Machine Learning

Muchang Bahng

Spring 2024

Contents

Machine learning in the 1980s have been focused on developing a rigorous theory of learning algorithms, and the field has been dominated by statisticians. They strived to develop the theoretical foundation of algorithms that can be implemented and applied to real-world data. With the advent of deep learning, the theory behind state-of-the-art algorithms, mostly black-box models, has slowed down, but their applications have exploded. It is now a field of trying out a bunch of things and sticking to what works.

These set of notes focuses on trying to provide the theoretical foundations of classical, interpretable machine learning algorithms, while my deep learning notes skim over the theory to focus on model architectures and applications (which is plenty to write about on its own). I've spent a good amount of time trying to create a map of machine learning, but after rewriting these notes multiple times. I've come to the conclusion that it is impossible to create a nice chronological timeline. Like math, you keep on revisiting the same topics over and over again, but at a higher level, and it's not as simple to organize everything into, say parametric vs nonparametric[1], supervised vs unsupervised[2], or discriminative vs generative models.[3] Therefore, I've settled (for now) on the following structure. Before you begin, the direct prerequisites are my notes in probability theory, functional analysis, frequenist and Bayesian statistics, and information theory. If you are new to machine learning, go over my notes on Stanford CS229, which simply covers basic algorithms and their implementation.

First, we will establish the theoretical foundations of ML by introducing the most abstract and general formulation of learning: statistical learning theory. In here, we will talk about function classes, what it means for a model to *learn*, what *overfitting* means in a mathematical sense, and some basic probability theorems that give us enough confidence that we can achieve good theoretical results.

Then we move on to supervised learning, which provides review from statistics and gives us ideas to build upon when talking about unsupervised learning.

1. Just like in every other class, we begin with low-dimensional linear regression, introducing it both in the frequentist and Bayesian perspectives. We expand a bit into modified versions of these models, through weighted least squares and different loss functions.
2. Then we consider the need for regularization in high-dimensional linear regression, where the covariates may be much lager than the samples. This leads us to introduce both lasso and ridge as convex approximations of forward stepwise regression. We interpret them in both the frequentist and Bayesian ways.
3. Now that we have covered linear regression, we can move onto *nonparametric regression*.
4. We complete regression and move to *linear classification*, with logistic/softmax regression, SVMs, and DLA, followed by *nonparametric classification*, such as decision trees.
5. It turns out that all these models are a specific instance of a more general model called *generalized linear models (GLMs)*. We develop the theory behind this model in general.
6. We conclude supervised learning by talking about *ensemble methods*, where we can take any combination of the models that we have learned so far to create new models that may have better performance.

Now we move onto unsupervised learning, which consists of every other problem dealing with a dataset without labels. The simplest algorithms are direct-clustering (K-means), direct density estimation (kernel density estimation), or direct dimensionality-reduction (PCA, UMAP). I call them *direct* because they interact with the data directly, as opposed to *latent variable* models that attempts to model the data by adding hidden random variables.

1. Direct clustering and density estimation models cover most algorithms you would learn in a introduction to machine learning course. Besides simple clustering models and kernel density estimation, there is

[1] K nearest neighbors is a nonparameteric model given that the data is not fixed. When the data is fixed, then our function search space is finite.

[2] There are semi-supervised or weakly supervised models, and models like autoencoders use a supervised algorithm without any labels.

[3] Using Bayes rule, we can always reduce generative models into discriminative models.

not a lot to learn here. However, I do focus on the high-dimensional case where we may need some tricks for computational efficiency.

2. Direct dimensionality reduction models tend to have more variants, and I go through the most widely used algorithms.
3. Now we start getting into latent variable models. Just as linear regression is the simplest supervised regression model, linear factor models are the unsupervised analogue, which attempts to model the latent and the visible variables through a linear relationship.
4. We expand on this into nonlinear latent variable models, such as latent variables living in a discrete space. The most popular is Gaussian mixture models, which uses a multinomial latent variable representing which "cluster" a sample lives in.
5. Finally we talk about a generalization of these models by modeling probability distributions with a *graph*, or a *Bayesian network*. This allows for high-dimensional distributions to be represented with much fewer parameters.

At this point we are done, and all that is left is to talk about some meta-training techniques, such as cross validation, information criterion, and practical methods for real-world problems.

For clarification, $\mathcal{D}$ can represent different things depending on the problem:

1. In a density estimation problem, where we have a single dataset $\mathbf{X}$, $\mathcal{D} = \mathbf{X}$ since this data tells us information about which distribution it could come from.
2. In a regression problem, $\mathcal{D} = \mathbf{Y}$, that is, $\mathcal{D}$ will always be the output data, not the input data $\mathbf{X}$. We can think of the input data $\mathbf{X}$ as always being fixed, and it is upon observation of the *outputs* $\mathbf{Y}$ on these inputs that gives us information.

In both the frequentist and Bayesian settings, the likelihood $p(\mathcal{D} \mid \mathbf{w})$ plays a central role. In the frequentist setting, the process is divided into two steps:

1. We optimize $\mathbf{w}$ with some **estimator**, with a popular one being the **maximum likelihood estimator**. A popular estimator is **maximum likelihood**, which seeks to maximize $p(\mathcal{D} \mid \mathbf{w})$ w.r.t. $\mathbf{w}$.
2. We optimize $\mathbf{w}$ with some **estimator**, with a popular one being the **maximum likelihood estimator**. A popular estimator is **maximum likelihood**, which seeks to maximize $p(\mathcal{D} \mid \mathbf{w})$ w.r.t. $\mathbf{w}$.
3. We fix the optimized $\mathbf{w}^*$ and error bars on this estimate are obtained by considering the distribution of possible datasets $\mathcal{D}$. One approach is **bootstrapping**, which goes as follows. Given our original dataset $\mathbf{X} = \{x^{(1)}, \dots, x^{(N)}\}$, we can create a new dataset $\mathbf{X'}$ by sampling N points at random from $\mathbf{X}$, with replacement, so that some points in $\mathbf{X}$ may be replicated in $\mathbf{X}'$, whereas other points in $\mathbf{X}$ may be absent in $\mathbf{X}'$. This process is repeated L times to generate L different datasets. Then, we can look at the variability of prediction between the different bootstrap data sets.

In a Bayesian setting, there is only a single dataset $\mathcal{D}$ and the uncertainity in the parameters is expressed through a probability distribution over $\mathbf{w}$. It also includes prior knowledge naturally in the form of prior distributions.

1 Statistical Learning Theory

Unlike unsupervised learning, which comes in many different shapes and forms (anomaly detection, feature extraction, density estimation, dimensionality reduction, etc.), supervised learning comes in a much cleaner format. In supervised learning, we consider an input space $\mathcal{X}$ and an output space $\mathcal{Y}$. We assume that there exists some unknown measure $\mathbb{P}$ over $\mathcal{X} \times \mathcal{Y}$, making this some probability space. We then assume that some data $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}$ is generated sampled *independently and identically (iid)* from $\mathbb{P}$. Now this assumption is quite strong and is almost always not the case, as different data can be correlated, but we will relax this assumption later. Let's formally construct this from the bottom up.

1. We start off with a general probability space $(\Omega, \mathcal{F}, \mathbb{P})$. This is our model of the world and everything that we are interested in.
2. A measurable function $X : \Omega \to \mathcal{X}$ extracts a set of features, which we call the **covariates** and induces a probability measure on $\mathcal{X}$, say $\mathbb{P}_X$.
3. Another measurable function $Y : \Omega \to \mathcal{Y}$ extracts another set of features called the **labels** and induces another probability measure on $\mathcal{Y}$, the **label set**, say $\mathbb{P}_Y$.
4. At this point the function $X \times Y$ is all we are interested in, and we throw away Ω since we only care about the distribution over $\mathcal{X} \times \mathcal{Y}$.
5. We model the generation of data from Ω by sampling N samples from $\mathbb{P}_{X \times Y}$, which we assume to be iid (this assumption will be relaxed later). This gives us the **dataset**
$$\mathcal{D} = \{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}_{i=1}^N$$

Now our goal is to construct a function $f : \mathcal{X} \to \mathcal{Y}$ that predicts Y from X, but we want to define some measure of how good our function is. We can use a loss function L to talk about this.

Definition 1.1 (Risk)

The **risk**, or **expected risk**, of function f is defined as

$$R(f) = \mathbb{E}_{X \times Y}[L(Y, f(X))] = \int_{\mathcal{X} \times \mathcal{Y}} L(y, f(x))\, d\mathbb{P}(x, y) \tag{1}$$

Clearly, we don't know what this risk is since we don't know the true measure $\mathbb{P}$, so we try to approximate it with the *empirical risk*.

Definition 1.2 (Empirical Risk)

The **empirical risk** of function f is defined as

$$\hat{R}_n(f) = \frac{1}{n} \sum_{i=1}^{n} L(y^{(i)}, f(x^{(i)})) \tag{2}$$

Definition 1.3 (Generlize)

A function f is said to **generalize** if

$$\lim_{n \to +\infty} \hat{R}_n(f) = R(f) \tag{3}$$

This gives us a way of computing with the actual data. Now two questions arise from this. First, how do we even choose the loss function L? Second, how do we know that the empirical risk is a good approximation of the true risk? The first question can be quite convoluted, but we introduce it with decision theory. The second has a simple answer with concentration of measure.

1.1 Decision Theory

How can we choose our loss functions? There are two ways of doing this, either through model assumptions or with domain knowledge. When talking about model assumptions, we assume that the residual distribution is of certain form, and the maximum likelihood formulation leads to a certain loss function. For example, assuming that the residuals are normally distributed leads to the squared loss or Laplacian residuals leads to the absolute value loss. These are just modeling assumptions, and if there are no specific assumptions, we are lost. The other way is through domain expertise which allows us to construct our own loss functions. Fortunately, there is a deeper theory behind the choice of loss functions, known as decision theory, which allows us to define loss functions from the get go rather than assume distributions taking particular forms.[4]

Definition 1.4 (Misclassification Loss)

The **misclassification loss** is defined as

$$L(y, \hat{y}) = \begin{cases} 0 & \text{if } y = \hat{y} \\ 1 & \text{if } y \neq \hat{y} \end{cases} \tag{4}$$

Example 1.1 (Misclassification Risk)

Substituting the misclassification loss function into the risk gives the **misclassification risk**.

$$R(f) = \mathbb{E}[\mathbb{1}_{\{Y \neq f(X)\}}] = \mathbb{P}(Y \neq f(X)) \tag{5}$$

and therefore our empirical risk is

$$\hat{R}(f) = \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}_{\{y^{(i)} \neq f(x^{(i)})\}} \tag{6}$$

which is just the number of misclassifications over the total number of samples.

However, depending on the context, the loss for misclassification one one label can be quite different from that of another label. Consider the medical example where you're trying to detect cancer. Falsely detecting a non-cancer patient as having cancer is not as bad as falsely detecting a cancer patient as not having cancer.

Definition 1.5 (Weighted Misclassification Loss)

The **loss matrix** K defines the loss that we incur when predicting the ith class on a sample with true label j.

$$L(y, \hat{y}) = \begin{cases} 0 & \text{if } y = \hat{y} \\ K_{ij} & \text{if } y = i \neq j = \hat{y} \end{cases} \tag{7}$$

Definition 1.6 (Squared Loss)

The **squared loss** is defined as

$$L(y, \hat{y}) = (y - \hat{y})^2 \tag{8}$$

Example 1.2 (Mean Squared Risk)

Substituting the squared loss function into the risk gives the **mean squared risk**.

$$R(f) = \mathbb{E}[(Y - f(X))^2] \tag{9}$$

[4] Credits to Edric for telling me this.

and therefore our empirical risk is

$$\hat{R}(f) = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - f(x^{(i)})^2 \tag{10}$$

Definition 1.7 (Absolute Loss)

The **absolute loss** is defined as

$$L(y, \hat{y}) = |y - \hat{y}| \tag{11}$$

1.2 Function Classes

Now that we've defined the risk and empirical risk, the true function that we want to find is the one that minimizes the empirical risk.

$$f^* = \underset{f \in \mathcal{F}}{\operatorname{argmin}}\, \hat{R}(f) \tag{12}$$

However, this depends on the function space $\mathcal{F}$ that we are minimizing over. If we chose f to be the space of all functions, then we just interpolate (fit perfectly over) the data[5], which is not good since we're **overfitting**. This is a problem especially in nonparametric supervised learning, and there are generally two ways to deal with this. The first is to use *localization*, which deals with local smoothing methods. The second is with **regularization**. The third is to restrict our class of functions to a smaller set. Perhaps we assume that nature is somewhat smooth and so naturally we want to work with smooth functions. There are two ways that we define smoothness, through Holder spaces that focus on local smoothness and Sobolev spaces that focus on global smoothness.

Definition 1.8 (L^p Space)

The $L^p(\mu)$ space is the normed vector space of all functions from $f : \mathcal{X} \to \mathbb{R}$ such that

$$||f||_p = \left(\int |f(x)|^p \, d\mu \right)^{1/p} < \infty \tag{13}$$

Theorem 1.1 (Countable Basis)

You can construct a countable orthonormal basis in $L^2(\mu)$ space.

There are a lot of well known orthonormal bases. For example, the Fourier basis, Legendre polynomials, Hermite polynomials, or wavelets. Therefore, every function can be expressed as a linear combination of this basis, and you can calculate coefficients by taking the inner product with the basis functions.

$$f(x) = \sum_{i=1}^{\infty} \alpha_i \phi_i(x) \text{ and } \alpha_i = \langle f, \phi_i \rangle \tag{14}$$

Now we can define Holder spaces. Holder spaces are used whenever we want to talk about local smoothness. For example, when we want to talk about local smoothing methods for regression and classification, talking about this smoothing is not quite possible if we don't have certain assumptions on the function. To make theory easier, we assume that the function has basic smoothness properties and this property is Holder smoothness. But note that these are ultimately assumptions.

[5] unless there were two different values of Y for the same X

Definition 1.9 (Holder Space)

For some $\beta \in \mathbb{N}$ and $L \in \mathbb{R}^+$, the $H(\beta, L)$ **Holder space** is the set of all functions $f : \mathcal{X} \subset \mathbb{R} \to \mathbb{R}$ such that

$$|f^{(\beta-1)}(y) - f^{(\beta-1)}(x)| \leq L||y - x|| \tag{15}$$

for all x, y. If we want $\mathcal{X}$ to be d-dimensional, then we want to bound the higher order total derivatives, and so $H(\beta, L)$ becomes all functions $f : \mathcal{X} \subset \mathbb{R}^d \to \mathbb{R}$ such that for all $\mathbf{s} = (s_1, \dots, s_d)$ with $|\mathbf{s}| = \beta - 1$,

$$|D^{\mathbf{s}} f(x) - D^{\mathbf{s}} f(x)| \leq L||y - x|| \tag{16}$$

for all $x, y \in \mathcal{X}$, where

$$D^{\mathbf{s}} = \frac{\partial^{|\mathbf{s}|}}{\partial x_1^{s_1} \dots \partial x_d^{s_d}} \tag{17}$$

The higher β is, the more smoothness we're demanding.

If $\beta = 1$, then this reduces to the set of all Lipschitz functions. It is most common to assume that $\beta = 2$, which means that the derivative is Lipschitz. This is not rigorously true, but by dividing both sides by $||y - x||$ and taking the limit to 0, we can say that it implies that there exists some finite second derivative bounded by L.

Definition 1.10 (Sobolev Space)

The **Sobelov space** $W_{m,p}$ is the space of all functions $f \in L_p(\mu)$ such that

$$||D^m f||_p \in L^p(\mu) \tag{18}$$

This is slightly stronger than the usual definition of Sobolev spaces since we requiring the derivative rather than the weak derivative. So m tells us how many derivatives we want well behaved and p tells us under which norm are the derivatives well behaved.

Now there is a related definition of a Sobelov ellipsoid that we'll be working with.

Definition 1.11 (Sobelov Ellipsoid)

Let $\theta = (\theta_1, \theta_2, \dots)$ be a sequence of real numbers. Then the set

$$\Theta_m = \left\{ \theta \mid \sum_{j=1}^{\infty} a_j^2 \theta_j^2 < C^2 \right\} \tag{19}$$

where $a_j^2 = (\pi \cdot j)^{2m}$. Note that since a_j is exploding, to stay finite the θ_j must be decaying.

This is useful because of the following theorem.

Theorem 1.2 (Conditions for Function being in Sobelov Space)

Given a function $f \in L^2(\mu)$ expanded in some orthonormal basis ϕ_j, then $f \in W_{m,2}$ if and only if the coefficients α_j die off fast enough in the sense that it is in the Sobelov ellipsoid.

Now let's talk about RKHS. Let's take the $L^2(\mu)$ space of functions $f : [0, 1] \to \mathbb{R}$ with $||f|| = \int f^2 \, d\mu < \infty$ and inner product $\langle f, g \rangle = \int f(x) g(x) \, d\mu$. It is known that if f_n converges to f in L^2, then it is not necessarily true that f converges pointwise since it can diverge on a sequence of sets that converge to measure 0. You probably don't want to work with functions that look like this, and that's what a RKHS is for. It gives you a nice class of functions that have good statistical properties but also are easy to compute with.

Definition 1.12 (Mercer Kernels)

A **Mercer kernel** is a function $K : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ that is symmetric and positive definite in the sense that for any collection $x_1, \dots, x_n$ of arbitrary size n,

$$\sum_i \sum_j c_i c_j K(x_i, x_j) \geq 0 \tag{20}$$

which is equivalent to saying that the matrix formed by evaluating these kernels at the pairs of points is positive semi-definite.

Example 1.3 (Gaussian Kernel)

The Gaussian kernel is defined

$$K(x, y) = \exp\left(-\frac{||x - y||^2}{\sigma^2} \right) \tag{21}$$

Now this kernel should tell us roughly how similar two points x and y are. Using this kernel, we want to build a function space. For this, we need Mercer's theorem.

Theorem 1.3 (Mercer's Theorem)

If we have a kernel K that is bounded

$$\sup_{x,y} K(x, y) < \infty \tag{22}$$

we can define a new operator T_K that maps functions to functions

$$T_K f(x) = \int K(x, y) f(y)\, dy = \iint K(x, y) f(x) f(y)\, dx\, dy \tag{23}$$

This operator is linear, meaning that it has an eigendecomposition and therefore there exists eigenfunctions ϕ_i s.t.

$$T_K \phi_i(x) = \int K(x, y) \phi_i(y)\, dy = \lambda_i \phi_i(x) \tag{24}$$

Then these eigenvalues are bounded and we can write the kernel as a sum of the eigenfunctions.

$$\sum_i \lambda_i < \infty, \quad K(x, y) = \sum_{i=1}^{\infty} \lambda_i \phi_i(x) \phi_i(y) \tag{25}$$

These ϕ_i's are the implicit high-dimensional features.

What do these eigenfunctions ϕ_i look like? Well, they tend to look like functions that tend to get wigglier and wigglier as i increases, indicating that λ_i must decrease in such a way that it still keeps the function smooth.

Now, we can fix the first term in the kernel and it will be function of the second term $K_x(\cdot) = K(x, \cdot)$. We do this for all $x \in \mathbb{R}$, which form the basis of our RKHS, and it consists of all functions that are linear combinations of these K_x's. For example, the functions

$$f = \sum_i \alpha_i K_{x_i} \text{ and } g = \sum_j \beta_j K_{x_j} \tag{26}$$

can consist of a finite number of perhaps different basis functions. Now this is clearly a vector space, and to upgrade this to a Hilbert space, we must define an inner product. This inner product (with respect to some

kernel K) is defined as

$$\langle f, g \rangle_K = \sum_{i,j} \alpha_i \beta_j K(x_i, x_j) \tag{27}$$

Exercise 1.1 (Inner Product of RKHS)

Show that the inner product of the RKHS is indeed an inner product.

The inner product induces a norm, and so by taking the completion of all linear combinations of the kernel basis functions we get our RKHS. Now since K_x is itself in the RKHS, we can take the inner product of f and K_x, which just gives us back the evaluation of f at x.

Definition 1.13 (Reproducing Kernel Hilbert Space)

Given a kernel K, the **reprducing kernel Hilbert space** $\mathcal{H}$ is the Hilbert space of all functions $f : \mathcal{X} \to \mathcal{Y}$ that can be expressed as a linear combination of the functions $\{K_x = K(x, \cdot)\}$. It has the inner product

$$\langle f, g \rangle_{\mathcal{H}} = \sum_{i,j} \alpha_i \beta_j K(x_i, x_j) \tag{28}$$

and also includes all of its limit points under this norm, making it a complete space.

Theorem 1.4 (Reproducing Property of RKHS)

An RKHS satisfies the **reproducing property**, which means that taking the inner product of a function f and a kernel K_x gives you the evaluation of f at x.

$$\langle f, K_x \rangle_{\mathcal{H}} = f(x) \tag{29}$$

and therefore it also means that $\langle K_x, K_x \rangle_{\mathcal{H}} = K(x, x)$. This also means that K_x is the evaluation functional in the dual space of $\mathcal{H}$ and this evaluation functional δ_x is continuous, which is not always true in functional analysis.

Proof.

We can evaluate from the inner product

$$f = \sum_i \alpha_i K_{x_i} \implies \langle f, K_x \rangle_K = \sum_i \alpha_i \langle K_{x_i}, K_x \rangle_K = \sum_i \alpha_i K(x_i, x) = f(x) \tag{30}$$

This reproducing property tends to be very useful, especially in the corollary below.

Corollary 1.1 (Convergence in RKHS)

Convergence in norm implies pointwise convergence in RKHS.

Proof.

Given that $f_n \to f$ in norm, we have that $||f_n - f|| \to 0$. Then for all points $x \in \mathcal{X}$,

$$|f_n(x) - f(x)| = |\langle f_n - f, K_x \rangle_{\mathcal{H}}| \leq ||f_n - f|| \cdot ||K_x|| \to 0 \tag{31}$$

Theorem 1.5 (Moore-Aronszajn)

Any positive definite function K is a reproducing kernel for some RKHS.

Proof.

We won't be too rigorous about this since this is not a functional analysis course. Assume that we have a positive definite kernel $K : X \times X \to \mathbb{R}$, where X is some measurable set, and we will show how to make a RKHS $\mathcal{H}_k$ such that K is the reproducing kernel on $\mathcal{H}$. It turns out that $\mathcal{H}_k$ is unique up to isomorphism. Since X exists, let us first define the set $S = \{k_x \mid x \in X\}$ such that $k_x(y) \coloneqq K(x, y)$. Now let us define the vector space V to be the span of S. Therefore, each element $v \in V$ can be written as

$$v = \sum_i \alpha_i k_{x_i}$$

Now we want to define an inner product on V. By expanding out the vectors w.r.t. the basis and the properties of bilinearity, we have

$$\langle k_x, k_y \rangle_V = \left\langle \sum_i \alpha_i k_{x_i}, \sum_i \beta_i k_{y_i} \right\rangle = \sum_{i,j} \alpha_i \beta_j K(x_i, y_j)$$

At this point, V is not necessarily complete, but we can force it to be complete by taking the limits of all Cauchy sequences and adding them to V. In order to complete the construction, we need to ensure that K is continuous and doesn't diverge, i.e.

$$\iint K^2(x, y)\, dx\, dy < +\infty$$

which is a property known as finite trace.[a]

[a] Too much to write down here at this point, but for further information look at *thearticlehere*.

Now at first glance, this abstract construction makes it hard to determine what kind of functions there are in a RKHS generated by some kernel. Conversely, given some RKHS, it's not always easy to know which kernel it came from.

Example 1.4 (Fourier Basis)

Let us take the vector space of all real functions f for which its Fourier transform is supported on some finite interval $[-a, a]$. This is a RKHS with the kernel function

$$K(x, y) = \frac{\sin(a(y - x))}{a(y - x)} \tag{32}$$

with the inner product $\langle f, g \rangle = \int f(x) g(x)\, dx$.

Example 1.5 (Some Sobelov Spaces are RKHS)

Let us take the Sobelov space $W_{1,2}$ of all functions $f : [0, 1] \to \mathbb{R}$ satisfying

$$\int (f'(x))^2\, dx < \infty \tag{33}$$

This is a RKHS with the kernel function

$$K(x,y) = \begin{cases} 1 + xy + \frac{xy^2}{2} - \frac{y^3}{6} & \text{if } 0 \leq y \leq x \leq 1 \\ 1 + xy + \frac{x^2 y}{2} - \frac{x^3}{6} & \text{if } 0 \leq x \leq y \leq 1 \end{cases} \tag{34}$$

Finally, remembering Mercer's theorem, we can decompose the Kernel into its eigenfunctions

$$K(x,y) = \sum_{j=1}^{\infty} \lambda_j \phi_j(x) \phi_j(y) \tag{35}$$

When you talk about feature maps (e.g. in support vector machines), you're really just creating the map from $x \in \mathcal{X}$ into the infinite dimensional vector space

$$x \mapsto \Phi(x) = \left(\sqrt{\lambda_1}\phi_1(x), \sqrt{\lambda_2}\phi_2(x), \dots \right) \tag{36}$$

and the inner product between two functions is actually the inner product between their feature maps. Therefore, you can either just work with x in the RKHS or work with the features Φ in a higher dimensional Euclidean space. Therefore, we can either work with f as a combination of kerenels or a linear combination of the eigenfunctions. The eigenfunctions are easier conceptually, but when we actually do computations, the kernel expansion is much easier.

$$f(x) = \sum_i \alpha_i K(x_i, x) = \sum_j \beta_j \phi_j(x) \tag{37}$$

When you're expanding with the eigenfunctions, you can just compute the inner product as

$$\langle f, g \rangle = \sum_i \frac{\alpha_i \beta_i}{\lambda_i} \tag{38}$$

and because f, g must satisfy some smoothness constraints, the α_i and β_i must die off quickly, making the sum finite. But we're never going to be actually computing this way since it's much easier to compute with the kernel expansion. This means that the ϕ_i's, which get wigglier (think of sine and cosine eigenbases) as i increases, must have decreasing coefficients.

When working with function classes, we tend to divide them into two broad categories.

Definition 1.14 (Parametric Models)

A **parametric model** is a set of functions $\mathcal{M}_{\boldsymbol{\theta}}$ that can be parameterized by a finite-dimensional vector. The elements of this model are hypotheses functions $h_{\boldsymbol{\theta}}$, with the subscript used to emphasize that its parameters are $\boldsymbol{\theta}$. We have the flexibility to choose any form of h that we want, and that is ultimately a model assumption that we are making.

Example 1.6 (Examples of Parametric Models)

1. If we assume $h : \mathbb{R}^D \to \mathbb{R}$ to be linear, then h lives in the dual of $\mathbb{R}^D$, which we know to be D-dimensional.
2. If we assume h to be affine, then this just adds one more dimension.
3. If we assume $h : \mathbb{R} \to \mathbb{R}$ to be a kth degree polynomial, then g can be parameterized by a $k+1$ dimensional θ.

However, parametric models may be limited in the way that we are assuming some form about the data. For certain forms of data, where we may have domain knowledge, it is reasonable to use parametric models, but there are cases when we will have absolutely no idea what the underlying distribution is. For example, think of classifying a $3 \times N \times N$ image as a cat or a dog. There is some underlying distribution in the space

$[255]^{3N^2} \times \{\text{cat}, \text{dog}\}$, but we have absolutely no idea how to parameterize this. Should it be a linear model or something else? This is when nonparametric models come in. They are not restricted by the assumptions concerning the nature of the population from which the sample is drawn.

Definition 1.15 (Nonparametric Models)

Nonparametric models are ones that cannot be expressed in a finite set of parameters. They may be countably or uncountably infinite.

1.3 Concentration of Measure

Concentration of measure is a tool used to prove a lot of theorems in statistical machine learning. I have another series of notes on this, but we'll stick to the key points.

Definition 1.16 (Hoeffding's Inequality)

Given $X_1, \dots, X_n$ are iid random variables with $a \leq X_i \leq b$, then for any $\epsilon > 0$,

$$\mathbb{P}\left(\left| \frac{1}{n} \sum_{i=1}^{n} X_i - \mathbb{E}[X] \right| \geq \epsilon \right) \leq 2 \exp\left(- \frac{2n\epsilon^2}{(b-a)^2} \right) \tag{39}$$

Therefore, if we apply it to some binary classifier $f : \mathcal{X} \to \{0, 1\}$, then we can say that the probability that the empirical risk deviates from the true risk is exponentially small.

$$\mathbb{P}(|\hat{R}(f) - R(f)| \geq \epsilon) \leq 2e^{-2n\epsilon^2} \tag{40}$$

But when we do empirical risk minimization (ERM), we not given a classifier, but we must *choose* it. So given our space of classifiers f, we can plot the true risk and the noisy empirical risk. The equation above states that at any given point the probability of it deviating by more than ϵ is exponentially small. But we want something stronger: we want to bound the probability of the supremum of the difference over the whole class $\mathcal{F}$.

$$\mathbb{P}\big(\sup_{f \in \mathcal{F}} |\hat{R}(f) - R(f)| \geq \epsilon \big) \tag{41}$$

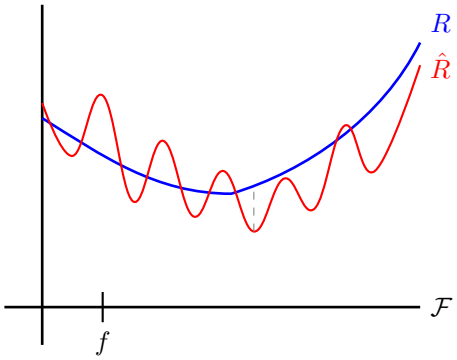


Figure 1: True risk of functions over $\mathcal{F}$ and its noisy empirical risk. We want to bound the maximum deviation of these two over the whole class.

This bound will depend on how *complex* the function class $\mathcal{F}$ is, and to measure this complexity, we introduce some definitions.

Definition 1.17 (Rademacher Complexity)

Given **Rademacher random variables** $\sigma_1, \dots, \sigma_n$ with $\mathbb{P}(\sigma_i = 1) = \mathbb{P}(\sigma_i = -1) = \frac{1}{2}$, the **Rademacher complexity** of a function class $\mathcal{F}$ is defined

$$\text{Rad}_n(\mathcal{F}) = \mathbb{E}\left[\sup_{f \in \mathcal{F}} \left| \frac{1}{n} \sum_{i=1}^{n} \sigma_i f(Z_i) \right| \right] \tag{42}$$

where the expectation is across the random σ_i's and the Z_i's, which are independent.

To get some intuition of what this is, let's consider a function class of a single function f. Then, the sup disappears and the term inside the absolute value sign becomes a 0-mean random variable. Now if we have a very complex function class $\mathcal{F}$ with a lot of "wiggly" functions, then this value should be large. In this case, imagine a game where you pick generate some random variables σ_i and the Z_i. Then, I pick a function that maximizes this value. How can I do that? If I can find a function f that matches the sign of the σ_i's ($+1$ or -1) at each of the values of Z_i, then this would be maximized. Therefore, if I have a sufficiently complex class, then I can pick a function that tracks your σ_i's. Another way of looking at it is given noise variables σ and Z, we're looking at the correlation between σ and $f(Z)$. If we can maximize this correlation, then this is a complex class.

Now this is the most natural way of defining the complexity of the class, and in some cases it can be explicitly computed. However, in most cases it cannot be, but it can be bounded be something that is computable, like the VC dimension.

Lemma 1.1 (Bigger Class, Bigger Complexity)

If $\mathcal{F} \subset \mathcal{G}$, then $\text{Rad}_n(\mathcal{F}) \leq \text{Rad}_n(\mathcal{G})$.

Lemma 1.2 (Convex Hull)

If $\mathcal{F}$ is a convex set, then $\text{Rad}_n(\mathcal{F}) = \text{Rad}_n(\text{conv}(\mathcal{F}))$, where $\text{conv}(\mathcal{F})$ is the convex hull of $\mathcal{F}$.

This lemma is quite useful since if we have a certain finite set of functions, then their convex hull can encompass quite a bit, and we can also easily compute that convex hull's Rademacher complexity. Since the extremes haven't changed, the complexity doesn't change, and this might suggest that the Rademacher complexity is a good measure.

Lemma 1.3 (Change of Complexity with Lipschitz Functions)

Consider a L-Lipschitz function g with $g(0) = 0$ and consider the class $\mathcal{F}$, then we can bound the class of functions $g \circ \mathcal{F} = \{g \circ f \mid f \in \mathcal{F}\}$.

$$\text{Rad}_n(g \circ \mathcal{F}) \leq 2L\text{Rad}_n(\mathcal{F}) \tag{43}$$

This constant multiplicative bound is also useful.

Definition 1.18 (Projection of Function Class onto Points)

Given a binary function class $\mathcal{F}$ with functions $f : \mathcal{X} \to \{0, 1\}$, let us denote the projection of $\mathcal{F}$ onto a set of points $z_1, \dots, z_n \in \mathcal{X}$ to be

$$\mathcal{F}_z = \mathcal{F}_{z_1,\dots,z_n} = \{(f(z_1), \dots, f(z_n)) \mid f \in \mathcal{F}\} \tag{44}$$

This projection determines the set of all possible binary labels that can be perfectly classified by some

function f.

Definition 1.19 (Shattering Number)

The **shattering number** of $\mathcal{F}$ is defined

$$s_n(\mathcal{F}) = s(\mathcal{F}, n) = \sup_{z_1,\dots,z_n} |\mathcal{F}_{z_1,\dots,z_n}| \tag{45}$$

The highest number that this can be is 2^n, since this is the number of possible binary vectors of length n. Given a set of n points $z_1, \dots, z_n$, we say that the function class $\mathcal{F}$ **shatters** this set if $\mathcal{F}_{z_1,\dots,z_n} = |2^n|$. That is, for every one of the 2^n labels on the points, there exists a function that can perfectly classify them.

Example 1.7 (Binary Functions)

Consider the function class $\mathcal{F}$ of all binary functions of the form

$$f(x) = \begin{cases} 1 & \text{if } x > t \\ 0 & \text{if } x \leq t \end{cases} \tag{46}$$

Then, the projection of $\mathcal{F}$ onto some $n = 3$ points is the set

$$\{(0,0,0), (0,0,1), (0,1,1), (1,1,1)\} \tag{47}$$

and this is true no matter how I pick the z_1, z_2, z_3, and so the Shattering number is $s_n(\mathcal{F}) = 4$.

Definition 1.20 (VC Dimension)

We know that the shattering number is bounded above by 2^n. For $n = 1$, it is reasonable that it achieves this bound, but as n grows, the Shattering number may die off. The **VC dimension** is the largest n number of points that can be shattered by the function class without misclassification.

$$n^{\text{VC}} := \sup_n \{s_n(\mathcal{F}) = 2^n\} \tag{48}$$

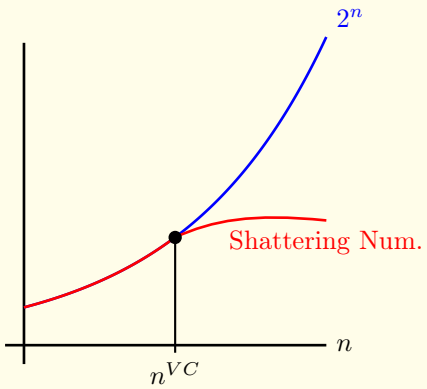


Figure 2: The Shattering number of $\mathcal{F}$ will grow exponentially until it reaches the VC dimension, at which point it will grow polynomially. The point at which it "dies off" is the VC dimension.

It turns out that there are very interesting properties about the VC dimension. One such fact is Sawyer's

lemma, which states that if the VC dimension is finite, then the rate of growth of the shattering number suddenly changes from exponential 2^n to polynomial n^{VC}, and this is what makes a lot of machine learning work.

Definition 1.21 (Subgaussian Random Variables)

A random variable X is **subgasussian** if

$$\mathbb{E}[e^{\lambda X}] \leq e^{\frac{\lambda^2\sigma^2}{2}} \tag{49}$$

Gaussians and bounded random variables are subgaussian.

Lemma 1.4 (Bound on Subgaussian Random Variables)

Given a set of iid subgaussian random variables $X_1, \dots, X_n$

$$\mathbb{E}\big[\max_{1\leq i\leq d} X_i\big] \leq \sigma\sqrt{2\log d} \tag{50}$$

Theorem 1.6 (Bound of Rademacher Complexity with Shattering Number)

The Rademacher complexity of a binary function class $\mathcal{F}$ is bounded by

$$\text{Rad}_n(\mathcal{F}) \leq \sqrt{\frac{2\log s_n(\mathcal{F})}{n}} \tag{51}$$

Proof.

Given the projection $\mathcal{F}_{z_1,\dots,z_n}$, we can use the law of iterated expectations on the Rademacher complexity.

$$\text{Rad}_n(\mathcal{F}) = \mathbb{E}\left[\sup_{f\in\mathcal{F}}\left|\frac{1}{n}\sum_{i=1}^n \sigma_i f(Z_i)\right|\right] \tag{52}$$

$$= \mathbb{E}_Z\left[\mathbb{E}_\sigma\left[\sup_{f\in\mathcal{F}}\left|\frac{1}{n}\sum_{i=1}^n \sigma_i f(Z_i)\right| \;\middle|\; Z_1, \dots Z_n\right]\right] \tag{53}$$

Note that in the inner expectation, since $f(Z_i)$ is now fixed, then are bounding a linear combination of a bunch of σ_i's, which are subgaussian. Using the bound above, we can reduce it to

$$\mathbb{E}_Z\left[\sqrt{\frac{2\log|F_{z_1,\dots,z_n}|}{n}}\right] \leq \sqrt{\frac{2\log s_n(\mathcal{F})}{n}} \leq \sqrt{\frac{2d\log n}{n}} \tag{54}$$

However, this is not the best possible bound, and in cases such as K means clustering in high dimensions, this VC bound is terrible. Now we move onto the big VC theorem which now bounds the supremum of the difference between the empirical risk and the true risk. To prove this, we need a few tricks, the first being the symmetrization trick using ghost samples.

Lemma 1.5 (Symmetrization Lemma)

Given a set of random variables $Z_1, \dots, Z_n$ and a function class $\mathcal{F}$, we can define ghost samples $Z_1', \dots, Z_n'$ that are iid copies of $Z_1, \dots, Z_n$. Then, we can bound the Rademacher complexity of the

function class $\mathcal{F}$ by

$$\mathbb{P}\left(\sup_{f \in \mathcal{F}} |\hat{R}(f) - R(f)| \geq \epsilon \right) \leq 2\mathbb{P}\left(\sup_{f \in \mathcal{F}} |\hat{R}(f) - \hat{R}'(f)| \geq \epsilon/2 \right) \tag{55}$$

where $\hat{R}, \hat{R}'$ is the empirical risk over the original and ghost samples, respectively.

Proof.

Assume that we have a function f that achieves this minimum. By the triangle inequality,

$$|\hat{R}(f) - R(f)| > t \text{ and } |\hat{R}'(f) - R(f)| < \frac{t}{2} \implies |\hat{R}(f) - \hat{R}'(f)| > \frac{t}{2} \tag{56}$$

We write this again as an indicator function.

$$\mathbb{1}(|\hat{R}(f) - R(f)| > t, |\hat{R}'(f) - R(f)| < \frac{t}{2}) \implies \mathbb{1}(|\hat{R}(f) - \hat{R}'(f)| > \frac{t}{2}) \tag{57}$$

and since the samples and the ghost samples are independent, we can take the probability over the ghost samples to get

$$\mathbb{1}(|\hat{R}(f) - R(f)| > t)\, \mathbb{P}_{Z'}(|\hat{R}'(f) - R(f)| < \frac{t}{2}) \implies \mathbb{P}_{Z'}(|\hat{R}(f) - \hat{R}'(f)| > \frac{t}{2}) \tag{58}$$

and the rest of the proof can be found online.

The reason we want this is that it removes the $R(f)$, which is some unknown true mean that can be hard to deal with since it takes infinite values. It's easier to work with two empirical risks than deal with the true risk.

Theorem 1.7 (VC Theorem/Inequality)

Given a binary function class $\mathcal{F}$, we have

$$\mathbb{P}\left(\sup_{f \in \mathcal{F}} |\hat{R}(f) - R(f)| \geq \epsilon \right) \leq 2S(\mathcal{F}, n)e^{-n\epsilon^2/8} \approx n^d e^{-n\epsilon^2/8} \tag{59}$$

You can see that the exponential term is from Hoeffding but there is an extra cost of taking the supremum over the whole function class, which is the shattering number.

Proof.

Given $Z_1, \dots, Z_n \sim \mathbb{P}$, we take a new set of random variables $Z_1', \dots, Z_n'$ that are iid copies of $Z_1, \dots, Z_n$, called *ghost samples*.

Therefore, for some classes of sets with finite VC dimension, the shattering term will grow polynomially in n but the exponential term decays faster, which is what makes this work. That's why as n grows, we can get a good bound on the supremum of this difference.

Theorem 1.8 ()

With probability $\geq 1-\delta$, we have

$$\sup_{f\in\mathcal{F}} |\hat{R}(f) - R(f)| \leq 2\text{Rad}_n(\mathcal{F}) + \sqrt{\frac{\log(2/\delta)}{2n}} \tag{60}$$

1.4 Bias Variance Noise Decomposition

Let's do some further analysis on this. When you take a supremum over a function class, it decomposes into 3 terms.

1. One of which quantifies how big the function class is (more variance).
2. One of which quantifies the distance between the truth and the function class (bias).
3. One is the noise term, which is the irreducible error.

Example 1.8 (Bias and Variance Tradeoff in Polynomial Regression)

Let's motivate this by trying to fit a polynomial on some data.

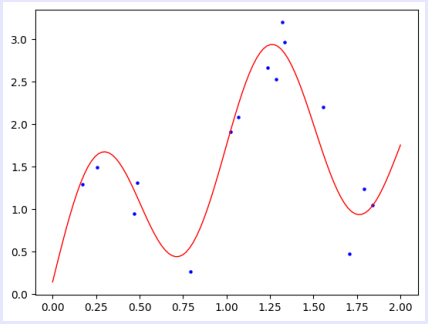
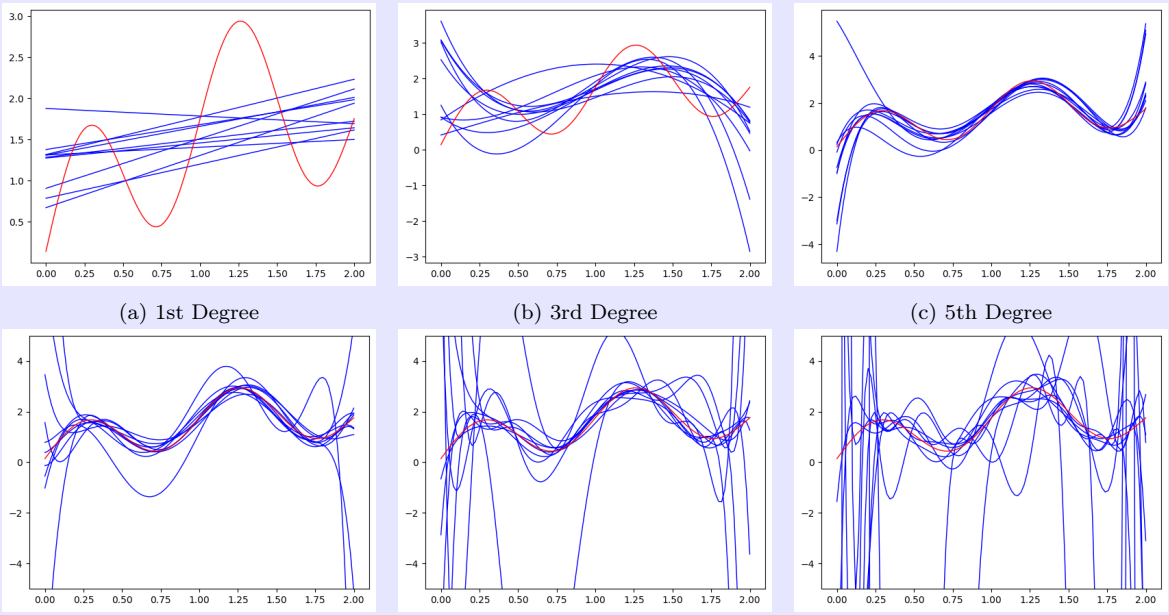


Figure 3: A sample of $|\mathcal{D}| = 15$ data points are generated from the function $f(x) = \sin(2\pi x) + 2\cos(x - 1.5)$ with Gaussian noise $N(0, 0.3)$ on the interval $[0, 1]$.

If we try to fit a polynomial function, how do we know which degree is best? Well the most simple thing is to just try all of them. To demonstrate this even further, I generated 10 different datasets $\mathcal{D}$ of size 15 taken from the same true distribution. The best fitted polynomials for each dataset is shown below.



(a) 1st Degree (b) 3rd Degree (c) 5th Degree

(d) 7th Degree (e) 9th Degree (f) 11th Degree

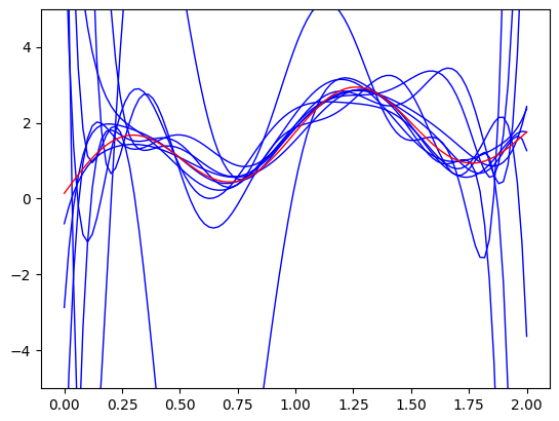
Figure 4: Different model complexities (i.e. different polynomial degrees) lead to different fits of the data generated from the true distribution. The lower degree best fit polynomials don't have much variability in their best fits but have high bias, while the higher degree best fit polynomials have very high variability in their best fits but have low bias. The code used to generate this data is here.

We already know that the 5th degree approximation is most optimal, and the lower degree ones are **underfitting** the data, while the higher degree ones are **overfitting**. As mentioned before, we can describe the underfitting and overfitting phenomena through the bias variance decomposition.

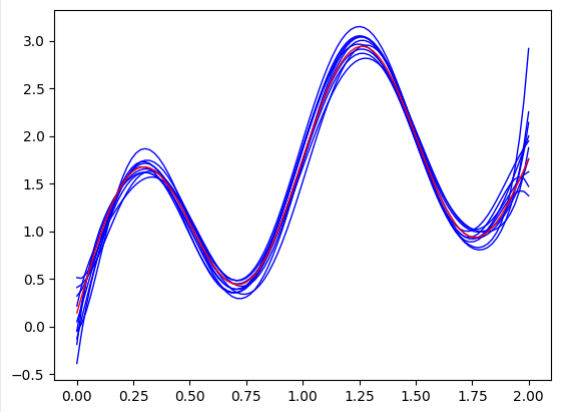
1. If we underfit the data, this means that our model is not robust and does not capture the patterns inherent in the data. It has a high bias since the set of function it encapsulates is not large enough to model $\mathbb{E}[Y \mid X]$. However, it has a low variance since if we were to take different samples of the dataset $\mathcal{D}$, the optimal parameters would not fluctuate.
2. What overfitting essentially means is that our model is too complex to the point where it starts to fit to the *noise* of the data. This means that the variance is high, since different samples of the dataset $\mathcal{D}$ would cause huge fluctuations in the optimal trained parameters $\boldsymbol{\theta}$. However, the function set would be large, and thus it would be close to $\mathbb{E}[Y \mid X]$, leading to a low bias.

Example 1.9 (Polynomial Regression Continued)

Another way to reduce the overfitting problem is if we have more training data to work with. That is, if we were to fit a 9th degree polynomial on a training set of not $N = 15$, but $N = 100$ data points, then we can see that this gives a much better fit. This makes sense because now the random variable $\mathcal{D}$, as a function of more random variables, has lower variance. Therefore, the lower variance in the dataset translates to lower variance in the optimal parameter.



(a) $M = 9, N = 15$



(b) $M = 9, N = 100$

Figure 5: Increasing the number of data points helps the overfitting problem. Now, we can afford to fit a 9th degree polynomial with reasonable accuracy.

1.5 Minimax Theory

2 Low Dimensional Linear Regression

In introductory courses, we start with linear predictors since it is easy to understand. We still start with linear predictors because in high-dimensional machine learning, even linear prediction can be hard as we will see. Low dimensional linear regression is what statisticians worked in back in the early days, where data was generally low dimensional.[6] Generally, we had $d < n$, but these days, we are in the regime where $d > n$. For example, in genetic data, you could have a sample of $n = 100$ people but each of them have genetic sequences at $d = 10^6$. When the dimensions become high, the original methods of linear regression tend to break down, which is why I separate low and high dimensional linear regression. The line tends to be fuzzy between these two regimes, but we will not worry about strictly defining that now.

In here, we start with **multiple linear regression**, which assumes that we have several covariates and one response. If we extend this to multiple responses (i.e. a response vector), this is called **multivariate linear regression**. The simple case for one response is called **simple linear regression**, and we will mention some nice formulas and intuition that come out from working with this.

Definition 2.1 (Linear Regression Model)

Given a dataset $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^n$, where $x^{(i)} \in \mathbb{R}^d$ with $x_1 = 1$ (which is what we do in practice to include an intercept term) and $y^{(i)} \in \mathbb{R}$, the multiple linear regression model is

$$y = \beta^T x + \epsilon \tag{61}$$

with the following assumptions:

1. *Weak exogeneity*: the covariates are observed without error.
2. *Linearity*: the mean of the variate is a linear combination of the parameters and the covariates.
3. *Gaussian errors*: the errors are Gaussian.[a]
4. *Homoscedasticity*: the errors (the observations of Y) have constant variance.
5. *Independence of errors*: The errors are uncorrelated.
6. *No multicollinearity*: more properly, the lack of perfect multicollinearity. Assume that the covariates aren't perfectly correlated.[b]

[a] We can relax this assumption when we get into generalized linear models, and in most cases we assume some closed form of the error for computational convenience, like when computing the maximum likelihood.

[b] This is the assumption that breaks down in high dimensional linear regression.

In order to check multicollinearity, we compute the correlation matrix.

Definition 2.2 (Correlation Matrix)

The correlation matrix of random variables $X_1, \dots, X_d$ is

$$\mathbf{C}_{ij} = \text{Corr}(X_i, X_j) = \frac{\text{Cov}(X_i, X_j)}{\sigma_{X_i} \sigma_{X_j}}$$

given that $\sigma_{X_i} \sigma_{X_j} > 0$. Clearly, the diagonal entries are 1, but if there are entries that are very close to 1, then we have multicollinearity.

Assume that two variables are perfectly correlated. Then, there would be pairs of parameters that are indistinguishable if moved in a certain linear combination. This means that the variance of $\hat{\boldsymbol{\beta}}$ will be very ill conditioned, and you would get a huge standard error in some direction of the β_i's. We can fix this by making sure that the data is not redundant and manually removing them, standardizing the variables, making a change of basis to remove the correlation, or just leaving the model as it is.

If these assumptions don't hold,

[6] Quoting Larry Wasserman, even 5 dimensions was considered high and 10 was considered massive.

1. *Weak exogeneity*: the sensitivity of the model can be tested to the assumption of weak exogeneity by doing bootstrap sampling for the covariates and seeing how the sampling affects the parameter estimates. Covariates measured with error used to be a difficult problem to solve, as they required errors-in-variables models, which have very complicated likelihoods. In addition, there is no universal fitting library to deal with these. But nowadays, with the availability of Markov Chain Monte Carlo (MCMC) estimation through probabilistic programming languages, it is a lot easier to deal with these using Bayesian hierarchical models (or multilevel models, or Bayesian graphical models—these have many names).
2. *Linearity*: the linear regression model only assumes linearity in the parameters, not the covariates. Therefore you could build a regression using non-linear transformations of the covariates, for instance,

$$Y = X_1\beta_1 + X_1^2\beta_2 + \log(X_1)\beta_3 \tag{62}$$

   If you need to further relax the assumption, you are better off using non-linear modelling.
3. *Constant variance*: the simplest fix is to do a variance-stabilising transformation on the data. Assuming a constant coefficient of variation rather than a constant mean could also work. Some estimation libraries (such as the `glm` package in R) allow specifying the variance as a function of the mean.
4. *Independence of errors*: this is dangerous because in the financial world things are usually highly correlated in times of crisis. The most important thing is to understand how risky this assumption is for your setting. If necessary, add a correlation structure to your model, or do a multivariate regression. Both of these require significant resources to estimate parameters, not only in terms of computational power but also in the amount of data required.
5. *No multicollinearity*: If the covariates are correlated, they can still be used in the regression, but numerical problems might occur depending on how the fitting algorithms invert the matrices involved. The t-tests that the regression produces can no longer be trusted. All the covariates must be included regardless of what their significance tests say. A big problem with multicollinearity, however, is overfitting. Depending on how bad the situation is, the parameter values might have huge uncertainties around them, and if you fit the model using new data their values might change significantly.[7] Multicollinearity is a favourite topic of discussion for quant interviewers, and they usually have strong opinions about how it should be handled. The model's intended use will determine how sensitive it is to ignoring the error distribution. In many cases, fitting a line using least-squares estimation is equivalent to assuming errors have a normal distribution. If the real distribution has heavier tails, like the t-distribution, how risky will it make decisions based on your outputs? One way to address this is to use a technique like robust-regression. Another way is to think about the dynamics behind the problem and which distribution would be best suited to model them—as opposed to just fitting a curve through a set of points.

2.1 Ordinary Least Squares

If we use a squared loss function, this is called **ordinary least squares**. It is a well known fact that the true regressor that minimizes this loss is

$$f^*(x) = \mathbb{E}[Y \mid X = x] \tag{63}$$

which is the conditional expectation of Y given X. This is the true regressor function, which is the best approximation of Y over the σ-algebra generated by X. This may or may not be linear.

[7] I suggest reading this Wikipedia article on multicollinearity, as it contains useful information: `https://en.wikipedia.org/wiki/Multicollinearity`

Theorem 2.1 (Least Squares Solution For Linear Regression)

Given the design matrix $\mathbf{X}$, we can present the linear model in vectorized form:

$$\mathbf{Y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon},\ \boldsymbol{\epsilon} \sim N(\mathbf{0}, \sigma^2\mathbf{I}) \tag{64}$$

The solution that minimizes the squared loss is

$$\boldsymbol{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y} \in \mathbb{R}^d$$
$$\text{Var}(\hat{\boldsymbol{\beta}}) = \hat{\sigma}^2(\mathbf{X}^T\mathbf{X})^{-1} \in \mathbb{R}^{d\times d}$$

Proof.

The errors can be written as $\boldsymbol{\epsilon} = \mathbf{Y} - \mathbf{X}\boldsymbol{\beta}$, and you have the following total sum of squared errors:

$$S(\boldsymbol{\beta}) = \boldsymbol{\epsilon}^T\boldsymbol{\epsilon} = (\mathbf{Y} - \mathbf{X}\boldsymbol{\beta})^T(\mathbf{Y} - \mathbf{X}\boldsymbol{\beta})$$

We want to find the value of $\boldsymbol{\beta}$ that minimizes the sum of squared errors. In order to do this, remember the following matrix derivative rules when differentiating with respect to vector $\mathbf{x}$.

1. $\mathbf{x}^T\mathbf{A} \mapsto \mathbf{A}$
2. $\mathbf{x}^T\mathbf{A}\mathbf{x} \mapsto 2\mathbf{A}\mathbf{x}$

Now this should be easy.

$$\begin{aligned} S(\boldsymbol{\beta}) &= \mathbf{Y}^T\mathbf{Y} - \boldsymbol{\beta}^T\mathbf{X}^T\mathbf{Y} - \mathbf{Y}^T\mathbf{X}\boldsymbol{\beta} + \boldsymbol{\beta}^T\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} \\ &= \mathbf{Y}^T\mathbf{Y} - 2\mathbf{Y}^T\mathbf{X}\boldsymbol{\beta} + \boldsymbol{\beta}^T\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} \\ \frac{\partial}{\partial\boldsymbol{\beta}}S(\boldsymbol{\beta}) &= -2\mathbf{X}^T\mathbf{Y} + 2\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} \end{aligned}$$

and setting it to $\mathbf{0}$ gives

$$2\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} - 2\mathbf{X}^T\mathbf{Y} = 0 \implies \mathbf{X}^T\mathbf{X}\boldsymbol{\beta} = \mathbf{X}^T\mathbf{Y}$$

and the variance of $\boldsymbol{\beta}$, by using the fact that $\text{Var}[\mathbf{A}\mathbf{X}] = \mathbf{A}\,\text{Var}[X]\mathbf{A}^T$, is

$$\text{Var}(\hat{\boldsymbol{\beta}}) = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\,\sigma^2\mathbf{I}\,\mathbf{X}(\mathbf{X}'\mathbf{X})^{-1} = \sigma^2(\mathbf{X}'\mathbf{X})^{-1}(\mathbf{X}'\mathbf{X})(\mathbf{X}'\mathbf{X})^{-1} = \sigma^2(\mathbf{X}'\mathbf{X})^{-1}$$

But we don't know the true σ^2, so we estimate it with $\hat{\sigma}^2$ by taking the variance of the residuals. Therefore, we have

$$\boldsymbol{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y} \in \mathbb{R}^d$$
$$\text{Var}(\hat{\boldsymbol{\beta}}) = \hat{\sigma}^2(\mathbf{X}^T\mathbf{X})^{-1} \in \mathbb{R}^{d\times d}$$

Example 2.1 (Copying Data)

What happens if you copy your data in OLS? In this case, our MLE estimate becomes

$$\left(\begin{pmatrix}X\\X\end{pmatrix}^T\begin{pmatrix}X\\X\end{pmatrix}\right)^{-1}\begin{pmatrix}X\\X\end{pmatrix}^T\begin{pmatrix}Y\\Y\end{pmatrix} =$$
$$= (X^TX + X^TX)^{-1}(X^TY + X^TY) = (2X^TX)^{-1}2X^TY = \hat{\beta}$$

and our estimate is unaffected. However, the variance shrinks by a factor of 2 to

$$\frac{\sigma^2}{2}(\mathbf{X}^T\mathbf{X})^{-1} \tag{65}$$

A consequence of that is that confidence intervals will shrink with a factor of $1/\sqrt{2}$. The reason is that we have calculated as if we still had iid data, which is untrue. The pair of doubled values are obviously dependent and have a correlation of 1.

Another way to solve the solution is through likelihood estimation.

Theorem 2.2 (Maximum Likelihood Estimation of Linear Regression)

Given a dataset $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^N$, our likelihood is

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

We can take its negative log, remove additive constants, and scale accordingly to get

$$\begin{aligned} \ell(\theta) &= -\frac{N}{2} \ln \sigma^2 - \frac{N}{2} \ln(2\pi) + \frac{1}{2\sigma^2} \sum_{i=1}^N \left(y^{(i)} - \boldsymbol{\theta}^T \mathbf{x}^{(i)}\right)^2 \\ &= \frac{1}{2} \sum_{i=1}^N \left(y^{(i)} - \boldsymbol{\theta}^T \mathbf{x}^{(i)}\right)^2 \end{aligned}$$

which then corresponds to minimizing the sum of squares error function.

Theorem 2.3 (Gradient Descent for Linear Regression)

Taking the gradient of this log likelihood w.r.t. θ gives

$$\nabla_\theta \ell(\theta) = \sum_{i=1}^N (y^{(i)} - \theta^T x^{(i)}) x^{(i)}$$

and running gradient descent over a minibatch $M \subset \mathcal{D}$ gives

$$\begin{aligned} \theta &= \theta - \eta \nabla_\theta \ell(\theta) \\ &= \theta - \eta \sum_{(x,y) \in M} (y - \theta^T x) x \end{aligned}$$

This is guaranteed to converge since $\ell(\theta)$, as the sum of convex functions, is also convex.
Note that since we can solve this in closed form, by setting the gradient to 0, we have

$$0 = \sum_{n=1}^N y^{(n)} \boldsymbol{\phi}(\mathbf{x}^{(n)})^T - \mathbf{w}^T \left(\sum_{n=1}^N \boldsymbol{\phi}(\mathbf{x}^{(n)}) \boldsymbol{\phi}(\mathbf{x}^{(n)})^T \right)$$

which is equivalent to solving the least squares equation

$$\mathbf{w}_{ML} = (\boldsymbol{\Phi}^T \boldsymbol{\Phi})^{-1} \boldsymbol{\Phi}^T \mathbf{Y}$$

Note that if we write out the bias term out explicitly, we can see that it just accounts for the translation (difference) between the average of the outputs $\bar{y} = \frac{1}{N} \sum_{n=1}^N y_n$ and the average of the basis functions $\bar{\phi_j} = \frac{1}{N} \sum_{n=1}^N \phi_j(\mathbf{x}^{(n)})$.

$$w_0 = \bar{y} - \sum_{j=1}^{M-1} w_j \bar{\phi_j}$$

We can also maximize the log likelihood w.r.t. σ^2, which gives the MLE

$$\sigma^2_{ML} = \frac{1}{N}\sum_{n=1}^{N}\left(y^{(n)} - \mathbf{w}_{ML}^T\boldsymbol{\phi}(\mathbf{x}^{(n)})\right)^2$$

Code 2.1 (MWE for OLS Linear Regression in scikit-learn)

Here is a minimal working example of performing linear regression with scikit-learn. Note that the input data must be of shape (n, d).

```python
import numpy as np
from sklearn.linear_model import LinearRegression

X = np.array([[1, 1], [1, 2], [2, 2], [2, 3]])
y = np.dot(X, np.array([1, 2])) + 3

model = LinearRegression()
model.fit(X, y)
print(X)
print(y)
print(model.score(X, y))
print(model.intercept_)
print(model.coef_)
print(model.predict(np.array([[3, 5]])))
```

```
[[1 1]
 [1 2]
 [2 2]
 [2 3]]
[ 6  8  9 11]
1.0
3.0000000000000018
[1. 2.]
[16.]
.
.
.
.
.
```

2.1.1 Bias Variance Decomposition

We can use what we have learned to prove a very useful result for the mean squared loss.

Theorem 2.4 (Pythagorean's Theorem)

The expected square loss over the joint measure $\mathbb{P}_{X\times Y}$ can be decomposed as

$$\mathbb{E}_{X\times Y}[(Y - h(X))^2] = \mathbb{E}_{X\times Y}[\left(Y - \mathbb{E}[Y \mid X]\right)^2] + \mathbb{E}_X[\left(\mathbb{E}[Y \mid X] - h(X)\right)^2] \tag{66}$$

That is, the squared loss decomposes into the squared loss of $\mathbb{E}[Y \mid X]$ and $g(X)$, which is the intrinsic misspecification of the model, plus the squared difference of Y with its best approximation $\mathbb{E}[Y \mid X]$, which is the intrinsic noise inherent in Y beyond the σ-algebra of X.

Proof.

We can write

$$\begin{aligned}
\mathbb{E}_{X\times Y}[L] &= \mathbb{E}_{X\times Y}\left[\left(Y - g(X)\right)^2\right] \\
&= \mathbb{E}_{X\times Y}\left[\left((Y - \mathbb{E}[Y \mid X]) + (\mathbb{E}[Y \mid X] - g(X))\right)^2\right] \\
&= \mathbb{E}_{X\times Y}[\left(Y - \mathbb{E}[Y \mid X]\right)^2] + \mathbb{E}_{X\times Y}[\{Y - \mathbb{E}[Y \mid X]\}\{\mathbb{E}[Y \mid X] - g(X)\}] \\
&\qquad + \mathbb{E}_X[\left(\mathbb{E}[Y \mid X] - g(X)\right)^2] \\
&= \mathbb{E}_{X\times Y}[\left(Y - \mathbb{E}[Y \mid X]\right)^2] + \mathbb{E}_X[\left(\mathbb{E}[Y \mid X] - g(X)\right)^2]
\end{aligned}$$

where the middle term cancels out due to the tower property.

We also proved a second fact: Since $\mathbb{E}[\big(\mathbb{E}[Y \mid X] - g(X)\big)^2]$ is the misspecification of the model, we cannot change this (positive) constant, so $\mathbb{E}\big[\big(Y - g(X)\big)^2\big] \geq \mathbb{E}[(Y - \mathbb{E}[Y \mid X])^2]$, with equality achieved when we perfectly fit g as $\mathbb{E}[Y \mid X]$ (i.e. the model is well-specified). Therefore, denoting $\mathcal{F}$ as the set of all $\sigma(X)$-measurable functions, then the minimum of the loss is attained when

$$\underset{g \in \mathcal{F}}{\operatorname{argmin}}\, \mathbb{E}[L] = \underset{g \in \mathcal{F}}{\operatorname{argmin}}\, \mathbb{E}\big[\big(Y - g(X)\big)^2\big] = \mathbb{E}[Y \mid X] \tag{67}$$

Even though this example is specific for the mean squared loss, this same decomposition, along with the bias variance decomposition, exists for other losses. It just happens so that the derivations are simple for the MSE, which is why this is introduced first. However, the derivations for other losses are much more messy, and sometimes may not hold rigorously. However, the general intuition that more complex models tend to overfit still hold true.

Now if we approximate $\mathbb{E}[Y \mid X]$ with our parameterized hypothesis $h_{\boldsymbol{\theta}}$, then from a Bayesian perspective the uncertainty in our model is expressed through a poseterior distribution over $\boldsymbol{\theta}$. A frequentist treatment, however, involves making a point estimate of $\boldsymbol{\theta}$ based on the dataset $\mathcal{D}$ and tries instead to interpret the uncertainty of this estimate through the following thought experiment: Suppose we had a large number of datasets each of size N and each drawn independently from the joint distribution $X \times Y$. For any given dataset $\mathcal{D}$, we can run our learning algorithm and obtain our best fit function $h^*_{\boldsymbol{\theta};\mathcal{D}}(\mathbf{x})$. Different datasets from the ensemble will give different functions and consequently different values of the squared loss. The performance of a particular learning algorithm is then assessed by taking the average over this ensemble of datasets, which we define $\mathbb{E}_{\mathcal{D}}[h_{\boldsymbol{\theta};\mathcal{D}}(\mathbf{x})] = \mathbb{E}_{(X \times Y)^N}[h_{\boldsymbol{\theta};\mathcal{D}}(\mathbf{x})]$. We are really taking an expectation over all datasets, meaning that the N points in each $\mathcal{D}$ must be sampled from $(X \times Y)^N$.

Consider the term $\big(\mathbb{E}[Y \mid X] - h_{\boldsymbol{\theta}}(X)\big)^2$ above, which models the discrepancy in our optimized hypothesis and the best approximation. Now, over all datasets $\mathcal{D}$, there will be a function $h_{\boldsymbol{\theta};\mathcal{D}}$, and averaged over all datasets $\mathcal{D}$ is $\mathbb{E}_{\mathcal{D}}[h_{\boldsymbol{\theta};\mathcal{D}}]$. So, the random variable below (of $\mathcal{D}$ and X) representing the stochastic difference between our optimized function $h_{\boldsymbol{\theta};\mathcal{D}}(X)$ and our best approximation $\mathbb{E}[Y \mid X]$ can be decomposed into

$$\begin{aligned}
\big(\mathbb{E}[Y \mid X] - h_{\boldsymbol{\theta};\mathcal{D}}(X)\big)^2 &= \big[\big(\mathbb{E}[Y \mid X] - \mathbb{E}_{\mathcal{D}}[h_{\boldsymbol{\theta};\mathcal{D}}(X)]\big) + \big(\mathbb{E}_{\mathcal{D}}[h_{\boldsymbol{\theta};\mathcal{D}}(X)] - h_{\boldsymbol{\theta};\mathcal{D}}(X)\big)\big]^2 \\
&= \big(\mathbb{E}[Y \mid X] - \mathbb{E}_{\mathcal{D}}[h_{\boldsymbol{\theta};\mathcal{D}}(X)]\big)^2 + \big(\mathbb{E}_{\mathcal{D}}[h_{\boldsymbol{\theta};\mathcal{D}}(X)] - h_{\boldsymbol{\theta};\mathcal{D}}(X)\big)^2 \\
&\quad + 2\big(\mathbb{E}[Y \mid X] - \mathbb{E}_{\mathcal{D}}[h_{\boldsymbol{\theta};\mathcal{D}}(X)]\big)\big(\mathbb{E}_{\mathcal{D}}[h_{\boldsymbol{\theta};\mathcal{D}}(X)] - h_{\boldsymbol{\theta};\mathcal{D}}(X)\big) \\
&= \big(\mathbb{E}[Y \mid X] - \mathbb{E}_{\mathcal{D}}[h_{\boldsymbol{\theta};\mathcal{D}}(X)]\big)^2 + \big(\mathbb{E}_{\mathcal{D}}[h_{\boldsymbol{\theta};\mathcal{D}}(X)] - h_{\boldsymbol{\theta};\mathcal{D}}(X)\big)^2
\end{aligned}$$

Averaging over all datasets $\mathcal{D}$ causes the middle term to vanish and gives us the expected squared difference between the two random variables, now of X.

Theorem 2.5 (Bias Variance Decomposition)

Therefore, we can write out the expected square difference between $h_{\boldsymbol{\theta}}$ and $\mathbb{E}[Y \mid X]$ as the sum of two terms.

$$\mathbb{E}_{\mathcal{D}}\big[\big(\mathbb{E}[Y \mid X] - h_{\boldsymbol{\theta}}(X)\big)^2\big] = \underbrace{\big(\mathbb{E}[Y \mid X] - \mathbb{E}_{\mathcal{D}}[h_{\boldsymbol{\theta};\mathcal{D}}(X)]\big)^2}_{(\text{bias})^2} + \underbrace{\mathbb{E}_{\mathcal{D}}\big[\big(\mathbb{E}_{\mathcal{D}}[h_{\boldsymbol{\theta};\mathcal{D}}(X)] - h_{\boldsymbol{\theta};\mathcal{D}}(X)\big)^2\big]}_{\text{variance}} \tag{68}$$

Let us observe what these terms mean:

1. The **bias** $\mathbb{E}[Y \mid X] - \mathbb{E}_{\mathcal{D}}[h_{\boldsymbol{\theta};\mathcal{D}}(X)]$ is a random variable of X that measures the difference in how the average prediction of our hypothesis function $\mathbb{E}_{\mathcal{D}}[h_{\boldsymbol{\theta};\mathcal{D}}(X)]$ differs from the actual prediction $\mathbb{E}[Y \mid X]$.
2. The **variance** $\mathbb{E}_{\mathcal{D}}\big[\big(\mathbb{E}_{\mathcal{D}}[h_{\boldsymbol{\theta};\mathcal{D}}(X)] - h_{\boldsymbol{\theta};\mathcal{D}}(X)\big)^2\big]$ is a random variable of X that measures the variability of each hypothesis function $h_{\boldsymbol{\theta}}(X)$ about its mean over the ensemble $\mathbb{E}_{\mathcal{D}}[h_{\boldsymbol{\theta};\mathcal{D}}(X)]$.

Therefore, we can substitute this back into our Pythagoras decomposition, where we must now take the expected bias and the expected variance. Therefore, we get

$$\text{Expected Loss} = (\text{Expected Bias})^2 + \text{Expected Variance} + \text{Noise} \tag{69}$$

where

$$\begin{aligned}(\text{Bias})^2 &= \mathbb{E}_X\left[\left(\mathbb{E}[Y \mid X] - \mathbb{E}_{\mathcal{D}}[h_{\boldsymbol{\theta};\mathcal{D}}(X)]\right)^2\right] \\ \text{Variance} &= \mathbb{E}_X\left[\mathbb{E}_{\mathcal{D}}\left[\left(\mathbb{E}_{\mathcal{D}}[h_{\boldsymbol{\theta};\mathcal{D}}(X)] - h_{\boldsymbol{\theta};\mathcal{D}}(X)\right)^2\right]\right] \\ \text{Noise} &= \mathbb{E}_{X\times Y}[\left(Y - \mathbb{E}[Y \mid X]\right)^2]\end{aligned}$$

2.1.2 Convergence Bounds

Let's get a deeper understanding on linear regression by examining the convergence of the empirical risk minimizer to the true risk minimizer. We can develop a naive bound using basic concentration of measure.

Theorem 2.6 (Exponential Bound)

Let $\mathcal{P}$ be the set of all distributions for $X \times Y$ supported on a compact set. There exists constants c_1, c_2 s.t. that the following is true. For any $\epsilon > 0$,

$$\sup_{\mathbb{P}\in\mathcal{P}} \mathbb{P}^n\left(r(\hat{\beta}_n) > r(\beta_*(\mathbb{P}) + 2\epsilon)\right) \leq c_1 e^{-nc_2\epsilon^2} \tag{70}$$

Hence

$$r(\hat{\beta}_n) - r(\beta_*) = O_{\mathbb{P}}\left(\sqrt{\frac{1}{n}}\right) \tag{71}$$

Proof.

However, this is not a very tight bound, and we can do better. Though the proof is very long and will be omitted.

Theorem 2.7 (Gyorfi, Kohler, Krzyzak, Walk, 2002 [1])

Let $\sigma^2 = \sup_x \text{Var}[Y \mid X = x] < \infty$. Assume that all random variables are bounded by $L < \infty$. Then

$$\mathbb{E}\int |\hat{\beta}^T x - m(x)|^2 \, d\mathbb{P}(x) \leq 8 \inf_{\beta} \int |\beta^T x - m(x)|^2 \, d\mathbb{P}(x) + \frac{Cd(\log(n)+1)}{n} \tag{72}$$

You can see that the bound contains a term of the form

$$\frac{d\log(n)}{n} \tag{73}$$

and under the low dimensional case, d is small and bound is good. However, as d becomes large, then we don't have as good of theoretical guarantees.

Theorem 2.8 (Central Limit Theorem of OLS)

We have

$$\sqrt{n}(\hat{\beta} - \beta) \xrightarrow{d} N(0, \Gamma) \tag{74}$$

where

$$\Gamma = \Sigma^{-1}\mathbb{E}\left[(Y - X^T\beta)^2 XX^T\right]\Sigma^{-1} \tag{75}$$

The covariance matrix Γ can be consistently estimated by

$$\hat{\Gamma} = \hat{\Sigma}^{-1}\hat{M}\hat{\Sigma}^{-1} \tag{76}$$

where

$$\hat{M}(j,k) = \frac{1}{n}\sum_{i=1}^{n} X_i(j)X_i(k)\hat{\epsilon}_i^2 \tag{77}$$

and $\hat{\epsilon}_i = Y_i - \hat{\beta}^T X_i$.

2.2 Simple Linear Regression

The simple linear regression is the special case of the linear regression with only one covariate.[8]

$$y = \alpha + x\beta \tag{78}$$

which is just a straight line fit. Interviewers like this model for its aesthetically pleasing theoretical properties. A few of them are described here, beginning with parameter estimation. For n pairs of (x_i, y_i),

$$y_i = \alpha + \beta x_i + \epsilon_i \tag{79}$$

To minimize the sum of squared errors

$$\sum_i \epsilon_i^2 = \sum_i (y_i - \alpha - \beta x_i)^2 \tag{80}$$

Taking the partial derivatives w.r.t. α and β and setting them equal to 0 gives

$$\sum_i (y_i - \hat{\alpha} - \hat{\beta}x_i) = 0$$
$$\sum_i (y_i - \hat{\alpha} - \hat{\beta}x_i)x_i = 0$$

From just the first equation, we can write

$$n\bar{y} = n\hat{\alpha} + n\hat{\beta}\bar{x} \implies y = \hat{\alpha} + \hat{\beta}\bar{x} \implies \hat{\alpha} = \bar{y} - \hat{\beta}\bar{x} \tag{81}$$

The second equation gives

$$\sum_i x_i y_i = \hat{\alpha} n\bar{x} + \hat{\beta}\sum_i x_i^2 \tag{82}$$

and substituting what we derived gives

$$\begin{aligned}\sum_i x_i y_i &= (\bar{y} - \hat{\beta}\bar{x})n\bar{x} + \hat{\beta}\sum_i x_i^2 \\ &= n\bar{x}\bar{y} + \hat{\beta}\left(\left(\sum_i x_i^2\right) - n\bar{x}^2\right)\end{aligned}$$

and so we have

$$\hat{\beta} = \frac{\left(\sum_i x_i y_i\right) - n\bar{x}\bar{y}}{\left(\sum x_i^2\right) - n\bar{x}^2} = \frac{\sum_i x_i y_i - \bar{x}y_i}{\sum x_i^2 - \bar{x}x_i} = \frac{\sum_i (x_i - \bar{x})y_i}{\sum_i (x_i - \bar{x})x_i} \tag{83}$$

Now we can use the identity

$$\sum_i (x_i - \bar{x})(y_i - \bar{y}) = \sum_i y_i(x_i - \bar{x}) = \sum_i x_i(y_i - \bar{y})$$

[8] I've included a separate section on this since this was especially important for quant interviews.

to substitute both the numerator and denominator of the equation to

$$\hat{\beta} = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sum_i (x_i - \bar{x})^2} = \frac{\text{cov}(x,y)}{\text{var}(x)} = \rho_{xy} \frac{s_y}{s_x}$$

where ρ_{xy} is the correlation between x and y, and the variance and covariance represent the sample variance and covariance (indicated in lower case letters). Therefore, the correlation coefficient ρ_{xy} is precisely equal to the slope of the best fit line when x and y have been standardized first, i.e. $s_x = s_y = 1$.

Example 2.2 (Switching Variables)

Say that we are fitting Y onto X in a simple regression setting with MLE β_1, and now we wish to fit X onto Y. How will the MLE slope change? We can see that

$$\beta_1 = \rho \frac{s_y}{s_x}, \quad \beta_2 = \rho \frac{s_x}{s_y}$$

and so

$$\beta_2 = \rho^2 \frac{1}{\rho} \frac{s_x}{s_y} = \rho^2 \frac{1}{\beta_1} = \beta_1 \frac{\text{var}(x)}{\text{var}(y)}$$

The reason for this is because regression lines don't necessarily correspond to one-to-one to a casual relationship. Rather, they relate more directly to a conditional probability or best prediction.

The **coefficient of determination** R^2 is a measure tells you how well your line fits the data. When you have your y_i's, their deviation around its mean is captured by the sample variance $s_y^2 = \sum_i (y_i - \bar{y})^2$. When we fit our line, we want the deviation of y_i around our predicted values $\hat{y}_i$, i.e. our sum of squared loss $\sum_i (y_i - \hat{y}_i)^2$, to be lower. Therefore, we can define

$$R^2 = 1 - \frac{\text{MSELoss}}{\text{var}(y)} = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2}$$

In simple linear regression, we have

$$R^2 = \rho_{yx}^2$$

An R^2 of 0 means that the model does not improve prediction over the mean model and 1 indicates perfect prediction. However, a drawback of R^2 is that it can increase if we add predictors to the regression model, leading to a possible overfitting.

Theorem 2.9 ()

The residual sum of squares (RSS) is equal to the a proportion of the variance of the y_i's.

$$\text{RSS} = \sum (y_i - \hat{y}_i)^2 = (1 - \rho^2) \sum (y_i - \bar{y})^2 \tag{84}$$

2.3 Weighted Least Squares

2.4 Mean Absolute Error

2.5 Significance Tests

This is not as emphasized in the machine learning literature, but it is useful to know from a statistical point of view.[9]

[9] This is also asked in quant interviews.

2.5.1 T Test

Given some multilinear regression problem where we must estimate $\boldsymbol{\beta} \in \mathbb{R}^{D+1}$ (D coefficients and 1 bias), we must determine whether there is actually a linear relationship between the x and y variables in our dataset $\mathcal{D}$. Say that we have a sample of N points $\mathcal{D} = \{(x_n, y_n)\}_{n=1}^N$. Then, for each ensemble of datasets $\mathcal{D}$ that we sample from the distribution $(X \times Y)^N$, we will have some estimator $\boldsymbol{\beta}$ for each of them. This will create a sampling distribution of $\boldsymbol{\beta}$'s where we can construct our significance test on.

So what should our sampling distribution of $\hat{\boldsymbol{\beta}}$ be? It is clearly normal since it is just a transformation of the normally distributed Y: $\hat{\boldsymbol{\beta}} \sim N(\boldsymbol{\beta}, \sigma^2(X^TX)^{-1})$. Therefore, only considering one element β_i here,

$$\frac{\hat{\beta}_i - \beta_i}{\sigma\sqrt{(X^TX)^{-1}_{ii}}} \sim N(0,1)$$

But the problem is that we don't know the true σ^2, and we are estimating it with $\hat{\sigma}^2$. If we knew the true σ^2 then this would be a normal, but because of this estimate, our normalizing factor is also random. It turns out that the residual sum of squares (RSS) for a multiple linear regression

$$\sum_i (y_i - x_i^T\beta)^2$$

follows a χ^2_{n-d} distribution. Additionally from the χ^2 distribution of RSS we have

$$\frac{(n-d)\hat{\sigma}^2}{\sigma^2} \sim \chi^2_{n-d}$$

where we define $\hat{\sigma}^2 = \frac{\text{RSS}}{n-d}$ which is an unbiased estimator for σ^2. Now there is a theorem that says that if you divide a $N(0,1)$ distribution by a χ^2_k/k distribution (with k degrees of freedom), then it gives you a t-distribution with the same degrees of freedom. Therefore, we divide

$$\frac{\frac{\hat{\beta}_i - \beta_i}{\sqrt{(X^TX)^{-1}_{ii}}}}{\hat{\sigma}} = \frac{\sigma \sim N(0,1)}{\sigma\chi^2_{n-d}/(n-d)} = \frac{\sim N(0,1)}{\chi^2_{n-d}/(n-d)} = t_{n-d}$$

where the standard error of the distribution is

$$\text{SE}(\hat{\beta}_i) = \sigma_{\hat{\beta}_i} = \sigma\sqrt{(X^TX)^{-1}_{ii}}$$

In ordinary linear regression, we have the null hypothesis $h_0 : \beta_i = 0$ and the alternative $h_a : \beta_i \neq 0$ for a two sided test or $h_a : \beta_i > 0$ for a one sided test. Given a certain significance level, we compute the critical values of the t-distribution at that level and compare it with the test statistic

$$t = \frac{\hat{\beta} - 0}{\text{SE}(\hat{\beta})}$$

Now given our β, how do we find the standard error of it? Well this is just the variance of our estimator $\boldsymbol{\beta}$, which is $\hat{\sigma}^2(\mathbf{X}^T\mathbf{X})^{-1}$, where $\hat{\sigma}^2$ is estimated by taking the variance of the residuals ϵ_i. When there is a single variable, the model reduces to

$$y = \beta_0 + \beta_1 x + \epsilon$$

and

$$\mathbf{X} = \begin{pmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{pmatrix}, \qquad \boldsymbol{\beta} = \begin{pmatrix} \beta_0 \\ \beta_1 \end{pmatrix}$$

and so

$$(\mathbf{X}'\mathbf{X})^{-1} = \frac{1}{n\sum x_i^2 - (\sum x_i)^2}\begin{pmatrix} \sum x_i^2 & -\sum x_i \\ -\sum x_i & n \end{pmatrix}$$

and substituting this in gives

$$\sqrt{\widehat{\text{Var}}(\hat{\beta}_1)} = \sqrt{[\hat{\sigma}^2(\mathbf{X}'\mathbf{X})^{-1}]_{22}} = \sqrt{\frac{\hat{\sigma}^2}{\sum x_i^2 - (\sum x_i)^2}} = \sqrt{\frac{\hat{\sigma}^2}{\sum (x_i - \bar{x}_i)^2}}$$

Example 2.3 ()

Given a dataset

```
Hours Studied for Exam 20 16 20 18 17 16 15 17 15 16 15 17 16 17 14
Grade on Exam 89 72 93 84 81 75 70 82 69 83 80 83 81 84 76
```

The hypotheses are $h_0 : \beta = 0$ and $h_a : \beta \neq 0$, and the degrees of freedom for the t-test is $df = N - (D+1) = 13$, where $N = 15$ is the number of datapoints and $D = 1$ is the number of coefficients (plus the 1 bias term). The critical values is ± 2.160, which can be found by taking the inverse CDF of the t-distribution evaluated at 0.975.
Now we calculate the t score. We have our estimate $\beta_1 = 3.216, \beta_0 = 26.742$, and so we calculate

$$\hat{\sigma}^2 = \frac{1}{15}\sum_{i=1}^{15}\big(y_i - (3.216x_i + 26.742)\big) = 13.426$$

$$\sum_i (x_i - \hat{x}_i)^2 = 41.6$$

and therefore, we can compute

$$t = \frac{\beta_1}{\sqrt{\hat{\sigma}^2 / \sum_i (x_i - \hat{x}_i)^2}} = \frac{3.216}{\sqrt{13.426/41.6}} = 5.661$$

and therefore, this is way further than our critical value of 2.16, meaning that we reject the null hypothesis.

Note that when multicolinearity is present, then $\sum_i (x_i - \hat{x}_i)^2$ will be very small causing the denominator to blow up, and therefore you cannot place too much emphasis on the interpretation of these statistics. While it is hard to see for the single linear regression case, we know that some eigenvalue of $(\mathbf{X}^T\mathbf{X})^{-1}$ will blow up, causing the diagonal entries $(\mathbf{X}^T\mathbf{X})^{-1}_{ii}$ to be very small. When we calculate the standard error by dividing by this small value, the error blows up.

Theorem 2.10 ()

We can compute this t-statistic w.r.t. just the sample size n and the correlation coefficient ρ as such.

$$t = \frac{\hat{\beta} - 0}{\text{SE}(\hat{\beta})}$$

and the denominator is simply

$$\text{SE}(\hat{\beta}) = \sqrt{\frac{\frac{1}{n-1}\sum (y_i - \hat{y})^2}{\sum (x_i - \bar{x})^2}} \implies t = \frac{\hat{\beta}\sqrt{\sum (x_i - \bar{x})^2}\sqrt{n-1}}{\sqrt{\sum (y_i - \hat{y})^2}} = \frac{\hat{\beta}\sqrt{\sum (x_i - \bar{x})^2}\sqrt{n-1}}{\sqrt{(1-\rho^2)}\sqrt{\sum (y_i - \bar{y})^2}}$$
$$= \frac{\rho}{\sqrt{1-\rho^2}}\sqrt{n-1}$$

where the residual sum of squares on the top can be substituted according to our theorem. Therefore

$$t = \frac{\rho}{\sqrt{1-\rho^2}}\sqrt{n-1} \tag{85}$$

2.5.2 F Test

Given that you have n data points that have been fit on a linear model, the F-statistic is based on the ratio of two variances.

2.6 Bayesian Linear Regression

3 High Dimensional Linear Regression

Now supposed that $d > n$, then the first problem is that we can no longer use least squares since X^TX is no longer invertible and the same problem happens with maximum likelihood. This is known as the **high dimensional** or **large p, small n** problem. The most straightforward way is simply to reduce the covariates to a dimension smaller than n. This can be done with three ways.

1. We perform PCA on the X and use the first k principal components where $k < n$.
2. We cluster the covariates based on their correlation. We can use one feature from each cluster or take the average of the covariates within each cluster.
3. We can screen the variables by choosing the k features that have the largest correlation with Y.

Once this is done, we are back in the low dimensional regime and can use least squares. Essentially, this is a way to find a good subset of the covariates, which can be formalized by the following. Let S be a subset of $[d]$ and let $X_S = (X_j : j \in S)$. If the size of S is not too large, we can regress Y on X_S instead of X.

Definition 3.1 (Best Subset Regression)

Fix $k < d$ and let $\mathcal{S}_k$ denote all subsets of size k. For a given $S \in \mathcal{S}_k$, let β_S be the best linear predictor for the subset S. We want to find the best subset S that minimizes the loss

$$\mathbb{E}[(Y - \beta_S^T X_S)^2] \tag{86}$$

which is equivalent to finding

$$\underset{\beta}{\operatorname{argmin}}\, \mathbb{E}[(Y - \beta^T X)^2] \text{ subject to } ||\beta||_0 \leq k \tag{87}$$

where $||\beta||_0$ is the number of non-zero entries in β.

There will be a bias variance tradeoff. As k increases, the bias decreases but the variance increases. We can approximate the risk with the training error, but the minimization is over all subset of size k, and so this is NP-hard. Therefore, best subset regression is infeasible, but we can approximate best subset regression in two different ways.

1. A greedy approximation leads to *forward stepwise regression*.
2. A convex relaxation of the problem leads to the *LASSO* regression.

It turns out that the theoretical guarantees and computational time for both are the same, but the Lasso is much more popular. It may be due to a cleaner form or that it's easier to study, but who knows.

A completely separate way is to use all the covariates, but instead of least squares, we shrink the coefficients towards 0. This is called *ridge regression* and is an example of a *shrinkage model*.

3.1 Ridge Regression

Ridge regression is used both in the high dimensional case or when our function space is too large/complex, which leads to overfitting. In the overfitting case, we have seen that either decreasing our function space or getting more training data helps. Another popular way is to add a **regularizing term** to the error function in order to discourage the coefficients from reaching large values, effectively limiting the variance over $\mathcal{D}$.

Definition 3.2 (Ridge Regression)

In **ridge regression**, we minimize

$$L(\beta) = ||Y - X\beta||^2 + \lambda||\beta||^2 \tag{88}$$

where we penalize according to the L2 norm of the coefficients.

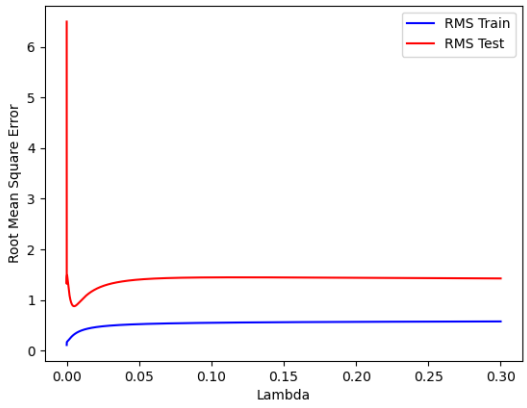


Figure 6: Even with a slight increase in the regularization term λ, the RMS error on the testing set heavily decreases.

Theorem 3.1 (Least Squares Solution for Ridge Regression)

The minimizer of the ridge loss is

$$\hat{\beta} = (X^T X + \lambda I)^{-1} X^T Y \tag{89}$$

Proof.

TBD

Theorem 3.2 (Bias Variance Decomposition of Ridge Regression)

TBD

From a computational point of view, we can see that by adding the λI term, it *dampens* the matrix so that it does become invertible (or well conditioned), allowing us to find a solution. The higher the λ term, the higher the damping effect. The next theorem compares the performance of the best ridge regression estimator to the best linear predictor.

Theorem 3.3 (Hsu, Kakade, Zhang, 2014 [2])

Suppose that $||X_i|| \leq r$ and let $\beta^T x$ be the best linear approximation to $m(x)$. Then, with probability at least $1 - 4e^{-t}$, we have

$$r(\hat{\beta}) - r(\beta) \leq \left(1 + O\left(\frac{1 + r^2/\lambda}{n}\right)\right) \frac{\lambda ||\beta||^2}{2} + \frac{\sigma^2}{n} \frac{\text{Tr}(\Sigma)}{2\lambda} \tag{90}$$

We can see that the λ term exists in the numerator on $\frac{\lambda||\beta||^2}{2}$ and in the denominator on $\frac{\text{Tr}(\Sigma)}{2\lambda}$. This is the bias variance tradeoff. The first term is the bias term, which is the penalty for not being able to fit the data as well. The second term is the variance term, which is the penalty for having a more complex model. So our optimal λ in the theoretical sense would be the one that minimizes the sum of these two terms. In practice, it's not this clean since we have unknown quantities in the formula, but just like how we did cross

validation over the model complexity, we can also do cross validation over the λ. The decomposition above just gives you a theoretical feeling of how these things trade off.

Code 3.1 (MWS of Ridge Regression in scikit-learn)

```python
import numpy as np
from sklearn.linear_model import Ridge

X = np.random.randn(10, 5)
y = np.random.randn(10)
# regularization parameter
model = Ridge(alpha=1.0)
model.fit(X, y)
print(model.score(X, y))
print(model.intercept_)
print(model.coef_)
print(model.predict(np.array([[1, 2, 3, 4, 5]])))
```

```
0.8605535024325397
-0.28291076492665157
[-0.10400521 -0.7587073
    -0.05116735  1.16236649
    -0.0401323 ]
[2.39097184]
.
.
.
.
.
.
```

Question 3.1 (To Do)

Bayesian interpretation of ridge regression.

3.2 Forward Stepwise Regression

Forward stepwise regression is a greedy algorithm that starts with an empty set of covariates and adds the covariate that most improves the fit. It avoids the NP-hardness of the best subset regression by adding covariates one by one.

Definition 3.3 (Greedy Forward Stepwise Regression)

Given your data $\mathcal{D}$, let's first standardize it to have mean 0 and variance 1.[a] You start off with a set $\mathcal{Q} = \{\}$ and choose the number of parameters K.

1. With each covariate $X = (X_1, \dots, X_n)$, we compute the correlation between it and the Y, which reduces to the inner product (since we standardized).

$$\rho_j = \langle Y, X_{:,j} \rangle = \frac{1}{n} \sum_{i=1}^{n} Y_i X_{ji} \tag{91}$$

2. Then, we take the covariate index that has the highest empirical correlation with Y, add it to $\mathcal{Q}$ and regress Y only on this covariate.

$$q_1 = \underset{j}{\text{argmax}}\, \rho_j, \quad \mathcal{Q} = \{q_1\}, \quad \hat{\beta}_{q_1} = \underset{\beta}{\text{argmin}}\, \frac{1}{n} ||Y - X_{:,q_1}\beta||^2 \tag{92}$$

3. Then you repeat the process. You take the residual values $r = Y - X_{:,q_1}\hat{\beta}_{q_1} \in \mathbb{R}^n$ compute the correlation between r and the remaining covariates, and pick our the maximum covariate index q_2. Then, you *repeat the regression from start* with these two covariates

$$q_2 = \underset{j}{\text{argmax}} \langle r, X_{:;j} \rangle, \quad \mathcal{Q} = \{q_1, q_2\}, \quad \hat{\beta}_{q_1,q_2} = \underset{\beta}{\text{argmin}}\, \frac{1}{n} ||Y - X_{:,[q_1,q_2]}\beta||^2 \tag{93}$$

Note that you're not going to get the same coefficient for $\hat{\beta}_{q_1}$ as before since you're doing two variable regression.

4. You get out the residual values $r = Y - X_{:,[q_1,q_2]}\hat{\beta}_{q_1,q_2} \in \mathbb{R}^n$ and keep repeating this process until you have K covariates in $\mathcal{Q}$.

[a] This may or may not be a good idea, since the variance of each covariate can tell you a lot about the importance of the covariate.

Again, there is a bias variance tradeoff in choosing the number of covariates K, but through cross-validation, we can find the optimal K. It is also easy to add constraints, e.g. if we wanted to place a restriction that two adjacent covariates can't be chosen, we can easily add this to the algorithm.

Theorem 3.4 (Rate of Convergence for Stepwise Regression)

Let $\hat{f}_K$ be the optimal regressor we get from K covariates in stepwise regression. Then, we have something like

$$||f - \hat{f}||^2 \leq c||f - f_K||^2 + \frac{\log n}{\sqrt{n}} \tag{94}$$

3.2.1 Stagewise Regression

Stagewise regression is a variant of forward stepwise regression where we add the covariate that most improves the fit, but we only take a small step in that direction. This is useful when we have a lot of covariates and we don't want to overfit.

3.3 Lasso Regression

The Lasso approximates the best subset regression by using a convex relaxation. In particular, the norm $||\beta||_0$ is not convex, but the L1 norm $||\beta||_1$ is. Therefore, we want relax our constraint equation as such:

$$\underset{||\beta||_0 \leq L}{\operatorname{argmin}} r(\beta) \mapsto \underset{||\beta||_1 \leq L}{\operatorname{argmin}} r(\beta) \tag{95}$$

This gives us a convex problem, which we can then solve. In fact, it turns out that optimizing the risk given the L1 restriction on the norm is equivalent to minimizing the risk plus a L1 penalty, as this is the Lagrangian form of the original equation (this is in convex optimization). Therefore, there exists a pair (L, λ) for which the two problems are equivalent

$$\underset{||\beta||_1 \leq L}{\operatorname{argmin}} r(\beta) = \underset{\beta}{\operatorname{argmin}} r(\beta) + \lambda||\beta||_1 \tag{96}$$

Definition 3.4 (LASSO Regression)

In **lasso regression**, we minimize the loss defined

$$\hat{R}(\beta) = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \beta^T x^{(i)})^2 + \lambda||\beta||_1 \tag{97}$$

where we penalize according to the L1 norm of the coefficients.

A question arises: Why use the L1 norm? The motivation behind this is that we want to model the L0 norm as much as possible but at the same time we want it to be convex. This turns out to be precisely the L1 norm. Unfortunately, there is no closed form solution for this estimator, but in convex optimization, we can prove that this estimator is sparse. That is, for large enough λ, many of the components of $\hat{\beta}$ are 0. The classical intuition for this is the figure below, where the equipotential lines have "corners." In fact for any $0 < p < 1$, there are also corners, but the problem with using these p-norms is that they are not convex.

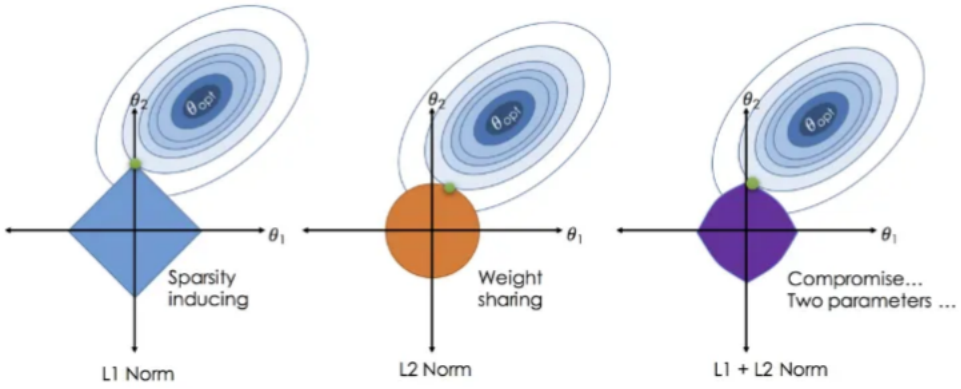


Figure 7: The ridge regularizer draws equipotential circles in our parameter space. The lasso draws a diamond, which tends to give a sparser solution since the loss is most likely to "touch" the corners of the contour plots of the regularizer. The elastic net is a linear combination of the ridge and lasso regularizers.

To motivate this even further, let us take the two vectors

$$a = \left(\frac{1}{\sqrt{d}}, \dots, \frac{1}{\sqrt{d}} \right) \qquad b = (1, 0, \dots, 0) \tag{98}$$

Then the L0, L1, and L2 norms of a are $d, \sqrt{d}, 1$ and those of b are $1, 1, 1$. We want to choose a norm that capture the sparsity of b and distinguishes it from b., The L0 norm clearly does this, but the L2 norm does not. The L1 norm is a good compromise between the two.

This now raises the question of how to determine a suitable regularization parameter λ. The next theorem shows a nice concentration property of the Lasso for bounded covariates.

Theorem 3.5 (Concentration of Lasso)

Given (X, Y), assume that $|Y| \leq B$ and $\max_j |X_j| \leq B$. Let

$$\beta^* = \underset{||\beta||_1 \leq L}{\operatorname{argmin}} \, r(\beta) \tag{99}$$

be the best sparse linear predictor in the L1 sense, where $r(\beta) = \mathbb{E}[(Y - \beta^T X)^2]$. Let our lasso estimator be

$$\hat{\beta} = \underset{||\beta||_1 \leq L}{\operatorname{argmin}} \, \hat{r}(\beta) = \underset{||\beta||_1 \leq L}{\operatorname{argmin}} \, \frac{1}{n} \sum_{i=1}^{n} (Y_i - \beta^T X_i)^2 \tag{100}$$

which minimizes the empirical risk. Then, with probability at least $1 - \delta$,

$$r(\hat{\beta}) \leq r(\beta^*) + \sqrt{\frac{16(L+1)^4 B^2}{n} \log \left(\frac{\sqrt{2}d}{\sqrt{\delta}} \right)} \tag{101}$$

Proof.

Code 3.2 (MWS of Lasso Regression in scikit-learn)

```
from sklearn.linear_model import Lasso

X = np.random.randn(10, 5)
y = np.random.randn(10)
# regularization parameter
model = Lasso(alpha=1e-1)
model.fit(X, y)
print(model.score(X, y))
print(model.intercept_)
print(model.coef_)
print(model.predict(np.array([[1, 2, 3, 4, 5]])))
```

```
0.47590269719236045
-0.8861298412689853
[0.         0.10767647
    0.24172197 0.7427863  0.
         ]
[3.02553422]
.
.
.
.
.
```

3.3.1 Soft Thresholding and Proximal Gradient Descent

3.4 Bayesian Regularization with Priors

We will now demonstrate how having a normal $\alpha\mathbf{I}$ prior around the origin in a Bayesian setting is equivalent to having a ridge penalty of $\lambda = \sigma^2/\alpha^2$ in a frequentist setting. If we have a Gaussian prior of form

$$p(\mathbf{w} \mid \alpha^2) = N(\mathbf{w} \mid \mathbf{0}, \alpha^2\mathbf{I}) = \left(\frac{1}{2\pi\alpha^2}\right)^{M/2} \exp\left(-\frac{1}{2\alpha^2}||\mathbf{w}||_2^2\right)$$

We can use Bayes rule to compute

$$\begin{aligned} p(\mathbf{w} \mid \mathbf{X}, \mathbf{Y}, \alpha^2, \sigma^2) &\propto p(\mathbf{Y} \mid \mathbf{w}, \mathbf{X}, \alpha^2, \sigma^2)\, p(\mathbf{w} \mid \mathbf{X}, \alpha^2, \sigma^2) \\ &= \left[\prod_{n=1}^{N} p(y^{(n)} \mid \mathbf{w}, \mathbf{x}^{(n)}, \alpha^2, \sigma^2)\right] p(\mathbf{w} \mid \mathbf{X}, \alpha^2, \sigma^2) \\ &= \left[\prod_{n=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y^{(n)} - h_{\mathbf{w}}(x^{(n)}))^2}{2\sigma^2}\right)\right] \cdot \left(\frac{1}{2\pi\alpha^2}\right)^{M/2} \exp\left(-\frac{1}{2\alpha^2}||\mathbf{w}||_2^2\right) \end{aligned}$$

and taking the negative logarithm gives us

$$\ell(\mathbf{w}) = \frac{1}{2\sigma^2}\sum_{n=1}^{N}\left(y^{(n)} - h_{\mathbf{w}}(\mathbf{x}^{(n)})\right)^2 + \frac{N}{2}\ln\sigma^2 + \frac{N}{2}\ln(2\pi) - \frac{M}{2}\ln(2\pi\alpha^2) + \frac{1}{2\alpha^2}||\mathbf{w}||_2^2$$

taking out the constant terms relative to $\mathbf{w}$ and multiplying by $2\sigma^2$ (which doesn't affect optima) gives us the ridge penalized error with a penalty term of $\lambda = \sigma^2/\alpha^2$.

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\left(y^{(n)} - h_{\mathbf{w}}(\mathbf{x}^{(n)})\right)^2 + \frac{\sigma^2}{\alpha^2}||\mathbf{w}||_2^2$$

But minimizing this still gives a point estimate of $\mathbf{w}$, which is not the full Bayesian treatment. In a Bayesian setting, we are given the training data $(\mathbf{X}, \mathbf{Y})$ along with a new test point $\mathbf{x}'$ and want to evaluate the predictive distribution $p(y \mid \mathbf{x}', \mathbf{X}, \mathbf{Y})$. We can do this by integrating over $\mathbf{w}$.

$$\begin{aligned} p(y \mid \mathbf{x}', \mathbf{X}, \mathbf{Y}) &= \int p(y \mid \mathbf{x}', \mathbf{w}, \mathbf{X}, \mathbf{Y})\, p(\mathbf{w} \mid \mathbf{x}', \mathbf{X}, \mathbf{Y})\, d\mathbf{w} \\ &= \int p(y \mid \mathbf{x}', \mathbf{w})\, p(\mathbf{w} \mid \mathbf{X}, \mathbf{Y})\, d\mathbf{w} \end{aligned}$$

where we have omitted the irrelevant variables, along with α^2 and σ^2 to simplify notation. By substituting the posterior $p(\mathbf{w} \mid \mathbf{X}, \mathbf{Y})$ with a normalized version of our calculation above and by noting that

$$p(y \mid \mathbf{x}', \mathbf{w}) = N(y \mid h_{\mathbf{w}}(\mathbf{x}'), \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{\big(y - h_{\mathbf{w}}(\mathbf{x}')\big)^2}{2\sigma^2} \right)$$

Now this integral may or may not have a closed form, but if we consider the polynomial regression with the hypothesis function of form

$$h_{\mathbf{w}}(x) = w_0 + w_1 x + w_2 x^2 + \ldots + w_{M-1} x^{M-1}$$

then this integral turns out to have a closed form solution given by

$$p(y \mid \mathbf{x}', \mathbf{X}, \mathbf{Y}) = N\big(y \mid m(x'), s^2(x')\big)$$

where

$$\begin{aligned} m(x') &= \frac{1}{\sigma^2} \boldsymbol{\phi}(x')^T \mathbf{S} \left(\sum_{n=1}^{N} \boldsymbol{\phi}(x^{(n)}) y^{(n)} \right) \\ s^2(x') &= \sigma^2 + \boldsymbol{\phi}(x')^T \mathbf{S} \boldsymbol{\phi}(x') \\ \mathbf{S}^{-1} &= \alpha^{-2} \mathbf{I} + \frac{1}{\sigma^2} \sum_{n=1}^{N} \boldsymbol{\phi}(x^{(n)}) \boldsymbol{\phi}(x')^T \end{aligned}$$

and $\boldsymbol{\phi}(x)$ is the vector of functions $\phi_i(x) = x^i$ from $i = 0, \ldots, M-1$.

4 Nonparametric Regression

4.1 K Nearest Neighbors Regression

When we want to do nonparametric regression, i.e. when dealing with nonlinear functions, we can construct a function that uses local averaging of its nearby points.

Example 4.1 (Local Averaging)

Say that we want to fit some function through a series of datapoints in simple regression (one covariate). Then, what we can do is take some sliding window and our vale of the function at a point x is the average of all values in the window $[x - \delta, x + \delta]$.

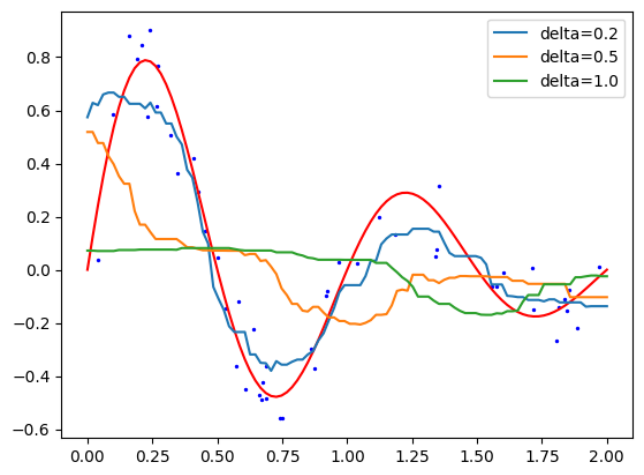


Figure 8: K means smoother

Code 4.1 (MWS of K Nearest Neighbor Regression in scikit-learn)

Local averaging is implemented as the K nearest neighbor regressor in scikit learn. It is slightly different in the way that it doesn't use the points within a certain δ away but rather the K nearest points. Either way, a minimal working example of this is

```
X = [[0], [1], [2], [3]]
y = [0, 0, 1, 1]
from sklearn.neighbors import KNeighborsRegressor
neigh = KNeighborsRegressor(n_neighbors=2)
neigh.fit(X, y)
print(neigh.predict([[1.5]]))
```

Note that since $\hat{f}$ is a combination of step functions, this makes it discontinuous at points.

4.2 Kernel Regression and Linear Smoothers

K nearest neighbor regression puts equal weights on both near and far points, as long as they are in the window. This may not be ideal, so a simple modification is to *weigh* these points according to their distance from the middle x. We can do this with a kernel, as the name suggests. Now this is not the same thing as a Mercer kernel in RKHS, so to distinguish that I will call it a *local averaging kernel*.

Definition 4.1 (Local Averaging Kernel)

A **kernel** is any smooth, symmetric, and non-negative function $K : \mathbb{R} \to \mathbb{R}$.

Definition 4.2 (Kernel Regression)

Given some datapoints, X, our fitted regressor is of form

$$\hat{f}(X) = \frac{\sum_i Y_i K\left(\frac{||X-X_i||}{h}\right)}{\sum_i K\left(\frac{||X-X_i||}{h}\right)} \tag{102}$$

where h is the **bandwidth** and the denominator is made sure so that the coefficients sum to 1. To get a clearer picture, we are really taking the weighted average of the Y_i's.

$$\hat{f}(X) = \sum_i Y_i \ell_i(X) \text{ where } \sum_i \ell_i(X) = 1 \tag{103}$$

Denoting $Y = (Y_1, \dots, Y_n) \in \mathbb{R}^n$ and the vector $f(X) = (f(X_1), \dots, f(X_n))$, if we can write the kernel function as $\hat{Y} = \hat{f}(X) = SY$, which in matrix form, is

$$\begin{bmatrix} \hat{Y}_1 \\ \vdots \\ \hat{Y}_n \end{bmatrix} = \begin{bmatrix} \hat{f}(X_1) \\ \vdots \\ \hat{f}(X_n) \end{bmatrix} = \begin{bmatrix} \ell_1(X_1) & \cdots & \ell_n(X_1) \\ \vdots & \ddots & \vdots \\ \ell_1(X_n) & \cdots & \ell_n(X_n) \end{bmatrix} \begin{bmatrix} Y_1 \\ \vdots \\ Y_n \end{bmatrix} \tag{104}$$

then we say that we have a **linear smoother**, with stochastic matrix S being our **smoothing matrix**.

The reason we'd like to have the weights to sum to 1 is that if we had data that was constant (i.e. all Y_i's are the same), then the fitted function should be constant at that value as well. Furthermore, the theme of linearity is important and will be recurring. The kernel estimator is defined for all X, but it's important to see its behavior at the training points X_i. The estimator $\hat{Y} = \hat{f}(X)$ is a linear combination of the Y_i's, and the coefficients $\ell_i(X_j)$ depend on the values of X_j. Therefore, we have $\hat{Y} = SY$, which is very similar to the equation $\hat{Y} = HY$ in linear regression, where H is the hat matrix that projects Y onto the column space of X. Nonparametric regression has the same form, but rather than being a projection, it is a linear smoothing matrix. Therefore, this theme unifies both linear regression and nonparametric regression. Linear smoothers, spline smoother, Gaussian processes, are all just different choices of the smoothing matrix S. However, not all nonparametric estimators are linear smoothers, as we will see later.

Here are some popular kernels.

Definition 4.3 (Gaussian Kernel)

The **Gaussian kernel** is defined as

$$K(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2} \tag{105}$$

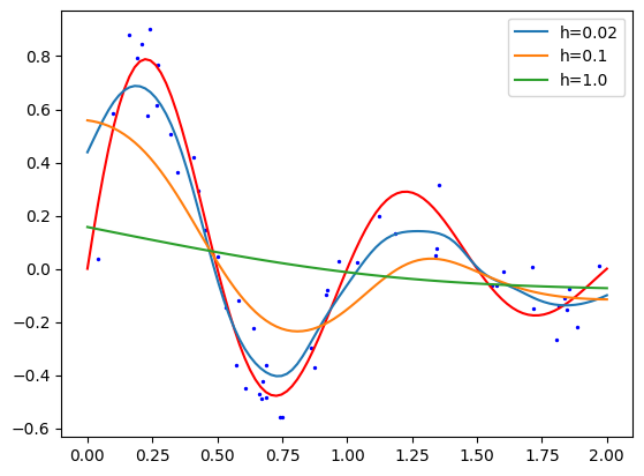


Figure 9: Gaussian kernel.

Definition 4.4 (Box-Car Kernel)

The **Box-Car kernel** is defined as

$$K(x) = \frac{1}{2}\mathbb{1}(|x| \leq 1) \tag{106}$$

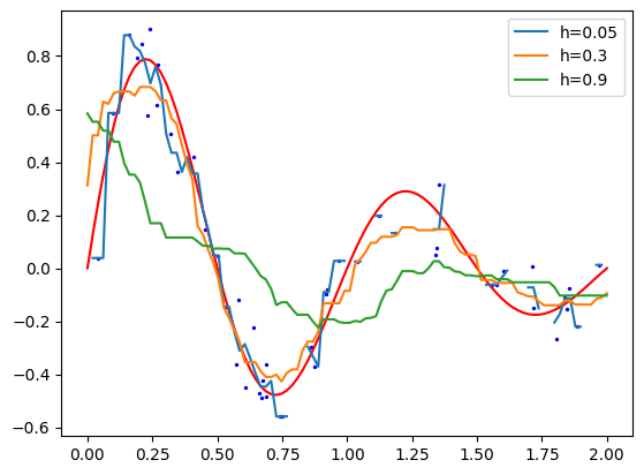


Figure 10: Boxcar kernel.

Definition 4.5 (Epanechnikov Kernel)

The **Epanechnikov kernel** is defined as

$$K(x) = \frac{3}{4}(1 - x^2)\mathbb{1}(|x| \leq 1) \tag{107}$$

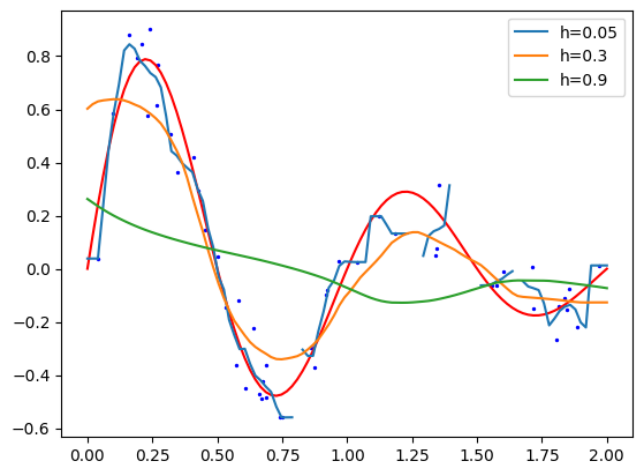


Figure 11: Epanechnikov kernel.

It turns out that from a theoretical point of view, the choice of the kernel doesn't really matter. What really matters is the bandwidth h since that is what determines the bias variance tradeoff. To see why, if $h = 0$, then it will simply interpolate the points and variance is extremely high, and if $h = \infty$, then the fitted function will be constant at $\bar{Y}$, leading to high bias. The following theorem formalizes this.

Theorem 4.1 (Bias Variance Tradeoff of Kernel Regression)

Suppose that $d = 1$ and that m'' is bounded. Also suppose that X has a nonzero, differentiable density p and that the support is unbounded. Then, the risk is

$$R_n = \frac{h_n^4}{4}\left(\int x^2 K(x)\right)^2 \int \left(m''(x) + 2m'(x)\frac{p'(x)}{p(x)}\right)^2 dx \tag{108}$$

$$+ \frac{\sigma^2 \int K^2(x)\,dx}{nh_n} \int \frac{dx}{p(x)} + o\left(\frac{1}{nh_n}\right) + o(h_n^4) \tag{109}$$

The first term is the squared bias and the second term is the variance.

Proof.

We first denote

$$\hat{f}(X) = \frac{\frac{1}{nh}\sum_{i=1}^n K\left(\frac{X - X_i}{h}\right) Y_i}{\frac{1}{nh}\sum_{i=1}^n K\left(\frac{X - X_i}{h}\right)} \tag{110}$$

where the denominator is the kernel density estimator $\hat{p}(X)$. Therefore, we rewrite

$$\hat{f}(x) - f(x) = \frac{\hat{a}(x)}{\hat{p}(x)} - f(x) \tag{111}$$

$$= \left(\frac{\hat{a}(x)}{\hat{p}(x)} - f(x)\right)\left(\frac{\hat{p}(x)}{p(x) + 1 - \frac{\hat{p}(x)}{p(x)}}\right) \tag{112}$$

$$= \frac{\hat{a}(x) - f(x)\hat{p}(x)}{p(x)} + \frac{(\hat{f}(x) - f(x))(p(x) - \hat{p}(x))}{p(x)} \tag{113}$$

as $n \to \infty$ both $\hat{f}(x) - f(x)$ and $p(x) - \hat{p}(x)$ going to 0, and since they're multiplied in the second

term, it will go to 0 very fast. So the dominant term is the first term, and we can write the above as approximately

$$\hat{f}(x) - f(x) \approx \frac{\hat{a}(x) - f(x)\hat{p}(x)}{p(x)} \tag{114}$$

TBD continued. Wasserman lecture 6, 10:00.

From the theorem above, we can see that if the bandwidth is small, then h^4 is small and the bias decreases. However, there is a h term in the denominator of the variance term, which also trades it off. We can furthermore see that the bias is sensitive to $p'/p(x)$. This means that if the density is steep, then the bias will be high. This is known as *design bias*, which is an underlying weakness in smoothing kernel regression. Another problem that is not contained in the theorem is the *boundary bias*, which states that if you're near the boundary of the distribution (i.e. near the boundary of its support), then the bias also explodes. This happens to be very nasty especially in high dimensions where most of the data tends to be near the boundary. It turns out that this can be easily fixed with local polynomial regression, which gets rid of this term in the bias without any cost to variance. This means that this is unnecessary bias.

Then you can apply regularization on this to get kernel ridge regression.

Code 4.2 (MWS of Kernel Ridge Regression in scikit learn)

```
from sklearn.kernel_ridge import KernelRidge
import numpy as np
n_samples, n_features = 10, 5
rng = np.random.RandomState(0)
y = rng.randn(n_samples)
X = rng.randn(n_samples, n_features)
krr = KernelRidge(alpha=1.0)
krr.fit(X, y)
```

4.3 Local Polynomial Regression

Now another way to think about the kernel estimator is as such. Suppose that you're doing linear regression on a bunch of points and you want to choose a c that minimizes the loss.

$$\sum_i (Y_i - c)^2 \tag{115}$$

You would just pick $c = \hat{Y}$. But if you are given some sort of locality condition, that the value of c should depend more on the values closer to it, you're really now minimizing

$$\sum_i (Y_i - c(x))^2 K\left(\frac{X_i - x}{h}\right) \tag{116}$$

Minimizing this by setting the derivative equal to 0 and solving gives us the kernel estimator. Therefore you're fitting some sort of local constant at a point X. But why fit a local constant, when you can fit a local line or polynomial? This is the idea behind local polynomial regression.

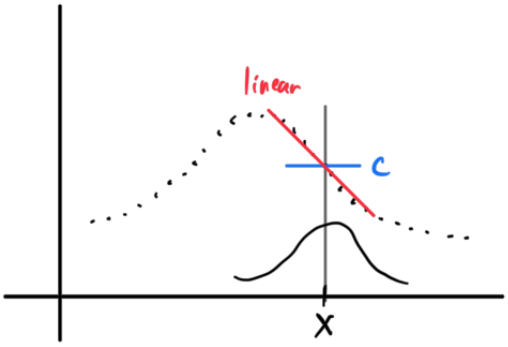


Figure 12: Rather than using a local constant, we can use a local linear estimator.

Therefore, we can minimize the modified loss.

Definition 4.6 (Local Polynomial Estimator)

The **local polynomial estimator** is a local linear kernel smoother that estimates the function $\hat{f}$ that minimizes the following loss.

$$\underset{\beta}{\text{argmin}} \sum_i K\left(\frac{X_i - x}{h}\right)\left(Y_i - (\beta_0(x) - \beta_1(x)(x - X_i) + \ldots + \beta_k(x)(x - X_i)^k)\right) \tag{117}$$

So we can fit a line

$$f(\mu) \approx \hat{\beta}_0(x) + \hat{\beta}_1(x)(\mu - x) \tag{118}$$

and simply remove the intercept term to get the local linear estimator.

$$\hat{f}(x) = \hat{\beta}_0(x) \tag{119}$$

Note that this is not the same as taking the constant estimate. We are extracting the fitted intercept term and so $\hat{\beta}_0(x) \neq c(x)$.

Theorem 4.2 (Weighted Least Squares)

The solution to the local linear estimator is similar to the weighted least squares solution.

$$\hat{\beta}(x) = \begin{pmatrix} \hat{\beta}_0(x) \\ \hat{\beta}_1(x) \end{pmatrix} = (X^T W X)^{-1} X^T W Y \tag{120}$$

where

$$X = \begin{pmatrix} 1 & X_1 - x \\ \vdots & \vdots \\ 1 & X_n - x \end{pmatrix} \qquad W = \begin{pmatrix} K\left(\frac{X_1 - x}{h}\right) & 0 & \cdots & 0 \\ 0 & K\left(\frac{X_2 - x}{h}\right) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & K\left(\frac{X_n - x}{h}\right) \end{pmatrix} \tag{121}$$

Computationally, it's similar to kernel regression and it gets rid of both the boundary and design bias.

4.4 Regularized: Spline Smoothing

This is not local, but it's a linear smoother.

4.5 Regularized: RKHS Regression

This is not local, but it's a linear smoother.

4.6 Additive Models

In the most general case, we want to create nonparametric regression functions of the form

$$Y = f(x_1, \dots, x_d) + \epsilon \tag{122}$$

We've done this for one dimensional case, but we can extend this to multiple dimensions through additive models of the form

$$Y = \sum_j f_j(x_j) + \epsilon \tag{123}$$

This gives us very interpretable models where we can clearly see the effect of each covariate on Y. Clearly, this is not as flexible as the previous model since they can't capture dependencies, but we can create sub-dependency functions and replace the form above to something like

$$Y = \sum_{i,j} f_{i,j}(x_i, x_j) + \epsilon \tag{124}$$

giving us more flexible models.

4.7 Nonlinear Smoothers, Trend Filtering

Tough example of the Dobbler function (like topologists sine curve). It's a pretty good fit but it's not too good since it's using a linear smoother (homogeneous). So we might need to fit it with nonlinear smoothers.

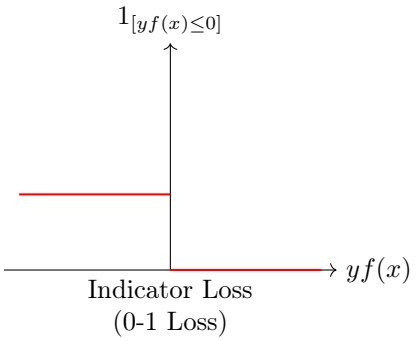
4.8 High Dimensional Nonparametric Regression

4.9 Regression Trees

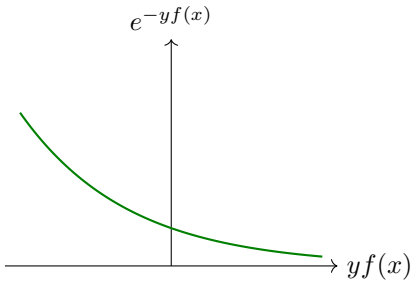
5 Linear Classification

5.1 Empirical Risk Minimizer

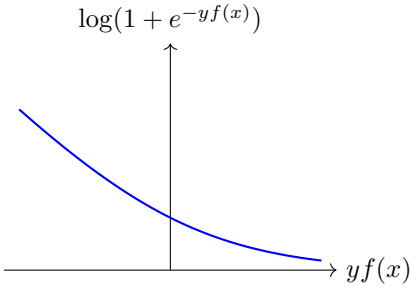
You literally just try to build a hyperplane to minimize the number of misclassifications, but this is not really differentiable and is hard. It's just a stepwise function. Therefore, you use a **surrogate loss function** to approximate the 0-1 loss function. The logistic uses some function, and the SVM uses the smallest convex function to approximate the 0-1 loss function. Here are some examples:



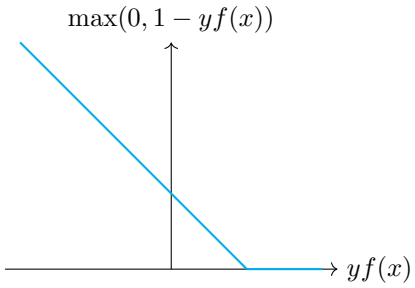
(a) Indicator/0-1 Loss can't be easily optimized.



(b) Exponential Loss for Adaboost.



(c) Log Loss for logistic regression.



(d) Hinge Loss for support vector machines.

Figure 13: Common loss functions used in classification

5.2 Perceptron

Definition 5.1 (Perceptron Model and Loss)

The simplest binary classification model is the **perceptron algorithm**. It is a discriminative parametric model that assigns

$$f_w(x) = \begin{cases} 1 & \text{if } w^T x + b \geq 0 \\ -1 & \text{if } w^T x + b < 0 \end{cases} \tag{125}$$

where we have chosen to label class $C_1 = 1$ and $C_2 = -1$. Note that unlike linear regression (and logistic regression, as we will see later), the perceptron is not a probabilistic model. It is a **discriminant function**, which just gives point estimates of the classes, not their respective probabilities. Like logistic regression, however, it is a linear model, meaning that the decision boundary it creates is always a linear (affine) hyperplane.

To construct the surrogate loss function, we would want a loss that penalizes not only if there is a misclassification, but how *far* that misclassified point is from the boundary. Therefore, if y and $\hat{y} = f_w(x)$ have the same sign, i.e. if $yf_w(x) > 0$, then the penalty should be 0, and if it is < 0, then the penalty should be proportional to the orthogonal distance of the misclassified point to the boundary, which is represented by

$-wTxy$ (where the negative sign makes this cost term positive).

Definition 5.2 (Surrogate Loss for Perceptron)

Therefore, our cost functions would take all the points and penalize all the terms by 0 if they are correctly classified and by $-\mathbf{w}^T\boldsymbol{\phi}^{(n)}y^{(n)}$ if incorrectly classified.

$$L(y,\hat{y}) = \sum_{n=1}[-\mathbf{w}^T\boldsymbol{\phi}^{(n)}y^{(n)}]_+ \text{ where } [f(\mathbf{x})]_+ := \begin{cases} f(\mathbf{x}) & \text{if } f(\mathbf{x}) > 0 \\ 0 & \text{else} \end{cases} \tag{126}$$

Note that this is a piecewise linear function and convex.

Code 5.1 (Perceptron in scikit-learn)

Let's implement this in scikit-learn, using two pipelines with different data standardization techniques to see the differences in the perceptron boundary.

```
from sklearn.pipeline import Pipeline
from sklearn.linear_model import Perceptron
from sklearn.preprocessing import QuantileTransformer, StandardScaler

pipe1 = Pipeline([
    ("scale", StandardScaler()),
    ("model", Perceptron())
])

pipe2 = Pipeline([
    ("scale", QuantileTransformer(n_quantiles=100)),
    ("model", Perceptron())
])
```

Figure 14

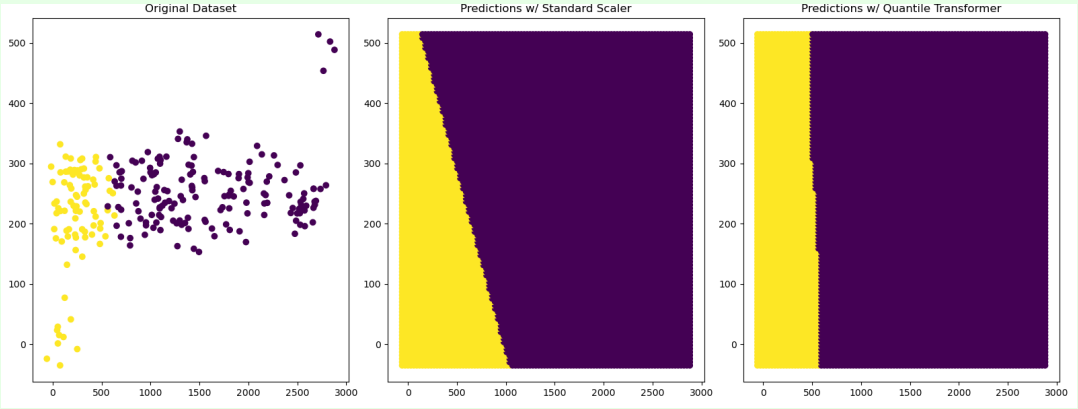


Figure 15: Perceptron Trained on Different Standardized Data

5.3 Logistic and Softmax Regression

We can upgrade from a discriminant function to a discriminative probabilistic model with **logistic regression**. In practice, we usually deal with **probabilistic models** where rather than giving a point estimate $\hat{y}$,

we attempt to model the *distribution* $\mathbb{P}_{Y|X=\hat{x}}$. Even though in the end, we will just output the mean μ of this conditional distribution, modeling the distribution allows us to quantify uncertainty in our measurements.

Definition 5.3 (Logistic Regression)

The **logistic regression** model is a linear model of the form

$$f_w(x) = \sigma(w^T x) = \frac{1}{1+e^{-w^T x}}, \text{ where } \sigma(x) \coloneqq \frac{1}{1+e^x} \tag{127}$$

It is different from linear regression in two ways:

1. In linear regression, we assumed that the targets are linearly dependent with the covariates as $y = w^T x + b$. However, this means that the hypothesis f_w is unbounded. Since we have two classes (say with labels 0 and 1), we must have some sort of *link function* σ that takes the real numbers and compresses it into the domain $[0, 1]$. Technically, we can choose any continuous, monotinically increasing function from $\mathbb{R}$ to $(0, 1)$. However, the following property of the sigmoid makes derivation of gradients very nice.

$$\sigma'(x) = \sigma(x)\big(1 - \sigma(x)\big) \tag{128}$$

2. Once this is compressed, we assume that the residual distribution is a Bernoulli.

Definition 5.4 (Binary Cross Entropy Loss as Surrogate Loss for Logistic Regression)

The surrogate loss for logistic regression is the **binary cross entropy loss**, which is defined as

$$L(y, \hat{y}) = -y \log \hat{y} - (1-y)\log(1-\hat{y}) \tag{129}$$

One important observation to make is that notice that the output of our hypothesis is used as a parameter to define our residual distribution.

1. In linear regression, the f_w was used as the *mean* μ of a Gaussian.
2. In logistic regression, the f_w is used also as the mean p of a Bernoulli.

The reason we want this sigmoid is so that we make the domains of the means of the residuals match the range of the outputs of our model. It's simply a manner of convenience, and in fact we could have really chose any function that maps $\mathbb{R}$ to $(0, 1)$.

Some questions may arise, such as "why isn't the variance parameter of the Gaussian considered in the linear model?" or "what about other residual distributions that have multiple parameters?" This is all answered by generalized linear models, which uses the output of a linear model as a *natural parameter* of the canonical exponential family of residual distributions.

Unfortunately, there is no closed form solution for logistic regression like the least squares solution in linear regression. Therefore, we can only resort to maximum likelihood estimation.

Theorem 5.1 (Maximum Likelihood Estimation for Logistic)

Given a dataset $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^N$, our likelihood is

$$L(\theta; \mathcal{D}) = \prod_{i=1}^n p(y^{(i)} \mid x^{(i)}; \theta) = \prod_{i=1}^N \big(h_\theta(x^{(i)})\big)^{y^{(i)}} \big(1 - h_\theta(x^{(i)})\big)^{1-y^{(i)}} \tag{130}$$

We can equivalently minimize its negative log likelihood, giving us the **binary cross entropy** loss

function

$$\ell(\theta) = -\log L(\theta) \tag{131}$$

$$= -\sum_{i=1}^{n} y^{(i)} \log h_\theta(x^{(i)}) + (1-y^{(i)}) \log(1-h_\theta(x^{(i)})) \tag{132}$$

Now taking the gradient for just a single sample $(x^{(i)}, y^{(i)})$ gives

$$\frac{\partial \ell}{\partial \theta} = \left(\frac{y^{(i)}}{\sigma(\theta^T x^{(i)})} - \frac{1-y^{(i)}}{1-\sigma(\theta^T x^{(i)})} \right) \frac{\partial}{\partial \theta} \sigma(\theta^T x^{(i)}) \tag{133}$$

$$= \frac{\sigma(\theta^T x^{(i)}) - y^{(i)}}{\sigma(\theta^T x^{(i)}) \left(1-\sigma(\theta^T x^{(i)})\right)} \sigma(\theta^T x^{(i)}) \left(1-\sigma(\theta^T x^{(i)})\right) x^{(i)} \tag{134}$$

$$= \left(h_\theta(x^{(i)}) - y^{(i)}\right) x \tag{135}$$

and summing it over some minibatch $M \subset \mathcal{D}$ gives

$$\nabla_\theta \ell_M = \sum_{(x,y) \in M} (y - h_\theta(x)) x \tag{136}$$

Therefore, the stochastic gradient descent algorithm is

$$\theta = \theta - \eta \nabla_\theta \ell(\theta) \tag{137}$$

$$= \theta - \eta \sum_{(x,y) \in M} (y - h_\theta(x)) x \tag{138}$$

We would like to extend this to the multiclass case.

Definition 5.5 (Softmax Function)

The softmax function is defined

$$o(\mathbf{x}) = \frac{e^{\mathbf{x}}}{||e^{\mathbf{x}}||} = \frac{1}{\sum_j e^{x_j}} \begin{pmatrix} e^{x_1} \\ \vdots \\ e^{x_D} \end{pmatrix} \tag{139}$$

What makes the softmax so popular is that the total derivative turns out to simplify functions a lot. The total derivative of the softmax can be derived as such.

Lemma 5.1 (Derivative of Softmax)

The derivative of the softmax is

$$Do(\mathbf{x}) = \text{diag}(o(\mathbf{x})) - o(\mathbf{x}) \otimes o(\mathbf{x}) \tag{140}$$

where $\otimes$ is the outer product. That is, let y_i be the output of the softmax. Then, for the 4×4 softmax function, we have

$$Do(\mathbf{x}) = \begin{pmatrix} y_1(1-y_1) & -y_1 y_2 & -y_1 y_3 & -y_1 y_4 \\ -y_2 y_1 & y_2(1-y_2) & -y_2 y_3 & -y_2 y_4 \\ -y_3 y_1 & y_3 y_3 & y_3(1-y_3) & -y_3 y_4 \\ -y_4 y_1 & -y_4 y_2 & -y_4 y_3 & y_4(1-y_4) \end{pmatrix} \tag{141}$$

Proof.

We will provide a way that allows us not to use quotient rule. Given that we are taking the partial derivative of y_i with respect to x_j, we can use the log of it to get

$$\frac{\partial}{\partial x_j}\log(y_i) = \frac{1}{y_i}\frac{\partial y_i}{\partial x_j} \implies \frac{\partial y_i}{\partial x_j} = y_i \frac{\partial}{\partial x_j}\log(y_i)$$

Now the partial of the log term is

$$\log y_i = \log\left(\frac{e^{x_i}}{\sum_l e^{x_l}} = x_i - \log\left(\sum_l e^{x_l}\right)\right. \tag{142}$$

$$\frac{\partial}{\partial x_j}\log(y_i) = \frac{\partial x_i}{\partial x_j} - \frac{\partial}{\partial x_j}\log\left(\sum_l e^{x_l}\right) \tag{143}$$

$$= 1_{i=j} - \frac{1}{\sum_l e^{x_l}} e^{x_j} \tag{144}$$

and plugging this back in gives

$$\frac{\partial y_i}{\partial x_j} = y_i(1_{i=j} - y_j) \tag{145}$$

It also turns out that the sigmoid is a specific case of the softmax. That is, given softmax for 2 classes, we have

$$o\begin{pmatrix}x_1\\x_2\end{pmatrix} = \frac{1}{e^{x_1}+e^{x_2}}\begin{pmatrix}e^{x_1}\\e^{x_2}\end{pmatrix} \tag{146}$$

So, the probability of being in class 1 is

$$\frac{e^{x_1}}{e^{x_1}+e^{x_2}} = \frac{1}{1+e^{x_2-x_1}} \tag{147}$$

and the logistic sigmoid is just a special case of the softmax function that avoids using redundant parameters. We actually end up overparameterizing the softmax because the probabilities must add up to one.

Definition 5.6 (Softmax Regression Model)

The softmax regression of K classes assumes a model of the form

$$h_\theta(x) = o(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{148}$$

where $\mathbf{W} \in \mathbb{R}^{K\times D}, \mathbf{b} \in \mathbb{R}^D$. Again, we have a linear map followed by some link function (the softmax) which allows us to nonlinearly map our unbounded linear outputs to some domain that can be easily parameterized by a probability distribution. In this case, our residual distribution is a **multinomial distribution**

$$y \sim \text{Multinomial}\big(h_{\mathbf{w}}(\mathbf{x})\big) = \text{Multinomial}\big([h_{\mathbf{w}}(\mathbf{x})]_1, \dots, [h_{\mathbf{w}}(\mathbf{x})]_K\big) \tag{149}$$

Definition 5.7 (Multiclass Cross Entropy Loss as Surrogate Loss for Softmax)

The surrogate loss for softmax regression is the **multiclass cross entropy loss**, which is defined as

$$L(\theta;\mathcal{D}) = -\sum_{i=1}^{N}\sum_{k=1}^{K} y_k^{(i)} \log\left(h_\theta(\mathbf{x}^{(i)})\right)_k \tag{150}$$

Theorem 5.2 (Maximum Likelihood Estimation for Softmax)

Since a closed form solution is not available for logistic regression, it is clearly not available for softmax. Therefore, we one hot encode our target variables as $\mathbf{y}^{(i)}$ and write our likelihood as

$$L(\theta; \mathcal{D}) = \prod_{i=1}^{N} \prod_{k=1}^{K} p(C_k \mid \mathbf{x}^{(i)})^{\mathbf{y}_k^{(i)}} = \prod_{i=1}^{N} \prod_{k=1}^{K} \left(\mathbf{h}_{\mathbf{W}}(\mathbf{x}^{(i)})\right)_k^{\mathbf{y}_k^{(i)}} \tag{151}$$

Taking the negative logarithm gives us the **cross entropy** loss function

$$\ell(\boldsymbol{\theta}) = -\sum_{i=1}^{N} \sum_{k=1}^{K} y_k^{(i)} \log\left(\mathbf{h}_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})\right)_k = -\sum_{i=1}^{N} \mathbf{y}^{(i)} \dot{\log}\left(\mathbf{h}_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})\right) \tag{152}$$

where $\cdot$ is the dot product. The gradient of this function may seem daunting, but it turns out to be very cute. Let us take a single sample $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$, drop the index i, and write

$$\begin{aligned} \mathbf{x} &\mapsto \mathbf{W}\mathbf{x} + \mathbf{b} = \mathbf{z} \\ \hat{\mathbf{y}} = \mathbf{a} &= o(\mathbf{z}) \\ L &= -\mathbf{y} \cdot \log(\mathbf{a}) = -\sum_{k=1}^{K} y_k \log(a_k) \end{aligned}$$

We must compute

$$\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial L}{\partial \mathbf{a}} \frac{\partial \mathbf{a}}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \boldsymbol{\theta}} \tag{153}$$

We can compute $\partial L / \partial \mathbf{z}$ as such, using our derivations for the softmax derivative above. We compute element wise.

$$\begin{aligned} \frac{\partial L}{\partial z_j} &= -\sum_{k=1}^{K} y_k \frac{\partial}{\partial z_j} \log(a_k) \\ &= -\sum_{k=1}^{K} y_k \frac{1}{a_k} \frac{\partial a_k}{\partial z_j} \\ &= -\sum_{k=1}^{K} \frac{y_k}{a_k} a_k (1_{\{k=j\}} - a_j) \\ &= -\sum_{k=1}^{K} y_k (1_{\{k=j\}} - a_j) \\ &= \left(\sum_{k=1}^{K} y_k a_j\right) - y_j \\ &= a_j \left(\sum_{k=1}^{K} y_k\right) - y_j \\ &= a_j - y_j \end{aligned}$$

and combining these gives

$$\frac{\partial L}{\partial \mathbf{z}} = (\mathbf{a} - \mathbf{y})^T \tag{154}$$

Now, computing $\partial \mathbf{z} / \partial \mathbf{W}$ gives us a 3-tensor, which is not ideal to work with. However, let us just

compute this with respect to the elements again. We have

$$z_k = \sum_{d=1}^{D} W_{kd} x_d + b_k$$
$$\frac{\partial z_k}{\partial W_{ij}} = \sum_{d=1}^{D} x_d \frac{\partial}{\partial W_{ij}} W_{kd}$$

where

$$\frac{\partial}{\partial W_{ij}} W_{kd} = \begin{cases} 1 & \text{if } i = k, j = d \\ 0 & \text{else} \end{cases} \tag{155}$$

Therefore, since d is iterating through all elements, the sum will only be nonzero if $k = i$. That is, $\frac{\partial z_k}{\partial W_{ij}} = x_j$ if $k = i$ and 0 if else. Therefore,

$$\frac{\partial \boldsymbol{z}}{\partial W_{ij}} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ x_j \\ 0 \\ \vdots \\ 0 \end{bmatrix} \leftarrow i\text{th element}$$

Now computing

$$\frac{\partial L}{\partial W_{ij}} = \frac{\partial L}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial W_{ij}} = (\mathbf{a} - \mathbf{y}) \frac{\partial \mathbf{z}}{\partial W_{ij}} = \sum_{k=1}^{K} (a_k - y_k) \frac{\partial z_k}{\partial W_{ij}} = (a_i - y_i) x_j \tag{156}$$

To get $\partial L / \partial W_{ij}$ we want a matrix whose entry (i, j) is $(a_i - y_i) x_j$. This is simply the outer product as shown below. For the bias term, $\partial \mathbf{z} / \partial \mathbf{b}$ is simply the identity matrix.

$$\frac{\partial L}{\partial \mathbf{W}} = (\mathbf{a} - \mathbf{y}) \mathbf{x}^T, \quad \frac{\partial L}{\partial \mathbf{b}} = \mathbf{a} - \mathbf{y} \tag{157}$$

Therefore, summing the gradient over some minibatch $M \subset [N]$ gives

$$\nabla_{\mathbf{W}} \ell_M = \sum_{i \in M} (\mathbf{h}_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}) - \mathbf{y}^{(i)})(\mathbf{x}^{(i)})^T, \quad \nabla_{\mathbf{b}} \ell_M = \sum_{i \in M} (\mathbf{h}_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}) - \mathbf{y}^{(i)}) \tag{158}$$

and our stochastic gradient descent algorithm is

$$\begin{aligned} \boldsymbol{\theta} = \begin{pmatrix} \mathbf{W} \\ \mathbf{b} \end{pmatrix} &= \begin{pmatrix} \mathbf{W} \\ \mathbf{b} \end{pmatrix} - \eta \begin{pmatrix} \nabla_{\mathbf{W}} \ell_M \\ \nabla_{\mathbf{b}} \ell_M \end{pmatrix} \\ &= \begin{pmatrix} \mathbf{W} \\ \mathbf{b} \end{pmatrix} - \eta \begin{pmatrix} \sum_{i \in M} (\mathbf{h}_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}) - \mathbf{y}^{(i)})(\mathbf{x}^{(i)})^T \\ \sum_{i \in M} (\mathbf{h}_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}) - \mathbf{y}^{(i)}) \end{pmatrix} \end{aligned}$$

5.3.1 Sparse Logistic Regression

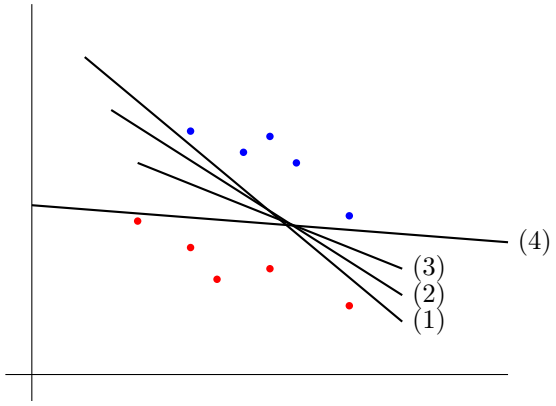
5.4 Support Vector Machines

Definition 5.8 (Hinge Loss)

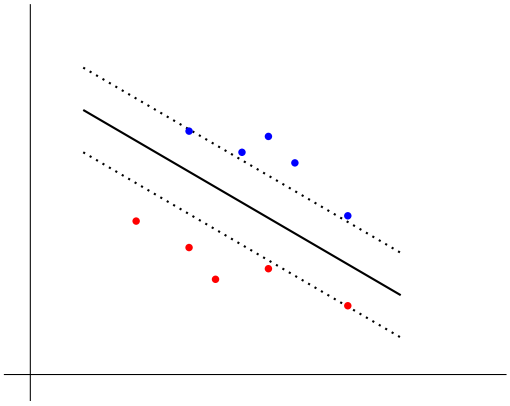
The **hinge loss** is a convex surrogate loss function for the 0-1 loss function. It is defined as

$$L(y, \hat{y}) = \max(0, 1 - y \cdot \hat{y}) \tag{159}$$

A support vector machine focuses only on the points that are most difficult to tell apart, whereas other classifiers pay attention all of the points. A SVM is a discriminative, non-probabilistic model. Let us first assume that our dataset $\mathcal{D} = \{\mathbf{x}_i, y_i\}$ is linearly separable with $y_i \in \{-1, +1\}$. Based on previous algorithms like the perceptron, it will find some separating hyperplane. However, there's an infinite number of separating hyperplanes as shown in Figure 16a. What support vector machines want to do is to find the best one, with the "best" defined as the hyperplane that maximizes the distance between either the closest positive or negative samples, shown in Figure 16*b*.



(a) Planes such as (1) and (4) are "too close" to the positive and negative samples.



(b) SVMs try to find the separating hyperplane with the best minimum margin.

Figure 16: Motivating problem

We want to formalize the concepts of these margins that we wish to maximize. To do this, we will define two terms.

Definition 5.9 (Geometric margin)

Given a point $\mathbf{x}_0$ and a hyperplane of equation $\mathbf{w} \cdot \mathbf{x} + b = 0$, the distance from $\mathbf{x}_0$ to the hyperplane, known as the **geometric margin**, can be computed with the formula

$$d = \frac{|\mathbf{x}_0 \cdot \mathbf{w} + b|}{||\mathbf{w}||} \tag{160}$$

Therefore, the geometric margin of the ith sample with respect to the hypothesis f is defined

$$\gamma_i = \frac{y_i \left(\mathbf{w} \cdot \mathbf{x}_i + b\right)}{||\mathbf{w}||} \tag{161}$$

We wish to optimize the parameters $\mathbf{w}, b$ in order to maximize the minimum of the geometric margins (the distance between the closest point and the hyperplane).

$$\underset{\mathbf{w},b}{\operatorname{argmax}} \min_i \gamma_i = \underset{\mathbf{w},b}{\operatorname{argmax}} \left\{ \frac{1}{||\mathbf{w}||} \min_i \left[y_i \left(\mathbf{w} \cdot \mathbf{x}_i + b \right) \right] \right\} \tag{162}$$

Direct solution of this optimization problem would be very complex, and so we convert this into an equivalent problem that is much easier to solve. Note that the solution to the above term is not unique. If there was a solution $(\mathbf{w}^*, b^*)$, then

$$\frac{y_i(\mathbf{w} \cdot \mathbf{x}_i + b)}{||\mathbf{w}||} = \frac{y_i(\lambda \mathbf{w} \cdot \mathbf{x}_i + \lambda b)}{||\lambda \mathbf{w}||} \tag{163}$$

That is, the geometric margin is not sensitive to scaling of the parameters of the hyperplane. Therefore, we can scale the numerator and the denominator by whatever we want and use this freedom to set

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) = 1$$

for the point that is closest to the surface. In that case, all data points will satisfy the constraints

$$y_n(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1$$

In the case of data points for which the equality holds, the constraints are said to be *active*, whereas for the remainder they are *inactive*. Therefore, it will always be the case that $\min_i \left[y_i \left(\mathbf{w} \cdot \mathbf{x}_i + b \right) \right] = 1$, and the constraint problem reduces to

$$\underset{\mathbf{w},b}{\operatorname{argmax}} \frac{1}{||\mathbf{w}||} = \underset{\mathbf{w},b}{\operatorname{argmin}} \frac{1}{2}||\mathbf{w}||^2 \text{ subject to constraints } y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1$$

This final step is the most significant step in this derivation and may be hard to wrap around the first time. So we dedicate the next subsubsection for this.

5.5 Functional and Geometric Margins

We could just work straight with this geometric margin, but for now, let's try to extend what we did with the perceptron into SVMs. We will find out that extending the concept of functional margins into SVMs leads to ill-defined problems. In the perceptron, we wanted to construct a function $f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b$ such that

$$y_i \, f(\mathbf{x}_i) \geq 0 \text{ for all } i = 1, 2, \ldots, N$$

Definition 5.10 (Functional Margin)

The value of $y_i \, f(\mathbf{x}_i)$ gives us our confidence on our classification, and in a way it represents a kind of distance away from the separating hyperplane (if this value was 0, then we would be 50 50 split on whether to label it positive or negative). Therefore, we shall define

$$\hat{\gamma}_i = y_i f(\mathbf{x}_i)$$

as the **functional margin** of $(\mathbf{w}, b)$ with respect to the training sample $(\mathbf{x}_i, y_i)$. Therefore, the smallest of the function margins can be written

$$\hat{\gamma} = \min_i \gamma_i$$

called the **function margin**.

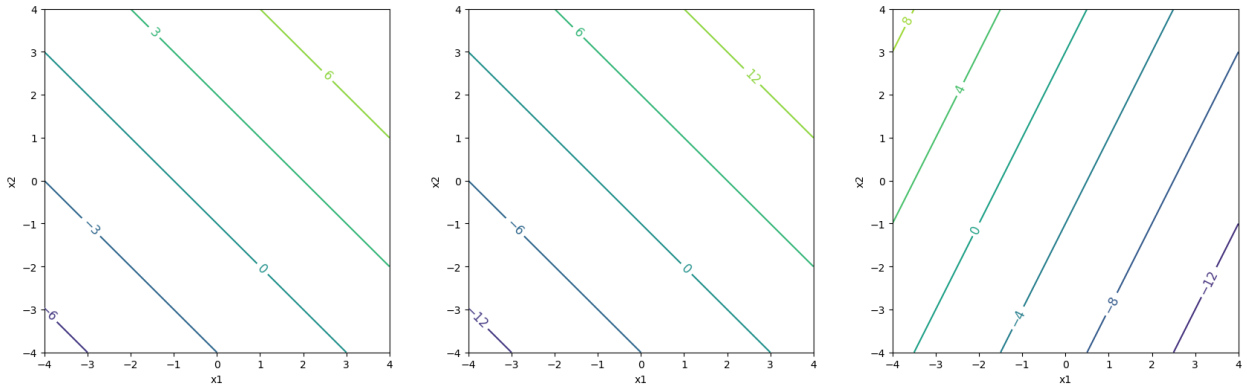
Note that the geometric margin and functional margin are related by a constant scaling factor. Given a sample $(\mathbf{x}_i, y_i)$, we have

$$\text{GeometricMargin} = \frac{y_i\left(\mathbf{w}\cdot\mathbf{x}_i + b\right)}{||\mathbf{w}||_2} = \frac{\text{FunctionalMargin}}{||\mathbf{w}||_2}$$

As we can see, the perceptron works with the functional margin, and since it does not care about how large the margin is (just whether it's positive or negative), we are left with an underdetermined system in which there exists infinite $(\mathbf{w}, b)$'s. Now what we want to do is impose a certain minimum margin $\gamma > 0$ and solve for $(\mathbf{w}, b)$ again, and keep increasing this γ until there is some unique solution. We can view this problem in two ways:

1. Take a specific minimum margin γ and find a $(\mathbf{w}, b)$, which may not exist, be unique, or exist infinitely that satisfies
$$y_i f(\mathbf{x}) = y_i(\mathbf{w}\cdot\mathbf{x} + b) \geq \gamma \text{ for all } i = 1, \dots, N$$
2. Take a specific $(\mathbf{w}, b)$ and calculate the maximum γ that satisfies the constraint equations above.

They're both equivalent problems, but both ill-posed if we look at (2). Since the samples are linearly separable by assumption, we can say that there exists some $\epsilon > 0$ such that $y_i f(\mathbf{x}_i) \geq \epsilon$ for all i. Therefore, if we just scale $(\mathbf{w}, b) \mapsto (\lambda\mathbf{w}, \lambda b)$ for some large λ, this leads to the solution for γ being unbounded. We can see in Figure 17 that we can increased confidence at no cost. Looking at (1), we can also see that if $(\mathbf{w}, b)$ does exist, then every other $(\lambda\mathbf{w}, \lambda b)$ for $\lambda > 1$ satisfies the property.



(a) $f(x) = x_1 + x_2 + 1$ (b) $f(x) = 2x_1 + 2x_2 + 2$ (c) $f(x) = -2x_1 + x_2 - 3$

Figure 17: From (a), you can see that simply multiplying everything by two automatically increases our confidence by 2, meaning that the functional margin can be scaled arbitrarily by scaing $(\mathbf{w}, b)$. There are still too many degrees of freedom in here and so extra constraints must be imposed.

5.5.1 Lagrange Duality

To minimize the equations with the constraint equations, we can use the method of Lagrange multipliers, which leads to to Lagrangian

$$\mathcal{L}(\mathbf{w}, b, \boldsymbol{\alpha}) = \frac{1}{2}||\mathbf{w}||^2 - \sum_i \alpha_i\big[y_i(\mathbf{w}\cdot\mathbf{x}_i + b) - 1\big]$$

We can take the gradients with respect to $\mathbf{w}$ and b and set them to 0, which gives the two conditions

$$\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i$$
$$0 = \sum_i \alpha_i y_i \mathbf{x}_i$$

Now let's substitute our evaluated $\mathbf{w}$ back into $\mathcal{L}$, which gives the **dual representation** of the maximum margin problem in which we maximize

$$\begin{aligned} L &= \frac{1}{2}\left(\sum_i \alpha_i y_i \mathbf{x}_i\right)\left(\sum_j \alpha_j y_j \mathbf{x}_j\right) - \sum_i \alpha_i y_i x_i \cdot \left[\sum_j \alpha_j y_j x_j\right] - \sum_i \alpha_i y_i b + \sum_i \alpha_i \\ &= \sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j \, \mathbf{x}_i \cdot \mathbf{x}_j \end{aligned}$$

The summation with the b in it is 0 since we can pull the b out and the remaining sum is 0 from before. Now the optimization only depends on the dot product $\mathbf{x}_i \cdot \mathbf{x}_j$ of all pairs of sample vectors, which is very interesting. We will see more of this when we talk about kernel methods. Now, we need to solve the dual problem

$$\max_{\boldsymbol{\alpha}} \mathcal{L}(\boldsymbol{\alpha})$$

which can be done using some generic quadratic programming solver or some other method to get the optimum $\boldsymbol{\alpha}^*$, which gives us

$$\mathbf{w}^* = \sum_i \alpha_i^* y_i \mathbf{x}_i$$

5.6 Nonseparable Case

5.7 Gaussian/Linear Discriminant Analysis

5.7.1 Discriminative vs. Generative Models

Now we introduce our first example of a generative model, which introduces another division between models (in addition to parametric vs nonparametric, frequentist vs bayesian). Generally, there are two ways to model $\mathbb{P}_{Y|X=x}$.

Definition 5.11 (Discriminative Models)

Discriminative models attempt to do this directly by modeling only the conditional probability distribution $\mathbb{P}_{Y|X=x}$. We don't care about the underlying distribution of X, but rather we just want to try and predict Y given X. Essentially, we are trying to approximate the conditional expectation $h(X) = \mathbb{E}[Y \mid X]$, which is the best we can do. Given $X = x$, we use our model of $\mathbb{P}_{Y|X=x}$, and our hypothesis function will predict the its mean.

$$h(x) = \mathbb{E}[Y \mid X = x] \tag{164}$$

Definition 5.12 (Generative Models)

Generative models approximate this conditional probability by taking a more general approach. They attempt to model the joint probability distribution $\mathbb{P}_{X\times Y}$ (also called **inference**), which essentially gives everything we need about the data. Doing this allows us to *generate* more data (hence the name), which may be useful.
One way to approximate the joint distribution is to model the conditional distribution $\mathbb{P}_{X|Y=y}$, which gives the distribution of each labels. Now combined with the probability measure $\mathbb{P}_Y$, we can get the joint distribution. Usually in classification, the $\mathbb{P}_Y$ is easy to approximate (the MLE is simply the fequencies of the labels), so conventionally, modeling $\mathbb{P}_{X\times Y}$ and modeling $\mathbb{P}_{X|Y=y}$ are considered the same thing. Once we have these, we can calculate the joint distribution, but in high-dimensional spaces this tends to be computationally hard. Therefore, we usually resort to simply calculating $\mathbb{P}_{X|Y=y}$ and then using Bayes rule to approximate

$$\mathbb{P}_{Y|X} = \frac{\mathbb{P}_{X|Y}\mathbb{P}_Y}{\mathbb{P}_X} \tag{165}$$

where the normalizing term is computed using Monte Carlo simulations.

This is the first example of a generative model. In GDA, we basically write the likelihood as

$$\prod_{i=1}^{n} p(x_i, y_i) = \prod_i p(x_i \mid y_i) p(y_i) \tag{166}$$

where each $p(x_i \mid y_i)$ is Gaussian and $p(y_i)$ is Bernoulli. This specifies $p(x_i, y_i)$ and therefore is called a generative model. In logistic regression, we have

$$\prod_{i=1} p(x_i, y_i) = \left(\prod_i p(y_i \mid x_i) \right) \left(\prod_i p(x_i) \right) \tag{167}$$

and the first term is the logistic function and the second term is unknown. We only use the first part to classify, and this is a discriminative model. You can be agnostic about the data generating process and you can work with less data since there are less things to fit. Some people ask why should you model more unless you have to, so people tend to try to model the minimum, which is why logistic regression is more popular.

5.7.2 Construction

GDA assumes that $\mathbb{P}(x \,|y)$ is distributed according to a multivariate Gaussian distribution. Let us assume that the input space is d-dimensional and this is a binary classification problem. We set

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

This method is usually applied using only one covariance matrix Σ. The distributions are

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

Now, what we have to do is optimize the distribution parameters $\pi \in (0,1)\mathbb{R}, \mu_0 \in \mathbb{R}^d, \mu_1 \in \mathbb{R}^d, \Sigma \in \text{Mat}(d \times d, \mathbb{R}) \simeq \mathbb{R}^{d \times d}$ so that we get the best-fit model. Assuming that each sample has been picked independently, this is equal to maximizing

$$L(\pi, \mu_0, \mu_1, \Sigma) = \prod_{i=1}^{n} \mathbb{P}\big(x^{(i)}, y^{(i)} \,;\, \pi, \mu_0, \mu_1, \Sigma\big) \tag{168}$$

which is really just the probability that we get precisely all these training samples $(x^{(i)}, y^{(i)})$ given the 4 parameters. This can be done by optimizing its log-likelihood, which is given by

$$\begin{aligned} l(\pi, \mu_0, \mu_1, \Sigma) &= \log \prod_{i=1}^{n} \mathbb{P}(x^{(i)}, y^{(i)}; \pi, \mu_0, \mu_1, \Sigma) \\ &= \log \prod_{i=1}^{n} \mathbb{P}(x^{(i)} \,|\, y^{(i)}; \mu_0, \mu_1, \Sigma)\, \mathbb{P}(y^{(i)}; \pi) \\ &= \sum_{i=1}^{n} \log \left(\mathbb{P}(x^{(i)} \,|\, y^{(i)}; \mu_0, \mu_1, \Sigma)\, \mathbb{P}(y^{(i)}; \pi) \right) \end{aligned}$$

and therefore gives the maximum likelihood estimate to be

$$\begin{aligned}
\pi &= \frac{1}{N}\sum_{n=1}^{N} 1\{y^{(n)} = 1\} \\
\boldsymbol{\mu}_0 &= \frac{\sum_{n=1}^{n} 1_{\{y^{(n)}=0\}}\mathbf{x}^{(n)}}{\sum_{n=1}^{N} 1_{\{y^{(n)}=0\}}} \\
\boldsymbol{\mu}_1 &= \frac{\sum_{n=1}^{n} 1_{\{y^{(n)}=1\}}\mathbf{x}^{(n)}}{\sum_{n=1}^{N} 1_{\{y^{(n)}=1\}}} \\
\boldsymbol{\Sigma} &= \frac{1}{N}\sum_{n=1}^{N}(\mathbf{x}^{(n)} - \mu_{y^{(n)}})(\mathbf{x}^{(n)} - \mu_{Y^{(i)}})^T
\end{aligned}$$

A visual of the algorithm is below, with contours of the two Gaussian distributions, along with the straight line giving the decision boundary at which $\mathbb{P}(y = 1 \,|\, x) = 0.5$.

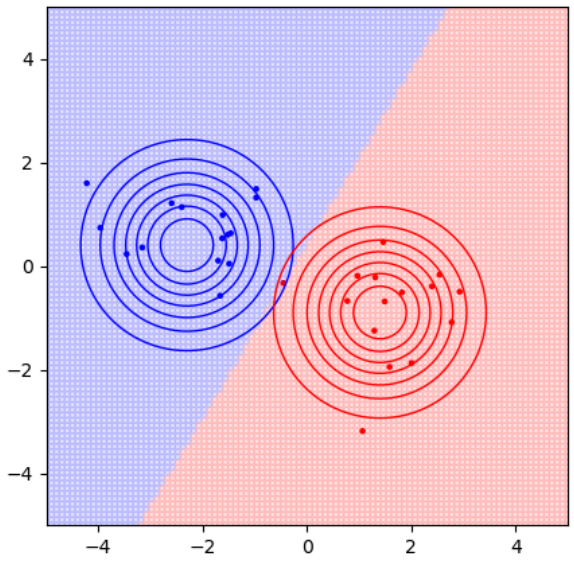


Figure 18: GDA of Data Generated from 2 Gaussisans centered at $(-2.3, 0.4)$ and $(1.4, -0.9)$ with unit covariance. The decision boundary is slightly off since MLE approximates the true means.

5.8 Fisher Linear Discriminant

6 Nonparametric Classification

6.1 K Nearest Neighbors

Question 6.1 (To Do)

Maybe similar like a kernel regression?

Given a bunch of points in a metric space $(\mathcal{X}, d)$ that have classification labels, we want to label new datapoints $\hat{\mathbf{x}}$ based on the labels of other points that already exist in our dataset. One way to look at it is to look for close points within the dataset and use their labels to predict the new ones.

Definition 6.1 (Closest Neighborhood)

Given a dataset $\mathcal{D} = \{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\}$ and a point $\hat{\mathbf{x}} \in (\mathcal{X}, d)$, let the **k closest neighborhood** of $\hat{\mathbf{x}}$ be $N_k(\hat{\mathbf{x}}) \subset [N]$ defined as the indices i of the k points in $\mathcal{D}$ that is closest to $\hat{\mathbf{x}}$ with respect to the distance metric $d_{\mathcal{X}}$.

Definition 6.2 (K Nearest Neighbors)

The **K Nearest Neighbors (KNN)** is a discriminative nonparametric supervised learning algorithm that doesn't have a training phase. Given a new point $\hat{\mathbf{x}}$, we look at all points in its k closest neighborhood, and $h(\hat{\mathbf{x}})$ will be equal to whatever the majority class will be in. Let us one-hot encode the labels $\mathbf{y}^{(i)}$ into $\mathbf{e}_i$'s, and the number of data point in the ith class can be stored in the variable

$$a_i = \sum_{i \in N_k(\hat{\mathbf{x}})} 1_{\{\mathbf{y}^{(i)} = \mathbf{e}_i\}} \tag{169}$$

which results in the vector storing the counts of labels in the k closest neighborhood

$$\mathbf{a} = (a_1, a_2, \dots, a_K) = \left(\sum_{i \in N_k(\hat{\mathbf{x}})} 1_{\{\mathbf{y}^{(i)} = \mathbf{e}_1\}}, \sum_{i \in N_k(\hat{\mathbf{x}})} 1_{\{\mathbf{y}^{(i)} = \mathbf{e}_2\}}, \dots, \sum_{i \in N_k(\hat{\mathbf{x}})} 1_{\{\mathbf{y}^{(i)} = \mathbf{e}_K\}} \right) \tag{170}$$

and take the class with the maximum element as our predicted label.

The best choice of K depends on the data:

1. Larger values of K reduces the effect of noise on the classification, but make boundaries between classes less distinct. The number of misclassified data points (error) increases.
2. Smaller values are more sensitive to noise, but boundaries are more distinct and the number of misclassified data points (error) decreases.

Too large of a K value may increase the error too much and lead to less distinction in classification, while too small of a k value may result in us overclassifying the data. Finally, in binary (two class) classification problems, it is helpful to choose K to be odd to avoid tied votes.

This is an extremely simple algorithm that may not be robust. For example, consider $K \geq 3$, and we are trying to label a point $\hat{\mathbf{x}}$ that happens to be exactly where one point is on our dataset $\mathbf{x}^{(i)}$. Then, we should do $h(\hat{\mathbf{x}}) = y^{(i)}$, but this may not be the case if there are no other points with class $y^{(i)}$ in the k closest neighborhood of $\mathbf{x}^{(i)}$. Therefore, we want to take into account the distance of our new points from the others.

Definition 6.3 (Weighted Nearest Neighbor Classifier)

Let us define a monotinically decreasing function $\omega : \mathbb{R}_0^+ \mapsto \mathbb{R}_0^+$. Given a point $i \in N_k(\hat{\mathbf{x}})$, we can construct the weight of our matching label as inversely proportional to the distance: $\omega_i[d(\hat{\mathbf{x}}, \mathbf{x}^{(i)})]$ and store them as

$$\mathbf{a} = (a_1, a_2, \dots, a_{\mathcal{K}}) = \left(\sum_{i \in N_k(\hat{\mathbf{x}})} \omega_i 1_{\{\mathbf{y}^{(i)} = \mathbf{e}_1\}}, \sum_{i \in N_k(\hat{\mathbf{x}})} \omega_i 1_{\{\mathbf{y}^{(i)} = \mathbf{e}_2\}}, \dots, \sum_{i \in N_k(\hat{\mathbf{x}})} \omega_i 1_{\{\mathbf{y}^{(i)} = \mathbf{e}_{\mathcal{K}}\}} \right) \tag{171}$$

and again take the class with the maximum element.

One caveat of KNN is in high dimensional spaces, as its performance degrades quite badly due to the curse of dimensionality.

Example 6.1 (Curse of Dimensionality in KNN)

Consider a dataset of N samples uniformly distributed in a d-dimensional hypercube. Now given a point $x \in [0,1]^d$, we want to derive the expected radius r_k required to encompass its k nearest neighbors. Let us define this ball to be $B_{r_k} \coloneqq \{z \in \mathbb{R}^d \mid ||z - x||_2 \leq r_k\}$. Since thse N points are uniformly distributed, the expected number of points contained in $B_{r_k}(x)$ is simply the proportion of the volume that $B_{r_k}(x)$ encapsulates in the box, multiplied by N. Therefore, for some fixed x and r, let us denote $Y(x, y)$ as the random variable representing the number of points contained within $B_r(x)$. By linearity of expectation and summing over the expectation for whether each point will be in the ball, we have

$$\mathbb{E}[Y(x,r)] = N \cdot \frac{\mu(B_r(x) \cap [0,1]^d)}{\mu([0,1]^d)}$$

where μ is the Lebesgue measure of $\mathbb{R}^d$. Let us assume for not that we don't need to worry about cases where the ball is not fully contained within the cube, so we can just assume that Y is only dependent on r: $Y(r)$. Also, since the volume of the hypercube is 1, $\mu([0,1]^d) = 1$ and we get

$$\mathbb{E}[Y(r)] = N \cdot C_d \cdot r^d$$

which we set equal to k and evaluate for r. C_d is a constant such that the volume of the hypersphere of radius r can be derived as $V = C_d \cdot r^d$. We therefore get

$$N \cdot C_d \cdot r_k^d = k \implies r_k = \left(\frac{k}{NC_d} \right)^{1/d}$$

It turns out that C_d decreases exponentially, so the radius r_k explodes as d grows. Another way of looking at this is that in high dimensions, the ℓ_2 distance between all the pairwise points are close in every single dimension, so it becomes harder to distinguish points that are close vs those that are far.

6.1.1 Approximate K Nearest Neighbors

6.2 Classification Trees

Definition 6.4 (Decision Trees)

Like K nearest neighbors, **decision trees** are discriminative nonparametric classification algorithms that involves creating some sort of tree that represents a set of decisions using a given set of input data $\mathbf{x}^{(i)}$ with its given classification $\mathbf{y}^{(i)}$. When predicting the class of a new input $\hat{\mathbf{x}}$, we would look at its attributes in some order, e.g. $\hat{x_1}, \hat{x}_2, \hat{x}_3$, and make a decision on which class it is in.

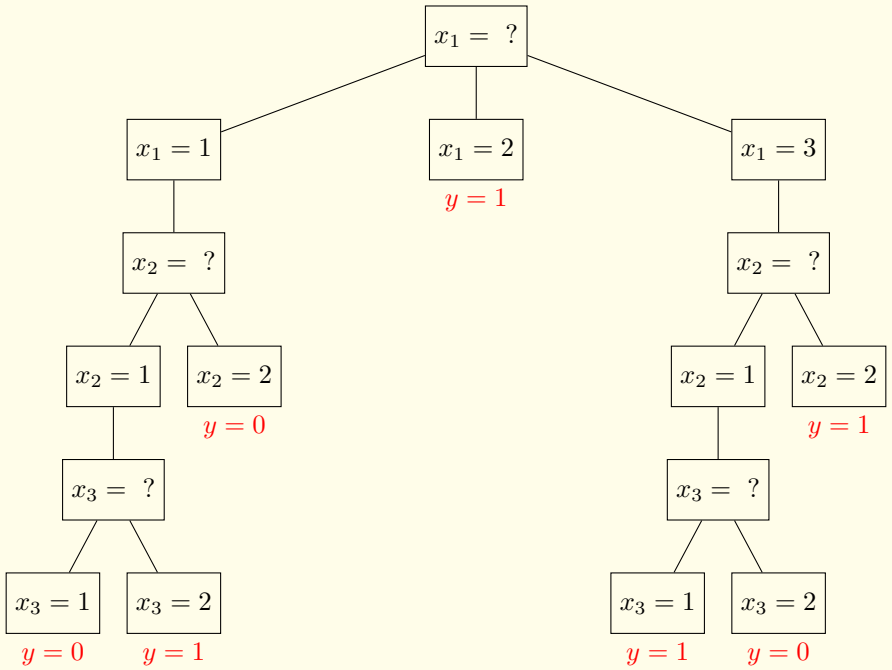


Figure 19: An example of a decision tree that splits at x_1 first, then x_2, and finally x_3. Note that you can still split on x_2 if $x_1 = 1$ and x_3 if $x_1 = 3$.

The decision tree tries to take advantage of some nontrivial covariance between X and Y by constructing nested partitions of the dataset $\mathcal{D}$, and within a partition, it predicts the label that comprises the majority.

For now, let us assume that $\mathcal{X}$ is a Cartesian product of discrete sets, and we will extend them to continuous values later. Let us look at an example to gain some intuition.

Example 6.2 (Restaurant Dataset)

Consider the following dataset.

	OthOptions	Weekend	WaitArea	Plans	Price	Precip	Restaur	Wait	Crowded	Stay?
x_1	Yes	No	No	Yes	$$$	No	Mateo	0-5	some	Yes
x_2	Yes	No	No	Yes	$	No	Juju	16-30	full	No
x_3	No	No	Yes	No	$	No	Pizza	0-5	some	Yes
x_4	Yes	Yes	No	Yes	$	No	Juju	6-15	full	Yes
x_5	Yes	Yes	No	No	$$$	No	Mateo	30+	full	No
x_6	No	No	Yes	Yes	$$	Yes	BlueCorn	0-5	some	Yes
x_7	No	No	Yes	No	$	Yes	Pizza	0-5	none	No
x_8	No	No	No	Yes	$$	Yes	Juju	0-5	some	Yes
x_9	No	Yes	Yes	No	$	Yes	Pizza	30+	full	No
x_{10}	Yes	Yes	Yes	Yes	$$$	No	BlueCorn	6-15	full	No
x_{11}	No	No	No	No	$	No	Juju	0-5	none	No
x_{12}	Yes	Yes	Yes	Yes	$	No	Pizza	16-30	full	Yes

Table 1: Dataset of whether to go to a restaurant for a date depending on certain factors.

Let us denote $\mathcal{D}$ as the dataset, and say that $F_1, \dots, F_d$ were the features. This is a binary classification problem, and we can count that there are 6 positives and 6 negative labels.

The simplest decision tree is the trivial tree, with one node that predicts the majority of the dataset. In this case, the data is evenly split, so without loss of generality we will choose $h_0(\mathbf{x}) = 1$. We want to quantify how good our model is, and so like always we use a loss function.

Just like how a linear model is completely defined by its parameter $\boldsymbol{\theta}$, a decision tree is completely defined by the sequences of labels that it splits on. Therefore, training this is equivalent to defining the sequence, but we can't define this sequence unless we can compare how good a given decision tree is, i.e. unless we have defined a proper loss function. Depending on the training, we can use a greedy algorithm or not, and we have the flexibility to choose whether or not we can split on the same feature multiple times.

Definition 6.5 (Misclassification Error)

We will simply use the misclassification loss function.

$$L(h; \mathcal{D}) = \frac{1}{N} \sum_{i=1}^{N} 1_{\{y^{(i)} \neq h(x^{(i)})\}} = 1 - \text{accuracy} \tag{172}$$

Minimizing this maximizes the accuracy, so this is a reasonable one to choose. How do we train this? Unlike regression, this loss is not continuous, so the gradient is 0, and furthermore the model isn't even parametric, so there are no gradients to derive!

Fortunately, the nature of the decision tree only requires us to look through the explanatory variables $x_1, \dots, x_n$ and decide which one to split.

Let us take a decision tree h and model the accuracy of it as a random variable: $1_{\{Y=h_0(X)\}} \sim \text{Bernoulli}(p)$, where p is the accuracy. A higher accuracy of h corresponds to a lower entropy, and so the entropy of the random variable is also a relevant indicator.

$$H(1_{\{Y=h_0(X)\}}) = p \log p + (1-p) \log(1-p)$$

Therefore, when we are building a tree, we want to choose the feature x_i to split based on how much it lowers the entropy of the decision tree.

To set this up, let us take our dataset $\mathcal{D}$ and set X_i as the random variable representing the distribution (a multinomial) of the $x_i^{(j)}$'s, and Y as the same for the $y^{(j)}$'s. This is our maximum likelihood approximation for the marginalized distribution of the joint measure $X \times Y = X_1 \times \ldots \times X_D \times Y$.

Given a single node, we are simply going to label every point to be whatever the majority class is in $\mathcal{D}$. Therefore, we start off with the entropy of our trivial tree $H(Y)$. Then, we want to see which one of the X_d features to split on, and so we can compute the conditional entropy $H(Y, X_d)$ to get the information gain $I(Y; X_d) = H(Y) - H(Y \mid X_d)$ for all $d = 1, \dots, D$. We want to find a feature X_d that maximize this information gain, i.e. decreases the entropy as much as possible (a greedy algorithm), and we find the next best feature (with or without replacement), so that we have a decreasing sequence.

$$H(X) \geq H(X; Y) \geq H(X; Y, Z) \geq H(X; Y, Z, W) \geq \ldots \geq 0$$

Example 6.3 (Crowded Restaurants)

Continuing the example above, since there are 6 labels of 0 and 1 each, we can model this $Y \sim \text{Bernoulli}(0.5)$ random variable, with entropy

$$H(Y) = \mathbb{E}[-\log_2 p(Y)] = \frac{1}{2}\big(-\log_2 \frac{1}{2}\big) + \frac{1}{2}\big(-\log_2 \frac{1}{2}\big) = 1$$

Now what would happen if we had branched according to how crowded it was, X_{crowded}. Then, our decision tree would split into 3 sections:

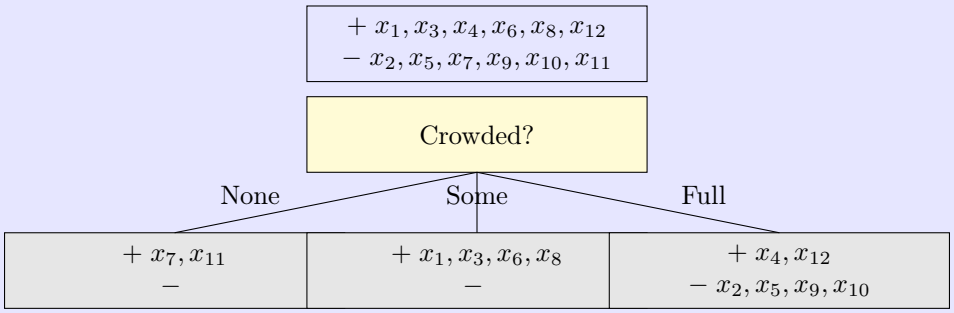


Figure 20: Visual of decision tree splitting according to how crowded it is.

In this case, we can define the multinomial distribution X_{crowded} representing the proportion of the data that is crowded in a specific level. That is, $X_{\text{crowded}} \sim \text{Multinomial}(\frac{2}{12}, \frac{4}{12}, \frac{6}{12})$, with

$$\mathbb{P}(X_{\text{crowded}} = x) = \begin{cases} 2/12 & \text{if } x = \text{ none} \\ 4/12 & \text{if } x = \text{ some} \\ 6/12 & \text{if } x = \text{ full} \end{cases} \tag{173}$$

Therefore, we can now compute the conditional entropy of this new decision tree conditioned on how crowded the store is

$$H(Y \mid X_{\text{crowded}}) = \sum_x \mathbb{P}(X_{\text{crowded}} = x) H(Y \mid X_{\text{crowded}} = x) \tag{174}$$

$$= \frac{2}{12} H(\text{Bern}(1)) + \frac{4}{12} H(\text{Bern}(0)) + \frac{6}{12} H(\text{Bern}(1/3)) = 0.459 \tag{175}$$

$$I(Y; X_{\text{crowded}}) = 0.541 \tag{176}$$

We would do this for all the features and greedily choose the feature that maximizes our information gain.

Example 6.4 (Ferrari F1 Race)

The Ferrari F1 team hired you as a new analyst! You were given the following table of the past race history of the team. You were asked to use information gain to build a decision tree to predict race wins. First, you will need to figure out which feature to split first.

Rain	Good Strategy	Qualifying	Win Race
1	0	0	0
1	0	0	0
1	0	1	0
0	0	1	1
0	0	0	0
0	1	1	1
1	0	1	0
0	1	0	1
0	0	1	1
0	0	1	1

Let $X \sim \text{Bernoulli}(1/2)$ be the distribution of whether a car wins a race over the data. Then its entropy is

$$H(X) = \mathbb{E}[-\log_2 p(x)] = \frac{1}{2}\big(-\log_2 \frac{1}{2}\big) + \frac{1}{2}\big(-\log_2 \frac{1}{2}\big) = 1$$

Let $R \sim \text{Bernoulli}(4/10), G \sim \text{Bernoulli}(2/10), Q \sim \text{Bernoulli}(6/10)$ be the distribution of the features rain, good strategy, and qualifying over the data, respectively. Then, the conditional entropy of X conditioned on each of these random variables is

$$\begin{aligned}
H(X \mid R) &= \mathbb{P}(R=1)\, H(X \mid R=1) + \mathbb{P}(R=0)\, H(X \mid R=0) \\
&= \frac{4}{10} \cdot -\big(1 \cdot \log_2 1 + 0 \cdot \log_2 0\big) + \frac{6}{10} \cdot -\big(\frac{1}{6} \cdot \log_2 \frac{1}{6} + \frac{5}{6} \cdot \log_2 \frac{5}{6}\big) \approx 0.390 \\
H(X \mid G) &= \mathbb{P}(G=1)\, H(X \mid G=1) + \mathbb{P}(G=0)\, H(X \mid G=0) \\
&= \frac{2}{10} \cdot -\big(1 \cdot \log_2 1 + 0 \cdot \log_2 0\big) + \frac{8}{10} \cdot -\big(\frac{3}{8} \cdot \log_2 \frac{3}{8} + \frac{5}{8} \log_2 \frac{5}{8}\big) \approx 0.763 \\
H(X \mid Q) &= \mathbb{P}(Q=1)\, H(X \mid Q=1) + \mathbb{P}(Q=0)\, H(X \mid Q=0) \\
&= \frac{6}{10} \cdot -\big(\frac{4}{6} \cdot \log_2 \frac{4}{6} + \frac{2}{6} \cdot \log_2 \frac{2}{6}\big) + \frac{4}{10} \cdot -\big(\frac{1}{4} \log_2 \frac{1}{4} + \frac{3}{4} \log_2 \frac{3}{4}\big) \approx 0.875
\end{aligned}$$

Therefore, the information gain are

$$\begin{aligned}
I(X;R) &= 1 - 0.390 = 0.610 \\
I(X;G) &= 1 - 0.763 = 0.237 \\
I(X;Q) &= 1 - 0.875 = 0.125
\end{aligned}$$

And so I would split on R, the rain, which gives the biggest information gain.

Finally, we can use the Gini index of $X \sim \text{Bernoulli}(p)$, defined

$$G(X) = 2p(1-p) \tag{177}$$

Example 6.5 (Ferrari Example Continued)

We do the same as the Ferrari example above but now with the Gini reduction. Let $X \sim \text{Bernoulli}(1/2)$ be the distribution of whether a car wins a race over the data. Then its Gini index, which I will label with $\mathcal{G}$, is

$$\mathcal{G}(X) = 2 \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{2}$$

Let $R \sim \text{Bernoulli}(4/10), G \sim \text{Bernoulli}(2/10), Q \sim \text{Bernoulli}(6/10)$ be the distribution of the features rain, good strategy, and qualifying over the data, respectively. Then we compute the conditional expectation

$$\begin{aligned}
\mathbb{E}[\mathcal{G}(X \mid R)] &= \mathbb{P}(R=1)\, \mathcal{G}(X \mid R=1) + \mathbb{P}(R=0)\, \mathcal{G}(X \mid R=0) \\
&= \frac{4}{10}\left[2 \cdot \frac{4}{4} \cdot \frac{0}{4}\right] + \frac{6}{10}\left[2 \cdot \frac{1}{6} \cdot \frac{5}{6}\right] \approx 0.167 \\
\mathbb{E}[\mathcal{G}(X \mid G)] &= \mathbb{P}(G=1)\, \mathcal{G}(X \mid G=1) + \mathbb{P}(G=0)\, \mathcal{G}(X \mid G=0) \\
&= \frac{2}{10}\left[2 \cdot \frac{2}{2} \cdot \frac{0}{2}\right] + \frac{8}{10}\left[2 \cdot \frac{3}{8} \cdot \frac{5}{8}\right] \approx 0.375 \\
\mathbb{E}[\mathcal{G}(X \mid Q)] &= \mathbb{P}(Q=1)\, \mathcal{G}(X \mid Q=1) + \mathbb{P}(Q=0)\, \mathcal{G}(X \mid Q=0) \\
&= \frac{6}{10}\left[2 \cdot \frac{4}{6} \cdot \frac{2}{6}\right] + \frac{4}{10}\left[2 \cdot \frac{1}{4} \cdot \frac{3}{4}\right] \approx 0.417
\end{aligned}$$

Therefore, the Gini reduction, which I'll denote as $I_{\mathcal{G}}$, is

$$\begin{aligned}
I_{\mathcal{G}}(X;R) &= 0.5 - 0.167 = 0.333 \\
I_{\mathcal{G}}(X;G) &= 0.5 - 0.375 = 0.125 \\
I_{\mathcal{G}}(X;Q) &= 0.5 - 0.417 = 0.083
\end{aligned}$$

Since branching across the feature R, the rain, gives the biggest Gini reduction, we want to split on the rain feature first.

6.2.1 Regularization

Given a dataset with D binary features, let $g(H, D)$ be the number of binary trees with depth at most H (including root node), with the restriction that the trees may not split on some variable multiple times within a path to a leaf node. Then, g can be defined recursively.

1. First, if $H = 1$, then $g(H, D) = 1$ always since we are just creating the trivial binary tree of one node.
2. If $D = 0$, then there are no features to split on and therefore we just have the single node $g(H, D) = 1$.
3. If $H > 1$ and $D > 0$, then say that we start with a node. We can either make this a leaf node by not performing any splitting at all, or split on one of the D variables. Then for each of the 2 nodes created on the split, we are now working with $D - 1$ features and a maximum height of $H - 1$ for each of the subtrees generated from the 2 nodes.

All this can be expressed as

$$g(H, D) = \begin{cases} 1 + D\left[g(H-1, D-1)\right]^2 & \text{if } H > 1, D > 0 \\ 1 & \text{if } H = 1 \text{ or } D = 0 \end{cases}$$

which is extremely large (in fact, NP hard). Therefore, some tricks like regularization must be implemented to limit our search space.

By defining the complexity of our decision tree $\Omega(h)$ as the number of nodes within the tree, we can modify our objective function to

$$L(h; \mathcal{D}) = \frac{1}{N} \sum_{i=1}^{N} 1_{\{y^{(i)} \neq h(x^{(i)})\}} + \lambda \Omega(h)$$

We can impose this constraint directly on the training algorithm, or we can calculate the regularized loss after the tree has been constructed, which is a method called **tree pruning**.

Given a large enough λ, we can in fact greatly reduce our search space by not considering any trees further than a certain point.

Theorem 6.1 ()

We describe a tree as a set of leaves, where leaf k is a tuple containing the logical preposition satisfied by the path to leaf k, denoted p_k, and the class label predicted by the leaf, denoted $\hat{y}_k$. For a dataset with d binary features, $p_k : \{0,1\}^d \to \{0,1\}$ is a function that returns 1 if a sample x_i satisfies the preposition, and 0 otherwise. That is, leaf k is $(p_k, \hat{y}_k)$, and a tree f with K leaves is described as a set $f = \{(p_1, \hat{y}_1), \dots, (p_K, \hat{y}_K)\}$. Assume that the label predicted by $\hat{y}_k$ is always the label for the majority of samples satisfying p_k. Finally, let $m_k = \sum_{i=1}^{n} p_k(x_i)$ denote the number of training samples "captured" by leaf k.
Given a (potentially optimal) tree

$$f = \{(p_1, \hat{y}_1), \dots, (p_\kappa, \hat{y}_\kappa), \dots, (p_K, \hat{y}_K)\},$$

the tree $f' = \{(p_1, \hat{y}_1), \dots, (p_{\kappa_1}, \hat{y}_{\kappa_1}), (p_{\kappa_2}, \hat{y}_{\kappa_2}), \dots, (p_K, \hat{y}_K)\}$ produced by splitting leaf $(p_\kappa, \hat{y}_\kappa)$ into two leaves $(p_{\kappa_1}, \hat{y}_{\kappa_1})$ and $(p_{\kappa_2}, \hat{y}_{\kappa_2})$ and any tree produced by further splitting $(p_{\kappa_1}, \hat{y}_{\kappa_1})$ or $(p_{\kappa_2}, \hat{y}_{\kappa_2})$ cannot be optimal if $m_\kappa < 2n\lambda$.

Proof.

Let c be the number of misclassifications in leaf $(p_\kappa, \hat{y}_\kappa)$. Since a leaf classifies according to the majority of m_κ, we must have

$$c \leq \frac{m_\kappa}{2} < n\lambda$$

By splitting leaf $(p_\kappa, \hat{y}_\kappa)$ into leaves $(p_{\kappa_1}, \hat{y}_{\kappa_1})$ and $(p_{\kappa_2}, \hat{y}_{\kappa_2})$, assume that we have reduced the number of misclassifications by $b \leq c$. Then, we have

$$\ell(f', \mathbf{X}, \mathbf{y}) = \ell(f, \mathbf{X}, \mathbf{y}) - \frac{b}{n}$$

However, we have increased the number of leaves by 1, and so

$$\lambda s(f') = \lambda s(f) + \lambda$$

Combining the last two equations, we have obtained

$$R(f', \mathbf{X}, \mathbf{y}) = R(f, \mathbf{X}, \mathbf{y}) + \lambda - \frac{b}{n}$$

However, we know that

$$\begin{aligned} b \leq c &\implies \frac{b}{n} \leq \frac{c}{n} < \frac{n\lambda}{n} = \lambda \\ &\implies -\frac{b}{n} > -\lambda \\ &\implies \lambda - \frac{b}{n} > \lambda - \lambda = 0 \end{aligned}$$

and so $R(f', \mathbf{X}, \mathbf{y}) > R(f, \mathbf{X}, \mathbf{y})$. This means that f' cannot be optimal according to our regularized objective. We have also proved that further splitting $(p_{\kappa_1}, \hat{y}_{\kappa_1})$ or $(p_{\kappa_2}, \hat{y}_{\kappa_2})$ cannot be optimal since we can just set $f = f'$, and apply the same argument.

7 Generalized Linear Models

Remember the linear model looked like this, where we use the conventional β notation to represent parameters.

$$Y = X^T\beta + \epsilon, \;\; \epsilon \sim N(0, \sigma^2 I) \tag{178}$$

which implies that $Y \mid X \sim N(X^T\beta, \sigma^2 I)$. Basically, given x, I assume some distribution of Y, and the value of x will help me guess what the mean of this distribution is. Note that we in here assume that only the mean depends on X. I could potentially have something crazy, like

$$Y \mid X \sim N(X^T\beta, (X^T\gamma)(XX^T + I))$$

where the covariance will depend on X, too, but in this case we only assume that that mean is dependent on X.

$$Y \mid X \sim N(\mu(X), \sigma^2 I)$$

where in the linear model, $\mu(X) = X^T\beta$. So, there are three assumptions we are making here:

1. $Y \mid X$ is Gaussian.
2. X only affects the mean of $Y \mid X$, written $\mathbb{E}[Y \mid X] = \mu(X)$.
3. X affects the mean in a linear way, such that $\mu(X) = X^T\beta$.

So the two things we are trying to relax are:

1. **Random Component**: the response variable $Y \mid X$ is continuous and normally distributed with mean $\mu = \mu(X) = \mathbb{E}[Y \mid X]$.
2. **Link**: I have a link that explains the relationship between the X and the μ, and this relationship is $\mu(X) = X^T\beta$.

So when talking about GLMs, we are not changing the fact that we have a linear function $X \mapsto X^T\beta$. However, we are going to assume that $Y \mid X$ now comes from a broader **family of exponential distributions**. Second, we are going to assume that there exists some **link function** g

$$g(\mu(X)) = X^T\beta$$

Admittedly, this is not the most intuitive way to think about it, since we would like to have $\mu(X) = f(X^T\beta)$, but here we just decide to call $f = g^{-1}$. Therefore, if I want to give you a GLM, I just need to give you two things: the conditional distribution $Y \mid X$, which can be any distribution in the exponential family, and the link function g.

We really only need this link function due to compatibility reasons. Say that $Y \mid X \sim \text{Bern}(p)$. Then, $\mu(X) = \mathbb{E}[Y \mid X]$ always lives in $[0, 1]$, but $X^T\beta$ always lives in $\mathbb{R}$. We want our model to be realistic, and we can clearly see the problem shown in Figure 21.

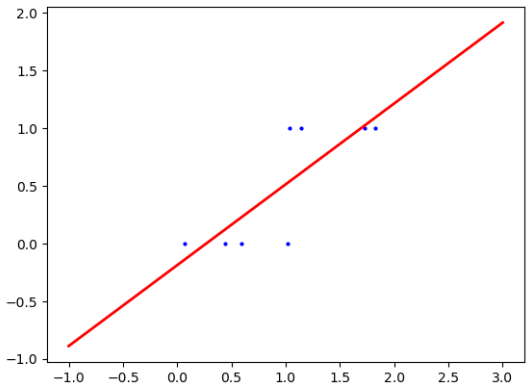


Figure 21: Fitting a linear model for Bernoulli random variables will predict a mean that is outside of $[0, 1]$ when getting new datapoints.

If $Y \mid X$ is some exponential distribution, then its support is always positive and so $\mu(X) > 0$. But if we stick to the old form of $\mu(X) = X^T\beta$, then $\text{Im}(\mu) = \mathbb{R}$, which is not realistic when we predict negative values. Let's take a couple examples:

Example 7.1 (Disease Epidemic)

In the early stages of a disease epidemic, the rate at which new cases occur can often increase exponentially through time. Clearly, $\mu(X) = \mathbb{E}[Y \mid X]$ should be positive and we should have some sort of exponential trend. Hence, if $\mu(x)$ is the expected number of cases on data x, a model of the form

$$\mu(x) = \gamma \exp(\delta x) \tag{179}$$

seems appropriate, where γ and δ are simply scaling factors. Clearly, $\mu(X)$ is not of the form $f(X^T\beta)$. So what I do is to transform μ in such a way that I can get something that is linear.

$$\log(\mu(X)) = \log(\gamma) + \delta X \tag{180}$$

which is now linear in X, of form $\beta_0 + \beta_1 X$. This will have some effects, but this is what needs to be done to have a genearlized linear model. Note that what I did to μ was take the log of it, and so the link function is $g = \log$, called the **log-link**. Now that we have chosen the g, we still need to choose what the conditional distribution $Y \mid X$ would be. This is determined by speaking with industry professionals, experience, and convenience. In this case, Y is a count, and since this must be a discrete distribution. Since it is not bounded above, we think Poisson.

Example 7.2 (Prey Capture Rate)

The rate of capture of preys, Y, by a hunting animal, tends to increase with increasing density of prey X, but eventually level off when the predator is catching as much as it can cope with. We want to find a perhaps concave function that levels off, and suitable model might be

$$\mu(X) = \frac{\alpha X}{h + X} \tag{181}$$

where α represents the maximum capture rate, and h represents the prey density at which the capture rate is half the maximum rate. Again, we must find some transformation g that turns this into a

linear function of X, and what we can do it use the **reciprocal-link**.

$$\frac{1}{\mu(X)} = \frac{h+X}{\alpha X} = \frac{h}{\alpha}\frac{1}{X} + \frac{1}{\alpha} \tag{182}$$

The standard deviation of capture rate might be approximately proportional to the mean rate, suggesting the use of a Gamma distribution for the response.

Example 7.3 (Kyphosis Data)

The Kyphosis data consist of measurements on 81 children following corrective spinal surgery. The binary response variable, Kyphosis, indicates the presence or absence of a postoperative deforming. The three covariates are: age of the child in months, number of the vertebrae involved in the operation, and the start of the range of the vertebrae involved. The response variable is binary so there is no choice: $Y \mid X$ is Bernoulli with expected value $\mu(X) \in (0,1)$. We cannot write $\mu(X) = X^T\beta$ because the right hand side ranges through $\mathbb{R}$, and so we find an invertible function that squishes $\mathbb{R}$ to $(0,1)$, and so we can choose basically any CDF.

For clarification, when writing a distribution like Bernoulli(p), or Binomial(n,p), Poisson(λ), or $N(\mu, \sigma^2)$, the hyperparameters that we usually work with we will denote as $\boldsymbol{\theta}$, and the space that this $\boldsymbol{\theta}$ lives in will denote Θ. For example, for the Bernoulli, $\Theta = [0,1]$, and for Poisson, $\Theta = [0, +\infty)$.

Ultimately, a GLM consists of three steps:

1. The observed input X enters the model through a linear function $\beta^T X$.
2. The conditional mean of response, is represented as a function of the linear combination

$$\mathbb{E}[Y \mid X] = \mu = f(\beta^T X) \tag{183}$$

3. The observed response is drawn from an exponential family distribution with conditional mean μ.

7.1 Exponential Family

We can write the pdf of a distribution as a function of the input x and the hyperparameters θ, so we can write $P_\theta(x) = p(\theta, x)$. For now, let's think that both $x, \theta \in \mathbb{R}$. Think of all the functions that depend on θ and x. There are many of them, but we want θ and x to interact in a certain way. The way that I want them to interact with each other is that they are multiplied within an exponential term. Now clearly, this is not a very rich family, so we are just slapping some terms that depend only on θ and only on x.

$$p_\theta(x) = \exp(\theta x) h(x) c(\theta)$$

But now if $\theta \in \mathbb{R}^k$ and $x \in \mathbb{R}^q$, then we cannot simply take the product nor the inner product, but what we can do is map both of them into a space that has the same dimensions, so I can take the inner product. That is, let us map $\boldsymbol{\theta} \mapsto \boldsymbol{\eta}(\boldsymbol{\theta}) \in \mathbb{R}^k$ and $\mathbf{x} \mapsto \mathbf{T}(\mathbf{x}) \in \mathbb{R}^k$, and so our exponential distribution form would be generalized into something like

$$p_{\boldsymbol{\theta}}(\mathbf{x}) = \exp\left[\boldsymbol{\eta}(\boldsymbol{\theta}) \cdot \mathbf{T}(\mathbf{x})\right] h(\mathbf{x}) c(\boldsymbol{\theta})$$

We can think of $c(\boldsymbol{\theta})$ as the normalizing term that allows us to integrate the pdf to 1.

$$\int_{\mathcal{X}} p_{\boldsymbol{\theta}}(\mathbf{x}) = c(\boldsymbol{\theta}) \int \exp\left[\boldsymbol{\eta}(\boldsymbol{\theta}) \cdot \mathbf{T}(\mathbf{x})\right] h(\mathbf{x})\, d\mathbf{x}$$

We can just push the $c(\boldsymbol{\theta})$ term into the exponential by letting $c(\boldsymbol{\theta}) = e^{-\log(c(\boldsymbol{\theta}))^{-1}}$ to get our definition.

Definition 7.1 (Exponential Family)

A **k-parameter exponential family** is a family of distributions with pdf/pmf of the form

$$p_{\boldsymbol{\theta}}(\mathbf{x}) = \exp\left[\boldsymbol{\eta}(\boldsymbol{\theta}) \cdot \mathbf{T}(\mathbf{x}) - B(\boldsymbol{\theta})\right] h(\mathbf{x})$$

The h term, as we will see, will not matter in our maximum likelihood estimation, so we keep it outside the exponential.

1. $\boldsymbol{\eta}$ is called the **canonical parameter**. Given a distribution parameterized by the regular hyperparameters $\boldsymbol{\theta}$, we would like to parameterize it in a different way $\boldsymbol{\eta}$ under the function $\boldsymbol{\eta} : \Theta \to \mathbb{R}$
2. $\mathbf{T}(\mathbf{x})$ is called the **sufficient statistic**.
3. $h(\mathbf{x})$ is a nonnegative scalar function.
4. $B(\boldsymbol{\theta})$ is the normalizing factor.

Let's look at some examples.

Example 7.4 (Gaussian)

If we put the coefficient into the exponential and expand the square term, we get

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

where

$$\boldsymbol{\eta}(\boldsymbol{\theta}) = \begin{pmatrix} \mu/\sigma^2 \\ -1/2\sigma^2 \end{pmatrix},\ T(x) = \begin{pmatrix} x \\ x^2 \end{pmatrix},\ B(\theta) = \frac{\mu^2}{2\sigma^2} + \log(\sigma\sqrt{2\pi}),\ h(x) = 1$$

This is not a unique representation since we can take the $\log(\sqrt{2\pi})$ out of the exponential, but why bother to do this when we can just stuff everything into B and keep h simple.

Example 7.5 (Gaussian with Known Variance)

If we have known variance, we can write the Gaussian pdf as

$$p_\theta(x) = \exp\left[\frac{\mu}{\sigma} \cdot \frac{x}{\sigma} - \frac{\mu^2}{2\sigma^2}\right] \cdot \frac{1}{\sigma\sqrt{2\pi}} e^{x^2/2\sigma^2}$$

where

$$\eta(\theta) = \frac{\mu}{\sigma},\ T(x) = \frac{x}{\sigma},\ B(\theta) = \frac{\mu^2}{2\sigma^2},\ h(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{x^2/2\sigma^2}$$

Example 7.6 (Bernoulli)

The pmf of a Bernoulli with θ is

$$\begin{aligned} p_\theta(x) &= \theta^x (1-\theta)^{(1-x)} \\ &= \exp\left[x \log(\theta) + (1-x)\log(1-\theta)\right] \\ &= \exp\left(x \log\left[\frac{\theta}{1-\theta}\right] - \log\left[\frac{1}{1-\theta}\right]\right) \end{aligned}$$

where

$$\eta(\theta) = \log\left[\frac{\theta}{1-\theta}\right],\ T(x) = x,\ B(\theta) = \log\left[\frac{1}{1-\theta}\right],\ h(x) = 1$$

Example 7.7 (Binomial with Known Number of Trials)

We can transform a binomial with known N as

$$\begin{aligned} p_\theta(x) &= \binom{N}{x} \theta^x (1-\theta)^{1-x} \\ &= \exp\left[x \ln\left(\frac{\theta}{1-\theta}\right) + \ln(1-\theta) \right] \cdot \binom{N}{x} \end{aligned}$$

where

$$\eta(\theta) = \ln\left(\frac{\theta}{1-\theta}\right),\ T(x) = x,\ B(\theta) = \ln(1-\theta),\ h(x) = \binom{N}{x}$$

Example 7.8 (Poisson)

The pmf of Poisson with θ can be expanded

$$\begin{aligned} p_\theta &= \frac{\theta^{-x}}{x!} e^{-\theta} \\ &= \exp\left[-\theta + x \log(\theta) - \log(x!) \right] \\ &= \exp\left[x \log(\theta) - \theta \right] \frac{1}{x!} \end{aligned}$$

where

$$\eta(\theta) = \log(\theta),\ T(x) = x,\ B(\theta) = \theta,\ h(x) = \frac{1}{x!}$$

However, the uniform is not in here. In fact, any distribution that has a support that does not depend on the parameter is not an exponential distribution.

Let us now focus on one parameter families where $\theta \in \Theta \subset \mathbb{R}$, which do not include the Gaussian (with unknown mean and variance, Gamma, multinomial, etc.), which has a pdf written in the form

$$p_\theta(x) = \exp\left[\eta(\theta)\, T(x) - B(\theta) \right] h(x)$$

7.1.1 Canonical Exponential Family

Now a common strategy in statistical analysis is to reparamaterize a probability distribution. Suppose a family of probability distributions $\{P_\theta\}$ is parameterized by $\theta \in \Theta \subset \mathbb{R}$. If we have an invertible function $\eta : \Theta \to \mathcal{T} \subset \mathbb{R}$, then we can paramaterize the same family with η rather than θ, with no loss of information. Typically, it is the case that η is invertible for exponential families, so we can just reparameterize the whole pdf and write

$$p_\eta(x) = \exp\left[\eta\, T(x) - \phi(\eta) \right] h(x)$$

where $\phi = B \circ \eta^{-1}$.

Definition 7.2 (Canonical One-Parameter Exponential Family)

A family of distributions is said to be in **canonical one-parameter exponential family** if its density is of form

$$p_\eta(x) = \exp\left[\eta\, T(x) - \phi(\eta) \right] h(x)$$

which is a subfamily of the exponential family. The function ψ is called the **cumulant generating function**.

Before we move on, let us just provide a few examples.

Example 7.9 (Poisson)

The Poisson can be represented as

$$p_\theta(x) = \exp\left[x \log \theta - \theta\right] \frac{1}{x!}$$

Now let $\eta = \log \theta \implies \theta = e^\eta$. So, we can reparamaterize the density as

$$p_\eta(x) = \exp\left[x\eta - e^\eta\right] \frac{1}{x!}$$

where $P_\eta = \text{Poisson}(e^\eta)$ for $\eta \in \mathcal{T} = \mathbb{R}$, compared to $P_\theta = \text{Poisson}(\theta)$ for $\theta \in \Theta = \mathbb{R}^+$.

Example 7.10 (Gaussian)

Recall that the Gaussian with known parameter σ^2 and unknown $\theta = \mu$ is in the exponential family, since we can expand it as

$$p_\theta(x) = \exp\left[\frac{\mu}{\sigma^2} \cdot x - \frac{\mu^2}{2\sigma^2}\right] \cdot \frac{1}{\sigma\sqrt{2\pi}} e^{x^2/2\sigma^2}$$

We can perform the change of parameter $\eta = \mu^2/2\sigma^2 \implies \mu = \sigma^2\eta$, and substituting this in will give the canonical representation

$$p_\eta(x) = \exp\left[\eta x - \frac{\sigma^2\eta^2}{2}\right] \cdot \frac{1}{\sigma\sqrt{2\pi}} e^{x^2/2\sigma^2}$$

where now $P_\eta = N(\sigma^2\eta, \sigma^2)$ for $\eta \in \mathcal{T} = \mathbb{R}$, compared to $P_\theta = N(\theta, \sigma^2)$ for $\theta \in \Theta = \mathbb{R}$, which is basically the same space.

Example 7.11 (Bernoulli)

The Bernoulli has an exponential form of

$$p_\theta(x) = \exp\left[x \log\left(\frac{\theta}{1-\theta}\right) + \log(1-\theta)\right]$$

Now setting $\eta = \log\left(\frac{\theta}{1-\theta}\right) \implies \theta = \frac{1}{1+e^{-\eta}}$, and so $B(\theta) = -\log(1-\theta) = -\log\left(\frac{e^{-\eta}}{1+e^{-\eta}}\right) = \log(1+e^\eta) = \psi(\eta)$, and so the canonical paramaterization is

$$p_\eta(x) = \exp\left[x\eta - \log(1+e^\eta)\right]$$

We present two useful properties of the exponential family.

Theorem 7.1 (Moments)

Let random variable X be in the canonical exponential family P_η

$$p_\eta(x) = e^{\eta T(x) - \psi(\eta)} h(x)$$

Then, the expectation and variance are encoded in the cumulant generating function in the following way

$$\mathbb{E}[T(X)] = \psi'(\eta) \qquad \text{Var}[T(X)] = \psi''(\eta)$$

Proof.

Example 7.12 ()

We show that this is consistent with the Poisson, normal, and Bernoulli distributions.

1. In the Poisson, $\psi(\eta) = e^\eta$, and so $\psi'(\eta) = e^\eta = \theta = \mathbb{E}[X]$. Taking the second derivative gives $\psi''(\eta) = e^\eta = \theta = \text{Var}[X]$, too.
2. In the Normal with known variance σ^2, we have $\psi(\eta) = \frac{1}{2}\sigma^2\eta^2$. So

$$\mathbb{E}[X] = \psi'(\eta) = \sigma^2\eta = \mu$$
$$\text{Var}[X] = \psi''(\eta) = \sigma^2$$

3. In the Bernoulli, we have $\psi(\eta) = \log(1 + e^{-\eta})$. Therefore,

$$\mathbb{E}[X] = \psi'(\eta) = \frac{x^\eta}{1 + x^\eta} = \frac{1}{1 + e^{-\eta}} = \theta$$
$$\text{Var}[X] = \psi''(\eta) = -\left(\frac{1}{1 + e^{-\eta}}\right)^2 e^{-\eta} \cdot -1 = \theta^2 \cdot \frac{1 - \theta}{\theta} = \theta(1 - \theta)$$

Theorem 7.2 (Convexity)

Consider a canonical exponential family with density

$$p_\eta(x) = e^{\eta T(x) - \psi(\eta)} h(x)$$

and natural parameter space $\mathcal{T}$. Then, the set $\mathcal{T}$ is convex, and the cumulant generating function ψ is convex on $\mathcal{T}$.

Proof.

This can be proven using Holder's inequality. However, from the theorem above, note that $\text{Var}[T(X)] = \psi''(\eta)$ must be positive since we are talking about variance. This implies that the second derivative of ψ is positive, and therefore must be convex.

We will look at a subfamily of the exponential family. Now remember that we introduce the functions $\boldsymbol{\eta}$ and $\mathbf{T}$ so that we can capture a much broader range of distributions, but if we have one parameter $k = 1$, then we can just set $\boldsymbol{\eta}(\boldsymbol{\theta})$ to be the new parameter θ. The **canonical exponential family** for $k = 1, y \in \mathbb{R}$, is defined to have the pdf

$$f_\theta(y) = \exp\left(\frac{y\theta - b(\theta)}{\phi} + c(y, \phi)\right) \tag{184}$$

where

$$h(y) = \exp\big(c(y, \phi)\big) \tag{185}$$

If ϕ is known, this is a one-parameter exponential family with θ being the **canonical parameter**, and if ϕ is unknown, the $h(y)$ term will not depend on θ, which we may not be able to split up into the exponential pdf form. In this case ϕ is called the **dispersion parameter**. For now, we will always assume that ϕ is known.

We can prove this for all other classes, too. We can think of the $c(y, \phi)$ as just a term that we stuff every other term into. What really differentiates the different distributions of the canonical exponential family is the $b(\theta)$. The form of b will determine whether this distribution is a Gaussian, or a Bernoulli, or etc. This b will capture information about the mean, the variance, the likelihood, about everything.

7.2 Cumulant Generating Function

Definition 7.3 (Score)

The **score** is the gradient of the log-likelihood function with respect to the parameter vector. That is, given that $L(\boldsymbol{\theta})$ is the likelihood, then

$$s(\boldsymbol{\theta}) \coloneqq \frac{\partial \log L(\boldsymbol{\theta}; \mathbf{x})}{\partial \boldsymbol{\theta}}$$

which gives a row covector.

Now, remember that the score also depends on the observations $\mathbf{x}$. If we rewrite the likelihood as a probability density function $L(\boldsymbol{\theta}; \mathbf{x}) = f(\mathbf{x}; \boldsymbol{\theta})$, then we can say that the expected value of the score is equal to 0, since

$$\begin{aligned}
\mathbb{E}[s(\boldsymbol{\theta})] &= \int_{\mathcal{X}} f(\mathbf{x}; \boldsymbol{\theta}) \frac{\partial}{\partial \boldsymbol{\theta}} \log L(\boldsymbol{\theta}; \mathbf{x}) \, d\mathbf{x} \\
&= \int_{\mathcal{X}} f(\mathbf{x}; \boldsymbol{\theta}) \frac{1}{f(\mathbf{x}; \boldsymbol{\theta})} \frac{\partial f(\mathbf{x}; \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \, dx \\
&= \frac{\partial}{\partial \boldsymbol{\theta}} \int_{\mathcal{X}} f(\mathbf{x}; \boldsymbol{\theta}) \, d\mathbf{x} \\
&= \frac{\partial}{\partial \boldsymbol{\theta}} 1 = 0
\end{aligned}$$

where we take a leap of faith in switching the derivative and integral in the penultimate line. Furthermore, we can get the second identity

$$\mathbb{E}\left[\frac{\partial^2 \ell}{\partial \theta^2}\right] + \mathbb{E}\left[\frac{\partial \ell}{\partial \theta}\right]^2 = 0$$

We can apply these two identities as follows. Since

$$\ell(\theta) = \frac{Y\theta - b(\theta)}{\phi} + c(Y; \phi)$$

therefore

$$\frac{\partial \ell}{\partial \theta} = \frac{Y - b'(\theta)}{\phi}$$

which yields

$$0 = \mathbb{E}\left[\frac{\partial \ell}{\partial \theta}\right] = \frac{\mathbb{E}[Y] - b'(\theta)}{\phi} \implies \mathbb{E}[Y] = \mu = b'(\theta)$$

On the other hand, we have

$$\frac{\partial^2 \ell}{\partial \theta^2} + \left(\frac{\partial \ell}{\partial \theta}\right)^2 = -\frac{b''(\theta)}{\phi} + \left(\frac{Y - b'(\theta)}{\phi}\right)^2$$

and from the previous result, we get

$$\frac{Y - b'(\theta)}{\phi} = \frac{Y - \mathbb{E}[Y]}{\phi}$$

together with the second identity, yields

$$0 = -\frac{b''(\theta)}{\phi} + \frac{\mathrm{Var}(Y)}{\phi^2} \implies \mathrm{Var}(Y) = \phi\, ''(\theta)$$

Since variance is always positive, this implies that $b'' > 0$ and therefore b must be convex.

7.3 Link Functions

Now let's go back to GLMs. In linear models, we said that the conditional expectation of Y given $X = \mathbf{x}$ must be a linear function in x

$$\mathbb{E}[Y \mid X = \mathbf{x}] = \mu(\mathbf{x}) = \mathbf{x}^T\beta$$

But if the conditional distribution takes values in some subset of $\mathbb{R}$, such as $(0, 1)$, then it may not make sense to write this as a linear function, since $X^T\beta$ has an image spanning $\mathbb{R}$. So what we need is a link function that relates, i.e. transforms the restricted subset of μ, onto the real line, so that now you can express it of the form $X^T\beta$.

$$g\big(\mu(X)\big) = X^T\beta$$

Again, it is a bit more intuitive to talk about g^{-1}, which takes our $X^T\beta$ and transforms it to the values that I want, so we will talk about both of them simultaneously. If g is our link function, we want it to satisfy 3 requirements:

1. g is continuously differentiable
2. g is strictly increasing
3. $\text{Im}(g) = \mathbb{R}$, i.e. it spans the entire real line

This implies that g^{-1} exists, which is also continuously differentable and is strictly increasing.

Example 7.13 ()

If I have a conditional distribution...

1. that is Poisson, then we want our μ to be positive, and so we need a link function $g : \mathbb{R}^+ \to \mathbb{R}$. One choice would be the logarithm

$$g(\mu(X)) = \log\big(\mu(X)\big) = X^T\beta$$

2. that is Bernoulli, then we want our μ to be in $(0, 1)$ and we need a link function $g : (0, 1) \to \mathbb{R}$. There are 2 natural choices, which may be the **logit** function

$$g(\mu(X)) = \log\left(\frac{\mu(X)}{1 - \mu(X)}\right) = X^T\beta$$

or the **probit** function

$$g(\mu(X)) = \Phi^{-1}\big(\mu(X)\big) = X^T\beta$$

where Φ is the CDF of a standard Gaussian. The two functions can be seen in Figure 22.

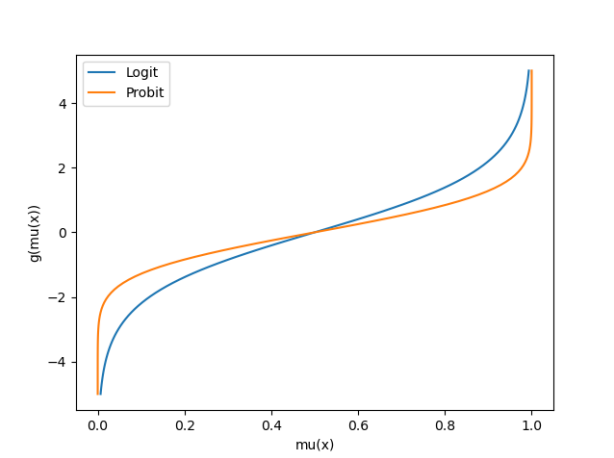


Figure 22: Logit and Probit Functions

Now there are many choices of functions we can take. In fact, if μ lives in $(0, 1)$, then we can really just take our favorite distribution that has a density that is supported everywhere in $\mathbb{R}$ and take the inverse cdf as our link. So far, we have no reason to prefer one function to another, but in the next section, we will see that there are more natural choices.

7.3.1 Canonical Link Functions

Now let's summarize what we have. We assume that the conditional distribution $Y \mid X = x$ follows a distribution in the exponential family, which we can completely characterize by the cumulant generating function ψ. For different values of x, the conditional distribution will be paramaterized by different $\eta(x)$, and the resulting distribution P_η will have some mean $\mu(x)$, which is usually not the natural parameter η. Now, let's forget about our knowledge that $\psi'(\eta) = \mu$, but we know that there is some relationship between $\eta \leftrightarrow \mu$.

Given an x, I need to use the linear predictor $x^T\beta$ to predict $\mu(x)$, which can be done through the link function g.

$$g\big(\mu(x)\big) = x^T\beta$$

Now what would be a natural way of choosing this g? Note that our natural parameter η for this canonical family takes value on the entire real line. I must construct a function g that maps μ onto the entire real line, and so why not make it map to η. Therefore, we have

$$\eta(x) = g\big(\mu(x)\big) = x^T\beta$$

Definition 7.4 (Canonical Link)

The function g that links the mean μ to the canonical parameter θ is called the **canonical link**.

$$g(\mu) = \theta$$

Now using our knowledge that $\psi'(\eta) = \mu$, we can see that

$$g = (\psi')^{-1}$$

This is indeed a valid link function.

1. $\psi'' > 0$ since it models the variance, and so ψ' is strictly increasing and so $g = (\psi')^{-1}$ is also strictly increasing.
2. The domain of ψ' is the real line since it takes in the natural parameter η which exists over $\mathbb{R}$, so $\text{Im}(g) = \mathbb{R}$.

So, given our cumulant generating function ψ and our link function g, both satisfying

$$\psi'(\eta) = \mu \text{ and } g(\mu) = x^T\beta$$

we can combine them to get

$$(g \circ \psi')(\eta) = g(\mu) = x^T\beta$$

and so, even though the mean of the response variable is not linear with respect to x, the value of $(g \circ \psi')(\eta)$ is indeed linear. In fact, if we choose the canonical link, then the equation

$$\eta = x^T\beta$$

means that the natural parameter of our conditional distribution in the exponential family is linear with respect to x! From this we can find the conditional mean $\mu(x)$.

The reason we focus on canonical link functions is because, when the canonical link is used, the components of the model (the parameters of the linear predictor) have an additive effect on the response variable in the

transformed (linked) scale, which makes the interpretation of the results easier. It's also worth noting that while using the canonical link function has some desirable properties, it is not always the best or only choice, and other link functions may be used if they provide a better fit for the data or make more sense in the context of the problem at hand.

Let us evaluate some canonical link functions.

Example 7.14 ()

The Bernoulli has the canonical exponential form of

$$p_\eta(x) = \exp\big[x\eta - \log(1+e^\eta)\big]$$

where $\eta = \log\left(\frac{\theta}{1-\theta}\right)$. Since we have prior knowledge that $\theta = \mu$ (i.e. the expectation of a Bernoulli is the hyperparameter θ itself), we have a function that maps $\mu \mapsto \eta$.

$$\eta = g(\mu) = \log\left(\frac{\mu}{1-\mu}\right)$$

which gives us our result. We can also take the inverse of $\psi' = \frac{e^\eta}{1+e^\eta}$ to get our result

$$g(\mu) = (\psi')^{-1}(\mu) = \log\left(\frac{\mu}{1-\mu}\right)$$

7.4 Likelihood Optimization

Now let us have a bunch of data points $\{(x_n, y_n)\}_{n=1}^N$. By our model assumption, we know that the conditional distribution $Y \mid X = x_n$ is now of an exponential family with parameter $\eta_n = \eta(x_n)$ and density

$$p_{\eta_n}(y_n) = \exp\big[y_n\eta_n - \psi(\eta_n)\big]h(y_n)$$

Now we want to do likelihood optimization on β (not η or μ), and to do this, we must rewrite the density function in a way so that it depends on β. Given a link function g, note the following relationship between β and η:

$$\begin{aligned}\eta_n = \eta(x_n) &= (\psi')^{-1}(\mu(x_n)) \\ &= (\psi')^{-1}\big(g^{-1}(x_n^T\beta)\big) \\ &= h(x_n^T\beta)\end{aligned}$$

where for shorthand notation, we define $h \coloneqq (g \circ \psi')^{-1}$. Subtituting this into the above likelihood, taking the product of all N samples, and logarithming the equation gives us the following log likelihood to optimize over β.

$$\ell(\beta) = \log \prod_{n=1}^N p_{\eta_n}(y_n) = \sum_{n=1}^N y_n h(x_n^T\beta) - \psi(h(x_n^T\beta))$$

where we dropped the $h(y_n)$ term at the end since it is a constant and does not matter. If g was the canonical link, then h is the identity, and we should have a linear relationship between $\eta(x_n) = x_n^T\beta$. This means that the η_n reduces only to $x_n^T\beta$, which is much more simple to optimize.

$$\ell(\beta) = \log \prod_{n=1}^N p_{\eta_n}(y_n) = \sum_{n=1}^N y_n x_n^T\beta - \psi(x_n^T\beta)$$

Note that the first term is linear w.r.t β, and ψ is convex, so the entire sum must be concave w.r.t. β. With this, we can bring in some tools of convex optimization to solve.

8 Ensemble Methods

The bias variance noise decomposition gives us a very nice way of explaining overfittting. That is, the bias (expectation of the squared difference between the true $\mathbb{E}[Y \mid X]$ and the expected trained hypothesis function $h_{\boldsymbol{\theta};\mathcal{D}}$) reduces, but the variance in this overfitted model increases. Therefore, if we had a slightly different dataset $\mathcal{D}$ sampled from $(X \times Y)^N$, then we might have a very different trained hypothesis since it's so sensitive to the data.

A way to treat this is through **ensemble learning**, where we train *multiple* models over slightly different datasets, and then average their predictions to get a better model. What do we mean by a better model? Well, we know that a too complex model has low bias but large variance, and a too simple model has high bias but low variance.

1. *Bagging* refers to taking a complex model and decreasing its variance. Even though each model is trained over a smaller dataset, resulting it being more noisy, the average of all these slightly more noisy models will hopefully bring down the variance more than what we have added.[10]
2. *Boosting* refers to taking a simple model and decreasing its bias. Each simple model, usually a weak learner, has relatively small search space, but by taking the aggregate of them, we can hopefully increase it whilst bounding the variance in some way. Usually, the dataset is reweighted such that the weak learner in the next iteration will correct the previous weak learner.

8.1 Bagging

Let's start off with the simpler of the two.

Definition 8.1 (Bootstrap Aggregating)

Given a dataset $\mathcal{D}$ of N samples and a model $\mathcal{M}$, **bagging** is an ensemble method done with two steps:

1. *Bootstrap.* Sample $\tilde{N}$ data points with replacement from $\mathcal{D}$ to get a dataset $\mathcal{D}_1$, and do this M times to get
$$\mathcal{D}_1, \mathcal{D}_2, \dots, \mathcal{D}_M \subset \mathcal{D}$$
2. *Aggregate.* For each sub dataset $\mathcal{D}_m$, train our model to get the optimal hypothesis $h^*_{\mathcal{D}_m}$. We should have M different hypothesis functions, each trained on each sub dataset.
$$h^*_{\mathcal{D}_1}, h^*_{\mathcal{D}_2}, \dots, h^*_{\mathcal{D}_M}$$

To predict the output on a new value $\hat{\mathbf{x}}$, we can evaluate all the $h^*_{\mathcal{D}_m}(\hat{\mathbf{x}})$.

Note that the bootstrapping step could be expanded to different types of subsampling.

Definition 8.2 (Pasting)

If random subsets (without replacement) are sampled from the original dataset $\mathcal{D}$, then this method is known as **pasting**.

Definition 8.3 (Random Subspaces)

When random subsets of the data are drawn as random subsets of the features, then this is known as **random subspaces**.

[10] This is why random forests have underlying trees that are somewhat as large as possible.

Definition 8.4 (Random Patches)

When random subsets of both the data and the features are chosen, then this is known as **random patches**.

Since the whole point of this algorithm is to reduce variance, bagging does not really overfit.

8.2 Random Forests

Definition 8.5 (Random Forests)

A **random forest** is a (random patch) bagging algorithm where the component models are decision trees.

8.3 Boosting

Now we delve more into the applied and computational aspects of machine learning. It's had a lot of empirical success and is more interesting from a theoretical perspective. It starts off with the *weak learning assumption*, which we introduce in the context of classification with the misclassification loss function. It is actually a specific case of PAC learners.

Definition 8.6 (Probability Approximately Correct Learner)

A **PAC learning** is an algorithm that learns a function class $\mathcal{H}$ with parameter $\delta > 0$ if there exists an $\epsilon > 0$ and the algorithm can find a $f \in \mathcal{H}$ with probability at least $1 - \delta$ s.t.

$$R(f) \leq \epsilon \tag{186}$$

i.e.

$$\mathbb{P}[R(f) \leq \epsilon] \geq 1 - \delta \tag{187}$$

This is quite a strong requirement, since it says that with probability at least $1 - \delta$ you must find an model f that is correct with a probability of $1 - \epsilon$, i.e. ϵ-good.

Definition 8.7 (Weak Learner)

A **weak learner** is an algorithm that learns a function class $\mathcal{H}$ with parameter $\delta > 0$ if there exists an $\gamma > 0$ and the algorithm can find a $f \in \mathcal{H}$ s.t.

$$\mathbb{P}[R(f) < 1/2 - \gamma] \geq 1 - \delta \tag{188}$$

for some $\delta > 0$, where γ is considered our edge. Another way to write it is that with probability of at least $1 - \delta$, we can find a function f s.t.

$$\mathbb{P}_{x,y \sim \mathcal{X} \times \mathcal{Y}}[f(x) \neq y] < 1/2 - \gamma \tag{189}$$

This essentially means that given some γ that measures how good our target predictor is compared to random guessing, the probability that we can find such a predictor with this edge is $1-\delta$. Furthermore, this case must hold true for all distributions $P \sim \mathcal{X} \times \mathcal{Y}$.

Therefore, a weak learner just means some algorithm that learns a model that is a bit better than random. For example, learning decision stumps may be a weak learner. Colloquially, a weak learner can be thought of as an algorithm that cannot get your training error to 0 and a strong learner can. The question is then, can we make a strong learner out of a bunch of weak learners? The general idea is that we want to iteratively

find a bunch of weak learners and slowly add them up to get a strong learner.

$$f = \sum_{i=1}^{n} f_i \tag{190}$$

where f is strong, f_i weak.

8.3.1 Adaptive Boosting (AdaBoost)

Let's start with Adaboost for binary classification.

Definition 8.8 (Adaboost for Binary Classification)

Given data $\{(x_i, y_i)\} \in \mathcal{X} \times \mathcal{Y}$, with $\mathcal{Y} = \{-1, +1\}$, we implement the following greedy algorithm.

1. You first set an n-vector weighting your samples, where the weight of the ith sample is $W_t(i)$.

$$W_1 = (\frac{1}{n}, \dots, \frac{1}{n}) \tag{191}$$

2. For $t = 1, \dots, T$, we do the following.
   (a) You run your weak learning algorithm, which will return your hypothesis h_t with probability $1 - \delta$ which is slightly better than random. We define its empirical error over the distribution W_t to be

$$\epsilon_t = R_{W_t}(h_t) = \mathbb{P}_{x_i \sim W_t}[h_t(x_i) \neq y_i] = \sum_{i=1}^{n} W_t(i) \cdot \mathbb{1}_{h_t(x_i) \neq y_i} \tag{192}$$

   This may be done differently by actually sampling n samples from this distribution and then computing proportion of misclassifications.
   (b) This new weak learner provides some information on the new weighted distribution. We would like to weight this weak learner h_t with some scale α_t to determine how important its vote is in the ensemble. We define this weighting to be

$$\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \epsilon_t}{\epsilon_t} \right) \tag{193}$$

   Note the following important properties. If $0 < \epsilon_t < 0.5$, then this does indeed mean that h_t is slightly better than random, so it would have a positive weighting. If $\epsilon_t = 0.5$, then it is random so no weighting. Finally, if $0.5 < \epsilon < 1.0$, then it is an extremely poor classifier and we are better off looking at the opposite of its prediction, meaning that α_t will be negative. This is also seen with the facts that as $\epsilon_t \to 0, 1$, then $\alpha_t \to +\infty, -\infty$.[a]
   (c) Then we set

$$W_{t+1}(i) \propto W_t(i) \exp\{-\alpha_t y_i h_t(x_i)\} = \begin{cases} e^{-\alpha_t} & \text{if } h_t(x_i) = y_i \\ e^{+\alpha_t} & \text{if } h_t(x_i) \neq y_i \end{cases} \tag{194}$$

   meaning that the new weights go up for incorrect labels and down for correct labels. We show proportional to since it is not normalized, but we can normalize it with the constant Z_t.
3. Your final strong classifier is then

$$f(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x) \right) \tag{195}$$

   which indeed is a sequential sum of these classifiers.

[a] In practice, ϵ cannot be 0 or 1 due to numerical reasons, so a small constant is added to prevent this from happening.

In this way, by weighting the incorrect labels higher, I am telling successive weak learner to give me a new weak hypothesis that tells me something new. This makes it so that the actual sequence of learned weak models are important, since the next h_{t+1} tries to fix the errors that the h_t makes.

Algorithm 8.1 (AdaBoost Algorithm)

The full algorithm for brevity is shown below.

Require: Training data $\{(x_i, y_i)\}_{i=1}^n$ where $x_i \in \mathcal{X}$, $y_i \in \{-1, +1\}$
Require: Number of iterations T
Require: Weak learning algorithm $\mathcal{A}$
Initialize weights $W_1(i) = \frac{1}{n}$ for $i = 1, \dots, n$
for $t = 1$ to T **do**
    Train weak learner $h_t = \mathcal{A}(\{(x_i, y_i)\}, W_t)$
    Calculate weighted error:
    $\epsilon_t = \sum_{i=1}^n W_t(i) \cdot \mathbb{1}_{h_t(x_i) \neq y_i}$
    if $\epsilon_t \geq \frac{1}{2}$ **then**
        break
    end if
    Calculate importance weight:
    $\alpha_t = \frac{1}{2} \ln(\frac{1-\epsilon_t}{\epsilon_t})$
    Update sample weights:
    for $i = 1$ to n **do**
        $W_{t+1}(i) = W_t(i) \cdot \exp(-\alpha_t y_i h_t(x_i))$
    end for
    Normalize weights:
    $Z_t = \sum_{i=1}^n W_{t+1}(i)$
    $W_{t+1}(i) = \frac{W_{t+1}(i)}{Z_t}$ for all i
end for
return Final classifier $f(x) = \text{sign}\left(\sum_{t=1}^T \alpha_t h_t(x)\right)$

We now actually show that this is a strong learner by showing that the training error goes to 0.

Theorem 8.1 (Exponential Decay of Training Error in AdaBoost)

Support that $\gamma \leq (1/2) - \epsilon_t$ for all t. Then our empirical risk decays exponentially with T.

$$\hat{R}(h) \leq e^{-2\gamma^2 T} \tag{196}$$

and hence, the training error goes to 0 quickly.

Proof.

Can be shown with the lemma.

$$Z_t = 2\sqrt{\epsilon_t(1-\epsilon_t)} \tag{197}$$

Sure, the training error goes to 0, but what we really care about is the generalization error. It turns out that we can prove things about this, but omitted for now.

Example 8.1 (AdaBoost with Stumps)

We can define our weak learning algorithm to be a decision stump with only 1 split. Doing adaboost gives something similar to a random forest (but not quite since its a bagging algorithm) with great generalization error.

Surprisingly, Adaboost has a tendency not to overfit, i.e. the variance does not explode. There is a lot of theory that tries to explain why this is the case, such as margin theory.

There are a lot of different ways to analyze AdaBoost. For many years, researchers did not think of it as having any connection to gradient descent or loss functions, but it actually does. AdaBoost can be viewed as optimizing the **exponential loss**

$$L(\mathbf{x}, y) = e^{-yf(\mathbf{x})} \tag{198}$$

so that the full empirical objective function is

$$L = \sum_i \exp\left(-\frac{1}{2} y_i \sum_{t=1}^{T} \alpha_t f_t(\mathbf{x}_i) \right) \tag{199}$$

which must be optimize w.r.t. the weights α_t and the parameters of each weak classifier f_t. This stepwise optimization is greedy and sequential, where we add one weak classifier at a time, choosing its parameters and α_t to be optimal w.r.t. L and then never change it again. It turns out that if we actually do keep things constant and solve the optimal parameters, it must be the case that $\alpha_t = \ln \frac{1-\epsilon_t}{\epsilon_t}$, which is why it is in the algorithm.[11] Furthermore, the exponential loss is an upper-bound on the misclassification loss, so if an exponential loss of 0 is achieved, then all training points are correctly classified.

8.3.2 Gradient Boosting

Gradient boosting can deal with both regression and classification problems, and so we will present it in full generality.

Definition 8.9 (Gradient Boosting)

Let us have data $\{(x_i, y_i)\} \in \mathcal{X} \times \mathcal{Y}$ and a differentiable loss function

$$L(\mathbf{y}, \hat{\mathbf{y}}) = \sum_{i=1}^{n} L(y_i, \hat{y}_i) \tag{200}$$

with also a *constant* stepsize η.

1. We first initialize the model with a constant value that minimizes the loss. So we have a single leaf as in our decision tree ensemble.

$$F_0 = \operatorname*{argmin}_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma) \tag{201}$$

   If we're doing regression with the MSE loss, then $\gamma = \bar{y}$, the average. This is our first predictor, which predicts $F_0(x) = \gamma$ for all x, and so F_0 is really just the constant n-vector $(\bar{y}, \dots, \bar{y})$. If we're doing binary classification, we can focus on the logits and initialize γ as the log-odds $\log(\frac{C_+}{C_-})$

2. For $t = 1, \dots, T$, we do the following.
   (a) We have the predicted values $F_{t-1}(x_i)$ for each sample x_i. We compute the negative gradient of the loss function w.r.t. the predicted value.

$$\mathbf{r}_t = -\frac{\partial L(\mathbf{y}, \hat{\mathbf{y}})}{\partial \hat{\mathbf{y}}}\bigg|_{\hat{y}=F_{t-1}(\mathbf{x})} = -\left(\frac{\partial L(y_1, \hat{y}_1)}{\partial \hat{y}_1}\bigg|_{\hat{y}_1=F_{t-1}(x_1)}, \dots, \frac{\partial L(y_n, \hat{y}_n)}{\partial y_n}\bigg|_{\hat{y}_n=F_{t-1}(x_n)} \right) \tag{202}$$

[11] Derivation here

Note that the vector above is a n-vector, and when we use the MSE loss, then the gradient just simplifies to the residual.

(b) We use our weak learning algorithm to train a weak model f_t on the residual values $\mathbf{r}_t$.

(c) We update

$$F_t = F_{t-1} + \eta \cdot f_t \tag{203}$$

3. In the end, we have

$$F_t = F_0 + \eta f_1 + \eta f_2 + \ldots + \eta f_T \tag{204}$$

consisting of a bunch of weak learners to make a strong learner.

In a way, we can think of this as an optimization problem in $\mathbb{R}^n$ (not $\mathbb{R}^d$!). We can think of $\hat{\mathbf{y}}$ representing the actual function f, and we're really doing gradient descent on the "function space" $\mathbb{R}^n$ by updating F_t.

Algorithm 8.2 (Gradient Boosting)

Here is the full algorithm for brevity.

Require: Training data $\{(x_i, y_i)\}_{i=1}^n$ where $x_i \in \mathcal{X}$, $y_i \in \mathcal{Y}$
Require: Differentiable loss function $L(y, \hat{y})$
Require: Number of iterations T
Require: Learning rate η
Require: Weak learning algorithm $\mathcal{A}$

// Initialize model with optimal constant value
$F_0 = \text{argmin}_\gamma \sum_{i=1}^n L(y_i, \gamma)$
// For regression (MSE): $F_0 = \frac{1}{n}\sum_{i=1}^n y_i$
// For binary classification: $F_0 = \log(\frac{C_+}{C_-})$
for $t = 1$ to T **do**
    // Compute negative gradient vector
    for $i = 1$ to n **do**
        $r_{t,i} = -\frac{\partial L(y_i, \hat{y}_i)}{\partial \hat{y}_i}\big|_{\hat{y}_i = F_{t-1}(x_i)}$
    end for
    // Train weak learner on pseudo-residuals
    $f_t = \mathcal{A}(\{(x_i, r_{t,i})\}_{i=1}^n)$
    // Update model with scaled weak learner
    for $i = 1$ to n **do**
        $F_t(x_i) = F_{t-1}(x_i) + \eta \cdot f_t(x_i)$
    end for
end for
return Final model $F_T(x) = F_0(x) + \eta \sum_{t=1}^T f_t(x)$
// Special cases for common loss functions:
// For MSE: $r_{t,i} = y_i - F_{t-1}(x_i)$ (actual residual)
// For LogLoss: $r_{t,i} = y_i - \sigma(F_{t-1}(x_i))$ where σ is sigmoid

---

Example 8.2 (Regression Trees)

If we have regression trees as our weak learners (pratically the max depth is 8 to 32 rather than stumps) with the L2 loss function.

1. The initial model will just constantly predict the average of the y_i's.
2. The r_t are just the pseudoresiduals

$$r_t = -\big(y_1 - f_{t-1}(x_1), \ldots, y_n - f_{t-1}(x_n)\big) \tag{205}$$

3. In case where there are multiple samples running to the same leaf node, the predicted values of the terminal nodes are the average of the y's of those samples.

Example 8.3 (Gradient Boosting Classification)

If we have classification trees as our weak learners, then

1. The initial model will just constantly predict the log odds $\log(C_+/C_-)$, where $C_\pm$ is the number of ones and zeros in the whole dataset. For multiclass there is probably a softmax variant of this.
2. In case where there are multiple samples running to the same leaf node, the predicted values of the terminal nodes are decided by majority.

The general ideas are pretty much the same between AdaBoost and gradient boost. We iteratively build a strong learner from weak learners. A few differences, however,

1. AdaBoost dynamically weighs the importance of each weak model, while gradient boost weak learners are equally weighted by η.
2. AdaBoost actively focuses on the samples where the previous weak learner got wrong, but gradient boost reduces the whole loss in general.
3. Gradient boost usually uses trees larger than stumps.

8.3.3 XGBoost

The final mainstream boosting algorithm is XGBoost. In regression, XGBoost fits to the residuals just like gradient boosting, but it uses unique regression trees. It is designed for large, complex datasets.

Definition 8.10 (XGBoost for Regression)

Let us have the same data $\{(x_i, y_i)\} \in \mathcal{X} \times \mathcal{Y}$ and the MSE loss

$$L(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{2}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{206}$$

with a constant stepsize ε (by default 3).

1. We first initialize the model with a constant value that minimizes the loss, which is just the average. So we have a single leaf as in our decision tree ensemble.

$$F_0 = \bar{y} \tag{207}$$

2. For $t = 1, \dots, T$, we do the following.
   (a) We have the predicted values $F_{t-1}(x_i)$ for each sample. We first compute the residuals, denoted $\mathbf{r}_0$. To build our next tree, we start off with a single node "containing" this set of residuals representing each data point.
   (b) We want to grow the decision tree, and we do this by splitting on the maximum gain in *similarity score*, defined for a collection of residuals $\mathbf{r}$ to be

$$s(\mathbf{r}) = \frac{\sum r_i}{\dim(\mathbf{r}) + \lambda} \tag{208}$$

   This score determines how well the set is clustered, and we would like well clustered residuals to be close together.λ is a regularization parameter used to decrease the score's sensitivity when splitting. Therefore, we first compute the score for the root node, and let us define the score of a tree as the sum of the scores of all its leaves. We want to split

greedily on this metric. We can keep on splitting until it reaches a certain number of levels (6), and then we can prune it based on whether the increase in score surpasses a threshold, called the *gain*. Note that as λ increases, it is easier to prune the tree.

(c) With our trained tree f_t, we add it to our cluster to iteratively build our final predictor.

$$F_t = F_{t-1} + \varepsilon \cdot f_t \tag{209}$$

9 Direct Clustering and Density Estimation

9.1 K Means Clustering

The simplest type of unsupervised learning is **clustering**. In the clustering problem, we are given a training set of *unlabeled* data

$$\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(n)}\} \tag{210}$$

and want to group the data into a few cohesive "clusters."

1. Determine the number of clusters that we want to find and set it as k (this can be a disadvantage if we do not know how many clusters we are looking for beforehand).
2. We initialize the **cluster centroids** $\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \dots, \boldsymbol{\mu}_k \in \mathbb{R}^d$ randomly or by some other method.
3. The next part takes the centroids and moves them to the center of each cluster accordingly. The following two steps are repeated until convergence (and convergence is guaranteed):
   (a) We assign each training sample $\mathbf{x}^{(i)}$ to the closest cluster centroid $\boldsymbol{\mu}_j$. That is, for every $i = 1, \dots, n$, set
   $$c^{(i)} \equiv \arg\min_j ||\mathbf{x}^{(i)} - \boldsymbol{\mu}_j||^2 \tag{211}$$
   where this argmin function returns the input to a function that yields the minimum (in this case, the number j that yields the minimum value of $||\mathbf{x}^{(i)} - \boldsymbol{\mu}_j||^2$ for each i).
   (b) We move each training cluster centroid $\boldsymbol{\mu}_j$ to the mean of the points assigned to it. That is, for each $j = 1, \dots, k$, set
   $$\boldsymbol{\mu}_j \equiv \frac{\sum_{i=1}^n 1\{c^{(i)} = j\}\, \mathbf{x}^{(i)}}{\sum_{i=1}^n 1\{c^{(i)} = j\}} \tag{212}$$

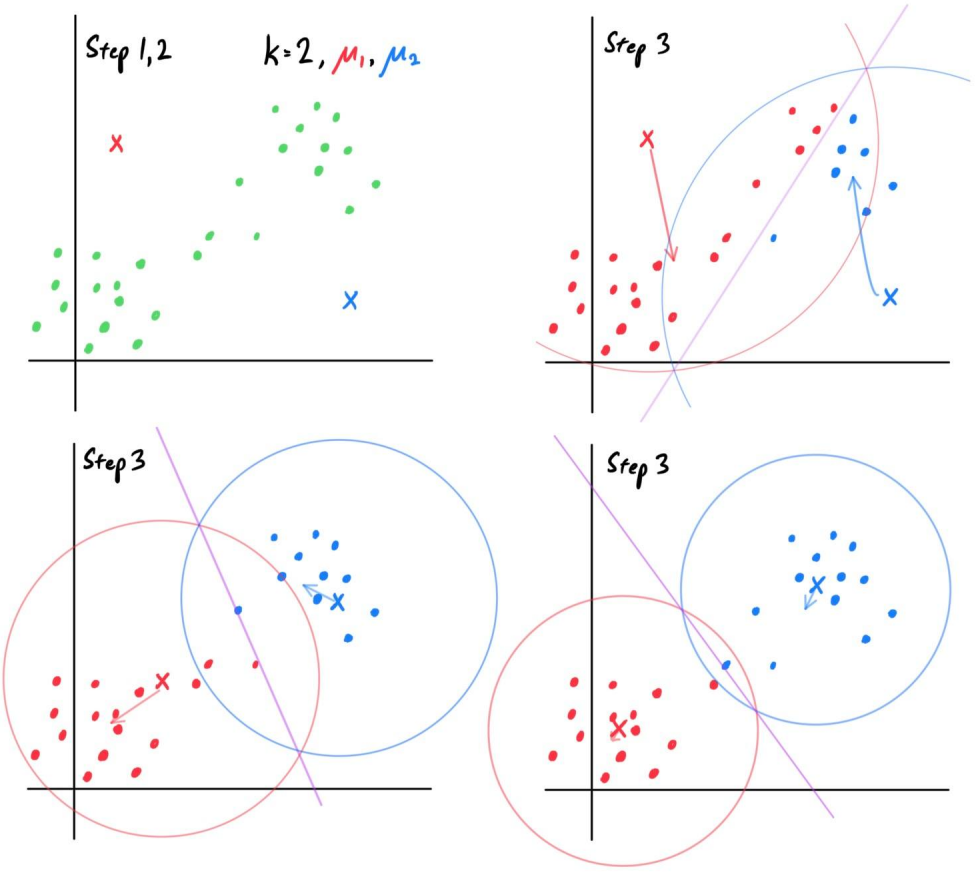


Figure 23: The steps can be visualized for a set of unlabeled data (green points) in $\mathbb{R}^2$ clustered into $k = 2$ groups (red and blue). The crosses represent the cluster centroids.

We can interpret this algorithm in another equivalent way. k-means is precisely coordinate descent on the cost function called the **distortion function**:

$$L(\boldsymbol{\mu}_1, \dots, \boldsymbol{\mu}_k) \equiv \sum_{i=1}^{n} \min_k ||\mathbf{x}^{(i)} - \boldsymbol{\mu}_k||^2 \tag{213}$$

but since L is not necessarily convex, it might be susceptible to local extrema.

9.2 Kernel Density Estimation

10 Direct Dimensionality Reduction

Dimensionality reduction is used for many purposes, such as preprocessing data, visualizing it, or encoding it in a sparser, more efficient way.

10.1 Principal Component Analysis

PCA finds low dimensional approximations to the data by projecting the data onto linear subspaces. To begin with some motivation, let a linear map $A : \mathbb{R}^D \to \mathbb{R}^D$ be full rank, which maps some set of n data points to the space of features. Then it is injective, and therefore for all data $x \in \mathbb{R}^D$ there exists a feature vector $z \in \mathbb{R}^D$ such that $z = Ax$. Generally, real-world data does not span the full space of D dimensions.[12] In fact, if we further assume that the data lies in a linear subspace, we want to compress it into a lower-dimensional vector such that the covariates in this lower dimensional space are also orthogonal, i.e. uncorrelated. We tackle both problems in 2 steps.

1. To compress this representation, we can take a data point $x \in \mathbb{R}^D$ and *approximate* it as a point $\hat{x} \in L_k$ for some k-dimensional subspace $L_k \subset \mathbb{R}^d$ (say that this is done with some function $P : \mathbb{R}^D \to L_k \subset \mathbb{R}^D$).
2. After this projection, we then want to extract the k features such that they are *orthogonal* (i.e. no correlation). This is done with a simple change of basis, which we denote $T : L_k \to \mathbb{R}^k$, giving us $\hat{z} = T\hat{x} = T(P(x))$. We can invert this map $T^{-1} : \mathbb{R}^k \to L_k$ to go from the orthogonalized compressed version $\hat{z}$ to the approximate full version $\hat{x}$.

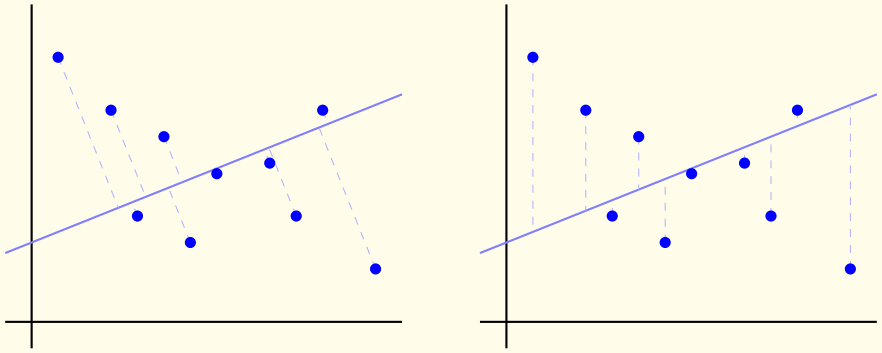
We are done! But which subspace do we choose? Let's formalize what the optimal subspace should be.

Definition 10.1 (Principal Subspace)

Let $X \in \mathbb{R}^d$ and $\mathcal{L}_k$ denote all k-dimensional linear subspaces. The **kth principal subspace** is defined as

$$\ell_k = \underset{\ell \in \mathcal{L}_k}{\operatorname{argmin}} \mathbb{E}\big(\min_{y \in \ell} ||\tilde{X} - y||^2 \big) \tag{214}$$

where $\tilde{X} = X - \mu$ and $\mu = \mathbb{E}[X]$. To parse this, let's fix a subspace ℓ. Then, the normalized data $\tilde{X}$ is a random vector and the minimum distance of $\tilde{X}$ onto the subspace ℓ is the inner min term. Taking the expectation of that gives us the expected distance of the data onto the subspace. The principal subspace is the subspace that minimizes this expected distance. The dimension reduced version of X is then $T_k(X) = \mu + \text{proj}_{\ell_k} X$.



(a) PCA minimizes the orthogonal distance to the subspace.

(b) Linear regression minimizes the residual distance to the subspace.

Figure 24: Note that this is in fact different from linear regression as it minimizes the expected *orthogonal distance* to the subspace, rather than the residual distance to the subspace as in linear regression.

[12] The *manifold hypothesis* that real-world data in high-dimensions actually lies on a lower-dimensional manifold.

We can see that by definition the properties of the principal subspace allows us to construct the best approximation of the points in a lower-dimensional subspace. This seems like a hard optimization problem, but it turns out that the theorem gives a simple solution. Note that we need to do 3 things:

1. Find such a subspace $L_k \subset \mathbb{R}^D$.
2. Find the projection $P_k : \mathbb{R}^D \to L_k \subset \mathbb{R}^D$. Note that by definition of the principal subspace P_k should be an *orthogonal* projection.
3. Find the bijection $T_k : L_k \to \mathbb{R}^k$.

It turns out that we can solve all three problems with the singular value decomposition.

Theorem 10.1 (Calculating the kth Principle Subspace)

Given our data matrix $X \in \mathbb{R}^{n \times d}$, let us normalize it to $\tilde{X} = X - \mu$ and then take the SVD of it.

$$\tilde{X} = U \Sigma V^T \tag{215}$$

where $U \in U(n) \subset \mathbb{R}^{n \times n}, V \in U(d) \subset \mathbb{R}^{d \times d}$ are orthogonal and $\Sigma \in \mathbb{R}^{n \times d}$ is diagonal that represents the singular values of X in decreasing order[a].

1. Then $\ell_k = \text{span}\{v_1, \dots, v_k\} \subset \mathbb{R}^d$, i.e. is the subspace spanned by the columns of V. By shifting it by μ, $\mu + \ell_k$ is the best affine subspace approximation of the x_i's.
2. The projection P is defined

$$\hat{x} = P_k(x) = \mu + \sum_{j=1}^{k} \langle x - \mu, v_j \rangle \, v_j = \sum_{j=1}^{k} \text{proj}_{v_j}(x - \mu) = \mu + \text{proj}_{\ell_k}(x - \mu) \tag{216}$$

   where we can rewrite it as the projection operator since the v_j's are orthonormal.
3. The change of basis T is defined with the mapping $v_j \in L_k \mapsto \sigma_j e_j \in \mathbb{R}^k$. Note that the v_j's form an orthogonal basis of L_k.

Note that as an immediate consequence, since T^{-1} maps e_j to $\sigma_j v_j$, we can write

$$e_j \xrightarrow{\Sigma_k} \sigma_j e_j \xrightarrow{V} \sigma_j v_j \tag{217}$$

where $V \in \mathbb{R}^{D \times D}$, and $\Sigma_k \in \mathbb{R}^{D \times k}$ is the submatrix of only the first k rows of Σ.

$$x = \mu + V_k^T \Sigma_k z \tag{218}$$

[a] We can make it decreasing by permuting the rows/columns of the unitary matrices U, V.

Proof.

We see that

$$X^T X = V \Sigma^T \Sigma V^T \tag{219}$$

Note that $X \neq V \Sigma^T$ in general. Now let $v_1, \dots, v_d$ be the columns of V. Then

$$X^T X [v_1, \dots, v_d] = X^T X V = V \Sigma^T \Sigma = [\sigma_1^2 v_1, \dots, \sigma_d^2 v_d] \tag{220}$$

Therefore, we can see that the way $X^T X$ acts on V That the v_i's are the eigenvectors of $X^T X$, with σ_i^2 the associated eigenvalues.

This theorem outlines the steps needed to do PCA. Now we state the algorithm.

Algorithm 10.1 (PCA Using SVD)

Given a dataset $X \in \mathbb{R}^{n \times d}$, let us denote the rows as x_i, and say that we are looking for a subspace of dimension k.

1. Compute the mean
$$\mu = \frac{1}{n} \sum_{i=1}^{n} x_i \in \mathbb{R}^d \tag{221}$$
2. Standardize the data $\tilde{X} = X - \mu$, i.e. $\tilde{x}_i = x_i - \mu$.
3. Compute the SVD $\tilde{X} = U \Sigma V^T$.
4. Compute the submatrices $V_k \in \mathbb{R}^{k \times k}$ and $\Sigma_k \in \mathbb{R}^{D \times k}$.
5. Define the projection operator $P_k(x) = \mu + \sum_{j=1}^{k} \langle x - \mu, v_j \rangle v_j$, the change of basis operator T, and the embedding operator $T^{-1}(z) = \mu + V_k \Sigma_k z$.

Algorithm 10.2 (Python Numpy Code)

Here is a Python snippet for reducing.

```
def PCA(X , num_components):
  X_meaned = X - np.mean(X , axis = 0)
  cov_mat = np.cov(X_meaned , rowvar = False)
  eigen_values , eigen_vectors = np.linalg.eigh(cov_mat)
  sorted_index = np.argsort(eigen_values)[::-1]
  sorted_eigenvalue = eigen_values[sorted_index]
  sorted_eigenvectors = eigen_vectors[:,sorted_index]
  eigenvector_subset = sorted_eigenvectors[:,0:num_components]
  X_reduced = np.dot(eigenvector_subset.transpose() , X_meaned.transpose() ).transpose()
  return X_reduced
```

Now a question arises: how do we know that this sample decomposition is a good approximation to the true decomposition? It comes from the fact that the sample covariance $\hat{\Sigma}$ is a good approximation of the true covariance Σ, which we will later prove using concentration of measure.

Theorem 10.2 (Risk)

The risk satisfies
$$R(k) = \mathbb{E}[||X - P_k(X)||^2] = \sum_{j=k+1}^{D} \lambda_j \tag{222}$$

It is essential that you plot the spectrum in decreasing order. This allows you to analze how well PCA is working. People often use the "elbow" technique to determine where to choose K, and we value
$$\frac{\sum_{j=1}^{k} \lambda_j}{\sum_{j=1}^{d} \lambda_j} \tag{223}$$
accounts for the **variance explained**, which should be high with K low. If you have to go out to dimension $K = 50$ to explain 90% of the variance, then PCA is not working. It may not work because of many reasons, such as there being nonlinear structure within the data.

It turns out that the elements of $\hat{\Sigma}$ are close entry-wise to those of Σ. But if this is true, then does it mean that the eigenvalues of the sample covariance matrix are close to the true eigenvalues of the covariance matrix? It turns out that the answer is no, and we need a proper metric to satisfy this assumption. The metric, as we can guess from linear algebra, is the operator norm, and we will show some results from matrix perturbation theory.

Definition 10.2 (Operator Norm)

The **operator norm** of a matrix A is defined as

$$||A|| = \sup_{x \neq 0} \frac{||Ax||}{||x||} \tag{224}$$

Lemma 10.1 ()

It turns out that

$$||\hat{\Sigma} - \Sigma|| = O_p\left(\frac{1}{\sqrt{n}}\right) \tag{225}$$

Theorem 10.3 (Weyl's Theorem)

If $\hat{\Sigma}$ and Σ are close in the operator norm, then their eigenvalues are close.

$$||\hat{\Sigma} - \Sigma|| = O_p\left(\frac{1}{\sqrt{n}}\right) \implies |\hat{\lambda}_j - \lambda_j| = O_p\left(\frac{1}{\sqrt{n}}\right) \tag{226}$$

This only talks about their eigenvalues, but this does not necessarily imply that the eigenvalues are close. We need an extra condition.

Theorem 10.4 (David-Kahan Theorem)

If $\hat{\Sigma}$ and Σ are close in the operator norm, and if the eigenvectors of Σ are well-conditioned, then the eigenvectors of $\hat{\Sigma}$ are close to the eigenvectors of Σ. More specifically,

$$||\hat{v}_j - v_j|| \leq \frac{2^{3/2}||\hat{\Sigma} - \Sigma||}{\lambda_j - \lambda_{j+1}} \tag{227}$$

10.1.1 Kernel PCA

Definition 10.3 (Kernel PCA)

Let N_i be the neighborhood around X_i. Then, we want to find a mapping $W : \mathbb{R}^n \to \mathbb{R}^k$ that minimizes

$$\min_W \sum_{i=1}^n \left|\left| X_i - \sum_{j \in N_i} W_{ij} X_j \right|\right|^2 \text{ where } \sum_j W_{ij} = 1 \tag{228}$$

We can constrain the weights in W so that anything that is not in the neighborhoods are 0.

10.2 Multi-Dimensional Scaling

Again, we want to reduce our dimension, but the goal is slightly different from PCA.

Definition 10.4 (Multi-Dimensional Scaling)

Given our data $X \in \mathbb{R}^d$, we want to construct a linear map $T : \mathbb{R}^d \to \mathbb{R}^k$ such that it preserves the pairwise differences between the data points. That is, we want to minimize the following loss function

$$\min_T \sum_{i \neq j} \left(d_{\mathbb{R}^k}(T(x_i), T(x_j)) - d_{\mathbb{R}^d}(x_i, x_j) \right) \tag{229}$$

where d_V is a distance metric in the space V.

Note that we can easily modify this formulation to preserve other structures, such as dot products, weights between distances, or different types of metrics in each space. It turns out that when the distance metric is the Euclidean L2 distance, then the solution to this linear map turns out to be PCA. This may be a more intuitive way to think about PCA, since we're trying to preserve the pairwise distances between the data points.

Theorem 10.5 (Equivalence of Classical MDS and PCA)

If the distance metric is the Euclidean L2 distance, then the solution to the MDS problem is equivalent to PCA. That is,

$$T_k = \operatorname*{argmin}_T \sum_{i \neq j} \left(||T(x_i) - T(x_j)||^2 - ||x_i - x_j||^2 \right) \tag{230}$$

Generally, if you don't use classical MDS, then you will get a different answer than PCA and there doesn't exist a closed form solution, so you'll have to minimize this numerically.

Example 10.1 (Non Classical MDS)

The loss

$$\sum_{i \neq j} \left(||T(x_i) - T(x_j)|| - ||x_i - x_j|| \right)^2 \tag{231}$$

does not give the same solution as PCA.

10.3 Isomap

Isomap is a bit different in the way that it tries to capture more of the global structure of the data, which brings advantages and disadvantages. It is simply a modification of MDS but with geodesic distances.

Definition 10.5 (Isomap)

You start off with the point cloud, but with every point, X_i, you find the local neighborhood N_i and you make a weighted graph over the whole dataset in the high dimensional space. Then, the distance between any two arbitrary points is the weighted sum of the path between them, calculated by Dijkstra's algorithm. Intuitively, this is an approximation of the geodesic distance between these two points on a manifold. Call this distance d_G. Then, we simply do MDS by minimizing

$$\min_T \sum_{i \neq j} \left(d_{\mathbb{R}^k}(T(x_i), T(x_j)) - d_G(x_i, x_j) \right) \tag{232}$$

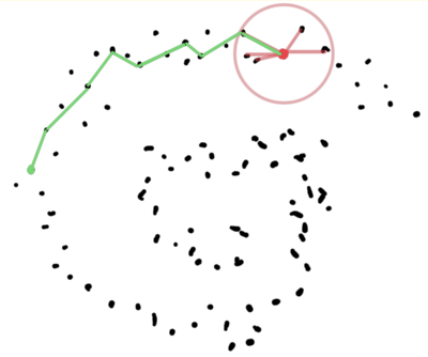


Figure 25: The classical example is the spiral manifold. The data lies in this manifold, and the geodesic distance helps us gain an accurate distance metric within this data.

The problem with this is that it is very sensitive to noise. For example, if we had a few points lying between the spirals, then the geodesic distance between the two spirals would be very small, and so the MDS would try to bring them closer together.

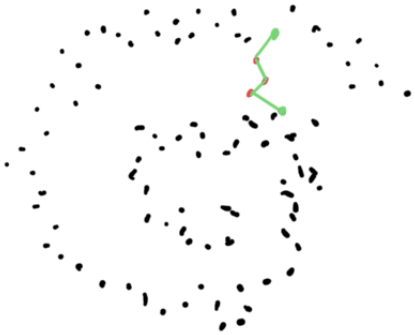


Figure 26: With extra noisy points (red), the geodesic distance can get corrupted.

To fix this, we use *diffusion maps*, which looks at all possible paths between two points and looks at some average of them, which increases robustness.

10.4 Local Linear Embedding

PCA and MDS are linear embedding methods. Let's move onto nonlinear ones. The first nonlinear models that we work with again use the idea of locality (remember kernel regression). You have data that is globally nonlinear, but if you look at a point and its local neighborhood around it, then it is approximately linear since we assume that it lives in some smooth manifold.

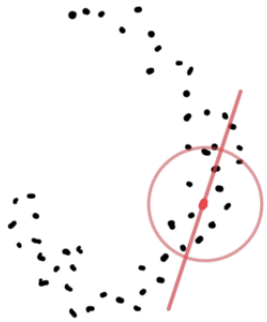


Figure 27: Local linear embedding assumes that the data is locally linear.

The concept of neighborhood can be defined in two ways. You can either just fix an ϵ and take the ϵ-ball around each point x_i. Or you can fix a k and take the k nearest neighbors of each point. The general idea of using kernel PCA is to take a local neighborhood of the data and construct some linear approximation of it.

10.5 UMAP

10.6 t-SNE

11 Linear Latent Variable Models

Note that in PCA, we have taken some data x in high-dimension D and reduced it to a lower-dimensional orthogonal representation in $\mathbb{R}^k$. In other words, the corresponding z represents x in another space, which we call a **latent space**. This concept of representing data from one space to another are called **factor analysis**, or **latent variable analysis**, and is an extremely important concept for state-of-the-art models.

Say that we want to do density estimation for the probability distribution of the covariates X, a random variable. We can try to model it directly, but this may be infeasible. Rather, what we do is "add" a latent distribution Z, creating the joint distribution (X, Z). This may look more complicated, but make two simplifying assumptions: we hope that we can model the latent Z in a simple form (like a Gaussian), and we can model the conditional probability $p(x \mid z)$ as some function f_θ parameterized by θ. We can therefore marginalize and find that

$$p_X(x) = \int_{z \in \mathbb{R}^k} p(x \mid z)\, p_Z(z)\, dz = \mathbb{E}_Z[p(X \mid Z)] \tag{233}$$

Like we do with everything else in math, we take a look at the simplest example: linear functions. In **linear factor models**, we start with the unknown covariate distribution $X \in \mathbb{R}^d$, and we create a latent variable $Z \in \mathbb{R}^k$ with some simple assumed distribution $Z \sim p_Z(z)$.[13] For general latent variable models, we have a **latent transform** $g : \mathbb{R}^k \to \mathbb{R}^d$ that links the two distributions together as $X = g(Z)$. Then, we assume that

$$X = \mu + WZ + \epsilon \tag{234}$$

where the noise ϵ is typically Gaussian and diagonal (but not necessarily the same component-wise variances). Finally, we can use techniques like MLE to estimate W, μ, and the parameters of ϵ.

The entire reason we want to do this is that we are hoping that we can construct a complex distribution X from a simple distribution Z with $d >> k$, connected by some well-studied function $X = g(Z)$. In the linear case, $W \in \mathbb{R}^{d \times k}$, and the latent variables z give a more compact, parsimonious explanation of dependencies between the components of the observations x.

Definition 11.1 (Factor Analysis)

Factor analysis is a specific case of a linear factor model where

$$X = \mu + WZ + \epsilon, \text{ where } z \in \mathcal{N}(0, I), \epsilon \sim \mathcal{N}\big(0, \text{diag}(\sigma_1^2, \dots, \sigma_k^2)\big) \tag{235}$$

It should be clear to us that X should be Gaussian[a] and that $\mathbb{E}[X] = \mu$, with

$$\begin{aligned}
\text{Var}[X] &= \mathbb{E}[(X - \mu)(X - \mu)^T] && (236)\\
&= \mathbb{E}[(WZ + \epsilon)(Z^T W^T + \epsilon^T)] && (237)\\
&= \mathbb{E}[WzzW^T] + \mathbb{E}[\epsilon\epsilon^T] && (238)\\
&= W\mathbb{E}[zz^T]W^T + \mathbb{E}[\epsilon\epsilon^T] && (239)\\
&= WW^T + \text{diag}(\sigma_1^2, \dots, \sigma_d^2) && (240)
\end{aligned}$$

The W, μ, and σ_i's can be estimated using MLE methods. Unfortunately, no closed form exists, so iterative methods are commonly applied.

[a] Since linear transformations of Gaussians are Gaussian

11.1 Probabilistic PCA

We want to take PCA and extend it to be a *generative model*, which allows you to sample data. In regular PCA, we saw that for some $z \in \mathbb{R}^k$ in the latent space, $\hat{x} = \mu + V_k \Sigma_k z$. Therefore, if we just change z from

[13] Usually, we also constrain it to be factorable (i.e. is the product of its marginal distributions: $p(z) = \prod_i p(z_i)$) so that it is easy to sample from. Occasionally, the stronger assumption of the z_i's being iid is made.

a point to a probability distribution (e.g. Gaussian), we can take a random variable $Z \sim \mathcal{N}(0, I)$ from $\mathbb{R}^k$, and then transform it to get a random variable $X = \mu + V_k \Sigma_k Z$, which will give a density.

$$X \sim \mathcal{N}\big(\mu, V_k \Sigma_k (V_k \Sigma_k)^T\big) = \mathcal{N}\big(\mu, X_k^T X_k\big) \tag{241}$$

Note that in here, X is a random variable that we are trying to fit to the data X_k. However, $X_k \in \mathbb{R}^{k \times n}$ with $k << n$, and so $X_k^T X_k \in \mathbb{R}^{n \times n}$ is not full rank, and so the distribution is restricted to strictly the k-dimensional subspace $L_k \subset \mathbb{R}^D$. We want to add a bit of noise beyond the subspace, so we add an extra small Gaussian ϵ around it. In general factor analysis above, we set ϵ to have an arbitrary diagonal Gaussian, but we just use an isotropic one $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$, giving us

$$X = \mu + V_k \Sigma_k Z + \epsilon \implies X \sim \mathcal{N}\big(\mu, X_k X_k^T + \sigma^2 I\big) \tag{242}$$

Note that PPCA is really a specific instance of factor analysis, and we assume that the latent variable z follows a standard Gaussian $\mathcal{N}(0, 1)$. It turns out that this extension has the flaw that the subspace generated by the MLE estimate of X will not necessarily correspond to the principal subspace of the data. But we can make this happen with one more assumption.

Definition 11.2 (Probabilistic PCA)

The **probabilistic PCA** model is a latent factor model of form

$$x \sim \mathcal{N}(\mu, WW^T + \sigma^2 I) \tag{243}$$

with the latent transform

$$X = g_{\mu, W, \sigma}(Z) = \mu + (WW^T + \sigma^2 I)^{1/2} Z \tag{244}$$

Optimizing this model is actually quite easy.

Theorem 11.1 (MLE of PPCA Model)

Given $X^{(i)} \sim X$ iid, the MLEs for W, μ, σ are

$$\mu_{MLE} = \frac{1}{N} \sum_{i=1}^{N} X^{(i)} \implies \hat{\mu}_{MLE} = \frac{1}{N} \sum_{i=1}^{N} x^{(i)} \tag{245}$$

$$\hat{\sigma}^2_{MLE} = \frac{1}{d-k} \sum_{j=k+1}^{d} \lambda_j \tag{246}$$

$$W_{MLE} = U_q (\Lambda_d - \hat{\sigma}^2_{MLE} I_d)^{1/2} R \tag{247}$$

Proof.

Given $X^{(i)} \sim X$ iid, the MLEs for W, μ, σ have a closed form, and model parameter estimation can be performed iteratively and efficiently. We have

$$\mu_{MLE} = \frac{1}{N} \sum_{i=1}^{N} X^{(i)} \implies \hat{\mu}_{MLE} = \frac{1}{N} \sum_{i=1}^{N} x^{(i)} \tag{248}$$

and setting the biased MLE estimator of the variance,

$$\widehat{\text{Var}}_{MLE}(\mu_{MLE}) = S = \frac{1}{N} \sum_{i=1}^{N} (X^{(i)} - \mu_{MLE})(X^{(i)} - \mu_{MLE})^T \tag{249}$$

we can derive the MLE of W.[a] We can find the MLE estimate of σ first by taking a look at

$C = \text{Var}[X] = WW^T + \sigma^2 I$. It is the sum of positive semidefinite patrices that are also symmetric, so by the spectral theorem it is diagonalizable and has full rank d. But WW^T is rank k, so $d-k$ of the eigenvalues of WW^T is 0, indicating that the same $d-k$ smallest eigenvalues of C is σ^2. Therefore, we can take the smallest $d-k$ eigenvalues of our MLE estimator of C, which is S, and average them to get our MLE for σ.

$$\hat{\sigma}^2_{MLE} = \frac{1}{d-k} \sum_{j=k+1}^{d} \lambda_j \tag{250}$$

We can approximate $WW^T = C - \sigma^2 I \approx S - \hat{\sigma}^2_{MLE} I$, and by further taking the eigendecomposition $C = U\Sigma U^T \implies WW^T = U(\Sigma - \sigma^2 I)U^T$ and cutting off the last $d-k$ smallest eigenvalues and their corresponding eigenvectors, we can get

$$W_{MLE} = U_q(\Lambda_d - \hat{\sigma}^2_{MLE} I_d)^{1/2} R \tag{251}$$

where the R just accounts for any unitary matrix.

[a] Note that W_{MLE} is not unique. Say that W^* is an MLE, then, for any unitary $U \in \mathbb{R}^{k \times k}$, we have $W^* W^{*T} = (W^*U)(W^*U)^T$.

Now as $\sigma \to 0$, the density model defined by PPCA becomes very sharp around these d dimensions spanned by the columns of W. At 0, our MLE of W is simplified and we have

$$X = W_{MLE} z + \mu_{MLE} + \epsilon = U_q \Lambda_q^{1/2} z + \mu_{MLE} \tag{252}$$

which essentially reduces to regular PCA. That is, the conditional expected value of z given X becomes an orthogonal projection of $X - \mu$ onto the subspace spanned by the columns of W. Intuitively, we can see that we are estimating the Gaussian, which corresponds to the mean squared distance from each $x^{(i)}$ to ℓ_k.

11.2 Linear Independent Component Analysis

In our setting, let us just assume that $\mu = 0$ and $\epsilon = 0$.

Definition 11.3 (Linear ICA)

In **linear ICA**, we have the simple model.

$$x = Wz \tag{253}$$

In here, $X \in \mathbb{R}^d$ is a mixture vector and $W \in \mathbb{R}^{d \times k}$ is a **mixing matrix**. Both W and z are unknown, and we need to recover them given x. We have 2 strong assumptions.

1. Each component of z is independent (not just uncorrelated). This is an easy enough assumption to intuit.
2. Independent components of z must *not* be Gaussian.[a]

[a] This is needed for us to be able to "unmix" the signals. To see why, just suppose z was Gaussian, and so the vector Rz is also Gaussian for any invertible R. Therefore, we could find an infinite number of solutions of form $x = WR^{-1}Rz$ and have no way to separate them.

There are further ambiguities with ICA.

1. Estimating the latent components up to a scaling factor.

$$x = (\alpha W)(\frac{1}{\alpha} z) \text{ for some } \alpha > 0 \tag{254}$$

We can fix this by forcing $\mathbb{E}[z_i^2] = 1$. However, there is still an ambiguity for the sign of hidden components, but this is insignificant in most applications.

2. Estimating the components up to permutation. We have

$$x = WP^{-1}Pz \tag{255}$$

for some permutation matrix P.

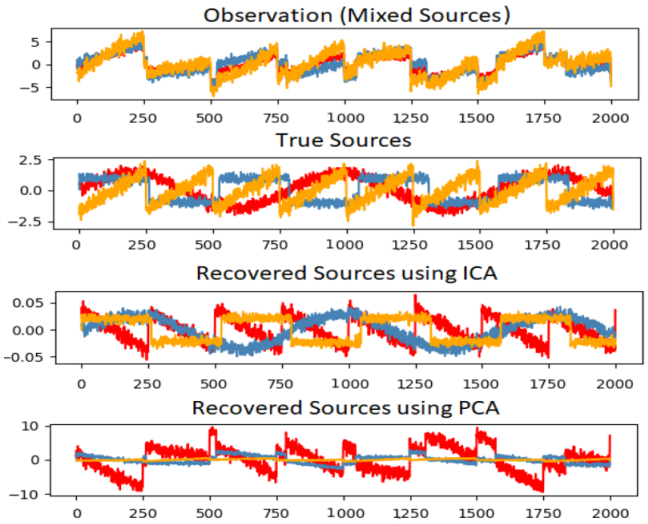
Now let's see how linear ICA actually estimates W and z. Once W is estimated, the latent components of a given test mixture vector, x^* is computed by $z^* = W^{-1}x^*$. So now all there's left to do is to estimate W, which we want to estimate so that $W^{-1}x$ is far from Gaussian. The reason for this is that given a bunch of independent non-Gaussian h_i's, if we mix them with a matrix that is not $\pm I$, then by CLT, a linear combination of random variables will tend to be Gaussian, and so for an arbitrary W we would expect x to be Gaussian. Therefore, what we want to do is guess some matrix A, and compute

$$Ax = AWh \tag{256}$$

and if we get things right, $A \approx W^{-1}$, and the result of Ax would look pretty non-Gaussian. If it it not the case, then AW will still be some mixing matrix, and so Ax would look Gaussian. So now the question reduces to how do we choose this A? There are multiple ways to measure non-Gaussianity:

1. The absolute or squared kurtosis, which is 0 for Gaussians. This is a differentiable function w.r.t. W, so we can try maximizing it. This is done for the sample kurtosis, of course.
2. Another measure is by maximizing the neg-entropy.

We can perform this on three mixed signals with additive noise, and ICA does very well, though again some recovered signals are scaled or permuted weirdly.



11.3 Slow Feature Analysis

Slow feature analysis also another special case of a linear factor model that uses information from time signals to learn invariant features. It is motivated by a general principle called the **slowness principle**. The idea is that the important characteristics of scenes change very slowly compared to the individual measurements that make up a description of a scene. For example, in computer vision, individual pixels can change very rapidly. If a zebra moves from left to right across the image, an individual pixel wil rapidly change from black to white. By comparison, the feature indicating whether a zebra is in the image will not change at all, and the feature describing the zebra's position will change slowly. Therefore, we want to regularize our model to learn features that change slowly over time.

We can apply the slowness principle to any differentiable model trained with gradient descent. That is, we

can add the following term to the loss function:

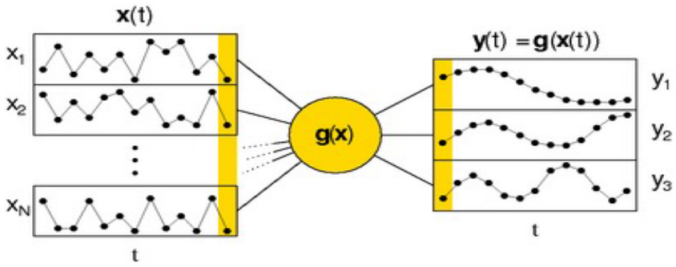
$$\lambda \sum_i d\big(f(x^{(t+1)}), f(x^{(t)})\big) \tag{257}$$

where λ is a hyperparameter determining the strength of the slowness regularization term, t is the time index, f is the feature extractor to be regularized, and d is the distance between $f(x^{(t)})$ and $f(x^{(t+1)})$. A common choice for d is the mean squared difference.

Essentially, given a set of time-varying input signals $x^{(t)}$, SFA learns a nonlinear function f that transforms x into slowly-varying output signals y. Obviously, we can't just take some trivial function like $f = 0$, so we have the following constraints

$$\mathbb{E}_t[f(x^{(t)})_i] = 0 \tag{258}$$

$$\mathbb{E}_t[f(x^{(t)})_i^2] = 1 \tag{259}$$



We can restrict the nonlinear f to some subspace of functions, and this becomes a standard optimization problem where we solve

$$\min_\theta \mathbb{E}_t\big[\big(f(x^{(t+1)})_i - f(x^{(t)})_i\big)^2\big] \tag{260}$$

11.4 Latent Dirichlet Allocation

11.5 Sparse Dictionary Learning

Latent variables can help us represent data in lower dimensions, but another advantage is that we can get *sparse* representations as well. What we want to do in sparse coding is that for each input $x^{(i)}$, we want to find a latent representation $z^{(i)}$ such that it is sparse (i.e. has many 0s) and also we can reconstruct the original input $x^{(i)}$ well. We have basically two things to optimize: the latent representations z and the decoding mechanism, which we can do with a *dictionary matrix* D. Note that we are optimizing for *both* the latent encodings and the decoding mechanism, and so this isn't a generative model.

Definition 11.4 (Sparse Dictionary Encoding Model)

The **sparse dictionary encoding model** is a representation model defined

$$X = g_D(Z) = DZ \tag{261}$$

where $D \in \mathbb{R}^{d \times k}$ is a **dictionary matrix** that decodes the latent $Z \in \mathbb{R}^k$ to $X \in \mathbb{R}^d$. Note that both the $z^{(i)}$'s and D are optimized, so we want to perform the *joint* optimization[a]

$$\min_D \frac{1}{N} \sum_{i=1}^N \min_{z^{(i)}} \underbrace{\frac{1}{2}||x^{(i)} - Dz^{(i)}||_2^2}_{\text{reconstruction error}} + \underbrace{\lambda||z^{(i)}||_1}_{\text{sparsity penalty}} \tag{262}$$

[a] To break this term down, let's just assume that we have a fixed dictionary D. Then, we just need to minimize with respect to each $h^{(t)}$. Now we can add the dictionary parameter back again.

Note that the reconstruction, or decoding, of $x = Dz$ is linear and explicit, but if we want to encode $x \mapsto z$, we need to substitute the x into the term above and minimize it w.r.t. D and z to solve it. Therefore, this encoder is an implicit and *nonlinear* function of x.

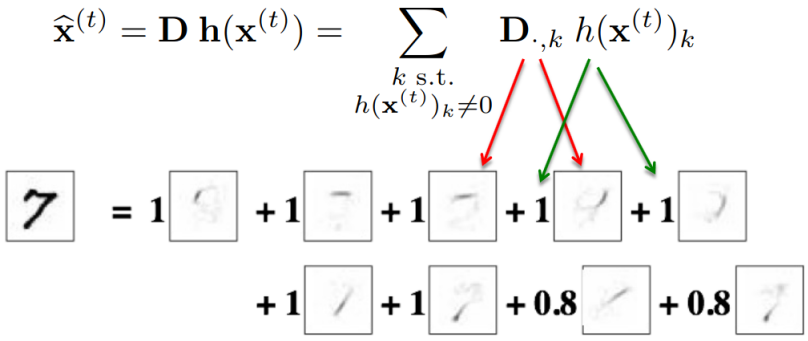


Figure 28: We can reconstruct an image of a seven as a linear combination of a set of images. Note that each of the images of strokes are columns of W and the coefficients make up the sparse vector h.

Let's think about how we can optimize the objective function w.r.t. h, keeping D constant. We can do stochastic gradient descent, which gives us the steps

$$\nabla_{h^{(t)}} \mathcal{L}(x^{(t)}) = D^T(Dh^{(t)} - x^{(t)}) + \lambda \operatorname{sign}(h^{(t)}) \tag{263}$$

but this wouldn't achieve sparsity since it overshoots the 0 all the time. Therefore, we can clip it, or we can use proximal gradient descent/ISTA to take a step, and shrink the parameters according to the L1 norm.

$$h^{(t)} = h^{(t)} - \alpha D^T(Dh^{(t)} - x^{(t)}) \tag{264}$$

$$h^{(t)} = \operatorname{shrink}(h^{(t)}, \alpha\lambda) \tag{265}$$

where $\operatorname{shrink}(a, b) = [\ldots, \operatorname{sign}(a_i)\ \max(|a_i| - b_i, 0), \ldots]$. This is guaranteed to converge if $1/\alpha$ is bigger than the largest eigenvalue of $D^T D$.

12 Graphical Models

The concept of using latent variables to model some process will be used over and over again. We have seen simple examples of latent linear models, but what about nonlinear ones? It turns out that these can be seen as a specific instance of *graphical models*.

When computing high-dimensional distributions, the parameters needed to encode this density scales badly. We can see that a general Gaussian mixture model in $\mathbb{R}^n$ with k clusters requires $O(n^2k)$ parameters. If we wanted to sample from a distribution of portraits, then the dimension n would be the resolution of the image. For a 1024×1024 image, this requires $n = 3 \cdot 2^{20}$ dimensions, and modeling it with a GMM is hopeless. Fortunately, for complex distributions there is usually some dependencies (e.g. between neighboring pixels) that we can take advantage of. This is exactly what graphical models do. They factor complex distributions so that the scaling is much better. While there are graphical models that do not use latent variables, most interesting applications of graphical models require latent variables, and so we will focus on that. Additionally, we will introduce the EM algorithm, which will be used repeatedly and is particularly important in optimizing *variational autoencoders* in deep learning.

12.1 Gaussian Mixture Models and EM Algorithm

Given a training set $\mathbf{x}^{(i)}{}_{i=1}^{n}$ (without the y-labels and so in the unsupervised setting), there are some cases where it may seem like we can fit multiple Gaussian distributions in the input space $\mathcal{X}$. For example, the points below seem like they can be fitted well with 3 Gaussians.

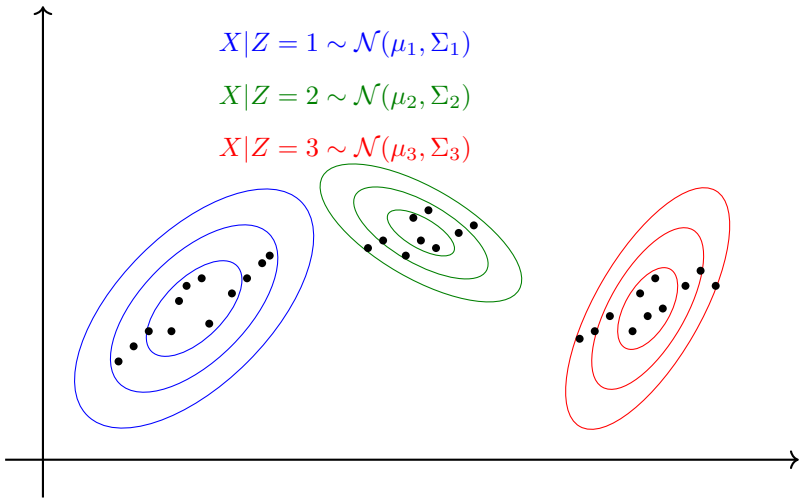


Figure 29: Example of data that can be fitted with 3 Gaussians

Therefore, we can construct a best-fit model as a composition of a multinomial distribution (to decide which one of the Gaussians $\mathbf{x}$ should follow) followed by a Gaussian. That is, to find the distribution of $\mathbf{x}$ and get the density function $p(\mathbf{x})$, we condition it on the random variable Z. More specifically, we let

$$Z \sim \text{Multinomial}(\boldsymbol{\phi}), \qquad \boldsymbol{\phi} = \begin{pmatrix} \phi_1 & \phi_2 & \dots & \phi_k \end{pmatrix} \text{ such that } \sum_{i=1}^{k} \phi_i = 1 \tag{266}$$

and define the conditional distributions as

$$\begin{aligned}
\mathbf{X} \mid Z = 1 &\sim \mathcal{N}(\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1) \\
\mathbf{X} \mid Z = 2 &\sim \mathcal{N}(\boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2) \\
&\dots \sim \dots \\
\mathbf{X} \mid Z = j &\sim \mathcal{N}(\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j) \\
&\dots \sim \dots \\
\mathbf{X} \mid Z = k &\sim \mathcal{N}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)
\end{aligned}$$

Therefore, our model says that each $\mathbf{x}^{(i)}$ was generated by randomly choosing $z^{(i)}$ from $1, \dots, k$ according to some multinomial, and then the $\mathbf{x}^{(i)}$ was drawn from one of the k Gaussians depending on $z^{(i)}$. This model is called the **mixture of Gaussians model**. The parameters of our model are:

- The vector $\boldsymbol{\phi} \in \mathbb{R}^k$ (which really has $k-1$ parameters) characterizing the multinomial distribution.
- The set of vectors $\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \dots, \boldsymbol{\mu}_k$ representing the mean vectors of each multivariate Gaussian. For simplicity, we'll denote this set of vectors as $\boldsymbol{\mu}$.
- The set of symmetric, positive-definite matrices $\boldsymbol{\Sigma}_1, \boldsymbol{\Sigma}_2, \dots, \boldsymbol{\Sigma}_k$ representing the covariance matrices of each multivariate Gaussian. For simplicity, we'll denote this set of matrices as $\boldsymbol{\Sigma}$.

We can write down the log-likelihood of the given data $\mathbf{x}^{(i)}$'s as a function of all the parameters above as:

$$\begin{aligned}
l(\boldsymbol{\phi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) &= \sum_{i=1}^{n} \log, p\big(\mathbf{x}^{(i)}; \boldsymbol{\phi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}\big) \\
&= \sum_{i=1}^{n} \log \left(\sum_{j=1}^{k} p\big(\mathbf{x}^{(i)} \mid z^{(i)} = j; \boldsymbol{\mu}, \boldsymbol{\Sigma}\big), p\big(z^{(i)} = j; \boldsymbol{\phi}\big) \right)
\end{aligned}$$

This equation above tells us the (log-) likelihood of the data landing on the $\mathbf{x}^{(i)}$'s given that we have parameters $\boldsymbol{\phi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}$. Note that since we only know that the *final* value of the ith sample is $\mathbf{x}^{(i)}$ and not anything at all about which value $z^{(i)}$ the ith sample had, there is an extra unknown in this model. That is, we do not know which one of the k Gaussians the $\mathbf{x}^{(i)}$ was generated from. These values $z^{(i)}$ are called the **hidden/latent variables**.

If we did know the values of the hidden variables $z^{(i)}$ (i.e. if we knew which of the k Gaussians each $\mathbf{x}^{(i)}$ was generated from), then our log likelihood function would be much more simple since now, our givens will be both $\mathbf{x}^{(i)}$ *and* $z^{(i)}$. Therefore, we don't have to condition on the $z^{(i)}$ and can directly calculate the log of the probability of us having sample values $(z^{(1)}, \mathbf{x}^{(1)}), (z^{(2)}, \mathbf{x}^{(2)}), \dots, (z^{(n)}, \mathbf{x}^{(n)})$.

$$\begin{aligned}
l(\boldsymbol{\phi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) &= \sum_{i=1}^{n} \log, p\big(\mathbf{x}^{(i)}; \boldsymbol{\phi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}\big) \\
&= \sum_{i=1}^{n} \log, p\big(\mathbf{x}^{(i)}, z^{(i)}; \boldsymbol{\phi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}\big) \\
&= \sum_{i=1}^{n} \log, p\big(\mathbf{x}^{(i)} \mid z^{(i)}; \boldsymbol{\mu}, \boldsymbol{\Sigma}\big)\, p\big(z^{(i)}; \boldsymbol{\phi}\big)
\end{aligned}$$

This model, with known $z^{(i)}$'s, is basically the GDA model, which is easy to calculate. That is, the maximum values of $\boldsymbol{\phi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}$ are:

$$\phi_j = \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}_{z^{(i)}=j}$$
$$\boldsymbol{\mu}_j = \frac{\sum_{i=1}^{n} \mathbb{1}_{z^{(i)}=j}\mathbf{x}^{(i)}}{\sum_{i=1}^{n} \mathbb{1}_{z^{(i)}=j}}$$
$$\boldsymbol{\Sigma}_j = \frac{1}{\sum_{i=1}^{n} \mathbb{1}_{z^{(i)}=j}} \sum_{i=1}^{n} \mathbb{1}_{z^{(i)}} \left(\mathbf{x}^{(i)} - \boldsymbol{\mu}_j\right), \left(\mathbf{x}^{(i)} - \boldsymbol{\mu}_j\right)^T$$

for $j = 1, \dots, d$. But since we do *not* know the values of $z^{(i)}$, we first try to "guess" the values of the $z^{(i)}$'s and then update the parameters of our model assuming our guesses are correct. Let us clarify some notation:

- The distribution that we will iteratively reassign over and over again is Z, with density $p_Z(z)$ that maps $z \mapsto \phi_z$, where $\boldsymbol{\phi}$ is a vector that represents the density. The algorithm will initialize p_Z and have it converge to the true multinomial density. Note that Z in this context could represent the true multinomial distribution Z or could represent the distributions iteratively produced by the algorithm that should converge onto the true Z (usually the latter).
- The k Gaussian distributions that we will iteratively reassign over and over again is $\mathcal{N}_1, \mathcal{N}_2, \dots, \mathcal{N}k$, with densities $p_{\mathcal{N}1}(\mathbf{x}), \dots, p_{\mathcal{N}k}(\mathbf{x})$ that maps $\mathbf{x} \mapsto p_{\mathcal{N}j}(\mathbf{x})$.
- The distribution of the entire random variable $\mathbf{X}$ will have density $p_X(\mathbf{x})$. Since we are iteratively reassigning the densities p_Z and $p\mathcal{N}_j$, this joint distribution of $\mathbf{X}$ will also get modified.

Definition 12.1 (EM Algorithm)

The **Expectation-Maximization (EM) Algorithm** has the following steps:

1. We initialize our values of θ, which can be chosen randomly or by **K-means initialization** (not explained here).
   - We can randomly assign our values of μ_j's and the Σ_j's in $\mathbb{R}^d$.
   - We can randomly assign the density of our guess multinomial $p_Z(z)$, represented by vector
   $$\phi = \begin{pmatrix} \phi_1 \\ \vdots \\ \phi_k \end{pmatrix} \text{ with } \sum_{j=1}^{k} \phi_j = 1$$
   where $p_Z(z) \equiv \phi_z$ for $z = 1, \dots, k$.
2. **(E Step)** Now that we have our prior guess of what Z and its density function p_Z is, we can calculate its posterior density function by taking one observed example $x^{(i)}$ and modifying p_Z to $p_Z^{(i)}$. This superscript (i) on the distribution p_Z indicates that this is a posterior density created from observing $x^{(i)}$. (The motivation for this construction is explained more specifically in the next section involving Jensen's inequality.) Using Bayes' rule, we should calculate n density functions
   $$p_Z^{(i)}(z) \equiv p_Z(z \mid x^{(i)}; \phi, \mu, \boldsymbol{\Sigma}) \text{ for } i = 1, \dots, n$$
   For easier notation, we let $\phi^{(i)}$ be the vector representation of the density $p_Z^{(i)}$. That is,
   $$\phi^{(i)} = \begin{pmatrix} \phi_1^{(i)} \\ \vdots \\ \phi_k^{(i)} \end{pmatrix} \text{ with } \sum_{j=1}^{k} \phi_j^{(i)} = 1$$
   where $p_Z^{(i)}(z) \equiv \phi_z^{(i)}$ for $z = 1, \dots, k$ and 0 otherwise. Then, we can calculate $\phi^{(i)}$ (and therefore $p_Z^{(i)}$) component-wise by calculating each $\phi_j^{(i)}$ (which is the probability of a point being in the

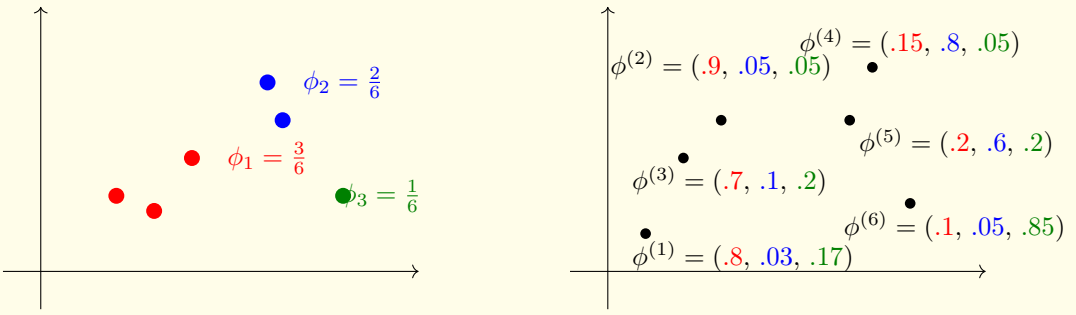
jth cluster, given that we observe example $x^{(i)}$):

$$\begin{aligned}\phi_j^{(i)} &= p_Z^{(i)}\big(z = j;\, \phi, \mu, \boldsymbol{\Sigma}\big) \\ &= p_Z\big(z = j \,|\, x^{(i)};\, \phi, \mu, \boldsymbol{\Sigma}\big) \\ &= \frac{p_{\mathcal{N}_j}\big(x^{(i)} \,|\, z^{(i)} = j;\, \mu, \boldsymbol{\Sigma}\big)\; p_Z\big(z^{(i)} = j;\, \phi\big)}{p_X\big(x^{(i)};\, \phi, \mu, \boldsymbol{\Sigma}\big)} \\ &= \frac{p_{\mathcal{N}_j}\big(x^{(i)} \,|\, z^{(i)} = j;\, \mu, \boldsymbol{\Sigma}\big)\; p_Z\big(z^{(i)} = j;\, \phi\big)}{\sum_{l=1}^{k} p_{\mathcal{N}_j}\big(x^{(i)} \,|\, z^{(i)} = l;\, \mu, \boldsymbol{\Sigma}\big)\; p_Z\big(z^{(i)} = l;\, \phi\big)}\end{aligned}$$

Note that we have everything we need to calculate the posterior probability distribution $p_Z^{(i)}(z)$ of a point being in any cluster.

- $p_{\mathcal{N}_j}(x^{(i)} \,|\, z^{(i)} = j)$ represents the conditional Gaussian density, which is completely defined because the parameters μ_j, Σ_j are already defined in initialization.
- $p_Z(z^{(i)} = j; \phi)$ is really just the probability ϕ_j that a given point is in the jth cluster, which we've also defined in initialization.
- $p_X(x^{(i)})$ represents the distribution of the entire random variable X of the entire training set. Knowing the first two and taking the sum gives this density function p_X.

Therefore, we should end up with n different k-vectors $\phi^{(1)}, \phi^{(2)}, \ldots, \phi^{(n)}$, each representing our best guess of what multinomial density $p_Z^{(i)}$ each $x^{(i)}$ had followed in order to be at the given points.



(a) Hard label assignments.

(b) Soft probability assignments.

Figure 30: Let us elaborate further on the intuition of this step. In the normal GDA with given values of $z^{(i)}$, we have $\phi_j = \frac{1}{n}\sum_{i=1}^{n} 1\{z^{(i)} = j\} = \frac{1}{n}\big(\text{Number of Samples in } j\text{th Gaussian}\big)$, which is a sum of "hard" guesses, meaning that each $x^{(i)}$ is undoubtedly in cluster j or not, and so to find out our best guess for the true vector ϕ, all we have to do is find out the proportion of all examples in each of the k groups and we're done (without needing to iterate). However, in our EM model, we do not know the $z^{(i)}$'s, and so the best we can do is give the *probability* $\phi_j^{(i)}$ that $x^{(i)}$ is in cluster j. So for each point $x^{(i)}$, the model has changed from it being undoubtedly in group $z^{(i)} = j$ to it having a probability of being in $\phi_j^{(i)}$ for $j = 1, \ldots, k$.

3. **(M Step)** With these n separate posterior estimates of Z for each observation $x^{(i)}$, we can simply average all of them and say that our best estimate of ϕ is

$$\phi = \frac{1}{n}\sum_{i=1}^{n} \phi^{(i)}$$

We can interpret the vectors $\phi^{(i)}$ as tuples where $\phi_j^{(i)}$ describes the expected "portion" of each sample $x^{(i)}$ to be in group j. So, we are adding up all the "portions" of the points that are expected to be in cluster j to get $\phi_j = \sum_{i=1}^{n} \phi_j^{(i)}$.

Now, given the jth Gaussian cluster, we would like to compute its mean μ_j. Since each $x^{(i)}$ has probability $\phi_j^{(i)}$ of being in cluster j, we can weigh each of the n points by $\phi_j^{(i)}$ (which determines how "relevant" $x^{(i)}$ is to cluster j) and average these (already weighted) points to get our "best-guess" of the mean μ_j.

$$\mu_j = \frac{\sum_{i=1}^n \phi_j^{(i)} x^{(i)}}{\sum_{i=1}^n \phi_j^{(i)}}$$

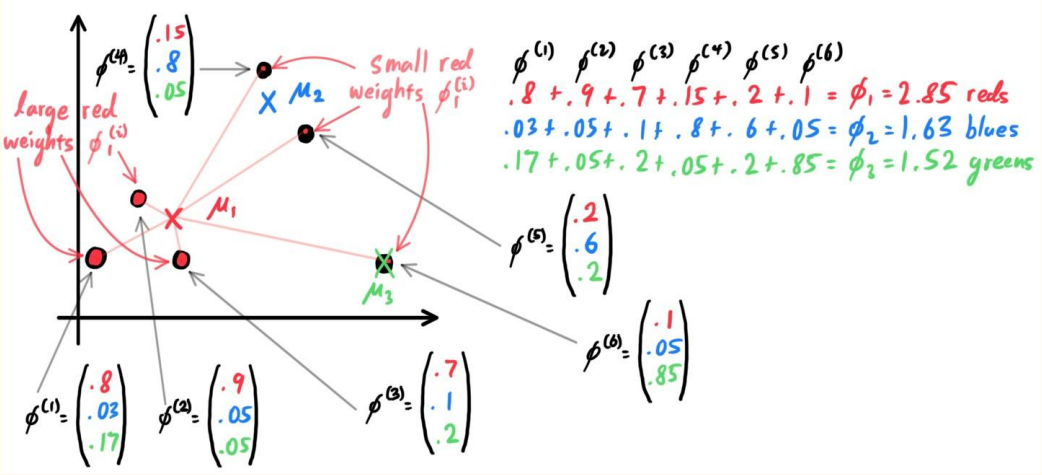


Figure 31

With this logic of weighted points, we finally update the covariance matrices Σ_j as below:

$$\Sigma_j = \frac{1}{\sum_{i=1}^n \phi_j^{(i)}} \sum_{i=1}^n \phi_j^{(i)} \left(x^{(i)} - \mu_j\right)\left(x^{(i)} - \mu_j\right)^T$$

4. Now, we have new values of $\phi, \mu_1, \dots, \mu_k, \Sigma_1, \dots, \Sigma_k$ that we can work with. With these new values, repeat steps 2 and 3 until convergence.

In summary, this entire algorithm results from modifying the "hard" data of each point $x^{(i)}$ being undoubtedly in one cluster to a model containing points $x^{(i)}$ that have been "smeared" around different clusters, with a probability $\phi_j^{(i)}$ being in cluster j.

Definition 12.2 (Mixture of Gaussians Algorithm: Summary)

Given a training set $\{x^{(i)}\}_{i=1}^n \in \mathbb{R}^d$, let us assume that the random variable X that these examples follow can be modeled by specifying a joint distribution of a multinomial and Gaussians. That is, it follows a Gaussian mixture model (GMM) of k Gaussian clusters. Let

- Z be the multinomial distribution representing which Gaussian cluster each example x falls in, with density represented by vector $\phi \in \mathbb{R}^k$ so that $\mathbb{P}(Z = j) \equiv \phi_j$.
- The set of conditional distributions

$$X \mid Z = j \sim \mathcal{N}(\mu_j, \Sigma_j) \text{ for } j = 1, 2, \dots, k$$

are multivariate Gaussian, with mean vectors $\mu_1, \dots, \mu_k$ and covariance matrices $\Sigma_1, \dots, \Sigma_k$. Let all the parameters be denoted as θ. Then, the EM algorithm is as such:

1. Initialize the multinomial vector ϕ, the μ_j's, and the Σ_j's.
2. **(E Step)** Calculate the n vectors

$$\phi^{(i)} = \begin{pmatrix} \phi_1^{(i)} \\ \vdots \\ \phi_k^{(i)} \end{pmatrix} \text{ for all } i = 1, \dots, n$$

that represent the posterior distribution of Z given observed $x^{(i)}$ by computing

$$\begin{aligned} \phi_j^{(i)} &= p_Z^{(i)}\big(z = j;\, \phi, \mu, \mathbf{\Sigma}\big) \\ &= p_Z\big(z = j \,|\, x^{(i)};\, \phi, \mu, \mathbf{\Sigma}\big) \\ &= \frac{p_{\mathcal{N}_j}\big(x^{(i)} \,|\, z^{(i)} = j;\, \mu, \mathbf{\Sigma}\big)\, p_Z\big(z^{(i)} = j;\, \phi\big)}{p_X\big(x^{(i)};\, \phi, \mu, \mathbf{\Sigma}\big)} \\ &= \frac{p_{\mathcal{N}_j}\big(x^{(i)} \,|\, z^{(i)} = j;\, \mu, \mathbf{\Sigma}\big)\, p_Z\big(z^{(i)} = j;\, \phi\big)}{\sum_{l=1}^{k} p_{\mathcal{N}_j}\big(x^{(i)} \,|\, z^{(i)} = l;\, \mu, \mathbf{\Sigma}\big)\, p_Z\big(z^{(i)} = l;\, \phi\big)} \end{aligned}$$

3. **(M Step)** Reassign the value of θ as

$$\begin{aligned} \phi &= \frac{1}{n} \sum_{i=1}^{n} \phi^{(i)} \\ \mu_j &= \frac{\sum_{i=1}^{n} \phi_j^{(i)} x^{(i)}}{\sum_{i=1}^{n} w_j^{(i)}} \text{ for } j = 1, \dots, n \\ \Sigma_j &= \frac{1}{\sum_{i=1}^{n} \phi_j^{(i)}} \sum_{i=1}^{n} \phi_j^{(i)} \big(x^{(i)} - \mu_j\big)\big(x^{(i)} - \mu_j\big)^T \text{ for } j = 1, \dots, n \end{aligned}$$

4. Repeat steps 2 and 3 until convergence.

12.2 Bayesian Networks (Directed Graphical Models)

Note that the whole purpose of directed graphical models is to model some sort of *causal* relationship between two random variables. Note that while this is successful in practice, there is really no way to know for sure about any causality.

Definition 12.3 (Bayesian Network)

A **Bayesian network**, also known as a **directed probability model**, is a directed acyclic graph of M nodes representing a joint probability distribution of M scalar random variables. An edge pointing $A \to B$ means that the B is conditionally dependent on A, and that there is a very clear casual relationship coming from A to B. The **parents** of a node x_i is denoted pa_i, and the entire joint distribution can be broken up as such:

$$p(\mathbf{x}) = \prod_{m=1}^{M} p(x_m \mid x_{\text{pa}_m}) \tag{267}$$

which is unique due to it being a DAG. Not only is a Bayesian network easy to parameterize. We can also sample from the joint distribution by sequentially sampling starting from the parents to the final children, and discarding the ones (marginalizing) that we don't wish to sample. This is known as **ancestral sampling**.

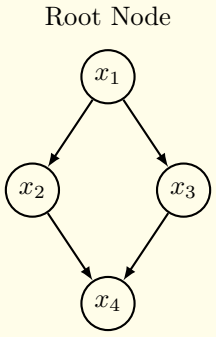


Figure 32

This following example cleared up any confusion when I learned Bayesian networks for the first time.

Example 12.1 (Relay Race)

Consider a 4×100m relay race where the final race time depends on multiple factors. We can model this as a Bayesian network where the total race time T depends on:

- Individual runner capabilities (R_1, R_2, R_3, R_4)
- Handoff success between runners (H_1, H_2, H_3)
- Individual leg performances (P_1, P_2, P_3, P_4)

The joint probability distribution factorizes as:

$$p(T, R_1, R_2, R_3, R_4, H_1, H_2, H_3, P_1, P_2, P_3, P_4) =$$
$$p(T|P_1, P_2, P_3, P_4) \prod_{i=1}^{4} p(R_i) \prod_{i=1}^{3} p(H_i|R_i, R_{i+1}) \prod_{i=1}^{4} p(P_i|R_i, H_{i-1})$$

where H_0 is undefined for P_1, and each runner's performance depends on their capability and the success of the previous handoff (except for the first runner). This network captures both the individual contributions and the critical dependencies between runners during baton exchanges.

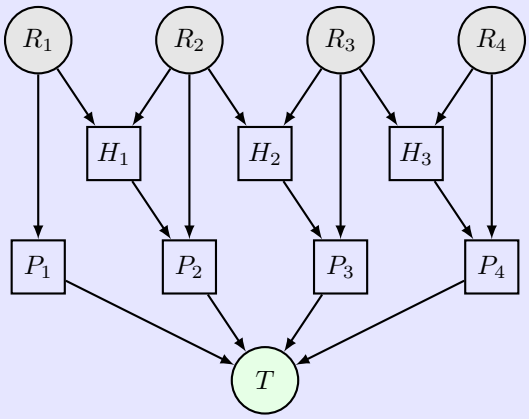


Figure 33: Bayesian Network for a 4x100m Relay Race. The graphical representation is much more compact and intuitive than simply writing out all the products.

Bayesian modelling with hierarchical priors.

Example 12.2 (Multinomial)

We first provide some motivation from a computational complexity perspective. Given a joint distribution of 2 random variables $\mathbf{x}_1, \mathbf{x}_2$, say which are multinomial with K classes, their joint distribution $p(\mathbf{x}_1, \mathbf{x}_2)$ is captured by K^2-1 parameters. For a general M random variables, then we have to keep a total of K^M-1 parameters, and this increases exponentially. By building a directed graph with say r maximum number of variables appearing on either side of the conditioning bar in a single probability distribution, then the computational complexity scales as $O(K^r)$, which may save a lot of time if $r << M$.

Extending upon this example, we can see that we want to balance two things:

1. Fully conncted graphs have completely general distributions and have $O(K^M-1)$ number of parameters (too complex).
2. If there are no links, the joint distribution fully factorizes into the product of its marginals and has $M(K-1)$ parameters (too simple) .

Graphs that have an intermediate level of connectivity allow for more general distributions compared to the fully factorized one, while requiring fewer parameters than the general joint distribution. One model that balances this out is the hidden markov model.

Example 12.3 (Chain Graph)

Consider an M-node Markov chain. The marginal distribution $p(\mathbf{x}_1)$ requires $K-1$ parameters, and the remaining conditional distributions $p(\mathbf{x}_i \mid \mathbf{x}_{i-1})$ requires $K(K-1)$ parameters. Therefore, the total number of parameters is

$$K-1+(M-1)(K-1)K \in O(MK^2) \tag{268}$$

which scales relatively well, and we have

$$p(\{\mathbf{x}_m\}) = p(\mathbf{x}_1) \prod_{m=2}^{M} p(\mathbf{x}_m \mid \mathbf{x}_{m-1}) \tag{269}$$

TBD

We can turn this same graph into a Bayesian model by introducing priors for the paramters. Therefore, each node requires an additional parent representing the distribution over parameters (e.g. prior can be Dirichlet)

$$p(\{\mathbf{x}_m, \mu_m\}) = p(\mathbf{x}_1 \mid \mu_1)p(\mu_1) \prod_{m=2}^{M} p(\mathbf{x}_m \mid \mathbf{x}_{m-1}, \mu_m)p(\mu_m) \tag{270}$$

with $p(\mu_m) = \text{Dir}(\mu_m \mid \alpha_m)$ for some predetermined fixed hyperparameter α_m.

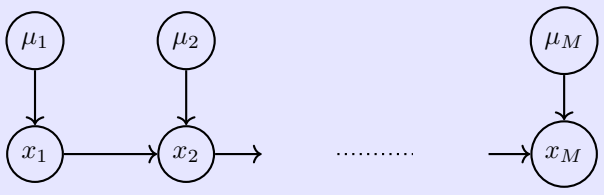


Figure 34

We could also choose to share a common prior over the parameters, trading flexibility for computational feasibility.

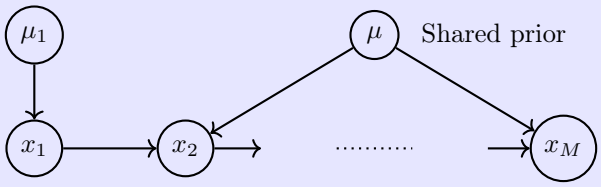


Figure 35

Another way to make more compact representations is through parameterized models. For example, if we have to compute $p(y = 1 \mid \mathbf{x}_1, \dots, \mathbf{x}_M)$, this in general has $O(K^M)$ parameters. However, we can obtain a more parsimonious form by using a logistic function acting on a linear combination of the parent variables

$$p(y = 1 \mid \mathbf{x}_1, \dots, \mathbf{x}_m) = \sigma\left(w_0 + \sum_{i=1}^{M} w_i x_i\right) = \sigma(\mathbf{w}^T \mathbf{x}) \tag{271}$$

We can look at an example how this is applied to sampling from high-dimensional Gaussian with **linear Gaussian models**.

Example 12.4 (Multivariate Gaussian)

Consider an arbitrary acyclic graph over D random variables, in which eachnode represents a single continuous Gaussian distribution with its mean given by a linear function of its parents.

$$p(x_i \mid \mathbf{pa}_i) = N\left(x_i \middle| w_{ij} x_j + b_j, v_i\right)$$

Given a multivariate Gaussian, let us try to decompose it into a directed graph. The log of the joint distribution takes form

$$\ln p(\mathbf{x}) = \sum_{i=1}^{D} \ln p(x_i \mid \text{pa}_i) = -\sum_{i=1}^{D} \frac{1}{2v_i}\left(x_i - \sum_{j \in \text{pa}_i} w_{ij} x_j - b_i\right)^2 + \text{const}$$

To compute the mean, we can see that by construction, every x_i is dependent on its ancestors, so

$$x_i = \sum_{j \in \text{pa}_i} w_{ij} x_j + b_i + \sqrt{v_i}\epsilon_i, \;\; \epsilon_i \sim N(0, 1)$$

so by linearity of expectation, we have

$$\mathbb{E}[x_i] = \sum_{j \in \text{pa}_i} w_{ij} \mathbb{E}[x_j] + b_i$$

So again, we can start at the top of the graph and compute the expectation. To compute covariance, we can obtain the i, jth element of $\mathbf{\Sigma}$ with a recurrence relation:

$$\begin{aligned}
\Sigma_{ij} &= \mathbb{E}[(x_i - \mathbb{E}[x_i])(x_j - \mathbb{E}[x_j])] \\
&= \mathbb{E}\left[(x_i - \mathbb{E}[x_i])\left(\sum_{k \in \text{pa}_j} w_{jk}(x_k - \mathbb{E}[x_k]) + \sqrt{v_i}\epsilon_j\right)\right] \\
&= \sum_{k \in \text{pa}_j} w_{jk} \Sigma_{ik} + I_{ij} v_j
\end{aligned}$$

If there were no links in the graphs, then the w_{ij}'s are 0, and so $\mathbb{E}[\mathbf{x}] = [b_1, \dots, b_D]$, making the covariance diagonal.If the graph is fully connected, then the total number of parameters is $D + D(D-1)/2$, which corresponds to a general symmetric covariance matrix.

Example 12.5 (Bilinear Gaussian Model)

Consider the following model

$$\begin{aligned} u &\sim N(0,1) \\ v &\sim N(0,1) \\ r &\sim N(uv,1) \end{aligned}$$

where the mean of r is a product of 2 Gaussians. This is also a parameterized model.

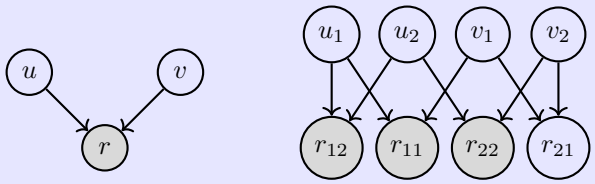


Figure 36

Definition 12.4 (Conditional Independence in Directed Graphs)

We say that a is independent of b given c if

$$p(a \mid b, c) = p(a \mid c)$$

or equivalently,

$$p(a, b \mid c) = p(a \mid b, c)\, p(b \mid c) = p(a \mid c)\, p(b \mid c)$$

Conveniently, we can directly read conditional independence properties of the joint distribution from the graph without any analytical measurements.

Example 12.6 (Conditional Independence on Dataset)

We can demonstrate conditional independence with iid data. Consider the problem of density estimation of some dataset $\mathcal{D} = \{x_i\}$ with some parameterized distribution of μ. Originally, the observations are not independent since they depend on μ.

$$p(\mathcal{D}) = \int_{\mu} p(\mathcal{D} \mid \mu)\, p(\mu)\, d\mu \tag{272}$$

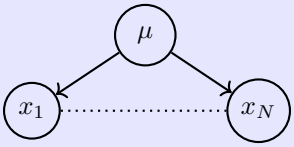


Figure 37

If we condition on μ and considered the joint over the observed variables, the variables are independent.

$$p(\mathcal{D} \mid \mu) = \prod_{n=1}^{N} p(x_n \mid \mu) \tag{273}$$

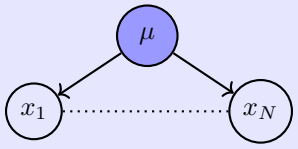


Figure 38

The example above identifies a node (the parent μ) where, if observed, causes the rest of the nodes to become independent. We can extend on this idea by taking an arbitrary x_i and finding a set of nodes such that if they are observed, then x_i is indepedent from every other node.

Definition 12.5 (Markov Blanket in Directed Graphs)

The **Markov blanket** of a node is the minimal set of nodes that must be observed to make this node independent of all other nodes. It turns out that the parents, children, and coparents are all in the Markov blanket.

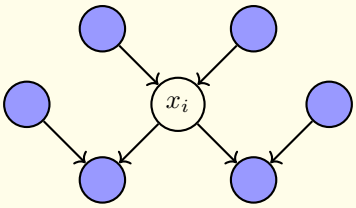


Figure 39

Note that

$$p(x_i \mid x_{j\neq i}) = \frac{p(x_1, \dots, x_M)}{\int p(x_1, \dots, x_M)\, dx} = \frac{\prod_k p(x_k \mid \text{pa}_k)}{\int \prod_k p(x_k \mid \text{pa}_k)\, dx_i} \tag{274}$$

One final interpretation is that we can view directed graphs as **distribution filters**. We take the joint probability distribution, will starts off as fully connected, and the directed graphs “filters" away the edges that are not needed. Therefore, the joint probability distribution $p(\mathbf{x})$ is only allows through the filter if and only if it satisfies the factorization property.

12.3 Markov Random Field (Undirected Graphical Models)

As the name implies, undirected models use undirected graphs, which are used to model relationships that go both ways rather than just one. Unlike directed graphs, which are useful for expressing casual relationships between random variables, undirected graphs are useful for expressing soft constraints between random variables.

Definition 12.6 (Conditional Independence in Undirected Graphs)

Fortunately, conditional independence is easier compared to directed models. We can say A is conditionally independent to B given C if C blocks all paths between any node in A and any node in B.

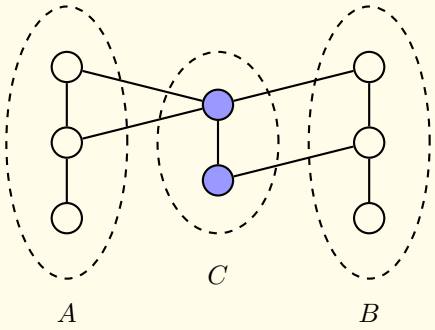


Figure 40: A is conditionally independent given C, denoted $A \perp\!\!\!\perp B|C$.

Definition 12.7 (Markov Blanket in Undirected Graphs)

The Markov blanket of a node, which is the minimal set of nodes that must be observered to make this node independent of the rest of the nodes, is simply the nodes that are directly connected to that node.

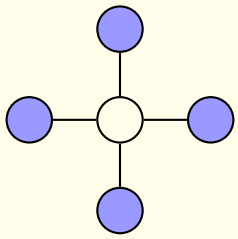


Figure 41: Once the neighbors of a node are realized, the node is independent of the rest of the nodes.

Therefore, the conditional distribution of x_i conditioned on all the variables in the graph is dependent only on the variables in the Markov blanket.

Now, let us talk about how we can actually define a probability distribution with this graph.

Definition 12.8 (Clique)

In an undirected graph, a **clique** is a set of nodes such that there exists a link between all pairs of nodes in that subset. A **maximal clique** is a clique such that it is not possible to include any other nodes in the set without it ceasing it to be a clique.

Given a joint random variable $\mathbf{x}$ represented by an undirected graph, the joint distribution is given by the product of non-negative potential functions over the maximal cliques

$$p(\mathbf{x}) = \frac{1}{Z} \prod_C \phi_C(x_C) \tag{275}$$

where

$$Z = \int p(\mathbf{x})\, d\mathbf{x} \tag{276}$$

is the normalizing constant, called the **partition function**. That is, each x_C is a maximal clique and ϕ_C is the nonnegative potential function of that clique.

This assignment looks pretty arbitrary. How do we know that any arbitrary joint distribution of $\mathbf{x}$, which has a undirected graphical representation, can be represented as the product of a bunch of functions over

the maximum cliques? Fortunately, there is a mathematical result that proves this.

Theorem 12.1 (Hammersley-Clifford)

The joint probability distribution of any undirected graph can be written as the product of potential functions on the maximal cliques of the graph. Furthermore, for any factorization of these potential functions, there exists an undirected graph for which is the joint.

Example 12.7 ()

For example, the joint distribution of the graph below

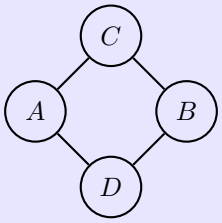


Figure 42

factorizes into

$$p(A,B,C,D) = \frac{1}{Z}\phi(A,C)\,\phi(C,B)\,\phi(B,D)\,\phi(A,D) \tag{277}$$

Note that each potential function ϕ is a mapping from the joint configuration of random variables in a clique to non-negative real numbers. The choice of potential functions is not restricted to having specific probabilistic interpretations, but since they must be nonnegative, we can just represent them as an exponential. The negative sign is not needed, but is a remnant of physics notation.

$$p(\mathbf{x}) = \frac{1}{Z}\prod_C \phi_C(x_C) = \frac{1}{Z}\exp\left\{-\sum_C E(x_C)\right\} = \frac{1}{Z}\underbrace{\exp\{-E(\mathbf{x})\}}_{\substack{\text{Boltzmann}\\ \text{distribution}}} \tag{278}$$

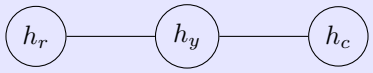
Any distribution that can be represented as the form above is called a **Boltzmann distribution**. So far, all we stated is that the joint probability distribution can be expressed as the product of a bunch of potential functions, but besides the fact that it is nonnegative, there is no probabilistic interpretation of these potentials (or equivalently, the energy functions). While this does give us greater flexibility in choosing potential functions, we must be careful in choosing them (e.g. choosing something like x^2 may cause the integral to diverge, making the joint not well-defined).

Clearly, these potential functions over the cliques should express which configuration of the local variables are preferred to others. It should assign higher values to configurations that are deemed (either by assumption or through training data) to be more probable. That is, each potential is like an "expert" that provides some opinion (the value) on a configuration, and the product of the values of all the potential represents the total opinion of all the experts. Therefore, global configurations with relatively high probabilities are those that find a good balance in satisfying the (possibly conflicting) influences of the clique potentials.

Example 12.8 (Transmission of Colds)

Say that you want to model a distribution over three binary variables: whether you or not you, your coworker, and your roommate is sick (0 represents sick and 1 represents healthy). Then, you can make simplifying assumptions that your roommate and your coworker do not know each other, so it is very unlikely that one of thme will give the other an infection such as a cold directly. Therefore,

we can model the indirect transmission of a cold from your coworker to your roommate by modeling the transmission of the cold from your coworker to you and then you to your roommate. Therefore, we have a model of form



One max clique contains h_y and h_c. The factor for this clique can be defined by a table and might have values resembling these.

	$h_y = 0$	$h_y = 1$
$h_c = 0$	2	1
$h_c = 1$	1	10

Table 2: States and Values of h_y and h_c

This table completely describes the potential function of this clique. Both of you are usually healthy, so the state $(1, 1)$ gets the maximum value of 1. If one of you are sick, then it is likely that the other is sick as well, so we have a value of 2 for $(0, 0)$. Finally, it is most unlikely that one of you is sick and the other healthy, which has a value of 1.

12.4 Hidden Markov Models

12.5 Expectation-Maximization Algorithm

EM algorithm is used to optimize latent variable models. Recall **Jensen's Inequality**: Given a convex function $f : \mathbb{R} \longrightarrow \mathbb{R}$ (meaning that $f''(x) \geq 0$ for all x) and a random variable X, we have

$$\mathbb{E}\big(f(X)\big) \geq f\big(\mathbb{E}(X)\big)$$

Moreover, if f is strictly convex, then $\mathbb{E}\big(f(X)\big) = f\big(\mathbb{E}(X)\big)$ holds true if and only if $X = \mathbb{E}(X)$ with probability 1 (i.e. if X is a constant).

Suppose we have an estimation problem given the training set $\{x^{(i)}\}_{i=1}^n$. We have latent variable model $p(x, z; \theta)$ with z being the latent variable of discrete, finite random variable Z, with density $p_Z(z)$. Let us denote the density of X as p_X. Then, the random variable X can be interpreted as us first generating z from Z, and then computing $X \,|\, Z = z$.

$$\text{Compute } X = \text{Compute } Z \text{ and then } \begin{cases} \text{Compute } X \,|\, Z = 1 \\ \text{Compute } X \,|\, Z = 2 \\ \dots \\ \text{Compute } X \,|\, Z = k \end{cases}$$

Let us clarify some notation:

- The distribution that we will iteratively reassign over and over again is Z, with density p_Z that maps $z \mapsto \phi_z$, where ϕ is a vector that represents the density.
- The k conditional (not necessarily Gaussian) distributions that we will iteratively reassign over and over again is $X_1, X_2, \dots, X_k$, with densities $p_{X_1}(x), \dots, p_{X_k}(x)$ that maps $x \mapsto p_{X_j}(x)$.
- The distribution of the entire random variable X will have density $p_X(x)$. Since we are iteratively reassigning the densities p_Z and p_{X_j}, this joint distribution of X will also get modified.

The EM algorithm in the general case has the following steps:

1. We initialize the value of θ in some way. Note that within this θ are the parametizations of the initial multinomial density p_Z, which is our initial "guess" of the distribution of Z.

2. **(E Step)** The log likelihood of the given data $\{x^{(i)}\}_{i=1}^n$ with respect to the parameter θ (which encodes all parameters of distribution Z and all $X \mid Z$) is

$$l(\theta) = \sum_{i=1}^{n} \log\, p_X\big(x^{(i)};\, \theta\big)$$

It turns out that explicitly finding the maximum likelihood estimates of the parameters θ is hard because it results in a difficult, non-convex optimization problem. So, we tackle this another way.

To start, we can see that the summation isn't too crucial, so we can focus on minimizing each $\log\, p_X\big(x^{(i)};\, \theta\big)$ and summing in the end. We can calculate this by conditioning over all $j = 1, \dots, k$ generated from Z (which we have guessed to have an initial density of p_Z). That is, we must find for each $i = 1, 2, \dots, n$

$$\begin{aligned}
\max_\theta \log\, p_X\big(x^{(i)};\, \theta\big) &= \max_\theta \log\, \left(\sum_{j=1}^{k} p_X\big(x^{(i)}, Z = j;\, \theta\big) \right) \\
&= \max_\theta \log \left(\sum_{j=1}^{k} p_X\big(x^{(i)} \mid Z = j;\, \theta\big) p_Z\big(j;\, \theta\big) \right) \\
&= \max_\theta \log \left(\sum_{j=1}^{k} p_{X_j}\big(x^{(i)};\, \theta\big) p_Z\big(j;\, \theta\big) \right)
\end{aligned}$$

To find this maximum value, we can focus on the first equality and see that by Jensen's inequality (with conCAVE, not convex, $f(x) = \log x$ over domain $x \in \mathbb{R}^+$), the following holds true for all θ and more importantly, for *any arbitrary density function* p_Z^{*i}.

$$\begin{aligned}
\log p_X\big(x^{(i)};\, \theta\big) &= \log \left(\sum_{j=1}^{k} p_X\big(x^{(i)}, Z = j;\, \theta\big) \right) \\
&= \log \left(\sum_{j=1}^{k} p_Z^{*i}(j)\, \frac{p_X\big(x^{(i)}, Z = j;\, \theta\big)}{p_Z^{*i}(j)} \right) \\
&= \log \left(\mathbb{E}_{j \sim p_Z^{*i}} \left(\frac{p(x^{(i)}, Z = j;\, \theta)}{p_Z^{*i}(j)} \right) \right) \\
&\geq \mathbb{E}_{j \sim p_Z^{*i}} \left(\log \left(\frac{p(x^{(i)}, Z = j;\, \theta)}{p_Z^{*i}(j)} \right) \right) \\
&= \sum_{j=1}^{k} p_Z^{*i}(j)\, \log \left(\frac{p(x^{(i)}, Z = j;\, \theta)}{p_Z^{*i}(j)} \right) = \mathrm{ELBO}\big(x^{(i)};\, p_Z^{*i}, \theta\big)
\end{aligned}$$

The final term, called the **evidence lower bound** (ELBO), is just the expectation of $\log \frac{p(x^{(i)}, Z = j;\, \theta)}{p_Z^{*i}(j)}$ with respect to j drawn from density p_Z^{*i}, which is denoted with $\mathbb{E}_{j \sim p_Z^{*i}}$.

Summing over all n examples, we have a lower bound for the entire log likelihood for *any* set of density functions $p_Z^{*1}, p_Z^{*2}, \dots, p_Z^{*n}$:

$$\begin{aligned}
l(\theta) = \sum_{i=1}^{n} \log p(x^{(i)};\, \theta) &\geq \sum_{i=1}^{n} \mathrm{ELBO}(x^{(i)};\, p_Z^{*i}, \theta) \\
&= \sum_{i=1}^{n} \sum_{j=1}^{k} p_Z^{*i}(j)\, \log \left(\frac{p(x^{(i)}, Z = j;\, \theta)}{p_Z^{*i}(j)} \right)
\end{aligned}$$

Our job now is to choose the correct density functions p_Z^{*i}'s such that the lower bound is maximized. It turns out that we can do even better: equality is satisfied if and only if we set

$$\begin{aligned} p_Z^{*i}(j) &\equiv p_Z\big(j \mid x^{(i)};\, \theta\big) \\ &\equiv p_Z^{(i)}\big(j;\, \theta\big) \text{ for all } i = 1, 2, \dots, n \end{aligned}$$

which is simply the posterior distribution of the multinomial given the observed sample $x^{(i)}$, which we can easily calculate using Bayes' rule. Substituting this into p_Z^{*i} leads to the equality

$$\begin{aligned} l(\theta) = \sum_{i=1}^{n} \log p(x^{(i)};\, \theta) &= \sum_{i=1}^{n} \text{ELBO}\big(x^{(i)}; p_Z^{(i)},\, \theta\big) \\ &= \sum_{i=1}^{n} \sum_{j=1}^{k} p_Z^{(i)}\big(j;\, \theta\big)\, \log\left(\frac{p(x^{(i)}, Z = j;\, \theta)}{p_Z^{(i)}\big(j;\, \theta\big)}\right) \\ &= \sum_{i=1}^{n} \sum_{j=1}^{k} p_Z\big(j \mid x^{(i)};\, \theta\big)\, \log\left(\frac{p(x^{(i)}, Z = j;\, \theta)}{p_Z\big(j \mid x^{(i)};\, \theta\big)}\right) \end{aligned}$$

In summary, this E step has taken the log-likelihood function $l(\theta)$ (representing (the log of) the probability of all the $x^{(i)}$'s landing where they are given the parameters θ), which is abstract and hard-to-optimize, and converted it into an equivalent form as the sum of a bunch of ELBO functions optimized with the density parameters begin assigned $p_Z^{*i} = p_Z^{(i)}$.

But remember that these optimal densities $p_Z^{*i} = p_Z^{(i)}$ make the right and left hand side equivalent only for a **fixed value** of θ! So, the right hand side is only equivalent to $l(\theta)$ only for that one value of θ, but as soon as we set θ to something else, the right hand side evaluated with $p_Z^{*i} = p_Z^{(i)}$ are not equal.

3. **(M Step)** Since we have found some equivalent form of $l(\theta)$ for the fixed θ that was initialized, we can now just maximize the right hand side with respect to θ, while fixing the $p_Z^{*i} = p_Z^{(i)}$'s. Therefore, we find and set the value of θ as

$$\begin{aligned} \theta &= \arg\, \max_{\theta} \sum_{i=1}^{n} \text{ELBO}\big(x^{(i)};\, p_Z^{(i)}, \theta\big) \\ &= \arg\, \max_{\theta} \sum_{i=1}^{n} \sum_{j=1}^{k} p_Z^{(i)}(j)\, \log\left(\frac{p(x^{(i)}, Z = j;\, \theta)}{p_Z^{(i)}(j)}\right) \\ &= \arg\, \max_{\theta} \sum_{i=1}^{n} \sum_{j=1}^{k} p_Z\big(j \mid x^{(i)};\, \theta\big)\, \log\left(\frac{p(x^{(i)}, Z = j;\, \theta)}{p_Z\big(j \mid x^{(i)};\, \theta\big)}\right) \end{aligned}$$

In the case where the parameter θ consist of $\phi, \mu_1, \dots, \mu_k, \Sigma_1, \dots, \Sigma_k$ like in the GMM model, it happens so that the maximum is found by computing ϕ to be the average of the $\phi^{(i)}$'s, each μ_j to be the weighed averages of the points, and each Σ_j as the equation above. For other distributions, the formula for the maximum must be mathematically found (or algorithmically computed) with respect to parameter θ.

4. We have now reassigned the entire value of θ, meaning that the parameters representing our guess of density p_Z of Z has also been modified. With this new value of θ, we repeat steps 2 and 3 until convergence.

For some intuition, we can visualize l as a function of θ. For the sake of visuals, we will assume that $\theta \in \mathbb{R}$ and $l : \mathbb{R} \longrightarrow \mathbb{R}$. On the contrary to what a visual is supposed to do, we want to point out that we cannot just visualize l as a curve in $\mathbb{R} \times \mathbb{R}$. This can be misleading since then it implies that the optimal θ value is easy to find, as shown in the left. Rather, we have no clue what the whole curve of l looks like, but we can get little snippets (right).

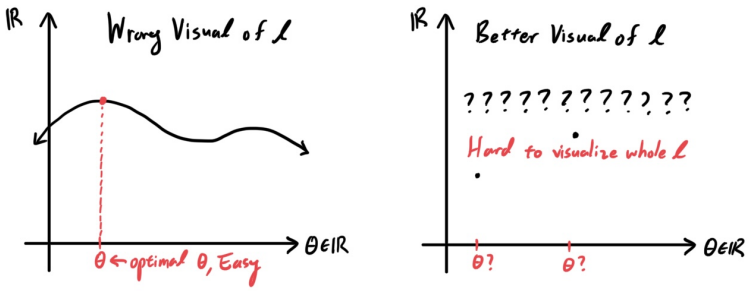


Figure 43

Rather, all we can do is hope to take whatever easier-to-visualize, lower-bound functions and maximize them as much as we can in hopes of converging onto l. Let us walk through the first two iterations of the EM algorithm. We first initialize θ to, say θ_0. This immediately induces the lower-bound ELBO-sum function $\sum_i \text{ELBO}(x^{(i)}; p_Z^{*i}, \theta)$, which takes in multinomial density functions $p_Z^{*i} = p_1, p_2, \dots$ and outputs different functions of θ that are valid lower bounds. Two of these possible lower-bound functions are shown (in green) for when we input some arbitrary density p_1, p_2. However, there exists a density $p_Z^{(i)}$ that produces not only the maximum possible lower-bound (called max ELBO, shown in red) but is equal to $l(\theta)$ for that density input $p_Z^{(i)}$. We maximize this function with respect to θ to get θ_1 as our next assignment of θ.

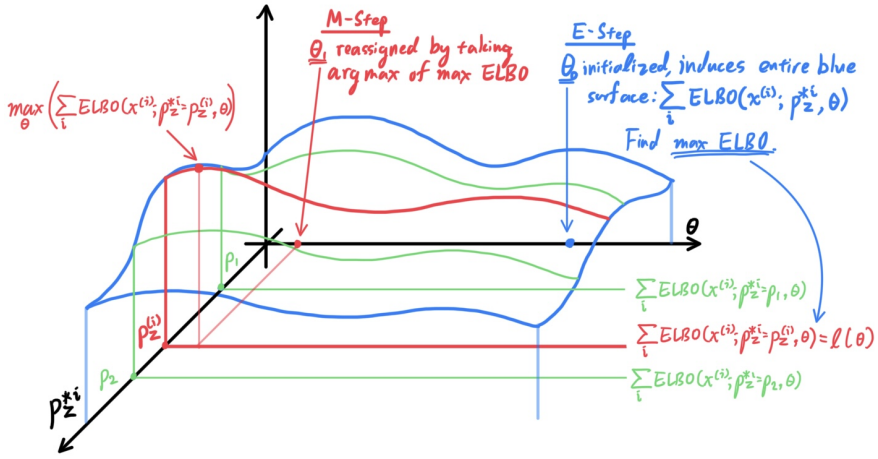


Figure 44

The next step is identical. Now that we have a new value of $\theta = \theta_1$, this induces the lower-bound ELBO-sum function $\sum_i \text{ELBO}(x^{(i)}; p_Z^{*i}, \theta)$ that also takes in multinomial densities p_Z^{*i} and outputs different functions of θ that are valid lower-bounds. Two possible lower bounds are shown (in green), but the maximum lower-bound (in blue) is produced when we input density $p_Z^{(i)}$. Since this max ELBO function is equal to θ for this fixed density input $p_Z^{(i)}$, we maximize this function with respect to θ to get θ_2 as our next assignment of θ.

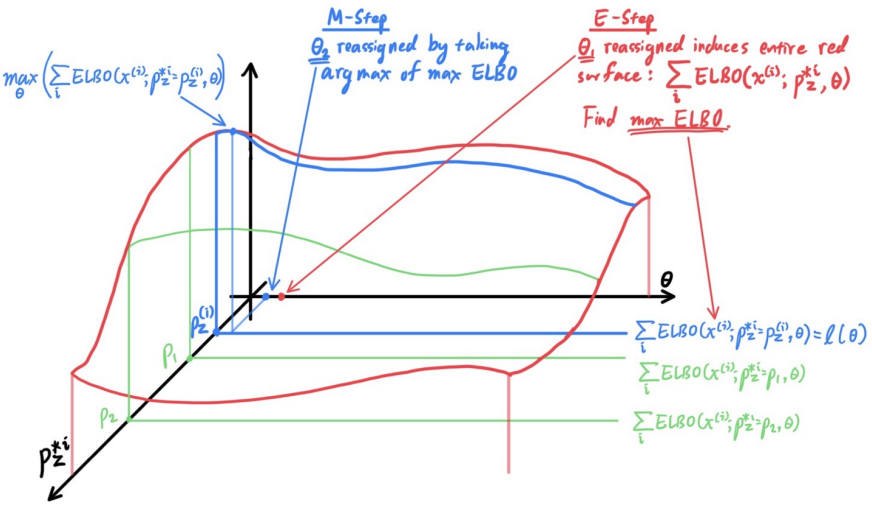


Figure 45

Definition 12.9 (EM Algorithm for General Estimation Problems)

Given a training set $\{x^{(i)}\}_{i=1}^n \in \mathbb{R}^d$, let us assume that the random variable X that these examples follow can be modeled by specifying a joint distribution of a multinomial and some arbitrary distributions. Let there be k clusters, and let

- Z be the multinomial distribution representing which Gaussian cluster each example x falls in, with density $p_Z(j)$ and represented by vector $\phi \in \mathbb{R}^k$ so that $\mathbb{P}(Z = j) = \phi_j$. Let the parameters of ϕ be encoded in θ.
- The set of conditional distributions

$$X \mid Z = j \sim X_j \text{ for } j = 1, 2, \dots, k$$

  are arbitrary distributions with some parameters, also all encoded in θ.

The EM algorithm is described as such:

1. Initialize θ.
2. **(E Step)** Since $l(\theta)$ is bounded below for all $p_Z^{*1}, \dots, p_Z^{*n}$ as

$$l(\theta) \equiv \sum_{i=1}^n \log p\big(x^{(i)}; \theta\big) \geq \sum_{i=1}^n \text{ELBO}\big(x^{(i)}; p_Z^{*i}, \theta\big)$$

   setting $p_Z^{*i}(j) = p_Z^{(i)}(j) = p_Z\big(j \mid x^{(i)}; \theta\big)$ for all $i = 1, \dots, n$ would put l into a new form for these specific fixed values of p_Z^{*i}.

$$l(\theta) = \sum_{i=1}^n \text{ELBO}\big(x^{(i)}; p_Z^{(i)}, \theta\big)$$

3. **(M Step)** We maximize this equivalent form of $l(\theta)$ with respect to θ whilst fixing the choice

of $p_Z^{(i)}$. That is, we set the value of θ as

$$\begin{aligned}\theta &= \arg\max_\theta \sum_{i=1}^{n} \text{ELBO}\big(x^{(i)}; p_Z^{(i)}, \theta\big) \\ &= \arg\max_\theta \sum_{i=1}^{n}\sum_{j=1}^{k} p_Z^{(i)}(j)\ \log\left(\frac{p(x^{(i)}, Z=j;\theta)}{p_Z^{(i)}(j)}\right) \\ &= \arg\max_\theta \sum_{i=1}^{n}\sum_{j=1}^{k} p_Z\big(j \mid x^{(i)};\theta\big)\ \log\left(\frac{p(x^{(i)}, Z=j;\theta)}{p_Z\big(j \mid x^{(i)};\theta\big)}\right)\end{aligned}$$

4. We have successfully updated θ. Now, we repeat steps 2 and 3 until convergence. Step 2 can bring improvements because we have changed the θ, which means that there is a new sum of ELBO functions of the θ that serves as a new lower bound.

13 Cross Validation

We have understood the theoretical foundations of overfitting and underfitting with the bias variance decomposition. But in practice, we don't have an ensemble of datasets; we just have one. Therefore, we don't actually know what the bias, the variance, or the noise is at all. Therefore, how do we actually *know* in practice when we are underfitting or overfitting? Easy. We just split our dataset into 2 different parts: the training set and testing sets.

$$\mathcal{D} = \mathcal{D}_{train} \sqcup \mathcal{D}_{test} \tag{279}$$

What we usually have is a **training set** that allows us to train the model, and then to check its performance we have a **test set**. We would train the model on the training set, where we will always minimize the loss, and then we would look at the loss on the test set. Though we haven't made a testing set, since we know the true model let us just generate more data and use that as our testing set. For each model, we can calculate the optimal $\boldsymbol{\theta}$, which we will denote $\boldsymbol{\theta}^*$, according to the **root mean squared loss**

$$h_{\boldsymbol{\theta}^*} = \underset{h_{\boldsymbol{\theta}}}{\text{argmin}} \sqrt{\frac{1}{N} \sum_{i=1}^{N} \left(y^{(i)} - h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}) \right)^2} \tag{280}$$

where division of N allows us to compare different sizes of datasets on equal footing, and the square root ensures that this is scaled correctly. Let us see how well these different order models perform on a separate set of data generated by the same function with Gaussian noise.

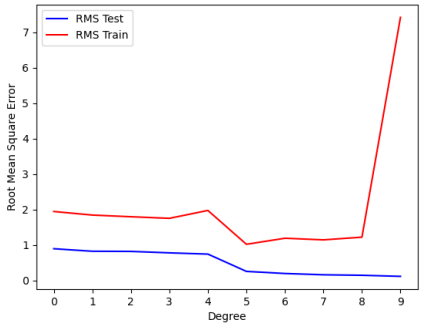


Figure 46: We can see that the RMS decreases monotonically on the training error as more complex functions become more fine-tuned to the data. However, when we have a 9th degree polynomial the RMS for the testing set dramatically increases, meaning that this model does not predict the testing set well, and performance drops.

Now we know that a more complex model (i.e. that captures a greater set of functions) is not necessarily the best due to overfitting. Therefore, researchers perform **cross-validation** by taking the training set $(\mathcal{X}, \mathcal{Y})$. We divide it into S equal pieces

$$\bigcup_{s=1}^{S} D_s = (\mathcal{X}, \mathcal{Y}) \tag{281}$$

Then, we train the model $\mathcal{M}$ on $S-1$ pieces of the data and then test it across the final piece, and do this S times for every test piece, averaging its perforance across all S test runs. Therefore, for every model $\mathcal{M}_k$, we must train it S times, for all K models, requiring KS training runs. If data is particularly scarce, we set $S = N$, called the **leave-one-out** technique. Then we just choose the model with the best average test performance.

The following result shows that cross-validation (data splitting) leads to an estimator with risk nearly as good as the best model in the class.

Theorem 13.1 (Gyorfi, Kohler, Krzyak, Walk (2002))

Let $\mathcal{M} = \{m_h\}$ be a finite class of regression estimators indexed by a parameter h, with m being the true risk minimizer, $m_{\hat{h}}$ being the empirical risk minimizer over the whole dataset $\mathcal{D}$, and m_H being the empirical risk minimizer over the test set $\mathcal{D}_{\text{test}}$ for ordinary least squares loss.

$$m_H = \underset{m_h}{\text{argmin}} \frac{1}{N} \sum_{i \in \mathcal{D}_{\text{test}}} (y_i - m_h(x_i))^2 \tag{282}$$

$$m_{\hat{h}} = \underset{m_h}{\text{argmin}} \frac{1}{N} \sum_{i \in \mathcal{D}} (y_i - m_h(x_i))^2 \tag{283}$$

If the data Y_i and estimators are bounded by L, then for any $\delta > 0$, we have

$$\mathbb{E} \int |m_H(x) - m(x)|^2 \, d\mathbb{P}(x) \leq (1+\delta)\mathbb{E} \int |m_{\hat{h}}(x) - m(x)|^2 \, d\mathbb{P}(x) + \frac{C(1 + \log|M|)}{n} \tag{284}$$

where $c = L^2(16/\delta + 35 + 19\delta)$.

Code 13.1 (Minimal Example of Train Test Split in scikit-learn)

To implement this in scikit-learn, we want to use the `train_test_split` class. We can also set a random state parameter to reproduce results.

```
from sklearn.model_selection import train_test_split

# Split into training (80\%) and test (20\%) data
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2,
    random_state=66)
```

However, this process requires a lot of training runs and therefore may be computationally infeasible. Therefore, various **information criterion** has been proposed to efficiently select a model.

13.1 Leave 1 Out Cross Validation

13.1.1 Generalized (Approximate) Cross Validation

13.1.2 Cp Statistic

13.2 K Fold Cross Validation

13.3 Data Leakage

13.4 Information Criterion

14 Practical Methodology

14.1 Model Selection

We've talked about the theory and implementation behind all these models, but in practice, how do we even use them? If we are trying to predict lung cancer in a patient, do we use linear regression, a nonparametric model, or something else? It's not clear at all what to do with the data. Unfortunately, this just comes with domain expertise and experience with data, but we can provide some general pointers.

As stated before, we have the flexibility to choose whatever model to train on. So how do we choose which form is the best? Well this is just an assumption that most researchers make, and this is called **model selection**.

Example 14.1 (Polynomial Regression)

The number of terms M, i.e. the degree $M-1$ of the polynomial

$$h_{\boldsymbol{\theta}}(x) = w_0 + w_1 x + w_2 x^2 + \ldots + w_{M-1} x^{M-1}$$

in polynomial regression gives us models with different complexities, where $\mathcal{M}_M$ determines the model with a $M-1$th degree polynomial.

Example 14.2 ()

Suppose I have data sampled data $x^{(1)}, \ldots, x^{(N)}$ on age at death for N people from an unknown distribution X. Then, possible models that model the distribution are

1. $\mathcal{M}_1$: the exponential distribution $p(x \mid \lambda) = \lambda e^{-\lambda y}$ with parameter $\theta = \lambda$.
2. $\mathcal{M}_2$: the gamma distribution $p(y \mid a, b) = (b^a / \Gamma(a)) y^{a-1} e^{-by}$ with parameter $\boldsymbol{\theta} = (a, b)$.
3. $\mathcal{M}_3$: the log-normal distribution with $X \sim N(\mu, \sigma^2)$ where $\boldsymbol{\theta} = (\mu, \sigma^2)$.

Example 14.3 ()

A mixture of Gaussians model

$$p(\mathbf{y}) = \sum_{m=1}^{M} \pi_m N(\mathbf{y} \mid \boldsymbol{\mu}_m, \boldsymbol{\Sigma}_m)$$

has submodels where we must determine the number of Gaussians M.

Now if we assume that the actual true distribution X or the true regressor $\mathbb{E}[Y \mid X]$ is contained within our model $\mathcal{M}$, then we say our model is **well-specified**. But since researchers have no idea what the data generating process is, so $\mathbb{E}[Y \mid X] \notin \mathcal{M}$. Hence there is the saying that saying that "all models are wrong," since we never know what the true data generating process is, and thus the quantity

$$\mathbb{E}[Y \mid X] - h^*_{\boldsymbol{\theta}}(X)$$

where $h^*_{\boldsymbol{\theta}}(X)$ is the optimized hypothesis functions within $\mathcal{M}$, will always be nonzero. How close we can get this quantity to 0 determines how useful the model is, and a misspecified model is fundamentally a convenient (or even necessary) assumption on the distribution underlying the data, which may only be a reasonable approximation.

14.2 Feature Engineering

This is also very domain specific.

14.3 Data Preprocessing

14.3.1 Feature Extraction

The simplest linear function for regression is simply

$$h_{\mathbf{w}}(\mathbf{x}) = w_0 + w_1 x_1 + \ldots + w_D x_D$$

This is called linear regression not because h is a linear function of $\mathbf{x}$. It is a linear function of $\mathbf{w}$. Therefore, we can fix nonlinear functions $\phi_j(\mathbf{x})$ and consider linear combinations of them.

$$h_{\mathbf{w}}(\mathbf{x}) = w_0 + \sum_{j=1}^{M-1} w_j \phi_j(\mathbf{x})$$

We usually choose a dummy basis function $\phi_0(\mathbf{x}) = 1$ for notational convenience, so that if $\boldsymbol{\phi}$ is the vector of the function ϕ_j, then we can write $h_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x})$. This mapping from the original variables $\mathbf{x} \in \mathbb{R}^D$ to the basis functions $\{\phi_j(\mathbf{x})\}$, which span a linear function space of dimension M, is called **preprocessing** or **feature extraction** of the data.

Example 14.4 ()

Here are some examples of how we can extract features.

1. The mapping from a single variable x to its powers
$$x \mapsto (1, x, x^2, \ldots, x^{M-1}) \tag{285}$$
2. The mapping from a configuration of K atoms with their momenta in $\mathbb{R}^{6K}$ to their atomic cluster expansion polynomials.
3. The Legendre polynomials, which form an orthonormal basis in the space of polynomials.
4. Using equally spaced Gaussian basis functions over the dataset.

Changing the input space from D dimensions to M dimensions (i.e. extracting our M features) gives the design matrix

$$\mathbf{X} = \begin{pmatrix} \mathbf{x}^{(1)} \\ \mathbf{x}^{(2)} \\ \mathbf{x}^{(3)} \\ \vdots \\ \mathbf{x}^{(n)} \end{pmatrix} \implies \boldsymbol{\Phi} = \begin{pmatrix} — & \phi(\mathbf{x}^{(1)}) & — \\ — & \phi(\mathbf{x}^{(2)}) & — \\ \vdots & \vdots & \vdots \\ — & \phi(\mathbf{x}^{(n)}) & — \end{pmatrix} \tag{286}$$

We have shown that the `PolynomialFeatures` transformer converts our features to a polynomial basis. We can do this for an arbitrary number of features, for example if we map $D = 2$ to a second degree polynomial, we would have the transformation

$$(x_1, x_2) \mapsto (1, x_1, x_2, x_1^2, x_1 x_2, x_2^2)$$

```
>>> import numpy as np
>>> from sklearn.preprocessing import PolynomialFeatures
>>> X = np.arange(6).reshape(3, 2)
>>> X
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> poly = PolynomialFeatures(2)
>>> poly.fit_transform(X)
```

```
array([[ 1.,  0.,  1.,  0.,  0.,  1.],
       [ 1.,  2.,  3.,  4.,  6.,  9.],
       [ 1.,  4.,  5., 16., 20., 25.]])
```

Sometimes, we are only worried about the interaction terms among features, so we can set the parameter `interaction_only=True`, which would, in the third degree case, transform the features

$$(x_1, x_2, x_3) \mapsto (1, x_1, x_2, x_3, x_1x_2, x_1x_3, x_2x_3, x_1x_2x_3)$$

Spline transformers are piecewise polynomials, which is also built in. We notice that it is cumbersome to transform the dataset `X` with the transformer, store it into another variable, and train the model on that. We can "combine" the transforming (even multiple layers of transformers) and the model by implementing a "pipeline," which is initialized by inputting a list of tuples (name and the object) and has the same methods as the model.

```
from sklearn.pipeline import Pipeline
model = Pipeline([("poly_transform", PolynomialFeatures(degree=2)),
                  ("lin_regression", LinearRegression())
                  ])
model.fit(X, y)
```

Now, let's talk about how we can implement a custom transformer. We basically have to create a new subclass that implements the `fit` (which always returns `self`) and the `transform` (which returns the transformed matrix) methods. Here we show for Gaussian basis functions.

```
from sklearn.base import BaseEstimator, TransformerMixin

class GaussianFeatures(BaseEstimator, TransformerMixin):
    """Uniformly spaced Gaussian features for one-dimensional input"""

    def __init__(self, N, width_factor=2.0):
        self.N = N
        self.width_factor = width_factor

    def fit(self, X, y=None):
        # create N centers spread along the data range
        self.centers_ = np.linspace(X.min(), X.max(), self.N)
        self.width_ = self.width_factor * (self.centers_[1] - self.centers_[0])
        return self

    def transform(self, X):
        transformed_rows = []
        for mu in self.centers_:
            transformed_rows.append(stats.norm.pdf(X, mu, self.width_))

        return np.hstack(tuple(transformed_rows))

model = Pipeline([("gauss_transform", GaussianFeatures(20)),
                  ("lin_regression", LinearRegression())
                  ])

N = 60
X = np.random.uniform(-1, 1, size=(N, 1))
Y = true_func(X) + np.random.normal(0, 0.3, size=(N, 1))

model = Pipeline([("gauss_transform", GaussianFeatures(10)),
```

```
                ("lin_regression", LinearRegression())
                ])
model.fit(X, Y)
```

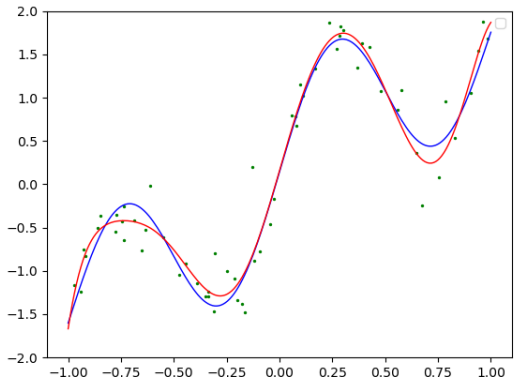
If we would like to impelment the fourier expansion of a function of form

$$f(x) = \frac{1}{2}a_0 + \sum_{n=1}^{N} a_n \cos(nx) + \sum_{n=1}^{N} b_n \sin(nx)$$

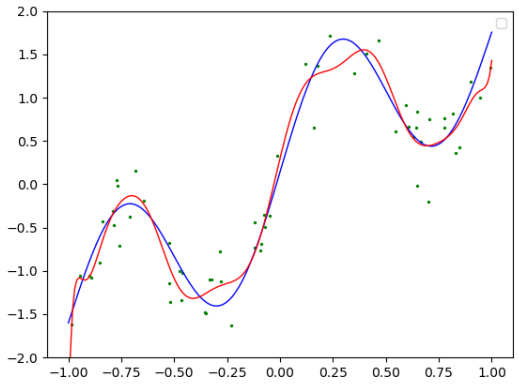
Then we would create the basis functions according to

```
class FourierFeatures(BaseEstimator, TransformerMixin):
    "Fourier Expansion for one-dimensional input"

    def __init__(self, N):
        self.N = N

    def fit(self, X, Y=None):
        return self

    def transform(self, X):
        transformed_columns = []
        transformed_columns.append(np.ones(shape=X.shape))

        for n in range(self.N):
            transformed_columns.append(np.sin(n * X))
            transformed_columns.append(np.cos(n * X))

        print(np.hstack(tuple(transformed_columns)).shape)
        return np.hstack(tuple(transformed_columns))
```

and both of them would give the following fits to our original function $f(x) = \sin(2\pi x) + 2\cos(x - 1.5)$.



(a) Fitting with 10 Gaussian basis functions.



(b) Fitting with 10 Fourier basis functions.

Figure 47

14.3.2 Standardizing Data

Standardizing typically meanss that our featuers will be rescaled to have the properties of a standard normal distribution with mean of 0 and a standard deviation of 1. Here are a few methods to scale our data, with their results shown on a dataset of 30 points in $\mathbb{R}^2$.

1. **StandardScaler**: This is probably the most used method for standardizing data. It standardizes features by removing the mean and scaling to unit variance. The standard score of a sample $x^{(n)}$ is $(x - \bar{x})/S$ where $\bar{x}$ is the mean of the training samples and S is the standard deviation of the training samples.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaled_data = scaler.fit_transform(data)
```

2. **MinMaxScaler**: While not technically "standardization," MinMaxScaler is another preprocessing method for scaling. It transforms features by scaling each feature to a given range, typically between zero and one, or so that the maximum absolute value of each feature is scaled to unit size.

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(data)
```

3. **MaxAbsScaler**: This scaler works similarly to the MinMaxScaler but scales in a way that the training data lies within the range $[-1, 1]$ by dividing through the largest maximum value in absolute value. It is meant for data that is already centered at zero or sparse data.

```
from sklearn.preprocessing import MaxAbsScaler
scaler = MaxAbsScaler()
scaled_data = scaler.fit_transform(data)
```

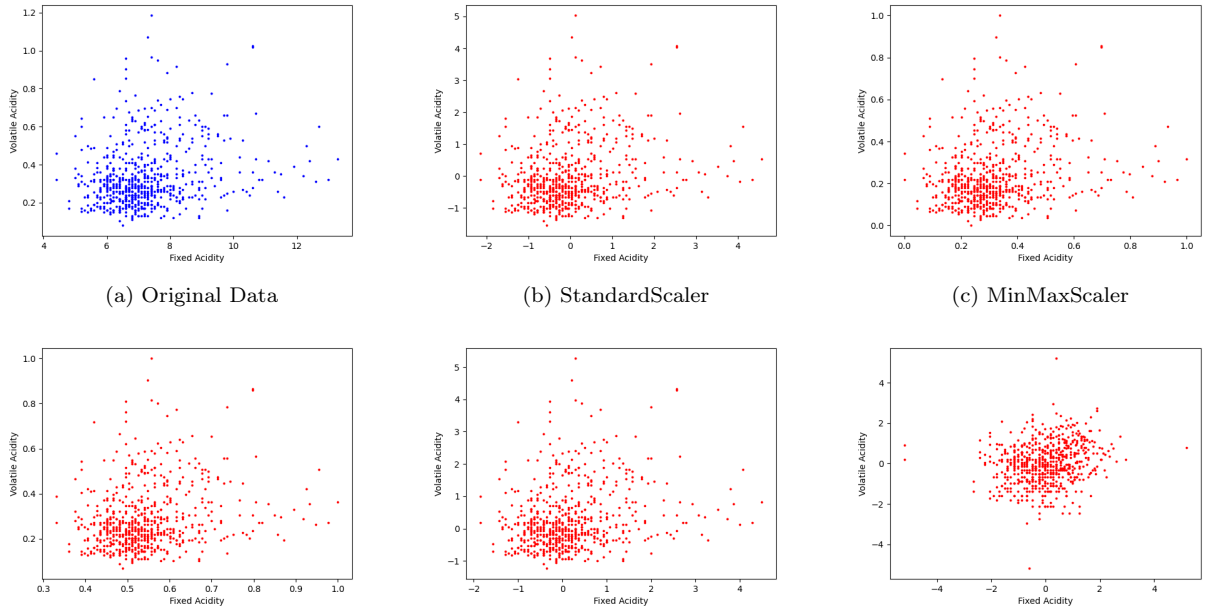
4. **RobustScaler**: This scaler removes the median and scales the data according to the quantile range (defaults to IQR: Interquartile Range). It's robust to outliers, which makes it a good choice if you have data with possible outliers.

```
from sklearn.preprocessing import RobustScaler
scaler = RobustScaler()
scaled_data = scaler.fit_transform(data)
```

5. **QuantileTransformer**: Note that the presence of outliers messes with our scaling. More generally for skewed distributions (like an exponential), a linear transformation does not take care of these outliers, so we would like some nonlinear preprocessing algorithm. One common one is the QuantileTransformer, which takes the quantiles (percentiles) of the dataset and transforms then so that those are equidistant from each other. By default, it divides up the data into 1000 quantiles.

```
from sklearn.preprocessing import QuantileTransformer
transformer = QuantileTransformer(n_quantiles = 100, output_distribution='normal')
transformed_data = transformer.fit_transform(data)
```

Let's talk about how these scalers will work on some data. We take a wine data with the two variables representing fixed acidity and volatile acidity.



(a) Original Data (b) StandardScaler (c) MinMaxScaler

(d) MaxAbsScaler (e) RobustScaler (f) QuantileTransformer

Figure 48: The StandardScaler simply standardizes the data to have 0 mean and unit variance.

It's important to note that whether you should standardize your data and how you should do it depends on the specific characteristics of your data and the machine learning algorithm you're using. For example, some algorithms, like many in deep learning, assume that all features are on the same scale. Others, like Decision Trees and Random Forests, do not require feature scaling at all.

14.4 Data Augmentation

References

[1] L. Györfi, M. Kohler, A. Krzyzak, and H. Walk. *A Distribution-Free Theory of Nonparametric Regression.* Springer Series in Statistics. Springer New York, 2002.

[2] Daniel Hsu, Sham M. Kakade, and Tong Zhang. Random design analysis of ridge regression, 2014.