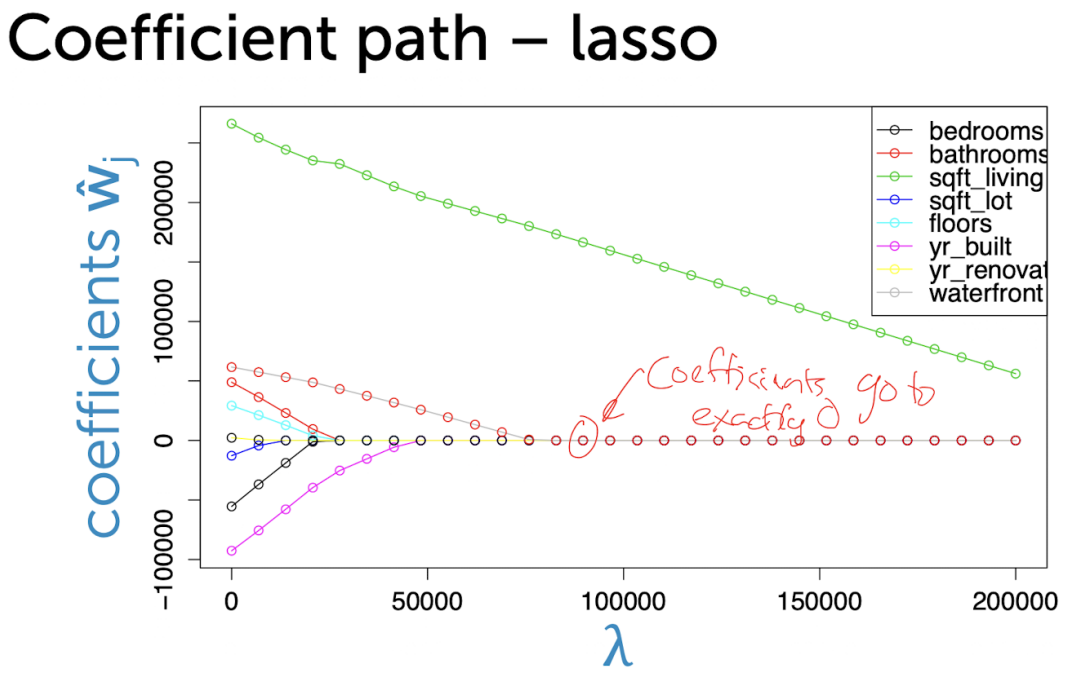
$w = \begin{bmatrix} -0.2 \\ 0.6 \\ -0.3 \end{bmatrix}$

you need something from convex optimization

1. coordinate descent
2. subgradient descent

Similar to the ridge regression, but this will tune parameter into a state of being balance of fit and sparsity

usually check with cross-validation



Yet there's a few problem with this, that is Lasso shrinks coefficients relative to LS solution, which leads to more bias, less variance. To avoid this we could reduce bias as follows:

1. Run lasso to select features
2. Run least squares regression with only selected features

A thing on Lasso and Ridge

With group of highly correlated features, lasso tends to select amongst them arbitrarily

- Often prefer to select all together

Often, empirically ridge has better predictive performance than lasso, but lasso leads to sparser solution

Elastic net aims to address these issues

- hybrid between lasso and ridge regression
- uses L1 and L2 penalties

$$\text{Loss} = \frac{1}{2n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda\left(\alpha_1||\beta||_2^2 + \alpha_2||\beta||_1\right)$$

coordinate descent for Lasso

We try to solve $\min_\theta \left\{\frac{1}{2m}|X\theta - y|_2^2 + \lambda|\theta|_1\right\}$, we can do the following for every θ_i:

fix all the other "θ"s, and try to optimize $\min_{\theta_j}\left\{\frac{1}{2m}||X\theta - y||_2^2 + \lambda|\theta_j|\right\}$

for every θ_j, the residual would be $r_j = y - X\theta + \theta_j X_j$, X_j is the jth column of X

And we can define ρ as $\rho = X_j^T r_j = X_j^T(y - X\theta + \theta_j X_j)$

Now use soft-threshold function which is $S(\rho, \lambda) = \text{sign}(\rho)\cdot\max(|\rho| - \lambda, 0)$

let $\theta_j := S(\rho, \lambda)$

$\arg\min_{w_j} \frac{1}{2m}\sum_{i=1}^{m}(y^{(i)} - w^T x^{(i)})^2 + \lambda\sum_{i=1}^{m}||w_i||$ $\quad r_j^{(i)} = \sum_{\substack{t=1 \\ t\neq j}}^{m} y^{(i)} - w_t x_t^{(i)}$

if we fix the other w_i leave out w_j

$\Rightarrow$ to minimize $\frac{1}{2m}\sum_{i=1}^{m}(r_j^{(i)} - x_j^{(i)} w_j)^2 + \lambda||w_j|| + C = \varphi(w_j)$

$$\nabla_{w_j}\varphi(w_j) = -\frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}(r_j^{(i)} - x_j^{(i)} w_j) + \lambda\,\text{sign}(w_j)$$

$$= \underbrace{\frac{1}{m}\sum_{i=1}^{m} x_j^{(i)2}}_{z_j} w_j - \underbrace{\frac{1}{m}\sum_{i=1}^{m} x_j^{(i)} r_j^{(i)}}_{\rho_j} + \lambda\,\text{sign}(w_j) = 0$$

$$\Rightarrow z_j w_j - \rho_j + \lambda\,\text{sign}(w_j) = 0$$

note that $\text{sign}(w_j) = \text{sign}(\rho_j - \lambda\,\text{sign}(w_j))$

$$\begin{cases} \text{when } \rho_j - \lambda > 0 & \text{we must have } w_j > 0 \quad w_j = \frac{\rho_j - \lambda}{z_j} \\ \text{when } \rho_j + \lambda < 0 & \text{we must have } w_j < 0 \quad w_j = \frac{\rho_j + \lambda}{z_j} \\ \text{whe } -\lambda \le \rho_j \le \lambda & \text{we must have } w_j = 0 \end{cases}$$

How we have $E(L(y_i, f_{\hat{w}}(x_i)))$

$$E(L(y_i, f_{\hat{w}}(x_i))) = E((y_i - f_{\hat{w}})^2)$$

$$= E((y_i - f + f - f_{\hat{w}})^2)$$

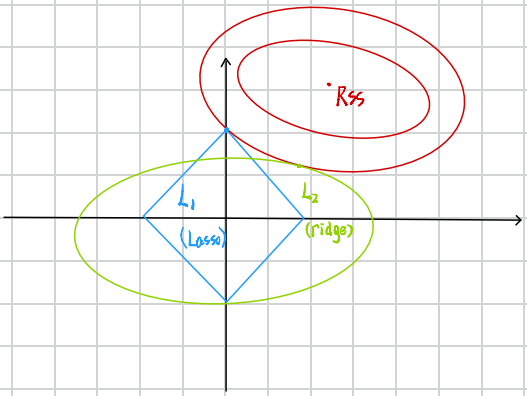
$$= \underbrace{E((y_i - f)^2)}_{\sigma^2} + \underbrace{2E(y_i - f)E(f - f_{\hat{w}})}_{E(y_i - f) = 0} + \underbrace{E((f - f_{\hat{w}})^2)}_{MSE}$$

$E(y_i - f) = 0 \Rightarrow$ noise ~ gaussian where $\mu = 0$

$$MSE = E((f - f_{\bar{w}} + f_{\bar{w}} - f_{\hat{w}})^2)$$

$$= \underbrace{E((f - f_{\bar{w}})^2)}_{\Rightarrow bias^2} + 2E(f - f_{\bar{w}})\underbrace{E(f_{\bar{w}} - f_{\hat{w}})}_{0} + \underbrace{E((f_{\bar{w}} - f_{\hat{w}})^2)}_{\Rightarrow variance}$$

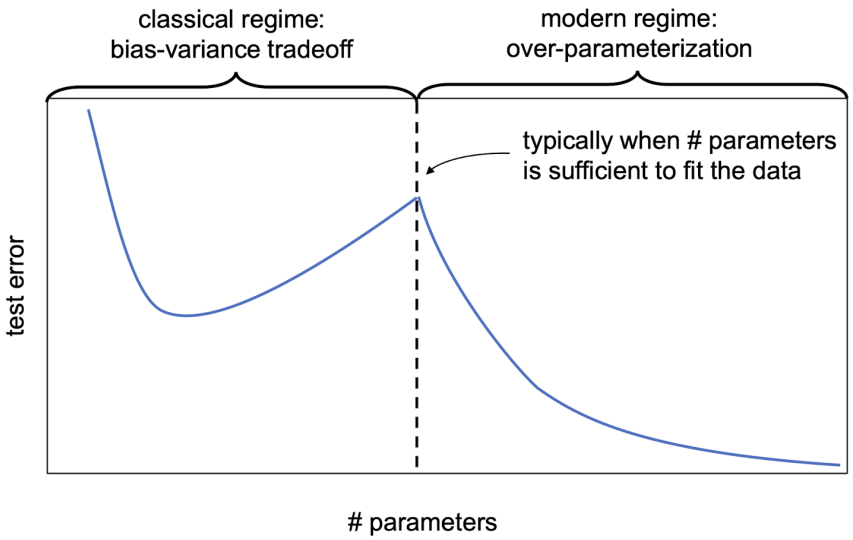
intuitive difference between L_1 and L_2 regulation



The double descent phenomenon and so more

The double descent phenomenon

For a long time, people believe that the test error(which is $\text{bias}^2 + \text{var}$) would usually first decrease to a minimum point then going upwards as the parameters grows(complexity), that is to say it's contour is convex. Yet recent works have demonstrated that the test error can present a "double descent" phenomenon in a range of machine earning models including linear models and deep neural networks.

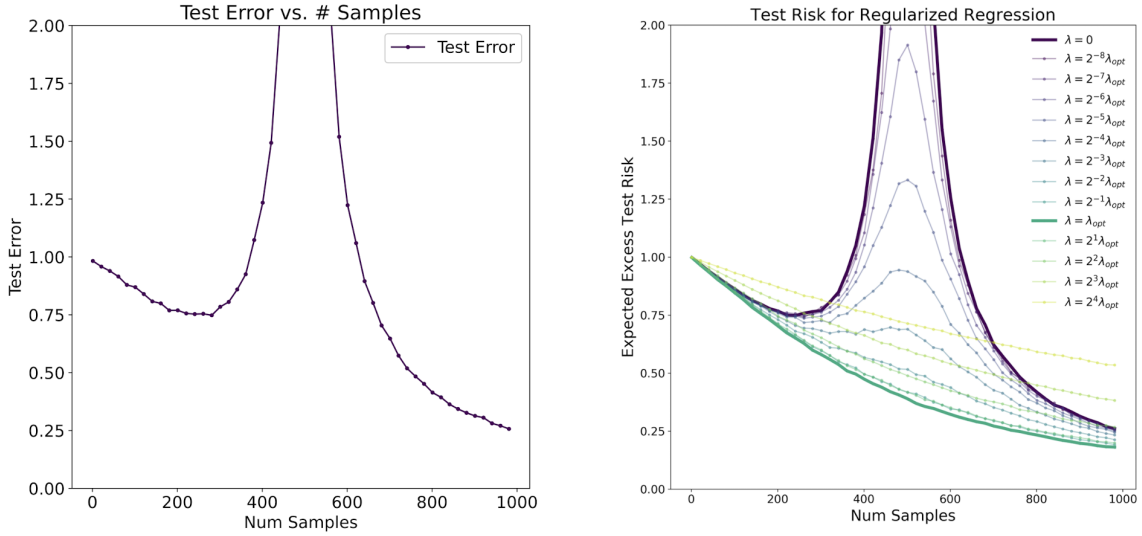


A typical model-wise double descent phenomenon. As the number of parameters increases, the test error first decreases when the number of parameters is smaller than the training data. Then in the over-parameterized regime, the test error decreases again.

Just think about the LLMs with over 100B dataset! To some extent, the over-parameterized regime with the second descent is considered as new to the machine learning community—partly because lightly-regularized, over-parameterized models are only extensively used in the deep learning era.

What we just talk about is just *the model-wise double descent*, additionally, we have another phenomenon for *sample-wise double descent*.

A priori, we would expect that more training examples always lead to smaller test errors—more samples give strictly more information for the algorithm to learn from. However, recent work observes that the test error is not monotonically decreasing as we increase the sample size. Instead, as shown in Figure, the test error decreases, and then increases and peaks around when the number of examples (denoted by n) is similar to the number of parameters (denoted by d), and then decreases again. To some extent, sample-wise double descent and model-wise double descent are essentially describing similar phenomena—the test error is peaked when $n \approx d$.



Left: The sample-wise double descent phenomenon for linear models. Right: The sample-wise double descent with different regularization strength for linear models. Using the optimal regularization parameter λ (optimally tuned for each n, shown in green solid curve) mitigates double descent.

The sample-wise double descent, or, in particular, the peak of test error at n ≈ d, suggests that the existing training algorithms evaluated in these experiments are far from optimal when n ≈d. This indicates that we may find a way to a better model to have a better play at n ≈ d, which could be either use less train examples or to use some regulations which would be discuss later.

The reason why the over-parameterized model performs so well are still in vague, one common explanation is that even without any explicit regulation, optimizers like gradient descent itself would provide some sort of implicit regulation. Like in the linear regression, where gradient descent would usually find the minimum norm solution when n<<d that fits the data, and the minimum norm regularizer turns out to be a sufficiently good for the over-parameterized regime (but it's not a good regularizer when n ≈d).

Finally, we also remark that the double descent phenomenon has been mostly observed when the model complexity is measured by the number of parameters. It is unclear if and when the number of parameters is the best complexity measure of a model. For example, in many situations, the norm of the models is used as a complexity measure. As shown in Figure right, for a particular linear case, if we plot the test error against the norm of the learnt model, the double descent phenomenon no longer occurs. This is partly because the norm of the learned model is also peaked around n≈d (See Figure(middle)). For deep neural networks, the correct complexity measure is even more elusive. The study of double descent phenomenon is an active research topic.

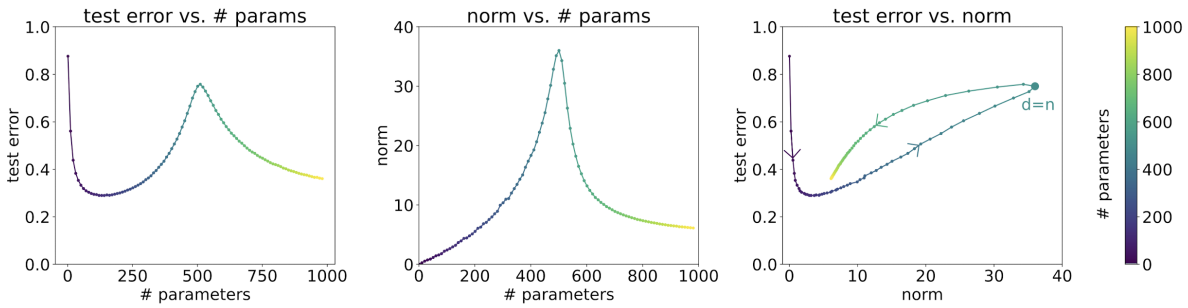


Figure 8.12: Left: The double descent phenomenon, where the number of parameters is used as the model complexity. Middle: The norm of the learned model is peaked around n ≈d. Right: The test error against the norm of the learnt model. The color bar indicate the number of parameters and the arrows indicates the direction of increasing model size. Their relationship are closer to the convention wisdom than to a double descent.

Sample complexity bounds

Preliminary

First we introduce two lemmas:

Lemma. (The union bound). Let $A_1, A_2, ..., A_k$ be k different events (that may not be independent). Then

$$P(A_1 \cup \cdots \cup A_k) \leq P(A_1) + ... + P(A_k)$$

Lemma. (Hoeffding inequality) Let $Z_1, ..., Z_n$ be n independent and identically distributed (iid) random variables drawn from a Bernoulli(ϕ) distribution. I.e., $P(Z_i = 1) = \phi$, and $P(Z_i = 0) = 1 - \phi$. Let $\hat{\phi} = (1/n) \sum_{i=1}^{n} Z_i$ be the mean of these random variables, and let any $\gamma > 0$ be fixed. Then

$$P(|\phi - \hat{\phi}| > \gamma) \leq 2 \exp(-2\gamma^2 n)$$

This lemma (which in learning theory is also called the **Chernoff bound**), this tells us the fact that the ϕ won't fall far from the mean $\hat{\phi}$, as the n is big enough. That is to say, when the n gets big, the $\hat{\phi}$ willl be a good estimate of ϕ itself.

And now cut to the chase, to simplify our exposition, let's restrict our attention to binary classification in which the labels are $y \in \{0, 1\}$. Everything we'll say here generalizes to other problems, including regression and multi-class classification.

We assume we are given a training set $S = \{(x^{(i)}, y^{(i)}); i = 1, ..., n\}$of size n, where the training examples $(x^{(i)}, y^{(i)})$ are drawn iid from some probability distribution $\mathcal{D}$. For a

hypothesis h, we define the training error (also called *the empirical risk* or *empirical error* in learning theory) to be

$$\hat{\varepsilon}(h) = \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}\{h(x^{(i)}) \neq y^{(i)}\}$$

This is just the fraction of training examples that h misclassifies.

When we want to make explicit the dependence of $\hat{\varepsilon}(h)$ on the training set S, we may also write this a $\hat{\varepsilon}_S(h)$. We also define *the generalization error* to be

$$\varepsilon(h) = P_{(x,y)\sim D}(h(x) \neq y)$$

Note that we have assumed that the training data was drawn from the same distribution Dwith which we're going to evaluate our hypotheses (in the definition of generalization error). This is sometimes also referred to as one of *the PAC assumptions*.

Consider the setting of linear classification, and let $h_\theta(x) = 1\{\theta^T x \geq 0\}$. What's a reasonable way of fitting the parameters θ? One approach is to try to minimize the training error, and pick

$$\hat{\theta} = \arg\min_{\theta} \hat{\varepsilon}_\theta(h)$$

We call this process *empirical risk minimization (ERM)*, and the resulting hypothesis output by the learning algorithm is $\hat{h} = h_{\hat{\theta}}$.

In our study of learning theory, it will be useful to abstract away from the specific parameterization of hypotheses and from issues such as whether we're using a linear classifier. We define the hypothesis class $\mathcal{H}$ used by a learning algorithm to be the set of all classifiers considered by it.

Empirical risk minimization can now be thought of as a minimization over the class of functions $\mathcal{H}$, in which the learning algorithm picks the hypothesis:

$$\hat{h} = \arg\min_{h \in \mathcal{H}} \hat{\varepsilon}(h)$$

The case of finite H

Consider a $\mathcal{H}$ which has only a finite number of models h_i in it, consisting of k hypotheses. Thus, $\mathcal{H}$ is just a set of k functions mapping from X to $\{0, 1\}$, and empirical risk minimization selects h to be whichever of these k functions has the smallest training error.

We're now doing two things: First, we will show that $\hat{\varepsilon}(h)$ is a reliable estimate of ε(h) for all h. Second, we will show that this implies an upper-bound on the generalization error of h.

Take any one, fixed, $h_i \in \mathcal{H}$. Consider a Bernoulli random variable Z whose distribution is defined as follows. We're going to sample $(x, y) \sim D$. Then, we set $Z = 1\{h_i(x) \neq y\}$. I.e., we're going to draw one example, and let Z indicate whether h_i misclassifies it. Similarly, we also define $Z_j = 1\{h_i(x^{(j)}) \neq y^{(j)}\}$. Since our training set was drawn iid from D, Z and the Z_j's have the same distribution.

We see that the misclassification probability on a randomly drawn example—that is, $\varepsilon(h)$—is exactly the expected value of Z (and Z_j). Moreover, the training error can be written

$$\hat{\varepsilon}(h_i) = \frac{1}{n} \sum_{j=1}^{n} Z_j$$

Thus, $\hat{\varepsilon}(h_i)$ is exactly the mean of the nrandom variables Z_j that are drawn iid from a Bernoulli distribution with mean $\varepsilon(h_i)$. So use the lemma before we have

$$P(|\varepsilon(h_i) - \hat{\varepsilon}(h_i)| > \gamma) \leq 2 \exp(-2\gamma^2 n)$$

But we don't just want to guarantee that $\varepsilon(h_i)$ will be close to $\hat{\varepsilon}(h_i)$ (with high probability) for just only one particular h_i. We want to prove that this will be true simultaneously for all $h \in \mathcal{H}$. To do so, let A_i denote the event that $|\varepsilon(h_i) - \hat{\varepsilon}(h_i)| > \gamma$. We've already shown that, for any particular A_i, it holds true that $P(A_i) \leq 2exp(-2\gamma^2 n)$. Thus, using the union bound, we have that

$$\begin{aligned} P\left(\exists h \in H \mid |\varepsilon(h_i) - \hat{\varepsilon}(h_i)| > \gamma\right) &= P(A_1 \cup \cdots \cup A_k) \\ &\leq \sum_{i=1}^{k} P(A_i) \\ &\leq k \cdot 2 \exp(-2\gamma^2 n) \\ &= 2k \exp(-2\gamma^2 n) \end{aligned}$$

If we subtract both sides from 1, we find that

$$\begin{aligned} P\left(\neg\exists h \in H \mid |\varepsilon(h_i) - \hat{\varepsilon}(h_i)| > \gamma\right) &= P\left(\forall h \in H \mid |\varepsilon(h_i) - \hat{\varepsilon}(h_i)| \leq \gamma\right) \\ &= 1 - P\left(\exists h \in H \mid |\varepsilon(h_i) - \hat{\varepsilon}(h_i)| > \gamma\right) \\ &\geq 1 - 2k \exp(-2\gamma^2 n) \end{aligned}$$

So, with probability at least $1 - 2k\exp(-2\gamma^2 n)$, we have that $\varepsilon(h)$ will be within γ of $\hat{\varepsilon}(h)$ for all $h \in H$. This is called a *uniform convergence result*.

There are three quantities of interest here: n, γ, and the probability of error; we can bound either one in terms of the other two.

For instance, we can ask the following question: Given γ and some $\delta > 0$, how large must n be before we can guarantee that with probability at least $1 - \delta$, training error will be within γ of generalization error? By setting $\delta = 2k\exp(-2\gamma^2 n)$, we have

note: γ the tol. between $\hat{\varepsilon}(h)$ and $\varepsilon(h)$ & the prob of $|\hat{\varepsilon}(h) - \varepsilon(h)| > \gamma$

$$n \geq \frac{1}{2\gamma^2} \log \frac{2k}{\delta}$$

$P \geq 1 - 2k\exp(-2\gamma^2 n)$
we want $p \geq 1 - \delta$
set $\delta = 2k\exp(-2\gamma^2 n)$

then with probability at least $1 - \delta$, we have that $|\varepsilon(h) - \hat{\varepsilon}(h)| \leq \gamma$ for all $h \in \mathcal{H}$.

This bound tells us how many training examples we need in order make a guarantee. The training set size n that a certain method or algorithm requires in order to achieve a certain level of performance is also called *the algorithm's sample complexity*.

Similarly, we can also hold n and δ fixed and solve for γ in the previous equation, and show [again, convince yourself that this is right!] that with probability $1 - \delta$, we have that for all $h \in \mathcal{H}$

$$|\hat{\varepsilon}(h) - \varepsilon(h)| \leq \sqrt{\frac{1}{2n} \log \frac{2k}{\delta}}$$

Now, let's assume that uniform convergence holds, what can we prove about the generalization of our learning algorithm that picked $\hat{h} = \arg\min_{h \in \mathcal{H}} \hat{\varepsilon}(h)$?

Define $h^* = \arg\min_{h \in \mathcal{H}} \varepsilon(h)$ to be the best possible hypothesis, then we have

$$\begin{aligned} \varepsilon(\hat{h}) &\leq \hat{\varepsilon}(\hat{h}) + \gamma \\ &\leq \hat{\varepsilon}(h^*) + \gamma \\ &\leq \varepsilon(h^*) + 2\gamma \end{aligned}$$

So, what we've shown is the following: If uniform convergence occurs, then the generalization error of $\hat{h}$ is at most 2γ worse than the best possible hypothesis in $\mathcal{H}$! Let's put all this together into a theorem.

Theorem. Let $|\mathcal{H}| = k$, and let any n, δ be fixed. Then with probability at least $1 - \delta$, we have that

$$\varepsilon(\hat{h}) \leq \min_{h \in H} \varepsilon(h) + 2\sqrt{\left(\frac{1}{2n} \log \frac{2k}{\delta}\right)}$$

This also quantifies what we were saying previously saying about the bias/variance tradeoff in model selection. Specifically, suppose we have some hypothesis class $\mathcal{H}$, and are considering switching to some much larger hypothesis class $\mathcal{H}\prime \supseteq \mathcal{H}$. If we switch to $\mathcal{H}\prime$, then the first term $\min_h \varepsilon(h)$ can only decrease (since we'd then be taking a min over a larger set of functions). Hence, by learning using a larger hypothesis class, our "bias" can only decrease. However, if k increases, then the second 2√·term would also increase. This increase corresponds to our "variance" increasing when we use a larger hypothesis class.

And last, we fix γ and δ we can also obtain the following sample complexity bound:

Corollary. Let $|\mathcal{H}| = k$, and let any δ, γ be fixed. Then for $\varepsilon(h) \leq \min_h \in H\varepsilon(h) + 2\gamma$ to hold with probability at least $1 - \delta$, it suffices that

$$n \geq \frac{1}{2\gamma^2} \log \frac{2k}{\delta} = O\left(\frac{1}{\gamma^2} \log \frac{k}{\delta}\right)$$

The case of infinite H

By using some interesting tricks, we can come a conclusion that if what we try to do is minimize training error, then in order to learn "well" using a hypothesis class that has d parameters, generally we're going to need on the order of a linear number of training examples in d.

(At this point, it's worth noting that these results were proved for an algorithm that uses empirical risk minimization. Thus, while the linear dependence of sample complexity on d does generally hold for most discriminative learning algorithms that try to minimize training error or some approximation to training error, these conclusions do not always apply as readily to discriminative learning algorithms. Giving good theoretical guarantees on many non-ERM learning algorithms is still an area of active research.)

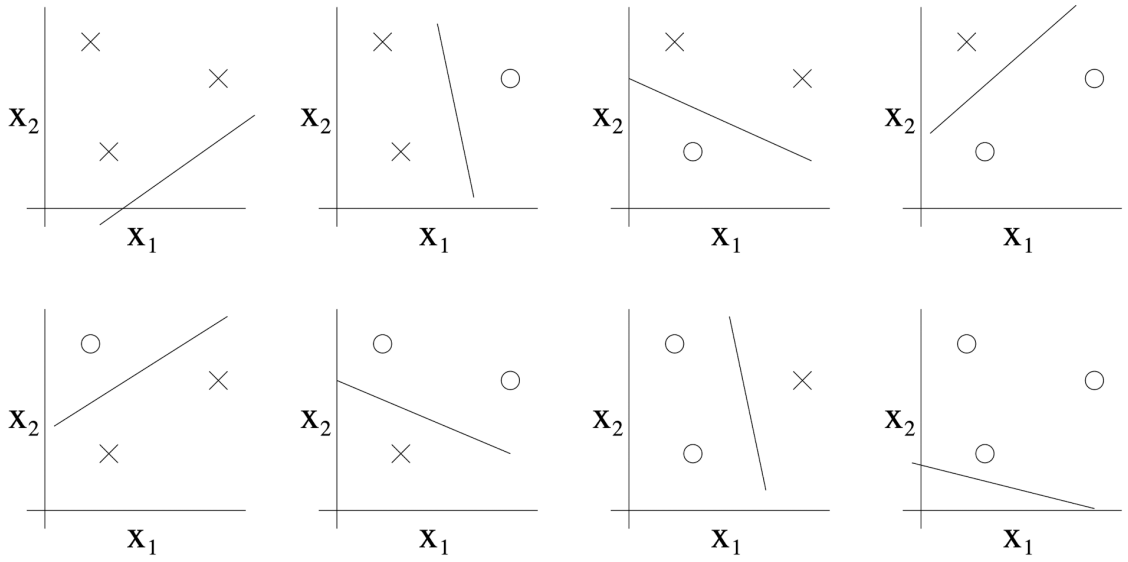
However, the problem here is that we've been talking too much about parameters of $\mathcal{H}$, but they should never really matter that much. To derive a more satisfying argument, let's define a few more things.

Given a set $S = \{x^{(i)}, ..., x^{(D)}\}$(no relation to the training set) of points $x^{(i)} \in X$, we say that $\mathcal{H}$ $\mathrm{shatters}$ S if $\mathcal{H}$ can realize any labeling on S. I.e., if for any set of labels $\{y^{(1)}, ..., y^{(D)}\}$, there exists some $h \in \mathcal{H}$ so that $h(x^{(i)}) = y^{(i)}$ for all i= 1,...D.

And we can define its *Vapnik-Chervonenkis dimension*, written $VC(\mathcal{H})$, to be the size of the largest set that is shattered by $\mathcal{H}$. (If $\mathcal{H}$ can shatter arbitrarily large sets, then $VC(\mathcal{H}) = \infty$

.)

For example, the $VC(\mathcal{H})$ of a linear classifier is 3, which mean it can only separate 3 point at most.



In order words, under the definition of the VC dimension, in order to prove that $VC(\mathcal{H})$ is at least D, we need to show only that there's at least one set of size D that $\mathcal{H}$ can shatter. And we have a theorem that

Theorem. Let $\mathcal{H}$ be given, and let $D = VC(\mathcal{H})$. Then with probability at least $1 - \delta$, we have that for all $h \in \mathcal{H}$

$$|\varepsilon(h) - \hat{\varepsilon}(h)| \leq O\left(\sqrt{\frac{D}{n}\log\frac{n}{D} + \frac{1}{n}\log\frac{1}{\delta}}\right)$$

Thus, with probability at least $1 - \delta$, we also have that:

$$\varepsilon(\hat{h}) \leq \varepsilon(h^*) + O\left(\sqrt{\frac{D}{n}\log\frac{n}{D} + \frac{1}{n}\log\frac{1}{\delta}}\right)$$

In other words, if a hypothesis class has finite VC dimension, then uniform convergence occurs as n becomes large. As before, this allows us to give a bound on $\varepsilon(h)$ in terms of $\varepsilon(h^*)$. We also have the following corollary:

Corollary. For $|\varepsilon(h) - \hat{\varepsilon}(h)| \leq \gamma$ to hold for all $h \in \mathcal{H}$(and hence $\varepsilon(h) \leq \varepsilon(h^*) + 2\gamma$)with probability at least $1 - \delta$, it suffices that $n = O_{\gamma,\delta}(D)$.

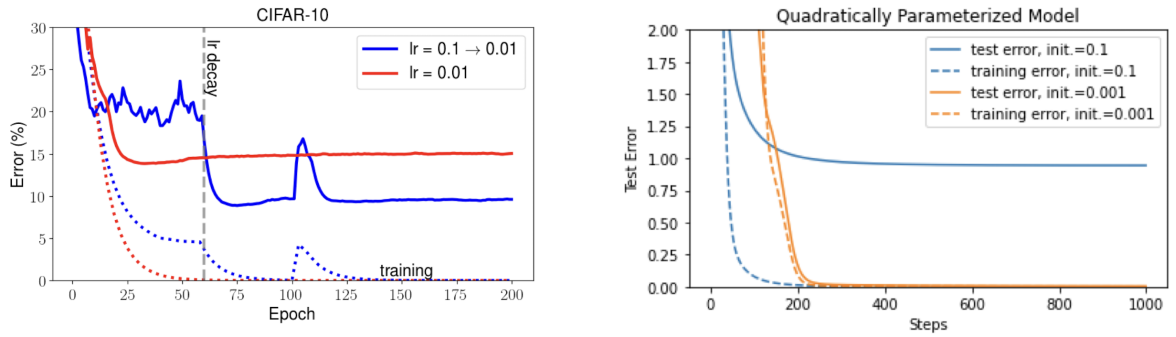
In other words, the number of training examples needed to learn “well” using His linear in the VC dimension of $\mathcal{H}$. It turns out that, for “most”hypothesis classes, the VC dimension (assuming a “reasonable” parameterization) is also roughly linear in the number of parameters.

Regularization and model selection

Implicit regularization effect

As we mentioned before, some optimizer have some kind regularization effect on their own. In most classical settings, the loss or regularized loss has a unique global minimum, and thus any reasonable optimizer should converge to that global minimum and cannot impose any additional preferences. However, in deep learning, oftentimes the loss or regularized loss has more than one (approximate) global minima, and difference optimizers may converge to different global minima. Though they might have similar train losses, they may actually have completely different nature and performance.

For example, it's possible that one global minimum gives a much more Lipschitz or sparse model than others and thus has a better test error. It turns out that many commonly-used optimizers (or their components) prefer or bias towards finding global minima of certain properties, leading to a better test performance.



Left: Performance of neural networks trained by two different learning rates schedules on the CIFAR-10 dataset. Although both experiments used exactly the same regularized losses and the optimizers fit the training data perfectly, the models' generalization performance differ much. **Right:** On a different synthetic dataset, optimizers with different initializations have the same training error but different generalization performance.

In summary, the takehome message here is that the choice of optimizer does not only affect minimizing the training loss, but also imposes implicit regularization and affects the generalization of the model. Even if your current optimizer already converges to a small training error perfectly, you may still need to tune your optimizer for a better generalization.

In many (but definitely far from all) situations, among those setting where optimization can succeed in minimizing the training loss, the use of larger initial learning rate, smaller initialization, smaller batch size, and momentum appears to help with biasing towards more generalizable solutions. However, a clearer and more specific condition for this remains an active research fields for researchers.

Model selection via cross validation

We'll introduce three kind of cross validation methods here.

1. **Hold-out cross validation** (also called **simple cross validation**)
    a. Randomly split S into S_{train} (say, 70% of the data) and S_{cv} (the remaining 30%). Here, S_{cv} is called the hold-out cross validation set.
    b. Train each model M_i on S_{train} only, to get some hypothesis h_i.
    c. Select and output the hypothesis h_i that had the smallest error $\hat{\varepsilon}_{S_{cv}}(h_i)$ on the hold out cross validation set. (Here $\hat{\varepsilon}_{S_{cv}}(h)$ denotes the average error of h on the set of examples in S_{cv}.) The error on the hold out validation set is also referred to as the validation error.

⚠️ Optionally, step 3 in the algorithm may also be replaced with selecting the model M_i according to $\arg\min_i \varepsilon_{S_{cv}}(h_i)$, and then retraining M_i on the entire training set S.

2. **k-fold cross validation**

   a. Randomly split S into k disjoint subsets of m/k training examples each. Lets call these subsets $S_1, ..., S_k$.

   b. For each model M_i, we evaluate it as follows:
   For
   $j = 1, ..., k$
   Train the model
   M_i on $S_1 \cup \cdots \cup S_{j-1} \cup S_{j+1} \cup \cdots S_k$ (i.e., train on all the data except S_j) to get some hypothesis h_{ij}.

   Test the hypothesis h_{ij} on S_j, to get $\hat{\varepsilon}_{S_j}(h_{ij})$.
   The estimated generalization error of model Mi is then calculated as the average of the
   $\hat{\varepsilon}S_j(h_{ij})$'s (averaged over j).

   c. Pick the model M_i with the lowest estimated generalization error, and retrain that model on the entire training set S. The resulting hypothesis is then output as our final answer.

⚠️ k = 10 is a common choice for us.

3. **leave-one-out cross validation**

   Just let k = n, which is the number of all the samples we have, we are applying the leave-one-out cross validation.

Bayesian statistics and regularization

Most times, we assume that the parameters θ to be a constant which just happen to be unknown for us. This is the typical *frequentist statistics* point of view, while in *the Bayesian statistics* point of view, θ is no longer some "unknown constant", it's a variable that follow some specific distribution.

In this approach, we would specify a prior distribution $p(\theta)$ on θ that expresses our "prior beliefs" about the parameters. Given a training set $S = \{(x^{(i)}, y^{(i)})\}_{i=1}^n$, when we are

asked to make a prediction on a new value of x, we can then compute the posterior distribution on the parameters

$$p(\theta \mid S) = \frac{p(S \mid \theta)p(\theta)}{p(S)} = \frac{p(\theta)\prod_{i=1}^{n} p(y^{(i)} \mid x^{(i)}, \theta)}{\int_{\theta} \left(\prod_{i=1}^{n} p(y^{(i)} \mid x^{(i)}, \theta)p(\theta)\right) d\theta}$$

In the equation above, $p(y^{(i)}|x^{(i)}, \theta)$ comes from whatever model you're using, like if you're using Bayesian logistic regression then $p(y^{(i)} \mid x^{(i)}, \theta) = h_\theta(x^{(i)})^{y^{(i)}} \left(1 - h_\theta(x^{(i)})\right)^{(1-y^{(i)})}$. Note that here we're using x, θ rather than $x; \theta$ because θ is now a variable.

When we are given a new test example xand asked to make it prediction on it, we can compute our posterior distribution on the class label using the posterior distribution on θ

$$p(y \mid x, S) = \int p(y \mid x, \theta)p(\theta \mid S)\, d\theta$$

Thus, for example, if the goal is to the predict the expected value of y given x, then we would output

$$E[y \mid x, S] = \int y\, p(y \mid x, S)\, dy$$

The procedure that we've outlined here can be thought of as doing "fully Bayesian" prediction, where our prediction is computed by taking an average with respect to the posterior $p(\theta|S)$ over θ. However, this could be computationally very difficult in real time. Thus, in practice we will instead approximate the posterior distribution for θ. One common approximation is to replace our posterior distribution for θ (as in Equation 9.4) with a single point estimate. *The MAP (maximum a posteriori)* estimate for θ is given by

$$\theta_{\text{MAP}} = \arg\max_{\theta} \prod_{i=1}^{n} p(y^{(i)} \mid x^{(i)}, \theta)p(\theta)$$

In practical applications, a common choice for the prior $p(\theta)$ is to assume that $\theta \sim N(0, \tau^2 I)$. Using this choice of prior, the fitted parameters θ_{MAP} will have smaller norm than that selected by maximum likelihood. In practice, this causes the Bayesian MAP estimate to be less susceptible to overfitting than the ML estimate of the parameters.

⚠️ Explaining the last eqaution

Why the MAP estimate should be $\theta_{\text{MAP}} = \arg\max_\theta \prod_{i=1}^{n} p(y^{(i)} \mid x^{(i)}, \theta)p(\theta)$?

Given

$$p(\theta \mid S) = \frac{p(S \mid \theta)p(\theta)}{p(S)}$$

and the $p(S)$ has noting todo with the θ, so what we try to maximize is actually

$$\theta_{\text{MAP}} = \arg\max_\theta p(S \mid \theta)p(\theta)$$

If we discompose $p(S \mid \theta)$ into $p(S \mid \theta) = \prod_{i=1}^{n} p(y^{(i)} \mid x^{(i)}, \theta)$ because (x, y) are idd, we have

$$\theta_{\text{MAP}} = \arg\max_\theta \prod_{i=1}^{n} p(y^{(i)} \mid x^{(i)}, \theta)p(\theta)$$

Unsupervised Learning

Clustering algorithms

Prototype-based clustering

Prototype-based clustering assumes that a clustering structure can be characterized by a set of *prototypes*, making it highly applicable in real-world clustering tasks. Generally, these algorithms begin by initializing the prototypes, then iteratively update them to find an optimal solution. Different methods of representing prototypes and updating them result in various clustering algorithms. We will introduce several well-known prototype-based clustering algorithms here.

k-means algorithm

The most common algorithm to play clustering is *k-means*, here in this section we'll introduce it. Notice that no we're talking about unsupervised learning, we have a set data that the labels $y^{(i)}$is not given.

For the set $\{x^{(1)}, \cdots, x^{(i)}, \cdots\}$, we first randomly choose k points as *the cluster centroids*, we denote them as $\mu_1, \mu_2 \ldots \mu_k$, and we can now apply the following algorithm

---

1. Initialize cluster centroids $\mu_1, \mu_2, ..., \mu_k \in \mathbb{R}^d$ randomly.

2. Repeat until convergence: $\{$

For every i, set

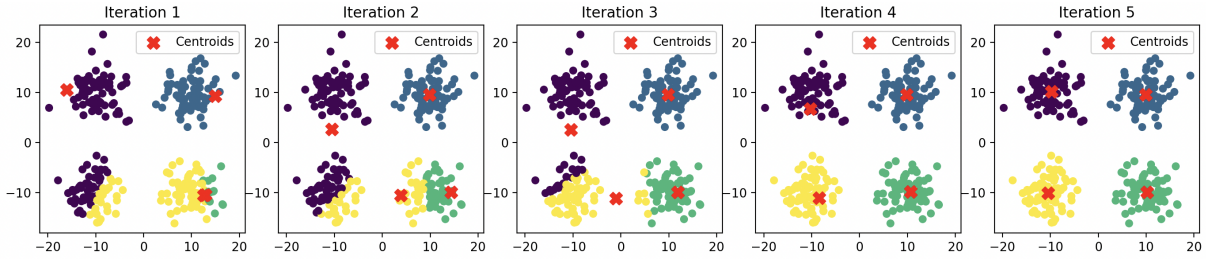
$$c^{(i)} := arg\min_j ||x^{(i)} - \mu_j||^2$$

For each j, set

$$\mu_j := \frac{\sum_{i=1}^{n} 1\{c(i) = j\} x^{(i)}}{\sum_{i=1}^{n} 1\{c(i) = j\}}$$

}

That will be look like this.



Is the k-means algorithm guaranteed to converge? Yes it is, in a certain sense. In particular, let us define *the distortion function* to be:

$$J(c, \mu) = \sum_{i=1}^{n} ||x^{(i)} - \mu_{c^{(i)}}||^2$$

It can be shown that k-means is exactly coordinate descent on J. Specifically, the inner-loop of k-means repeatedly minimizes J with respect to c while holding μ fixed, and then minimizes J with respect to μ while holding c fixed.

The distortion function J is a non-convex function, and so coordinate descent on J is not guaranteed to converge to the global minimum. In other words, k-means can be susceptible to local optima. Very often k-means will work fine and come up with very good clusterings despite this. But if you are worried about getting stuck in bad local minima, one common thing to do is run k-means many times.

Learning vector quantization

Learning vector quantization(LVQ) is quite familiar with k-means, yet it deals with another kind of data where all samples from our dataset X is labeled with y, that is we are actually dealing with sets like $(x^{(i)}, y^{(i)})$ right now. This makes LVQ a kind of supervised learning algorithm indeed.

Consider a dataset that contains m samples, and each of the sample $x^{(i)}$ has a label $y^{(i)}$, the total kind of labels is given by q, and now it's our job to find a centroid for each kind, that is to say we need to find a set $(p_1, p_2, \cdots p_q)$ where p_i is the centroid for the i-th kind. Like in k-means, we

can randomly initialize a set of prototypes from each kind of samples, we denote them as $p_1, p_2, \cdots p_q$, and we can now apply the following algorithm:

---

1. Initialize cluster centroids $p_1, p_2, \cdots p_q \in \mathbb{R}^d$ randomly.

2. Repeat until convergence: $\{$

$$\text{Randomly pick } (x^{(j)}, y^{(j)}) \text{ and calculate the distance between } (x^{(j)}, y^{(j)}) \text{ and } p_i$$

$$d_{ji} = ||x^{(j)} - p_i||_2$$

$$\text{Denote } i^* = \arg\min_{i \in \{1,2,3 \ldots q\}} d_{ji}, \text{ that is the closest centroid to } x^{(j)}, \text{ we then do}$$

$$\begin{cases} \text{if } t_{i^*} = y^{(j)}, p_{i^*} := p_{i^*} + \eta(x^{(j)} - p_{i^*}) \\ \\ \text{if } t_{i^*} \neq y^{(j)}, p_{i^*} := p_{i^*} - \eta(x^{(j)} - p_{i^*}) \end{cases}$$

$\}$

That is to say, if the closest prototype p_{i^*} haas the same type as $x^{(j)}$, then we set p_{i^*} a bit closer to the $x^{(j)}$, otherwise we do the opposite.

After learning a set of prototype vectors $(p_1, p_2, \ldots, p_q)$, the sample space X can be partitioned into clusters. Any sample $x^{(i)}$ will belong to the cluster of the nearest prototype vector. Each prototype vector p_i defines a region R_i in which every sample is closer to p_i than to any other prototype. This concept, called **vector quantization**, allows for **lossy compression** by representing all samples in R_i with p_i. I.e. we can compress the dimension of our input X, but only at some cost for there is always a bit difference between $x^{(i)}$ and it's corresponding p_i. This partitioning of X is also known as *Voronoi tessellation*.

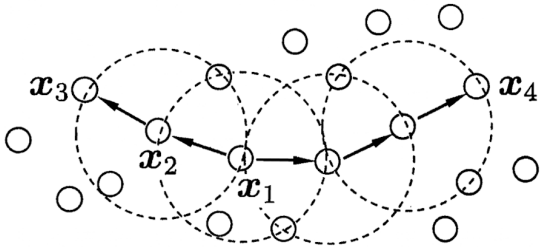
Density-based clustering*

Now, consider a scenario where we have a wide range of scattered discrete points with an uneven distribution. In some areas, points tend to cluster closely, while in others, points are sparsely distributed. It's intuitive that we could use the density of points to cluster them. Specifically, if there exists an area V where the point density exceeds a certain threshold, then V can naturally be considered a cluster within the given point set S.

Here we introduce the most common density-based clustering algorithm——*DBSCAN(Density-Based Spatial Clustering of Applications with Noise)*. To apply this algorithm, we need to define a few properties of the point density and strictly express what when instinctively feel in a mathematical way.

- **Core object**: Let p_{min} be the threshold we set, and ϵ be the range we set, then a point x_i is a core object if the amount of the points within x_i's ϵ-neighborhood is no less than p_{min}, i.e. $N_\epsilon(x_i) \geq p_{min}$.

- **Directly density-reachable**: If $x_j \in N_\epsilon(x_i)$, then x_j is directly density-reached by x_i.
- **Density-reachable**: x_j is density-reachable for x_i if we can find a series of points $p_{i1}, p_{i2}, \cdots, p_{im}$, that p_{ik} is directly density reachable for p_{ik+1} and x_i, x_j at two ends of the point serie.
- **Density connected**: x_i and x_j is density connected if there exists some x_k, that x_i and x_j are both density reachable by x_k.



x_1 is a core object, x_2 is density-reachable from x_1, x_3 is density-reachable from x_2, and x_3 is density-connected with x_4.

You should note that reachability does not hold symmetry, but connectivity does hold symmetry.

So now, our algorithm can be put it this way: we need to find for our given parameters ϵ and p_{min}, we want to find a subset that satisfies:

1. connectivity: $x_i \in C, x_j \in C \implies x_i$ is density-connected with x_j.
2. maximality: $x_i \in C, x_j$ is density-reachable from $x_i \implies x_j \in C$.

It will be of no hard to prove that any subset satisfies these two properties is a cluster of points set C.

So what we do now is:

DBSCAN

1. For all x in C, find all possible core objects, and denote them as $P = \{p_1, p_2, \cdots, p_m\}$.
2. Initialize unvisited sample set: r = all samples

   while $P \neq \emptyset$ do

   Record current unvisited samples: $r_{old} = r$

   Randomly select a core object p_i from P, initialize queue $Q =< p_i >$, and remove p_i from r

   while $Q \neq \emptyset$ do

   Get the first sample q from Q

   if $N_\epsilon(q) \geq p_{min}$ then set $\Delta = N_\epsilon(q) \cap r$, and add samples in Δ to Q, at last remove Δ from r

   end while

   $k = k + 1$, create cluster $C_k = r_{old}/r$ (romove r) and let $P = P/C_k$

   end while
3. And the set $\{C_1, C_2, \cdots, C_K\}$ is what we need.

EM algorithms

EM for mixture of Gaussians

Consider a situation, where we have a set of $\{x^{(1)} \dots x^{(n)}\}$, suppose we have (or may we say "guess") some $z^{(i)}$, so that for each joint distribution $p(x^{(i)}, z^{(i)}) = p(x^{(i)}|z^{(i)})p(z^{(i)})$, and we also wish that the $x^{(i)}|z^{(i)} = j \sim \mathcal{N}(\mu_j, \Sigma_j)$, and $z^{(i)}$ follows some multinomial where $p(z^{(i)} = j)$ is given by ϕ_j. Thus, our model posits that each $x^{(i)}$ was generated by randomly choosing $z^{(i)}$ from $\{1, ..., k\}$, and then $x^{(i)}$ was drawn from one of k Gaussians depending on $z^{(i)}$. This is called *the mixture of Gaussians model*. Note that here z is what we know as *latent random variables.*

The parameters of our model are thus ϕ, μ and Σ, especially we call ϕ *the mixing coefficients*, which represent the proportion of each Gaussian component in the overall mixture. To estimate them, we can write down the likelihood of our data:

$$\begin{aligned}\ell(\varphi, \mu, \Sigma) &= \sum_{i=1}^{n} \log p(x^{(i)}, \phi, \mu, \Sigma) \\ &= \sum_{i=1}^{n} \log \sum_{z^{(i)}=1}^{k} p(x^{(i)}|z^{(i)};, \mu, \Sigma) p(z^{(i)}; \phi)\end{aligned}$$

Note that here in the equation the $z^{(i)} = 1$ means we choose the first Gaussian that $z^{(i)}$ picks, that means we want a weighted average of the probability of which Gaussian did $x^{(i)}$ come from.

However, if we set to zero the derivatives of this formula with respect to the parameters and try to solve, we'll find that it is not possible to find the maximum likelihood estimates of the parameters in closed form.

The random variables $z^{(i)}$ indicate which of the k Gaussians each $x^{(i)}$ had come from. Note that if we knew what the $z^{(i)}$'s were, the maximum likelihood problem would have been easy. Specifically, we could then write down the likelihood as (i.e. we already know which Gaussian did x come from)

$$\ell(\varphi, \mu, \Sigma) = \sum_{i=1}^{n} \log p(x^{(i)}|z^{(i)}; \mu, \Sigma) + \log p(z^{(i)}; \phi)$$

As before, we could have

$$\begin{aligned}\phi_j &= \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}\{z^{(i)} = j\} \\ \mu_j &= \frac{\sum_{i=1}^{n} \mathbf{1}\{z^{(i)} = j\} x^{(i)}}{\sum_{i=1}^{n} \mathbf{1}\{z^{(i)} = j\}} \\ \Sigma_j &= \frac{\sum_{i=1}^{n} \mathbf{1}\{z^{(i)} = j\} (x^{(i)} - \mu_j)(x^{(i)} - \mu_j)^T}{\sum_{i=1}^{n} \mathbf{1}\{z^{(i)} = j\}}\end{aligned}$$

Indeed, we see that if the $z^{(i)}$'s were known, then maximum likelihood estimation becomes nearly identical to what we had when estimating the parameters of the Gaussian discriminant analysis model, except that here the $z^{(i)}$'s playing the role of the class labels.

However, in our density estimation problem, the $z^{(i)}$'s are not known. What can we do?

The EM algorithm is an iterative algorithm that has two main steps. Applied to our problem, in the E-step, it tries to "guess" the values of the $z^{(i)}$'s. In the M-step, it updates the

parameters of our model based on our guesses. Since in the M-step we are pretending that the guesses in the first part were correct, the maximization becomes easy. Here's the algorithm:

$$
\begin{aligned}
&\text{Repeat until convergence:}\{ \\
&\quad \text{(E-step) For each } i, j, \text{set} \\
&\qquad w_j^{(i)} := p(z^{(i)} = j \mid x^{(i)}; \phi, \mu, \Sigma) \\
&\quad \text{(M-step) Update the parameters:} \\
&\qquad \phi_j := \frac{1}{n} \sum_{i=1}^{n} w_j^{(i)} \\
&\qquad \mu_j := \frac{\sum_{i=1}^{n} w_j^{(i)} x^{(i)}}{\sum_{i=1}^{n} w_j^{(i)}} \\
&\qquad \Sigma_j := \frac{\sum_{i=1}^{n} w_j^{(i)} (x^{(i)} - \mu_j)(x^{(i)} - \mu_j)^T}{\sum_{i=1}^{n} w_j^{(i)}}
\end{aligned}
$$

In the E-step, we calculate the posterior probability of our parameters the $z^{(i)}$'s, given the $x^{(i)}$ and using the current setting of our parameters. I.e., using Bayes rule, we obtain:

$$
p(z^{(i)} = j \mid x^{(i)}; \phi, \mu, \Sigma) = \frac{p(x^{(i)} \mid z^{(i)} = j; \mu, \Sigma) p(z^{(i)} = j; \phi)}{\sum_{l=1}^{k} p(x^{(i)} \mid z^{(i)} = l; \mu, \Sigma) p(z^{(i)} = l; \phi)}
$$

Here, $p(x^{(i)}|z^{(i)} = j; \mu, \Sigma)$ is given by evaluating the density of a Gaussian with mean μ_j and covariance Σ_j at $x^{(i)}$; $p(z^{(i)} = j; \phi)$ is given by ϕ_j, and so on. The values $w_j^{(i)}$ calculated in the E-step represent our "soft" guesses for the values of $z^{(i)}$.

Similar to K-means, it is also susceptible to local optima, so reinitializing at several different initial parameters may be a good idea. It's clear that the EM algorithm has a very natural interpretation of repeatedly trying to guess the unknown $z^{(i)}$'s; but how did it come about, and can we make any guarantees about it, such as regarding its convergence? In the following chapter we will see, it's convergence is well guaranteed.

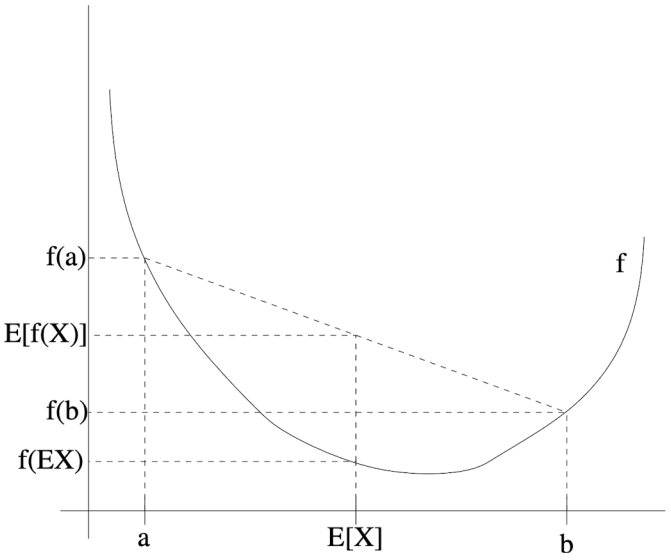
Jensen's inequality

Let f be a function whose domain is the set of real numbers. Recall that f is a convex function if $f''(x) \geq 0$ (for all $x \in R$). In the case of f taking vector-valued inputs, this is generalized to the condition that its hessian H is positive semi-definite ($H \geq 0$). If $f''(x) > 0$ for all x, then we say f is strictly convex (in the vector-valued case, the

corresponding statement is that H must be positive definite, written $H > 0$). Jensen's inequality can then be stated as follows:

Theorem: Let f be a convex function, and let X be a random variable. Then:

$$f[\mathrm{E}[X]] \leq \mathrm{E}[f(X)]$$



If the function happened to be concave, you can just inverse the inequality.

General EM algorithms

Suppose we have an estimation problem in which we have a training set $\{x^{(1)}, ..., x^{(n)}\}$ consisting of n independent examples. We have a latent variable model $p(x, z; \theta)$ with z being the latent variable (which for simplicity is assumed to take finite number of values). The density for x can be obtained by marginalized over the latent variable z:

$$p(x^{(i)}; \theta) = \sum_z p(x^{(i)}, z^{(i)}; \theta)$$

We wish to fit the parameters θ by maximizing the log-likelihood of the data, defined by

$$\ell(\theta) = \sum_{i=1}^{n} \log p(x^{(i)}; \theta)$$

Now we can rewrite this as

$$\ell(\theta) = \sum_{i=1}^{n} \log \sum_{z} p(x^{(i)}, z^{(i)}; \theta)$$

But, explicitly finding the maximum likelihood estimates of the parameters θ may be hard since it will result in difficult non-convex optimization problem. Here, the $z^{(i)}$'s are the latent random variables; and it is often the case that if they were observed, then maximum likelihood estimation would be easy. In such a setting, the EM algorithm gives an efficient method for maximum likelihood estimation.

Let's first consider maximize with only one $x^{(i)}$, calling it x, then we shall have

$$\log p(x; \theta) = \sum_{z} \log p(x, z; \theta)$$

Let Q be a distribution over the possible values of z. (That is, $\sum_z Q(z) = 1, Q(z) \geq 0$).

Now we have

$$\begin{aligned} \log p(x; \theta) &= \log \sum_{z} p(x, z; \theta) \\ &= \log \sum_{z} \frac{Q(z)}{Q(z)} p(x, z; \theta) \\ &\geq \sum_{z} Q(z) \log \frac{p(x, z; \theta)}{Q(z)} \end{aligned} \tag{11.1}$$

Here for the precise steps, read the handscript I wrote which I appended later.

Now, for any distribution Q, the formula gives a lower-bound on $\log p(x; \theta)$. There are many possible choices for the Q's. Which should we choose? Well, if we have some current guess θ of the parameters, it seems natural to try to make the lower-bound tight at that value of θ. I.e., we will make the inequality above hold with equality at our particular value of θ.

To do so, we can find that what we need to do is just to set

$$Q(z) = p(z|x; \theta)$$

For convenience, we call the expression in Equation (11.1) *the evidence lower bound (ELBO)* and we denote it by

$$\text{ELBO}(x; Q, \theta) = \sum_z Q(z) \log \frac{p(x, z; \theta)}{Q(z)} \tag{11.2}$$

With this equation, we can re-write equation (11.1) as

$$\forall Q, \theta, x, \quad \log p(x; \theta) \geq \text{ELBO}(x; Q, \theta) \tag{11.3}$$

Intuitively, the EM algorithm alternatively updates $Q$and θ by a) setting $Q(z) = p(z|x; \theta)$ following the maximization result so that $\text{ELBO}(x; Q, \theta) = \log p(x; \theta)$ for xand the current θ, and b) maximizing $\text{ELBO}(x; Q, \theta)$ w.r.t θ while fixing the choice of Q.

Recall that all the discussion above was under the assumption that we aim to optimize the log-likelihood log $p(x; \theta)$ for a single example x. It turns out that with multiple training examples, the basic idea is the same and we only needs to take a sum over examples at relevant places. Next, we will build the evidence lower bound for multiple training examples and make the EM algorithm formal.

Recall we have a training set $\{x^{(1)}, ..., x^{(n)}\}$. Note that the optimal choice of Q is $p(z|x; \theta)$, and it depends on the particular example x. Therefore here we will introduce n distributions $Q_1, ..., Q_n$, one for each example $x^{(i)}$. For each example $x^{(i)}$, we can build the evidence lower bound

$$\begin{aligned} \log p(x^{(i)}; \theta) &\geq \text{ELBO}(x^{(i)}; Q_i, \theta) \\ &= \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})} \end{aligned}$$

Taking sum over all the examples, we obtain a lower bound for the log-likelihood

$$\begin{aligned} \ell(\theta) &\geq \sum_i \text{ELBO}(x^{(i)}; Q_i, \theta) \\ &= \sum_i \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})} \end{aligned} \tag{11.4}$$

Like before, the Q_i that attains equality satisfies

$$Q_i(z^{(i)}) = p(z^{(i)}|x^{(i)}; \theta)$$

Now, for this choice of the Q_i's, Equation (11.11) gives a lower-bound on the log-likelihood ℓ that we're trying to maximize. This is the E-step. In the M-step of the algorithm, we then

maximize our formula in Equation (11.11) with respect to the parameters to obtain a new setting of the θ's. Repeatedly carrying out these two steps gives us the EM algorithm, which is as follows:

$$
\begin{aligned}
&\text{Repeat until convergence:}\{ \\
&\quad \text{(E-step) For each } i, j, \text{set} \\
&\qquad Q_i(z^{(i)}) := p(z^{(i)} \mid x^{(i)}; \theta) \\
&\quad \text{(M-step) Update the parameters:} \\
&\qquad \theta := \arg\max_{\theta} \sum_{i=1}^{n} \text{ELBO}(x^{(i)}; Q_i, \theta) \\
&\qquad\quad = \arg\max_{\theta} \sum_{i} \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})} \\
&\}
\end{aligned}
$$

How do we know if this algorithm will converge? Well, suppose $\theta^{(t)}$ and $\theta^{(t+1)}$ are the parameters from two successive iterations of EM. We will now prove that $\ell(\theta^{(t)}) \leq \ell(\theta^{(t+1)})$, which shows EM always monotonically improves the log-likelihood. The key to showing this result lies in our choice of the Q_i's. Specifically, on the iteration of EM in which the parameters had started out as $\theta^{(t)}$, we would have chosen $Q_i^{(t)}(z^{(i)}) := p(z^{(i)}|x^{(i)}; \theta^{(t)})$. We saw earlier that this choice ensures that Jensen's inequality, as applied to get Equation (11.4), holds with equality, and hence

$$
\ell(\theta^{(t)}) = \sum_{i=1}^{n} \text{ELBO}(x^{(i)}; Q_i^{(t)}, \theta^{(t)})
$$

The parameters $\theta^{(t+1)}$ are then obtained by maximizing the right hand side of the equation above. Thus,

$$
\begin{aligned}
\ell(\theta^{(t+1)}) &\geq \sum_{i=1}^{n} \text{ELBO}(x^{(i)}; Q_i^{(t)}, \theta^{(t+1)}) \quad \text{(because inequality (11.11) holds f} \\
&\geq \sum_{i=1}^{n} \text{ELBO}(x^{(i)}; Q_i^{(t)}, \theta^{(t)}) \quad \text{(see reason below)} \\
&= \ell(\theta^{(t)}) \quad \text{(by the last equation)}
\end{aligned}
$$

where the last inequality follows from that $\theta^{(t+1)}$ is chosen explicitly to be

Now we have that $x^{(i)} \sim$ some distribution, and we set a latent $z^{(i)}$, so that $p(x;\theta)$ can be obtained with $P(x^{(i)};\theta) = \sum_{z^{(i)}} P(x^{(i)}, z^{(i)};\theta)$, so we can denote the $l(\theta)$ as $l(\theta) = \prod_{i=1}^{n} P(x^{(i)};\theta) = \prod_{i=1}^{n} \sum_{z^{(i)}} P(x^{(i)}, z^{(i)};\theta)$

the log form would be $l(\theta) = \sum_{i=1}^{n} \log \sum_{z^{(i)}} P(x^{(i)}, z^{(i)};\theta)$ let Q be a distribution over z. $\sum_z Q(z) = 1$

Consider only one x, we have

$$l(\theta) = \log \sum_z P(x,z;\theta) = \log \sum_z Q(z) \frac{P(x,z;\theta)}{Q(z)}$$ given this could be seen as the expectation of $\frac{P(x,z;\theta)}{Q(z)}$ given $Q(z)$,

we could use Jensen's that $f(Ex) \geq E[f(x)]$ so $\log\left(Q(z) \sum_z \frac{P(x,z;\theta)}{Q(z)}\right) \geq \sum_z Q(z) \log \frac{P(x,z;\theta)}{Q(z)}$.

Now that we have $l(\theta) \geq \sum_z Q_z \log \frac{P(x,z;\theta)}{Q(z)}$, the right term is what we known as the ELBO.

As we know, to reach the equaty of Jensen's, we must let $E(x) =$ constant

so $\frac{P(x,z;\theta)}{Q(z)} = C$ that is to say $Q(z) \propto P(x,z;\theta)$ Given $\sum_z Q(z) = 1$ we can have

$\frac{\sum_z P(x,z;\theta)}{\sum_z Q(z)} = C$, so further we know $Q(z) = \frac{P(x,z;\theta)}{C} = \frac{P(x,z;\theta)}{\sum_z P(x,z;\theta)} = \frac{P(z|x;\theta)P(x;\theta)}{P(x;\theta)} = P(z|x;\theta)$

So to maximize our ELBO w.r.t Q, we must have $Q(z) = P(z|x;\theta)$

With this, now we have that $l(\theta) = \log P(x;\theta) \geq \sum_z Q(z) \log \frac{P(x,z;\theta)}{Q(z)} = ELBO(x, Q, \theta)$

Now consider n $x^{(i)}$ and n corresponding Q_i, we have $\log P(x^{(i)};\theta) \geq ELBO(x^{(i)}, Q_i, \theta)$

Put together we get $l(\theta) = \sum_i \log P(x^{(i)};\theta) \geq \sum_i \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{P(x^{(i)}, z^{(i)};\theta)}{Q_i(z^{(i)})}$ where we also attain equality when

$Q_i = P(z^{(i)}|x^{(i)};\theta)$.

And for our transformation we have $ELBO(x^{(i)}, Q_i, \theta) = \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{P(x^{(i)}, z^{(i)};\theta)}{Q_i(z^{(i)})} = \sum_{z^{(i)}} Q_i(z^{(i)}) \log P(x^{(i)}, z^{(i)};\theta) - \sum_{z^{(i)}} Q_i(z^{(i)}) \log Q_i(z^{(i)})$ (entropy of $Q(z)$)

$= E_{z\sim Q}(\log P(x^{(i)}, z^{(i)};\theta)) - E_{z\sim Q}(\log Q_i(z^{(i)}))$ Denote P_z as the marginal distribution of z under $P(x^{(i)}, z^{(i)};\theta)$ where we

have $P_z(z^{(i)}) = \sum_x P(x^{(i)}, z^{(i)};\theta) = \sum_x P(z^{(i)}|x^{(i)};\theta) P(x^{(i)};\theta)$. So the former equation now $\sum_{z^{(i)}} Q_i(z^{(i)}) \log P(x^{(i)}|z^{(i)};\theta) + \sum_{z^{(i)}} Q_i(z^{(i)}) \log P(z^{(i)};\theta) - \sum_{z^{(i)}} Q_i(z^{(i)}) \log Q_i(z^{(i)})$

By introducing KL divergence where $D_{KL}(Q\|P_z) = \sum_z Q(z) \log \frac{Q(z)}{P_z}$, we notice that $\sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{P(z^{(i)};\theta)}{Q_i(z^{(i)})} = -\sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{Q_i(z^{(i)})}{P(z^{(i)};\theta)}$

now we have $ELBO(x^{(i)}, Q_i, \theta) = E_{z\sim Q}[x^{(i)}|z^{(i)};\theta] - D_{KL}(Q\|P_z)$

Note that $P(x^{(i)};\theta) = \frac{P(x^{(i)}, z^{(i)};\theta)}{P(z^{(i)}|x^{(i)};\theta)} = \frac{P(x^{(i)}|z^{(i)};\theta)P(z^{(i)};\theta)}{P(z^{(i)}|x^{(i)};\theta)}$, so the equation equals to

$$\sum_{z^{(i)}} Q_i(z^{(i)}) \log P(x^{(i)}, z^{(i)};\theta) - \sum_{z^{(i)}} Q_i(z^{(i)}) \log P(z^{(i)}|x^{(i)};\theta) - \sum_{z^{(i)}} Q_i(z^{(i)}) \log Q_i(z^{(i)}) + \sum_{z^{(i)}} Q_i(z^{(i)}) \log P(z^{(i)}|x^{(i)};\theta) = \underbrace{E_{z\sim Q}[x^{(i)};\theta]}_{\downarrow \log P(x;\theta)} - D_{KL}(Q\|P_{z|x})$$

$$\arg\max_{\theta} \sum_{i=1}^{n} \mathrm{ELBO}(x^{(i)}; Q_i^{(t)}, \theta)$$

Hence, EM causes the likelihood to converge monotonically. Given the result that we just showed, one reasonable convergence test would be to check if the increase in $\ell(\theta)$ between successive iterations is smaller than some tolerance parameter, and to declare convergence if EM is improving $\ell(\theta)$ too slowly.

The EM can also be viewed an alternating maximization algorithm on $\mathrm{ELBO}(Q, \theta)$, in which the E-step maximizes it with respect to Q(check this yourself), and the M-step maximizes it with respect to θ.(Note we need to overload ELBO(·) with $\mathrm{ELBO}(Q, \theta) = \sum_{i=1}^{n} \mathrm{ELBO}(x^{(i)}; Q_i, \theta)$)

There are some other interpretation for ELBO(·), such as

$$\begin{aligned}\mathrm{ELBO}(x; Q, \theta) &= \mathbb{E}_{z\sim Q}[\log p(x, z; \theta)] - \mathbb{E}_{z\sim Q}[\log Q(z)] \\ &= \mathbb{E}_{z\sim Q}[\log p(x \mid z; \theta)] - D_{\mathrm{KL}}(Q \| p_z)\end{aligned}$$

where we use p_z to denote the marginal distribution of z (under the distribution $p(x, z; \theta)$)

The *KL divergence* is given by

$$D_{\mathrm{KL}}(Q \| p_z) = \sum_{z} Q(z) \log \frac{Q(z)}{p(z)}$$

Another form is

$$\mathrm{ELBO}(x; Q, \theta) = \log p(x) - D_{\mathrm{KL}}(Q \| p_{z|x})$$

Mixture of Gaussians revisited

Armed with our general definition of the EM algorithm, let’s go back to our old example of fitting the parameters ϕ, μ and Σ in a mixture of Gaussians. For the sake of brevity, we carry out the derivations for the M-step updates
only for
ϕ and μ_j, and leave the updates for Σ_j as an exercise for the reader.

The E-step is easy. Following our algorithm derivation above, we simply calculate

$$w_j^{(i)} = Q_i(z^{(i)} = j) = p(z^{(i)} = j \mid x^{(i)}; \phi, \mu, \Sigma)$$

Here, "$Q_i(z^{(i)} = j)$" denotes the probability of $z^{(i)}$ taking the value j under the distribution Q_i.

Next, in the M-step, we need to maximize, with respect to our parameters ϕ, μ, Σ, the quantity

$$\begin{aligned}
&\sum_{i=1}^{n} \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \phi, \mu, \Sigma)}{Q_i(z^{(i)})} \\
&= \sum_{i=1}^{n} \sum_{j=1}^{k} Q_i(z^{(i)} = j) \log \frac{p(x^{(i)} \mid z^{(i)} = j; \mu, \Sigma)\, p(z^{(i)} = j; \phi)}{Q_i(z^{(i)} = j)} \\
&= \sum_{i=1}^{n} \sum_{j=1}^{k} w_j^{(i)} \left[\log \frac{\left(\frac{1}{(2\pi)^{d/2} |\Sigma_j|^{1/2}} \exp\left(-\frac{1}{2} (x^{(i)} - \mu_j)^T \Sigma_j^{-1} (x^{(i)} - \mu_j) \right) \cdot \phi_j \right)}{w_j^{(i)}} \right]
\end{aligned}$$

Let's maximize this with respect to μ_l. If we take the derivative with respect to μ_l, we find

$$\nabla_{\mu_l} \text{quantity} = \sum_{i=1}^{n} w_l^{(i)} \left[\Sigma_l^{-1} x^{(i)} - \Sigma_l^{-1} \mu_l \right]$$

Setting this to zero and solving for μ_l therefore yields the update rule

$$\mu_l := \frac{\sum_{i=1}^{n} w_l^{(i)} x^{(i)}}{\sum_{i=1}^{n} w_l^{(i)}}$$

which was what we had in the previous set of notes.

Let's do one more example, and derive the M-step update for the parameters ϕ_j. Grouping together only the terms that depend on ϕ_j, we find that we need to maximize

$$\sum_{i=1}^{n} \sum_{j=1}^{k} w_j^{(i)} \log \phi_j$$

However, there is an additional constraint that the ϕ_j's sum to 1, so we need to construct Lagrangian

$$L(\phi) = \sum_{i=1}^{n} \sum_{j=1}^{k} w_j^{(i)} \log \phi_j + \beta \left(\sum_{j=1}^{k} \phi_j - 1 \right)$$

We therefore have our M-step updates for the parameters ϕ_j

$$\phi_j := \frac{\sum_{i=1}^{n} w_j^{(i)}}{n}$$

This means that the ϕ_j comes from an average of n possible Q_i distributions.

And we do the same thing to w.r.t Σ_l, we get

$$\Sigma_j = \frac{\sum_{i=1}^{n} w_j^{(i)} (x^{(i)} - \mu_j)^T (x^{(i)} - \mu_j)}{\sum_{i=1}^{n} w_j^{(i)}}$$

PCA and ICA

PCA

What is PCA (*principle component analysis*) you may wonder? Generally speaking, PCA is a method to apply a automatically feature engineering without any handcraft work or any proposed assumption. Consider a situation where we try to explore the relation between motorcycle and the expertise of the driver.

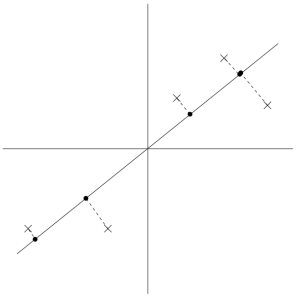
Since the situation is about motorcycle, we can suggest that we have two input variables which is x_1 that represents kilometers per hour and x_2 that represents mph (miles per hour). This is two variables as we can tell, and they have different values obviously. But here's the question: these two parameters are strongly connected—— you can convert them by simply multiply a constant!

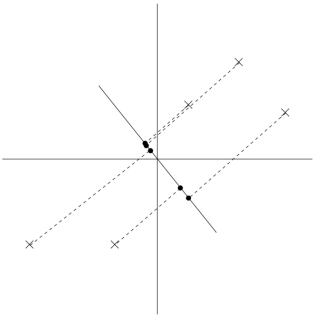
And it's intuitively for us to think that this could happen in some other situations, where some variables in our dataset are strongly correlated, and we can't always try to select and cross out these variables manually. Can we find a efficient way to find these variables automatically and create some new variables by presenting them in a new variable? The answer is yes. And that will be the main topic of this subsection.

But before we develop our PCA algorithm, we must apply normalization to our data so that they can have a mean of 0, a variance of 1. To do so, we apply the following normalization:

$$x_j^{(i)} \leftarrow \frac{x_j^{(i)} - \mu_j}{\sigma_j}$$

And we can start to develop our PCA algorithm. Suggest our latent main variable u on some axis which go through the origin. We can give two image to make what we are going to do more understandable.





The two figures above give us a clear image of how we can project the origin points(variable pairs) on to the new axis. But which one is a better axis? You may say the first one intuitively? But how can we quantify this?

Consider the projection of $x \in \mathbb{R}^2$ on the new axis u, take the unit vector u of it, and the projection of x on u is given by $x^T u$, and we can clearly see that we get a better axis when the variance of this projections is larger.

Thus we can get our problem here that is to maximize:

$$\begin{aligned}\frac{1}{n}\sum_{i=1}^{n}(x^{(i)T}u)^2 &= \frac{1}{n}\sum_{i=1}^{n}u^T x^{(i)}x^{(i)T}u \\ &= u^T(\frac{1}{n}\sum_{i=1}^{n}x^{(i)}x^{(i)T})u\end{aligned}$$

We easily recognize that the maximizing this subject to $||u||_2 = 1$ gives the principal eigenvector of $\Sigma = \frac{1}{n}\sum_{i=1}^{n} x^{(i)}x^{(i)T}$ (prove this with Lagrangian yourself), which is just the empirical covariance matrix of the data (assuming it has zero mean).

To summarize, we have found that if we wish to find a 1-dimensional subspace with with to approximate the data, we should choose u to be the principal eigenvector of Σ. More

generally, if we wish to project our data into a k-dimensional subspace ($k < d$), we should choose $u_1, ..., u_k$ to be the top k eigenvectors of Σ. The u_i's now form a new, orthogonal basis for the data.

Then, to represent $x^{(i)}$ in this basis, we need only compute the corresponding vector

$$y^{(i)} = \begin{bmatrix} u_1^T x^{(1)} \\ \vdots \\ u_k^T x^{(k)} \end{bmatrix} \in \mathbb{R}^k$$

PCA is therefore also referred to as a *dimensionality reduction algorithm*. The vectors $u_1, ..., u_k$ are called the *first k principal components* of the data.

PCA can also be derived by picking the basis that minimizes the approximation error arising from projecting the data onto the k-dimensional subspace spanned by them.

PCA has many applications. First, compression—representing $x^{(i)}$'s with lower dimension $y^{(i)}$'s—is an obvious application.

Another standard application is to preprocess a dataset to reduce its dimension before running a supervised learning learning algorithm with the $x^{(i)}$'s as inputs.

Apart from computational benefits, reducing the data's dimension can also reduce the complexity of the hypothesis class considered and help avoid overfitting. PCA can also be viewed as a denoising algorithm sometimes and holds a place in face recognition.

ICA

Now let's talk about ICA (Independent Components Analysis), which is quite similar to PCA in some ways, that is, we try to find a new basis in which to represent our data. However, the goal is very different.

Consider the classic "cocktail party problem". Here, d speakers are speaking simultaneously at a party, and any microphone placed in the room records only an overlapping combination of the dspeakers' voices. But lets say we have d different microphones placed in the room, and because each microphone is a different distance from each of the speakers, it records a different combination of the speakers' voices. Using these microphone recordings, can we separate out the original d speakers' speech signals?

To formalize this problem, we imagine that there is some data $s \in \mathbb{R}^d$ that is generated via d independent sources. What we observe is

$$x = As$$

Here A is what we know as the *mixing matrix*, and we don't know the exact value of it. Repeated observation gives us a dataset $\{x^{(i)}; i = 1, ..., n\}$, and our goal is to recover the sources $s^{(i)}$ that had generated our data ($x^{(i)} = As^{(i)}$). In our cocktail party problem, $s^{(i)}$ is an d-dimensional vector, and $s_j^{(i)}$ is the sound that speaker j was uttering at time i. Also, $x^{(i)}$ in an d-dimensional vector, and $x_j^{(i)}$ is the acoustic reading recorded by microphone $j$at time i.

Let $W = A^{-1}$ be the *unmixing matrix*. Our goal is to find W, so that given our microphone recordings x(i), we can recover the sources by computing $s^{(i)} = Wx^{(i)}$. Denote w_i^T as the i-th row of W, we can then use a convenient notation of $s_j^{(i)} = w_j^T x^{(i)}$.

ICA ambiguities

And now let's think about a question: to what extend can the W be recovered? It's inevitable that there must be some loss between the real audio and what we get as $s = W^{-1}x$, because part of the information must be lost during the recording and recovery!

So let's formulate this problem. Consider P be any d-by-d permutation matrix. This means that each row and each column of P has exactly one "1". You can see why this is a permutation matrix from the following example:

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Given only the $x^{(i)}$'s, there will be no way to distinguish between W and PW. Specifically, the permutation of the original sources is ambiguous, which should be no surprise. Fortunately, this does not matter for most applications. Further, it's easy to see that there is no way to recover the right scaling of $x^{(i)}$, i.e. if a single column of A were scaled by a factor of α, and the corresponding source were scaled by a factor of $1/\alpha$, then there is again no way to determine that this had happened given only the $x^{(i)}$'s. However, at least for the cocktail party problem, this doesn't matter as well for the scaling only affects the volume of the speech and nothing else.

Are these the only sources of ambiguity in ICA? It turns out that they are, so long as the sources s_i are non-Gaussian. To see why the data can't be Gaussian, we can consider an example in which $n = 2$, and $s \sim \mathcal{N}(0, I)$. Note here the I is the identity matrix.

Suppose we observe some $x = As$, then, the distribution of x will be Gaussian, $x \sim \mathcal{N}(0, AA^T)$, since:

–

$$E_{s\sim\mathcal{N}(0,I)}[x] = E[As] = AE[s] = 0AA^T$$

$$\begin{aligned} Cov[x] &= E_{s\sim\mathcal{N}(0,I)}[xx^T] = E[Ass^TA^T] \\ &= AE[ss^T]A^T = A\cdot Cov[s]\cdot A^T \end{aligned}$$

Note here we used our assumption $s \sim \mathcal{N}(0, I)$.

Now, let R be an arbitrary orthogonal (less formally, a rotation/reflection) matrix, so that $RR^T = R^TR = I$, and let $A' = AR$. Then if the data had been mixed according to A' instead of A, we would have instead observed $x' = A's$. The distribution of x' is also Gaussian, $x' \sim N(0, AA^T)$, since

$$\begin{aligned} E_{s\sim\mathcal{N}(0,I)}[x'(x')^T] &= E[A'ss^T(A')^T] \\ &= E[ARss^T(AR)^T] \\ &= ARR^TA^T = AA^T \end{aligned}$$

Hence, whether the mixing matrix is A or A', we would observe data from a $\mathcal{N}(0, AA^T)$ distribution. Thus, there is no way to tell if the sources were mixed using A and A'.

Our argument above was based on the fact that the multivariate standard normal distribution is rotationally symmetric. Despite the bleak picture that this paints for ICA on Gaussian data, it turns out that, so long as the data is not Gaussian, it is possible, given enough data, to recover the d independent sources.

Densities and linear transformations

Before moving on, we want to briefly talk about the effect of linear transformations on densities.

Suppose a random variable s is drawn according to some density $p_s(s)$. For simplicity, assume for now that $s \in \mathbb{R}$is a real number. Now, let the random variable x be defined according to $x = As$(here, $x \in \mathbb{R}, A \in \mathbb{R}$). Let p_x be the density of x. What is p_x?

Unlike what may come to your instinction, that

$$p_x(x) = p_s(Wx)$$

The real answer is

$$p_x(x) = p_s(Wx)\cdot |W|$$

You can prove this yourself by many means!

ICA algorithm

We are now ready to derive an ICA algorithm. We describe an algorithm by Bell and Sejnowski, and we give an interpretation of their algorithm as a method for maximum likelihood estimation. (This is different from their original interpretation involving a complicated idea called the infomax principal which is no longer necessary given the modern understanding of ICA.)

We suppose that the distribution of each source s_j is given by a density p_s, and that the joint distribution of the sources s is given by

$$p(s) = \prod_{j=1}^{d} p_s(s_j)$$

Note that by modeling the joint distribution as a product of marginals, we capture the assumption that the sources are independent. Using our formulas from the previous section, this implies the following density on $x = As = W^{-1}s$:

$$p(x) = \prod_{j=1}^{d} p_s(w_j^T x) \cdot |W|$$

Recall that, given a real-valued random variable z, its cumulative distribution function (CDF) F is defined by $F(z_0) = P(z \leq z_0) = \int_{-\infty}^{z_0} p_z(z)dz$ and the density is the derivative of the cdf: $p_z(z) = F'(z)$.

Thus, to specify a density for the s_i's, all we need to do is to specify some CDF for it.

Following our previous discussion, we cannot choose the Gaussian CDF, as ICA doesn't work on Gaussian data. What we'll choose instead as a reasonable "default" CDF that slowly increases from 0 to 1, is the sigmoid function $g(s) = 1/(1 + e^{-s})$. Hence, $p_s(s) = g'(s)$.

The square matrix W is the parameter in our model. Given a training set $\{x^{(i)}; i = 1, ..., n\}$, the log likelihood is given by

$$\ell(W) = \sum_{i=1}^{n} \Big(\sum_{j=1}^{d} \log g'(w_j^T x^{(i)}) + log|W| \Big)$$

We would like to maximize this in terms W. By taking derivatives and using the fact (from the first set of notes) that $\nabla_W |W| = |W|(W^{-1})^T$, we easily derive a stochastic gradient ascent learning rule. For a training example $x^{(i)}$, the update rule is:

$$W := W + \alpha \left(\begin{bmatrix} 1 - 2g(w_1^T x^{(i)}) \\ 1 - 2g(w_2^T x^{(i)}) \\ \vdots \\ 1 - 2g(w_d^T x^{(i)}) \end{bmatrix} x^{(i)T} + (W^T)^{-1} \right)$$

where α is the learning rate. After the algorithm converges, we then compute $s^{(i)} = Wx^{(i)}$ to recover the original sources.

> 📌 **Remark**
>
> When writing down the likelihood of the data, we implicitly assumed that the $x^{(i)}$'s were independent of each other (for different values of i; note this issue is different from whether the different coordinates of $x^{(i)}$ are independent), so that the likelihood of the training set was given by $\prod_i p(x^{(i)}; W)$. This assumption is clearly incorrect for speech data and other time series where the $x^{(i)}$'s are dependent, but it can be shown that having correlated training examples will not hurt the performance of the algorithm if we have sufficient data. However, for problems where successive training examples are correlated, when implementing stochastic gradient ascent, it sometimes helps accelerate convergence if we visit training examples in a randomly permuted order. (I.e., run stochastic gradient ascent on a randomly shuffled copy of the training set.)

Reinforcement Learning

Reinforcement learning

MDP

To fully describe a reinforcement learning process, we use something we know as *the Markov decision processes*. MDP is indeed a tuple of $(S, A, P_{sa}, \gamma, R)$, where this denotation indicates

1. S: a set of states we may step in.
2. A: a set of action we may take.
3. P_{sa}: are the state transition probabilities. For each state $s \in S$ and action $a \in A$, P_{sa} is a distribution over the state space. We'll say more about this later, but briefly, P_{sa} gives the distribution over what states we will transition to if we take action a in state s.
4. γ: γ is what we know as the discount rate, formally, is called the *discount factor*.
5. R: $S \times A \to R$ is the reward function. (Rewards are sometimes also written as a function of a state S only, in which case we would have $R : S \to R$).

The dynamics of an MDP proceeds as follows: We start in some state s_0, and get to choose some action $a_0 \in A$ to take in the MDP. As a result of our choice, the state of the MDP randomly transitions to some successor state s_1, drawn according to $s_1 \sim P_{s_0 a_0}$.

And by following this process, we get a complete and entire MDP. Upon visiting the sequence of states $s_0, s_1, ...$ with actions $a_0, a_1, ...$, our total payoff is given by a simple equation

$$R(s_0) + \gamma R(s_1) + \cdots \gamma^n R(s_n) + \cdots$$

And given that, our goal in reinforcement learning is now clear as all we need to do is maximize the expectation

$$\mathbb{E}[R(s_0) + \gamma R(s_1) + \cdots \gamma^n R(s_n) + \cdots]$$

Note that the reward at timestep t is discounted by a factor of γ_t. Thus, to make this expectation large, we would like to accrue positive rewards as soon as possible (and postpone negative rewards as long as possible).

A *policy* is any function $\pi : S \to A$ mapping from the states to the actions. We say that we are executing some policy π if, whenever we are in state s, we take action $a = \pi(s)$. We also define the *value function* for a policy π according to

$$V^{\pi}(s) = \mathbb{E}[R(s_0) + \gamma R(s_1) + \cdots \gamma^n R(s_n) + \cdots | s_0 = s, \pi]$$

Given a fixed policy π, its value function V^{π} satisfies the *Bellman equations*:

$$V^{\pi}(s) = R(s) + \gamma \sum_{s' \in S} p_{sa}(s') V^{\pi}(s')$$

This equation is clearly enough to be understood instantly. It consists of two parts. First, the *immediate reward* $R(s)$ that we get right away simply for starting in state s, and second, the expected sum of future discounted rewards. Examining the second term in more detail, we see that the summation term above can be rewritten $E_{s'} \sim P_s \pi(s)[V^{\pi}(s\prime)]$. This is the expected sum of discounted rewards for starting in state s', where s' is distributed according $P_s \pi(s)$, which is the distribution over where we will end up after taking the first action $\pi(s)$ in the MDP from state s.

Bellman's equations can be used to efficiently solve for V^{π}. Specifically, in a finite-state MDP ($|S| < \infty$), we can write down one such equation for $V^{\pi}(s)$ for every state s. This gives us a set of |S|linear equations in |S| variables (the unknown $V^{\pi}(s)$'s, one for each state), which can be efficiently solved for the $V^{\pi}(s)$'s.

Note here that we may need some specific linear equation set solver to help us with dealing a bunch of equations.

Now we can officially define **the optimal value function** as V^*, we can denote that

$$V^*(s) = R(s) + \max_{a \in A} \gamma \sum_{s' \in S} p_{sa}(s') V^{\pi}(s')$$

We could also define a π^* as

$$\pi^*(s) = arg \max_{a \in A} \gamma \sum_{s' \in S} p_{sa}(s') V^{\pi}(s')$$

Thus the π^* gives a action that will maximize the "max" term in the former equation.

It is a fact that for every state s and every policy π, we have

$$V^*(s) = V^{\pi^*}(s) \geq V(s)$$

In other words, π^* as defined in Equation is the optimal policy. Note that π^* has the interesting property that it is **the optimal policy** for all states s. That is to say, for all state s, maximize the second term attains π^*.

Value iteration and policy iteration

In this section we will introduce two kind of ways to learn a reinforcement learning agent. For now, we consider MDPs with finite states and finite action space only, and we also assume that we already know state transaction probabilities P_{sa} and the reward function R.

For the first algorithm, *the value iteration* we do

1. For each state s, initialize $V(s) := 0$
2. for until convergence do
3. For every state update

$$V(s) := R(s) + \max_{a \in A} \gamma \sum_{s'} P_{sa}(s') V(s')$$

This algorithm can be thought of as repeatedly trying to update the estimated value function using Bellman Equations.

There are two possible ways of performing the updates in the inner loop of the algorithm. In the first, we can first compute the new values for $V(s)$ for every state s, and then overwrite

all the old values with the new values. This is called a *synchronous update*. In this case, the algorithm can be viewed as implementing a “Bellman backup operator” that takes a current estimate of the value function, and maps it to a new estimate. Alternatively, we can also perform *asynchronous updates*.
Here, we would loop over the states (in some order), updating the values one at a time.

The policy iteration algorithm proceeds as follows:

Thus, the inner-loop repeatedly computes the value function for the current policy, and then updates the policy using the current value function. (The policy π found in step (b) is also called the policy that is greedy with respect to V.) Note that step (a) can be done via solving Bellman’s equations as described earlier, which in the case of a fixed policy, is just a set of $|S|$ linear equations in $|S|$ variables. After at most a finite number of i, V will finally converge to V^* and thus leading π converge to π^*.

1. Initialize π randomly
2. for until convergence do
3. Let $V := V^\pi$
4. For each state, let

$$V(s) := R(s) + \max_{a \in A} \gamma \sum_{s'} P_{sa}(s') V(s')$$

There isn’t currently universal agreement over which algorithm is better. For small MDPs, policy iteration is often very fats and converges with very few iterations. For this reason, in practice value iteration seems to be used more often than policy iteration.

Learning a model for MDPs

In many realistic problems, we are not given state transition probabilities and rewards explicitly, but must instead estimate them from data. (Usually, S, A and γ are known.) So now it’s our priority to know the P_{sa}, and thus perform the further steps. How can get the P_{sa}? Most instinct idea would be to use the frequency to estimate the probability.

Consider the pendulum cart problem, we can actually do a series of trials just like the expression below:

$$s_0^{(1)} \xrightarrow{a_0^{(1)}} s_1^{(1)} \xrightarrow{a_1^{(1)}} \cdots s_i^{(1)} \xrightarrow{a_i^{(1)}}$$

$$s_0^{(2)} \xrightarrow{a_0^{(2)}} s_1^{(2)} \xrightarrow{a_1^{(2)}} \cdots s_i^{(2)} \xrightarrow{a_i^{(2)}}$$

Here, $s_i^{(j)}$ is the state we were at time i of trial j, and $a_i^{(j)}$ is the corresponding action that was taken from that state. In practice, each of the trials above might be run until the MDP terminates (such as if the pole falls over in the inverted pendulum problem), or it might be run for some large but finite number of timesteps.

Given this “experience” in the MDP consisting of a number of trials, we can then easily derive the maximum likelihood estimates for the state transition probabilities:

$$P_{sa}(s') = \frac{\#\text{times took we action a in state s and got to s'}}{\#\text{times we took action a in state s}}$$

Or, if the ratio above is “0/0”—corresponding to the case of never having taken action a in state s before—the we might simply estimate $P_{sa}(s')$ to be $1/|S|$. (I.e., estimate P_{sa} to be the uniform distribution over all states.)

Note that, if we gain more experience (observe more trials) in the MDP, there is an efficient way to update our estimated state transition probabilities using the new experience. Specifically, if we keep around the counts for both the numerator and denominator terms of the given equation, then as we observe more trials, we can simply keep accumulating those counts.

Using a similar procedure, if R is unknown, we can also pick our estimate of the expected immediate reward $R(s)$ in state s to be the average reward observed in state s.

Now we can apply value iteration our policy iteration w.r.t. our estimated probabilities and reward functions. AN example is given here where we combined the value iteration and our model trainning.

1. Initialize π randomly
2. Repeat
    (a).Execute π in the MDP for some number of trials.
    (b).Using the accumulated experience in the MDP,
        update our estimates for P_{sa}(and R, if applicable).
    (c).Apply value iteration with the estimated state transition
        probabilities and rewards to get a new estimated value function V
    (d).Update π to be the greedy policy with respect to V.

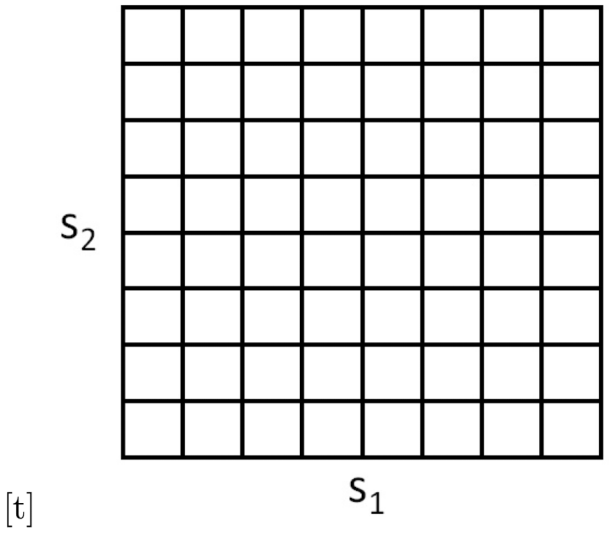
We note that, for this particular algorithm, there is one simple optimization that can make it run much more quickly. Specifically, in the inner loop of the algorithm where we apply value iteration, if instead of initializing value iteration with $V = 0$, we initialize it with the solution found during the previous iteration of our algorithm, then that will provide value iteration with a much better initial starting point and make it converge more quickly.

Continuous state MDPs

So far we've been talking about MDPs with only a finite number of states, now we want to discuss MDPs with infinite states and actions (maybe), and see if we had a solution for this kind of questions.

Discretization

The most common idea is of course to discretize the finite state into pieces so we can use our former ways. For example, if we have 2d states (s_1, s_2), we can use a grid to discretize the state space:



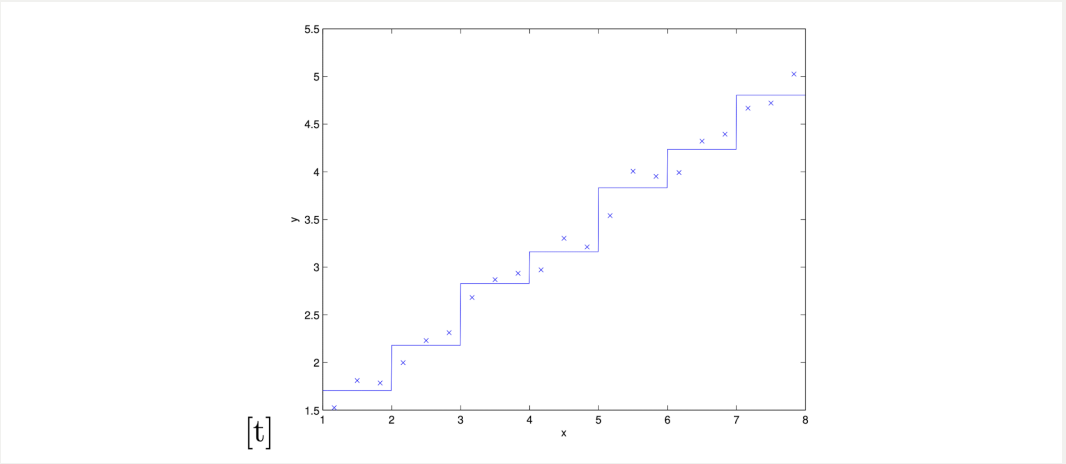
[t]

Here, each grid cell represents a separate discrete state $\hat{s}$. We can then approximate the continuous-state MDP via a discrete-state one $(\hat{S}, A, P_{sa}, \gamma, R)$, where $\hat{S}$ is the set of discrete states, P_{sa} are our state transition probabilities over the discrete states, and so on. We can then use value iteration or policy iteration to solve for the $V^*(\hat{s})$ and $\pi^*(\hat{s})$ in the discrete state $\text{MDP}(\hat{S}, A, P_{sa}, \gamma, R)$. When our actual system is in some continuous-valued state $s \in S$ and we need to pick an action to execute, we compute the corresponding discretized state $\hat{s}$, and execute action $\pi^*(\hat{s})$.

This discretization approach can work well for many problems. However, there are two downsides.

First, it uses a fairly naive representation for V^* (and π^*). Specifically, it assumes that the value function is takes a constant value over each of the discretization intervals (i.e., that the value function is piecewise constant in each of the grid cells).

📌 It’s like fitting a linear regression with zigzagging lines!



A second downside of this representation is called the *curse of dimensionality*. Suppose $S = R^d$, and we discretize each of the d dimensions of the state into k values. Then the total number of discrete states we have is k^d. This grows exponentially quickly in the dimension of the state space d, and thus does not scale well to large problems. For example, with a $10d$ state, if we discretize each state variable into 100 values, we would have $100^{10} = 10^{20}$ discrete states, which is far too many to represent even on a modern desktop computer.

As a rule of thumb, discretization usually works extremely well for 1d and 2d problems (and has the advantage of being simple and quick to implement). Perhaps with a little bit of cleverness and some care in choosing the discretization method, it often works well for problems with up to 4d states. If you’re extremely clever, and somewhat lucky, you may even get it to work for some 6d problems. But it very rarely works for problems any higher dimensional than that.

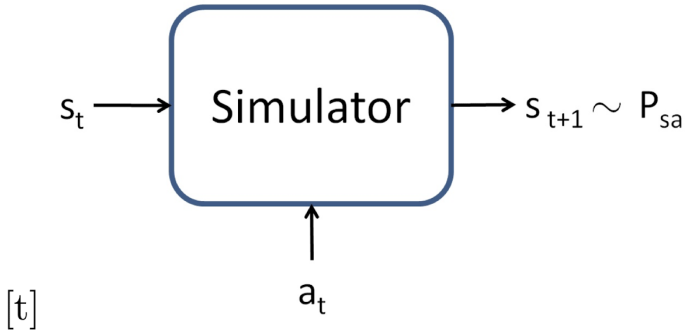
Value function approximation

We now describe an alternative method for finding policies in continuous-state MDPs, in which we approximate V^* directly, without resorting to discretization. This approach, called *value function approximation*, has been successfully applied to many RL problems.

Using a model or simulator

To develop a value function approximation algorithm, we will assume that we have a model, or *simulator*, for the MDP. Informally, a simulator is a black-box that takes as input any (continuous-valued) state s_t and action

a_t, and outputs a next-state s_{t+1} sampled according to the state transition probabilities $P_{s_t a_t}$:



[t]

We can develop two different kind of simulators. First is a simulator of real physics simulation. Like in the pendulum cart we can simulate the process with physics laws and calculate the state in real world.

The second one is we do a series of trials like we did before, and then use a machine learning model to predict our probabilities. For example, suppose we execute n trials in which we repeatedly take actions in an MDP, each trial for T timesteps. We would then observe n state sequences like the following:

$$s_0^{(1)} \to s_1^{(1)} \to \cdots \to s_T^{(1)}$$
$$s_0^{(2)} \to s_1^{(2)} \to \cdots \to s_T^{(2)}$$

We can then apply a learning algorithm to predict s_{t+1} as a function of s_t and a_t. For example, one may choose to learn a linear model of the form

$$s_{t+1} = As_t + Ba_t$$

using an algorithm similar to linear regression. Here, the parameters of the model are the matrices A and B, and we can estimate them using the data collected from our n trials, by picking

$$arg\min_{A,B} \sum_{i=1}^{n} \sum_{t=0}^{T-1} ||s_{t+1}^{(i)} - As_t^{(i)} + Ba_t^{(i)}||_2^2$$

We could also potentially use other loss functions for learning the model. For example, it has been found in recent work Luo et al. [2018] that using $\left\|\cdot\right\|_2$ norm (without the square) may be helpful in certain cases.

Having learned A and B, one option is to build a *deterministic model*, in which given an input s_t and a_t, the output s_{t+1} is exactly determined. Alternatively, we may also build a *stochastic model*, in which s_{t+1} is a random function of the inputs, by modeling it as

$$s_{t+1} = As_t + Ba_t + \epsilon_t$$

where here ϵ_t is a noise term.

Here, we've written the next-state s_{t+1} as a linear function of the current state and action; but of course, non-linear functions are also possible. Specifically, one can learn a model $s_{t+1} = A\phi_s(s_t) + B\phi_a(a_t)$, where ϕ_s and ϕ_a are some non-linear feature mappings of the states and actions. Alternatively, one can also use non-linear learning algorithms to learn all these.

Fitted value iteration

We now describe the *fitted value iteration algorithm* for approximating the value function of a continuous state MDP. In the sequel, we will assume that the problem has a continuous state space $S = R^d$, but that the action space A is small and discrete.

> 📌 In practice, most MDPs have much smaller action spaces than state spaces. Figure this out yourself.

Recall that in value iteration, we would like to perform the update

$$\begin{aligned} V(s) &:= R(s) + \gamma \max_a \int P_{sa}(s')V(s')ds' \\ &= R(s) + \gamma \max_a E_{s' \sim P_{sa}}[V(s')] \end{aligned} \tag{1}$$

The main idea of this algorithm is that we are going to approximately carry out the step where we estimate the value function using a supervised learning algorithm over a finite samples of $s^{(1)}, \cdots, s^{(n)}$. Our estimate would be

$$V(s) = \theta^T \phi(s)$$

For each state s in our finite sample of n states, fitted value iteration will first compute a quantity $y^{(i)}$, which will be our approximation to $R(s) + \gamma \max_a E_{s' \sim P_{sa}}[V(s')]$ (the right hand side of Equation 1). Then, it will apply a supervised learning algorithm to try to get $V(s)$ close to $R(s) + \gamma \max_a E_{s' \sim P_{sa}}[V(s')]$ (or, in other words, to try to get $V(s)$ close to $y^{(i)}$).

In detail, the algorithm is as follows:

1. Randomly sample $s^{(1)}, s^{(2)}, \ldots, s^{(n)} \in S$.
2. Initialize $\theta := 0$.
3. Repeat {
   For $i = 1, \ldots, n$ {
     For each action $a \in A$ {
       Sample $s'_1, \ldots, s'_k \sim P_{s^{(i)},a}$ (using a model of the MDP).
       Set $q(a) = \frac{1}{k} \sum_{j=1}^{k} \left[R(s^{(i)}) + \gamma V(s'_j) \right]$.
       // Hence, $q(a)$ is an estimate of $R(s^{(i)}) + \gamma \mathbb{E}_{s' \sim P_{s^{(i)},a}}[V(s')]$.
     }
     Set $y^{(i)} = \max_a q(a)$.
     // Hence, $y^{(i)}$ is an estimate of $R(s^{(i)}) + \gamma \max_a \mathbb{E}_{s' \sim P_{s^{(i)},a}}[V(s')]$.
   }
   // In the original value iteration algorithm (over discrete states), we u
      the value function according to $V(s^{(i)}) := y^{(i)}$.
   // In this algorithm, we want $V(s^{(i)}) \approx y^{(i)}$, which we'll achieve using
      supervised learning (linear regression).
   Set $\theta := \arg\min_\theta \frac{1}{2} \sum_{i=1}^{n} \left(\theta^T \phi(s^{(i)}) - y^{(i)} \right)^2$.
}

Even though we've discussed about this in the field of the linear regression, still we can use some different learning algorithms as well.

Unlike value iteration over a discrete set of states, fitted value iteration cannot be proved to always to converge. However, in practice, it often does converge (or approximately converge), and works well for many problems.

Note also that if we are using a deterministic simulator/model of the MDP, then fitted value iteration can be simplified by setting $k = 1$ in the algorithm. This is because the expectation in Equation (1) becomes an expectation over a deterministic distribution, and so a single example is sufficient to exactly compute that expectation. Otherwise, in the algorithm above,

we had to draw k samples, and average to try to approximate that expectation (see the definition of $q(a)$, in the algorithm pseudo-code).

Finally, fitted value iteration outputs V, which is an approximation to V^*. This implicitly defines our policy. Specifically, when our system is in some state s, and we need to choose an action, we would like to choose the action

$$arg\max_a E_{s' \sim P_{sa}}[V(s')] \tag{2}$$

The process for computing/approximating this is similar to the inner-loop of fitted value iteration, where for each action, we sample $s'_1, ..., s'_k \sim P_{sa}$ to approximate the expectation. (And again, if the simulator is deterministic, we can set $k = 1$.)

In practice, there are often other ways to approximate this step as well. For example, one very common case is if the simulator is of the form $s_{t+1} = f(s_t, a_t) + \epsilon_t$, where f is some deterministic function of the states (such as

$f(s_t, a_t) = As_t + Ba_t$), and ϵ is zero-mean Gaussian noise. In this case, we can pick the action given by

$$arg\max_a V(f(s, a))$$

In other words, here we are just setting $\epsilon t = 0$ (i.e., ignoring the noise in the simulator), and setting $k = 1$. Equivalent, this can be derived from Equation (2) using the approximation

$$\begin{aligned} E_{s'}[V(s')] &\approx V(E_{s'}[s']) \\ &= V(f(s, a)) \end{aligned}$$

where here the expectation is over the random $s' \sim P_{sa}$. So long as the noise terms ϵ_t are small, this will usually be a reasonable approximation.

However, for problems that don't lend themselves to such approximations, having to sample $k|A|$ states using the model, in order to approximate the expectation above, can be computationally expensive.

Connections between Policy and Value Iteration

In the policy iteration, we typically use linear system solver to compute V^π. Alternatively, one can also *the iterative Bellman updates*, similarly to the value iteration, to evaluate V^π, as in the Procedure $VE(\cdot)$ in Line 1 of Algorithm below.

Variant of Policy Iteration

1. procedure $VE(\pi, k)$ ◁ To evaluate V_π
2. Option 1: initialize $V(s) := 0$;
   Option 2: Initialize from the current V in the main algorithm.
3. for $i = 0$ to $k - 1$ do
4. For every state $s,$ update
5. $V(s) := R(s) + \gamma \sum_{s'} P_{s,\pi(s)}(s')V(s').$
6. return V

Require: hyperparameter k.

7. Initialize π randomly.
8. for until convergence do
9. Let $V = VE(\pi, k)$.
10. For each state $s,$ let $\pi(s) := \arg\max_{a \in A} \sum_{s'} P_{s,a}(s')V(s').$

Here if we take option 1 in Line 2 of the Procedure VE, then the difference between the Procedure VE from the value iteration would be that on Line 4 we're now using action from π instead of the greedy action.

Using the Procedure VE, we can build the given algorithm, which is a variant of policy iteration that serves an intermediate algorithm that connects policy iteration and value iteration.

Note that here if we choose option 2 in Line 2, and set $k = 0$, we are actually facing a value iteration. In other words, they both interleave the updates we showed in the algorithm, where one updates the V and the other updates the acting policy(you may figure that the update of acting policy in value iteration is invisible).

Algorithm 8 alternate between k steps of updating V and one step of updating π, whereas value iteration alternates between 1 steps of updating V and one step of updating π. This indicates that the algorithm would be no faster than the value iteration.

On the other hand, if k steps of update V can be done much faster than k times a single step of update V, then taking additional steps of equation update V in group might be useful. This is what policy iteration is leveraging — the linear system solver can give us the result of Procedure VE with $k = \infty$ much faster than using the Proced- ure VE for a large k. On

the flip side, when such a speeding-up effect no longer exists, e.g.,, when the state space is large and linear system solver is also not fast, then value iteration is more preferable.

LQR, DDP and LQG

Finite-horizon MDPs

Recall that in the previous chapters we define a MDPs where we have a optimal Bellman equation to offer us a optimal policy and value function. However, here in this chapter we want something that's more a general situation.

So let's place ourselves in a more general settings.

1. To cover both discrete and continue situations, we use notation

$$E_{s' \sim P_{sa}} V^{\pi^*}(s')$$

   instead of the $\sum_{s' \in S} P_{sa}(s') V^{\pi^*}(s')$ we used before.

2. We assume that rewards once again depend on both action and state that is to say

$$R : S \times A \to \mathbb{R}$$

This implies that the previous mechanism for computing the optimal action is changed into

$$\pi^*(s) = \arg\max_{a \in A} R(s,a) + \gamma E_{s' \sim P_{sa}} V^{\pi^*}(s')$$

3. Instead of a infinite time horizon MDP, we consider a finite time horizon one here with the the horizon being T, i.e.

$$(S, A, P_{sa}, T, R)$$

In this setting, our definition of payoff is going to be (slightly) different:

$$R(s_0, a_0) + R(s_1, a_1) + \cdots + R(s_T, a_T)$$

instead of

$$\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t)$$

Notice why we remove the discount factor in the previous formula, because here in the finite horizon we no longer need a discount factor to converge the entire payoff which is of a infinite scale under the infinite horizon.

In this new setting, things behave quite differently. First, the optimal policy π^*might be non-stationary, meaning that it changes over time. In other words, now we have

$$\pi^{(t)} : S \to A$$

Why in the finite horizon we have a non-stationary policy? Consider a game your playing where you have only finite steps to take and you want to both save the princess (way more important) and collect the diamond. So suggesting that at some specific step, your are only one step away from the diamond and three steps form the princess. But you only got 2 steps remaining! Of course you should give up your origin policy under which we try to get to princess.

4. This observation allows us to use time dependent dynamics

$$s_{t+1} \backsim P_{s_t, a_t}^{(t)}$$

meaning that the transition's distribution $P_{s_t, a_t}^{(t)}$ changes over time. The same thing can be said about $R(t)$. Combining the former remarks, we now have our tuple as

$$(S, A, P_{s_t,a_t}^{(t)}, T, R^{(t)})$$

Remark: notice that the above formulation would be equivalent to adding the time into the state.

The value function at time t for a policy π is then defined in the same way as before, as an expectation over trajectories generated following policy π starting in state s.

$$V_t(s) = E\left[R^{(t)}(s_t, a_t) + \cdots + R^{(T)}(s_T, a_T) | s_t = s, \pi\right]$$

So now our question is

$$In\ the\ finite\ horizon\ how\ do\ we\ find\ optimal\ value\ function$$
$$V_t^*(s) = \max_\pi V_t^\pi(s)$$

It turns out that Bellman's equation for Value Iteration is made for *Dynamic Programming*. This may come as no surprise as Bellman is one of the fathers of dynamic programming and the Bellman equation is strongly related to the field. To understand how we can simplify the problem by adopting an iteration-based approach, we make the following observations:

1. First we need to notice that at the end of the process the value function is simple

$$\forall s \in S : V_T^*(s) := \max_{a \in A} R^{(T)}(s, a) \tag{1}$$

2. For any other state at time t where $t \in [0, T)$, we suppose we know the value function of the next time step V_{t+1}^*, and then we can simply find that the current value function is given by

$$\forall t < T, s \in S : V_t^*(s) := \max_{a \in A} \left[R^{(t)}(s, a) + E_{s' \sim P_{sa}^{(t)}}[V_{t+1}^*(s')]\right] \tag{2}$$

So now we can come to a clear structure of a DP algorithm.

Algorithm

1. compute V_T^* using equation (1).
2. for $t = T - 1, ..., 0$:

   compute V_t^* using V_{t+1}^* using equation (2).

📌 Note

We can interpret standard value iteration as a special case of this general case, but without keeping track of time.

It turns out that in the standard setting, if we run value iteration for T steps, we get a γ^T approximation of the optimal value iteration (geometric convergence).

At last we give *the Bellman Contraction Theorem.*

Theorem: Let B denote the Bellman update and $||f(x)||_\infty := \sup_x |f(x)|$. If V_t denotes the value function at the t-th step, then

$$\begin{aligned} ||V_{t+1} - V^*||_\infty &= ||B(V_t) - V^*||_\infty \\ &\leq \gamma ||V_t - V^*||_\infty \\ &\leq \gamma^t ||V_1 - V^*||_\infty \end{aligned}$$

This indicates that the error between our V_t and V^* will be contracting at the speed of γ.

Linear Quadratic Regulation (LQR)

In this section we'll be mainly talk about a special kind of problem which is quadratic and thus can lead to an **exact solution.** This model is widely used in robotics, and a common technique in many problems is to reduce the formulation to this framework.

First let's state our assumptions in this special situation. We place ourselves in the continuous setting, with

$$S = \mathbb{R}^d, A = \mathbb{R}^d$$

and we'll assume **linear transitions** (with noise w)

$$s_{t+1} = A_t s_t + B_t a_t + w_t$$

where $A_t \in \mathbb{R}^{d \times d}$, $B_t \in \mathbb{R}^{d \times d}$ are matrices and $w_t \sim \mathcal{N}(0, \Sigma_t)$ is some gaussian noise (with zero mean). As we'll show in the following paragraphs, it turns out that the noise, as long as it has zero mean, does not impact the optimal policy!

And we have our most assumption where R is given as a quadratic function of action and state:

$$R^{(t)}(s_t, a_t) = -s_t^T U_t s_t - a_t^T W_t a_t$$

Note here our rewards are always negative, which means the best we can do is to stay in a stationary state. For example, if $U_t = I_d$ (the identity matrix) and $W_t = I_d$, then $R_t = -||s_t||^2 - ||a_t||^2$, meaning that we want to take smooth actions (small norm of at) to go back to the origin (small norm of s_t). This could model a car trying to stay in the middle of lane without making impulsive moves(inverted pendulum cart!)

So now we can cover a two step algorithm for LQR:

STEP 1 suppose that we don't know the matrices A, B, Σ. To estimate them, we can follow the ideas outlined in the Value Approximation section of the RL notes. First, collect transitions from an arbitrary policy. Then, use linear regression to find $\arg\min_{A,B} \sum_{i=1}^{n} \sum_{t=0}^{T-1} \left|\left|s_{t+1}^{(i)} - As_t^{(i)} + Ba_t^{(i)}\right|\right|^2$. Finally, use a technique seen in Gaussian Discriminant Analysis to learn Σ.

STEP 2 Derive the optimal policy using DP

$$\begin{cases} s_{t+1} = A_t s_t + B_t a_t + w_t \quad A_t, B_t, U_t, W_t, \Sigma_t \text{ known} \\ R^{(t)}(s_t, a_t) = -s_t^T U_t s_t - a_t^T W_t a_t \end{cases}$$

So if we want to compute the V_t^*, we can simply do a DP like below:

1. **Initialization step**

   For the last time step T,

   $$\begin{aligned} V_T^*(s_T) &= \max_{a_T \in A} R_T(s_T, a_T) \\ &= \max_{a^T \in A} -s_T^\top U_T s_T - a_T^\top W_t a_T \\ &= -s_T^\top U_t s_T \end{aligned}$$

2. **Recurrence step**

   Let $t < T$. Suppose we know V_{t+1}^*.

   Fact 1: It can be shown that if V_{t+1}^* is a quadratic function in st, then V_t^* is also a quadratic function. In other words, there exists some matrix Φ and some scalar Ψ such that

   $$\begin{aligned} &\text{if } V_{t+1}^*(s_{t+1}) = s_{t+1}^\top \Phi_{t+1} s_{t+1} + \Psi_{t+1} \\ &\text{then } V_t^*(s_t) = s_t^\top \Phi_t s_t + \Psi_t \end{aligned}$$

For time step $t = T$, we had $\Phi_t = -U_T$ and $\Psi_T = 0$.

Fact 2: We can show that the optimal policy is just a linear function of the state. Knowing V^*_{t+1} is equivalent to knowing Φ_{t+1} and Ψ_{t+1}, so we just need to explain how we compute Φ_t and Ψ_t from Φ_{t+1} and Ψ_{t+1} and the other parameters of the problem.

$$\begin{aligned} V^*_t(s_t) &= s_t^\top \Phi_t s_t + \Psi_t \\ &= \max_{a_t} R^{(t)}(s_t, a_t) + \mathbb{E}_{s_{t+1} \sim P^{(t)}} \left[V^*_{t+1}(s_{t+1}) \right] \\ &= \max_{a_t} -s_t^\top U_t s_t - a_t^\top V_t a_t + \mathbb{E}_{s_{t+1} \sim \mathcal{N}(A_t s_t + B_t a_t, \Sigma_t)} \left[s_{t+1}^\top \Phi_{t+1} s_{t+1} + \right. \end{aligned}$$

where the second line is just the definition of the optimal value function and the third line is obtained by plugging in the dynamics of our model along with the quadratic assumption.

Since this is quadratic this could be simply optimized and we get a^*_t

$$\begin{aligned} a^*_t &= (B_t^\top \Phi_{t+1} B_t - V_t)^{-1} B_t \Phi_{t+1} A_t \cdot s_t \\ &= L_t \cdot s_t \end{aligned}$$

where

$$L_t := (B_t^\top \Phi_{t+1} B_t - W_t)^{-1} B_t \Phi_{t+1} A_t$$

which is an impressive result: our optimal policy is linear in s_t. Given a^*_t we can solve for Φ_t and Ψ_t. We finally get *the Discrete Ricatti equations*

$$\begin{aligned} \Phi_t &= A_t^\top \Phi_{t+1} - \Phi_{t+1} B_t \left(B_t^\top \Phi_{t+1} B_t - W_t \right)^{-1} B_t^\top \Phi_{t+1} A_t - U_t \\ \Psi_t &= -\text{tr}(\Sigma_t \Phi_{t+1}) + \Psi_{t+1} \end{aligned}$$

Fact 3: we notice that Φ_t depends on neither Ψ nor the noise Σ_t! As L_t is a function of A_t, B_t and Φ_{t+1}, it implies that the optimal policy also does not depend on the noise! (But Ψ_t does depend on Σ_t, which implies that V^*_t depends on Σ_t.)

Then, to summarize, the LQR algorithm works as follows

1. (if necessary) estimate parameters A_t, B_t, Σ_t.
2. initialize $\Phi_T := -U_T$ and $\Psi_T := 0$.

3. iterate from $t = T - 1 ... 0$ to update Φ_t and Ψ_t using Φ_{t+1} and Ψ_{t+1} using the discrete Ricatti equations. If there exists a policy that drives the state towards zero, then convergence is guaranteed!

> 📌 As the optimal policy does not depend on Ψ_t, and the update of Φ_t only depends on Φ_t, it is sufficient to update only Φ_t!

From non-linear dynamics to LQR

It turns out that a lot of problems can be reduced to LQR, even if dynamics are non-linear. Consider the inverted pendulum cart problem, obviously this is not linear. So now the question come to us is: *how can we linearize this system?*

Linearization of dynamics

Let's suppose that at time t, the system spends most of its time in some state $\bar{s}_t$ and the actions we perform are around $\bar{a}_t$. For the inverted pendulum, if we reached some kind of optimal, this is true: our actions are small and we don't deviate much from the vertical.

In the simple case where the state is one-dimensional and the transition function F does not depend on the action, we would write something using Taylor's expansion

$$s_{t+1} = F(s_t) \approx F(\bar{s}_t) + F'(\bar{s}_t) \cdot (s_t - \bar{s}_t)$$

More generally we have

$$s_{t+1} \approx F(\bar{s}_t, \bar{a}_t) + \nabla_s F(\bar{s}_t, \bar{a}_t) \cdot (s_t - \bar{s}_t) + \nabla_a F(\bar{s}_t, \bar{a}_t) \cdot (a_t - \bar{a}_t)$$

And now it's finally linear, we can write $s_{t+1} \approx As_t + Bs_t + \kappa$, where κ is a constant, and we need to get rid of it. Do what we did before, the constant term can be absorbed into s_t by artificially increasing the dimension by one.

Differential Dynamic Programming (DDP)

Consider a system where our agent has to follow a certain trajectory (say a rocket). How can we make this discrete? We'll apply a method called *differential dynamic programming*, to solve this. The main steps are as below:

1. come up with a nominal trajectory using a naive controller, that approximate the trajectory we want to follow. In other words, our controller is able to approximate the gold trajectory with

$$s_0^*, a_0^* \rightarrow s_1^*, a_1^* \rightarrow \dots$$

2. linearize the dynamics around each trajectory point s_t^*, in other words

$$s_{t+1} \approx F(s_t^*, a_t^*) + \nabla_s F(s_t^*, a_t^*) \cdot (s_t - s_t^*) + \nabla_a F(s_t^*, a_t^*) \cdot (a_t - a_t^*)$$

   where s_t, a_t would be our current state and action. Now that we have a linear approximation around each of these points, we can use the previous section and rewrite

$$s_{t+1} = A_t \cdot s_t + B_t \cdot a_t$$

   (notice that in that case, we use the non-stationary dynamics setting that we mentioned at the beginning of this section)

   Note We can apply a similar derivation for the reward $R(t)$, with a second-order Taylor expansion. That way, our reward could be rewritten as

$$R_t(s_t, a_t) = -s_t^\top U_t s_t - a_t^\top W_t a_t$$

3. Now, you can convince yourself that our problem is strictly re-written in the LQR framework. Let's just use LQR to find the optimal policy π_t. As a result, our new controller will (hopefully) be better!

   Note: Some problems might arise if the LQR trajectory deviates too much from the linearized approximation of the trajectory, but that can be fixed with reward-shaping...

4. Now that we get a new controller (our new policy π_t), we use it to produce a new trajectory

$$s_0^*, \pi_0(s_0^*) \rightarrow s_1^*, \pi_1(s_1^*) \rightarrow ... \rightarrow s_T^*$$

   note that when we generate this new trajectory, we use the real F and not its linear approximation to compute transitions, meaning that

$$s_{t+1}^* = F(s_t^*, a_t^*)$$

Linear Quadratic Gaussian (LQG)

So far we've been working with a basic assumption that state is definitely available for us, i.e. we can make clear and distinguished definition of every state and make decision with accordance to the given state.

However, for many real-life situations, like self-driving cars, it'e hard to tell what state we're really in——you can't name the exact state when driving on the road, there are just too many of them! So for problems like this, we can just get a ambiguous assumption for the state rather than a exact state. As this might not hold true for most of the real-world problems, we need a new tool to model this situation: *Partially Observable MDPs*.
A POMDP is an MDP with an extra observation layer. In other words, we introduce a new variable
o_t, that follows some conditional distribution given the current state s_t

$$o_t|s_t \sim O(o|s)$$

Formally, a finite-horizon POMDP is given by a tuple

$$(S, O, A, P_{sa}, T, R)$$

Within this framework, the general strategy is to maintain a **belief state** (distribution over states) based on the observation $o_1, ..., o_t$. Then, a policy in a POMDP maps this belief states to actions. So in the next section, we try to extend our LQR structure to this new genre of problems and get a fixed paradigm.

Remember one important assumption about our LQR structure is that out state transition function is linear, so now we make the same assumption as before:

$$\begin{cases} s_{t+1} = As_t + Ba_t + w_t \\ y_t = Cs_t + v_t \end{cases}$$

Here $y_t \in R^n$ with $m < n$ is our observation and $C \in R^{n \times d}$ is a compression matrix and v_t is the sensor noise (also gaussian, like w_t). Note that the reward function $R(t)$ is left unchanged, as a function of the state (not the observation) and action. Also, as distributions are gaussian, the belief state is also going to be gaussian. In this new framework, let's give an overview of the strategy we are going to adopt to find the optimal policy:

STEP 1:

first, compute the distribution on the possible states (the belief state), based on the observations we have. In other words, we want to compute the mean $s_{t|t}$ and the covariance $\Sigma_{t|t}$ of

$$s_{t|t}|y_1 \dots y_t \backsim N(s_{t|t}, \Sigma_{t|t})$$

Note here the denotation $s_{t|t}$ means distribution of of s_t given by the observation at time t. To perform the computation efficiently over time, we'll use the *Kalman Filter algorithm* (used on-board Apollo Lunar Module!).

STEP 2

Now that we have the distribution, we'll use the mean $s_{t|t}$ as the best approximation for s_t.

STEP 3

Then set the action $a_t := L_t s_t|_t$ where L_t comes from the regular LQR algorithm.

Intuitively, to understand why this works, notice that $s_{t|t}$ is a noisy approximation of s_t (equivalent to adding more noise to LQR) but we proved that LQR is independent of the noise!

And we need to make more explicit explanation about step 1 here. We'll cover a simple case where there is no action dependence in our dynamics (but the general case follows the same idea). Suppose that

$$\begin{cases} s_{t+1} = A \cdot s_t + w_t, w_t \sim N(0, \Sigma_s) \\ y_t = C \cdot s_t + v_t, v_t \sim N(0, \Sigma_y) \end{cases}$$

As noises are Gaussians, we can easily prove that the joint distribution is also Gaussian

$$\begin{pmatrix} s_1 \\ \vdots \\ s_t \\ y_1 \\ \vdots \\ y_t \end{pmatrix} \sim N(\mu, \Sigma) \qquad \text{for some } \mu, \Sigma$$

then, using the marginal formulas of gaussians (see Factor Analysis section), we would get

$$s_t|y_1 \dots y_t \backsim N(s_{t|t}, \Sigma_{t|t})$$

However, computing the marginal distribution parameters using these formulas would be computationally expensive! It would require manipulating matrices of shape $t \times t$. Recall that inverting a matrix can be done in $O(t^3)$, and it would then have to be repeated over the time steps, yielding a cost in $O(t^4)$!

The Kalman filter algorithm provides a much better way of computing the mean and variance, by updating them over time in constant time in t! The kalman filter is based on two basics steps. Assume that we know the distribution of $s_t|y_1, ..., y_t$:

predict step compute $s_{t+1}|y_1, ..., y_t$

update step compute $s_{t+1}|y_1, ..., y_{t+1}$

In other words, the process looks like

$$(s_t|y_1,...,y_t) \xrightarrow{predict} (s_{t+1}|y_1,...,y_t) \xrightarrow{update} (s_{t+1}|y_1,...,y_{t+1}) \xrightarrow{predict} \dots$$

predict step

Suppose that we know the distribution of

$$s_t|y_1 \dots y_t \backsim N(s_{t|t}, \Sigma_{t|t})$$

then, the distribution over the next state is also a gaussian distribution

$$s_{t+1}|y_1 \dots y_t \backsim N(s_{t+1|t}, \Sigma_{t+1|t})$$

where

$$\begin{cases} s_{t+1|t} = As_{t|t} \\ \Sigma_{t+1|t} = A\Sigma_{t|t}A^\top + \Sigma_s \end{cases}$$

updating step

given $s_{t+1|t}$ and $\Sigma_{t+1|t}$ such that

$$s_{t+1}|y_1, ..., y_t \sim N(s_{t+1|t}, \Sigma_{t+1|t})$$

we can prove that

$$s_{t+1}|y_1 \dots y_{t+1} \backsim N(s_{t+1|t+1}, \Sigma_{t+1|t+1})$$

where

$$\begin{cases} s_{t+1|t+1} = s_{t+1|t} + K_t(y_{t+1} - Cs_{t+1|t}) \\ \Sigma_{t+1|t+1} = \Sigma_{t+1|t} - K_t \cdot C \cdot \Sigma_{t+1|t} \end{cases}$$

with

$$K_t := \Sigma_{t+1|t} C^\top (C \Sigma_{t+1|t} C^\top + \Sigma_y)^{-1}$$

The matrix K_t is called the *Kalman gain*.

Now, if we have a closer look at the formulas, we notice that we don't need the observations prior to time step t! The update steps only depends on the previous distribution. Putting it all together, the algorithm first runs a forward pass to compute the $K_t, \Sigma_{t|t}$ and $s_{t|t}$ (sometimes referred to as $\hat{s}$ in the literature). Then, it runs a backward pass (the LQR updates) to compute the quantities Ψt,Ψt and Lt. Finally, we recover the optimal policy with $a^*t = L_t s_{t|t}$.

Policy Gradient (REINFORCE)

We will present a model-free algorithm called REINFORCE that does not require the notion of value functions and Q functions as the last chapter of our ML notes.

It's turned out that it's easier to present REINFORCE in a finite horizon, so we'll be starting in a finite setting. Suggest $\tau = (s_0, a_0, ..., s_{T-1}, a_{T-1}, s_T)$ denote a trajectory, where $T < \infty$ is the length of the trajectory. Moreover, REINFORCE only applies to learning a randomized policy. We use $\pi_\theta(a|s)$ to denote the probability of the policy π_θ outputting the action a at state s. The other notations will be the same as in previous notes.

The advantage of applying REINFORCE is that we only need to assume that we can sample from the transition probabilities P_{sa} and can query the reward function $R(s, a)$ at state s and action a,1 but we do not need to know the analytical form of the transition probabilities or the reward function. We do not explicitly learn the transition probabilities or the reward function either.

Let s_0 be sampled from some distribution μ. We consider optimizing the expected total payoff of the policy π_θ over the parameter θ defined as.

$$\eta(\theta) \triangleq \mathbb{E}\left[\sum_{t=0}^{T-1} \gamma^t R(s_t, a_t)\right] \tag{1}$$

Recall that $s_t \sim P_{s_{t-1}a_{t-1}}$ and $a_t \sim \pi_\theta(\cdot\,|s_t)$. Also note that $\eta(\theta) = E_{s_0 \sim P}[V^{\pi_\theta}(s_0)]$ if we ignore the difference between finite and infinite horizon.

We want to compute the $\eta(\theta)$ with gradient ascent, while the challenge here is that we don't have any knowledge of the transition probabilities and reward function. Let $P_\theta(\tau)$ denote the distribution of τ (generated by the policy π_θ), and let $f(\tau) = \sum_{t=0}^{T-1} \gamma^t R(s_t, a_t)$. We can rewrite $\eta(\theta)$ as

$$\eta(\theta) = E_{\tau \sim P_\theta}[f(\tau)]$$

We face a similar situations in the variational auto-encoder (VAE) setting covered in the previous lectures, where the we need to take the gradient w.r.t to a variable that shows up under the expectation — the distribution P_θ depends on θ. Recall that in VAE, we used the re-parametrization techniques to address this problem. However it does not apply here because we do know not how to compute the gradient of the function f. We only have an efficient way to evaluate the function f by taking a weighted sum of the observed rewards, but we do not necessarily know the reward function itself to compute the gradient.

The REINFORCE algorithm uses an another approach to estimate the gradient of $\eta(\theta)$. We start with the following derivation:

$$\begin{aligned}
\nabla_\theta \mathbb{E}_{\tau \sim P_\theta}[f(\tau)] &= \nabla_\theta \int P_\theta(\tau) f(\tau) d\tau \\
&= \int \nabla_\theta (P_\theta(\tau) f(\tau)) d\tau \quad \text{(swap integration with gradient)} \\
&= \int (\nabla_\theta P_\theta(\tau)) f(\tau) d\tau \quad \text{(because } f \text{ does not depend on } \theta) \\
&= \int P_\theta(\tau) (\nabla_\theta \log P_\theta(\tau)) f(\tau) d\tau \\
&\quad \left(\text{because } \nabla_\theta \log P_\theta(\tau) = \frac{\nabla_\theta P_\theta(\tau)}{P_\theta(\tau)} \right) \\
&= \mathbb{E}_{\tau \sim P_\theta} \left[(\nabla_\theta \log P_\theta(\tau)) f(\tau) \right]
\end{aligned} \tag{2}$$

Now we have a sample-based estimator for $\nabla_\theta E_{\tau \sim P_\theta}[f(\tau)]$. Let $\tau(1), ..., \tau(n)$ be n empirical samples from P_θ (which are obtained by running the policy π_θ for n times, with T steps for each run). We can estimate the gradient of $\eta(\theta)$ by

$$\begin{aligned}
\nabla_\theta \mathbb{E}_{\tau \sim P_\theta}[f(\tau)] &= \mathbb{E}_{\tau \sim P_\theta} \left[(\nabla_\theta \log P_\theta(\tau)) f(\tau) \right] \\
&\approx \frac{1}{n} \sum_{i=1}^{n} (\nabla_\theta \log P_\theta(\tau^{(i)})) f(\tau^{(i)})
\end{aligned} \tag{3}$$

So now the problem has come to how to compute the $\log P_\theta(\tau)$. Here we derive a formula for $\log P_\theta(\tau)$ and calculate it's gradient w.r.t θ. Using the definition of τ, we have

$$P_\theta(\tau) = \mu(s_0)\pi_\theta(a_0|s_0)P_{s_0a_0}(s_1)\pi_\theta(a_1|s_1)P_{s_1a_1}(s_2)\cdots P_{s_{T-1}a_{T-1}}(s_T)$$

Here recall that μ to used to denote the density of the distribution of s_0. It follows that

$$\log P_\theta(\tau) = \log \mu(s_0) + \log \pi_\theta(a_0|s_0) + \log P_{s_0a_0}(s_1) + \log \pi_\theta(a_1|s_1) \\ + \log P_{s_1a_1}(s_2) + \cdots + \log P_{s_{T-1}a_{T-1}}(s_T)$$

Take gradient w.r.t θ, we get

$$\nabla_\theta \log P_\theta(\tau) = \nabla_\theta \log \pi_\theta(a_0|s_0) + \nabla_\theta \log \pi_\theta(a_1|s_1) + \cdots + \nabla_\theta \log \pi_\theta(a_{T-1}|$$

Plugging the equation above into equation (3), we conclude that

$$\begin{aligned} \nabla_\theta \eta(\theta) = \nabla_\theta \mathbb{E}_{\tau\sim P_\theta}[f(\tau)] &= \mathbb{E}_{\tau\sim P_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\cdot f(\tau)\right] \\ &= \mathbb{E}_{\tau\sim P_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\cdot \sum_{t=0}^{T-1}\gamma^t R(s_t,a_t)\right] \end{aligned} \tag{4}$$

This is our *Policy Gradient Theorem*. We estimate the RHS of the equation above by empirical sample trajectories, and the estimate is unbiased. The vanilla REINFORCE algorithm iteratively updates the parameter by gradient ascent using the estimated gradients.

Now we give some interpretation about the equation (4).

The quantity $\nabla_\theta P_\theta(\tau) = \sum_{t=0}^{T-1} \nabla_\theta log \pi_\theta(a_t|s_t)$ is intuitively the direction of the change of θ that will make the trajectory τ more likely to occur (or increase the probability of choosing action $a_0, ..., a_{t-1}$), and $f(\tau)$ is the total payoff of this trajectory. Thus, by taking a gradient step, intuitively we are trying to improve the likelihood of all the trajectories, but with a different emphasis or weight for each τ (or for each set of actions $a_0, a_1, ..., a_{t-1}$). So obviously, if τ turns out to be very rewarding, i.e. f(\tau) is really large, we will try really hard move towards a way that can increase the probability of the trajectory τ.

An interesting fact that follows from formula (2) is that

$$\mathbb{E}_{\tau\sim P_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\right] = 0 \tag{5}$$

This is because our expectation is unbiased, when there do not exist any extra reward, the expectation of the sum of the gradient of all possible trajectory η is for sure 0. We can also make a strict math proof. To do whis, we suggest $f(\tau) = 1$ which is a constant, so we can rewrite we formula (4), and noticing that LHS is zero now because the gradient of a constant is for sure zero, and the RHS is exactly the LHS of (5).

In fact, one can verify that $E_{a_t \sim \pi_\theta(\cdot|st)}[\nabla_\theta \log \pi_\theta(a_t|s_t)] = 0$ for any fixed t and s_t. This fact has two consequences. First, we can simplify formula (4) to

$$\begin{aligned} \nabla_\theta \eta(\theta) &= \sum_{t=0}^{T-1} \mathbb{E}_{\tau \sim P_\theta} \left[\nabla_\theta \log \pi_\theta(a_t|s_t) \cdot \sum_{j=0}^{T-1} \gamma^j R(s_j, a_j) \right] \\ &= \sum_{t=0}^{T-1} \mathbb{E}_{\tau \sim P_\theta} \left[\nabla_\theta \log \pi_\theta(a_t|s_t) \cdot \sum_{j \geq t}^{T-1} \gamma^j R(s_j, a_j) \right] \end{aligned} \tag{6}$$

where the second equality follows from

$$\begin{aligned} &\mathbb{E}_{\tau \sim P_\theta} \left[\nabla_\theta \log \pi_\theta(a_t|s_t) \cdot \sum_{0 \leq j < t} \gamma^j R(s_j, a_j) \right] \\ &= \mathbb{E} \left[\mathbb{E} \left[\nabla_\theta \log \pi_\theta(a_t|s_t) | s_0, a_0, \ldots, s_{t-1}, a_{t-1}, s_t \right] \cdot \sum_{0 \leq j < t} \gamma^j R(s_j, a_j) \right] \\ &= 0 \quad (\text{because } \mathbb{E} \left[\nabla_\theta \log \pi_\theta(a_t|s_t) | s_0, a_0, \ldots, s_{t-1}, a_{t-1}, s_t \right] = 0) \end{aligned}$$

The logic here is that the past reward ($j < t$) will not effect the current decision (gradient). Note that here we used the law of total expectation. The outer expectation in the second line above is over the randomness of $s_0, a_0, ..., a_{t-1}, s_t$, whereas the inner expectation is over the randomness of a_t (conditioned on$s_0, a_0, ..., a_{t-1}, s_t$.)

The second consequence of $\mathbb{E}_{\tau \sim P_\theta} \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \right] = 0$ is the following: for any value $B(s_t)$ that only depends on s_t, it holds that

$$\begin{aligned} &E_{\tau \sim P_\theta}[\nabla_\theta \log \pi_\theta(a_t|s_t) \cdot B(s_t)] \\ &= E \left[E \left[\nabla_\theta \log \pi_\theta(a_t|s_t) | s_0, a_0, \ldots, s_{t-1}, a_{t-1}, s_t \right] B(s_t) \right] \\ &= 0 \quad (\text{because } E \left[\nabla_\theta \log \pi_\theta(a_t|s_t) | s_0, a_0, \ldots, s_{t-1}, a_{t-1}, s_t \right] = 0) \end{aligned}$$

It follows from equation (6) and the equation above that

$$
\begin{aligned}
\nabla_\theta \eta(\theta) &= \sum_{t=0}^{T-1} E_{\tau \sim P_\theta} \left[\nabla_\theta \log \pi_\theta(a_t|s_t) \cdot \sum_{j \geq t}^{T-1} \gamma^j R(s_j, a_j) - \gamma^t B(s_t) \right] \\
&= \sum_{t=0}^{T-1} E_{\tau \sim P_\theta} \left[\nabla_\theta \log \pi_\theta(a_t|s_t) \cdot \left(\sum_{j \geq t}^{T-1} \gamma^{j-t} R(s_j, a_j) - B(s_t) \right) \right]
\end{aligned}
$$

Therefore, we will get a different estimator for estimating the $\nabla \eta(\theta)$ with a difference choice of $B(\cdot)$. The benefit of introducing a proper $B(\cdot)$ — which is often referred to as a **baseline** — is that it helps reduce the variance of the estimator. It turns out that a near optimal estimator would be the expected future payoff $\mathbb{E}\left[\sum_{j \geq t}^{T-1} \gamma^{j-t} R(s_j, a_j) | s_t\right]$, which is pretty much the same as the value function $V^{\pi_\theta}(s_t)$ (if we ignore the difference between finite and infinite horizon.) Here one could estimate the value function $V^{\pi_\theta}(\cdot)$ in a crude way, because its precise value doesn't influence the mean of the estimator but only the variance. This leads to a policy gradient algorithm with baselines stated in algorithm given below.

Vanilla policy gradient with baseline

for $i = 1, \cdots$ **do**

Collect a set of trajectories by executing the current policy. Use $R_{\geq t}$ as a shorthand for $\sum_{j \geq t}^{T-1} \gamma^{j-t} R(s_j, a_j)$.

Fit the baseline by finding a function B that minimizes

$$
\sum_{\tau} \sum_{t} (R_{\geq t} - B(s_t))^2
$$

Update the policy parameter θ with the gradient estimator

$$
\sum_{\tau} \sum_{t} \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot (R_{\geq t} - B(s_t))
$$