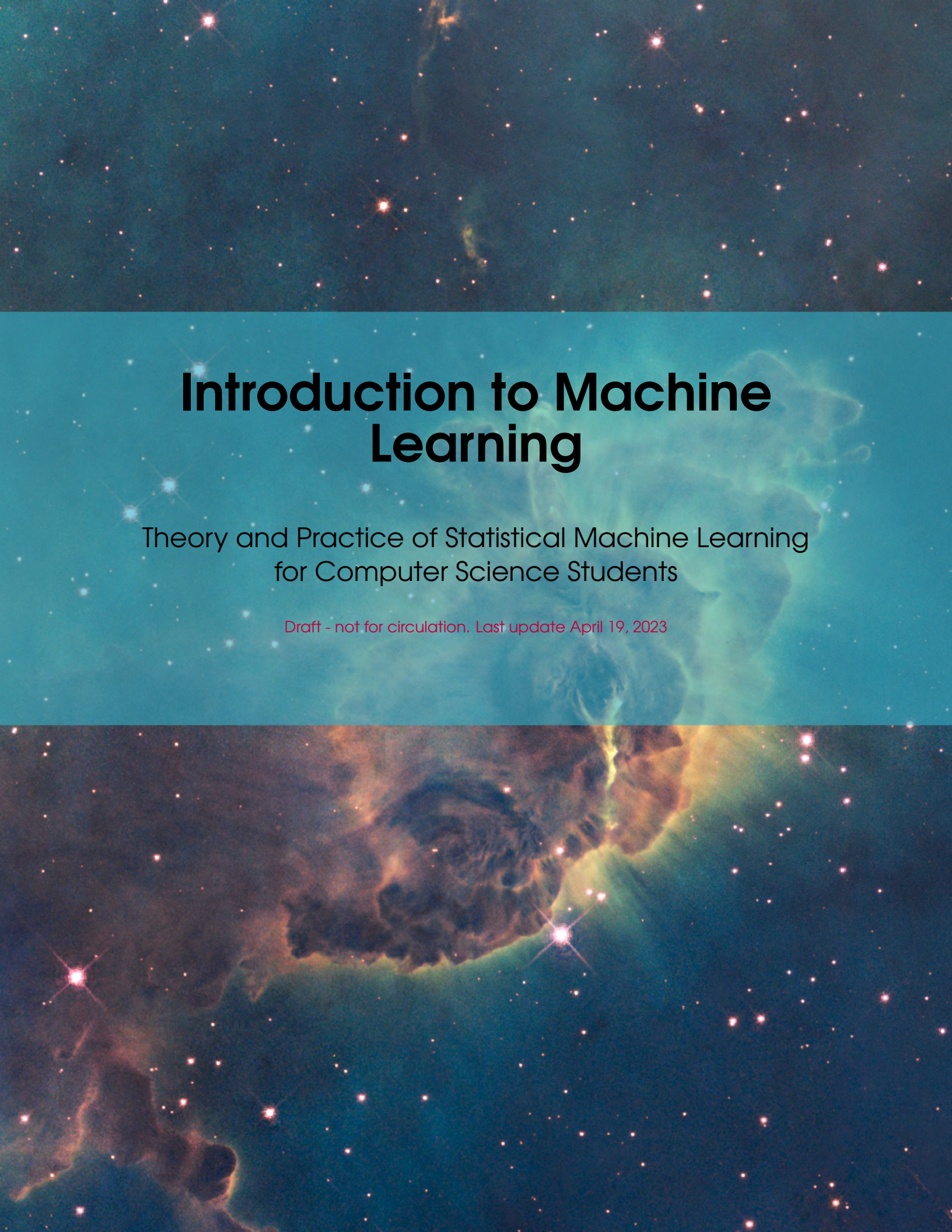


Introduction to Machine Learning

Theory and Practice of Statistical Machine Learning
for Computer Science Students

Draft - not for circulation. Last update April 19, 2023

INTRODUCTION TO MACHINE LEARNING

Matan Gavish and Gilad Green

Contents

1. Estimation Theory

"All models are wrong, but some are useful" — George E.P. Box, in *Science and Statistics*, 1976

In the study and practice of machine learning we try and define models in attempt to explain different observed phenomena, and into which we pour our understanding of the problem. These models, sometimes incorporating a probabilistic aspect, often depend on unknown parameters, whose values are to be inferred from given data.

Suppose for example we would like to know the current time. We do not have a means of telling the time ourselves, and so we ask people around us. We asked an individual and were told that the time is 13:15. Since people tend to approximate the time, it is possible that the true time is not 13:15 as told but rather 13:13. To get a better accurate assessment of the true time we decide to ask multiple individuals and record their answers. We organize the answers as *observations* denoted by $x_1,\ldots,x_m$, for m the number of individuals asked. Intuitively, we can now *estimate/approximate* the true time by averaging over the observations $\frac{1}{m}\sum x_i$.

It is helpful to consider the observations $x_1,\ldots,x_m$ as the *realization* of the *random variables* $X_1,\ldots,X_m$. These are called a *repeated sample* or simply a *sample*. By doing so, we emphasis the probabilistic nature of the problem and can devise a simplified probabilistic model for it. This model, will capture the essence of the problem by incorporating our prior knowledge and assumptions on the manner in which the observations behave (i.e. the underlying probability distribution).

Given a sample $x_1,\ldots,x_m$ for which we assume some underlying probability distribution $\mathcal{P}$, we say that $x_1,\ldots,x_m$ are *identically distributed* if they are all the realizations of random variables over the same probability distribution function, i.e. $X_1,\ldots,X_m \sim \mathcal{P}$. We further say that $x_1,\ldots,x_m$ are independent identically distributed (i.i.d) if they are all the realizations of random variables which are independent random variables and identically distributed. We denote this as $X_1,\ldots,X_m \overset{i.i.d}{\sim} \mathcal{P}$ for $X_1 = x_1,\ldots,X_m = x_m$ the realizations of $X_1,\ldots,X_m$. To simplify notation we often write $x_1,\ldots,x_m \overset{i.i.d}{\sim} \mathcal{P}$ (even though $x_1,\ldots,x_m$ are not random variables) in the sense that the corresponding random variables are i.i.d.

R Note that you will commonly see a dual usage of the word *sample*, sometimes for a specific x_i and sometimes for the whole set $x_1,\ldots,x_m$.

Probability versus Statistics

Throughout this chapter, and the rest of this book, we will be using concepts from both probability theory and from statistics. Though both are related fields of mathematics dealing with analyzing the chances of events happening, there are some key fundamental differences in the manner in which they address the matter.

The field of probability is first and foremost a theoretical study, asking the outcome of mathematical definitions. It deals with *predicting* how likely are events to occur in a given setup. Statistics on the other hand, is primarily an applied study, that attempts to explain observed phenomena in the real world. It involves the *analysis of past* events.

In the context of the time estimation problem above

- If we are to think as a probabilistic, we would notice the answers take a wide continuous range of values. We would then *assume* they follow some distribution, say Gaussian, and ask what is the probability of predicting a specific time.
- If we are to think as a statistician, though noticing the answers take a wide continuous range of values, we would be concerned with the question of "how do we assure these observations follow a Gaussian distribution?". We would then use these past observations to decide if they are indeed consistent with the Gaussian distribution assumption.

We will use both fields to tackle the different learning problems we will face.

1.1 Estimation of Distribution Parameters

Often the underlying probability distribution $\mathcal{P}$ is characterized by some set of parameters $\theta \in \Theta$. θ denotes a vector of parameters and Θ the set with all possible values for the parameters. For example, a Poisson distribution $Poisson(\lambda)$ is characterized by the parameter λ, so $\theta := \{\lambda\}$ and $\Theta := \mathbb{N}_0$. In the case of a normal distribution $\mathcal{N}(\mu,\sigma^2)$, it is characterized by μ,σ^2, so $\theta := \{\mu,\sigma^2\}$ and $\Theta := \mathbb{R}\times\mathbb{R}_+$.

In the case of (parametric) estimation theory, we assume some underlying probability distribution $\mathcal{P}(\theta)$ characterized by the parameters θ. Then, given a sample $x_1,\ldots,x_m$, drawn according to $\mathcal{P}(\theta)$ and for which we *do not* know the true value of θ, we wish to choose θ^* "best" expressing the true value of θ. The attempt to find θ^* is called *estimating* θ. We formulate this problem by defining a decision function (or decision rule) $\delta_m : \mathbb{F}^m \to \Theta$ mapping the observations to the parameter space. For simplicity we shall omit the index and write $\delta(x_1,\ldots,x_m)$. Since we can devise many different decision functions, the problem of choosing $\theta^* \in \Theta$ becomes a problem of choosing the *optimal* decision function δ from the set Δ:

$$\Delta := \{\delta(x_1,\ldots,x_m) : \mathbb{F}^m \to \Theta\} \tag{1.1}$$

where the notion of what does an optimal decision function means will be covered later. In the context of machine learning we refer to this set Δ by the term *hypothesis class* and each function in it as an *hypothesis*.

Definition 1.1.1 Let $\delta \in \Delta$ be a decision function. Then $\delta(x_1,\ldots,x_m)$ is called a point statistical estimator of a parameter θ, or simply an estimator.

Let us return to the example above of determining the current time. Given the answers of m different individuals $x_1,\ldots,x_m$ we stated that we could simply average the answers and approximate the true time as $\frac{1}{m}\sum x_i$. In fact, the function $f : \mathbb{R}^m \to \mathbb{R}$ defined by $f(x_1,\ldots,x_m) = \frac{1}{m}\sum x_i$ *is* a decision rule, and when using it over the random variables $X_1,\ldots,X_m$ (whose realizations are $x_1,\ldots,x_m$) we in fact defined an estimator.

1.1.1 Estimation of Univariate Gaussian Parameter

Recall the probability density function of the normal distribution.

Definition 1.1.2 A random variable x has a *normal distribution* with expectation μ and variance σ^2 if it has a PDF of the form:

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

In this case we write: $x \sim \mathcal{N}(\mu, \sigma^2)$

As mentioned before, in the case of a normal distribution $\theta := \{\mu, \sigma^2\}$. Consider the problem of estimating the expected value μ and the variance σ^2 for a given sample drawn i.i.d from a normal distribution characterized by μ, σ^2: $x_1, \ldots, x_m \overset{i.i.d}{\sim} \mathcal{N}(\mu, \sigma^2)$. To do so we should define decision rules that take the m observations and output an estimation of the expected value and variance. We define the estimators:

- ***Sample mean***: an estimator for the population mean

$$\hat{\mu}_X := \frac{1}{m} \sum_{i=1}^{m} x_i \tag{1.2}$$

- ***Sample variance***: an estimator for the population variance

$$\hat{\sigma}_X^2 := \frac{1}{m-1} \sum_{i=1}^{m} (x_i - \hat{\mu}_X)^2 \tag{1.3}$$

where the $\hat{}$ (hat) notation expressed that $\hat{\mu}_X, \hat{\sigma}_X^2$ are estimators of the true parameters μ, σ^2. Whenever context is clear, we will omit the subscript X. These estimators of the sample mean and sample variance are not exclusive for normal distribution, and will be used for other distributions as well.

Note that the sample variance is somewhat different from the variance of the x_i's where we divide by $m-1$ rather than by m. The factor by which we divide (m or $m-1$) is determined by what is known as the number of *degrees of freedom*. Though beyond the scope of this book, an estimator's number of degrees of freedom is the number of independent observations used for the estimation minus the number of parameters used for the estimation. In the case of the sample variance estimator (1.3) we provide m samples but use one parameter (being the estimator of the sample mean). Thus, we divide by $m-1$. The difference between the two options will be discussed shortly under bias of estimators (Definition 1.1.6).

1.1.2 Properties of Estimators

From the definition of the sample variance (1.3) we understand that we have a great deal of freedom when designing different estimators. Indeed, we might also consider the following functions as estimators of the sample variance:

$$\hat{\sigma}^2 := \frac{1}{m} \sum (x_i - \hat{\mu}), \quad \hat{\sigma}^2 := \frac{1}{m} \sum |x_i - \hat{\mu}|, \quad \hat{\sigma}^2 := \frac{1}{m-1} \sum |x_i - \hat{\mu}|$$

Therefore, the question is how to choose the "right"/"best" estimator, what does it actually mean to be the "right"/"best" estimator, and does this decision depend on the given problem? Consider for example the following scenario. Suppose $x_1, \ldots, x_m \overset{i.i.d}{\sim} \mathcal{N}(\mu, 1)$ and the two following estimators for the expectation:

$$\hat{\mu}_1(x_1, \ldots, x_m) = \frac{1}{m} \sum x_i$$
$$\hat{\mu}_2(x_1, \ldots, x_m) = 1$$

Clearly, choosing $\hat{\mu}_2$ does not seem like a smart decision as for any true value of the distribution's expectation $\mu \neq 1$ our estimation will probably be wrong. However, if the true value is indeed $\mu = 1$ then the sample mean estimator $\hat{\mu}_1$ will be out-performed by $\hat{\mu}_2$.

And so, to try and answer the questions above we need to devise ways to evaluate estimators. To do so we define two types of objects:

- Properties of estimators - these will be desired properties that an estimator might fulfill, and that will help us choose an estimator.
- A loss function - A measure to compare between the estimated values obtained by the estimator and the true value of the parameters.

Then, using these objects we can define in what sense is one estimator better than another. Usually this will be done in the form of finding an estimator that minimizes some loss function out of the set of estimators fulfilling some property.

Throughout this book we will encounter different examples where we will switch from using one estimator with another depending on what we are trying to achieve.

1.1.2.1 Loss Functions

Definition 1.1.3 A *loss function* is a function that maps an event onto a real number: $L : \Omega \to \mathbb{R}$

Intuitively, a loss function represents the *cost* associated with the event. In the context of decision theory such an event will be the estimation outputted by an estimator, and we will often want to find the estimator minimizing this cost. Here are a few examples of different loss functions that we will encounter throughout the book:

- The ℓ_{0-1} loss function defined as: $L(\hat{\theta}, \theta) := \mathbb{1}_{\hat{\theta}=\theta}$.
- The ℓ_ε loss function defined as: $L_\varepsilon(\hat{\theta}, \theta) := \mathbb{1}_{||\hat{\theta}-\theta||\leq\varepsilon}$ for some predetermined norm and $\varepsilon \geq 0$.
- The absolute-value (ℓ_1) loss function defined as: $L(\hat{\theta}, \theta) := \left|\hat{\theta} - \theta\right|$.
- The quadratic or squared-error (ℓ_2) loss function defined as: $L(\hat{\theta}, \theta) := \left(\hat{\theta} - \theta\right)^2$.

For different scenarios different loss functions are used. For example, in the case of estimating the current time, perhaps estimating the exact time is less important than being close to it. If that is the case, we might prefer using the absolute-value or squared-error loss functions. Or perhaps we are not concerned with how far is our estimation as long as it is within a certain deviance from the actual time. In such case we might prefer using the ℓ_ε loss function.

Another term associated with the notion of loss functions is the *risk function* of a decision rule.

Definition 1.1.4 Let X be a set of observations and $\delta(X)$ a decision rule for parameter θ. The *risk function* of δ and θ with respect to a loss function L is defined as the expected loss:

$$\mathcal{R}(\theta, \delta) := \mathbb{E}_X\left[L(\theta, \delta(X))\right]$$

One such risk function, which we will see in chapter 2, is the Mean Squared Error (MSE) risk function. This is the risk function with respect to the squared-error loss function $\mathcal{R}(\theta, \hat{\theta}) := \mathbb{E}_X[(\theta - \hat{\theta}(X))^2]$.

1.1.2.2 Bias, Variance & Consistency

Unbiasedness

One of the most desirable properties of an estimator is to be unbiased. That is, that on average our estimation equals to the true parameter estimated. Formally,

Definition 1.1.5 Let $\delta(x_1,\ldots,x_m)$ be an estimator for a parameter θ. The difference

$$d = \delta(x_1,\ldots,x_m) - \theta$$

is called the *error* of the estimator δ.

Definition 1.1.6 Let $\delta(x_1,\ldots,x_m)$ be an estimator for a parameter θ. The quantity

$$Bias_\theta\left[\delta(x_1,\ldots,x_m)\right] := \mathbb{E}_{x_1,\ldots,x_m|\theta}\left[d\right] = \mathbb{E}_{x_1,\ldots,x_m|\theta}\left[\delta(x_1,\ldots,x_m)\right] - \theta$$

for $x_1,\ldots,x_m|\theta$ denoting the probability of sample $x_1\ldots,x_m$ from $\mathcal{P}(\theta)$, is called the *bias* (or systematic error) of the estimator δ.

Definition 1.1.7 Let $\delta(x_1,\ldots,x_m)$ be an estimator for a parameter θ. δ is said to be *unbiased* if

$$\forall\theta\in\Theta \quad Bias_\theta\left[\delta(x_1,\ldots,x_m)\right] = 0 \text{ or equivelantly } \forall\theta\in\Theta \quad \mathbb{E}_{x_1\ldots,x_m|\theta}\left[\delta(x_1,\ldots,x_m)\right] = \theta$$

Let us revisit the previously defined sample mean (1.2) and sample variance (1.3) estimators and show that these are both unbiased estimators. Beginning with the sample mean estimator then

$$\mathbb{E}(\hat{\mu}) = \mathbb{E}\left(\frac{1}{m}\sum_{i=1}^{m}x_i\right) = \frac{1}{m}\sum_{i=1}^{m}\mathbb{E}(x_i) = \frac{1}{m}\sum_{i=1}^{m}\mathbb{E}(X) = \mathbb{E}(X) = \mu \tag{1.4}$$

where we used the linearity property of the expectation and that the samples are i.i.d (so they have the same expectation). Similarly we can show that the sample variance is unbiased:

$$\begin{aligned}\mathbb{E}(\hat{\sigma}^2) &= \mathbb{E}\left(\tfrac{1}{m-1}\textstyle\sum_{i=1}^{m}(x_i-\hat{\mu})^2\right)\\ &= \tfrac{1}{m-1}\textstyle\sum_{i=1}^{m}\mathbb{E}\left((x_i-\hat{\mu})^2\right)\\ &= \tfrac{1}{m-1}\textstyle\sum_{i=1}^{m}\mathbb{E}\left(x_i^2 - 2x_i\tfrac{1}{m}\sum_{j=1}^{m}x_j + \tfrac{1}{m^2}\sum_{k,j=1}^{m}x_kx_j\right)\end{aligned}$$

The X_i's are i.i.d which implies that $\mathbb{E}(x_i) = \mathbb{E}(X)$ and $\mathbb{E}\left(x_i^2\right) = \mathbb{E}\left(X^2\right)$ for every i, as well as that $\mathbb{E}(x_kx_j) = \mathbb{E}(x_k)\mathbb{E}(x_j) = \mathbb{E}^2(X)$ for every $k\neq j$. Substituting into the above sums we obtain that:

$$\mathbb{E}\left(\hat{\sigma}^2\right) = \mathbb{E}\left(X^2\right) - \mathbb{E}^2(X) = Var(X) = \sigma^2$$

In a similar manner we can show that the estimator of sample variance where we divide by m instead of $m-1$ is a *biased* estimator. Ex.2

Variance

Another useful property of an estimator, that provides us with an indication of how well is the estimator performing, is its variance. Since an estimator is a function of (a set of) random variables, it is itself a random variable. Therefore, the variance of an estimator is simply the variance of a random variable, where the probability space is defined over the events of obtaining a given set of samples.

Definition 1.1.8 Let $\delta(x_1,\ldots,x_m)$ be an estimator for a parameter θ. The *variance* of δ is defined as

$$Var(\delta) := \mathbb{E}_{x_1,\ldots,x_m|\theta}\left[\left(\delta(x_1,\ldots,x_m) - \mathbb{E}_{x_1,\ldots,x_m|\theta}\left[\delta(x_1,\ldots,x_m)\right]\right)^2\right]$$

where expectation is taken over the event of sampling $x_1,\ldots,x_m$ from $\mathcal{P}(\theta)$.

Let us compute the variance of the sample mean estimator (1.2). Let $x_1,\ldots,x_m$ be a set of independent random variables with identical variance σ^2. As the sample mean estimator is the mean of m random variables we

could simply:

$$\begin{aligned} Var(\hat{\mu}) &= Var\left(\frac{1}{m}\sum x_i\right) \\ &= \frac{1}{m^2}Var\left(\sum x_i\right) \\ &\overset{(*)}{=} \frac{1}{m^2}\sum Var(x_i) \\ &\overset{(**)}{=} \frac{1}{m^2}\cdot m\cdot\sigma^2 \\ &= \frac{\sigma^2}{m} \end{aligned}$$

where we used the assumption that the samples are independent $(*)$ and with identical variance $(**)$.

Consistency

Going back to the task of determining a given time where we defined our estimator as the sample mean, it seems intuitive that the more individuals asked the more accurate (i.e closer to the true value) our estimation should be. Let us formulate this as a property an estimator might fulfill.

Definition 1.1.9 A sequence $\{Z_n\}$ of random variables is said to *converge in probability* to a random variable Z if

$$\forall \varepsilon > 0 \ \lim_{n\to\infty} \mathbb{P}(|Z_n - Z| > \varepsilon) = 0$$

and is denoted by $Z_n \xrightarrow{p} Z$.

Definition 1.1.10 An estimator $\hat{\theta}_n(x_1,\ldots,x_n)$ of parameter θ is said to be *consistent* if it converges in probability to the true value of θ:

$$\forall \varepsilon > 0 \ \lim_{n\to\infty} \mathbb{P}\left(\left|\hat{\theta}_n - \theta\right| > \varepsilon\right) = 0$$

Let us understand these definitions as it is to be applied for an estimator. Since an estimator is a function of the random variables $x_1,\ldots,x_m \sim \mathcal{P}$ it is itself a random variable. Consider a set of estimators (random variables) $\{\hat{\theta}_n\}_{n=1}^{\mathbb{N}}$, each calculated over the first n samples. In addition, denote θ the constant random variable valuing as the true parameter estimated. Therefore, $\hat{\theta}_n \xrightarrow{p} \theta$ means that as we increase the sample size n, the probability of our estimation deviating from the true parameter by more than a fixed amount ε tends to zero. It can be shown that the sample mean has the property of being a consistent estimator.

In this section we have discussed the first and second moments of an estimator (first moment in the form of an estimator's bias). Being a random variable, we can also discuss the distribution of a random variable, as seen shortly in Lab 01 - Data Simulation and Sampling.

1.1.3 The Maximum Likelihood Estimator

And so, we can now define a criteria by which we define what does it mean for an estimator to be "optimal". For example, for a set of estimators Δ, we can decide that the optimal estimator is one that is *unbiased* and that achieves the minimal variance out of all estimators in Δ. This approach is known as the Minimum Variance Unbiased (MVU). This is a logical decision. Unbiased estimators are right "on average", and having a small as possible variance means we are often not far from this average. Notice also that by this definition there could be more than a single "optimal" estimator in Δ.

Another approach is to look for the *Maximum Likelihood Estimator*. This approach suggests choosing the estimator under which the observed data was *most likely*. Suppose we obtained the observations $x_1,\ldots,x_m \overset{i.i.d}{\sim} \mathcal{P}(\theta)$. For the probability distribution function $\mathcal{P}$ we define the likelihood function.

Definition 1.1.11 Let X be a random variable following some probability distribution $\mathcal{P}$ with a density function f that depends on a parameter $\theta \in \Theta$. The *likelihood* function is

$$\mathcal{L}(\theta|x) := f_\theta(x)$$

for x the realization of X.

For example, consider the case of the univariate Gaussian distibution seen above (1.1.2). Then, for $\theta = \{\mu, \sigma^2\}$ the likelihood function under $\mathcal{P} = \mathcal{N}(\mu, \sigma^2)$ is:

$$\mathcal{L}(\theta|x_i) = f_\theta(x_i) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x_i-\mu)^2}{2\sigma^2}\right), \quad i \in [m] \tag{1.5}$$

In the case where the samples $x_1, \ldots, x_m$ are drawn i.i.d the likelihood is:

$$\begin{aligned} \mathcal{L}(\theta|x_1,\ldots,x_m) &= f_\theta(x_1,\ldots,x_m) \\ &= \prod_{i=1}^m f_\theta(x_i) \\ &= \frac{1}{\left(2\pi\sigma^2\right)^{m/2}} \exp\left(-\frac{1}{2\sigma^2}\sum(x_i-\mu)^2\right) \end{aligned} \tag{1.6}$$

Using the likelihood function derived for the univariate Gaussian distribution we can now provide each θ with a quantity (the likelihood) of how likely is it to have generated the observed data.

■ **Example 1.1** Let $x_1, x_2, x_3, x_4 \overset{i.i.d}{\sim} \mathcal{N}(\mu, \sigma^2)$ where $x_1 = -1, x_2 = 0, x_3 = 0, x_4 = 1$ and $\sigma^2 = 1$. Let us calculate the likelihood for:

- $\mu = 0$:

$$\begin{aligned} \mathcal{L}(\mu = 0, \sigma^2 = 1|x_1, x_2, x_3, x_4) &= \frac{1}{(2\pi)^2} \exp(-\frac{1}{2}\sum_{i=1}^4 x_i^2) \\ &= \frac{1}{4\pi^2} \exp(-\frac{2}{2}) \\ &\approx 0.00931 \end{aligned}$$

- $\mu = 1$:

$$\begin{aligned} \mathcal{L}(\mu = 1, \sigma^2 = 1|x_1, x_2, x_3, x_4) &= \frac{1}{(2\pi)^2} \exp(-\frac{1}{2}\sum_{i=1}^4 (x_i-1)^2) \\ &= \frac{1}{4\pi^2} \exp(-\frac{2^2+1+1}{2}) \\ &\approx 0.00126 \end{aligned}$$

Hence, given x_1, x_2, x_3, x_4, it is more *likely* that the real value of μ is 0 than 1. ■

And so, given the likelihood function we define the Maximum Likelihood Estimator (MLE) as $\theta \in \Theta$ that maximizes the likelihood function. Formally:

Definition 1.1.12 Let $\mathcal{L}$ be the likelihood function of some probability distribution $\mathcal{P}$ characterized by $\theta \in \Theta$. Let $X \sim \mathcal{P}(\theta)$ be a random variable and x its realization. The *Maximum Likelihood Estimator* (MLE) for θ is

$$\hat{\theta}^{MLE} := \underset{\theta \in \Theta}{\operatorname{argmax}} \ \mathcal{L}(\theta|x)$$

Let us derive the MLE for the expectation of a univariate Gaussian distribution when σ^2 is known. Let $x_1, \ldots, x_m \overset{i.i.d}{\sim} \mathcal{N}(\mu, \sigma^2)$. So:

$$\hat{\mu}^{MLE} = \underset{\mu \in \mathbb{R}}{\operatorname{argmax}} \ \mathcal{L}(\mu|x_1, \ldots, x_m, \sigma^2)$$

Since the logarithm function is a monotonous increasing transformation the maximizer of the likelihood function is also the maximizer of the *log-likelihood*. Thus:

$$\begin{aligned}
\hat{\mu}^{MLE} &= \underset{\mu\in\mathbb{R}}{\text{argmax}}\ \mathcal{L}(\mu|x_1,\ldots,x_m,\sigma^2) \\
&= \underset{\mu\in\mathbb{R}}{\text{argmax}}\ \log\mathcal{L}(\mu|x_1,\ldots,x_m,\sigma^2) \\
&= \underset{\mu\in\mathbb{R}}{\text{argmax}}\ \log\left(\prod_{i=1}^{m}\frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(-\frac{(x_i-\mu)^2}{2\sigma^2}\right)\right) \\
&= \underset{\mu\in\mathbb{R}}{\text{argmax}}\ \log\left(\frac{1}{\left(2\pi\sigma^2\right)^{m/2}}\exp\left(-\frac{1}{2\sigma^2}\sum\left(x_i-\mu\right)^2\right)\right) \\
&= \underset{\mu\in\mathbb{R}}{\text{argmax}}\ \log\left(\exp\left(-\frac{1}{2\sigma^2}\sum\left(x_i-\mu\right)^2\right)\right) \\
&= \underset{\mu\in\mathbb{R}}{\text{argmax}}\ -\sum_{i=1}^{m}\left(x_i-\mu\right)^2
\end{aligned} \tag{1.7}$$

And so, to find the maximizer, we calculate the derivative with respect to μ and equate to zero:

$$\begin{gathered}
\frac{\partial}{\partial\mu}\left(-\sum_{i=1}^{m}\left(x_i-\mu\right)^2\right) = -\sum_{i=1}^{m}\frac{\partial(x_i-\mu)^2}{\partial\mu} = \sum_{i=1}^{m}2\left(x_i-\mu\right) = 0 \\
\Downarrow \\
\hat{\mu}^{MLE} = \frac{1}{m}\sum_{i=1}^{m}x_i
\end{gathered} \tag{1.8}$$

Notice that when we previously used the sample mean estimator (1.2) we have therefore actually used the maximum likelihood estimator for the expectation. Similarly, we can derive that the sample variance defined above is the MLE of the variance.

1.1.3.1 Beyond Maximum Likelihood - A Bayesian Approach

Suppose we are facing the task of estimating average human height. We observe $x_1,\ldots,x_m$ a set of i.i.d samples reflecting the heights of m individuals. In order to estimate the average human height $\hat{\theta}$, we must first specify a principle by which we select an estimator.

In the section above we defined the likelihood function $\mathcal{L}(\theta|x)$ and the strategy of selecting the estimator that maximizes the likelihood function - the MLE. Let us assume that $x|\theta$ follows a normal distribution. If our sample is $x_1 = 2m, x_2 = 2.03m, x_3 = 1.95m, x_4 = 2.1m, x_5 = 1.65m$ then the MLE is the sample mean: $\hat{\theta}^{MLE} = 1.94m$. This however seems to be an odd estimation. Even without looking at our observations it seems logical that over the general population the average human height should probably be lower than 1.9m and perhaps higher than 1.5m. This seems logical as we have some additional knowledge (a prior belief) on the problem. Therefore, the question is how could we include this additional knowledge in our estimation?

Let us go back to the likelihood function. We defined it as the PDF of an observation x under a probability distribution characterized by θ. We interpret $\mathcal{L}(\theta|x)$ as "how likely is θ "given" the observation x". The term "given" is itself interpreted as "while fixing the observation x" and not in the sense of probability theory. We could however further *assume* that both X and θ are random variables which have some *joint probability distribution* function $f_{X,\Theta}(x,\theta)$. By doing so we can derive a new criteria for selecting an estimator. Under such assumption of a joint probability distribution $f_{X,\Theta}(x,\theta)$, the expression $x|\theta$ can be understood through the conditional distribution:

$$f_{X|\Theta=\theta}(x) = \frac{f_{X,\Theta}(x,\theta)}{f_{\Theta}(\theta)} \tag{1.9}$$

Same as before, we can choose the MLE as $\hat{\theta}^{MLE}$ that maximizes $f_{X|\Theta=\theta}(x)$.

Now, by applying Bayes' theorem of conditional distributions we can express the conditional $\Theta|X$ as:

$$\overbrace{f_{\Theta|X=x}(\theta)}^{posterior} = \frac{\overbrace{f_{X|\Theta=\theta}(x)}^{likelihood} \cdot \overbrace{f_\Theta(\theta)}^{prior}}{\underbrace{f_X(x)}_{evidence}} \tag{1.10}$$

where $f_{X|\Theta=\theta}(x)$ is the likelihood function, $f_\Theta(\theta)$ is the marginal distribution of Θ and $f_X(x)$ the marginal distribution of X functioning as a normalization factor. The marginal distribution f_Θ reflects our *belief* regarding the true value of θ *before* (i.e *prior*) observing the data. Therefore, the *A-Posteriori* distribution $f_{\Theta|X=x}$, i.e the conditional of θ *after* observing the data, is the weighting of the likelihood function by our prior belief and the evidence of the data.

By using the A-Posteriori distribution, we can derive the Maximum A-Posteriori estimator (MAP):

$$\hat{\theta}^{MAP} := \underset{\theta\in\Theta}{\text{argmax}}\, f_{\Theta|X=x}(\theta) = \underset{\theta\in\Theta}{\text{argmax}}\, \frac{f_{X|\Theta=\theta}(x)\cdot f_\Theta(\theta)}{f_X(x)} = \underset{\theta\in\Theta}{\text{argmax}}\, f_{X|\Theta=\theta}(x)\cdot f_\Theta(\theta) \tag{1.11}$$

Namely, the estimator that takes into account both what is observed (the likelihood) and the prior belief.

Returning to the task of estimating human heights, let us further assume that the average human height follows a normal distribution centered around 1.7m with a variance of 0.01m. Therefore we assume that:

$$\theta \sim \mathcal{N}(1.7, 0.01) \quad (1)$$
$$x_i|\theta \overset{i.i.d}{\sim} \mathcal{N}(\theta, \sigma^2)\ \forall i \quad (2)$$

where we set σ^2 as $(\hat{\sigma}^2)^{MLE} \approx 0.17$. Under these assumptions and the sample above $x_1 = 2m, x_2 = 2.03m, x_3 = 1.95m, x_4 = 2.1m, x_5 = 1.65m$ we can now evaluate different values of θ:

- For $\theta = 1.94$ (the likelihood maximizer) we find that:

$$\log\left(f_{X|\theta=1.94,\sigma^2=0.17}(x_1,\ldots,x_5)\cdot f_\Theta(1.94)\right) = \log\left(f_\Theta(1.94)\cdot\prod_i f_{\theta=1.94,\sigma^2=0.17}(x_i)\right) = 1.264$$

- For $\theta = 1.7$ (the prior's expectation) we find that:

$$\log\left(f_{X|\theta=1.7,\sigma^2=0.17}(x_1,\ldots,x_5)\cdot f_\Theta(1.94)\right) = \log\left(f_\Theta(1.7)\cdot\prod_i f_{\theta=1.7,\sigma^2=0.17}(x_i)\right) = 1.442$$

Thus, under this prior we see that the MLE does not achieve the maximal posterior probability.

In general, the field of statistical learning splits between these two approaches: frequentistic methods which aim to express and maximize the likelihood function, and Bayesian methods which assume some prior distribution and derive estimators that use both the likelihood and the prior (one of these estimators is the MAP estimator seen above). It can be shown that when we correctly assume a prior distribution the Bayesian estimators are superior to the MLE. The challenge however with the Bayesian methods lies precisely in how to obtain such prior. Though this book focuses on frequentistic methods, we shortly discuss some Bayesian methods in section 3.5.

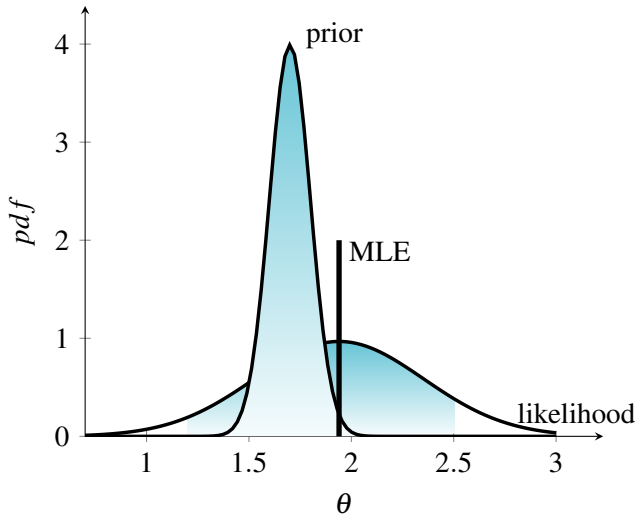


Figure 1.1: *Likelihood and prior distributions for human heights example*

1.1.4 Measures of Concentration For Estimation Tasks

Occasionally, providing an estimation for some parameter of interest is not sufficient, and we are further interested in providing some bounds describing how "good"/"close" is our estimation to the true value. Consider for example the following scenario. Suppose we have a coin for which we want to estimate its bias, that is with what probability p flipping the coin will result with "Heads" being up and with probability $1-p$ "Tails" being up. Formally, we can think (i.e. model) a single coin flip as a Bernoulli random variable X which takes the value of 1 for "Heads" or 0 for "Tails".

We shall denote the probability distribution of X by $\mathcal{D}_p$ such that:

$$\mathcal{D}_p(X) = \begin{cases} p & X = 1 \\ 1-p & X = 0 \end{cases}, \quad p \in [0,1] \tag{1.12}$$

To estimate the value of p we take a sequence of m tosses of the coin $X_1,\ldots,X_m \overset{i.i.d}{\sim} Ber(p)$ and denote the results of the tosses (i.e. the samples) by $S = \{x_1,\ldots,x_m\}$ where x_i refers to the result of the random variable X_i. Lastly, we denote the probability distribution of the m coin tosses by $\mathcal{D}_p^m$, and therefore the probability of obtaining a specific sample S by $\mathcal{D}_p^m(S)$. To simplify notation we will omit the writing of p and write $\mathcal{D}^m(S)$.

Now, given such sample we would like to devise a *learning algorithm*, $\mathcal{A}$, to estimate/predict the true value of p. So, a coin prediction learning algorithm is a procedure which takes as input a sample $S = \{x_1,\ldots,x_m\}$, drawn according to $\mathcal{D}^m$, and outputs estimation of p. This estimation is denoted by $\mathcal{A}(S)$ or $\hat{p}(S)$ or simply $\hat{p}$. A straightforward strategy (i.e. algorithm) is to estimate p simply by calculating the empirical proportion of "Heads" (ones):

$$\hat{p}(S) = \tfrac{1}{m}\sum x_i \tag{1.13}$$

It can be shown that this estimator is the MLE of the problem. Notice, that as we have shown earlier, the empirical mean is an unbiased estimator for the expectation (1.4). Namely, if we sample many sets of samples $S_i = \left\{x_1^{(i)},\ldots,x_m^{(i)}\right\}$ and for each run our algorithm outputs $\hat{p}_i$, then the expected value over all the $\hat{p}_i$s is p. Formally, $\mathbb{E}_S[\hat{p}(S)] = p$. This clearly seems like a desirable property for our algorithm. However, since we seldomly have many different sets of samples and as a given sample S is finite, we do not expect our estimation of $\hat{p}$ to be exact. Instead we acknowledge that the algorithm might yield a $\hat{p}$ which satisfies $|\hat{p}-p| \leq \varepsilon$, for

some $\varepsilon \in (0,1)$ which is called the *accuracy* parameter. That is, our algorithm might not return exactly p, but it returns a result $\hat{p}$ which is "close enough".

Even then, unless p equals 1 or 0, there is always *some* chance that the drawn sample would be highly non-representative. We might for example, though unlikely, end up with a sample of all "Heads" or all "Tails" regardless to the true value of p. So it is impossible to obtain a guarantee that $|\hat{p}-p| \leq \varepsilon$ holds with absolute certainty. Hence, we introduce a *confidence* parameter $\delta \in (0,1)$, and require that the event $\{S : |\hat{p}-p| > \varepsilon\}$ occurs with a probability of at most δ. In other words, we require our algorithm to be such that the probability of flipping the coin m times and obtaining a sequence S that causes it to produce an inaccurate estimation $\hat{p}(S)$ ('inaccurate' meaning $|\hat{p}-p| > \varepsilon$), is smaller or equal to δ.

Intuitively, the larger the number of flips, the more information we have about the coin and the better chance we have to satisfy the accuracy and confidence requirements. Since the accuracy and confidence parameters ε and δ are fixed, there should be some finite number of flips $m_{\mathcal{A}}$ (which depends on ε, δ and $\mathcal{A}$) such that for any sample of size $m \geq m_{\mathcal{A}}$ our algorithm satisfies the above accuracy and confidence requirements. The function that given ε, δ returns $m_{\mathcal{A}}$ such that accuracy and confidence requirements are met is called the *sample complexity* function.

Measure Concentration Using Markov's Inequality

And so, we would like to ask "How good is our learning algorithm"? We already know that on average our learning algorithm will output the correct answer. In addition, as the empirical mean is a consistent estimator we know that the estimation will become more accurate as the number of samples m increases. What we do not know is how many samples do we need, and can we somehow bound the probability of being inaccurate (i.e. the confidence) by some function depending on the number of samples. To do so we begin with using the most basic measure of concentration by applying Markov's inequality.

Theorem 1.1.1 — Markov's Inequality. Let X be a non-negative random variable with a finite expectation $\mathbb{E}[X]$. For any real value $a > 0$:

$$\mathbb{P}(X \geq a) \leq \frac{\mathbb{E}[X]}{a}$$

Proof. Let $f(x)$ be the density function of x. Since X is non-negative so $f(x) = 0$ for $x < 0$. Thus,

$$\frac{\mathbb{E}[X]}{a} = \frac{1}{a}\int_0^\infty f(x)x\,dx \geq \frac{1}{a}\int_{x=a}^\infty f(x)x\,dx \geq \frac{1}{a}\int_{x=a}^\infty f(x)a\,dx = \mathbb{P}(X \geq a)$$

■

Notice that $|\hat{p}-p|$ is a non-negative random variable. Therefore, by fixing some accuracy level $\varepsilon \in (0,1)$, we can bound from above the probability of $\hat{p}$ deviating from p by more than ε:

$$\mathcal{D}^m[|\hat{p}-p| \geq \varepsilon] \leq \frac{\mathbb{E}[|\hat{p}-p|]}{\varepsilon}$$

Recall the definition of variance $Var(A) = \mathbb{E}\left[A^2\right] - \mathbb{E}^2[A]$. Since variance is non-negative, we can conclude

that $\mathbb{E}^2[A] \leq \mathbb{E}\left[A^2\right]$. So the expectation can be bounded from above by:

$$\begin{array}{rcccc} \mathbb{E}^2[|\hat{p}-p|] & \leq & \mathbb{E}\left[|\hat{p}-p|^2\right] & = & \mathbb{E}\left[(\hat{p}-p)^2\right] \\ & \overset{unbiased}{=} & \mathbb{E}\left[(\hat{p}-\mathbb{E}[\hat{p}])^2\right] & = & Var(\hat{p}) \\ & = & Var\left(\frac{1}{m}\sum x_i\right) & = & \frac{1}{m^2}Var\left(\sum x_i\right) \\ & \overset{i.i.d}{=} & \frac{1}{m^2}m \cdot Var(x_1) & = & \frac{p(1-p)}{m} \\ & \leq & \frac{1}{4m} & & \end{array}$$

$$\Downarrow$$

$$\mathbb{E}[|\hat{p}-p|] \leq \frac{1}{\sqrt{4m}}$$

where we used the fact that the variance of a Bernoulli random variable is $p(1-p) \leq 1/4$. Put together:

$$\mathcal{D}^m[|\hat{p}-p| \geq \varepsilon] \leq \frac{\mathbb{E}[|\hat{p}-p|]}{\varepsilon} \leq \frac{1}{\sqrt{4m\varepsilon^2}} \tag{1.14}$$

Thus we have obtained a bound on the confidence for a given accuracy, as a function of the number of samples. We can now express the above as follows. If we select m to be $m \geq \left\lceil \frac{1}{4\delta^2\varepsilon^2} \right\rceil$ then the right-hand side ($\frac{1}{\sqrt{4m\varepsilon^2}}$) is smaller or equals to δ. So, for any $\varepsilon, \delta \in (0,1)$, if we sample $m_{\mathcal{A}}(\varepsilon,\delta) \geq \left\lceil \frac{1}{4\delta^2\varepsilon^2} \right\rceil$ samples then this learning algorithm achieves:

$$\mathcal{D}_p^m[|\hat{p}(S)-p| \geq \varepsilon] \leq \delta$$

Namely, the algorithm is accurate enough (as specified by ε) with a high enough confidence (as specified by $1-\delta$).

Measure Concentration Using Chebyshev's Inequality

We can improve the upper bound seen in (1.14) (i.e find a sample complexity function that will need less samples) by using Chebyshev's inequality.

Theorem 1.1.2 — Chebyshev's Inequality. Let X be a random variable with a finite mean $\mathbb{E}[X]$ and variance $Var(X)$. Then, for every $\varepsilon > 0$:

$$\mathbb{P}[|X-\mathbb{E}[X]| \geq \varepsilon] \leq \frac{Var(X)}{\varepsilon^2}$$

Proof. Consider the random variable $Y = (X-\mathbb{E}[X])^2$. This is a non-negative random variable and as such we can apply Markov's inequality over it. We obtain that $\mathbb{P}\left[(X-\mathbb{E}[X])^2 \geq \varepsilon^2\right] \leq \frac{Var(X)}{\varepsilon^2}$. To conclude the proof, observe that $\mathbb{P}[|X-\mathbb{E}[X]| \geq \varepsilon] = \mathbb{P}\left[(X-\mathbb{E}[X])^2 \geq \varepsilon^2\right]$. ■

So, given a sample $x_1, \ldots, x_m \overset{i.i.d}{\sim} Ber(p)$ we can now bound the deviance of our estimator $\hat{p}$ from its expected value as follows:

$$\begin{array}{rcccc} \mathbb{P}[|\hat{p}-\mathbb{E}[\hat{p}]| \geq \varepsilon] & \overset{unbiased}{=} & \mathbb{P}[|\hat{p}-p| \geq \varepsilon] & \overset{Chebyshev}{\leq} & \frac{Var(\hat{p})}{\varepsilon^2} \\ & = & \frac{1}{\varepsilon^2}Var\left(\frac{1}{m}\sum x_i\right) & \overset{i.i.d}{=} & \frac{1}{\varepsilon^2 \cdot m^2}\sum Var(x_i) \\ & = & \frac{p(1-p)}{m\varepsilon^2} & \leq & \frac{1}{4m\varepsilon^2} \end{array}$$

The bound obtained using Chebyshev's inequality tends to zero as $\frac{1}{m}$ while the one obtained from Markov's inequality tends to zero as $\frac{1}{\sqrt{m}}$. By solving for m $\delta = \frac{1}{4m\varepsilon^2}$ we derive a tighter bound for the sample complexity, $m_{\mathcal{A}}(\varepsilon,\delta) \geq \left\lceil \frac{1}{4\delta\varepsilon^2} \right\rceil$.

Measure Concentration Using Hoeffding's Inequality

A natural question which arises is whether the obtained bound is optimal (tight). Indeed, we can further improve the bound by exploiting the fact that our random variable not only has a finite variance, but it is also bounded between 0 and 1. For this end we use Hoeffding's inequality for the average of independent and bounded random variables.

Theorem 1.1.3 — Hoeffding's inequality. Let $X_1, \ldots X_m$ be independent and bounded random variables with $a_i \leq X_i \leq b_i$. Let $\overline{X} = \frac{1}{m}\sum_{i=1}^{m} X_i$. Then,

$$\mathbb{P}\left[\left|\overline{X} - \mathbb{E}\left[\overline{X}\right]\right| \geq \varepsilon\right] \leq 2\exp\left(\frac{-2m^2\varepsilon^2}{\sum_{i=1}^{m}(b_i - a_i)^2}\right)$$

Corollary 1.1.4 Let $X_1, \ldots X_m$ be a sequence of m i.i.d random variables, each with an expectation value $\mathbb{E}[X]$ and all of which are bounded: $a \leq X_i \leq b$. Denote $\overline{X} = \frac{1}{m}\sum_{i=1}^{m} X_i$ then:

$$\mathbb{P}\left[\left|\overline{X} - \mathbb{E}[X]\right| \geq \varepsilon\right] \leq 2\exp\left(\frac{-2m\varepsilon^2}{(b-a)^2}\right)$$

Proof. Notice that as $X_1, \ldots, X_m$ all share the same expectation and bounds then

$$\mathbb{E}\left[\overline{X}\right] = \frac{1}{m}\sum_{i=1}^{m}\mathbb{E}[X] = \mathbb{E}[X], \quad \sum_{i=1}^{m}(b_i - a_i)^2 = m \cdot (b-a)^2$$

By placing these expressions in Lemma 1.1.3 we conclude the inequality. ■

By applying Hoeffding's inequality to the case of the coin prediction problem for a sample of size m, we obtain that $\mathcal{D}_p^m\left[|\hat{p} - p| \geq \varepsilon\right] \leq 2\exp\left(-2m\varepsilon^2\right)$. Therefore, by using Hoeffding's inequality we are able to get a bound which converges exponentially in m. By taking $m_{\mathcal{A}}(\varepsilon, \delta) \geq \left\lceil \frac{1}{2\varepsilon^2} \cdot \log\left(\frac{2}{\delta}\right) \right\rceil$ samples we obtain that this probability is bound above by δ as required.

1.2 Multivariate Distributions

Up to now, we only dealt with random variables taking a single value. However, we often face a situation where an observation consists of multiple properties. As different properties may (or may not) influence one another we wish to define how the set of properties jointly behaves.

■ **Example 1.2** Consider the question of describing human weight and height. It is possible to model (i.e. describe how the data behaves) each of these properties independently using some distribution. Suppose $w_1, \ldots, w_m$ are the weights (in kilograms) of m individuals and $h_1 \ldots, h_m$ are the heights (in centimeters) of the same individuals. Let us assume that the weights and heights each follow some normal distribution:

$$w_1, \ldots, w_m \overset{i.i.d}{\sim} \mathcal{N}(75, 3)$$
$$h_1, \ldots, h_m \overset{i.i.d}{\sim} \mathcal{N}(170, 5)$$

However, we know that the properties of weight and height are linked. There exists (in the data) a connection between the two such that, in general, the taller an individual is the higher the weight. As such, we would prefer to model the data using a *joint distribution* that describes the pairs of weights and heights $(w_1, h_1), \ldots, (w_m, h_m)$.
■

Definition 1.2.1 Given random variables $X_1,\ldots,X_d$, that are defined on a probability space, the *joint probability distribution* for $X_1,\ldots,X_d$ is a probability distribution that gives the probability that each of $X_1,\ldots,X_d$ falls in any particular range (for continuous RVs) or discrete set (for discrete RVs) of values specified for that variable.

Definition 1.2.2 Two random variables X_1 and X_2 are *jointly continuous* if there exists a non-negative function $f_{X_1,X_2}:\mathbb{R}^2\to\mathbb{R}$, such that, for any set $A\in\mathbb{R}^2$, it holds that

$$\mathcal{D}\left((x_1,x_2)\in A\right)=\int_A f_{X_1,X_2}(x_1,x_2)\,dx_1dx_2$$

The function f_{X_1,X_2} is called the *joint probability density function (JPDF)* of X_1 and X_2. Often, if the context is clear, one omits the subscripts of f and simply writes $f(x_1,x_2)$.

Definition 1.2.3 A (column) *random vector* $X\coloneqq(X_1,\ldots,X_d)^\top$, is a finite collection of random variables, denoted $X_1,\ldots,X_d$, defined on a common probability space $(\Omega,\mathcal{D})$.

Returning to the task of describing a distribution of human weights and heights we can now denote the observations using random vectors $\mathbf{x}_1,\ldots,\mathbf{x}_m\in\mathbb{R}^d$ for $d=2$. We could then consider how does a specific feature manifest in the sample. For the i'th property, $\mathbf{x}_1(i),\ldots,\mathbf{x}_m(i)$ are the realizations random variables $X_1^{(i)},\ldots,X_m^{(i)}$. Another form of writing $\mathbf{x}_1(i),\ldots,\mathbf{x}_m(i)$ is by using two indices, the first standing for the sample's index and the second for the property $x_{1,i},\ldots,x_{m,i}$.

When dealing with more than a single random variable we can ask how do the different variables jointly vary. For two jointly distributed real-valued random variables X,Y with finite second moments, the covariance between X and Y is $cov(X,Y)=\mathbb{E}[X-\mathbb{E}[X]]\,\mathbb{E}[Y-\mathbb{E}[Y]]$. The covariance between a random variable and itself is $cov(X,X)=\mathbb{E}[X-\mathbb{E}[X]]^2=Var(X)$. The covariance $cov(X,Y)$ sometimes denoted as $\sigma(X,Y)$. We arrange the covariances between d different random variables in the form of a *covariance matrix*

Definition 1.2.4 Let $X\coloneqq(X_1,\ldots,X_d)^\top$ be a random vector. The *Covariance Matrix* Σ is the $d\times d$ matrix whose (i,j) entry is the covariance $\Sigma_{ij}\coloneqq\sigma(X_i,X_j)$:

$$\Sigma\coloneqq\begin{pmatrix}\mathbb{E}\left[(X_1-\mathbb{E}[X_1])(X_1-\mathbb{E}[X_1])\right] & \ldots & \mathbb{E}\left[(X_1-\mathbb{E}[X_1])(X_d-\mathbb{E}[X_d])\right]\\ \vdots & \ddots & \vdots\\ \mathbb{E}\left[(X_d-\mathbb{E}[X_d])(X_1-\mathbb{E}[X_1])\right] & \ldots & \mathbb{E}\left[(X_d-\mathbb{E}[X_d])(X_d-\mathbb{E}[X_d])\right]\end{pmatrix}$$

- In matrix notation we can express the covariance matrix as:

$$\Sigma\coloneqq\mathbb{E}\left[(X-\mathbb{E}[X])(X-\mathbb{E}[X])^\top\right]$$

- The diagonal elements of Σ are $\sigma(X_i,X_i)\equiv\sigma_{X_i}^2\equiv Var(X_i)$. Σ is a symmetric positive semi-definite matrix.

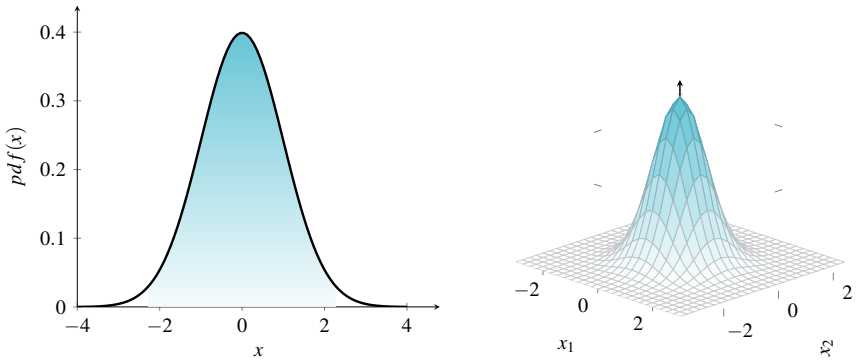
Now, using the notion of random vectors, joint distributions and covariance matrix we can define *multivariate normal distributions*.

Definition 1.2.5 A random vector $X\coloneqq(X_1,\ldots,X_d)^\top$ has a *multivariate normal distribution* with expec-

tation $\mu \in \mathbb{R}^d$ and covariance matrix $\Sigma \in \mathbb{R}^{d\times d}$ if it has a joint PDF of the form:

$$f(X) = \frac{1}{\sqrt{(2\pi)^d |\Sigma|}} \exp\left\{-\frac{1}{2}(X-\mu)^\top \Sigma^{-1}(X-\mu)\right\}$$

In this case we write: $X \sim \mathcal{N}(\mu, \Sigma)$



(a) *Univariate Gaussian* $\mathcal{N}(0,1)$ ***(b)*** *Bivariate Gaussian* $\mathcal{N}(0, I_2)$

Figure 1.2: *Visualization of Uni- and Bivariate standard normal distributions*

Observe that Definition 1.2.5 is a generalization of Definition 1.1.2, i.e. when $d = 1$ both definitions are same. In the case of modeling human weight and height suppose the covariance between the properties is 0.2. We then denote the bivariate normal distribution and the samples drawn from it by:

$$\mathbf{x}_1, \ldots, \mathbf{x}_m \overset{i.i.d}{\sim} \mathcal{N}\left(\begin{bmatrix} 75 \\ 170 \end{bmatrix}, \begin{bmatrix} 3 & 0.2 \\ 0.2 & 5 \end{bmatrix}\right)$$

Sometimes, we are interested only in how a subset of the variates distributes, without the influence of the rest. For example, what is the distribution of human height only. To do so, we would like to look at the *marginal distribution* of that subset.

Definition 1.2.6 The *marginal distribution* of a subset of coordinates of random variables $\mathbf{x}$ with a joint probability distribution, is the probability distribution of the variables in the set:

$$f(\mathbf{x}) = \int_{\mathbf{y}} f(\mathbf{x}, \mathbf{y})\, d\mathbf{y}$$

where the $\mathbf{y}$ integration is over all the random variables not in $\mathbf{x}$.

Let us find the marginal distribution of a bivariate Gaussian. Let $\mathbf{x} \sim \mathcal{N}(\mu, \Sigma)$ where $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \in \mathbb{R}^2$ and

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

To find the PDF of the marginal distribution of (w.l.o.g) x_1, Observe that we can write the PDF as follows:

$$\begin{aligned} f(\mathbf{x}) &= \frac{1}{\sqrt{(2\pi)^2|\Sigma|}}\exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^\top\Sigma^{-1}(\mathbf{x}-\mu)\right) \\ &= \frac{1}{\sqrt{(2\pi)^2\sigma_1^2\sigma_2^2}}\exp\left(-\frac{1}{2}\begin{bmatrix} x_1-\mu_1 & x_2-\mu_2 \end{bmatrix}\begin{bmatrix} \sigma_1^{-2} & 0 \\ 0 & \sigma_2^{-2} \end{bmatrix}\begin{bmatrix} x_1-\mu_1 \\ x_2-\mu_2 \end{bmatrix}\right) \\ &= \frac{1}{\sqrt{(2\pi)^2\sigma_1^2\sigma_2^2}}\exp\left(-\frac{1}{2}\left(\frac{x_1-\mu_1}{\sigma_1}\right)^2-\frac{1}{2}\left(\frac{x_2-\mu_2}{\sigma_2}\right)^2\right) \\ &= \frac{1}{\sqrt{2\pi\sigma_1^2}}\exp\left(-\frac{1}{2}\left(\frac{x_1-\mu_1}{\sigma_1}\right)^2\right)\cdot\frac{1}{\sqrt{2\pi\sigma_2^2}}\exp\left(-\frac{1}{2}\left(\frac{x_2-\mu_2}{\sigma_2}\right)^2\right) \end{aligned}$$

Using the definition of the marginal distribution:

$$\begin{aligned} f(x_1) &= \int_{-\infty}^{\infty} f(x_1,x_2)\,dx_2 \\ &= \int_{-\infty}^{\infty}\frac{1}{\sqrt{2\pi\sigma_1^2}}\exp\left(-\frac{1}{2}\left(\frac{x_1-\mu_1}{\sigma_1}\right)^2\right)\cdot\frac{1}{\sqrt{2\pi\sigma_2^2}}\exp\left(-\frac{1}{2}\left(\frac{x_2-\mu_2}{\sigma_2}\right)^2\right)dx_2 \\ &= \frac{1}{\sqrt{2\pi\sigma_1^2}}\exp\left(-\frac{1}{2}\left(\frac{x_1-\mu_1}{\sigma_1}\right)^2\right)\cdot\int_{-\infty}^{\infty}\frac{1}{\sqrt{2\pi\sigma_2^2}}\exp\left(-\frac{1}{2}\left(\frac{x_2-\mu_2}{\sigma_2}\right)^2\right)dx_2 \end{aligned}$$

Notice that now we integrate a function of a univariate Gaussian for all values $x_2 \in (-\infty,+\infty)$. Therefore this integral equals to 1 and we are left with:

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

Which by definition 1.1.2 is a univariate Gaussian of the form $x_1 \sim \mathcal{N}(\mu_1,\sigma_1^2)$. Notice that this result is to be expected. The covariance between the two properties is zero, namely they are uncoordinated and as such they do not influence each others distribution.

1.2.1 Estimators of Multivariate Gaussian Distribution

Just as we were interested in estimating parameters in the univariate case so we are in the multivariate case. The sample mean estimator of a multivariate distribution is simply the univariate estimator in each of the coordinates:

$$\hat{\mu} := \begin{bmatrix} \vdots \\ \hat{\mu}_j \\ \vdots \end{bmatrix} = \begin{bmatrix} \vdots \\ \frac{1}{m}\sum_i x_{i,j} \\ \vdots \end{bmatrix} \tag{1.15}$$

To define an estimator for the covariance matrix we must first define an estimator for the sample covariance between two random variables. The unbiased estimator of the *sample covariance* of the i'th and j'th random variables is given by:

$$\hat{\sigma}(X_i,X_j) = \frac{1}{m-1}\sum_{k=1}^{m}(x_{k,i}-\hat{\mu}_i)(x_{k,j}-\hat{\mu}_j) \tag{1.16}$$

where $\hat{\mu}_i$ is the sample mean of the random variable X_i.
In particular, notice that for the case where $i=j$ we are left with the unbiased estimator previously seen. We can now define the sample covariance matrix $\hat{\Sigma}$:

Definition 1.2.7 Let $X=(X_1,\ldots,X_d)^\top$ be a d-dimensional random vector. Let $\mathbf{x}_1,\ldots,\mathbf{x}_m \in \mathbb{R}^d$ be m i.i.d samples realizing X. The *sample covariance matrix* is a square $d\times d$ matrix $\hat{\Sigma}$ such that $\hat{\Sigma}_{i,j} = \hat{\sigma}(X_i,X_j) \quad i,j=1,\ldots,d$.

In matrix notation, for $\mathbf{X} \in \mathbb{R}^{m \times d}$ the matrix whose rows are the samples $\mathbf{x}_1, \ldots, \mathbf{x}_m$, the (biased) estimator for the sample covariance matrix is given by:

$$\hat{\Sigma} := \frac{1}{m} \sum_{i=1}^{m} (\mathbf{x}_i - \hat{\mu})(\mathbf{x}_i - \hat{\mu})^\top = \frac{1}{m} \widetilde{\mathbf{X}}^\top \widetilde{\mathbf{X}}$$

for $\widetilde{\mathbf{X}}$ being the centered matrix: $\widetilde{\mathbf{X}}_{\cdot,i} := \mathbf{X}_{\cdot,i} - \hat{\mu}$. The unbiased estimator is given by:

$$\hat{\Sigma} := \frac{1}{m-1} \sum_{i=1}^{m} (\mathbf{x}_i - \hat{\mu})(\mathbf{x}_i - \hat{\mu})^\top = \frac{1}{m-1} \widetilde{\mathbf{X}}^\top \widetilde{\mathbf{X}}$$

■ **Example 1.3** Let $\mathbf{X} = \begin{pmatrix} 150 & 45 \\ 170 & 74 \\ 184 & 79 \end{pmatrix}$ be samples of height and weight of 3 different individuals. To calculate the sample covariance matrix we begin with centering the data. That is, subtract the empirical mean from each sample. The sample mean is: $\hat{\mu} = (168, 66)^\top$, so:

$$\mathbf{X}_{centered} = \mathbf{X} - \begin{pmatrix} 168 & 66 \\ 168 & 66 \\ 168 & 66 \end{pmatrix} = \begin{pmatrix} 150 & 45 \\ 170 & 74 \\ 184 & 79 \end{pmatrix} - \begin{pmatrix} 168 & 66 \\ 168 & 66 \\ 168 & 66 \end{pmatrix} = \begin{pmatrix} -18 & -21 \\ 2 & 8 \\ 16 & 13 \end{pmatrix}$$

Now, following the definition of the sample covariance matrix:

$$\hat{\Sigma} = \frac{1}{3-1} \mathbf{X}_{centered}^\top \mathbf{X}_{centered} = \begin{pmatrix} 292 & 301 \\ 301 & 337 \end{pmatrix}$$

■

1.3 Summary, Labs & Exercises

Exercises

Theoretical Questions

1. Let $x_1, x_2, \ldots \overset{i.i.d}{\sim} \mathcal{P}$ be a sample of infinity size drawn from some probability distribution function $\mathcal{P}$ with finite expectation and variance. Show that the sample mean estimator $\hat{\mu}_n = \frac{1}{n}\sum x_i$ calculated over the first n samples is a consistent estimator.

2. Consider the estimator of sample variance with *unknown* sample mean defined as $\hat{\sigma}^2 = \frac{1}{m}\sum (x_i - \hat{\mu})^2$, for $\hat{\mu}$ the sample mean estimator. Show that this is a biased estimator and find its systematic error.

3. Consider the estimator of sample variance with *known* sample mean defined as $\hat{\sigma}^2 := \frac{1}{m}\sum (x_i - \mu)^2$. Show that this is an unbiased estimator. Explain why the estimator in the question above is biased while the current estimator is unbiased.

4. Consider the estimator of sample variance with *unknown* sample mean defined as $\hat{\sigma}^2 = \frac{1}{m-1}\sum (x_i - \hat{\mu})^2$, for $\hat{\mu}$ the sample mean estimator. Show that this is an unbiased estimator.

5. Let $x_1, \ldots, x_m \overset{i.i.d}{\sim} \mathcal{N}(\mu, \sigma^2)$. Derive the MLE for the variance σ^2 and show that it is in fact the estimator seen in question Ex.2.

6. Let $\mathbf{x}_1, \ldots, \mathbf{x}_m \overset{i.i.d}{\sim} \mathcal{N}(\mu, \Sigma)$ be m observations sampled i.i.d from a multivariate Gaussian with expectation of $\mu \in \mathbb{R}^d$ and a covariance matrix $\Sigma \in \mathbb{R}^{d \times d}$. Derive the likelihood function of $\mathcal{N}(\mu, \Sigma)$. Hint: follow the approach used to derive the likelihood function for the univariate case (1.6).

7. Prove that multivariate Gaussian distributions are closed under affine transformations. That is, for $\mathbf{x} \sim \mathcal{N}(\mu, \Sigma)$, $\mu \in \mathbb{R}^d$, $\Sigma \in \mathbb{R}^{d \times d}$ and fixed $A \in \mathbb{R}^{m \times d}, b \in \mathbb{R}^m$ then $A\mathbf{x} + b$ follows a multivariate Gaussian distribution. What is the expectation and covariance matrix?

8. Let $X \sim \mathcal{N}(\mu, \Sigma)$ be a multivarite Gaussian distribution with $\mu \in \mathbb{R}^d$ and $\Sigma \in \mathbb{R}^{d \times d}$. Denote $X = [X_1 | X_2]^\top$ some partitioning of the coordinates of X. Prove that the conditional distribution $X_1 | X_2$ is also a multivariate Gaussian and find its expectation and covariance matrix.

9. Let $X \sim \mathcal{N}(\mu, \Sigma)$ be a multivarite Gaussian distribution with $\mu \in \mathbb{R}^d$ and $\Sigma \in \mathbb{R}^{d \times d}$. Denote $X = [X_1 | X_2]^\top$ some partitioning of the coordinates of X. Prove that the marginal distribution X_1 is also a multivariate Gaussian and find its expectation and covariance matrix.

Practical Questions

Clone the IML.HUJI GitHub repository and setup a working python environment as explained on GitHub page.

1. Implement the `fit, pdf` and `log_likelihood` functions in class `UnivariateGaussian`, file `learners.GaussianEstimators.py`. Follow the details specified in class and function documentation.
   (a) Using `numpy.random.normal` draw 1000 samples $x_1, \ldots, x_{100} \overset{i.i.d}{\sim} \mathcal{N}(10, 1)$ and fit a univariate Gaussian. What are the estimations of the expectation and variance?

   (b) Over previously drawn samples, fit a series of models of increasing samples size: 10, 20,...,100, 110,...1000. Plot the absolute distance between the estimated- and true value of the expectation, as a function of the sample size. What estimator property are we able to conclude for the sample mean estimator?

   (c) Compute the PDF of the previously drawn samples using the model fitted above (over all samples) and plot the empirical PDF distribution.

   (d) Over a sample $S = \{1, 5, 2, 3, 8, -4, -2, 5, 1, 10, -10, 4, 5, 2, 7, 1, 1, 3, 2, -1, -3, 1, -4, 1, 2, 1\}$ which of the following models is more likely to have generated these samples: $\mathcal{N}(1, 1)$ or $\mathcal{N}(10, 1)$?

2. Implement the `fit, pdf` and `log_likelihood` functions in class `MultivariateGaussian`, file `learners.GaussianEstimators.py`. Follow the details specified in class and function documentation.
   (a) Using `numpy.random.multivariate_normal` draw 1000 samples $\mathbf{x}_1, \ldots, \mathbf{x}_{1000} \overset{i.i.d}{\sim} \mathcal{N}(\mu, \Sigma)$

$$\mu = \begin{bmatrix} 0 \\ 0 \\ 4 \\ 0 \end{bmatrix}, \quad \Sigma = \begin{bmatrix} 1 & 0.2 & 0 & 0.5 \\ 0.2 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0.5 & 0 & 0 & 1 \end{bmatrix}$$

   Fit a multivariate Gaussian and evaluate the estimated expectation and covariance matrix with respect to the true parameters.

   (b) Using the samples drawn in the question above calculate the log-likelihood for models with expectation $\mu = [f_1, 0, f_3, 0]^\top$ and the true covariance matrix defined above, where f_1, f_3 get values returned from `np.linspace(-10, 10, 200)`. Plot a heatmap of $f1$ values as rows, f_3 values as columns and color coded by the calculated log likelihood. What can be learned from the plot? What configuration of (f_1, f_3) values achieved maximal likelihood?

2. Regression Models

2.1 Batch Supervised Regression Models

Machine learning models can be designed to accomplish many different tasks. We'll start with a very common task: *batch supervised regression on the domain* $\mathbb{R}^d$. To understand what the words "batch", "supervised", "regression" and "on the domain $\mathbb{R}^d$" mean in the context of machine learning, here is an example.

Imagine we work for an online store and would like to predict the "customer lifetime value", that is, the total future net revenue that an online customer will provide to the store. To do so, we choose different customer properties that we think might be relevant such as age, income, total spending in the website, average monthly visits, etc. Suppose that we choose d different properties. The vector space defined over all possible values of these d properties (commonly called *d features*) is our *sample domain*. We denote it by $\mathcal{X}$. In our example, $\mathcal{X} := \mathbb{R}^d$, where d the number of features we will use. The set containing all the possible values for the quantity we want to predict is called the *response set* and denoted $\mathcal{Y}$. In our example, this set contains all possible customers' lifetime value, so that we can use $\mathcal{Y} = \mathbb{R}$.

A machine learning problem where we are given a *sample* $x \in \mathcal{X}$, and would like to predict a *label* $y \in \mathcal{Y}$ associated with it, is called a *prediction problem*. When our prediction for the label of a new sample $x \in \mathcal{X}$ is based on past observations of pairs $(x,y) \in \mathcal{X} \times \mathcal{Y}$, we say that the learning problem *supervised*. When the sample domain is the Euclidean space $\mathcal{X} = \mathbb{R}^d$, we can say that each sample is a features vector with d features. When the label set is the real line $\mathcal{Y} = \mathbb{R}$, we say that the prediction problem is a *regression problem*. So we now know what the phrase "supervised regression prediction learning problem on the domain $\mathbb{R}^d$" means.

How can we predict customer lifetime value? It makes sense to start by collecting all the d features we chose for m customers, for who we already know the lifetime value. This way, we can hope to detect patterns that relate the d features of a customer $x \in \mathbb{R}^d$ to their lifetime value $y \in \mathbb{R}$. This set of m *observations*, of the form $S = \{(x_i, y_i)\}_{i=1}^m$ will be our *training sample* or *training dataset*. We denote it by S. We would like to use our training dataset to find a way to predict, as accurately as possible, the lifetime values of any new customer using solely their feature vector. This setup, where we are given a training set of m labeled samples, and would

like to use them in order to create a prediction rule that can predict the label of any new sample $\mathbf{x} \in \mathcal{X}$ we may encounter, is called *batch supervised learning*.

A *prediction model* is a way to represent a functional relation that maps a sample $x \in \mathcal{X}$ to a response $\in \mathcal{Y}$. So, we will *assume* that there exists some function $f : \mathcal{X} \to \mathcal{Y}$ that captures the relation we are interested in for each sample $x \in \mathcal{X}$ and its response $y \in \mathcal{Y}$. This function f is unknown to us and we would like to find it. It may be deterministic or it may contain a random component.

For simplicity, let us begin by assuming that the functional relation between $x \in \mathcal{X}$ and $y \in \mathcal{Y}$ is *deterministic*. So we assume that there exists a function $f : \mathcal{X} \to \mathcal{Y}$ such that, each sample we observe, now or in the future, is of the form (x, y) with $y = f(x)$. In particular for our training set $y_i = f(x_i)$ for every training sample $i = 1, \ldots, m$. Our goal is to *learn* f from a training sample $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^{m}$, so we can estimate or predict the value $f(x)$ for a new value x. A sample we haven't seen in our training set – a new sample – is sometimes called a *test* sample. Using the training sample, which we denote by S, we will create a function that we hope is as similar as possible to the unknown function f. This function is the *prediction rule* or *decision rule* and we denote it by $\hat{f}$ or h_S (The notation h_S emphasizes that our prediction rule very much depends on the training sample S).

For reasons we discuss later (section 4.2), whenever we try to model a functional relation f, we restrict ourselves to functions f in a specified family of functions. Such a family is referred to as a *hypothesis class*. While we can build regression models over any domain $\mathcal{X}$, the simplest domain to consider is the Euclidean space $\mathbb{R}^d$ where each point x is a feature vector with d real numbers. In this chapter (and in most of this book) we consider the case $\mathcal{X} := \mathbb{R}^d$.

2.2 The Linear Regression Model

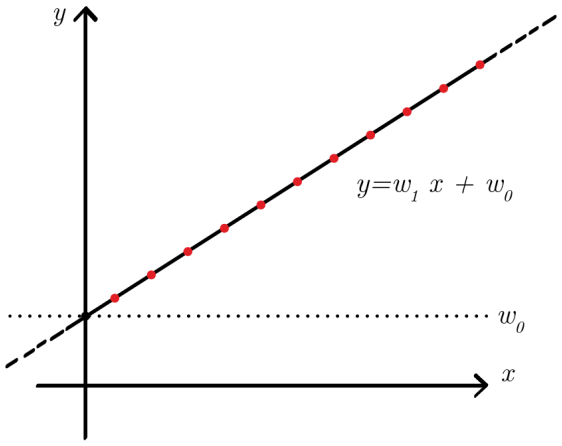


Figure 2.1: *Illustration of a linear regression model with $\mathcal{X} = \mathbb{R}$ and $\mathcal{Y} = \mathbb{R}$. Red dots are samples. The solid curve is the learned prediction rule $\hat{f}$.*

Hypothesis Class

Let us assume that the relation $\mathcal{X} \to \mathcal{Y}$ is *linear*. This is perhaps the simplest relation we can describe. Formally, we define the *linear model*, or the *linear hypothesis class*, as the set of linear functions from the

domain set to the response set:

$$\mathcal{H}_{reg} := \left\{ h(x_1,\ldots,x_d) = w_0 + \sum_{i=1}^{d} x_i w_i \;\middle|\; w_0, w_1, \ldots, w_d \in \mathbb{R} \right\} \tag{2.1}$$

In statistics, learning f from a training sample is known as *linear regression*[1]. Each function $h \in \mathcal{H}_{reg}$ is characterized by the *weights* (also known as regression coefficients) $w_1,\ldots,w_d$ representing the d features and an *intercept* w_0. To simplify the notation, for a given sample $\mathbf{x} = (x_1,\ldots,x_d)^\top \in \mathbb{R}^d$ we add a zero-th coordinate with the value of 1, and define $\mathbf{x} = (1,x_1,\ldots,x_d)^\top \in \mathbb{R}^{d+1}$. Using this notation each function in the linear hypothesis class can be written in the form $h(\mathbf{x}) := \langle \mathbf{x}, \mathbf{w} \rangle = \mathbf{x}^\top \mathbf{w}$. For the remainder of this chapter, we will assume that the intercept is already incorporated into the weights vectors, so we can define linear hypothesis class equivalently as

$$\mathcal{H}_{reg} := \left\{ h_{\mathbf{w}}(\mathbf{x}) = \mathbf{x}^\top \mathbf{w} \mid \mathbf{w} \in \mathbb{R}^{d+1} \right\} \tag{2.2}$$

Note that by convention, the first coordinate of $\mathbf{w}$ is the intercept w_0. So, given a training set S, we are looking for a vector $\mathbf{w} \in \mathbb{R}^{d+1}$ such that $y_i = \mathbf{x}_i^\top \mathbf{w}$ for all $i \in [m]$.

The regression matrix

Let us arrange the training data $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ in matrix form. We define the *response vector* as the column vector $\mathbf{y} \in \mathbb{R}^m$ and the *regression matrix* (or *design matrix*) $\mathbf{X} \in \mathbb{R}^{m \times (d+1)}$ as follows.

$$\mathbf{X} = \begin{bmatrix} — & \mathbf{x}_1 & — \\ — & \mathbf{x}_2 & — \\ & \vdots & \\ — & \mathbf{x}_m & — \end{bmatrix} \qquad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$$

Note that m rows of $\mathbf{X}$ represent our m training samples and the $d+1$ columns of $\mathbf{X}$ represent the intercept and d features. In this notation, we are looking for a vector $\mathbf{w} \in \mathbb{R}^{d+1}$ that satisfies a system of m linear equations in the variable $\mathbf{w}$,

$$\mathbf{X}\mathbf{w} = \mathbf{y} \tag{2.3}$$

At this point, we will assume that $m \geq d+1$, namely, that we have *enough training samples* so that the linear system (2.3) is *not under-determined*. In practical terms, this means that we have at least as many training samples as we have features. In our online store example, this means that we must collect data on $m \geq d+1$ customers before we start training our regression model, where d is the number of features we collect on each customer (e.g. age, income, total spending, number of monthly visits to the website, etc).

2.2.1 Designing A Learning Algorithm

2.2.1.1 Realizablity

Recall that to derive the problem of finding $\mathbf{w} \in \mathbb{R}^{d+1}$ that satisfies (2.3) we have restricted ourselves to describing functional relations $\mathcal{X} \xrightarrow{f} \mathcal{Y}$ such that $f \in \mathcal{H}_{reg}$. The case where there exists a solution for (2.3) is called the *Realizable* case. Let $\hat{\mathbf{w}}$ be a solution for (2.3), then the prediction rule we choose is $\hat{f}(\mathbf{x}) = \mathbf{x}^\top \hat{\mathbf{w}}$.

The case where there is no $f \in \mathcal{H}_{reg}$ that satisfies the system of equations (i.e there is no solution for the system) is called the *Non-Realizable* case. In this case, since we decided to choose a prediction rule in $\mathcal{H}_{reg}$, we must settle for finding $\hat{f} \in \mathcal{H}_{reg}$ which is *"most fitting"* for our purposes.

Our learning algorithm for linear regression must address both the realizable and non-realizable cases. In the realizable case, to find the rule f, all we need to do is solve the linear system (2.3) for $\mathbf{w}$. But what will we do in the non-realizable case, where $f \notin \mathcal{H}_{reg}$? How should we choose the prediction rule $\hat{f}$?

[1] The name "regression" refers to a statistical phenomenon known as "regression to the mean".

2.2.1.2 Empirical Risk Minimization

As we have seen in the previous chapter (subsubsection 1.1.2.1), one way to choose $\hat{f} \in \mathcal{H}_{reg}$ in the non-realizable case is to assign each $f \in \mathcal{H}_{reg}$ with some measure of quality, through the use of some *loss function*. This function will provide a measure of quality for the hypothesis by comparing between the true- and predicted values:

$$\sum_{i=1}^{m} L\left(f\left(\mathbf{x}_i\right), \hat{f}\left(\mathbf{x}_i\right)\right), \quad i = 1, \ldots, m$$

Two commonly used loss functions for regression problems are the *Absolute Value Loss* and *Squared Loss* functions.

$$L\left(y, \hat{f}\left(\mathbf{x}\right)\right) := \left|y - \hat{f}\left(\mathbf{x}\right)\right|, \quad L\left(y, \hat{f}\left(\mathbf{x}\right)\right) := \left(y - \hat{f}\left(\mathbf{x}\right)\right)^2$$

For the remaining of the chapter we will focus on the square loss function. As we are concerned for the performance of an estimator $\hat{f}$ on a new data point $\mathbf{x}$, we hope to minimize the *risk* (i.e. expected loss, 1.1.4) embodied in choosing $\hat{f}$ as our estimator, with respect to the squared loss:

$$\mathcal{R}\left(f, \hat{f}\right) := \mathbb{E}_{\mathbf{x}}\left[L\left(f(\mathbf{x}), \hat{f}(\mathbf{x})\right)\right] = \mathbb{E}_{\mathbf{x}}\left[\left(f(\mathbf{x}) - \hat{f}(\mathbf{x})\right)^2\right]$$

where the expectation is taken over the selection of the test sample $\mathbf{x}$. However, as we do not have access to the underlying distribution and only have the training set, we will evaluate this risk *empirically*, that is, over *the training data*. And so, using the sample mean as an estimator of the expectation, the *empirical risk* embodied in choosing $\hat{f}$ (with respect to the squared loss) is

$$\frac{1}{m} \sum_{i=1}^{m} \left(y_i - \hat{f}(\mathbf{x}_i)\right)^2$$

In the case of the square loss, the empirical risk of the linear function $\hat{f}\left(\mathbf{x}_i\right) = \mathbf{x}_i^\top \hat{\mathbf{w}}$ is given by:

$$\frac{1}{m} \sum_{i=1}^{m} \left(y_i - \mathbf{x}_i^\top \hat{\mathbf{w}}\right)^2 = \frac{1}{m} ||\mathbf{y} - \mathbf{X}\hat{\mathbf{w}}||^2 = \frac{1}{m} \left(\mathbf{y} - \mathbf{X}\hat{\mathbf{w}}\right)^\top \left(\mathbf{y} - \mathbf{X}\hat{\mathbf{w}}\right) \tag{2.4}$$

We will then pick the estimator which *minimizes* the empirical risk. This strategy for choosing $\hat{f}$ is known as *Empirical Risk Minimization* (ERM). Since we are in search of the minimizer we often omit the constant value of $\frac{1}{m}$.

2.2.1.3 Least Squares Optimization Problem

Minimizing the empirical risk of (2.4) means minimizing the sum of squares of the deviations of the responses from a linear function. In other words, we choose the linear function in $\mathcal{H}_{reg}$ that is closest to the responses in terms of the squared error distance. The deviation $y_i - \mathbf{x}_i^\top \mathbf{w}$ is called the i-th *residual* and the total empirical risk in our case is called *Residual Sum of Squares* (or *RSS*):

$$\mathrm{RSS}_{\mathbf{X},\mathbf{y}}\left(\mathbf{w}\right) := ||\mathbf{y} - \mathbf{X}\mathbf{w}||^2$$

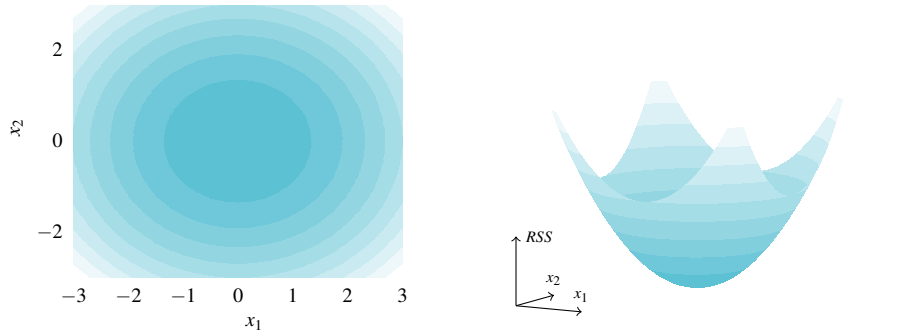
To simplify notation we often write $\mathrm{RSS}\left(\mathbf{w}\right)$ keeping the dependence on $\mathbf{X}, \mathbf{y}$ implicit. So to learn the linear function by empirical Risk minimization we want to find

$$\underset{\mathbf{w} \in \mathbb{R}^d}{\operatorname{argmin}} \ \mathrm{RSS}\left(\mathbf{w}\right) = \underset{\mathbf{w} \in \mathbb{R}^d}{\operatorname{argmin}} \ ||\mathbf{y} - \mathbf{X}\mathbf{w}||^2 \tag{2.5}$$

It is important to notice that the optimization problem (2.5) addresses both the realizable and non-realizable cases:

- In the realizable case, as $\mathbf{y} \in \mathrm{Im}\left(\mathbf{X}\right)$ we know there exists at least one solution $\hat{\mathbf{w}}$ such that $\mathbf{X}\hat{\mathbf{w}} = \mathbf{y}$. Such a solution will achieve a value of zero. As the RSS function is bounded below by zero, such a solution is therefore a minimizer of the RSS.

- In the non-realizable case, as $\mathbf{y} \notin \text{Im}(\mathbf{X})$ there is no solution $\hat{\mathbf{w}}$ such that $\mathbf{X}\hat{\mathbf{w}} = \mathbf{y}$. Therefore, no vector $\hat{\mathbf{w}}$ will achieve a value of zero for the RSS objective. Instead, we decide to find a vector that is "good enough" in the sense of minimizing the squared loss.



(a) *Contour lines of the RSS function*

(b) *3D visualization of the paraboloid defined by the RSS*

Figure 2.2: *Visualization of RSS function*

A *necessary* condition for $\mathbf{w}$ to be a minimizer of the function $||\mathbf{y} - \mathbf{X}\mathbf{w}||^2$ is that all its partial derivative vanish at $\mathbf{w}$. Recalling the definition of the inner product, this condition can be written as:

$$\frac{\partial}{\partial w_j}\text{RSS}(\mathbf{w}) = -2\sum_{i=1}^{m}(\mathbf{x}_i)_j \cdot (y_i - \mathbf{x}_i\mathbf{w}) = 0 \tag{2.6}$$

for all $j = 0, \ldots, d$, where $(\mathbf{x}_i)_j$ is the j-th entry of $\mathbf{x}_i$. It is the $x_{j,i}$ element of the matrix $\mathbf{X}$. Notice that this constructs a system of $d+1$ linear equations in $\mathbf{w}$. We can organize (2.6) as such to get the form below. Recall that we have already derived this function in .13.

$$\nabla\text{RSS}(\mathbf{w}) = -2\mathbf{X}^\top(\mathbf{y} - \mathbf{X}\mathbf{w}) = 0 \tag{2.7}$$

2.2.1.4 The Normal Equations

So a minimizer of (2.5) must also be a solution for the following linear system, known as the **Normal Equations**:

$$\mathbf{X}^\top(\mathbf{y} - \mathbf{X}\mathbf{w}) = 0 \quad \Longleftrightarrow \quad \mathbf{X}^\top\mathbf{y} = \mathbf{X}^\top\mathbf{X}\mathbf{w} \tag{2.8}$$

Geometric Interpretation

Let us derive a geometric interpretation of linear regression and gain a better understanding what the solution to (2.8) might be like. We usually think of $\mathbf{X} \in \mathbb{R}^{m\times(d+1)}$ as a matrix that consists of m rows, one for each training sample. Instead, we can equivalently think of $\mathbf{X}$ as a matrix that consists of $d+1$ columns, one for each feature (and the intercept). Define

$$\mathbf{X} := \begin{bmatrix} | & & | \\ \varphi_0 & \cdots & \varphi_d \\ | & & | \end{bmatrix}$$

and recall that the vector space spanned by the columns of $\mathbf{X}$ is:

$$\text{span}(\varphi_0, \ldots, \varphi_d) = \text{Im}(\mathbf{X}) \subset \mathbb{R}^m$$

Since we assume $m \geq d+1$, $Im(\mathbf{X})$ is a linear subspace of $\mathbb{R}^m$. If we have many more samples than features, $m \gg d+1$, then $\mathrm{Im}(\mathbf{X})$ is just a small subspace of $\mathbb{R}^m$. If we have the minimal number of samples possible, $m = d+1$, and the vectors $\varphi_0, \ldots, \varphi_d$ form an independent set, then the subspace fills the entire space: $\mathrm{Im}(\mathbf{X}) = \mathbb{R}^m$.

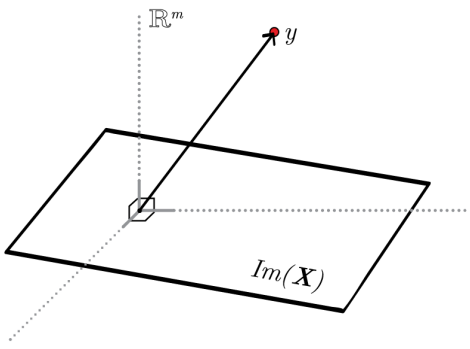
Now, consider the response vector $\mathbf{y} \in \mathbb{R}^m$:

- If $\mathbf{y} \in \mathrm{Im}(\mathbf{X})$ then by definition $\mathbf{y}$ is a linear combination of $\varphi_0, \ldots, \varphi_d$ and there exists a vector $\mathbf{w} \in \mathbb{R}^{d+1}$ such that $\mathbf{X}\mathbf{w} = \mathbf{y}$. This is the realizable case. We can now differentiate between two sub-cases:
  - If $\varphi_0, \ldots, \varphi_d$ are linearly independent, then $\mathbf{y}$ can be expressed as a *unique* linear combination of the columns of $\mathbf{X}$. In this case the linear system (2.8) has a unique solution.
  - If however $\varphi_0, \ldots, \varphi_d$ are in fact linearly dependent, then there are infinitely many ways to express $\mathbf{y}$ as a linear combination of the columns of $\mathbf{X}$. Any one of these ways is a valid solution for (2.8).
- If $\mathbf{y} \notin \mathrm{Im}(\mathbf{X})$ then $\mathbf{y}$ is not a linear combination of $\varphi_0, \ldots, \varphi_d$. As such there is no vector $\mathbf{w} \in \mathbb{R}^{d+1}$ that satisfies $\mathbf{X}\mathbf{w} = \mathbf{y}$. This is the non-realizable case. In this case we decided to choose the vector $\mathbf{w}$ for which $\mathrm{RSS}(\mathbf{w}) = ||\mathbf{X}\hat{\mathbf{w}} - \mathbf{y}||$ is minimal.

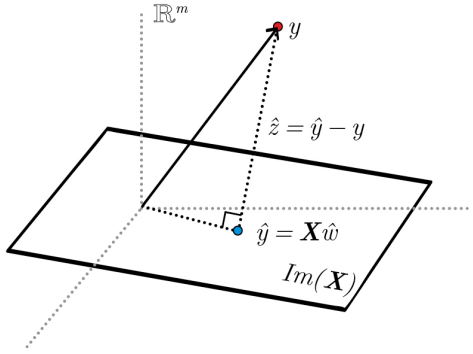
Now we are able to understand what is "normal" about the normal equations (2.8). Observe that the equations (2.6), from which we have derived the normal equations, can be equivalently written as

$$\langle \varphi_j, \mathbf{y} - \mathbf{X}\mathbf{w} \rangle = 0 \quad j = 0, \ldots, d \tag{2.9}$$

We conclude that $\mathbf{w}$ is a solution to the normal equations if and only if $\mathbf{y} - \mathbf{X}\mathbf{w}$ is perpendicular to $\varphi_0, \ldots, \varphi_d$. Since these vectors span the subspace $\mathrm{Im}(\mathbf{X})$, another way to write this is $\mathbf{y} - \mathbf{X}\mathbf{w} \in \mathrm{Im}(\mathbf{X})^{\perp}$.



(a) *In the non-realizable case, the response vector* $\mathbf{y}$ *lies outside* $Im(\mathbf{X})$*, the subspace spanned by the columns of* $\mathbf{X}$*. In this case there is no solution for the system* $\mathbf{X}\mathbf{w} = \mathbf{y}$*.*



(b) *If* $\hat{\mathbf{w}}$ *is a solution to the normal equations, then* $\hat{\mathbf{y}} = \mathbf{X}\hat{\mathbf{w}}$ *is an orthogonal projection of the response vector* $\mathbf{y}$ *onto* $Im(\mathbf{X})$*. The difference* $\hat{\mathbf{z}} = \mathbf{y} - \hat{\mathbf{y}}$ *is therefore perpendicular (normal) to* $Im(\mathbf{X})$*.*

Figure 2.3: Geometric interpretation of linear regression

Let $\hat{\mathbf{w}}$ be a solution to the normal equations and define $\hat{\mathbf{y}} := \mathbf{X}\hat{\mathbf{w}}$. Note that $\hat{\mathbf{y}}$, the vector where the i-th entry is the prediction on the i-th training sample $\mathbf{x}_i$, is $\hat{\mathbf{y}} \in \mathrm{Im}(\mathbf{X})$. In this notation, when solving the normal equations, namely when seeking to minimize the RSS, we minimize $||\mathbf{y} - \hat{\mathbf{y}}||^2$. Define the *residual vector* $\hat{\mathbf{z}} := \mathbf{y} - \hat{\mathbf{y}}$. Note that from (2.9) we get that $\hat{\mathbf{z}} \in \mathrm{Im}(\mathbf{X})^{\perp}$. In other words, if $\hat{\mathbf{w}}$ is a solution to the normal equations then

$\hat{\mathbf{y}} = \mathbf{X}\hat{\mathbf{w}}$ is the *orthogonal projection* of $\mathbf{y}$ on $\mathrm{Im}(\mathbf{X})$ and $\hat{\mathbf{z}} = \mathbf{y} - \hat{\mathbf{y}}$ is a *normal* (a perpendicular vector) to $\mathrm{Im}(\mathbf{X})$. Hence the name the "normal equations".

Solving The Normal Equations

As we have seen, from a geometric perspective, if $m \geq d+1$, solving the normal equations means finding a vector $\hat{\mathbf{w}}$ such that $\hat{\mathbf{y}} = \mathbf{X}\hat{\mathbf{w}}$ is the orthogonal projection of $\mathbf{y}$ on $\mathrm{Im}(\mathbf{X})$. We can deduce from this two important facts about the existence and uniqueness of a solution to the normal equations:

- **Existence:** As a linear system, the normal equations can have either (i) no solutions, (ii) a unique solution, or (iii) an infinite number of solutions that constitute an affine subspace. From the geometric interpretation we see that (i) is impossible. Indeed it can be shown that the normal equations must have at least one solution, so that they have a unique solution or an infinite number of solutions.
- **Uniqueness:**
  - Case 1 - If the columns of $\mathbf{X}$ form a linearly independent set (equivalently, if $\dim(Ker(\mathbf{X})) = 0$) then the projection $\hat{\mathbf{y}}$ can be described uniquely as a linear combination of the columns $\varphi_0, \ldots, \varphi_d$, namely, there exists a unique $\hat{\mathbf{w}}$ such that $\hat{\mathbf{y}} = \mathbf{X}\hat{\mathbf{w}}$. This vector of coefficients $\hat{w}$ is a unique solution to the normal equations.
  - Case 2 - If the columns of $\mathbf{X}$ contain linear dependencies (equivalently, if $\dim(Ker(\mathbf{X})) > 0$) then the projection $\hat{\mathbf{y}}$ can be described as infinitely many linear combinations of the columns $\varphi_0, \ldots, \varphi_d$. Any such linear combination will suffice and we simply need to find *one* vector $\hat{w}$ such that $\hat{y} = \mathbf{X}\hat{\mathbf{w}}$.

Case 1: Linearly Independent Feature Vectors

If the features are linearly independent then the kernel of $\mathbf{X}$ is trivial ($\dim(Ker(\mathbf{X})) = 0$). Now, consider the square and symmetric matrix $\mathbf{X}^\top\mathbf{X}$. As $Ker(\mathbf{X}) = Ker(\mathbf{X}^\top\mathbf{X})$, then $\mathbf{X}^\top\mathbf{X}$ too has a trivial kernel. This means that $\mathbf{X}^\top\mathbf{X}$ is invertible and that a vector $\mathbf{w}$ satisfies $\mathbf{X}^\top\mathbf{y} = \mathbf{X}^\top\mathbf{X}\mathbf{w}$ if and only if $\mathbf{w} = \left[\mathbf{X}^\top\mathbf{X}\right]^{-1}\mathbf{X}^\top\mathbf{y}$. So in this case the unique solution to the normal equations is Ex.1

$$\hat{\mathbf{w}} := \left[\mathbf{X}^\top\mathbf{X}\right]^{-1}\mathbf{X}^\top\mathbf{y} \tag{2.10}$$

Lastly, as the RSS function is a convex function (.22) we conclude that the unique solution $\hat{\mathbf{w}}$ is a minimizer of the function.

Notice when deriving a geometric interpretation of the normal equations (Figure 2.3) we argued that the minimizer will orthogonally project $\mathbf{y}$ onto the subspace spanned by the columns of $\mathbf{X}$. Looking at $\hat{\mathbf{y}} = \mathbf{X}\hat{\mathbf{w}} = \mathbf{X}\left[\mathbf{X}^\top\mathbf{X}\right]^{-1}\mathbf{X}^\top\mathbf{y}$ we indeed find that $\mathbf{X}\left[\mathbf{X}^\top\mathbf{X}\right]^{-1}\mathbf{X}^\top$ is an orthogonal projection matrix onto the column space of $\mathbf{X}$. Ex.2

■ **Example 2.1** Let us find the estimator $\hat{\mathbf{w}}$ for the following scenario. Suppose we are interested in estimating the running times in a 100 meters long race, besed on an athlete's height and weight. We gathered the details of the 4 top ranking athletes in the 2016 Rio Olympics:

Athlete	Weight (*kg*)	Height (*cm*)	Running Time (*sec*)
Usain Bolt	94	195	9.81
Justin Gatlin	79	185	9.89
Andre de Grasse	70	176	9.91
Yohan Blake	80	180	9.93

So the features are the *weight, height* and the response is *running time*. To fit a linear regression model to the

data we begin with arranging it in a matrix and adding the intercept:

$$\mathbf{X} := \begin{bmatrix} 1 & 94 & 195 \\ 1 & 79 & 185 \\ 1 & 70 & 176 \\ 1 & 80 & 180 \end{bmatrix}, \quad \mathbf{y} := \begin{bmatrix} 9.81 \\ 9.89 \\ 9.91 \\ 9.93 \end{bmatrix}$$

As we have proven above, the estimator is given by the closed form of $\hat{\mathbf{w}} := \left[\mathbf{X}^\top \mathbf{X}\right]^{-1} \mathbf{X}^\top \mathbf{y}$. Over given data we obtain that $\hat{\mathbf{w}} \approx (11.38, 0.003, -0.009)^\top$ (up to rounding up numbers).

Next, let us use this estimator to estimate the running times of a new sample $\mathbf{x} = (1, 74, 176)^\top$:

$$\hat{y} = \mathbf{x}^\top \hat{\mathbf{w}} = \left\langle \begin{bmatrix} 1 \\ 74 \\ 176 \end{bmatrix}, \begin{bmatrix} 11.38 \\ 0.003 \\ -0.009 \end{bmatrix} \right\rangle = 10.018$$

■

Case 2: Linearly Dependent Feature Vectors

Next, let us consider the case of features that are linearly dependent. In this case the columns of $\mathbf{X}$ are linearly dependent and $\dim(Ker(\mathbf{X})) > 0$. Therefore, there are infinitely many was to express the projection $\hat{\mathbf{y}}$ as a linear combination of the columns of $\mathbf{X}$. Since we need some way, it would be convenient if we could find a solution $\hat{\mathbf{w}}$ that is close to the origin in $\mathbb{R}^{d+1}$ (rather than a solution with very large norm, say). One way to do that is by using the SVD of $\mathbf{X}$.

Definition 2.2.1 Let $\mathbf{X} \in \mathbb{R}^{m \times (d+1)}$ and let $\mathbf{X} = U\Sigma V^\top$ be its SVD. The **Moore-Penrose pseudoinverse** of $\mathbf{X}$ is $\mathbf{X}^\dagger = V\Sigma^\dagger U^\top$ where $\Sigma^\dagger$ is a $(d+1) \times m$ diagonal matrix defined by:

$$\Sigma^\dagger_{i,i} = \begin{cases} 1/\Sigma_{i,i} & \Sigma_{i,i} \neq 0 \\ 0 & \Sigma_{i,i} = 0 \end{cases}$$

This is a generalization of the inverse matrix and indeed when the matrix $\mathbf{X}$ is invertible then then $\mathbf{X}^\dagger = \mathbf{X}^{-1}$. Using the definition above, let us find a minimizer of (2.5).

Claim 2.2.1 Let $\mathbf{X}, \mathbf{y}$ be a regression problem where $m \geq d+1$. If $\dim(Ker(\mathbf{X})) \neq 0$ then $\hat{\mathbf{w}} = \mathbf{X}^\dagger \mathbf{y}$ is a minimizer of the RSS (2.5).

Proof. Denote the rank of $\mathbf{X}$ by r for which, since the kernel of $\mathbf{X}$ is non-trivial, $r \in [1, d+1)$. Let $X = U\Sigma V^\top$ be the SVD of $\mathbf{X}$ and $\sigma_1 \geq \ldots \geq \sigma_r > 0$. Recall the columns of U and V provide orthonormal bases for the four fundamental subspaces:

$$\begin{array}{lll} U_{\mathcal{R}} \in \mathbb{R}^{m \times r} & \mathcal{R}(\mathbf{X}) & = span\{\mathbf{u}_1, \ldots, \mathbf{u}_r\} \\ V_{\mathcal{R}} \in \mathbb{R}^{(d+1) \times r} & \mathcal{R}(\mathbf{X}^\top) & = span\{\mathbf{v}_1, \ldots, \mathbf{v}_r\} \\ U_{\mathcal{N}} \in \mathbb{R}^{m \times (m-r)} & \mathcal{N}(\mathbf{X}^\top) & = span\{\mathbf{u}_{r+1}, \ldots, \mathbf{u}_m\} \\ V_{\mathcal{N}} \in \mathbb{R}^{(d+1) \times (d+1-r)} & \mathcal{N}(\mathbf{X}) & = span\{\mathbf{v}_{r+1}, \ldots, \mathbf{v}_{d+1}\} \end{array}$$

Denote $\mathcal{S} \in \mathbb{R}^{r \times r}$ the diagonal matrix with the r positive singular values on its main diagonal: $\mathcal{S} := diag(\sigma_1, \ldots, \sigma_r)$. Using these notations, recall the compact SVD form of $\mathbf{X}$ (8). So

$$\mathbf{X} := U\Sigma V^\top = \begin{bmatrix} U_{\mathcal{R}} & U_{\mathcal{N}} \end{bmatrix} \begin{bmatrix} \mathcal{S} & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} V_{\mathcal{R}}^\top \\ V_{\mathcal{N}}^\top \end{bmatrix} = U_{\mathcal{R}} \mathcal{S} V_{\mathcal{R}}^\top = \widetilde{U}\widetilde{\Sigma}\widetilde{V}^\top$$

Now we solve the normal equations for $\mathbf{w}$, substituting $\mathbf{X}$ with its compact SVD form:

$$\begin{aligned} \mathbf{X}^\top \mathbf{y} &= \mathbf{X}^\top \mathbf{X}\mathbf{w} \\ \widetilde{V}\widetilde{\Sigma}^\top \widetilde{U}^\top \mathbf{y} &= \widetilde{V}\widetilde{\Sigma}^\top \widetilde{U}^\top \widetilde{U}\widetilde{\Sigma}\widetilde{V}^\top \mathbf{w} \\ \widetilde{\Sigma}\widetilde{U}^\top \mathbf{y} &= \widetilde{\Sigma}^2 \widetilde{V}^\top \mathbf{w} \end{aligned}$$

Since $\widetilde{\Sigma} \in \mathbb{R}^{r\times r}$ of full rank then $\widetilde{\Sigma}^{-1}$ exists and therefore:

$$\mathbf{w} = \widetilde{V}\widetilde{\Sigma}^{-1}\widetilde{U}^\top \mathbf{y}$$

Lastly, using the Moore-Perose pseudoinverse definition we "expand" the compact SVD form and conclude that:

$$\begin{aligned} \hat{\mathbf{w}} &= \widetilde{V}\widetilde{\Sigma}^{-1}\widetilde{U}^\top \mathbf{y} \\ &= V\Sigma^\dagger U^\top \mathbf{y} \\ &= \mathbf{X}^\dagger \mathbf{y} \end{aligned}$$

■

An important property of the pseudoinverse is that for a linear system of equations $A\mathbf{x} = \mathbf{b}$ with an infinite number of solutions, then $A^\dagger \mathbf{b}$ is a solution with minimal ℓ_2 norm, namely

$$A^\dagger \mathbf{b} = \operatorname{argmin}\left\{ ||\mathbf{x}||_2 \;\middle|\; A\mathbf{x} = \mathbf{b} \right\} \tag{2.11}$$

and therefore $\hat{\mathbf{w}} := X^\dagger \mathbf{y}$ is the solution closest to the origin with respect ot the Euclidean norm.

It can be shown that when dealing with a matrix of linearly independent columns ($\dim(Ker(\mathbf{X})) = 0$) then the previously found solution $\hat{\mathbf{w}} = \left[\mathbf{X}^\top \mathbf{X}\right]^{-1} \mathbf{X}^\top \mathbf{y}$ equals to $\mathbf{X}^\dagger \mathbf{y}$. We conclude that the formula $\mathbf{X}^\dagger \mathbf{y}$ *always* gives us a solution to the normal equations: the unique solution if the solution is unique, and the solution with minimal ℓ_2 norm if not. Ex.4

Lastly, recall that when deriving a geometric intuition for what a minimizer of the normal equations should achieve, we realized that it would provide an orthogonal projection of $\mathbf{y}$ onto $Im(\mathbf{X})$. Indeed, in the non-singular form of solution then:

$$\hat{\mathbf{y}} = \mathbf{X}\hat{\mathbf{w}} = \overbrace{\mathbf{X}\left[\mathbf{X}^\top \mathbf{X}\right]^{-1}\mathbf{X}^\top}^{P_{\mathbf{X}}} \mathbf{y}$$

where $P_{\mathbf{X}}$ being square and symmetric is a projection matrix onto the range of $\mathbf{X}$:

$$P_{\mathbf{X}}^2 = \left(\mathbf{X}\left[\mathbf{X}^\top \mathbf{X}\right]^{-1}\mathbf{X}^\top\right)\left(\mathbf{X}\left[\mathbf{X}^\top \mathbf{X}\right]^{-1}\mathbf{X}^\top\right) = \mathbf{X}\left[\mathbf{X}^\top \mathbf{X}\right]^{-1}\mathbf{X}^\top = P_{\mathbf{X}}$$

In the general case for $\hat{\mathbf{w}} = \mathbf{X}^\dagger \mathbf{y}$ then:

$$\begin{aligned} \hat{\mathbf{y}} &= \mathbf{X}\hat{\mathbf{w}} = \mathbf{X}\mathbf{X}^\dagger \mathbf{y} = U\Sigma V^\top V\Sigma^\dagger U^\top \mathbf{y} = U\Sigma\Sigma^\dagger U\mathbf{y} \\ &= U\left[\begin{array}{c|c} I_r & 0 \\ \hline 0 & 0 \end{array}\right] U^\top \mathbf{y} = U_{\mathcal{R}} U_{\mathcal{R}}^\top \mathbf{y} = \textstyle\sum_{i=1}^r \mathbf{u}_i \mathbf{u}_i^\top \mathbf{y} \end{aligned}$$

for $rank(\mathbf{X}) = r$, providing an orthogonal projection matrix onto the range of $\mathbf{X}$.

R In this chapter we have only delt with the case where $m \geq d+1$, namely that we are more samples than features, and were therefore able to find solutions to the linear system. If however $m < d+1$ we do not have enough data to learn a map $\mathcal{X} \xrightarrow{f} \mathcal{Y}$. Our hypothesis class is a $d+1$ dimensional linear subspace, so to learn we require at least $d+1$ samples. In general, the larger d, the more complicated the more complex our hypothesis class and therefore the more samples we will need in order to learn it. This intuition will be further formulated in **??**.

2.2.2 Numerical Implementation Considerations

So far we have designed the learning algorithm. Now we want to *implement* it, namely, write efficient code that implements the algorithm that we have designed. The field of *numerical linear algebra* assists in addressing this challenge as the implementation of every machine learning algorithm is eventually reduced to performing linear algebra computations (e.g. matrix-vector or matrix-matrix products, matrix inverses and matrix decompositions). In the case of linear regression, as we have seen, to write software that trains a linear regression model, we need to be able to calculate a matrix SVD.

In your basic linear algebra courses you worked with mathematical objects over real and complex vector spaces. Likely, you did not stop to wonder how to compute the inverse of a matrix on a computer. This is not as simple as it may sound. Computers do not calculate over $\mathbb{R}$, they use bits and more specifically floating-point arithmetics with finite precision. There is an entire field in the intersection of mathematics and computer science, known as numerical linear algebra, that studies the accuracy and complexity of algorithms for computing linear algebraic quantities and matrix decompositions. As a machine learning expert, you must be as knowledgable as possible regarding the numerical implementation of your learning algorithms. You should care *deeply* about how your algorithms are implemented and when they break numerically.

Let's see a simple example for a numerical consideration in our case of linear regression (We will discover that the SVD is even more useful than we thought). Recall that if $\dim(Ker(\mathbf{X})) = 0$, and equivalently if $\mathbf{X}^\top\mathbf{X}$ is not singular (i.e. is invertible) then we have a simple formula for training our linear regression model. But what happens if $\mathbf{X}^\top\mathbf{X}$ is "almost singular"?

Sometimes $\mathbf{X}^\top\mathbf{X}$ is formally invertible but *close to singular*. This happens if columns of $\mathbf{X}$ are almost co-linear or if one column of $\mathbf{X}$ is almost spanned by other columns. When this happens, if we are not careful we will run into numerical trouble. For example:

- Suppose we use the formula $\hat{\mathbf{w}} = \left[\mathbf{X}^\top\mathbf{X}\right]^{-1}\mathbf{X}^\top\mathbf{y}$ and try to compute $[\mathbf{X}^\top\mathbf{X}]^{-1}$ using (say) Gauss elimination, we'll find that Gauss elimination may yield wildly incorrect results.
- Suppose we use the pseudoinverse formula and compute $\mathbf{X}^\dagger$. When $\mathbf{X}^\top\mathbf{X}$ is close to singular, we'll discover that the smallest singular values σ_i of $\mathbf{X}$ are very very small; when we try to compute $1/\sigma_i$ for the pseudoinverse with floating-point arithmetics, $1/\sigma_i$ will not be precise.

There is a simple practical solution for this problem: we choose a "numerical precision threshold" $\varepsilon > 0$ in advance. We can choose, say, $\varepsilon := 10^{-8}$. We then change the definition of the pseudoinverse slightly and define

$$\Sigma^{\dagger,\varepsilon}_{i,i} = \begin{cases} 1/\sigma_i & \sigma_i > \varepsilon \\ 0 & \sigma_i \leq \varepsilon \end{cases}.$$

This ensures that even if the columns of $\mathbf{X}$ are close to being linearly dependent our implementation will be numerically stable.

2.2.3 A Statistical Model - Adding Noise

So far we assumed that the response y was a deterministic function of the sample $\mathbf{x}$, and that there was some deterministic function $f : \mathcal{X} \to \mathcal{Y}$ that underlies the relation $\mathcal{X} \to \mathcal{Y}$. This is an unrealistic assumption - in reality, measurements always contain randomness. In our online store example, we may consider that the revenue y measured for a customer is the sum of a deterministic component $\mathbf{x}^\top\mathbf{w}$ (where $\mathbf{x}$ is the customer's feature vector) and some random component z. This means that our dataset will not look like Figure 2.1 even if it is well described by the linear model. Instead is more likely to look like Figure 2.4.

To address this problem we describe a probabilistic model of the data. Let us assume, as before, that the relation $\mathcal{X} \to \mathcal{Y}$ linear, but with an additional factor capturing randomness in the relation. Suppose now that there exists a function $f : \mathcal{X} \to \mathcal{Y}$ such that the response for sample $\mathbf{x}$ is $y = f(\mathbf{x}) + \varepsilon$ where ε is some random variable. We assume that the noise ε in a sample is identically distributed and independent of the noise in any

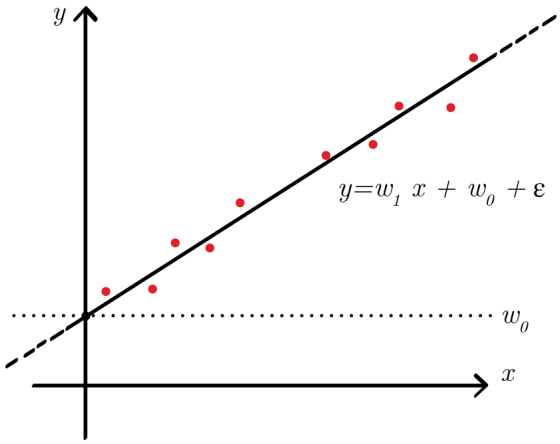


Figure 2.4: *Illustration of a linear regression model where training data is noisy*

other sample. In particular, our training sample S is

$$\left(x_i\,,\,f\left(\mathbf{x}_i\right)+\varepsilon_i\right) \qquad i=1,\ldots,m$$

with $\varepsilon_1,\ldots\varepsilon_m$ being i.i.d: independent and identically distributed. Let us adapt the learning algorithm we designed for the deterministic case to the probabilistic (noisy) case. Let us choose the linear hypothesis class $\mathcal{H}_{reg}$ as before, so that our learning algorithm will output a linear prediction rule. We also assume that we have enough training data to learn, namely that $m \geq d+1$. We assume that there is a vector $\mathbf{w} \in \mathbb{R}^{d+1}$ such that for every sample vector $\mathbf{x}_i$ in our data follows the model:

$$y_i = \mathbf{x}_i^\top \mathbf{w} + \varepsilon_i$$

Denoting the noise vector $\boldsymbol{\varepsilon} := (\varepsilon_1,\ldots,\varepsilon_m)^\top$ we have in matrix notation

$$\mathbf{y} = \mathbf{X}\mathbf{w} + \boldsymbol{\varepsilon}$$

Note that the vector $\mathbf{y}$ will typically not be in $Im(\mathbf{X})$, so that system $\mathbf{y} = \mathbf{X}\mathbf{w}$ has no solutions. As before, using the square loss function, and learning by the empirical risk minimization principal then:

$$\hat{\mathbf{w}} := \underset{\mathbf{w}\in\mathbb{R}^{d+1}}{\operatorname{argmin}} \; ||\mathbf{y}-\mathbf{X}\mathbf{w}||^2$$

This means that our learning algorithm remains the same. We learn by solving the normal equation. As mentioned, $\mathbf{y} \notin Im(\mathbf{X})$ since the noise "pushed" $\mathbf{y}$ out of $Im(\mathbf{X})$. As we have seen, solving the normal equations is equivalent to projecting $\mathbf{y}$ back onto $Im(\mathbf{X})$, so our algorithms effectively attempts to remove the noise and recover the original prediction rule f.

2.2.3.1 An Alternative Approach: Maximum Likelihood

Above we derived an algorithm for learning the linear model using the principle of empirical risk minimization for the square loss. We developed it under the noiseless assumption, but discovered that the algorithm can be used even under noise. Now, let us consider a completely different principle to learning the linear model. Suppose that there is noise, and assume further that the noise is Gaussian $\varepsilon_i \overset{i.i.d}{\sim} \mathcal{N}(0,\sigma^2)$. This means that the i-th observation is independently distributed $y_i \sim \mathcal{N}(\mathbf{x}_i^\top \mathbf{w}, \sigma^2)$. In vector notation, we are assuming that the responses in our training sample follow a multivariate Gaussian distribution:

$$\mathbf{y} \sim \mathcal{N}\left(\mathbf{X}\mathbf{w}, \sigma^2 I_m\right) \tag{2.12}$$

Now, suppose we *knew* the weight vector $\mathbf{w}$, we could then ask the following question: Given a fixed design matrix $\mathbf{X}$ and a known coefficients vector $\mathbf{w}$, what is the probability of observing the response vector $\mathbf{y}$? As each sample is independent to the others, the probability density is the product of the Gaussian densities of each sample

$$p(\mathbf{y}|\mathbf{w}) = \prod_{i=1}^{m} \mathcal{N}\left(y_i|\mathbf{x}_i^\top \mathbf{w}, \sigma^2\right) = \prod_{i=1}^{m} \left[\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(\mathbf{x}_i^\top \mathbf{w} - y)^2}{2\sigma^2}\right)\right] \tag{2.13}$$

This is a question in probability: we know $\mathbf{w}$ and ask for the chance to observe $\mathbf{y}$. However, when we design a learning algorithm, we are actually interested in the reverse question. We have the training sample, including the response vector $\mathbf{y}$. We are interested in a way to choose a linear prediction rule in $\mathcal{H}_{reg}$ and, equivalently, a vector $\mathbf{w}$. We can ask: what is the most "likely" value of $\mathbf{w}$ given the response vector that we observed. This is the Maximum Likelihood approach (subsection 1.1.3) where we choose $\mathbf{w}$ for which the probability density of getting the observed $\mathbf{y}$ is maximal.

As we assumed Gaussian noise, we could express the likelihood function (**??**) of $\mathbf{w}$ using the density function of the Gaussian distribution (1.2.5).

$$\begin{aligned}
\mathcal{L}(\mathbf{w}|X,\mathbf{y}) &= \prod_{i=1}^{m} \mathcal{N}\left(y_i|\mathbf{x}_i^\top \mathbf{w}, \sigma^2\right) \\
&= \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{\left(\mathbf{x}_i^\top \mathbf{w} - y_i\right)^2}{2\sigma^2}\right) \\
&= \frac{1}{\left(2\pi\sigma^2\right)^{m/2}} \prod_{i=1}^{m} \exp\left(-\frac{\left(\mathbf{x}_i^\top \mathbf{w} - y_i\right)^2}{2\sigma^2}\right) \\
&= \frac{1}{\left(2\pi\sigma^2\right)^{m/2}} \exp\left(-\frac{1}{2\sigma^2} \sum_{i=1}^{m} \left(\mathbf{x}_i^\top \mathbf{w} - y_i\right)^2\right)
\end{aligned}$$

Now we can use the likelihood function to derive the MLE for our linear regression model (2.12):

$$\begin{aligned}
\hat{\mathbf{w}}^{MLE} &= \underset{\mathbf{w}}{\operatorname{argmax}}\ \mathcal{L}(\mathbf{w}|\mathbf{y}) \\
&= \underset{\mathbf{w}}{\operatorname{argmax}}\ \log \mathcal{L}(\mathbf{w}|\mathbf{y}) \\
&= \underset{\mathbf{w}}{\operatorname{argmax}}\ \log \exp\left(-\frac{1}{2\sigma^2} \sum_{i=1}^{m} \left(\mathbf{x}_i^\top \mathbf{w} - y_i\right)^2\right) \\
&= \underset{\mathbf{w}}{\operatorname{argmin}}\ \sum_{i=1}^{m} \left(\mathbf{x}_i^\top \mathbf{w} - y_i\right)^2
\end{aligned}$$

We therefore conclude that the maximum likelihood estimator (assuming i.i.d Gaussian noise) gives an *identical* learning algorithm to the one developed using "least squares", namely, using the principle of empirical risk minimization for the square loss.

2.2.4 Categorical Variables

The learning algorithm we have developed can be used whenever $\mathcal{X} = \mathbb{R}^d$ (namely, samples are Euclidean feature vectors) and $\mathcal{Y} = \mathbb{R}$. An issue that often comes up is the following: some of the features we wish to use are numeric (for example, speed in km/h or age in years) while others take values in some discreet set (for example, car manufacturer or house color). To use our model, we must convert all features to numerical values - even those which are non-numeric and take values in a discreet set. In statistics, such features are called *categorical features* or *categorical variables*. As a preliminary step to using the learning algorithm we have developed, which assumes numeric features, we must encode the values of a categorical variable as numerical values.

The set of all possible values that a certain categorical feature can take are called the *levels* of that categirical feature. For example, suppose we look at a training set, where we have some categorical variable describing car color taking values in $\{white, black, red, green\}$. To encode this feature as a numerical feature, we might consider the following idea: we encode each of the levels as some numerical value, and replace the value *white* by 0, *black* by 1, *red* by 2 and *green* by 3. Now, we use this numerical feature vector, obtained using

this encoding, and fit a linear model. This feature will have a linear coefficient and will participate linearly in the linear prediction. However, how should we interpret a coefficient of say 2? Does it mean that as we move from value 0 (white) to value 1 (black) the expected change in the response will increase in 2? What if instead of the mapping above we would have replaced the value *white* by 0, *black* by -1, *red* by 1 and *green* by 3. What would the obtained coefficient mean at this point?

Therefore, when the levels of a categorical variable do not have a natural ordering (colors, for example, do not have a natural order), we cannot simply map each category to a numeric value under the same feature vector. Instead, a categorical variable of k levels turns into $k-1$ binary numeric feature vectors. Each but one of the levels corresponds to one of these new binary numeric features. In our example, we will replace the "color" with three numeric features x_{white}, x_{black} and x_{red}. Then, for a sample where color is *red*, we will have $(x_{white}, x_{black}, x_{red}) = (0,0,1)$ and for a sample where color is *green* (the one level that does not correspond to a new feature), we will have $(x_{white}, x_{black}, x_{red}) = (0,0,0)$. Notice that the feature vectors that we add to the training regression matrix $\mathbf{X}$ are mutually orthogonal. This has advantages statistically and also from a numeric computation perspective. We see that encoding of a categorical feature with k levels into $k-1$ numeric features is defined by a map. This map is known as a *contrast* in the statistical literature.

Two very important considerations to take into account when working with contrasts are: (i) what happens when a test sample contains a level of the categorical variable we have not seen in the training set? (It might be wise to include a level "other" to handle this case). (ii) If the number of levels k is high, we won't want to add $k-1$ new feature vectors to the regression matrix. We might keep the most frequently occurring levels (as seen in the training set), and map all other levels to a new level we may call "other", in order to keep the number of levels in the contrast as small as possible.

Finally, what do we do when the levels have a natural order? A categorical variable whose levels have a natural order is known *ordered categorical variable.* Some examples are: position rank (junior, senior, etc); experience level (no experience, some experience, a lot of experience, master); survey items opinion items such as "I agree with this statement" (do not agree, neutral, agree somewhat, agree, strongly agree), and so on. For ordered categorical variables, it makes sense to use a more sophisticated contrast than the simple binary contrast described above - perhaps a contrast that will capture the order between the levels.

2.3 Coefficient of Determination - R^2

When fitting a linear regression model we search for $\mathbf{w} \in \mathbb{R}^{d+1}$ which minimizes the RSS

$$\underset{\mathbf{w}\in\mathbb{R}^d}{\operatorname{argmin}} \operatorname{RSS}(\mathbf{w}) = \underset{\mathbf{w}\in\mathbb{R}^d}{\operatorname{argmin}} ||\mathbf{y} - \mathbf{X}\mathbf{w}||^2 \tag{2.14}$$

Notice however, that for a given model $\mathbf{w}$ and a sample $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$, the quantity $\operatorname{RSS}(\mathbf{w})$ is not very informative as it has an arbitrary range, determined by the scaling of the data. As such, we cannot tell if an RSS of, say, $10,000$ represents a successful fit to the training data. If the scaling of the data is in 0.1 this is an enormous error, but if the scaling of the data is in the millions then this error is not too bad. We would therefore like to derive some quantity with an intuitively meaningful range of values, which is insensitive to the scaling of the data.

The R-squared (R^2) provides a measure of the goodness of fit of a model. It measures how good are the model predictions in approximating the real data and is defined as the fraction of variance of $\mathbf{y}$ explained by the model

$$R^2 := 1 - \frac{\text{Unexplained Variation}}{\text{Total Variation}} = 1 - \frac{\text{SSE}}{\text{SST}} \tag{2.15}$$

The unexplained variation is the sum of the squared errors (SSE), while the total variation is total sum of squares (SST)

$$\text{SSE} := \sum (y_i - \hat{y}_i)^2, \quad \text{SST} := \sum (y_i - \bar{y})^2 \tag{2.16}$$

It holds that SSE $\leq$ SST and thus $R^2 \in [0,1]$, providing an intuitive measure of how good is the fit. If $R^2 = 1$ it means that the amount of unexplained variation in y is zero, namely a perfect fit. If $R^2 = 0$ it means our model does not explain any of the observed variation in y.

2.3.1 Connection With Correlation Coefficient

An interesting observation regarding the R^2 is its connection to the Pearson correlation coefficient. The Pearson correlation coefficient is defined as follows:

$$\rho_{A,B} \coloneqq \frac{cov(A,B)}{\sigma_A \sigma_B} \tag{2.17}$$

This measure, denoted also as r, captures the strength and direction of the *linear* relation between the two random variables A and B. Its values range between $[-1,1]$ where a value of 1 indicates a perfect linear relation between A and B, a value of 0 indicates no linear relation between A and B, and a value of -1 indicates a perfect linear anti-relation between A and B (i.e. $A = -B$).

It holds, that in the case of a linear LS regression with an intercept and a *single* variate then $R^2 = \rho^2_{y,x}$. In the multivariate case then $R^2 = \rho^2_{y,\hat{y}}$.

2.4 Polynomial fitting

Next, we would like to address the question of what type of functions we can learn using the linear regression model. Could we, for example, use the tools developed above, to learn the function $p(x)$ seen in Figure 2.5? And then, given some new value of x predict the value of $p(x)$?

Clearly this function is not a linear function. However, if we look closely at this polynomial $y = x^3 - 3x^2 + \frac{1}{2}x + 2$ and think of it not as a function of x but rather as a function of the coefficients of the polynomial, we realize that this is indeed a linear function. This function simply does not take the original values of x but instead some transformation of the original values: $x \mapsto \left(1, x, x^2, x^3\right)^\top$.

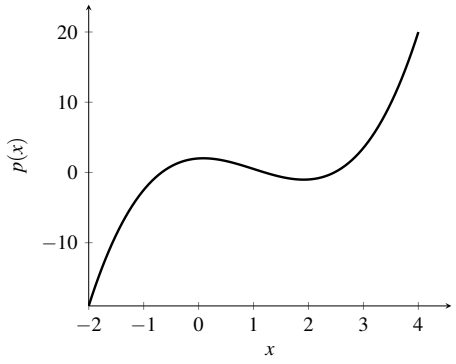


Figure 2.5: *A univariate polynomial* $p(x) = x^3 - 3x^2 + \frac{1}{2}x + 2$

To formulate the above understanding let $\psi_0, \ldots, \psi_k : \mathbb{R}^d \to \mathbb{R}$ be a set of functions such that each ψ_j receives a sample $\mathbf{x} \in \mathbb{R}^d$ and outputs a single value - namely, each ψ_j computes a single feature. Now, using these functions, that are referred to as *basis functions*, we can describe a relation that is *linear* in the parameters $\mathbf{w}$ but could be non-linear in the original input data:

$$y = \sum_{j=0}^{k} \psi_j(\mathbf{x}) \cdot w_j = \boldsymbol{\psi}(\mathbf{x})^\top \mathbf{w}, \qquad \boldsymbol{\psi}(\mathbf{x}) \coloneqq (\psi_0(\mathbf{x}), \ldots, \psi_k(\mathbf{x}))^\top \tag{2.18}$$

For the specific case of (univariate) polynomial fitting we would like to describe a polynomial relation between $\mathcal{X} = \mathbb{R}$ and $\mathcal{Y} = \mathbb{R}$ of degree at most $k \in \mathbb{N}$, but linear in the coefficients. The hypothesis class fitting this relation is:

$$\mathcal{H}^k_{poly} = \left\{ x \mapsto p_{\mathbf{w}}(x) \,|\, \mathbf{w} \in \mathbb{R}^{k+1} \right\} \tag{2.19}$$

where $p_{\mathbf{w}}(x) = \sum_{i=0}^{k} w_i x^i$. Thus, the set of basis functions is $\psi_j(x) = x^j$ for any $j \in \{0,\ldots,k\}$. As before, given a training sample $S = \{(x_i, y_i)\}_{i=1}^m$ we would like to choose a coefficients vector $\mathbf{w}$, best describing the coefficients of the polynomial. To do so we solve, as before, the LS problem:

$$\hat{\mathbf{w}} := \underset{\mathbf{w}\in\mathbb{R}^{k+1}}{\operatorname{argmin}} \frac{1}{m} \sum \left(y_i - p_{\mathbf{w}}(x_i)\right)^2 = \underset{\mathbf{w}\in\mathbb{R}^{k+1}}{\operatorname{argmin}} \frac{1}{m} \sum \left(y_i - \psi(x_i)^\top \mathbf{w}\right)^2 \tag{2.20}$$

Notice that the design matrix $\mathbf{X}$, defined over the transformation $\psi(\mathbf{x})$, is the Vandermonde matrix:

$$\mathbf{X} := \begin{bmatrix} — & \psi(x_1) & — \\ — & \psi(x_2) & — \\ & \vdots & \\ — & \psi(x_m) & — \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^k \\ 1 & x_2 & x_2^2 & \cdots & x_2^k \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_m & x_m^2 & \cdots & x_m^k \end{bmatrix}$$

Since we assume that the x_i's are different from one another, the design matrix $\mathbf{X}$ is of full rank. This means that solving this linear (in $\mathbf{w}$) system of equations can be done as we have seen for the non-singular case above. After finding $\hat{\mathbf{w}}$, we can predict the value of the unknown function p at a new point x using the value $p_{\hat{\mathbf{w}}}(x)$.

R Here we discuss polynomial fitting where $\mathcal{X} = \mathbb{R}$. With very little adaptation, we could also allow the input data to be $\mathcal{X} = \mathbb{R}^d$, $d > 1$. In such cases the defined polynomial could include terms of multiplication of two (or more) features. We will encounter such an example in 9.

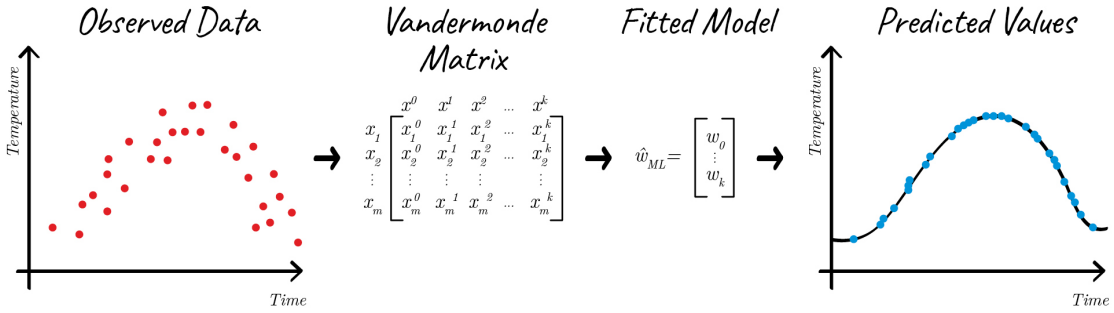


Figure 2.6: Scheme of Polynomial Fitting

2.4.1 Bias and Variance

In the sections above we have seen that given a regression problem $\mathbf{X}, \mathbf{y}$, where $\mathbf{y} = \mathbf{X}\mathbf{w} + \varepsilon$, $\varepsilon \sim \mathcal{N}\left(0, \sigma^2 I_m\right)$, the estimator $\hat{\mathbf{w}} \in \mathbb{R}^d$ minimizing $||\mathbf{y} - \mathbf{X}\hat{\mathbf{w}}||$ is given by $\hat{\mathbf{w}} := \mathbf{X}^\dagger \mathbf{y}$. It is important to notice that as $\mathbf{y}$ is a random variable, the LS estimator is too a random variable. Therefore, let us look at different properties of this estimator, specifically the *bias* (1.1.6) and *variance* (1.1.8):

$$Bias\left(\hat{\theta}\right) = \mathbb{E}\left[\hat{\theta}\right] - \theta, \quad Var\left(\hat{\theta}\right) = \mathbb{E}\left[\left(\hat{\theta} - \mathbb{E}\left[\hat{\theta}\right]\right)^2\right]$$

Both these properties involve calculating the expectation of $\hat{\theta}$, but over what is the expectation calculated? An estimator is a decision function over a sample, used to estimate some parameter. Therefore, the expectation is over the selection of the samples:

$$Bias\left(\hat{\theta}\right) = \mathbb{E}_S\left[\hat{\theta}\right] - \theta, \quad Var\left(\hat{\theta}\right) = \mathbb{E}_S\left[\left(\hat{\theta} - \mathbb{E}_S\left[\hat{\theta}\right]\right)^2\right]$$

for $S \overset{i.i.d}{\sim} (\mathcal{X} \times \mathcal{Y})^m$. As we are not assuming any probability distribution over $\mathcal{X}$ this is equivalent to calculating the expectation over the sampling of $\varepsilon \sim \mathcal{N}\left(0, \sigma^2 I_m\right)$ and obtaining $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ where $y_i = \mathbf{x}_i^\top \mathbf{w} + \varepsilon_i$. And so, going back to the LS estimator, it can be shown that this is an *unbiased* estimator with variance of $\sigma^2\left[\mathbf{X}^\top\mathbf{X}\right]^{-1}$.

To understand how bias and variance help quantify the quality of our estimation, let us revisit polynomial fitting. Consider for example the polynomial

$$Y = X^4 - 2X^3 - 0.5X^2 + 1 + \varepsilon \quad \varepsilon \sim \mathcal{N}(0,2) \tag{2.21}$$

and let $x_1, \ldots, x_m$ be a set of samples where $x_i \in [-2,2]$. For these observations let us create 10 different datasets, generated according to the model above. For each dataset we use $x_1, \ldots, x_m$ and generate the response value $y_1, \ldots, y_m$ with the addition of the noise. Figure Figure 2.7 shows the different datasets generated by the model above and the fitted polynomial of degree 1. Black, red and blue points represent the true model, the observed data-points (with the sample noise) and the fitted model over the observed data-points. Notice how the different datasets yield different predicted models. This is the randomness of the prediction, driven by the randomness of the trainset. Over these datasets we can now ask, for each value of x, what is the average prediction and its variance:

- In green is the average prediction of y for a given x across all datasets. The difference between the green and black lines capture the concept of the bias.
- In grey is the area of $\mathbb{E}[\hat{y}] \pm 2 \cdot Var(\hat{y})$ for a given x, also known as the confidence interval. The wider this area is, the more out prediction of $\hat{y}$ varies for different samples. This area captures the concept of the variance.

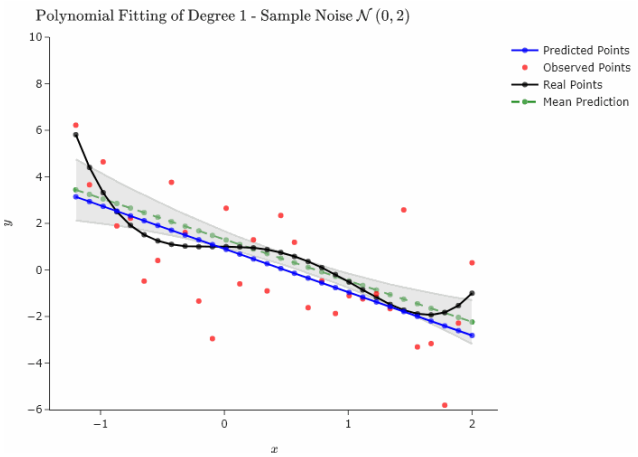


Figure 2.7: ***Polynomial Fitting:*** *Fitted polynomial of degree* 1 *over different datasets differing only in values of added sample noise.* *Regression Meothods Examples*

Two phenomena are visible. The first is that the average distance of the fitted model (in green) and the true model (in black) is large. This means that our hypothesis class doesn't have sufficient expressive power to

learn the true model. As such, we conclude that the *bias* of our estimator is high. The second is that the fitted models over different datasets do not differ by much. As such, we conclude that the *variance* of our estimator is low.

Next, consider the same setup as before but with the fitting of a polynomial of degree 8 (Figure 2.8). This time the different between the average prediction at each x and the true value of x is lower, while the differences between the fitted models (as indicated by the confidence intervals) is much higher. So the **bias** is low and the **variance** is high. As we enable more "flexible" (i.e. complex) models we are able to fit a model better to our given sample. However, as seen in Figure 2.8, if the model is too complex we might actually be fitting a model to the noise, rather than the actual true signal.

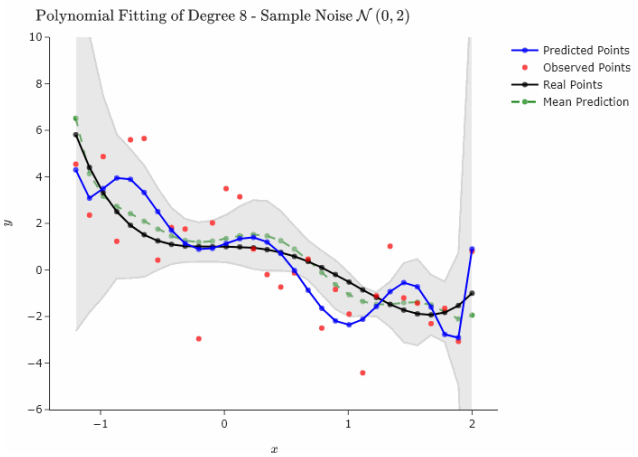


Figure 2.8: ***Polynomial Fitting:*** *Fitted polynomial of degree* 8 *over different datasets differing only in values of added sample noise. Regression Meothods Examples*

R It is important to note that what is seen in the figures are not the bias and variance of $\hat{\mathbf{w}}^{LS}$ themselves but how these manifest over the shown datasets.

Interestingly, these two properties of bias and variance are linked. Let $\hat{\mathbf{y}} = \hat{\mathbf{y}}(S)$ denote the estimator of $\mathbf{y}$ when using $\hat{\mathbf{w}}^{LS}$, and $\mathbf{y}^*$ the true $\mathbf{y}$ values. When solving the regression problem we wanted to minimize the mean square error between $\hat{\mathbf{y}}$ and $\mathbf{y}^*$. What would be the expected MSE value?

Denote $\overline{\mathbf{y}} = \mathbb{E}[\hat{\mathbf{y}}]$ so:

$$\begin{aligned}
\mathbb{E}\left[(\hat{\mathbf{y}} - \mathbf{y}^*)^2\right] &= \mathbb{E}\left[(\hat{\mathbf{y}} - \overline{\mathbf{y}} + \overline{\mathbf{y}} - \mathbf{y}^*)^2\right] \\
&= \mathbb{E}\left[(\hat{\mathbf{y}} - \overline{\mathbf{y}})^2\right] + 2(\hat{\mathbf{y}} - \overline{\mathbf{y}})\,\mathbb{E}[\hat{\mathbf{y}} - \overline{\mathbf{y}}] + (\overline{\mathbf{y}} - \mathbf{y}^*)^2 \\
&= \mathbb{E}\left[(\hat{\mathbf{y}} - \overline{\mathbf{y}})^2\right] + (\overline{\mathbf{y}} - \mathbf{y}^*)^2 \\
&= Var(\hat{\mathbf{y}}) + Bias^2(\hat{\mathbf{y}})^2
\end{aligned} \tag{2.22}$$

Namely, we could *decompose* the generalization error (expected square loss between prediction and true value) into a variance component and a (squared) bias component.

$$MSE(\hat{\mathbf{y}}) = Var(\hat{\mathbf{y}}) + Bias^2(\hat{\mathbf{y}}) \tag{2.23}$$

This means, that whenever we devise some estimator over our training data, the generalization error is influenced by both these factors. This is called the *Bias-Variance Trade-off*.

2.5 Summary and Exercises

Exercises

Theoretical Questions

1. Let $\mathbf{A}$ be some matrix. Show that $\text{Ker}(\mathbf{A}) = \text{Ker}\left(\mathbf{A}^\top \mathbf{A}\right)$.
2. Let $\mathbf{X}$ be a design matrix with independent columns. Prove that $\mathbf{X}^\top \left[\mathbf{X}^\top \mathbf{X}\right]^{-1} \mathbf{X}^\top$ is an orthogonal projection matrix onto the column space of $\mathbf{X}$.
3. Let $\mathbf{A}$ be an invertible matrix. Show that $\mathbf{A}^\dagger = \mathbf{A}^{-1}$.
4. Let $\mathbf{X}, \mathbf{y}$ be a linear regression problem where the columns of $\mathbf{X}$ are linearly independent. Show that $\left[\mathbf{X}^\top \mathbf{X}\right]^{-1} \mathbf{X}^\top = {\mathbf{X}^\dagger}^\top \mathbf{y}$.
5. Let $\mathbf{X}, \mathbf{y}$ be a regression problem such that $\mathbf{y} = \mathbf{X}\mathbf{w} + \varepsilon,\ \varepsilon \sim \mathcal{N}\left(0, \sigma^2 I_m\right)$ and $\hat{\mathbf{w}} := \mathbf{X}^\dagger \mathbf{y}$ the LS estimator. Show that $\hat{\mathbf{w}}$ is an unbiased estimator.
6. Let $\mathbf{X}, \mathbf{y}$ be a regression problem such that $\mathbf{y} = \mathbf{X}\mathbf{w} + \varepsilon,\ \varepsilon \sim \mathcal{N}\left(0, \sigma^2 I_m\right)$ and $\hat{\mathbf{w}} := \mathbf{X}^\dagger \mathbf{y}$ the LS estimator. Prove that the variance of $\hat{\mathbf{w}}$ is $\sigma^2 \left[\mathbf{X}^\top \mathbf{X}\right]^{-1}$.

Practical Questions

Fitting A Linear Regression Model

In this part you will fit a linear regression model to the House Sales dataset.

1. Implement the `mean_square_error` function in the `metrics.loss_functions.py` file as described in the function documentation.

2. Implement the `LinearRegression` class in the `learners.regressors.linear_regression.py` file as described in the documentation. When implementing the `_loss` function be sure to call the `mean_square_error` function previously implemented.

3. Implement the `load_data` function in the `house_sales_prediction.py` and call the function. The function receives the path to the house sales dataset and returns a design matrix and response vector after performing any necessary preprocessing:
   - Addressing missing/invalid values for different features.
   - Treating categorical features.
   - Adding additional features that might be beneficial for predicting the price of the house.
   - Remove irrelevant features.

   Elaborate on the data exploration and decision making process that lead to the final preprocessing code.

4. Implement the `split_train_test` function in the `utils.utils.py` file as described in the function documentation. Call the function in the `house_sales_prediction.py` file, splitting the loaded dataset into a train set (75%) and test set (25%).

5. After splitting the dataset, fit a linear regression model (using your implementation of the `LinearRegression` class) over increasing percentages of the *train set* and evaluate performance over the *test set*. Do so in the following manner:
   - Iterate for every value $p = 10\%, 11\%, \ldots, 100\%$ of the train set samples.
   - Sample p of the train set, fit a model over these samples and evaluate performance over the test set.
   - Repeat the procedure of sampling, fitting and evaluating 10 times for every value of p.

   Plot the average loss, as well as a confidence interval of $\text{mean}(loss) \pm 2 \cdot \text{std}(loss)$, as a function of $p\%$. Explain the results seen in the plot.

6. Repeat the procedure of splitting the train- and test-sets and of fitting the model for different sample sizes (i.e questions 4,5) multiple times. Plot the average of loss averages as well as the confidence interval as a function of $p\%$. Explain what is the difference between the current evaluation approach

and the previous.

Performing Polynomial Fitting

In this part you will perform polynomial fitting to the Daily Temperature of Major Cities dataset.

1. Implement the `PolynomialFitting` class in the `learners.regressors.polynomial_fitting.py` file as specified in class documentation. Avoid repeating code from the `LinearRegression` class.

2. Implement the `load_data` function in the `city_tempretaure_prediction.py` file.
   - When loading the dataset remember to deal with invalid data.
   - Use the `parse_dates` argument of the `pandas.read_csv` to set the type of the `'Date'` column.
   - Add a `'DayOfYear'` column based on the `'Date'` column. This column will be the feature to be used for the polynomial fitting.

3. Subset the dataset to contain samples only from your own country. Investigate how the average daily temperature (`'Temp'` column) change as a function of the `'DayOfYear'`.
   - Scatter plot this relation, and color code the data points by the different years. What polynomial degree might be suitable for this data?
   - Group the samples by `'Month'` and plot a bar plot showing for each month the standard deviation of the daily temperatures. Suppose you fit a polynomial model (with the correct degree) over data sampled uniformly at random from this dataset, and then use it to predict temperatures from random days across the year. Based on this plot, do you expect a model to succeed equally over all months or are there times of the year where it will perform better than on others?

4. Subset the dataset to contain samples from 4 countries, your own, a second in the same hemisphere, and two others in the opposite hemisphere. Group the samples according to `'Country'` and `'Month'` and calculate the average and standard deviation of the temperature. Plot a line plot of the average monthly temperature, with error bars (using the standard deviation) color coded by the country. Based on this graph, do all countries share a similar pattern? For which other countries is the model fitted for your own country likely to work well and for which not?

5. Over the subset containing observations only from your own country perform the following:
   - Randomly split the dataset into a training set (75%) and test set (25%).
   - For every value $k \in [1,10]$, fit a polynomial model of degree k using the training set.
   - Record the loss of the model over the test set, rounded to 2 decimal places.

   Print the test error recorded for each value of k. In addition plot a bar plot showing the test error recorded for each value of k. Based on these which value of k best fits the data? In the case of multiple values of k achieving the same loss select the simplest model of them. Are there any other values that could be considered?

6. Fit a model over the entire subset of records from your own country (i.e do not split for a train- and test set) using the k chosen above. Plot a bar plot showing the model's error over each of the other countries. Explain your results based on this plot and the results seen in question 3.

3. Classification

3.1 Classification Overview

In the previous chapter we discussed learning a regression problem where the response is a continuous value $\mathcal{Y} = \mathbb{R}$. When the response set $\mathcal{Y}$ is a finite set, this is a **classification** problem. We distinguish between classification problems where $|\mathcal{Y}| = 2$ (such as $\mathcal{Y} = \{\pm 1\}$ or $\mathcal{Y} = \{0,1\}$) and multi-classification problems where $\mathcal{Y} = \{1,\ldots,k\}$. In the binary classification problem (or just "classification"), we provide a "yes"/"no" prediction. In a multi-class classification, we predict one of $k > 2$ classes. For most, we restrict out discussion only to binary classification problems, though all methods below can be generalized to k classes. Also, we will only deal with the Euclidean sample space $\mathcal{X} = \mathbb{R}^d$, namely, each sample has d **features**. Therefore out setup is as follows:

$$\mathcal{X} := \mathbb{R}^d, \mathcal{Y} := \{\pm 1\}$$

Classification Problems Examples

- Predict whether a patient will develop a certain medical condition, or not.
- Predict whether a user will like a new product, or not.
- Determine if a given network traffic pattern is one of a cyber attack or not.
- Determine whether an art work is an original or forged.
- Determine whether a given email is spam or not.
- Detect fraud on credit card transactions.
- Predict whether a loan applicant will default on the loan.
- (Multi-class) What are the objects seen in a given picture.

■ **Example 3.1** Seen in Figure 3.1 are samples of the "South Africa Heart Disease" dataset. Given the parameters of blood pressure, smoking, family history, etc., could we predict who has/will have coronary heart disease (chd)? Notice that some of the features are numerical (e.g. tabacco, ldl, etc.) while some are categorical (e.g famhist). ■

Visualizing The Feature Space

When given a learning problem it is important to try and get intuition into "what the data looks like". In the case of a training sample for binary classification, we can plot the different axes and color by the label

	sbp	tobacco	ldl	adiposity	famhist	typea	obesity	alcohol	age	chd
0	160	12.00	5.73	23.11	Present	49	25.30	97.20	52	1
1	144	0.01	4.41	28.61	Absent	55	28.87	2.06	63	1
2	118	0.08	3.48	32.28	Present	52	29.14	3.81	46	0
3	170	7.50	6.41	38.03	Present	51	31.99	24.26	58	1
4	134	13.60	3.50	27.78	Present	60	25.99	57.34	49	1
...	...	...	...	...	...	...	...	...	...	...
457	214	0.40	5.98	31.72	Absent	64	28.45	0.00	58	0
458	182	4.20	4.41	32.10	Absent	52	28.61	18.72	52	1
459	108	3.00	1.59	15.23	Absent	40	20.09	26.64	55	0
460	118	5.40	11.61	30.79	Absent	64	27.35	23.97	40	0
461	132	0.00	4.82	33.41	Present	62	14.70	0.00	46	1

Figure 3.1: Example classification dataset: *South African Heart Data from Elements of Statistical Learning*

(Figure 3.2). This task is more difficult for data of higher dimensions, but attempting to imagine it in such cases will help understand what models might fit better to the specific task.

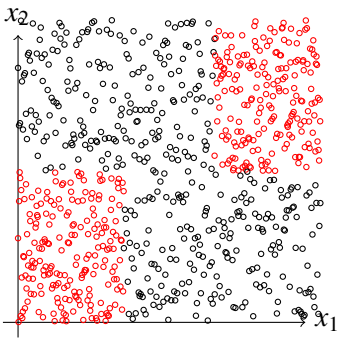


Figure 3.2: Classification training sample in $\mathbb{R}^2$***:*** *Where samples are positioned in space according to the values of their features and color coded by their label.*

3.1.1 Type-I and Type-II Errors

When we discussed regression problems we decided to measure the performance of a given hypothesis using the square loss (and mentioned that we could also use the absolute loss). In the case of classification our hypothesis outputs a label $\{\pm 1\}$ which we want to compare with the true labels also in $\{\pm 1\}$. It therefore makes less sense to measure "how far" is the prediction from the true value (as we do using the squared loss). Instead we would like to measure if we were correct or not. A very straight forward way to evaluate the performance of a classification predictor is to simply count the number of correctly classified samples. That is, given a prediction rule $h : \mathcal{X} \to \{\pm 1\}$ and a labeled sample $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$, the *misclassification* loss of h on this sample is:

$$L_S(h) := \sum_{i=1}^{m} \mathbb{1}_{y_i \neq h(\mathbf{x}_i)} = |\{i | y_i \neq h(\mathbf{x}_i)\}| \tag{3.1}$$

Can there be any problems or issues with the misclassification loss? After all, it just counts the number of times h was wrong - the number of times h misclassified a sample. In practice, there are two kinds of errors the

classifier can make, and making each kind of error might have very different implications, or costs. Therefore, simply counting the total number of errors may not be a useful performance measure.

■ **Example 3.2 — Credit Decisions.** Suppose we are building a classifier that predicts whether a bank customer seeking a loan is credit-worthy and will return a given loan or not. We choose the labels such that -1 means "not credit worth - deny loan", and 1 means "credit worthy - approve loan". Denote y_i the true label and $\hat{y}_i$ the classifier-predicted label of sample i. The two errors this classifier might make have very different consequences:

- If $y_i = -1$ and $\hat{y}_i = 1$, the classifier predicted that a non-credit-worthy customer will return the loan. If we act on this prediction, and the customer defaults on the loan, the bank loses all the loan sum.
- On the other hand, if $y_i = 1$ and $\hat{y}_i = -1$, the classifier predicted that a credit-worthy customer, which would have paid the interest and returned the loan in full, is not credit-worthy and should be denied the loan. If we act on this recommendation, the bank loses the interest it would have earned on the loan.

Which of the two errors is more serious? Which of the two errors cost more for the bank? If we could choose which error should we avoid "at all costs" and which error could we "allow to happen", what would we choose? ■

■ **Example 3.3 — Drug safety.** Let us look at a more extreme example to help illustrate this point. We are creating a classifier to predicts whether a certain drug **is safe** to use for a particular person, or **unsafe/** deadly/ dangerous to use. We choose the labels such that -1 means "unsafe drug - do not use" and 1 means "safe drug - ok to use". Similar to before our errors are:

- If $y_i = -1$ and $\hat{y}_i = 1$, the classifier recommends to give a drug which is actually potentially deadly.
- If $y_i = 1$ and $\hat{y}_i = -1$, the classifier recommends that the patient should avoid a drug which is actually safe to use.

■

Therefore, we see that depending on the context of the classification problem, the two kinds of errors can have very different costs. We name the first error, the one we would like to avoid at all costs, the *Type-I error* and the second error as *Type-II error*. By choosing what label is "negative" and what label is "positive" we essentially defined what error is the Type-I error. As such, given a classification problem we try to choose the "negative" and "positive" labels such that the error we are more concern of (and therefore would like to avoid more) is the Type-I Error. That is, the error of misclassifying a negative sample by predicting it as a positive sample.

Returning to the drug safety example 3.3, we can assign the following meaning to the labels: $y = -1$ (negative) means the new drug is safe to use and $y = 1$ (positive) means the new drug is dangerous. In this case the Type-I error means that we decided not to offer a safe drug. If however we reverse the meaning such that $y = -1$ (negative) means the drug is dangerous and $y = 1$ (positive) means the new drug is safe, then the Type-I error means that we have decided to offer a dangerous drug. In this case this assignment of labels is the more serious of the two kinds of errors we can make.

For the classification problem of "is this email spam or not" how would you choose the labels? What are the two errors a spam detector can make? which one is the one we really want to avoid? So, which of the labels "spam email" and "not spam, valid email" would you label "negative" and which is "positive"?

3.1.2 Measurements of performance

With the decision on "positive" and "negative" labels, we define four basic terms: True Positive (TP), False Positive (FP), True Negative (TN) and False Negative(FN). These terms refer to the prediction made for a sample with respect to its true label. So suppose a sample's true label is $y = -1$ (negative). If a classifier predicts:

- $\hat{y} = -1$ we term this as true negative.

- $\hat{y} = 1$ we term this as false positive.

Now, suppose a sample's true label is $y = 1$ (positive) then if a classifier predicts:

- $\hat{y} = -1$ we term this as false negative.
- $\hat{y} = 1$ we term as true positive.

Therefore, the false negative and false positive cases are the misclassification errors. False positive is what we referred to as Type-I error and false negative is what we called Type-II error. These four options are shown in **??**.

Using these four basic groups we can devise more domain-specific measurements. Denote by P the number of positive samples and N the number of negative samples then:

- The *Error Rate* is the number of misclassification out of all predictions: $(FP+FN)/(P+N)$.
- The *Accuracy* is the number of correct classification out of all predictions: $(TP+TN)/(P+N) = 1-ErrorRate$.
- The *Recall/ Sensitivity/ True-Positive-Rate (TPR)* is the number of truthfully positive predictions out of all positive samples: TP/P.
- The *False-Positive-Rate (FPR)* is the number of of falsely positive predictions out of all negative samples: FP/N.

There are many more measurements that can be defined from the four basic ones presented with different fields using different measurements. In Computer Science we often encounter the TPR and FPR for reasons described below.

3.1.3 Decision Boundaries

Let h be a binary classification rule in $\mathbb{R}^d$. (Suppose, for example, that we used a training sample to select h from some hypothesis class $\mathcal{H}$). We can feed any point $\mathbf{x} \in \mathbb{R}^d$ into h and get one of two classes. This means that we can view $\mathbb{R}^d$ as disjoint union of two sets:

$$\mathbb{R}^d = \{\mathbf{x}|h(\mathbf{x}) = 1\} \biguplus \{\mathbf{x}|h(\mathbf{x}) = 0\}$$

These sets can be very simple (two half-spaces) or very complicated. The boundary between these two sets is called the *decision boundary*: a test sample on one side of the boundary will be classified to one class by h, and a test sample on the other side of the boundary will be classified to the other class.

Different classifies, derived from different hypothesis classes, will generate different decision boundaries (Figure 3.3). Observing these over different data scenarios is helpful to understand is modeled by the different classifiers. It can also help get a qualitative assessment of the bias and variance of the classifiers.

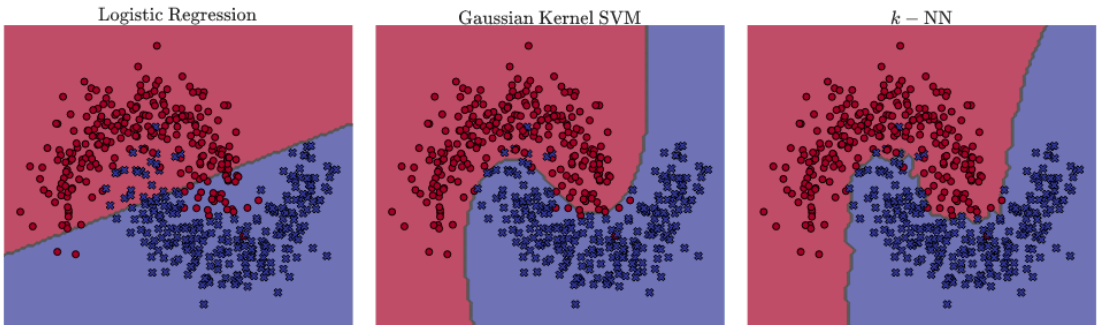


Figure 3.3: Decision Boundaries *of classifiers fitted over moons dataset. Classification Examples*

3.1.4 Studying A New Classifier

For the rest of this chapter we will discuss different sorts of classifiers. As there are numerous types of classifiers, each for tailored for specific data scenarios, it is important to understand how to read about a new classifier. Therefore, when going over the classifiers below keep in mind the following guiding questions:

- How does it model the classification problem? and what are the assumptions made on the data?
- What is the hypothesis class defined? and how does the decision boundary looks like?
- What is the learning principle we use? and how does the algorithm match the learning principle?
- How can the learning principle can be implemented computationally? What is the time complexity of the algorithm and are there any considerations of numeric stability?
- What is done in the training step? and how, given a trained model, to predict for new samples?
- Is the model interpretable? Are we provided with estimations of class probabilities?
- Are we facing a single model or rather a family of models with some parameters for choosing specific models from this family? How do these parameters affect the bias-variance tradeoff?
- When will we decide to use this learning algorithm? What are its advantages and disadvantages?

3.2 Half-Space Classifier

Similar to linear regression, one of the simplest families of classifiers is that of linear classifiers. In these, we are interested in separating a given dataset into two classes using a linear separator function, as seen in Figure 3.4. It will be convenient to work with the class labels of $\mathcal{Y} := \{\pm 1\}$.

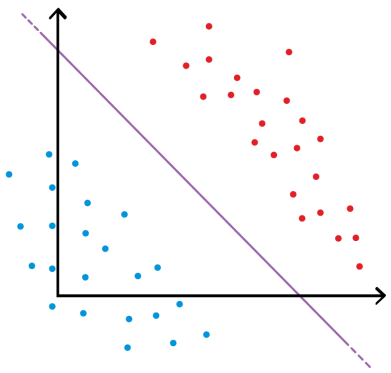


Figure 3.4: Half-space Classification Illustration: *For a domain-set $\mathcal{X} \in \mathbb{R}^2$ the two classes, coded as red and blue colors, are linearly separable .*

Similar to the definition used in linear regression (2.2), the family of linear functions can be described as the set of functions of the form $\mathbf{x} \mapsto \mathbf{x}^\top \mathbf{w} + b,\ \ \mathbf{w} \in \mathbb{R}^d, b \in \mathbb{R}$. The linearity refers to the functions being linear in the parameters $\mathbf{w}$. Unlike in the regression setup, here we are interested in a mapping to a discrete response value.

TODO: Add illustration of transformation To define a linear separator we begin with defining a hyperplane. For $\mathbf{w} \in \mathbb{R}^d$ and $b \in \mathbb{R}$, the *hyperplane* $(\mathbf{w}, b)$ is the set of points $\{\mathbf{x} | \langle \mathbf{w}, \mathbf{x} \rangle + b = 0\}$. Using a hyperplane we can then state if a given point is "below" or "above" it, and thus separate $\mathbb{R}^d$ into two groups (i.e classes). To understand *how* does a hyperplane perform such separation, consider the transformation that it does to $\mathbf{v} \in \mathbb{R}^d$. Let $(\mathbf{w}, b)$ be a hyperplane where w.l.o.g let $b = 0$. Recall that the orthogonal projection of $\mathbf{v}$ onto $\mathbf{w}$ is given by $\langle \mathbf{v}, \hat{\mathbf{w}} \rangle \hat{\mathbf{w}}$ for $\hat{\mathbf{w}} := \mathbf{w}/||\mathbf{w}||$. Equivalently, this can be written as $\hat{\mathbf{w}}\hat{\mathbf{w}}^\top \mathbf{v}$ where $\hat{\mathbf{w}}\hat{\mathbf{w}}^\top$ is the orthogonal projection matrix onto the (one-dimensional) subspace spanned by $\hat{\mathbf{w}}$, and $\hat{\mathbf{w}}^\top \mathbf{v}$ is the *coordinate* of $\mathbf{x}$ *in* (i.e. once embedded) the subspace. Once we have embedded $\mathbf{v}$ in this subspace, we can now ask where is

its embedding relative to the hyperplane. Notice that for points on the hyperplane itself, it holds that their embedding $\mathbf{v}^\top \mathbf{w}$ equals to zero. We define the *positive-* and *negative half-spaces* of $(\mathbf{w}, b)$ as the sets of points

$$HS_+ \coloneqq \{\mathbf{x} | \langle \mathbf{w}, \mathbf{x} \rangle + b > 0\}, \qquad HS_- \coloneqq \{\mathbf{x} | \langle \mathbf{w}, \mathbf{x} \rangle + b < 0\} \tag{3.2}$$

where together with the hyperplane it holds that $\mathbb{R}^d = HS_+ \uplus \{\mathbf{x} | \mathbf{x}^\top \mathbf{w} = 0\} \uplus HS_-$. Following the same reasoning as above, we now see that the embedding of points $\mathbf{v} \in HS_+$, given by $\mathbf{v}^\top \mathbf{w}$ will be positive and the embedding of points $\mathbf{v} \in HS_-$, given by $\mathbf{v}^\top \mathbf{w}$ will be negative. Using an equivalent way to define the halfspaces

$$HS_+ \coloneqq \{\mathbf{x} | \text{sign}(\langle \mathbf{w}, \mathbf{x} \rangle + b) > 0\}, \qquad HS_- \coloneqq \{\mathbf{x} | \text{sign}(\langle \mathbf{w}, \mathbf{x} \rangle + b) < 0\} \tag{3.3}$$

we can define the hypothesis class of half-spaces in $\mathbb{R}^d$.

$$\mathcal{H}_{half} \coloneqq \left\{ h_{\mathbf{w},b}(\mathbf{x}) \coloneqq \text{sign}\left(\mathbf{x}^\top \mathbf{w} + b\right) \mid \mathbf{w} \in \mathbb{R}^d, b \in \mathbb{R} \right\} \tag{3.4}$$

The case where $b = 0$ is called the *homogeneous* case, as the hyperplane is a linear subspace going through the origin. When $b \neq 0$ the hyperplane does not go through the origin and is called the non-homogeneous case.

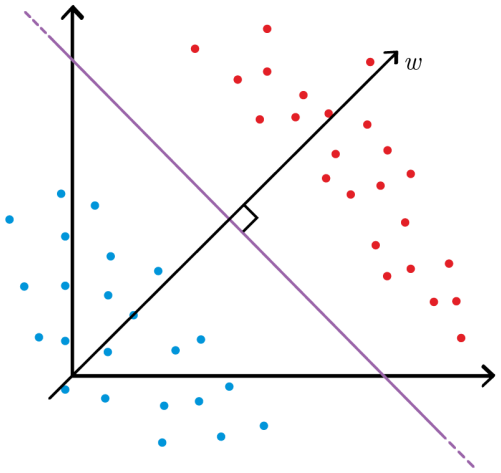


Figure 3.5: *Corresponding Hyperplane to* $\mathbf{w}^\perp$

R Notice, that by definition the hyperplane $\{\mathbf{x} | \mathbf{x}^\top \mathbf{w} = 0\}$ is the set of points perpendicular to $\mathbf{w}$, that is $\mathbf{w}^\perp = \{\mathbf{x} | \mathbf{x}^\top \mathbf{w}\}$. As such, each vector $\mathbf{w} \in \mathbb{R}^d$ defines a hyperplane $\mathbf{w}^\perp$ that divides $\mathbb{R}^d$ into two half-spaces.

Given a sample $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$, we would like to find a hypothesis $h_{\mathbf{w},b} \in \mathcal{H}_{half}$ such that all data points in S that are labeled 1 are on the one side of the hyper-plane and all those labeled -1 are on the other side. To find such a hypothesis we must first make the assumption that the dataset is *linearly separable*. That is, there exists a hyper-plane such that samples of opposing labels are on opposite sides. Mathematically, we assume that

$$\exists \mathbf{w} \in \mathbb{R}^d, b \in \mathbb{R} \quad s.t \quad \forall i \in [m] \quad y_i \cdot \text{sign}(\langle \mathbf{x}, \mathbf{w} \rangle + b) = 1$$

or equivalently since the inner product will be negative for all samples with $y_i < 0$ and positive for all samples with $y_i > 0$:

$$\exists \mathbf{w} \in \mathbb{R}^d, b \in \mathbb{R} \quad s.t \quad \forall i \in [m] \quad y_i \cdot (\langle \mathbf{x}, \mathbf{w} \rangle + b) > 0 \tag{3.5}$$

Note, that assuming that a given training set is linearly separable is a *realizability assumption*. Namely, the labels are generated by a function in our hypothesis class $\mathcal{H}_{half}$. For simplicity, let us consider the homogeneous case where $b = 0$, since in the non-homoneneous case we can always shift the data by some fixed vector such that the separating hyperplane goes through the origin. So the hypothesis class of linear separators is of the form:

$$\mathcal{H}_{half} := \left\{ h_{\mathbf{w}}(\mathbf{x}) = \text{sign}\left(\mathbf{x}^\top \mathbf{w}\right) \;\middle|\; \mathbf{w} \in \mathbb{R}^d \right\} \tag{3.6}$$

3.2.1 Learning Linearly Separable Data Via ERM

To train a model over the defined hypothesis class of homogenous half-spaces ($\mathbf{w} \in \mathbb{R}^d, b = 0$) observe the following: for any hypothesis $h_{\mathbf{w}} \in \mathcal{H}_{half}$, the misclassified training samples are exactly those where $y_i \cdot sign\left(\mathbf{x}^\top \mathbf{w}\right) = -1$ or equivalently $y_i \cdot \mathbf{x}^\top \mathbf{w} < 0$. So defining the loss of a given hypothesis over S is:

$$L_S(h_{\mathbf{w}}) := \sum_{i=1}^{m} \mathbb{1}_{y_i \cdot \mathbf{x}^\top \mathbf{w} < 0} \tag{3.7}$$

Since we are assuming realizability (i.e. that S is linearly separable), we would like to find $h_{\mathbf{w}} \in \mathcal{H}_{half}$ that perfectly separates the training set. Such a hypothesis will be one that achieves $L_S(h_{\mathbf{w}}) = 0$. In other words, we are applying the ERM principle and seeking for any separating hyperplane $\mathbf{w}^\perp$, corresponding to a hypothesis $h_{\mathbf{w}}$ that minimizes the empirical risk $L_S(h_{\mathbf{w}})$.

So the next task is finding a compulationally efficient algorithm to find the desired hypothesis. As we are applying the ERM principle we would like to efficiently compute

$$\hat{\mathbf{w}} := \underset{\mathbf{w} \in \mathbb{R}^d}{\text{argmin}}\; L_S(h_{\mathbf{w}}) \tag{3.8}$$

where since assuming realizability, we know there exists a vector $\mathbf{w}^* \in \mathbb{R}^d$ such that $y_i \langle \mathbf{x}_i, \mathbf{w}^* \rangle > 0 \quad i = 1, \ldots, m$ (3.5). Notice, that if we define $\bar{\mathbf{w}} = \frac{\mathbf{w}^*}{\gamma}$ for $\gamma = \min_i (y_i \langle \mathbf{x}, \mathbf{w}^* \rangle)$ we have that

$$y_i \langle \mathbf{x}_i, \bar{\mathbf{w}} \rangle = \frac{1}{\gamma} y_i \langle \mathbf{x}_i, \mathbf{w}^* \rangle \geq 1 \quad i = 1, \ldots, m \tag{3.9}$$

Thus, it is enough to search for a vector $\hat{\mathbf{w}}$ that satisfies

$$y_i \cdot \text{sign}\left(\mathbf{x}^\top \hat{\mathbf{w}}\right) \geq 1 \quad \forall i = 1, \ldots, m \tag{3.10}$$

Convex Optimization

In (3.10) we therefore understand that the problem of finding a linear separator is in fact a problem of finding a vector that satisfies m linear constraints. As these constraints are all convex constraints this is an exaple of what is known as a *convex optimization problem*. Specifically it is a case of a linear program.

Definition 3.2.1 An *optimization problem* over $\mathbb{R}^d$ has the general form:

$$\begin{array}{ll} \text{minimize} & f_0(\mathbf{x}) \\ \text{subject to} & f_i(\mathbf{x}) \leq b_i \quad i = 1, \ldots, n \end{array}$$

where $\mathbf{x}$ is the optimization variable, $f_0 : \mathbb{R}^d \to \mathbb{R}$ is the objective function and $f_i : \mathbb{R}^d \to \mathbb{R}$ are the constraint functions. It is implicitly implied that the optimization problem happens over $\text{dom}(f_0) \subset \mathbb{R}^d$, the domain of f_0.

Then, naturally, a convex optimization problem is an optimization problem as above in which $f_0, f_1, \ldots, f_n$ are all convex functions. When these functions are all linear, this is a linear programming problem

In general, optimization problems are hard to solve computationally. We take special interest in *convex optimization* problems since they have a unique solution, and that solution can be found in computationally tractable ways. A great deal is known about *convex optimization algorithms*, which are iterative numerical algorithms that converge to the solution of a convex optimization problem. There are general solvers, which will solve a convex problem in the general form above, and there are specialized solvers for specific types, or families, of convex optimization problems. A specialized solver is typically preferred, as it leverages some particular structure of the problem to solve it more efficiently, using less space, etc.

Why is convex optimization interesting for machine learning? In supervised learning, we would like to choose a hypothesis $h \in \mathcal{H}$ from our selected hypothesis class, based on some learning principle (such as ERM). Many learning principles are formulated as optimization problems, namely, the h chosen by the learning algorithm is given as the minimizer of some quantity. So implementation of the learning algorithm needs to solve an optimization problem.

Sometimes, our hypothesis class is equivalent to a Euclidean space. When this happens, our learning principle reduces to solving an optimization problem, namely, the hypothesis we choose $h \in \mathcal{H}$ is found as a minimum over $\mathbb{R}^d$ or a subset of $\mathbb{R}^d$ of some objective function, usually a loss function. When this objective is convex, we can use convex optimization algorithms to implement our learning algorithm efficiently.

3.2.2 Solving ERM for Half-Spaces

Returning to the problem of the half-spaces classifier, we have seen that a hyperplane $\mathbf{w}^\perp$ minimizing the empirical risk is in fact a solution to the following linear program:

$$\begin{array}{ll} \text{minimize} & 0 \\ \text{subject to} & y_i \cdot \langle \mathbf{x}, \mathbf{w} \rangle \geq 1 \quad i = 1, \ldots, m \end{array} \tag{3.11}$$

Such an optimization problem, where the objective is trivial, it is a *feasibility* problem. That is, we are looking for any vector which satisfies the constraints. To solve this we can apply some generic solver for linear programs.

3.2.2.1 The Perceptron Algorithm

Another way for finding a separating hyperplane using the ERM principle is by using the Perceptron algorithm, suggested by Frank Rosenblatt in 1958. This is an iterative algorithm that constructs a series of vectors $\mathbf{w}^{(0)}, \mathbf{w}^{(1)}, \ldots$, where each vector is derived from the vector preceeding it. At each iteration t we search for a sample i which is misclassified by $w^{(t)}$. Then, we update $\mathbf{w}^{(t)}$ by moving it in the direction of the misclassified sample $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + y_i \mathbf{x}_i$.

Algorithm 1 Batch-Perceptron

procedure PERCEPTRON($S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$)
  $\mathbf{w}^{(1)} \leftarrow 0$ ▷ Initialize parameters
  for $t = 1, 2, \ldots$ **do**
    if $\exists i \; s.t. \; y_i \left\langle \mathbf{w}^{(t)}, \mathbf{x}_i \right\rangle \leq 0$ **then** ▷ If there exists a misclassified sample
      $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + y_i \mathbf{x}_i$
    else
      return $\mathbf{w}^{(t)}$
    end if
  end for
end procedure

Our goal when using the Perceptron algorithm is to find a vector $\mathbf{w}$ such that $y_i \cdot \mathbf{x}_i^\top \mathbf{w} > 0 \quad i = 1, \ldots, m$. Notice that as

$$y_i \langle \mathbf{w}^{(t+1)}, \mathbf{x}_i \rangle = y_i \langle \mathbf{w}^{(t)} + y_i \mathbf{x}_i, \mathbf{x}_i \rangle = y_i \langle \mathbf{w}^{(t)}, \mathbf{x}_i \rangle + ||\mathbf{x}_i||^2$$

the update rule of the Perceptron iteratively adjusts the hyperplane to be "more correct" on the i'th sample. It can be shown that in the realizable case the algorithm is guaranteed to terminate, returning a solution that correctly classifies all the samples.

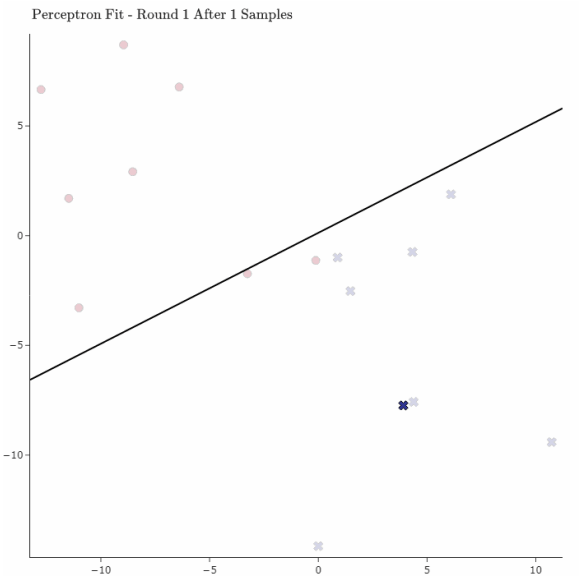


Figure 3.6: ***Perceptron Fitting:*** *Fit a separating hyperplane using Perceptron algorithm. Classification Examples*

R The Perceptron algorithm is in fact a simple case of the more general algorithm of Subgradient Descent covered in chapter 10.

3.2.3 Learner ID Card

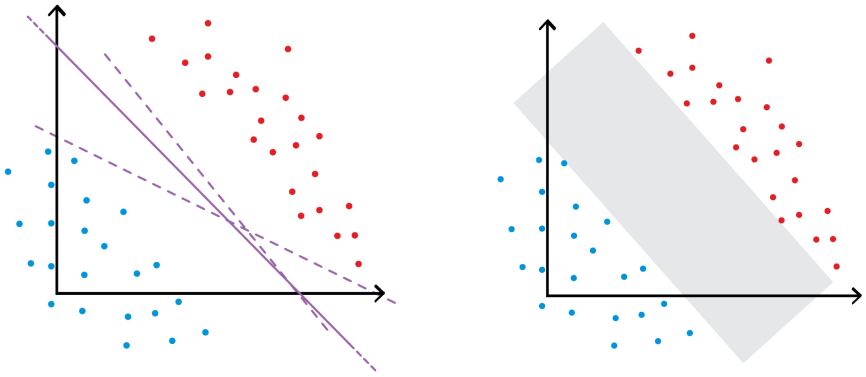
- **Hypothesis class:** the class of linear separators (3.6)
- **Learning principle used for training:** ERM for misclassification loss
- **Computational implementation:** Linear programming or Perceptron
- **Interpretability:** We do not have any specific insight into why a solution was chosen besides it simply satisfying the conditions.
- **Family of models:** No.
- **Storing fitted model:** Fitted model is the vector $\mathbf{w}$ perpendicular to the hyperplane defining the half-space. To store the model we simply store the d coefficients of the vector. In the case of non-homogeneous halfspace we also store the intercept coordinate
- **Prediction of new sample:** $\hat{y}_{new} := \text{sign}\left(\mathbf{x}_{new}^\top \mathbf{w} + b\right)$
- **When to use:** Since realizability assumption rarely holds this classifier is only used as a simple baseline

3.3 Support Vector Machines (SVM)

When using the half-space classifier seen above, we encounter two problems:

- Solution Uniqueness: When we are searching for a separating half-space the solution is not unique. That is, there could be more than a single vector satisfying the constraints of (3.11) and achieving the minimal empirical loss (of zero when assuming realizability or any other positive number if not). As such we are faced with the problem of which one to choose. Figure 3.7 illustrates the existance of multiple separating hyper-planes.

- Realizability: A more severe problem rises when we chose to work with the ERM learning principle for selecting the hypothesis, but the data is not linearly separable (non-realizible case). In this case the optimization problem described in 3.11 is computationally hard.



(a) *Algorithm selected hyper-plane (solid line) and additional optional separating hyperplanes (dashed lines)*

(b) *All hyper-planes linearly separating the data*

Figure 3.7: *Illustration of the existance of multiple separating hyper-planes*

Returning to the hypothesis class of non-homogeneous separating half-spaces $\mathcal{H}_{half}$ (3.4), we would like to describe a different learning principle that will be able to cope with both problems above: finds a unique hyperplane and that can be implemented computationally efficiently even when data is not linearly separable (i.e with polynomial running time given the input).

3.3.1 Maximum Margin Learning Principle

This learning principle is the one of maximum margin.

Definition 3.3.1 Let $(\mathbf{w},b) \in \mathbb{R}^d \times \mathbb{R}$ be a hyperplane and $u \in \mathbb{R}^d$. Define the distance between $(\mathbf{w},b)$ and u by:

$$d\left((\mathbf{w},b),u\right) := \min_{v:\langle v,w\rangle+b=0} ||u-v||$$

(namely, the Euclidean distance between u and the closest point on the hyperplane)

Definition 3.3.2 Let $(\mathbf{w},b) \in \mathbb{R}^d \times \mathbb{R}$ be a hyperplane and $S = u_1,\ldots,u_m \in \mathbb{R}^d$ a set of points. The *margin* of $(\mathbf{w},b)$ and S is the smallest distance between the hyperplane and any point:

$$M\left((\mathbf{w},b),S\right) := \min_{i\in[m]} d\left((\mathbf{w},b),u_i\right)$$

The margin of a given hyperplane with respect to S is therefore the minimal distance between the hyperplane and a sample in S. It seems logical that a hyperplane with a larger margin is more likely to still satisfy all

separability constraints even if S is slightly different.

So, the new learning principle is: choose $h_{\mathbf{w},b} \in \mathcal{H}_{half}$ that has the *largest margin* with respect to our training data S. Figure 3.8 illustrates the margins of two different potential hyperplanes. Based on these, we would prefer selecting the hyper-plane that is in the center of the grey area. The vectors closest to the hyperplane determine the margin. They are called *support vectors* and hence this learner's name.

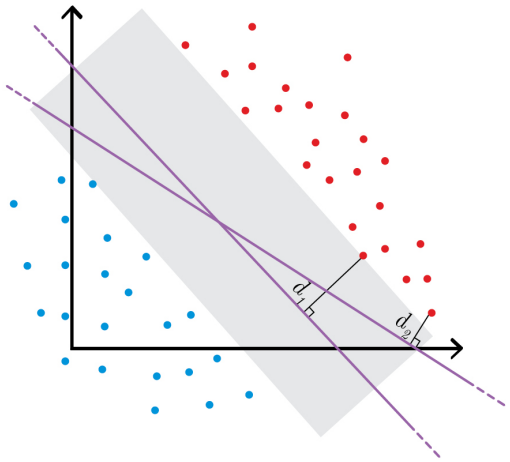


Figure 3.8: *Margin of specified hyper-planes*

3.3.2 Hard-SVM

Let us start with the *realizable case*. To implement our learning principle of maximal margin, we need to search, among all the separating hyperplanes of S, for the hyperplane with maximum margin. Namely, the hypothesis $h_{\mathbf{w},b} \in \mathcal{H}_{half}$ our learner will choose is the solution to the following optimization problem:

$$\begin{array}{ll} \text{maximize} & M\left((\mathbf{w},b),S\right) \\ \text{subject to} & y_i \cdot (\langle \mathbf{x}_i, \mathbf{w} \rangle + b) > 0 \quad i = 1,\ldots,m \end{array} \tag{3.12}$$

The optimization variables are $\mathbf{w} \in \mathbb{R}^d, b \in \mathbb{R}$. Comparing with the linear program of half-spaces (3.11) we see that the constraints are kept, which ensure the hyperplane chosen separates the training sample, but instead of a trivial objective, we seek to maximize the margin.

3.3.2.1 Solving Hard-SVM

So is the Hard-SVM a convex optimization problem? Recall, that by our optimization problem (3.12), we are searching of a separating hyperplane that maximizes the margin from all points. As for any $c > 0$ it holds that $(\mathbf{w},b) = (c\mathbf{w},cb)$ we can w.l.o.g constraint ourselves to $||\mathbf{w}|| = 1$. This way, each hyperlane has a unique vector $\mathbf{w}$ that corresponds to it.

In order to calculate the margin $M\left((\mathbf{w},b),S\right)$ we first must define what does it mean to measure a distance between a point in space and a set of points. For the purpose of the SVM classifier we define the distance as follows. Let $\mathbf{x} \in \mathbb{R}^d$ and $B \subseteq \mathbb{R}^d$. The distance from $\mathbf{x}$ to B: is $\inf_{\mathbf{v} \in B} ||\mathbf{x} - \mathbf{v}||$.

Lemma 3.3.1 Let $(\mathbf{w},b) \in \mathbb{R}^d \times \mathbb{R}$ be a hyperplane where $||w|| = 1$ and $\mathbf{x} \in \mathbb{R}^d$ then the distance between $\mathbf{x}$ and the hyperplane $(\mathbf{w},b)$ is $|\langle \mathbf{x}, \mathbf{w} \rangle + b|$.

* We saw that adding a 1 coordinate to the feature vector allows us to express the bias term. Therefore, for brevity, we would neglect it and only consider $|\langle \mathbf{x}, \mathbf{w} \rangle|$ instead of $|\langle \mathbf{x}, \mathbf{w} \rangle + b|$.

Proof. Since we define the distance between $\mathbf{x}$ and the hyperplane as the minimal distance between $\mathbf{x}$ and a vector in the hyperplane we are in fact looking at the orthogonal projection of $\mathbf{x}$ onto the hyperplane.

Similar to the way we have dealt with the intercept term in linear regression, let us assume that $\mathbf{x}, \mathbf{w} \in \mathbb{R}^{d+1}$, $x_0 = 1$ and w_0 represents the intercept. Let $P = \mathbf{w}\mathbf{w}^\top$ be the projection matrix onto span$(\mathbf{w})$ and consider the matrix $(I - P)$. Firstly, this is a projection matrix as

$$(I-P)^2 = I^2 - 2P + P^2 = I - 2P + P = I - P$$

Secondly, it holds that $Im\,(I-P)$ is the hyperplane defined by $(\mathbf{w}, b)$. Let $\mathbf{u} \in \mathbb{R}^d$ then :

$$\begin{aligned} \langle (I-P)\,\mathbf{u}, \mathbf{w} \rangle &= \langle \mathbf{u} - P\mathbf{u}, \mathbf{w} \rangle = \langle \mathbf{u}, \mathbf{w} \rangle - \langle P\mathbf{u}, \mathbf{w} \rangle = \langle \mathbf{u}, \mathbf{w} \rangle - \left\langle \mathbf{u}, P^\top \mathbf{w} \right\rangle \\ &= \langle \mathbf{u}, \mathbf{w} \rangle - \left\langle \mathbf{u}, \mathbf{w}\mathbf{w}^\top \mathbf{w} \right\rangle = \langle \mathbf{u}, \mathbf{w} \rangle - \langle \mathbf{u}, \mathbf{w} \rangle = 0 \end{aligned}$$

Thus $(I-P)$ is an orthogonal projection matrix onto the hyperplane $(\mathbf{w}, b)$ and $(I-P)\mathbf{x}$ is the orthogonal projection of $\mathbf{x}$ onto $(\mathbf{w}, b)$. Therefore, the distance between $\mathbf{x}$ and the hyperplane is given by:

$$||\mathbf{x} - (I-P)\mathbf{x}|| = ||\mathbf{x} - \mathbf{x} + P\mathbf{x}|| = \left|\left|\mathbf{w}\mathbf{w}^\top \mathbf{x}\right|\right| = ||\langle \mathbf{x}, \mathbf{w} \rangle \cdot \mathbf{w}|| = |\langle \mathbf{x}, \mathbf{w} \rangle| \cdot ||\mathbf{w}|| = |\langle \mathbf{x}, \mathbf{w} \rangle|$$

■

So, as the margin between a given hyperplane $(\mathbf{w}, b)$ and a set of points S is the minimal distance between the hyperplane and any point in the set, we derive that our optimization problem is in fact of the form:

$$\begin{array}{ll} \underset{(\mathbf{w},b):||\mathbf{w}||=1}{\text{argmax}} & \min_{i \in [m]} |\langle \mathbf{w}, \mathbf{x}_i \rangle + b| \\ \text{subject to} & y_i \cdot (\langle \mathbf{x}_i, \mathbf{w} \rangle + b) > 0 \quad i = 1, \ldots, m \end{array} \tag{3.13}$$

While the constraints enforce $\mathbf{w}$ to define a separating hyperplane, the objective will make us choose a separating hyperplane with the maximal margin. We further simplify the optimization problem. Consider a *feasible* solution $\mathbf{w}$ to the problem (i.e. that satisfies all constraints). It holds that $|\langle \mathbf{x}_i, \mathbf{w} \rangle + b| = y_i\,(\langle \mathbf{x}_i, \mathbf{w} \rangle + b)$. Hence, we can rewrite (3.13) as:

$$\begin{array}{ll} \underset{(\mathbf{w},b):||\mathbf{w}||=1}{\text{argmax}} & \min_{i \in [m]} y_i\,(\langle \mathbf{x}_i, \mathbf{w} \rangle + b) \\ \text{subject to} & y_i \cdot (\langle \mathbf{x}_i, \mathbf{w} \rangle + b) > 0 \quad i = 1, \ldots, m \end{array} \tag{3.14}$$

Notice that the constraints are now redundant. If $\mathbf{w}$ is infeasible then $\min_i y_i \left(\mathbf{x}_i^\top \mathbf{w} + b\right) < 0$, achieving a lower objective than any feasible solution. Therefore, we can re-write the problem as:

$$\underset{(\mathbf{w},b):||\mathbf{w}||=1}{\text{argmax}} \quad \min_{i \in [m]} y_i\,(\langle \mathbf{x}_i, \mathbf{w} \rangle + b) \tag{3.15}$$

And lastly, we notice that we can represent (3.15) as a norm minimization problem instead of margin maximization problem:

Claim 3.3.2 Consider the following optimization problem:

$$\underset{(\mathbf{w},b)}{\text{argmin}} \; ||\mathbf{w}||^2 \qquad \text{subject to} \quad y_i\,(\langle \mathbf{x}_i, \mathbf{w} \rangle + b) \geq 1 \quad i = 1, \ldots, m \tag{3.16}$$

If $(\mathbf{w}^*, b^*)$ is an optimal solution to (3.16) then $\left(\hat{\mathbf{w}}, \hat{b}\right)$ is an optimal solution to (3.15), where $\hat{\mathbf{w}} = \frac{\mathbf{w}^*}{||\mathbf{w}^*||}$, $\hat{b} = \frac{b^*}{||\mathbf{w}^*||}$.

Proof. Let $(\mathbf{w}, b)$ be a feasible solution to (3.15) and denote the margin achieved by $(\mathbf{w}, b)$ by γ then:

$$y_i\left(\langle \mathbf{x}_i, \mathbf{w}\rangle + b\right) \geq \gamma \quad i = 1, \ldots, m$$

Since $\gamma = \min_i y_i\left(\langle \mathbf{x}_i, \mathbf{w}\rangle + b\right)$ it holds that:

$$y_i\left(\left\langle \mathbf{x}_i, \frac{\mathbf{w}}{\gamma}\right\rangle + \frac{b}{\gamma}\right) \geq 1 \quad i = 1, \ldots, m$$

meaning that $(\frac{\mathbf{w}}{\gamma}, \frac{b}{\gamma})$ is a feasible solution to (3.16). Let $(\mathbf{w}^*, b^*)$ be an optimal solution for (3.16). As such, it achieves the minimal norm out of all feasible solutions and specifically it means that:

$$||\mathbf{w}^*|| \leq \left|\left|\frac{\mathbf{w}}{\gamma}\right|\right| = \frac{1}{\gamma}||\mathbf{w}|| = \frac{1}{\gamma}$$

where the last equality is due to $(\mathbf{w}, b)$ being a feasible solution to (3.15) and therefore $\mathbf{w}$ a unit vector. Consider $(\hat{\mathbf{w}}, \hat{b})$ achieved from $(\mathbf{w}^*, b^*)$. It follows that for all $i \in [m]$:

$$y_i\left(\langle \mathbf{x}_i, \hat{\mathbf{w}}\rangle + \hat{b}\right) = y_i\left(\left\langle \mathbf{x}_i, \frac{\mathbf{w}^*}{||\mathbf{w}^*||}\right\rangle + \frac{b^*}{||\mathbf{w}^*||}\right) = \frac{1}{||\mathbf{w}^*||} y_i\left(\langle \mathbf{x}_i, \mathbf{w}^*\rangle + b^*\right) \geq \frac{1}{||\mathbf{w}^*||} \geq \gamma$$

with $\hat{\mathbf{w}}$ being a unit vector. Thus, $(\hat{\mathbf{w}}, \hat{b})$ achieves a higher or equal objective to (3.15) from any feasible solution, concluding optimality. ■

This means in fact that maximizing the margin is equivalent to minimizing the size of the hyperplane.

Definition 3.3.3 An optimization problem is called a *Quadratic Program* (QP) if it can be written in the following form:

$$\begin{array}{ll} \min\limits_{\mathbf{w}\in\mathbb{R}^n} & \frac{1}{2}\mathbf{w}^\top Q\mathbf{w} + \mathbf{a}^\top \mathbf{w} \\ \text{such that} & A\mathbf{w} \leq \mathbf{d} \end{array}$$

where $Q \in \mathbb{R}^{n\times n}, A \in \mathbb{R}^{m\times n}, \mathbf{a} \in \mathbb{R}^n, \mathbf{d} \in \mathbb{R}^m$ are fixed vectors and matrices.

The optimization problem written in (3.16) is a Quadratic Program (QP) for which there exist efficient solvers. By using them to solve problem (3.16) we can obtain an optimal solution for the Hard-SVM optimization problem.

R But so how is it that minimizing $||\mathbf{w}||^2$ is equivalent to maximizing the margin? Let us denote the width of the total margin (i.e. the sum of margin from both sides) by l, and let x_+ and x_- be the positive- and negative support vectors . To calculate the value of l we will project the vector $x_+ - x_-$ onto the normalized normal $\mathbf{w}$:

$$\begin{aligned} l &= \left\langle x_+ - x_-, \frac{\mathbf{w}}{||\mathbf{w}||}\right\rangle \\ &= \left(\langle x_+, \mathbf{w}\rangle - \langle x_-, \mathbf{w}\rangle\right)||\mathbf{w}|| \\ &= \left(1 - b - (-1 - b)\right)/||\mathbf{w}|| \\ &= 2/||\mathbf{w}|| \end{aligned}$$

where support vectors satisfy $y_i\left(\langle \mathbf{w}, \mathbf{x}_i\rangle + b\right)$ and that for positive samples $y_i = 1$ and negative samples $y_i = -1$. This shows how minimizing $||\mathbf{w}||$ maximizes l.

3.3.3 Soft-SVM

The basic assumption of Hard-SVM is that the training sample is *linearly separable*, that is, that the realizability assumption holds. If that is not the case then the optimization problem has no solutions as for any candidate $(\mathbf{w}, b)$ at least one of the constraints $y_i \cdot \left(\mathbf{x}_i^\top \mathbf{w} + b\right) \geq 1$ cannot be satisfied.

However, what if the training sample is *almost* linearly separable? That is, what if most of the samples are linearly separable with only a few violating the constraints by "not too much"? Recall that if $y_i \cdot \left(\mathbf{x}_i^\top \mathbf{w} + b\right) < 0$ then sample $\mathbf{x}_i$ is on the "wrong side" of the hyperplane. This means that:

$$\exists \xi_i > 0 \quad s.t. \quad y_i \cdot \left(\mathbf{x}_i^\top \mathbf{w} + b\right) \geq 1 - \xi_i$$

Therefore, sample $\mathbf{x}_i$ is on the "wrong" side of the *margin* by an amount proportional to ξ_i (Figure 3.9). To allow training samples to violate the constraints "a little", we modify the optimization problem to:

$$\begin{array}{ll} \text{minimize} & ||\mathbf{w}||^2 \\ \text{subject to} & \begin{cases} y_i \cdot \left(\mathbf{x}_i^\top \mathbf{w} + b\right) \geq 1 - \xi_i & i = 1, \ldots, m \\ \xi_i \geq 0 \quad \wedge \quad \frac{1}{m} \sum_{i=1}^m \xi_i \leq C \end{cases} \end{array} \tag{3.17}$$

where $C > 0$ is a constant we specify. The variables $\xi_1, \ldots, \xi_m$ are new auxiliary variables we introduce (sometimes known as slack variables). Notice that the larger we choose C to be, the more violations of margin we allow. On the one hand, we want to allow "noisy" samples to violate the margin, so the hyperplane will ignore them. On the other hand, if we allow too many violations, we lose touch with the training sample and its structure. This is exactly the bias-variance trade-off: the larger C, the more freedom the learner has to "chase after the training sample".

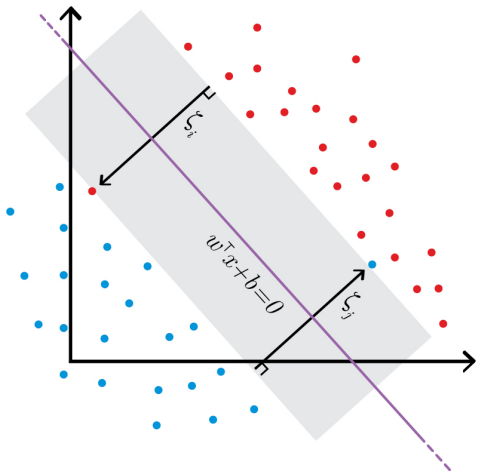


Figure 3.9: *Slack variables of data-points that are on the "wrong" side of the hyper-plane.*

Instead of specifying C directly, we often prefer working with a slightly different optimization problem, where instead of constraining the value of $\frac{1}{m}\sum \xi_i$ we jointly minimize the norm of $\mathbf{w}$ (related to the margin) and the average of ξ_i (corresponding margin violations).

$$\begin{array}{ll} \underset{\mathbf{w}, b, \{\xi_i\}}{\text{minimize}} & \lambda \, ||\mathbf{w}||^2 + \frac{1}{m} \sum_{i=1}^m \xi_i \\ \text{subject to} & y_i \cdot \left(\mathbf{x}_i^\top \mathbf{w} + b\right) \geq 1 - \xi_i, \; \xi_i \geq 0 \quad i = 1, \ldots, m \\ & \lambda \geq 0 \end{array} \tag{3.18}$$

To simplify the above optimization problem let us define the *hinge* loss function:

$$\ell^{hinge}(a) := \max\{0, 1-a\}, \; a \in \mathbb{R} \tag{3.19}$$

Claim 3.3.3 Given a training set $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ and hyperplane $(\mathbf{w}, b)$, the Soft-SVM optimization problem (**??**) is equivalent to

$$\underset{\mathbf{w},b}{\operatorname{argmin}} \left(\lambda ||\mathbf{w}||^2 + L_S^{hinge}((\mathbf{w}, b))\right)$$

where $L_S^{hinge}((\mathbf{w}, b)) := \frac{1}{m}\sum \ell^{hinge}\left(y_i \cdot \mathbf{x}_i^\top \mathbf{w}\right)$

Proof. Given a specific hyperplane $(\mathbf{w}, b)$ consider the minimization over $\xi_1, \ldots, \xi_m$. Since we defined the auxiliary variables to be nonnegative, the optimal assignment of ξ_i is

$$\xi_i := \begin{cases} 0 & y_i\left(\mathbf{x}_i^\top \mathbf{w} + b\right) \\ 1 - y_i\left(\langle \mathbf{x}_i, \mathbf{w}\rangle + b\right) > 1 & otherwise \end{cases}$$

Thus $\xi_i = \ell^{hinge}\left(y_i\left(\mathbf{x}_i^\top \mathbf{w} + b\right)\right)$ ■

The hyper-parameter λ controls the trade-off between the norm of $\mathbf{w}$ and the violations of margin.

- The larger λ, the less sensitive the solution will be to the term $\frac{1}{m}\sum_{i=1}^m \xi_i$, and will allow more violations.
- The smaller λ, the more sensitive and will allow less violations.

Therefore when we consider the parameter λ in (3.18) (or equivalently C in (3.17)), we are in fact considering different learners within a *family of learners*, where each member of the family is specified by a specific value of λ (or C). These different family members can be placed somewhere along the hypothesis complexity axis. Thus, changing the value of λ (or C) moves us along the bias-variance tradeoff. λ is known as a *regularization parameter*. This topic is covered in chapter 6.

3.3.4 Learner ID Card

- **Hypothesis class:** the class of non-homogeneous linear separators (3.4)
- **Learning principle used for training:** Maximal margin
- **Computational implementation:** Quadratic Program
- **Interpretability:** Retrieved solution does not provide meaningful insights regarding predictions
- **Family of models:** The Soft-SVM learner provides us with a set of models, indexed by the regularization parameter $\lambda \in [0, \infty)$
- **Storing fitted model:** Fitted model is the vector $\mathbf{w}$ perpendicular to the hyperplane defining the half-space as well as the intercept coordinate
- **Prediction of new sample:** $\hat{y}_{new} := \text{sign}\left(\mathbf{x}_{new}^\top \mathbf{w} + b\right)$
- **When to use:** By itself this learner should be used as a simple baseline. However, after embedding the data in some high-dimensional space (kernelization) this becomes a powerful learner (chapter 9)

3.4 Logistic Regression

3.4.1 A Probabilistic Model For Noisy Labels

Let us revisit the model of linear regression. Recall that when assuming Gaussian errors (2.2.3) we modeled the relation between the domain and response spaces as $\mathbf{y} = \mathbf{X}\mathbf{w} + \varepsilon$ for $\varepsilon \sim \mathcal{N}\left(0, \sigma^2 I_m\right)$. Notice that as ε is a random variable, $\mathbf{y}$ too is a random variable distributing as a multi-variate Gaussian:

$$\mathbf{y} \sim \mathcal{N}\left(\mathbf{X}\mathbf{w}, \sigma^2 I_m\right) \tag{3.20}$$

Focusing on a single pair $(\mathbf{x}_i, y_i)$, we can think of the above as the *conditional probability* of y_i given $\mathbf{x}_i$:

$$p(y_i|\mathbf{x}_i, \mathbf{w}) = \mathcal{N}\left(y_i|\phi_{\mathbf{w}}(\mathbf{x}_i), \sigma^2\right) \quad \text{where} \quad \phi_{\mathbf{w}}(\mathbf{x}) = \mathbf{x}^\top \mathbf{w} \tag{3.21}$$

where the notation of $\mathcal{N}(y_i|\mathbf{x}_i, \mathbf{w})$ means the probability of observing the response y_i for the feature vector of $\mathbf{x}_i$ and coefficients vector $\mathbf{w}$. We also condition on $\mathbf{w}$ (though not a random variable) to explicitly state the dependence on the model parameters. In other words, we assumed that each sample $(\mathbf{x}, y)$ is such that the *expected value* of the label y is linear in $\mathbf{x}$. As we are dealing with a regression model and $y_i \in \mathbb{R}$, the support of the random variable $y_i|\mathbf{x}_i, \mathbf{w}$ is $\mathbb{R}$.

Let us adapt the model above to fit for classification problems. We would like to assume that y_i distributes *Bernoulli* with the probability $p(\mathbf{x}_i)$ of y_i being 1 depending on the specific feature vector $\mathbf{x}_i$.

$$p(y_i|\mathbf{x}_i) = Ber(y_i|p(\mathbf{x}_i)) \tag{3.22}$$

How shall $p(\mathbf{x}_i)$ relate with $\mathbf{x}_i$? Unlike the linear regression model, we cannot assume a linear function $\phi_{\mathbf{w}}(\mathbf{x}) = \mathbf{x}^\top \mathbf{w}$ as $\phi_{\mathbf{w}} \in \mathbb{R}$ while $p(\mathbf{x}_i)$ is restricted to $[0,1]$. Instead, we would like to choose some *link* function $\phi_{\mathbf{w}} : \mathbb{R} \to [0,1]$ that is monotone increasing and maps $(-\infty, \infty)$ bijectively to $(0,1)$. Define the relation to be:

$$p(y_i|\mathbf{x}_i, \mathbf{w}) = Ber(y_i|\phi_{\mathbf{w}}(\mathbf{x}_i)), \quad \phi_{\mathbf{w}} := \sigma\left(\mathbf{x}^\top \mathbf{w}\right) \tag{3.23}$$

where σ is the *sigmoid* function, also known as the *logistic* function:

$$\sigma(\mathbf{a}) := \frac{e^{\mathbf{a}}}{e^{\mathbf{a}} + 1} \tag{3.24}$$

This function is indeed monotone increasing and maps $(-\infty, \infty)$ bijectively to $(0,1)$:

- As $\mathbf{x}^\top \mathbf{w} \to -\infty$ then $\sigma\left(\mathbf{x}^\top \mathbf{w}\right) \to 0$. This means that it is "very unlikely" that the label is 1: $p(y_i = 1|\mathbf{x}_i, \mathbf{w}) \to 0$.
- As $\mathbf{x}^\top \mathbf{w} \to \infty$ then $\sigma\left(\mathbf{x}^\top \mathbf{w}\right) \to 1$. This means that it is "very likely" that the label is 1: $p(y_i = 1|\mathbf{x}_i, \mathbf{w}) \to 1$.

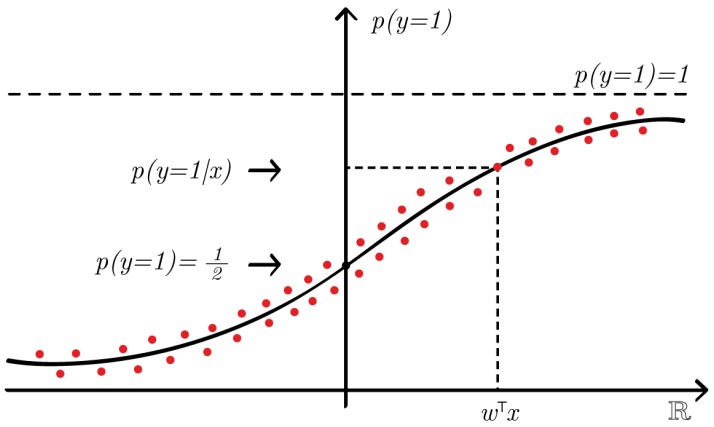


Figure 3.10: Illustration of fitted logit function *for values corresponding to* $\mathbf{x}^\top \mathbf{w}$.

R In (3.23) we modeled the logistic regression model for binary classification problems. Notice that the Bernoulli distribution can be seen as a private case of the Multinomial distribution $Multinomial(p_1, \ldots, p_K)$, $\sum_p i = 1, 0 \leq p_i \leq 1$. We can expand the above logistic regression model to fit multi-classification problems by extending the sigmoidal function to what is known as the softmax function $\sigma(\mathbf{a}) = e^{\mathbf{a}_i} / \sum_{j=1}^{K} e^{\mathbf{a}_j}$

3.4.1.1 The Hypothesis Class

So we would like to define the hypothesis class of logistic regression as:

$$\mathcal{H}_{logistic} := \left\{ h_{\mathbf{w}}(\mathbf{x}) = \sigma\left(\mathbf{x}^\top \mathbf{w}\right) | \mathbf{w} \in \mathbb{R}^{d+1} \right\} \tag{3.25}$$

where $\mathbf{w} \in \mathbb{R}^{d+1}$ since we incorporate the intercept variable into $\mathbf{w}$ (and a zeroth coordinate of 1 to $\mathbf{x}$) similar to the way we did in the linear regression hypothesis class. Notice that the hypotheses are defined $h_{\mathbf{w}} : \mathbb{R}^{d+1} \to [0,1]$ and not $h_{\mathbf{w}} : \mathbb{R}^{d+1} \to \{0,1\}$ as required for classification problems. Since $\{0,1\} \subset [0,1]$, we can use the training sample to select a function in $\mathcal{H}_{logistic}$. This means we will be able to train a model, but how will we predict over new samples? Suppose our learner chose some $h_{\mathbf{w}} \in \mathcal{H}_{logistic}$. As we think of $h_{\mathbf{w}}(\mathbf{x})$ as an estimate of the probability that the label corresponding to $\mathbf{x}$ is 1, we can use it for classification. If $h_{\mathbf{w}}(\mathbf{x})$ is low the label is likely to be 0. If $h_{\mathbf{w}}(\mathbf{x})$ is high, the label is likely to be 1. Choosing some *cutoff* value $\alpha \in [0,1]$, our class prediction will be: $\hat{y} := \mathbb{1}_{h_{\mathbf{w}}(\mathbf{x}) > \alpha}$. To choose a fitting value for α we can calculate the Type-I and Type-II errors (subsection 3.1.1) of the classifier and plot its ROC curve (3.4.4)

3.4.1.2 Learning Via Maximum Likelihood

Once we have defined the logistic regression model (3.23) and hypothesis class (3.25), we would like to come up with a learner. To do so we will use the *maximum likelihood principle*. Recall, that by the maximum likelihood principle, we estimate the parameters (the desired hypothesis) as those that have the highest probability, given the data.

Let $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ be our sample of independent observations, assuming that $\mathbf{y}_i \sim Ber(\phi_{\mathbf{w}}(\mathbf{x}))$ where ϕ is the logistic function. Therefore, the likelihood of $\mathbf{w} \in \mathbb{R}^{d+1}$ is:

$$\begin{aligned}
\mathcal{L}(\mathbf{w}|\mathbf{X},\mathbf{y}) &= \mathbb{P}(y_1,\ldots,y_m|\mathbf{X},\mathbf{w}) \\
&= \prod \mathbb{P}(y_i|\mathbf{x}_i,\mathbf{w}) \\
&= \prod_{i:y_i=1} \mathbb{P}(y_i|\mathbf{x}_i,\mathbf{w}) \cdot \prod_{i:y_i=0} \mathbb{P}(y_i|\mathbf{x}_i,\mathbf{w}) \\
&= \prod_{i:y_i=1} p_i(\mathbf{w}) \cdot \prod_{i:y_i=0} (1-p_i(\mathbf{w})) \\
&= \prod p_i(\mathbf{w})^{y_i} (1-p_i(\mathbf{w}))^{1-y_i}
\end{aligned} \tag{3.26}$$

where $p_i(\mathbf{w}) = \phi_{\mathbf{w}}(\mathbf{x}_i)$. Since the log function is monotone increasing we can maximize the log-likelihood $\ell(\mathbf{w}) := \log \mathcal{L}(\mathbf{w})$ instead:

$$\begin{aligned}
\ell(\mathbf{w}|\mathbf{X},\mathbf{y}) &= \sum_{i=1}^m \left[y_i \log\left(p_i(\mathbf{w})\right) + (1-y_i)\log\left(1-p_i(\mathbf{w})\right) \right] \\
&= \sum_{i=1}^m \left[y_i \log\left(\frac{e^{\mathbf{x}_i^\top \mathbf{w}}}{1+e^{\mathbf{x}_i^\top \mathbf{w}}}\right) + (1-y_i)\log\left(\frac{1}{1+e^{\mathbf{x}_i^\top \mathbf{w}}}\right) \right] \\
&= \sum_{i=1}^m \left[y_i \cdot \mathbf{x}_i^\top \mathbf{w} - \log\left(1+e^{\mathbf{x}_i^\top \mathbf{w}}\right) \right]
\end{aligned} \tag{3.27}$$

And therefore, choosing the function $h \in \mathcal{H}_{logistic}$ by applying the maximum likelihood principle means that:

$$\hat{\mathbf{w}} := \underset{\mathbf{w} \in \mathbb{R}^{d+1}}{\operatorname{argmax}} \sum_{i=1}^m \left[y_i \cdot \mathbf{w}^\top \mathbf{x}_i - \log\left(1+e^{\mathbf{w}^\top \mathbf{x}_i}\right) \right] \tag{3.28}$$

R Instead of deriving the learner using the maximum likelihood principle, we could derive it using the ERM principle with the following loss function: $\ell(h_{\mathbf{w}}) := \log\left(1+\exp\left(-y\langle \mathbf{w}, \mathbf{x}\rangle\right)\right)$. We would have reached the same optimization expression.

3.4.2 Computational Implementation

Now that we have defined the hypothesis class and an optimization problem to find the desired hypothesis, the next step is finding an efficient algorithm to solve it. By working with the logistic function, the resulting log-likelihood expression is **concave** function of the optimization variable $\mathbf{w}$. This means, that instead of solving the maximization problem (3.28) we can solve the minimization of minus the log-likelihood, which is convex. As such, there are general algorithms for finding the minima of such functions.

While there is no closed form for the maximizer $\hat{\mathbf{w}}$, as logistic regression is a very useful learner, there is a custom iterative algorithm that usually converges quickly to $\hat{\mathbf{w}}$. This algorithm is based on the second-order descent method of **Newton-Raphson** iterations, broadly discussed at subsection 10.2.4.

3.4.3 Interpretability

One important properly of the logistic regression learner is interpretability both in the sense of which features were important for the model and why was a certain prediction given. When working with $\mathcal{X} = \mathbb{R}^d$, we gather many feature, and might think that some of them are important for prediction while others less. Similar to linear regression, we are able to ask "which features were important" for the model by simply investigating the entries of the fitted coefficients vector $\mathbf{w}$. Features corresponding to coefficients of values close to zero have a small impact on prediction and therefore these features are less important for the model. Features corresponding to coefficients of values far from zero have a large impact on prediction and are therefore important for the model.

Then, for a given sample $\mathbf{x}$, by looking at entries corresponding to important features we can understand why the model predicted $\hat{y}$. If in entries of $\mathbf{x}$ corresponding important features there are large (positive or negative) values, they will have much influence the outcome. If these values are of same sign as of the coefficients then the expression $\mathbf{x}^\top \mathbf{w}$ will be larger, increasing the likelihood of the prediction being 1. If these values are of opposite signs to the coefficients then the expression $\mathbf{x}^\top \mathbf{w}$ will become smaller, decreasing the likelihood of the prediction being 1.

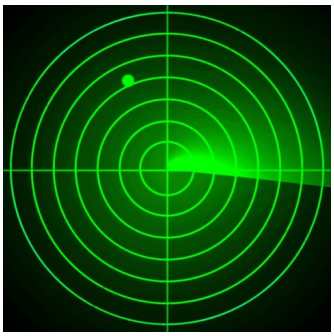
3.4.4 Predictions Over New Samples & The ROC Curve

As we will encounter later in this chapter, in many classification scenarios we face the following question: Suppose we trained some classifier $h \in \mathcal{H}$ and derive classifications by the following rule: for some cutoff value $\alpha \in [0,1]$

$$\hat{y} := \mathbb{1}_{h(\mathbf{x})>\alpha}$$

How do we choose α? There is an important *tradeoff* in the selection of α. If we set α to be very high we are mainly going to predict 0. By doing so we are *less* likely to have false-positives (which is the error we try to avoid at all cost), for which we are pleased. However, at the same time, we are *also* more likely to have false-negatives. So by setting α too high we might have low a FPR but "miss" (misclassify) most of the positive samples. On the other extreme, if we set α to be very low we are mainly going to predict 1. So we will be *more* likely to have false-positives, but at the same time will be *less* likely to have false-negatives. So if we set α too low, we might have a high FPR but "catch" (correctly classify) most of the positive samples. Therefore, we see that changing $\alpha \in [0,1]$ governs some trade-off between the chances of making a Type-I error (false-positives) and correctly classifying positive samples.

This trade-off was first studied during World War II, when radar was invented. The designer of the radar had to choose when to put a green dot on the radar, indicating a target detected there. Sometime radar waves would bounce off back from clouds or birds, and the designer had to choose a *threshold* α. If the radar pulse returning is stronger than α, the radar screen would show a green dot. If weaker than α, no dot. Now, if α is set too low (say $\alpha = 0.1$), the screen would be full of a thousand green dots - since any bird or cloud (with, say, $h(\mathbf{x} = 0.2)$) would be classified as *positive*, a target. So that the radar will be full of false positives, false targets, and will be useless. On the other hand, if α is set too high (say $\alpha = 0.9$) then enemy airplanes (with, say, $h(\mathbf{x} = 0.8)$) will not appear on the screen, since they will be classified by mistake as birds, and the radar again would be useless.



The radar engineers developed a way to visualize this tradeoff, which is still used today in machine learning. After training some linear model (choosing some hypothesis $h \in \mathcal{H}$) we make a grid of values of $\alpha \in [0,1]$. For each value of α we create a classifier by thresholding h at α, and calculate the number of Type-I and Type-II errors the classifier makes over a test sample that was not used for training. We plot a parametric curve of TPR (true positive rate) against FPR (false positive rate) when α is the parameter. This curve is called the *Receiver Operating Characteristic (ROC)* curve. It is continuous, increasing and goes from $(0,0)$ in the FPR-TPR plane (for $\alpha = 0$ we classify everything as negative, so no false positives and not true positives) to $(1,1)$ (for $\alpha = 1$ we classify everything as positive, so false positive rate is 1 - we make every possible Type-I error - and also true positive rate is 1 - we "catch" all the positive samples).

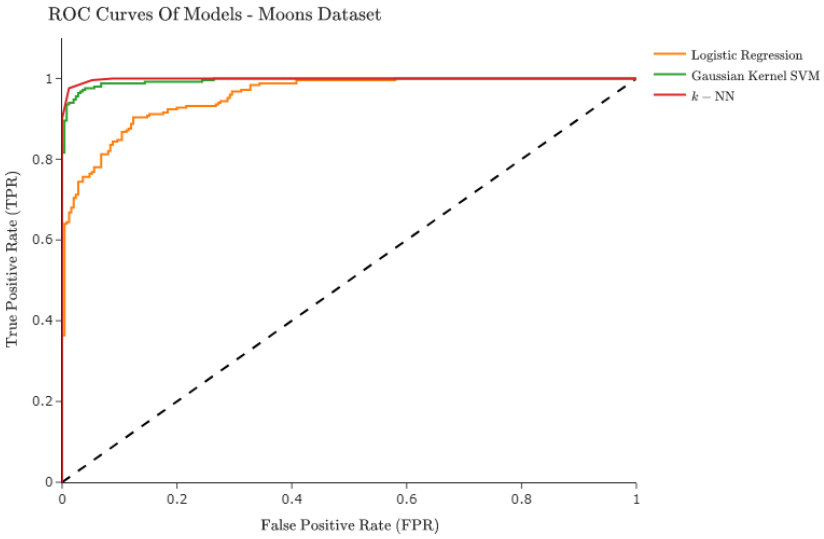


Figure 3.11: ROC Curve *of classifiers fitted over moons dataset.* *Classification Examples*

If the ROC curve is a linear line from $(0,0)$ to $(1,1)$, the classifier is just a random guess. If a classifier has an ROC curve that is close to this linear line, it's a poor classifier. Now convince yourself that if the ROC curve rises sharply from $(0,0)$, for example makes a "jump" to $(FPR = 0.1, TPR = 0.9)$, it's a good classifier - we are able to correctly detect 0.9 of the positive samples at the price of 0.1 false positive rate.

Plotting the ROC curve of a classifier has a few different uses:

- **Tuning α:** It allows us to see the tradeoff, provided by the classifier, between Type-I errors and correct detection of positive samples, so we can choose the tuning of α we would like to work with for the actual prediction.
- **AUC - Area Under Curve:** A performance measure for the tradeoff itself. This performance measure

evaluates the prediction rule h we chose without having to decide on α - it measures the quality of the **tradeoff** provided by h, a tradeoff from which we must choose a specific point in order to actually classify new samples. AUC is simply the *definite integral* of the ROC curve on the segment $[0,1]$ - the area under the AUC curve. As mentioned above, AUC around $1/2$ means that h is poor - more specifically, that the *tradeoff* provided by h is poor. AUC is bounded from above by 1, so an AUC close to 1 means h offers an excellent trade-off, and in this case we expect to be able to find a cutoff α that gives a classifier with very few false-positives and very high detection rate (true positive rate).

- **Comparing candidate rules:** Suppose we have a couple of candidate rules h_1, h_2 (or more). For example, maybe we trained some classifier on the same training sample with different features, or maybe we trained two different types of classifiers over the same data, and we are wondering which one to use. Now we have a problem - we can't turn h_i into an actual classification rule without choosing a cutoff α_i, but would like to compare h_1 to h_2 without committing to a cutoff - to compare the tradeoff offered by h_1 to that offered by h_2. It is very useful here to plot the two ROC curves of h_1 and h_2 on a single axis - and visually compare the tradeoffs they offer.

3.4.5 Learner ID Card

- **Hypothesis class:** The composite of the sigmoidal function over the linear functions (3.25)
- **Learning principle used for training:** Maximum likelihood
- **Computational implementation:** Specialized iterative method based on Netwon-Raphson iterations, or a general convex solver. When using a general convex solver we must pay attention to the effective rank of the regression matrix, similar to linear regression. Near-singular regression matrices will lead to numerical instabilities
- **Interpretability:** Given a fitted model we can interprate which features drive the classification as well as understand why a given sample was predicted as it was
- **Family of models:** As seen in chapter 6 it is possible to add regularization terms to control the bias-variance properties of the fitted model
- **Storing fitted model:** Store the regression coefficients vector $\mathbf{w}$
- **Prediction of new sample:** To perform predictions we must specify a thresholding parameter $\alpha \in (0,1)$. Once we chose a value of α then prediction is performed by $\hat{y}_{new} := \mathbb{1}_{\sigma(\mathbf{x}_{new}^\top \mathbf{w}+b)\geq\alpha}$
- **When to use:** The logistic regression learner, especially when adding regularization terms, is a powerful learner. It is always good to try it, especially when classes are more or less balanced.

For the logistic regression learner we have adapted the hypothesis class of linear regression by composing it with the sigmoid function:

$$\mathcal{H}_{logistic} := \left\{ \mathbf{x} \mapsto \sigma\left(\mathbf{x}^\top \mathbf{w}\right) \;\middle|\; \mathbf{w} \in \mathbb{R}^d \right\}$$

Then, we have derived an optimization problem using the maximum likelihood principle (3.28). To computationally implement the learner there are specialized iterative methods based on Newton-Raphson iterations, or general convex solvers. It is important to note that just like linear regression we must pay attention to the effective rank of the regression matrix. Near-singular matrices will cause numerical problems.

Given a trained model, we have to specify a threshold parameter α, which will provide some good tradeoff between the FPR and TPR. Once we have specified α prediction of a new sample is given by $\hat{y} := \mathbb{1}_{sigm(\mathbf{x}_{new}^\top \mathbf{w})\geq\alpha}$.

3.5 Bayes Classifiers

When deriving the logistic regression model, we assumed a probability distribution over the response set $\mathcal{Y}$ and treated the observations as deterministic values influencing the distribution of the response. We could however *assume* that both the response set and the domain set follow some *joint probability distribution* $\mathcal{F}$ over $\mathcal{X} \times \mathcal{Y}$. Under such assumption we are now able to look at the data from two different perspectives. Given the sample $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$, we could first consider $\mathbf{x}_1, \ldots, \mathbf{x}_m$ as fixed and learn $y_i|\mathbf{x}_i$. That is, the distribution of the response given the observation. This is the perspective used in the logistic regression, as

well as the linear regression, model. For the second perspective we consider $y_1, \ldots, y_m$ as fixed and ask how do the observations depend on the responses $\mathbf{x}_i | y_i$.

$$\overbrace{f_{Y|X=\mathbf{x}}(y) \cdot f_X(\mathbf{x})}^{\text{Perspective I}} = \overbrace{f_{X,Y}(\mathbf{x}, y)}^{\substack{\text{Joint Probability} \\ \text{Distribution}}} = \overbrace{f_{X|Y=y}(\mathbf{x}) \cdot f_Y(y)}^{\text{Perspective II}} \tag{3.29}$$

■ **Example 3.4** Consider the classification task of separating images of cats and dogs. Let the domain space be RBG images of 1024-by-1024 pixels and the response set be $\mathcal{Y} := \{\text{cat}, \text{dog}\}$. Further assume there exists some joint probability distribution $\mathcal{F}$ over $\mathcal{X} \times \mathcal{Y}$. Considering the first perspective where we wish to learn the conditional distribution $y_i | \mathbf{x}_i$. By doing so we try to discriminate a given picture being either of cat or of dog. That is, we simply try to understand how to differentiate between these two possibilities. If however we consider the second perspective, we wish to learn the conditional distribution of $\mathbf{x}_i | y_i$. This probability distribution describes what sort of observations might be seen for a given response. That is, what do pictures of cats or of dogs look like. ■

Besides providing insights into the manner in which different samples behave - how do cat pictures look? how do dog pictures look? - what benefit do we get from considering this second perspective? How can it be used for predicting the response of a given sample? To answer this question we use the Bayes' Law of conditional probability. Given a joint probability distribution function $f_{X,Y}$ over the observation $\mathbf{x}$ and response y we can express the conditional distribution $y | \mathbf{x}$ using the conditional distribution $\mathbf{x} | y$:

$$f_{Y|X=\mathbf{x}}(y) = \frac{f_{X|Y=y}(\mathbf{x}) \cdot f_Y(y)}{f_X(\mathbf{x})}$$

From this relation we are able to derive the Bayes Optimal classifier.

3.5.1 Maximum Aposteriori Estimation

Definition 3.5.1 Let $f_{\mathcal{D}}$ be a joint probability distribution function over $\mathcal{D} := \mathcal{X} \times \mathcal{Y}$. The *Bayes Optimal Classifier* is defined as

$$h^{Bayes}(\mathbf{x}) := \underset{y \in \mathcal{Y}}{\text{argmax}}\ f_{Y|X=\mathbf{x}}(y)$$

That is, we predict the response that (given the observation) achieves the highest probability. Since we are searching for a maximizer, we can use Bayes' Law to express the Bayes Optimal classifier as

$$h^{Bayes}(\mathbf{x}) := \underset{y \in \mathcal{Y}}{\text{argmax}}\ f_{Y|X=\mathbf{x}}(y) = \underset{y \in \mathcal{Y}}{\text{argmax}}\ \frac{f_{X|Y=y}(\mathbf{x}) f_Y(y)}{f_X(\mathbf{x})} \overset{(*)}{=} \underset{y \in \mathcal{Y}}{\text{argmax}}\ f_{X|Y=y}(\mathbf{x}) f_Y(y)$$

where $(*)$ is because given $\mathbf{x}$, $f_X(\mathbf{x})$ is constant over the different values of y. Notice that we have already seen this sort of algorithm (in the context of regression) when mentioning Bayesian statistics (subsubsection 1.1.3.1). The conditional $f_{X|y=y}$ is the *likelihood function* whose maximizer is the maximum likelihood estimator. It assesses the probability of observing $\mathbf{x}$ given that the response was y. The marginal distribution $f_Y(y)$ is the *prior* distribution and it assesses the *a-priori* probability (i.e belief) of receiving a sample with such a response. Therefore, the Bayes Optimal classifier weights the MLE according to the different class probabilities. The conditional distribution $f_{Y|X=x}$ is called the *a-posteriori* distribution - the probability of the response *after* (i.e give) observing $\mathbf{x}$. This estimator is called the *Maximum A-Posteriori Estimator* (MAP) and is often denoted as $\hat{y}^{MAP}$.

What is left to explain where does the "Optimal" in "Bayes Optimal" comes from. For that, let us consider the misclassification loss function. It therefore holds that the Bayes Optimal classifier achieves the minimal misclassification risk out of all possible classifiers.

Theorem 3.5.1 Let $f_{X,Y}$ be a joint probability distribution function over $\mathcal{X} \times \mathcal{Y}$ for $\mathcal{Y} = [K]$, $K \in \mathbb{N}$. The Bayes optimal classifier is the optimal classifier with respect to the misclassification error. Namely that for any hypothesis $h : \mathcal{X} \to \mathcal{Y}$ it holds that $L_{\mathcal{D}}(h^{Bayes}) \leq L_{\mathcal{D}}(h)$.

Proof. Let $h : \mathcal{X} \to \mathcal{Y}$ be some hypothesis. The generalization error of h with respect to the misclassification loss is given by

$$\begin{array}{rclrcl} L_{\mathcal{D}}(h) & = & \mathbb{E}_{\mathbf{x},y}\left[h(\mathbf{x}) \neq y\right] & = & \int_{\mathbf{x},y} f_{X,Y}(\mathbf{x},y) \cdot \mathbb{1}_{h(\mathbf{x})\neq y} d(\mathbf{x},y) \\ & = & \int_{\mathbf{x}} f_X(\mathbf{x}) \sum_y f_{Y|X=\mathbf{x}}(y) \cdot \mathbb{1}_{h(\mathbf{x})\neq y} d\mathbf{x} & = & \int_{\mathbf{x}} f_X(\mathbf{x}) \left(1 - f_{Y|X=\mathbf{x}}(y) \cdot \mathbb{1}_{h(\mathbf{x})=y}\right) d\mathbf{x} \\ & = & \int_{\mathbf{x}} f_X(\mathbf{x}) \left(1 - f_{Y|X=\mathbf{x}}(h(\mathbf{x}))\right) d\mathbf{x} & & \end{array}$$

Since the Bayes classifier is defined to return the class maximizing the posterior then $f_{Y|X=\mathbf{x}}(h(\mathbf{x})) \leq f_{Y|X=\mathbf{x}}(h^{Bayes}(\mathbf{x}))$ and therefore:

$$\begin{aligned} L_{\mathcal{D}}(h) &= \int_{\mathbf{x}} f_X(\mathbf{x}) \left(1 - f_{Y|X=\mathbf{x}}(h(\mathbf{x}))\right) d\mathbf{x} \\ &\geq \int_{\mathbf{x}} f_X(\mathbf{x}) \left(1 - f_{Y|X=\mathbf{x}}(h^{Bayes}(\mathbf{x}))\right) d\mathbf{x} \\ &= \mathop{\mathbb{E}}_{(\mathbf{x},y)\sim\mathcal{D}} \left[h^{Bayes}(\mathbf{x}) \neq y\right] \\ &= L_{\mathcal{D}}\left(h^{Bayes}\right) \end{aligned}$$

Thus, $L_{\mathcal{D}}(h^{Bayes}) \leq L_{\mathcal{D}}(h)$ and we conclude the optimality of the Bayes classifier. ■

It is important to note that in reality *we do not know the underlying distribution* $\mathcal{D}$, to whom all we have is a mere window in the form of the dataset. Therefore, we *cannot program* an algorithm that would find the maximizer of the posterior distribution. As such, we must think of the Bayes optimal classifier as an *Oracle* - a "black box" entity capable of solving the problem of finding the maximizer of the posterior.

If however implementing the Bayes optimal classifier is not feasible is it of any practical use? Though we do not know $\mathcal{D}$ we might still have some prior insights into the manner by which the data behaves. For example we might have some prior knowledge regarding the prevalence of different labels or we might assume that for different responses different feature values are more likely to be observed. We can incorporate these insights into the derived model. Then, if these assumptions hold, we might be able to provide a realization of the Bayes optimal classifier. Below are two examples for such realizations.

3.5.2 Linear Discriminant Analysis

The Linear Discriminant Analysis (LDA) algorithm is a realization of the Bayes Optimal classifier. In this model we assume that samples of different labels have different Gaussian distributions.

Definition 3.5.2 Let $\Omega = \{1, \ldots, K\}$ for $K \in \mathbb{N}$ be a sample space. A random variable $X : \Omega \to [0,1]$ follows a *Multinomial* distribution with parameter $\pi \in [0,1]^K, \sum \pi_i = 1$ if $\mathbb{P}(X = j) = \pi_j$, $j \in [K]$. We denote $X \sim \text{Mult}(\pi)$.

Explicitly, the LDA model assumes the following generative model:

1. Each sample "selects" a label y_i according to a multinomial distribution with K classes.
2. Then, the sample itself is drawn from the conditional probability of $X|Y$ where X denotes the random variable of sampling some samples $X = \mathbf{x}$ and Y denotes the random variable of $Y = y$, $y \in [K]$. The distribution used to model $X|Y$ is a Gaussian distribution where each label is characterized by a different mean vector $\{\mu_k \in \mathbb{R}^d\}_{k=1}^K$ but the same covariance matrix $\Sigma \in \mathbb{R}^{d\times d}$.

Namely, for any $i \in 1,\ldots,m$ we assume that:

$$\begin{aligned} y_i &\sim \text{Mult}(\pi) \\ \mathbf{x}_i|y_i &\sim \mathcal{N}(\mu_{y_i}, \Sigma) \end{aligned} \tag{3.30}$$

Under these assumptions, predicting the class of a new sample is done by simply using the Bayes Law over the Bayes Optimal classifier to get:

$$\hat{y}(\mathbf{x}) := \underset{y}{\text{argmax}}\ \mathbb{P}(y|\mathbf{x}) = \underset{y}{\text{argmax}}\ \frac{\mathbb{P}(\mathbf{x}|y)\,\mathbb{P}(y)}{\mathbb{P}(\mathbf{x})}$$

Claim 3.5.2 Let $f_{X,Y}$ be the pdf of $\mathcal{D}$ a joint distribution over $\mathcal{X} \times \mathcal{Y}$ and suppose

$$y \sim \text{Mult}(\pi), \quad y|\mathbf{x} \sim \mathcal{N}(\mu_k, \Sigma)$$

for $\pi \in [0,1]^K$, $\sum \pi_k = 1$. Then the Bayes Optimal classifier is given by:

$$\hat{y}(\mathbf{x}) := \underset{k}{\text{argmax}}\ a_k^\top \mathbf{x} + b_k, \quad a_k := \Sigma^{-1}\mu_k,\ b_k := \log(\pi_k) - \tfrac{1}{2}\mu_k^\top \Sigma^{-1}\mu_k$$

Proof. To prove the claim we begin with expressing $f_{Y|X}(k)$ using Bayes Law:

$$f_{Y|X=\mathbf{x}}(k) = \frac{f_{X|Y=k}(\mathbf{x}) \cdot f_Y(k)}{f_X(\mathbf{x})} = \frac{f_{X|Y=k}(\mathbf{x}) \cdot f_Y(k)}{\sum_{k'} f_{X|Y=k'}(\mathbf{x}) \cdot f_Y(k')} = \frac{\pi_k \cdot \mathcal{N}(\mathbf{x}|\mu_k, \Sigma)}{\sum_{k'} \pi_{k'} \cdot \mathcal{N}(\mathbf{x}|\mu_{k'}, \Sigma)}$$

Notice, that since we wish to maximize the posterior distribution $f_{Y|X}$ for a *given* sample $\mathbf{x}$, we can ignore the evidence $f_X(\mathbf{x})$. Then, by the pdf of the multivatiate Gaussian (**??**) then:

$$\begin{aligned} \text{argmax}_k f_{Y|X=\mathbf{x}}(k) &= \text{argmax}_k \frac{f_{X|Y=k}(\mathbf{x}) \cdot f_Y(k)}{f_X(\mathbf{x})} \\ &= \text{argmax}_k \pi_k \cdot \mathcal{N}(\mathbf{x}|\mu_k, \Sigma) \\ &= \text{argmax}_k \pi_k \cdot \tfrac{1}{Z} \exp\left(-\tfrac{1}{2}(\mathbf{x}-\mu_k)^\top \Sigma^{-1}(\mathbf{x}-\mu_k)\right) \\ &= \text{argmax}_k \log(\pi_k) - \tfrac{1}{2}\mathbf{x}^\top \Sigma^{-1}\mathbf{x} + \mathbf{x}^\top \Sigma^{-1}\mu_k - \tfrac{1}{2}\mu_k^\top \Sigma^{-1}\mu_k \\ &= \text{argmax}_k \log(\pi_k) + \mathbf{x}^\top \Sigma^{-1}\mu_k - \tfrac{1}{2}\mu_k^\top \Sigma^{-1}\mu_k \end{aligned}$$

for $Z := \sqrt{(2\pi)^d |\Sigma|}$ the Gaussians' normalization factor. Denote $a_k := \Sigma^{-1}\mu_k, b_k := \log(\pi_k) - \frac{1}{2}\mu_k \Sigma^{-1}\mu_k$ and we conclude that $\hat{y}(\mathbf{x}) = \text{argmax}_k\ a_k^\top \mathbf{x} + b_k$. ■

Notice, that we were able to remove $\mathbf{x}^\top \Sigma^{-1}\mathbf{x}$ since we assumed that the classes are generated from Gaussians with the *same* covariance matrix, and therefore the expression did not depend on any specific class k. The claim above, besides showing that under the LDA assumptions we are dealing with a Bayes classifier, also tells us something about the classifier learned. Looking at the expression derived from the assumptions, we see that the classification is in fact done by some *linear separator/discriminant*.

It can be shown, by taking the $\log -$*likelihood* ratio between the likelihood for being classified for a class divided the likelihood for being classified for the other class, that the decision boundary between the classes is linear, similar to Figure 3.12.

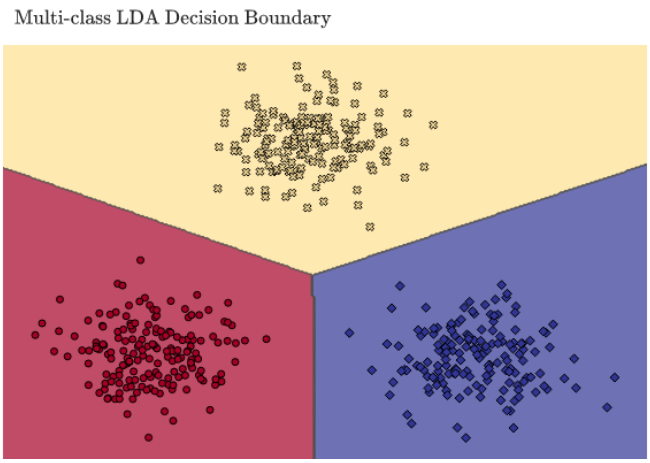


Figure 3.12: LDA Decision Boundaries *for a multiclass setup of three Gaussians. Classification Examples*

Learning A LDA Classifier

So, in order to predict using an LDA classifier we need to know the class probabilities π, the Gaussian centers $\{\mu_i\}$ and the covariance matrixmeans Σ. To do so we derive the maximum likelihood estimators. Let us generalize the binary LDA model (i.e. of classification) to a multiclassification LDA model.
Then, the LDA model assumptions are:

$$\begin{aligned} y &\sim \text{Mult}(\pi) \\ \mathbf{x}|y=k &\sim \mathcal{N}(\mu_k, \Sigma) \end{aligned} \tag{3.31}$$

Given a training set $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ then the likelihood is given by:

$$\begin{aligned} \mathcal{L}(\Theta|\mathbf{X},\mathbf{y}) &= f_{X,Y|\Theta}\left(\{(\mathbf{x}_i,y_i)\}_{i=1}^m\right) \\ &\overset{iid}{=} \prod_{i=1}^m f_{X,Y|\Theta}(\mathbf{x}_i,y_i) \\ &= \prod_{i=1}^m f_{X|Y=y_i}(\mathbf{x}_i)\cdot f_{Y|\Theta}(y_i) \\ &= \prod_{i=1}^m \mathcal{N}(\mathbf{x}_i|\mu_{y_i},\Sigma)\cdot \text{Mult}(y_i|\pi) \end{aligned}$$

Since the log transformation is monotonous increasing finding the maximizer of the likelihood is equivalent to finding the maximizer of the log-likelihood.

$$\begin{aligned} \ell(\Theta|\mathbf{X},\mathbf{y}) &= \log\left(\textstyle\prod_i \mathcal{N}(\mathbf{x}_i|\mu_{y_i},\Sigma)\cdot \text{Mult}(y_i|\pi)\right) \\ &= \textstyle\sum_i \log\left(\mathcal{N}(\mathbf{x}_i|\mu_{y_i},\Sigma)\right) + \log\left(\text{Mult}(y_i|\pi)\right) \\ &= \textstyle\sum_i \log\left(\frac{1}{\sqrt{(2\pi)^d|\Sigma|}}\exp\left(-\frac{1}{2}(\mathbf{x}_i-\mu_{y_i})^\top\Sigma^{-1}(\mathbf{x}_i-\mu_{y_i})\right)\right) + \log(\pi_{y_i}) \\ &= \textstyle\sum_i \log(\pi_{y_i}) - \frac{d}{2}\log(2\pi) - \frac{1}{2}\log|\Sigma| - \frac{1}{2}(\mathbf{x}_i-\mu_{y_i})^\top\Sigma^{-1}(\mathbf{x}_i-\mu_{y_i}) \\ &= \textstyle\sum_k \left[n_k\cdot\log(\pi_k) - \frac{1}{2}\sum_{i|y_i=k}(\mathbf{x}_i-\mu_k)^\top\Sigma^{-1}(\mathbf{x}_i-\mu_k)\right] - \frac{md}{2}\log(2\pi) - \frac{m}{2}\log|\Sigma| \end{aligned}$$

for $n_k = \sum_i \mathbb{1}_{y_i=k}$. To find the maximizers we derive with respect to the different parameters $\{\pi_k\},\{\mu_k\},\Sigma$ and equate to zero. However, before doing so recall the constraint on π: $\pi \in [0,1]^K$, $\sum_k \pi_k = 1$. To solve the optimization problem with the constraint we use the Lagrange Multipliers method. Since the constraint is $\sum_k \pi_k = 1 \iff \sum_k \pi_k - 1 = 0$, we define the function $g(\pi) = \sum_k \pi_k - 1$ and the Lagrangian

$$\mathcal{L} = \ell(\Theta|\mathbf{X},\mathbf{y}) - \lambda g(\pi)$$

Now, we derive with respect to each of the parameters including λ. Beginning with the class probabilities then:

$$\frac{\partial \mathcal{L}}{\partial \pi_k} = \frac{\partial}{\partial \pi_k} \ell(\Theta|\mathbf{X},\mathbf{y}) - \lambda \frac{\partial}{\partial \pi_k} g(\pi) = \frac{n_k}{\pi_k} - \lambda = 0 \quad \Rightarrow \quad \pi_k = \frac{n_k}{\lambda} \tag{3.32}$$

To find the value of λ we replace π in the constraint with the expression found in (3.32).

$$1 = \sum_k \pi_k = \sum_k \frac{n_k}{\lambda} \quad \Longleftrightarrow \quad \lambda = m$$

and therefore, the MLE of the class probabilities are $\hat{\pi}_k^{MLE} = \frac{n_k}{m}$. To find the MLE of the Gaussian parameters notice that the log-likelihood above is identical to the one derived of a single Gaussian while considering only samples sharing the same label. As such

$$\hat{\mu}_k^{MLE} = \frac{1}{n_k} \sum_i \mathbb{1}_{y_i=k} \mathbf{x}_i, \qquad \hat{\Sigma}^{MLE} = \frac{1}{m} \sum_i \left(\mathbf{x}_i - \hat{\mu}_{y_i}^{MLE}\right) \left(\mathbf{x}_i - \hat{\mu}_{y_i}^{MLE}\right)^\top \tag{3.33}$$

Looking closely at the expressions derived we in fact realize that the MLE predicts the values proportional to what is found in the training set.

R This estimator described in 3.33 is known as the *pooled covariance* where we take into account that different samples originate from different Gaussians and therefore should use the sample mean estimator of their own Gaussian. This is a *biased* pooled covariance estimator. The unbiased estimator is given by replacing the $\frac{1}{m}$ factor with $\frac{1}{m-K}$, where K is the number of classes.

3.5.3 Quadratic Discriminant Analysis

In the LDA algorithm we assumed the data is generated from a set of Gaussians, differing in their mean but sharing the same covariance matrix (3.30). The Quadratic Discriminant Analysis algorithm allows different covariance matrices. That is, for any $i \in 1,\ldots,m$ we assume that:

$$\begin{aligned} y_i &\sim \text{Mult}(\pi) \\ \mathbf{x}_i|y_i &\sim \mathcal{N}(\mu_{y_i}, \Sigma_{y_i}) \end{aligned} \tag{3.34}$$

By enabling different covariance matrices the quadratic expression (in $\mathbf{x}$) of $\mathbb{P}(y=k|\mathbf{x})$ does not cancel out. This in turn causes the decision boundaries between classes to be quadratic rather than linear. In both Figure 3.12 and Figure 3.13 the same data was used to fit either the LDA or the QDA models. We can see that while the decision boundaries of the LDA fit (Figure 3.12) are linear, in the case of QDA (Figure 3.13) we get curved (quadratic) boundaries.

Learning A QDA Classifier

Fitting a QDA classifier is very similar to the process of fitting a LDA classifier. The difference is in the estimation of the covariance matrices. In this case we fit a different covariance matrix for each class based on the samples of the class:

$$\hat{\Sigma}_k^{MLE} := \frac{1}{m_k} \sum_{i:y_i=k} \left(\mathbf{x}_i - \hat{\mu}_k^{MLE}\right) \left(\mathbf{x}_i - \hat{\mu}_k^{MLE}\right)^\top \tag{3.35}$$

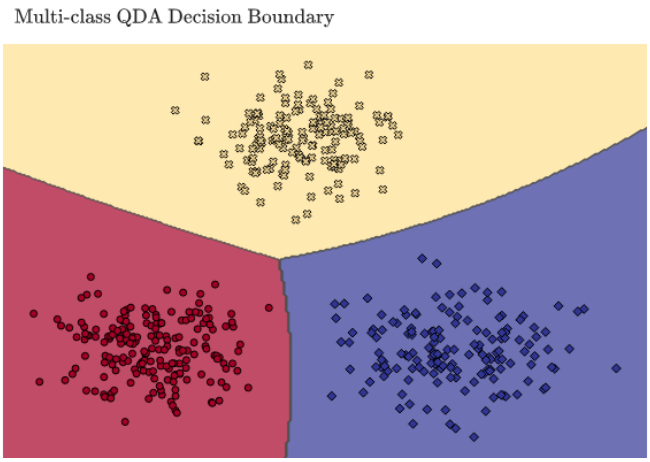


Figure 3.13: QDA Decision Boundaries *for a multiclass setup of three Gaussians. Classification Examples*

3.6 Nearest Neighbors

Nearest Neighbors classifiers are a popular, simple and effective learner in which we predict a sample's response based on a set of "nearest" training samples. This classifier is **not** based on the paradigm of a hypothesis class and learning principle. It is a *model free* learner and has no training stage. Instead, when given a training set, we store it in some manner. Then, when we are given a new sample to predict its response we "simply" find the subset of training samples nearest to the new sample, with respect to some measure of distance, and make a prediction based on the responses of those neighbor samples.

This family of learners are part of a wider *graph-based approach* for learning. In this approach we first define some graph structure over the samples - forming the nodes of the graph. Then, using the constructed graph we perform training and prediction. The different learners differ in how to define edges in the graph; are they weighted or not? how to transition between nodes; and how is this structure used for training and prediction. Another graphed-based approach that will be seen later is of Spectral Clustering subsection 8.2.2.

3.6.1 Prediction Using k-NN

Let us begin with the simplest form of k-NN. The first step is to determine two hyper-parameters required by the algorithm: an integer k (the number of neighbors to use) and a distance function $\rho : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}_+$. We can decide for example to use the square Euclidean norm $\rho(\mathbf{x}_1, \mathbf{x}_2) := ||\mathbf{x}_1 - \mathbf{x}_2||^2$ or a weighted square norm giving different importance levels to different features $\rho(\mathbf{x}_1, \mathbf{x}_2) := \sum \omega_i ((\mathbf{x}_1)_j - (\mathbf{x}_2)_j)^2$.

Then, given a training set $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ the prediction is done as follows:

Algorithm 2 k-NN

procedure k-NN(k, ρ, S,$\mathbf{x}'$)▷ Where k, ρ are the pre-determined hyper-parameters, S the training set and $\mathbf{x}'$ the sample to predict for

Compute distance from $\mathbf{x}'$ with respect to ρ: $\forall \mathbf{x} \in S \quad d_{\mathbf{x}} := \rho(\mathbf{x}', \mathbf{x})$.

Denote $\pi = (\pi_1, \ldots, \pi_m)$ the permutation of $(1, \ldots, m)$ such that $d_{\mathbf{x}_{\pi_1}} \leq \ldots \leq d_{\mathbf{x}_{\pi_m}}$

Select k nearest samples $\mathbf{x}_{\pi_1}, \ldots, \mathbf{x}_{\pi_k}$ and predict by majority vote:

$$\hat{y} := \underset{y \in \{0,1\}}{\operatorname{argmax}} \sum_{i=1}^{k} \mathbb{1}_{y_{\pi_i} = y}$$

return $\hat{y}$

end procedure

3.6.2 Selecting Value of k Hyper-Parameter

A very important aspect in k-NN is the chosen value of k. Though methods for determining the "right" k will be discussed in future chapters, let us dwell on a few cases:

- $k = 1$**:** The test point is given the label of the single nearest neighbor in the training set. Such classifier has a very low bias but very high variances.
- $k = m$**:** The classifier predicts a single label for any given test sample, regardless to its values. It will predict the majority vote of the training set labels. In this case the bias is very high and the variance is zero.

As we change k we change the bias-variance tradeoff, with larger values of k creating simpler models while smaller values of k creating more complex models (Figure 3.14).

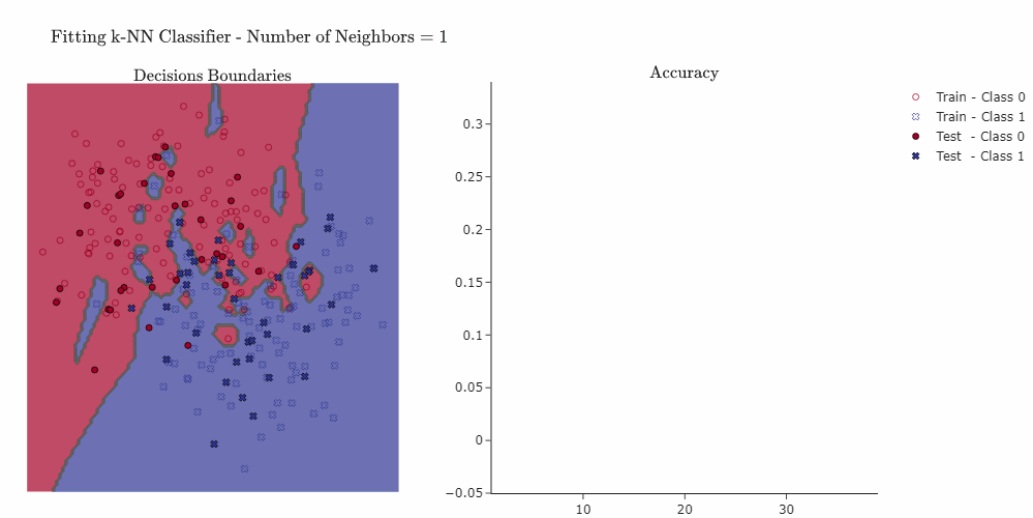


Figure 3.14:* *Decision Boundaries of k-NN: *Fitting model over dataset for different values of k.*
Classification Examples

3.6.3 Computational Implementation

Implementing a k-nearest-neighbors classifier is very easy on small datasets, but becomes computationally challenging (either in terms of execution time or in terms of space) when d and/or m are large. There are generally three types of implementation approaches:

- **Brute force implementation:** We keep the entire training sample S in storage during the entire prediction process. For each new test sample $\mathbf{x} \in \mathbb{R}^d$ we calculate $\rho(\mathbf{x}, \mathbf{x}_i)$ $i \in [m]$ and partially sort to find the k smallest distances.
  Suppose ρ is the Euclidean distance. What are the computational costs of prediction? As the sample space is $\mathbb{R}^d$ computing the distance between two points is $\mathcal{O}(d)$. Doing so for all points in the dataset is $\mathcal{O}(dm)$. Next we want to retrieve the k nearest train samples. If $k \ll m$ we can retrieve k times the sample of minimal distance (without repeating previously selected samples) in a time complexity of $\mathcal{O}(km)$. However, if $k \approx ....$ then it is more computationally efficient to sort all distances and then select the k minimal. Lastly, summation over selected samples is done in $\mathcal{O}(k)$. All together the time complexity of such approach is $\mathcal{O}(dm + km)$. In terms of space complexity we must store distances of all training samples and thus $\mathcal{O}(m)$.

- **Exact nearest neighbors search with preprocessed data structure:** Depending on the selection of ρ, we could pre-process the training sample and construct a special data structure. After doing so in the training step, we can use this data structure to quickly locate the k nearest neighbors of a given test sample. In the case of ρ being the Euclidean distance we could use an algorithm such as *kd-tree*.

- **Fast randomized nearest neighbors search:** Beyond the scope of this course.

3.6.4 Learner ID Card

- **Hypothesis class:** This is a "model free" learner and therefore has no hypothesis class
- **Learning principle used for training:** There is no training phase for this learner
- **Computational implementation:** Different strategies for calculating the nearest neighbors. Simplest method is by brute force nearest neighbor search
- **Interpretability:** We do not know why s given sample was predicted as it was. We can only explain

near what other training samples it is
- **Family of models:** Indexed according to the hyper-parameter k
- **Storing fitted model:** Must store the entire training sample or some preprocessed data structure
- **Prediction of new sample:** Find the k samples in the training set closest (with respect to the used metric) to the given sample. Predict based on the majority vote of these samples
- **When to use:** When implementation is computationally feasible try this model.

3.7 Decision Trees

Decision Trees are classification and regression methods by which we partition the sample space into disjoint parts. Then, given such a partition, the response of our classification (or regression) is computed based on the training samples in the partition of the observation in question. These methods are very intuitive and yet capture many interesting aspects of learning. We will discuss some of these aspects in details in the following chapter.

R To this day, one of the more powerful classification and regression algorithms is what is called: Classification And Regression Trees (CART) Random Forest. It uses the power of committee decisions over the basic decision trees to achieve very good performances. Some of the aspects of this algorithm will be discussed in later chapters.

3.7.1 Axis-Parallel Partitioning of $\mathbb{R}^d$

Earlier in this chapter we have discussed two classifiers that use piecewise-constant prediction rules: half-spaces and SVM. For both, the hypothesis class consisted of half-spaces where prediction was determined by position of sample with respect to the hyper-plane. For decision trees we will describe a more complicated piecewise-constant prediction rule (more complex hypotheses). Let us consider a rule that partitions the sample space $\mathbb{R}^d$ into **axis-parallel boxes, or "hyper-rectangles"** where each box is associated with labels 1 or -1. The learner's task would be to use the training sample to "chop" the samples space $\mathcal{X}$ int a disjoint union of axis-parallel boxes, and to assign a class prediction to each box.

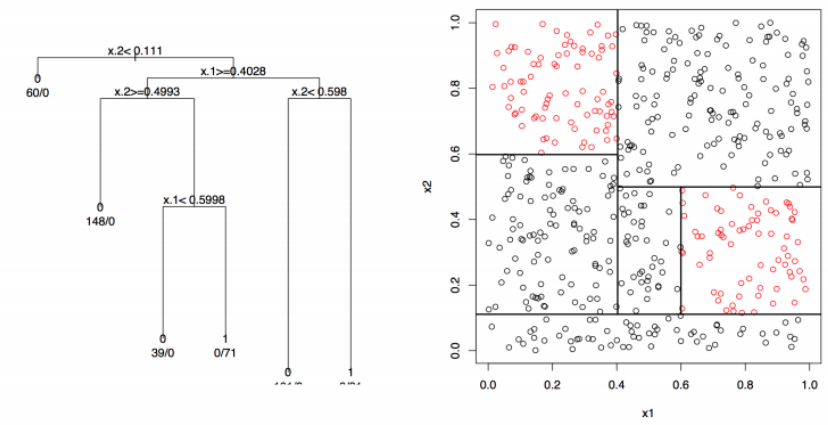


Figure 3.15: *Decision tree and induced partitioning of $\mathbb{R}^2$ sample space*

To make out hypothesis $\mathcal{H}_{CT}$ class smaller and simpler, we will focus on disjoint unions of boxes that are obtained by iteratively splitting an existing box into two smaller boxes along one of the axes:

- We start with the whole sample space $\mathbb{R}^d$.
- By selecting some coordinate $i_1 \in [d]$ and some value $t_1 \in \mathbb{R}$ we split R^d into two axis-parallel "boxes" (half-spaces). We obtain:

$$B_1^+ = \left\{\mathbf{x} \in \mathbb{R}^d | \mathbf{x}_{i_1} > t_1\right\}, \quad B_1^- = \left\{\mathbf{x} \in \mathbb{R}^d | \mathbf{x}_{i_1} \leq t_i\right\}$$

- Next, by focusing of some previously split "box" B_j^s for $s \in \{-,+\}$, we can again select some coordinate $i_{j+1} \in [d]$ and some splitting value $t_{j+1} \in \mathbb{R}^d$ to obtain B_{j+1}^-, B_{j+1}^+. Notice that B_{j+1}^- and B_{j+1}^+ are disjoint, and if following this procedure then by induction they are also disjoint from any other obtained box.

Note that the partitions obtained this way are special - most partitions of $\mathbb{R}^d$ into axis-aligned boxes are not Tree Partitions. Namely, cannot be constructed by such a top-down iterative chopping procedure.

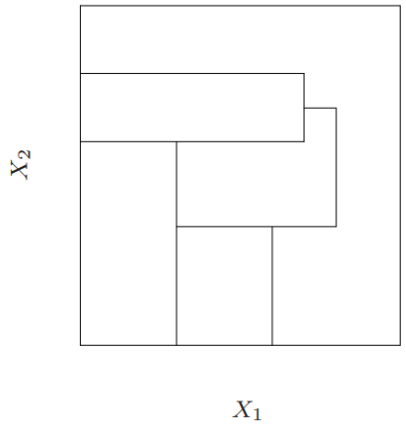


Figure 3.16: *Partitioning $\mathbb{R}^2$ into axis-aligned boxes not describing a tree partition*

3.7.2 Classification & Regression Trees

The hypothesis class $\mathcal{H}_{CT}$ we will consider consists of piecewise-constant functions, that assign a class prediction (1 or 0) to each box in a Tree Partition. Unless we restrict it somehow, the class contains all piecewise-constant functions supported on all Tree Partitions of $\mathbb{R}^d$ (to any number of boxes). Formally, for a Tree Partition $\mathbb{R}^d = \biguplus_{j=1}^{N} B_j$ of $\mathbb{R}^d$ into N boxes, and label assignments $c_j \in \{0,1\}$ $(j = 1,\ldots,N)$ assigning label c_j to box B_j, the hypothesis $h \in \mathcal{H}_{CT}$ is a function $h : \mathbb{R}^d \to \{0,1\}$ defined by

$$h(\mathbf{x}) \coloneqq \sum_{j=1}^{N} c_j \mathbb{1}_{\mathbf{x}\in B_j}$$

■ **Example 3.5** Consider the following scenario: suppose someone comes into a hispital emergency room. The first step of triage is to determine - fast - whether they are in a life-threatening medical emergency, or else they can wait in line and receive treatment in a little while. The triage uses a sequence of yes/no questions, such as: Is the patient conscious yes/no?

- If not conscious: classify as **emergency**
- If patient is conscious: is the patient's pulse < 40 beats per minute?
  - If yes (pulse < 40): classify as **emergency**
  - If no, (pulse ≥ 40): is the patient's pulse > 130 beats per minute?
    - If yes (pulse > 130): is the patient's systolic blood pressure < 80 mmHg?
      - If yes classify as **emergency**
      - If not, is the patient's systolic blood pressure > 140 mmHg? If yes, classify as **emergency**. Otherwise, classify as **no emergency**
    - If not classify as **no emergency**

This is a decision tree that uses three features: conscious (a binary categorical feature), pulse (a numerical feature) and blood pressure (also a numerical feature). See if you can we write a diagram for this decision tree in the shape of a tree, where every node is a question, and every leaf is a decision / classification. The root of the tree is the first question ("conscious yes/no?"). Now observe that every function in our Classification Trees hypothesis class $\mathcal{H}_{CT}$ is equivalent to a decision tree. In the notations of the generic example above, the first question is: "$x_{i_1} > t_1$-yes/no?". If yes, we ask the second question "$x_{i_{2,1}} > t_{2,1}$ - yes/no?". If not, we ask

the second question: "$x_{i_{2,2}} > t_{2,2}$ - yes/no?". And so on, until there are no more splits and we have reached a box over which the function in $\mathcal{H}_{CT}$ is constant. If the constant value is 1, we classify / predict class 1. If the constant value is 0, we predict class 0. This is why our hypothesis class is called - classification trees ■

3.7.3 Growing a Classification Tree

Having defined our hypothesis class, the next question is what learning principle to use. That is, how shall we select $h_s \in \mathcal{H}_{CT}$ based on the training sample S. Suppose we have already obtained a Tree Partition of $\mathbb{R}^d$ is some manner, that consists of N disjoint boxes $\mathbb{R}^d = \biguplus_{j=1}^{N} B_j$. Let $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^{m}$ be our training set and denote the predicted label assigned to box B_j by $\hat{y}(B_j) \in \{0,1\}$. As such, the number of misclassification errors that are incurred by the training sample that fall inside B_j is

$$\sum_{\mathbf{x}_i \in B_j} \mathbb{1}\left[y_i \neq \hat{y}(B_j)\right]$$

Let us begin learning by applying the ERM principle. Denote the fraction of misclassified samples with label $y \in \{0,1\}$ in some box B by:

$$\ell_S(B,y) \coloneqq \frac{1}{|B|} \sum_{\mathbf{x}_i \in B} \mathbb{1}\left[y_i \neq y\right]$$

The label that will minimize the empirical risk for those training samples in box B is the **majority vote** over the labels. So, for any sample falling in box B we would predict

$$\hat{y}_S(B) \coloneqq \underset{y \in \{0,1\}}{\operatorname{argmin}} \ \ell_S(B,y)$$

Applying over the entire Tree Partition, minimizing the empirical risk is achieved by labeling box B_j with $\hat{y}_S(B_j)$, $j \in [N]$. Therefore, for a given training set S, every Tree Partition corresponds with a unique label assignment, and as such, a unique classification tree $h \in \mathcal{H}_{CT}$ that minimizes the empirical risk. It seems therefore, that finding the desired ERM tree is done by solving

$$h^* \coloneqq \underset{h \in \mathcal{H}_{CT}}{\operatorname{argmin}} \ L_S(h)$$

where $L_S(h)$ is the misclassification error of h over S. Looking back at the described procedure is it therefore possible to describe which tree would minimize the empirical risk and therefore be selected (for any training set S)? Consider the tree where the number of leaves is $|S|$ and each sample is in a box containing only itself. Following the prediction rule we devised, such a tree would achieve $L_S(h) = 0$. Though we achieved the lowest possible empirical risk, this tree will fail to generalize to new samples. To cope with this problem we should limit the number of levels in the classification tree (equivalent for limiting number of leaves). Denote $\mathcal{H}_{CT}^k$ the hypothesis class of all tree partitions with at most k levels. Now, we will choose k and then using the ERM principle return

$$h^* \coloneqq \underset{h \in \mathcal{H}_{CT}^k}{\operatorname{argmin}} \ L_S(h) \tag{3.36}$$

Selecting A Value For k:

Note that by adding the hyper-parameter k we now have *a family of hypothesis classes*, one for each value of k. The value of k controls the size of the hypothesis class and therefore controls the bias-variance tradeoff (Figure 3.17):

- For small values of k, the hypothesis class is smaller, containing trees of smaller sizes. Therefore the ERM learner will have a **higher** bias as it can only select simple Tree Partitions. It will also have a **lower** variance: as the boxes are very large, the labels assigned to each box are based on a majority vote of typically many training samples. Therefore changing a few training samples will barely change the selected hypothesis.
- For large values of k, the hypothesis class is much more complex, with more "specialized" trees in it. Therefore the ERM learner will have a **lower** bias and **higher** variance.

Later in the course we will introduce a few methods for selecting the value of k.

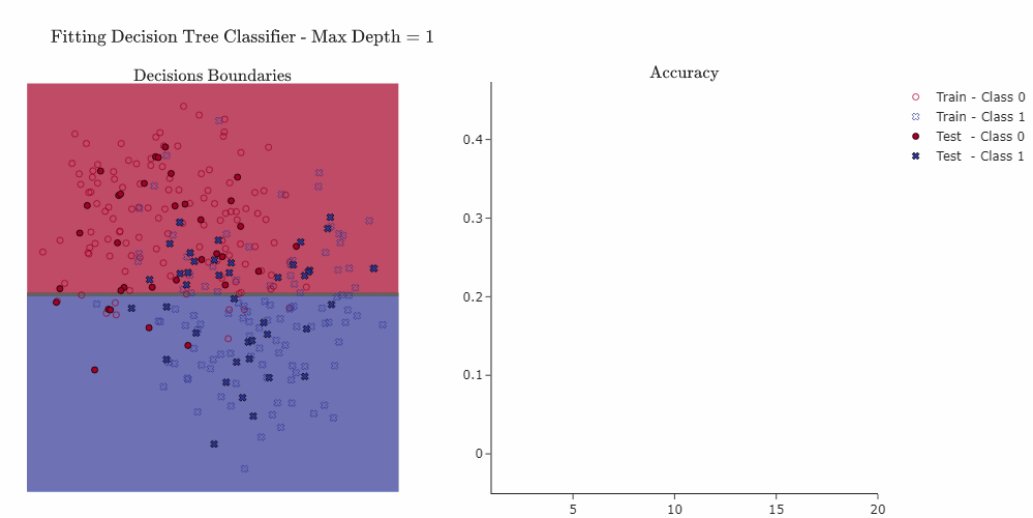


Figure 3.17: ***Decision Boundaries of Decision Trees:*** *Fitting model over dataset for different values of max depth k. Classification Examples*

3.7.4 CART Heuristic For Growing Trees

The next challenge is how to find the minimizer of (3.36) computationally? So far, all the ERM learners we encountered were **computationally tractable**:

- The linear regression optimization problem was based on ERM. We were able to find aclosed form expression for the minimizer.
- The Half-space classifier was based on ERM and lead to a simple convex optimization problem.

In contrast to those examples the search space over $\mathcal{H}_{CT}^k$ is exponentially large and has no Euclidean or other structure to be used. Finding an ERM solution would mean to use brute-force search, which is infeasible. In fact, it has been proven that implementing ERM on $\mathcal{H}_{CT}^k$ is an NP-Hard problem with respect to the training sample size[1] .

This is our first encounter with the bitter truth that though the ERM principle is nice, it is often impossible to implement efficiently, especially when the hypothesis class has no Euclidean structure. Therefore, we must resort to defining and using **heuristics**: an approach to solving the optimization problem that is not guaranteed to be optimal, but is still sufficient for finding a solution. While the definition of decision trees, the hypothesis class and prediction assignment to boxes in a tree partition are all canonical, there are several different heuristic approaches to the way we "grow a decision tree', namely to the way we choose an hypothesis in practice.

One common approach, coming out of the statistical learning community, is called **Classification and Regression Trees** (CART). This heuristic cinsists of two stages: **growing** the tree, resulting in a tree that is a little too large, and then **pruning** it to bring it down to the most effective size.

Suppose we have chosen k to be the maximal tree depth. The heuristic of growing a full decision tree with at most k levels will proceed top-down, starting from $\mathbb{R}^d$ and progressively chopping each box into two boxes. A given box is **not chopped** if either:

- The maximum number of levels k has been reached.

[1] By reduction from "three dimensional matching", see Hyafil and Rovest, "Constructing Optimal Binary Decision Trees is NP-Complete", Information Processing Letters 5(1), 1976

- The box has reached a pre-determined minimal number of training samples. At the very least we would not split a box if it consists of only a single training sample.

Chopping is done by finding the **best** coordinate, at the **best** value to chop, and whenever you chop given each half-box the **best** class assignment, in the sense of minimizing misclassification error over the training sample. Formally, the pseudocode of the CART heuristic is seen in Algorithm 3.

Algorithm 3 CART - For Growing Classification Tree

procedure CART($S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$, k, m_{min})
  Tree-Partition $\leftarrow \emptyset$
  Boxes $\leftarrow \left\{\mathbb{R}^d\right\}$ ▷ Entire sample space as initial box
  while Boxes $\neq \emptyset$ **do**
    $B \leftarrow$ pop(Boxes)
    if $|B| \leq m_{min}$ or depth at B reached k **then**
      Tree-Partition $\leftarrow$ Tree-Partition $\cup \{B\}$
      continue
    end if
    for all feature $i \in [d]$ **do** ▷ Scan all features
      for all threshold value $t \in \mathbb{R}$ **do** ▷ Scan thresholds for features
        Split B along coordinate i at value t:

$$B_{i,t}^+ := \left\{\mathbf{x} \in \mathbb{R}^d | \mathbf{x}_i > t\right\}, \quad B_{i,t}^- := \left\{\mathbf{x} \in \mathbb{R}^d | \mathbf{x}_i \leq t\right\}$$

        Let $\hat{y}(B_{i,t}^{\pm})$ denote the class assignment for boxes $B_{i,t}^{\pm}$.
        Let $\ell_S(B_{i,t}^{\pm}, \hat{y}(B_{i,t}^{\pm}))$ denote the empirical risk incurred by $\hat{y}(B_{i,t}^{\pm})$.
        Define $g_i(t) := \ell_S(B_{i,t}^+, \hat{y}(B_{i,t}^+)) + \ell_S(B_{i,t}^-, \hat{y}(B_{i,t}^-))$
      end for
      Set the **best** splitting point: $t_i \leftarrow \text{argmin}_{t \in \mathbb{R}}\, g_i(t)$
    end for
    Select **best** feature to split by: $i^* \leftarrow \text{argmin}_{i \in [d]}\, g_i(t_i)$
    Boxes $\leftarrow$ Boxes $\cup \left\{B_{i^*,t_{i^*}}^+, B_{i^*,t_{i^*}}^-\right\}$ ▷ Split box by best feature and threshold
  end while
  return *Boxes*
end procedure

Time Complexity Analysis

Before we introduced the CART heuristic we described that solving the ERM principle over this hypothesis class is NP-Complete and therefore cannot be done in polynomial time. Let us see that the CAR heuristic can indeed be computed efficiently.

The algorithm iteratively splits boxes into two by finding a coordinate $i \in [d]$ and a value $t \in \mathbb{R}$. It scans all d coordinates and for each scans all possible values of t. Notice, that even though $t \in \mathbb{R}$ we do not need to try all values and it suffices to check only the values in the i'th coordinate of the training sample: $\{\mathbf{x}_i | \mathbf{x} \in S\}$. As we have m samples we only need to evaluate for at most m values, giving each step a time complexity of $\mathcal{O}(md)$.

The next question is how many steps will the algorithm perform? We know that the algorithm will terminate after growing a tree with at most k levels. Such a tree will have at most $2^k - 1$ nodes and leaves. Though this

seems exponentinal in the given input (notice that the hyper-parameter k is also part of the input) we can upper bound this value. As we do not allow empty boxes (and in fact any box with less than some minimal number of samples) the number of nodes (including leaves) in the tree is at most m. We therefore conclude that the time complexity of the CART heuristic is $\mathcal{O}\left(m^2 d\right)$

Pruning a Decision Tree

Pruning a tree means cutting off unnecessary branches. The tree obtained when we are done with the "growing" stage of CART may be too large. A tree too large means some of the boxes are too small, so we are not in an optimal point on the bias-variance tradeoff. It could help reduce the generalization error to merge some of the boxes together, so that the majority votes to determine the box label assignment would be based on larger sets of training samples. Merging two boxes is equivalent, from the decision tree perspective, to merging to leaves together and removing the node between them. Hence, "pruning". We will complete the CART heuristic when we discuss regularization (6.0.1).

3.7.5 Interpretability

One of the great advantages of a classification tree is that it is very interpretable. To understand which features were important in the classification process, we just look at the nodes (the splits) and see which features the classification tree algorithm chose to split on. A feature that never appeared in any split has not been useful for classification of the training sample. A feature that appears once or more (remember that the algorithm can choose to split on some feature again and again in different areas of R^d) has been useful. To understand why a new sample was classified the way it was classified, we just follow the tree from top to bottom, and see how each answer to each question went.

3.7.6 Learner ID Card

- **Hypothesis class:** The piecewise-constant functions induced by Tree Partitions (axis-aligned rectangles) of depth at most k: $\mathcal{H}_{CT}^k$
- **Learning principle used for training:** ERM
- **Computational implementation:** Implementation of ERM is NP-Hard. Therefore we use heuristics such as the top-down greedy heuristic of CART
- **Interpretability:** A very interpretable learner. Simply reading the tree structure
- **Family of models:** Indexed by k the maximal tree depth
- **Storing fitted model:** To store this model we must store for each node the split information: the coordinate i by which to split and the threshold value t. In addition we need to store the label assignment at the leafs of the tree
- **Prediction of new sample:** Navigate top-down along the tree until reaching a leaf
- **When to use:** This classifier is used as a simple baseline or to get a highly interpretable rule that is easy to explain and plot. Otherwise, classification trees are used to construct "random forests" which are covered in 5.3.4

3.8 Summary and Exercises

Exercises

Theoretical Questions

1. Prove that following Hard-SVM optimization problem is a Quadratic Programming problem:

$$\underset{(\mathbf{w},b)}{\operatorname{argmin}} ||\mathbf{w}||^2 \quad \text{s.t.} \quad \forall i \, y_i \left(\langle \mathbf{w}, \mathbf{x}_i \rangle + b\right) \geq 1$$

   That is, find matrices Q and A and vectors $\mathbf{a}$ and $\mathbf{d}$ such that the above problem can be written in the following format

$$\underset{\mathbf{v}\in\mathbb{R}^n}{\operatorname{argmin}} \tfrac{1}{2}\mathbf{v}^\top Q\mathbf{v} + \mathbf{a}^\top \mathbf{v} \quad \text{s.t.} \quad A\mathbf{v} \leq \mathbf{d}$$

2. The ***Gaussian*** **Naive Bayes** classifier assumes a multinomial prior and independent feature-wise Gaussian likelihoods:

$$\begin{aligned} y &\sim \text{Multinomial}\left(\boldsymbol{\pi}\right) \\ x_j | y = k &\overset{ind.}{\sim} \mathcal{N}\left(\mu_{kj}, \sigma_{kj}^2\right) \end{aligned} \tag{3.37}$$

   for $\boldsymbol{\pi}$ a probability vector: $\boldsymbol{\pi} \in [0,1]^K, \sum \pi_j = 1$.

   (a) Suppose $x \in \mathbb{R}$ (i.e each sample has a single feature). Given a trainset $\{(x_i, y_i)\}_{i=1}^m$ fit (i.e find expressions for the maximum likelihood estimators) a Gaussian Naive Bayes classifier solving (3.5.1) under assumptions (3.37).

   (b) Suppose $\mathbf{x} \in \mathbb{R}^d$ (i.e each sample has d feature). Given a trainset $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ fit a Gaussian Naive Bayes classifier solving (3.5.1) under assumptions (3.37).

3. The ***Poisson*** **Naive Bayes** classifier assumes a multinomial prior and independent feature-wise Poisson likelihoods:

$$\begin{aligned} y &\sim \text{Multinomial}\left(\boldsymbol{\pi}\right) \\ x_j | y = k &\overset{ind.}{\sim} \text{Poi}\left(\lambda_{kj}\right) \end{aligned} \tag{3.38}$$

   for $\boldsymbol{\pi}$ a probability vector: $\boldsymbol{\pi} \in [0,1]^K, \sum \pi_j = 1$.

   (a) Suppose $x \in \mathbb{R}$ (i.e each sample has a single feature). Given a trainset $\{(x_i, y_i)\}_{i=1}^m$ fit a Poisson Naive Bayes classifier solving (3.5.1) under assumptions (3.38).

   (b) Suppose $\mathbf{x} \in \mathbb{R}^d$ (i.e each sample has d feature). Given a trainset $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ fit a Poisson Naive Bayes classifier solving (3.5.1) under assumptions (3.38).

Practical Questions

Clone the IML.HUJI GitHub repository and setup a working python environment as explained on GitHub page.

Running Perceptron On Different Data Scenarios

Implement the `misclassification_error` function in the `metrics/loss_functions.py` file and the Perceptron algorithm in the `learners/classifiers/perceptron.py` file as described in documentation.

In the `exercises/classifiers_evaluation.py` file, implement the `run_perceptron` function as described in documentation. To retrieve the loss at each iteration *do not* change the previously implemented Perceptron class. Instead *specify a callback function* which receives the object and uses it's `loss` function to calculate the loss over the training set. Store these values in an array to be used for plotting.

1. Fitting and plotting over the `datasets/linearly_seprable.npy` dataset, what can we learn from the plot?

2. Next run the Perceptron algorithm over the `datasets/linearly_inseprable.npy` dataset and plot its loss as a function of the iterations. What is the difference between this plot and to the one in the previous question? How can we explain the difference in terms of the objective and parameter space?

Compating LDA and Gaussian Naive Bayes

Complete the following implementations:

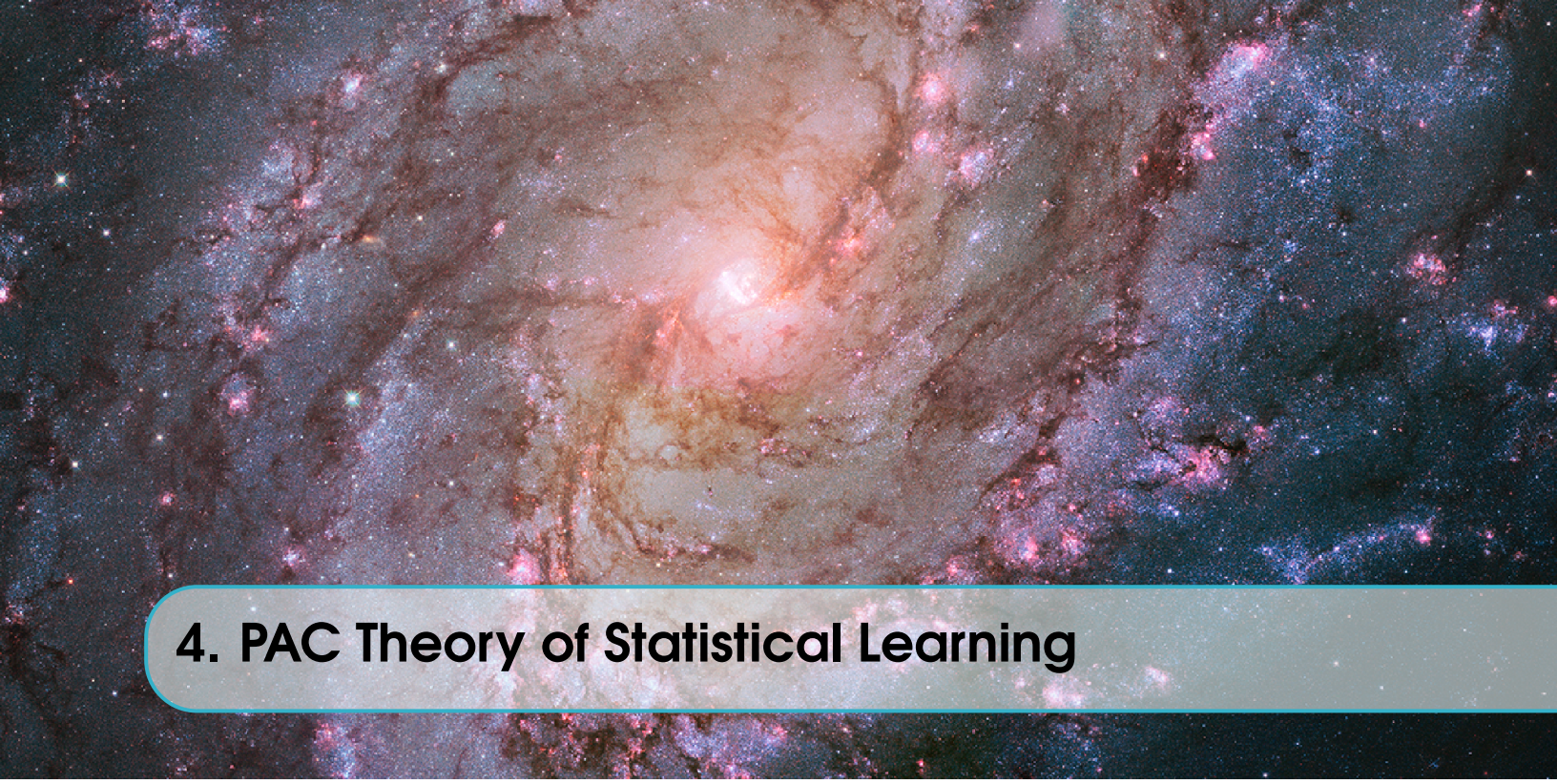
- Implement the `accuracy` function in the `metrics/loss_functions.py` file as described in the function documentation.
- Implement the `LDA` classifier in the `learners/classifiers/linear_discriminant_analysis.py` file as described in the class documentation. Use expressions derived in class.
- Implement the `GaussianNaiveBayes` classifier in the `learners/classifiers/gaussian_naive_bayes.py` file as described in the class documentation. Use expressions derived in question 3b of the theoretical part.

Then, implement and answer the following questions:

1. In the `compare_gaussian_classifiers` function, `classifiers_evaluation.py` file, load the `datasets/gaussians1.npy` dataset. Fit both the Gaussian Naive Bayes and LDA algorithms previously implemented. Plot the following:
   - A single figure with two subplots:
     (a) 2D scatter-plot of samples, with marker color indicating Gaussian Naive Bayes *predicted* class and marker shape indicating *true* class.
     (b) 2D scatter-plot of samples, with marker color indicating LDA *predicted* class and marker shape indicating *true* class.
     (c) Provide classifier name and accuracy (over train) in sub-plot title
   - For both subplots add:
     (a) Markers (colored black and shaped as 'X') indicating the center of fitted Gaussians.
     (b) An ellipsis (colored black) centered in Gaussian centers and shape dictated by fitted covariance matrix.
   - Specify dataset name in figure title.

   Explain what can be learned from the plots above regarding the distribution used to sample the data?

2. Repeat the procedure above (while avoiding code repetition) for `datasets/gaussians2.npy`. What is the difference between the two scenarios? What can be learned regarding the distribution used to sample the data? Which of the two classifiers better matches this dataset and why?



4. PAC Theory of Statistical Learning

4.1 A Theoretical framework for learning

The basic questions in machine learning are: Which tasks are learnable? How do we learn learnable tasks? How many training samples do we need in order to learn them? In this chapter we develop the *PAC theory of learnability*, which provides us, within its definitions and assumptions, a complete answer to these questions, for *batch supervised learning*.

A Data-generation Model

The two basic assumptions in the PAC framework are:

- There exists a (deterministic) function f which is the correct classifier, i.e., for every $\mathbf{x}$ there is a single correct label, given by $y = f(\mathbf{x})$. We shall refer to this case as the *PAC Model*.

- All samples, either in the training set or in any future test set are independent and identically distributed (i.i.d) random variables, i.e., they are sampled independently using a distribution $\mathcal{D}$ over the sample space $\mathcal{X}$. In particular this means that the probability of drawing the sequence $\mathbf{x}_1, \ldots, \mathbf{x}_m$ (which equals, due to the previous assumption, to the probability of getting the sequence $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ with $y_i = f(\mathbf{x}_i)$) is given by $\mathbb{P}(S) = \prod_{i=1}^m \mathcal{D}(\mathbf{x}_i)$.

R Later on, in section 4.4, we shall relax these assumptions and consider a more general case which we will call the *Agnostic PAC model* and in which the *same* $\mathbf{x}$ may appear with *different* labels. It means that the labels themselves will be random, at least to some extent, and therefore also the probability density will be defined over $\mathcal{X} \times \mathcal{Y}$.

Let us compare the assumptions of the PAC model to the linear regression model covered in chapter 2. In both cases we have received a set of examples $\mathbf{x}_1, \ldots, \mathbf{x}_m \in \mathcal{X}$. In the case of linear regression we implicitly assumed all the $\mathbf{x}_i$'s had equal importance, for example, in their contribution to the global error. Now, the $\mathbf{x}_i$'s are i.i.d sampled according to $\mathcal{D}$, i.e., they have different probabilities to appear and therefore may have different weights in the loss function. Another difference is that in the case of the linear model we started with

the assumption $y_i = f(\mathbf{x}_i)$ where f was deterministic and linear. Here, we still assume f to be deterministic, but it may take the form of any possible function from $\mathcal{X}$ to $\mathcal{Y}$, not only linear.

In the linear model we eventually relaxed the deterministic assumption and considered y to be a random function of $\mathbf{x}$ of the form: $y_i = f(\mathbf{x}_i) + z_i$. An analogous generalization will take place also here, once we consider the more general, agnostic, case. Finally we note that, although this chapter will focus on *classifiers*, many principles we will encounter will hold also for linear regression problems.

Generalization Error for Classifiers

We define the *generalization error* of a hypothesis h as the probability to obtain an $\mathbf{x}$ for which $h(\mathbf{x})$ is different than the true $f(\mathbf{x})$:

$$L_{\mathcal{D},f,\ell}(h) \coloneqq \mathbb{E}_{\mathbf{x}\sim\mathcal{D}}\left[\ell(h(\mathbf{x}), f(\mathbf{x}))\right] \tag{4.1}$$

where $\mathcal{D}$ and f are unknowns and ℓ some loss function. The generalization error is also called the *True Error*, or the *Risk* (Definition 1.1.4). For a classification task, we can measure this error with respect to the misclassification error.

$$L_{\mathcal{D},f,\ell}(h) \;\coloneqq\; \mathbb{E}_{\mathbf{x}\sim\mathcal{D}}\left[\ell(h(\mathbf{x}), f(\mathbf{x}))\right] = \mathbb{P}_{\mathbf{x}\sim\mathcal{D}}\left[h(\mathbf{x}) \neq f(\mathbf{x})\right] \;=\; \mathcal{D}\left(\{\mathbf{x}\in\mathcal{X} : h(\mathbf{x}) \neq f(\mathbf{x})\}\right) \tag{4.2}$$

Note, one should be critical about an error measure that counts the total number of misclassification errors of a classifier. Recall the distinction we make between the Type-I and Type-II errors, where often the one is worse than the other. For now however, we only consider the generalization error with respect to the misclassification error and therefore do not distinguish between the two types of errors and simply denote by $L_{\mathcal{D},f}$.

The Fundamental Theorem of Statistical Learning

So our task is to design a learning algorithm (a learner), $\mathcal{A}$. The algorithm will receive a training sample of size m, $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$, and outputs a prediction rule (a hypothesis) $h : \mathcal{X} \to \mathcal{Y}$. For classification problems, for example, $\mathcal{Y} = \{\pm 1\}$, the framework we develop has a much broader applicability.

We assume that the data points, both in the training set and in the test set, are generated by sampling independently $\mathcal{X}$ using a distribution $\mathcal{D}$, which is unknown to us. The labels y are fixed: given a particular $\mathbf{x}$ there exists some deterministic function f such that $y = f(\mathbf{x})$, where f is unknown to us and the training set is our only view to what f does.

Finally, the performance of any candidate rule $h : \mathcal{X} \to \mathcal{Y}$ that our learner may produce, will be evaluated by how well it will perform on future, unseen samples. This is done using the expected misclassification rate $L_{\mathcal{D},f}(h) \;\equiv\; \mathbb{P}_{\mathbf{x}\sim\mathcal{D}}[h(\mathbf{x}) \neq f(\mathbf{x})]$. The next sections will be devoted for a detailed understanding of the following important definitions:

Definition 4.1.1 A hypothesis class, $\mathcal{H}$, is a *PAC Learnable hypothesis class* if there exist a learning algorithm $\mathcal{A}$ and a function $m_{\mathcal{H},\mathcal{A}} : (0,1)^2 \to \mathbb{N}$ with the following property:

- For every $\varepsilon, \delta \in (0,1)$
- For every distribution $\mathcal{D}$ over $\mathcal{X}$
- For every labeling function $f : \mathcal{X} \to \{\pm 1\}$ such that there exists $h^* \in \mathcal{H}$ which satisfies $L_{\mathcal{D},f}(h^*) = 0$

when running the learning algorithm $\mathcal{A}$ on $m \geq m_{\mathcal{H},\mathcal{A}}(\varepsilon, \delta)$ i.i.d samples generated by $\mathcal{D}$ and labeled by f, the algorithm returns a hypothesis $h_S = \mathcal{A}(S)$ such that, with probability of at least $1-\delta$ (over the choice of the training samples), we have $L_{\mathcal{D},f}(h_S) \leq \varepsilon$.

$$\mathcal{D}^m\left(S \text{ s.t. } L_{\mathcal{D},f}(h_S) \leq \varepsilon\right) \geq 1 - \delta$$

where $\mathcal{D}^m$ is the probability space of sampling m samples from D. Denote the minimal sample size required for the above conditions to hold with respect to ε, δ and with respect to any algorithm, by $m_{\mathcal{H}}(\varepsilon, \delta) = \min_{\mathcal{A}} m_{\mathcal{H},\mathcal{A}}(\varepsilon, \delta)$. The function $m_{\mathcal{H}} : (0,1)^2 \to \mathbb{N}$ is called the *Sample Complexity* of the PAC

learnable hypothesis class $\mathcal{H}$.

Definition 4.1.2 Let $\mathcal{H} \subseteq \{\pm 1\}^{\mathcal{X}}$ be a hypothesis class. For a subset $C \subset \mathcal{X}$ let $\mathcal{H}_C$ be the restriction of $\mathcal{H}$ to C, namely, $\mathcal{H}_C = \{h_C : h \in \mathcal{H}\}$, where for $h : \mathcal{X} \to \mathcal{Y}$, $h_C : C \to \mathcal{Y}$ is the function such that $h_C(\mathbf{x}) = h(\mathbf{x})$ for every $\mathbf{x} \in C$. Define the *VC-dimension* of $\mathcal{H}$ by:

$$VCdim(\mathcal{H}) = \max\left\{|C| \;\middle|\; C \subset \mathcal{X} \text{ and } |\mathcal{H}_C| = 2^{|C|}\right\}$$

By choosing PAC learnability as our interpretation of what learnability means, definitions Definition 4.1.1 and Definition 4.1.2 provide a *well defined necessary and sufficient condition* for when learning is possible and the minimal training sample size needed in order to learn. This framework also provides a "universal" learner that successfully learns given a sufficiently large training set. In short, we have a full theory of batch learning - a full theory of when it is possible to generalize from a training sample to new samples, and how to do it. This result is sometimes known as "*The Fundamental Theorem of Statistical Learning*" and states that:

- A hypothesis class $\mathcal{H}$ is PAC-learnable *if and only if* $VCdim(\mathcal{H})$ is finite.
- The sample complexity of a hypothesis class with a finite VC-dimension is given approximately by

$$m_{\mathcal{H}}(\varepsilon, \delta) \sim \frac{VCdim(\mathcal{H}) + \log(1/\delta)}{\varepsilon}$$

- The ERM rule achieves this minimum. Namely, when learning is possible, ERM learns with a minimial number of examples.

The first step in gaining a detailed understanding of PAC-learnability, VC-dimension and the fundamental theorem will be to present a different perspective of the above framework.

4.1.1 Learning as a Game - First Attempt

The framework in section 4.1 can be thought of as a *game* between us and Nature, with a random payoff. The game proceeds as follows. First, the number of training samples m is determined in advanced as a game parameter. We perform the first step and choose some learner $\mathcal{A} : (\mathcal{X} \times \mathcal{Y})^m \to \mathcal{H}$. This is our strategy. Then, Nature makes the second step and chooses a probability distribution $\mathcal{D}$ and a labeling function f. This is Nature's strategy. Importantly, Nature knows the strategy we chose when she chooses her strategy. To calculate the game's payoff, an i.i.d sample S of size m is drawn according to $\mathcal{D}$ and labeled according to f, both of which were chosen by Nature. This set is then fed into the learner $\mathcal{A}$ that we chose to obtain the prediction rule $h_S := \mathcal{A}(S)$. The notation h_S emphasizes that the prediction rule we learn, h_S, strongly depends on the random sample S. The payoff is the generalization error, with respect to the misclassification error, of the returned hypothesis $L_{\mathcal{D},f}(h_S)$.

Learnability as a game - first attempt:
- Fix a sample size m and then:
- Us: choose a strategy (a learner) $\mathcal{A}$
- Adversary: choose a strategy that consists of $\mathcal{D}$ over $\mathcal{X}$, and a label function $f : \mathcal{X} \to \mathcal{Y}$.
- Sample S of size m is drawn i.i.d according to $\mathcal{D}$ and fed into $\mathcal{A}$, to produce a prediction rule
  - $h_S = \mathcal{A}(S)$.
  - Payoff: $L_{\mathcal{D},f}(h_S)$

Our goal in the game is to end up with an $L_{\mathcal{D},f}(h_S)$ which is as small as possible while Nature is an adversary that wants $L_{\mathcal{D},f}$ to be as large as possible. Note that the payoff is random as it depends on the sample S that

was drawn. If we are unlucky, and the sample S is "bad" (namely, does not represent $\mathcal{D}$ very well), then the rule h_S our learner produces might not generalize well and the random loss (for that draw of S) will be high.

Under such setting, what is the best strategy for us? How to best design the learner $\mathcal{A}$? Remember that Nature will know what we chose, and can try to be "cruel", namely, to choose $\mathcal{D}$ and f that our learner did not prepare well for. Also, there is always a chance that we will draw a "bad" sample S, which will mislead us in choosing a rule h_S that will not generalize well. So, is there a way to defend ourselves against "cruel" strategies $\mathcal{D}, f$ that Nature might play, and against unlucky draws of a training sample S? Is there anything at all that can be said in this generality about the problem of learning (i.e., the problem of generalizing from training samples to new samples)?

4.1.2 Probably Correct & Approximately Correct Learners

The generalization loss, $L_{\mathcal{D},f}(h_S)$, is random since it depends on the randomly-drawn training sample, S. Therefore, in general, one should talk about the *probability* that a learner has a certain loss. In particular, saying that $L_{\mathcal{D},f}(h_S)$ *never* exceeds a certain value, actually means that (for the specific f, $\mathcal{D}$ and $\mathcal{A}$) the probability of drawing a training sample S for which $\mathcal{A}$ produces a rule with a loss smaller than a certain value, is 1. This brings us to the following definition:

Definition 4.1.3 Let $\varepsilon \in (0,1)$. We say that a learner $\mathcal{A}$ is *Approximately Correct* with an *accuracy* ε, if and only if

- For every $\mathcal{D}$ over $\mathcal{X}$
- For every labeling function $f : \mathcal{X} \to \{\pm 1\}$ such that $\exists h^* \in \mathcal{H}, L_{\mathcal{D},f}(h^*) = 0$

We are certain (with probability 1) that for any training sample, S, drawn using $\mathcal{D}$, $\mathcal{A}$ will output a prediction rule, h_S, with a loss smaller or equal to ε:

$$\mathcal{D}\left(S \middle| L_{\mathcal{D},f}(h_S) \leq \varepsilon\right) = 1$$

We say that $\mathcal{H}$ is ε-approximately correct if $\mathcal{A}$ is approximately correct with accuracy ε.

Is there an accuracy parameter, $\varepsilon \in (0,1)$, for which we can find an approximately correct learner? Unfortunately, the answer to this question is no. For any ε, Nature always has a strategy that can make sure that there is a non-zero probability, even if very small, to get a "pathological" training sample S that does not represent $\mathcal{D}$ at all. The resulting rule h_S can be wrong on most of $\mathcal{X}$, which would cause a loss arbitrarily close to 1, higher than any specified ε. We shall prove this assertion by giving an example of one such strategy that Nature can use.

Claim 4.1.1 For every $\varepsilon \in (0,1)$ there exists $\mathcal{D}(\mathcal{X})$ and $f : \mathcal{X} \to \mathcal{Y}$ such that $\mathcal{D}\left(S | L_{\mathcal{D},f}(h_S) \leq \varepsilon\right) < 1$

Proof. Let $\varepsilon \in (0,1)$ and consider any two points $\mathbf{x}, \mathbf{x}' \in \mathcal{X}$. Let us denote by S' a training sample that happened to have only $\mathbf{x}'$'s in it (in particular, S' does not contain $\mathbf{x}$): $S' = (\mathbf{x}_1, y_1), ..., (\mathbf{x}_m, y_m)$ with $\mathbf{x}_i = \mathbf{x}'$ and $y_i = f(\mathbf{x}')$ for $i = 1, ..., m$. Although such a sample might be rare, it can not be excluded. Therefore our algorithm, $\mathcal{A}$, should specify what label its output rule, $h_{S'}$, should predict on other points beside $\mathbf{x}'$, in particular on $\mathbf{x}$, in case it receives S' as an input. Assume, without a loss of generality, that we choose $h_{S'}(\mathbf{x}) = +1$, that is, if the training sample is S' then $\mathcal{A}$ will predict $+1$ on $\mathbf{x}$.

Nature's strategy comprises of two parts. First, it reduces $\mathcal{X}$ to a 2-point space by choosing $\mathcal{D}$ that vanishes everywhere except on $\mathbf{x}$ and $\mathbf{x}'$. More specifically, it chooses $\mathcal{D}$ with $\mathcal{D}(\mathbf{x}) = \gamma$, $\mathcal{D}(\mathbf{x}') = 1 - \gamma$, where γ is a number satisfying $0 < \varepsilon < \gamma < 1$. Second, Nature chooses a labeling function f with $f(\mathbf{x}) = -h_{S'}(\mathbf{x}) = -1$. The probability of obtaining S' is not zero since it is given by $(1-\gamma)^m > 0$. Therefore, to show that Nature's

strategy prevents $\mathcal{A}$ from being approximately correct with an accuracy ε, all we need to show is that the loss in the case of obtaining the set S', is greater than ε. Indeed:

$$L_{\mathcal{D},f}(h_{S'}) = \mathcal{D}(\mathbf{x}) \cdot \mathbb{1}_{f(\mathbf{x}) \neq h_{S'}(\mathbf{x})} + \mathcal{D}(\mathbf{x}') \cdot \mathbb{1}_{f(\mathbf{x}') \neq h_{S'}(\mathbf{x}')}$$

where $\mathbb{1}_{a \neq b}$ is 1 if $a \neq b$ and 0 otherwise. Both terms on the right are non-negative and therefore:

$$L_{\mathcal{D},f}(h_{S'}) \geq \mathcal{D}(\mathbf{x}) \cdot \mathbb{1}_{f(\mathbf{x}) \neq h_{S'}(\mathbf{x})} = \gamma \mathbb{1}_{-1 \neq 1} = \gamma > \varepsilon$$

So, for any fixed $\varepsilon \in (0,1)$ and for any strategy $\mathcal{A}$ we might play, Nature has a strategy $\mathcal{D}, f$ such that there is a non-zero probability over the choice of training samples of length m, that will have $L_{\mathcal{D},f}(h_S) > \varepsilon$. As ε was arbitrary, that means that Nature can, with a non-vanishing probability, cause the game to end with a loss arbitrarily close to 1.

■

In the above example, even if very small, with probability $(1-\gamma)^m$ we get a "bad" training sample, S, that does not represent $\mathcal{D}$ good enough, and does not allow us to generalize. We therefore conclude, that given any accuracy parameter $\varepsilon \in (0,1)$, no learner $\mathcal{A}$ can guarantee, that with probability 1, the loss will not exceed ε. Nature can always find a strategy for which $L_{\mathcal{D},f}(h_S) > \varepsilon$ will have a non-zero probability to occur.

So the possibility of bad samples means that we should not aspire for an *absolute* certainty in achieving a limited loss. We will accept the fact that on such bad training samples, the learner $\mathcal{A}$ might fail completely (i.e., produce h_S with a potentially high loss). We will therefore only require a *limited* certainty, that we will call *confidence*, for having a limited loss. This means that we will demand that the probability of drawing a bad sample will not exceed a certain threshold, $\delta \in (0,1)$. This parameter too will be specified prior to the beginning of the game, together with ε.

Since we no longer demand absolute confidence in having a limited loss, perhaps we can instead require absolute accuracy (zero loss), but with a limited confidence? That is, perhaps we can require that at least for those good training samples (the ones that will have a probability of at least $1-\delta$ to appear) the loss will vanish?

Definition 4.1.4 Let $\delta \in (0,1)$. We say that a learner, $\mathcal{A}$, is *Probably Correct* with a *confidence* δ, if the probability to obtain a training sample, S, for which $\mathcal{A}$ will output a prediction rule, h_S, with a perfect accuracy (zero loss), is greater than or equal to $1-\delta$:

$$\mathcal{D}^m\left(S \text{ s.t. } L_{\mathcal{D},f}(h_S) = 0\right) \geq 1 - \delta$$

Note that in definition 4.1.3 we required prefect confidence $\delta = 0$ (i.e., with probability 1) to have a certain accuracy ($0 < \varepsilon < 1$), while in definition 4.1.4 we require a certain confidence $0 < \delta < 1$ to have a prefect accuracy $\varepsilon = 0$. In addition, although it is named "confidence", δ actually means *lack of confidence* since the greater it is, the less sure we can be that our loss is limited.

Is there a confidence parameter $\delta \in (0,1)$, for which we can find a probably correct learner? Here too, the answer is no. For any δ, Nature always has a strategy that with probability greater than $1-\delta$ will cause the rule to have a non-vanishing loss: $L_{\mathcal{D},f}(h_S) > 0$.

Recall that zero loss means that, with probability 1 with respect to $\mathcal{D}$, the predicted label is always correct. Nature can play a $\mathcal{D}$ that gives a finite but tiny probability to a certain $\mathbf{x} \in \mathcal{X}$. Then, with high probability, the training sample will not include $\mathbf{x}$. So, whatever label the learner assigns to $\mathbf{x}$, it will be a guess and therefore might be incorrect, causing a non-zero loss. Again, we shall prove this assertion through an example of such a strategy that Nature may adopt.

■ **Example 4.1** Let $\delta \in (0,1)$ and consider any two points $\mathbf{x}, \mathbf{x}' \in \mathcal{X}$. Similar to 4.1.2, we consider the training sample $S' = (\mathbf{x}_1, y_1), ..., (\mathbf{x}_m, y_m)$ with $\mathbf{x}_i = \mathbf{x}'$ and $y_i = f(\mathbf{x}')$ for $i = 1, ..., m$, and assume that once this sample is fed into our algorithm, it will predict $h_{S'}(\mathbf{x}) = +1$.

As before, Nature's strategy is to chooses a labeling function f with $f(\mathbf{x}) = -h_{S'}(\mathbf{x}) = -1$ and a distribution, $\mathcal{D}$, with $\mathcal{D}(\mathbf{x}) = \gamma$, $\mathcal{D}(\mathbf{x}') = 1 - \gamma$. This time however, γ is chosen such that $\delta < (1-\gamma)^m$ and $\gamma > 0$. This means that when m is large, the probability of getting $\mathbf{x}$, namely, γ, is chosen to be very small. Therefore in the present case S' is a typical sample, in the sense that its probability to appear, $(1-\gamma)^m$, is greater than δ.

The same calculation as in example 4.1.2 yields $L_{\mathcal{D},f}(h_{S'}) \geq \gamma$ and since $\gamma > 0$ that would mean a non-zero loss. Since the probability of obtaining S' is greater than δ, the probability of ending up with a non-zero loss is also greater than δ. In other words, our learner $\mathcal{A}$ is not a probably correct one. ■

We conclude that for any confidence parameter $\delta \in (0,1)$, no learner $\mathcal{A}$ can guarantee, that with probability of at least $1-\delta$, the loss will vanish: Nature can always find a strategy for which $L_{\mathcal{D},f}(h_S) > 0$ will have a probability greater than δ to occur. Even a "typical" training sample may miss small areas in $\mathcal{X}$. On these areas the resulting h_S might be wrong.

Examples 4.1.2 and 4.1 teach us that no matter what learner we construct, we can never have an absolute confidence that our loss will be limited, neither we can have a limited-confidence that we can obtain an absolute accuracy. What we *might* be able is to have *some* confidence that our loss will not exceed *some* threshold. This brings us to the following definition.

Definition 4.1.5 Let $\delta, \varepsilon \in (0,1)$. We say that a learner, $\mathcal{A}$, is *Probably Approximately Correct* with a confidence δ and an accuracy ε if and only if the probability of obtaining a training sample S, for which $\mathcal{A}$ will output a prediction rule h_S with a loss that does not exceeds ε, is greater than or equal to $1-\delta$:

$$\mathcal{D}^m\left(S \text{ s.t. } L_{\mathcal{D},f}(h_S) \leq \varepsilon\right) \geq 1 - \delta$$

It is important to understand the difference between the *accuracy* ε and the *confidence* δ.

- Recall that we first draw the training sample S at random. Then, the learning algorithm runs on this random sample and its prediction is therefore random. If S is by chance "weird" (not representing $\mathcal{D}$ well), the rule h_S produced will be "wrong", namely, it will not generalize well. The number δ is the probability of failure due to a "weird" sample S.
- After the learner outputs a rule h_S, it is then tested on a new sample. The new sample is also random. $L_{\mathcal{D},f}(h_S)$ is the expected fraction of errors h_S will make, i.e., its accuracy, over such data. The number ε refers to that accuracy.

4.1.3 Learning As A Game - Second Attempt

Since we can only hope to build a *Probably Approximately* correct learner, we will update the game definition. The sample size m will no longer be fixed. Instead, the accuracy ε and confidence δ, which our learner is required to achieve, are specified as game parameters. We get to decide on m as a part of our strategy. Note that there is a subtle nuance in notation now. When we write $\mathcal{A}$ for the learner, we actually mean a *sequence* of learners - one for each sample size m. It would be better to write $\mathcal{A}_m : (\mathcal{X} \times \mathcal{Y})^m \to \mathcal{Y}^{\mathcal{X}}$. However, to keep the notation simple we will continue writing $\mathcal{A}$, and understand that there is in fact a dependence also on m.

Definition 4.1.6 — The Learning Game (second version).
Fix the desired accuracy and confidence parameters $\varepsilon, \delta \in (0,1)$ and then:

- We choose a sample size m and a learner $\mathcal{A}$, both of which may depend on (ε, δ).
- Nature knows our strategy, and, after us, chooses a strategy that consists of a probability distribution

$\mathcal{D}$ over $\mathcal{X}$, and a label function $f : \mathcal{X} \to \mathcal{Y}$. Nature's strategy may too depend on the specified (ε, δ) and also on our choice of m and $\mathcal{A}$.

- A sample S of size m is drawn according to $\mathcal{D}$ and is labeled according to f.
- The sample S is fed into $\mathcal{A}$ to produce a prediction rule h_S.
- The payoff is $L_{\mathcal{D},f}(h_S)$. It is random since S is random and therefore h_S is random.
- Nature does her best to win. So we will look for learners $\mathcal{A}$ for which there is a probability of at least $1-\delta$ that the loss $L_{\mathcal{D},f}(h_S)$ will not exceed ε, no matter what strategy $\mathcal{D}, f$ Nature might play.
- To determine if we were successful in the game, we play the game many many times. Each time both us and Nature play the same strategies. Each time however, the training samples drawn are different. Calculate the probability over the random draws of training samples S, of the event $\left\{S \sim \mathcal{D}^m \,\middle|\, L_{\mathcal{D},f}(h_S) \leq \varepsilon\right\}$. If this probability is found to be larger than $1-\delta$, that is, if the learner $\mathcal{A}$ we chose was Probably Approximately correct with accuracy ε and confidence δ - against Nature's best strategy - we say that *we have been successful (with regards to the parameters ε, δ).*

The game perspective, although phrased differently, is equivalent to the more standard description of our learning challenge, which is the following: The learner doesn't know $\mathcal{D}$ and f. The learner receives an accuracy parameter ε and a confidence parameter δ. It then asks for training data, S, containing $m(\varepsilon, \delta)$ examples (that is, the number of examples can depend on the value of ε and δ, but it can not depend on the unknown $\mathcal{D}$ or f). Finally, the learner should output a hypothesis h_S, that depends only on ε, δ and the training sample S drawn, such that with probability of at least $1-\delta$ it holds that $L_{\mathcal{D},f}(h_S) \leq \varepsilon$. That is, the learner should be Probably Approximately correct, with the specified accuracy ε and confidence δ.

4.2 No Free Lunch and Hypothesis Classes

It turns out that, unfortunately, we cannot, in general, be successful against Nature even in the second version of our game. With no restrictions placed on the choice of $\mathcal{D}$ or f, we can not be confident-enough that we can find an accurate-enough h_S. This is true no matter how large is our sample size m.

In examples 4.1.2 and 4.1 above, Nature used $\mathcal{D}$ that vanishes everywhere in the sample space $\mathcal{X}$ except for two points. In particular, Nature's strategies were enough to prevent us from constructing a learner that enjoys confidence $\delta = 0$ or accuracy $\varepsilon = 0$ for any $\mathcal{X}$ with two or more points. In the following example, Nature's strategy can be applied only if $\mathcal{X}$ has an infinite number of points (though it is enough to have a countable infinite amount of them).

■ **Example 4.2** Suppose that $|\mathcal{X}| = \infty$ and follow the steps of the second version of the game for some $\delta \in (0,1)$ and some $0 < \varepsilon < \frac{1}{2}$.

- We fix m.
- We now decide what label to predict on a point that does not appear in the training sample, S. We could, for example, decide that whenever a point, say, $\mathbf{x}_4$, does not appear in S but $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$ appear all with the label +1, then we take $h_S(\mathbf{x}_4) = 1$. But if these 3 points all show up with the label -1, then we take $h_S(\mathbf{x}_4) = -1$. That would mean that the value we choose for $h_S(\mathbf{x}_4)$ would depend on S. However, we will restrict ourselves to choosing the *same* value, $h_S(\mathbf{x}_4)$, whenever S does not contain $\mathbf{x}_4$, independently of which specific S it is. In other words, we choose a function, $g(\mathbf{x})$, and if a point $\mathbf{x} \in \mathcal{X}$ is *not* observed in S, then at this point, the rule, $h_S(\mathbf{x})$ that $\mathcal{A}$ outputs, will satisfy $h_S(\mathbf{x}) = g(\mathbf{x})$, independently of the specific S we got.
- Nature knows m, so it picks some finite set $C \subset \mathcal{X}$ with $|C| > 2m$, and chooses $\mathcal{D}$ to vanish outside C while being uniform over it: for any $\mathbf{x} \in C$, $\mathcal{D}(\mathbf{x}) = \frac{1}{|C|} < \frac{1}{2m}$.
- Nature also knows $g(\mathbf{x})$, so as a labeling function f Nature plays a sinister move and chooses $f(\mathbf{x}) = -g(\mathbf{x})$ for all $\mathbf{x} \in \mathcal{X}$, namely, just the opposite of what h_s will predict on unseen points.
- Now, let S be a training sample. Let $supp(S) \subset C$ denote the set of different points $\mathbf{x} \in \mathcal{X}$, that appears in S. Recall that the same $\mathbf{x}$ may appear in S several times and therefore the $|supp(S)|$ can take any value between 1 and m. Since $|supp(S)| \leq m$ and since $\mathcal{D}$ is uniform over C, we have

$\mathcal{D}(\{\mathbf{x} \in \mathcal{X} \setminus supp(S)\}) \geq 1/2$. In other words, the probability that a new test point drawn according to $\mathcal{D}$ was not seen in S is at least $1/2$.

- Now, as the game requires, we feed the sample S into $\mathcal{A}$ and obtain $h_S = \mathcal{A}(S)$. What is the loss? Regardless of what h_S predicts on points seen in S, a test point has probability $\geq 1/2$ to be in the unseen part $\mathcal{X} \setminus supp(S)$. Thus, with probability of at least $1/2$, the rule h_S will predict $h_S(\mathbf{x}) = g(\mathbf{x})$ and will be wrong, since $f(\mathbf{x}) = -g(\mathbf{x})$. Therefore, $L_{\mathcal{D},f}(h_S) \geq 1/2$.
- But this happens for *every* training sample S. So, Nature's strategy ensures that with probability 1 (over the choice of training samples according to $\mathcal{D}$), the game results in a loss $L_{\mathcal{D},f}(h_S) \geq 1/2$.
- So, we can not find a learner $\mathcal{A}$ that will be Probably Approximately correct (for any δ and for $\varepsilon < \frac{1}{2}$) regardless of Nature's strategy $\mathcal{D}, f$. Asking for a larger training sample won't help - if we increase m, Nature will just choose a larger set C and a distribution $\mathcal{D}$ which is uniform over that larger C.

■

What went wrong? Nature could choose *any* labeling function, f, that she wanted, and we tried to learn (to generalize/predict) f from a sample that was too small compared to the number of possible functions Nature could choose from. We find that we just cannot design a Probably Approximately correct learner if the set of possible labeling functions is "too large".

This is known as *"No Free Lunch" Theorem*[1]: without assuming anything in advance on the label function, f, learning is impossible. Equivalently, if the set of possible labeling function f is too large, then Nature can play a function that we can't learn. Even if we take a larger sample, Nature, in turn, chooses a more complicated f. So if no limitations are placed on Nature's choice of label function, learning is impossible.

Example 4.2 demonstrated the essence of the No Free Lunch Theorem, but was not a proof of it, since we restricted ourselves to $g(\mathbf{x})$ that was independent of S. That restriction made life easier for Nature since it enabled her to choose $f(\mathbf{x}) = -g(\mathbf{x})$ as a label function that maximized the error of our learner. In reality, we could choose different $g_S(\mathbf{x})$'s for each of the different S's that did not contain $\mathbf{x}$, thus limiting Nature's ability to maximize the loss. In any case, example 4.2 gives the crucial bits of intuition which we will need later on.

As mentioned above, there are several theorems that can be called a "No Free Lunch" - basically any theorem that shows that without some prior knowledge on the labeling function, that is, when there are too many possibilities for f, learning it is impossible. Below is an actual, formal, No Free Lunch theorem. Its proof, which uses the notion of Agnostic PAC that we will encounter a bit later, can be found in the book "Understanding Machine Learning" (Theorem 5.1, and exercise 3 in section 5.5 in that book).

Theorem 4.2.1 — No Free Lunch. Let $\mathcal{X}$ be a sample domain, $|\mathcal{X}| = \infty$. For any $\varepsilon \in (0, \frac{1}{2})$, there exists $\delta \in (0,1)$, such that for any $\mathcal{A}$ there exists $\mathcal{D}, f$ for which:

- For any $m \in \mathbb{N}$, when running $\mathcal{A}$ over $S \sim \mathcal{D}$ of size m
- then $\mathcal{D}\left(S | L_{\mathcal{D},f}(h_S) \geq \varepsilon\right) \geq \delta$

Note that $L_{\mathcal{D},f} = 1/2$ is the loss that one gets from a completely *random guess* of labels. Thus, the condition $\varepsilon < 1/2$ in the above theorem implies that without prior assumptions on f we can not guarantee a prediction which will have a lesser loss than a random guess.

Needing Hypothesis Classes

The No Free Lunch principle implies that to be able to learn, the learner *must* receive enough prior knowledge about the function f. In other words, we should assume that the target f comes from some *hypothesis class*, $\mathcal{H} \subset \mathcal{Y}^{\mathcal{X}}$, or at least that it resembles closely some functions in that class. The class of functions $\mathcal{H}$ should not be too broad or else the problem we encountered in example 4.2 might reappear.

[1] There are in fact several "No Free Lunch" theorems revolving around the same idea

Realizability Assumption

Suppose that a hypothesis class $\mathcal{H}$ is specified for our game, meaning that we restrict the choice of rules that our algorithm can predict to a certain family of functions. The *realizable case* is when Nature must play a function $f \in \mathcal{H}$. Actually, we don't care if Nature plays a function f that is not in $\mathcal{H}$ as long as there is a function h^* in $\mathcal{H}$ which is identical to f on all points over which $\mathcal{D}$ does not vanish so that we will never see examples where $f(\mathbf{x}) \neq h^*(\mathbf{x})$, neither in the training nor in the test samples. So the formal mathematical *Realizability Assumption* is this: Nature plays a function f such that there exists $h^* \in \mathcal{H}$ with $L_{\mathcal{D},f}(h^*) = 0$.

The learner is given $\mathcal{H}$ before the learning starts and will only output $h_S \in \mathcal{H}$. In other words, for a training sample of size m the learner (learning algorithm) is a map $\mathcal{A} : (\mathcal{X} \times \mathcal{Y})^m \to \mathcal{H}$ such that $S \mapsto h \in \mathcal{H}$. As before, we will continue to abuse notation and write $\mathcal{A}$ instead of $\mathcal{A}_m$, and when we say "the learning algorithm $\mathcal{A}$" we will sometimes mean "a sequence of learners $\{\mathcal{A}_m\}_{m=1}^{\infty}$, one for each possible sample size".

Theorem 4.2.1 told us that if $|\mathcal{X}| = \infty$ then $\mathcal{H} = \mathcal{Y}^{\mathcal{X}}$ is too large to learn. This brings us to ask the following questions:

- What are the "small enough" hypothesis classes $\mathcal{H}$ for which we *can* find a Probably Approximately correct learner? And what are the "too large" hypothesis classes $\mathcal{H}$ for which we *cannot*?
- Assume we have a "small enough" $\mathcal{H}$, in the sense that for every ε, δ we have at least one strategy $(m, \mathcal{A})$ such that $\mathcal{A}$ is Probably Approximately correct with accuracy ε and confidence δ, no matter how Nature plays. This means that for every ε, δ there is a *minimal number of training samples*, which we will denote by $m_{\mathcal{H}}(\varepsilon, \delta)$, for which there exists at least one algorithm with the required accuracy and confidence. Can we find this minimal function $m_{\mathcal{H}}(\varepsilon, \delta)$? Is there a relation between the "size" of the hypothesis class $\mathcal{H}$ and $m_{\mathcal{H}}(\varepsilon, \delta)$?
- Assume we have a "small enough" $\mathcal{H}$. Can we find a concrete learner $\mathcal{A}$ that always succeeds in learning functions from $\mathcal{H}$? If yes, how many training samples m does $\mathcal{A}$ need to always succeed in learning a function from $\mathcal{H}$ (always be Probably Approximately correct, no matter how Nature plays)?
- Can we find the *most training-data efficient* learner, namely a learner that can succeed with the minimal number of samples $m_{\mathcal{H}}(\varepsilon, \delta)$ mentioned above?

4.2.1 Learning As A Game - Third Version

To reach a final version of our learning game, we now include the hypothesis class to it.

Definition 4.2.1 — The Learning Game (third version).

Fix desired accuracy $\varepsilon \in (0,1)$ and confidence $\delta \in (0,1)$. Fix a hypothesis class $\mathcal{H} \subset \mathcal{Y}^{\mathcal{X}}$.

- We choose a sample size m and a learner $\mathcal{A} : (\mathcal{X}, \mathcal{Y})^m \to \mathcal{H}$. Both m and $\mathcal{A}$ can depend on (ε, δ).
- Nature knows our strategy and, after us, chooses strategy that consists of a probability distribution $\mathcal{D}$ over $\mathcal{X}$, and a label function *from the hypothesis class*, $f \in \mathcal{H}$. That is, Nature's strategy depends not only on ε, δ, m and $\mathcal{A}$ but also on the hypothesis class, $\mathcal{H}$. (This rule of the game is somewhat stricter than necessary since, as mentioned above, we could allow Nature to choose a function $f \notin \mathcal{H}$ as long as there exists $h^* \in \mathcal{H}$ with $L_{\mathcal{D},f}(h^*) = 0$).
- A sample S of size m is drawn according to $\mathcal{D}$ and is labeled according to f.
- The sample S is fed into $\mathcal{A}$ to produce a prediction rule $h_S = \mathcal{A}(S)$. Note that $h_S \in \mathcal{H}$.
- The payoff is $L_{\mathcal{D},f}(h_S)$. It is random since S is random and therefore h_S is random.
- Nature does her best to win. So we will look for learners $\mathcal{A}$ for which there is a probability of at least $1 - \delta$ that the loss $L_{\mathcal{D},f}(h_S)$ will not exceed ε, no matter what strategy $\mathcal{D}, f$ Nature might play.
- To determine if we were successful in the game, we play the game many many times (both us and Nature play the same strategies, just the training samples drawn are different). We count and calculate the probability, over the random draws of training samples S, of the event $\{S \sim \mathcal{D}^m \,|\, L_{\mathcal{D},f}(h_S) \leq \varepsilon\}$. If this probability is found to be greater than $1 - \delta$, that is, if the learner $\mathcal{A}$ we chose was Probably Approximately correct with accuracy ε and confidence δ - against Nature's best strategy - we say that *we have been successful (with regards to the parameters ε, δ and hypothesis*

class $\mathcal{H}$).

4.2.2 Example: Threshold Functions

We saw above that if $|\mathcal{X}| = \infty$ and $\mathcal{H}$ is the class of all functions from $\mathcal{X}$ to $\mathcal{Y}$, namely $\mathcal{H} = \mathcal{Y}^{\mathcal{X}}$, we can't be successful in the third version of the learning game. On the other hand, we know that if we take a very small $\mathcal{H}$, for example, one that contains only a single function, learning is possible and is in fact trivial. So we know that when the hypothesis class is "small enough", it *is* possible to be successful in the third version of the learning game even when the sample space is infinite, which is the reason why learning is possible. What is left to find out is how broad $\mathcal{H}$ can be chosen while maintaining the possibility to learn.

In the following example we will have an infinite (in fact uncountably infinite) sample space, and a very simple hypothesis class. We will see that we can be successful in the third version of the game, for any ε, δ. Consider the domain $\mathcal{X} = \mathbb{R}$, i.e., there is only one feature. We still consider a classification problem but use the label set $\mathcal{Y} = \{0,1\}$ instead of $\{\pm 1\}$, in order to be able to use the standard definition of Threshold Functions.

Definition 4.2.2 The set of function:

$$\mathcal{H}_{th} = \{x \mapsto h_\theta(x) : \theta \in \mathbb{R}\}$$

where $h_\theta(x) = \mathbb{1}_{x>\theta}$ and $h_\infty(x) = 0, h_{-\infty}(x) = 1$ for all $x \in \mathbb{R}$, is called the Hypothesis Class of *Threshold Functions* over $\mathbb{R}$.

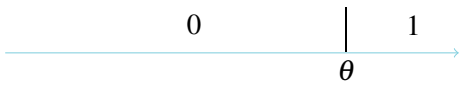


Figure 4.1: *An example of a threshold function*

For the hypothesis class $\mathcal{H}_{th}$ our learning game takes the following form. Functions in $\mathcal{H}_{th}$ classify points on the real line as 0 up to a certain threshold point. Beyond that threshold, they classify all points as 1. Nature chooses one of these functions, that is, it chooses a threshold θ (which is unknown to us) and a distribution $\mathcal{D}$ over the real line. We receive a training sample S of labeled points, and would like to successfully predict the label of future samples. Our job is therefore to determine, as accurately as possible, the unknown cutoff θ.

Now that we were given a hypothesis class, $\mathcal{H}_{th}$, we should specify our strategy, which consists of the number of samples m we need and a learning algorithm $\mathcal{A}$ that will process a training sample and produce a decision rule. As before let $S = \{(x_i, y_i)\}_{i=1}^m$ be the training set. As we will see, our choice of learner will not depend on ε or δ but our choice of m will certainly depend on them.

Learning Algorithm for Threshold Functions

The training data may take the form shown in Figure 4.2 below, but not the form shown in Figure 4.3, which is forbidden because it does not obey the Realizability Assumption (the assumption that Nature chooses $h \in \mathcal{H}$). One possible algorithm, which follows the ERM principle, is the following:

Algorithm 4 Find Threshold Function

Return hypothesis $h_{\theta_{alg}}(x)$ where

- If $y_i = 1$ for all $i = 1, \ldots, m$, then $\theta_{alg} = -\infty$, (the rule classifying all points as 1)
- If $y_i = 0$ for all $i = 1, \ldots, m$, then $\theta_{alg} = +\infty$, (the rule classifying all points as 0)
- In all other cases $\theta_{alg} = \max_i \{x_i : y_i = 0\}$

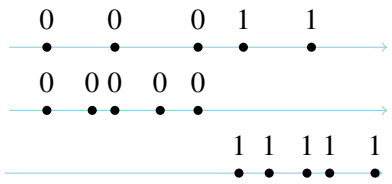


Figure 4.2: *Valid training data for $\mathcal{H}_{th}$*

Figure 4.3: *Invalid training data for $\mathcal{H}_{th}$ - violates the Realizability Assumption*

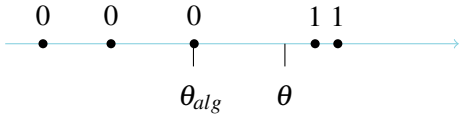


Figure 4.4: *Algorithm for predicting threshold functions*

Number of Samples

Given $\varepsilon, \delta \in (0,1)$ we would like to know what is the sample size m that guarantees that with probability at least $1-\delta$, the true error is at most ε.

Claim 4.2.2 Let $m \geq \log(1/\delta)/\varepsilon$ then for every $\varepsilon, \delta \in (0,1)$, for every $\mathcal{D}$ over $\mathcal{X}$ and for every labeling function $f_\theta \in \mathcal{H}_{th}$, when running Algorithm 4 over m i.i.d samples generated by $\mathcal{D}$ and labeled by f_θ, the algorithm outputs a hypothesis $h_{\theta_{alg}} = \mathcal{A}(S)$ such that:

$$\mathcal{D}^m \left\{ S | L_{\mathcal{D},f_\theta}\left(h_{\theta_{alg}}\right) \leq \varepsilon \right\} \geq 1-\delta$$

Proof. Let $\mathcal{D}$ over $\mathcal{X}$ be some probability distribution and $f_\theta \in \mathcal{H}_{th}$ be some labeling function. From the properties of the algorithm it follows that $\theta_{alg} \leq \theta$. Therefore, the prediction rule produced by the algorithm will only misclassify samples $x \in \mathbb{R}$ such that $\theta_{alg} < x \leq \theta$. Thus, the algorithm's generalization error is given by:

$$L_{\mathcal{D},f_\theta}(h_{\theta_{alg}}) = \mathcal{D}\left(\left(\theta_{alg}, \theta\right]\right)$$

where $(a,b] \coloneqq \{x \,|\, a < x \leq b, a, b \in \mathbb{R}\}$. Observe that if $\mathcal{D}((-\infty,\theta]) < \varepsilon$ then $\mathcal{D}\left(\left(\theta_{alg},\theta\right]\right) < \varepsilon$, meaning that the generalization error is always (that is, with probability 1, regardless to S and m) smaller than ε as required. If $\mathcal{D}((-\infty,\theta]) \geq \varepsilon$ then there exists $\theta' \leq \theta$ such that $\mathcal{D}((\theta',\theta]) = \varepsilon$.

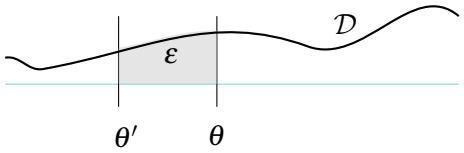


Figure 4.5: *Domain space are for which algorithm will misclassify*

For such θ', what is the relation between θ' and θ_{alg}?

- If $\theta' \leq \theta_{alg} \leq \theta$ it means that there exists a sample $(x_i, y_i) \in S$ with $\theta' < x_i \leq \theta$ for which $y_i = 0$ (because $x_i \leq \theta$). In such case $L_{\mathcal{D},f_\theta}(h_{\theta_{alg}}) = \mathcal{D}(\left(\theta_{alg}, \theta\right]) \leq \mathcal{D}((\theta', \theta]) = \varepsilon$ as required.
- If however, $\theta_{alg} \leq \theta' \leq \theta$ it means that non of the samples in S reside in (θ', θ). This is the case for which the generalization error will exceed ε, and whose probability we wish to bound by δ. What is the probability of such event? That is, of *not* getting any such a sample in the m-sized training set?

$$\mathcal{D}\left(x \middle| x \notin \left(\theta', \theta\right]\right) = 1 - \mathcal{D}\left(\left(\theta', \theta\right]\right) = 1 - \varepsilon \quad \Longrightarrow \quad \mathcal{D}\left(\forall x \in S \middle| x \notin \left(\theta', \theta\right]\right) = (1-\varepsilon)^m$$

We therefore wish that $(1-\varepsilon)^m \leq \delta$. By solving for m, and since $1 - \varepsilon \leq e^{-\varepsilon}$ we obrain that

$$(1-\varepsilon)^m \leq e^{-m\varepsilon} \leq \delta \quad \Longrightarrow \quad m \geq \frac{\log(1/\delta)}{\varepsilon}$$

Thus, for any ε, δ, any $\mathcal{D}$ over $\mathcal{X}$, and any $f_\theta \in \mathcal{H}_{th}$, if running Algorithm 4 over $m \geq \log(1/\delta)/\varepsilon$ i.i.d samples generated by $\mathcal{D}$ and labeled by f_θ, the algorithm outputs a hypothesis $h_{\theta_{alg}} = \mathcal{A}(S)$ such that:

$$\mathcal{D}^m\left\{S \middle| L_{\mathcal{D},f_\theta}\left(h_{\theta_{alg}}\right) \leq \varepsilon\right\} \geq 1 - \delta$$

■

Threshold Functions - Conclusion

We saw that, for $\mathcal{X} = \mathbb{R}$, $\mathcal{Y} = \{0,1\}$ and $\mathcal{H} = \mathcal{H}_{th}$, we have a strategy (a choice of sample size $m = m_{\mathcal{H}_{th}}(\varepsilon, \delta)$ and a learning algorithm $\mathcal{A}$) that is *always successful against Nature*, for any values ε, δ specified.

Recall that for $\mathcal{X} = \mathbb{R}$, $\mathcal{Y} = \{0,1\}$ and $\mathcal{H} = \{h |\ h : \mathbb{R} \to \mathbb{R}\}$ there were values of ε, δ for which we *could not be successful* regardless of how we played. In fact, we could not be successful for any $\varepsilon < 1/2$. So whether we can be successful for any ε and δ seems to be a property of the hypothesis class we choose. This leads us to the famous definition of a *Probably Approximately Correct (PAC)* learnable hypothesis class.

4.3 PAC Learning

Definition 4.3.1 A hypothesis class, $\mathcal{H}$, is *PAC Learnable* if there exists a learning algorithm $\mathcal{A}$ and a function $m_{\mathcal{H},\mathcal{A}} : (0,1)^2 \to \mathbb{N}$ with the following property that:

- For every $\varepsilon, \delta \in (0,1)$
- For every distribution $\mathcal{D}$ over $\mathcal{X}$
- For every labeling function $f : \mathcal{X} \to \{\pm 1\}$ such that there exists $h^* \in \mathcal{H}$ that satisfies $L_{\mathcal{D},f}(h^*) = 0$

when running the learning algorithm $\mathcal{A}$ on $m \geq m_{\mathcal{H},\mathcal{A}}(\varepsilon, \delta)$ i.i.d samples generated by $\mathcal{D}$ and labeled by f, the algorithm returns a hypothesis $h_S = \mathcal{A}(S)$ such that, with probability of at least $1 - \delta$ (over the choice

of the training samples), we have $L_{\mathcal{D},f}(h_S) \leq \varepsilon$:

$$\mathcal{D}^m\left(\left\{S \middle| L_{\mathcal{D},f}(h_S) \leq \varepsilon\right\}\right) \geq 1-\delta$$

Denote the minimal sample size required for the above definition to hold with respect to ε, δ and with respect to any algorithm, by

$$m_{\mathcal{H}}(\varepsilon,\delta) = \min_{\mathcal{A}} m_{\mathcal{H},\mathcal{A}}(\varepsilon,\delta)$$

The function $m_{\mathcal{H}} : (0,1)^2 \to \mathbb{N}$ is called the *Sample Complexity* of the PAC learnable hypothesis class $\mathcal{H}$.

Note that PAC learnability is only one possible definition of learning that can account for the fundamental limitations on accuracy and confidence. We could, for example, settle for a specific, "good enough", values of δ and ε, instead of requiring that the above condition holds for *any* $\varepsilon, \delta \in (0,1)$. Such *weak learners* will be discussed in later chapters.

4.3.1 PAC Learnability of Finite Hypothesis Classes

To better understand when a hypothesis class is PAC learnable, let us first consider the case where $\mathcal{H}$ is a *finite* hypothesis class. Though finite, this hypothesis class is still very large. For example, consider $\mathcal{H}$ as the set all the functions from $\mathcal{X}$ to $\mathcal{Y}$ that can be implemented using a Python program of length at most b, for b fixed and large. Or, take $\mathcal{H}$ to be all the functions from $\mathcal{X}$ to $\mathcal{Y}$ where $|\mathcal{X}|$ and $|\mathcal{Y}|$ are finite.

One might expect that there would not be much to say in this generality, without specific details of $\mathcal{X}$ and $\mathcal{H}$. Surprisingly, it turns out that there is a simple type of learner, called *Empirical Risk Minimization*, that is always successful on finite hypothesis classes and in fact on many other hypothesis classes that we will encounter later. The idea behind these powerful learners is very natural: try to be as correct as possible on the training data. In other words, find the function h in $\mathcal{H}$ that gives the correct label to the maximal number of points in the training sample, S.

Definition 4.3.2 For $h \in \mathcal{H}$ and $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ we define the *empirical risk* by

$$L_S(h) = \frac{1}{m}\left|\{i : h(\mathbf{x}_i) \neq y_i\}\right|$$

A rule, h, with $y_i = h(\mathbf{x}_i)$ for $i = 1,\ldots,m$, that is, with $L_S(h) = 0$, is called *Consistent* with the training sample, S.

Definition 4.3.3 A learning algorithm $\mathcal{A}$ is called an *Empirical Risk Minimization (ERM)* learner, and is denoted by $ERM_{\mathcal{H}}$, if and only if for every S it outputs a prediction rule h_S that minimizes the empirical risk $L_S(h)$:

$$ERM_{\mathcal{H}} : S \mapsto \operatorname*{argmin}_{h\in\mathcal{H}} L_S(h)$$

Notice that since $L_S(h) \geq 0$ and we are minimizing over a finite class, a minimum exists. For each given S, the minimum may be obtained by more than one h in $\mathcal{H}$, in which case $ERM_{\mathcal{H}}$ actually refers to a *set of algorithms*, each of which returns one of the minimizers. *Under the realizability assumption* we know that the lower bound $L_S(f) = 0$ is achievable. In other words, under our assumption that $f \in \mathcal{H}$, an ERM learner will always return a consistent rule. Hence, in the realizable case, the terms "consistent rule" or "ERM rule" are synonyms.

Learning Finite Classes

Theorem 4.3.1 Let $\varepsilon, \delta \in (0,1)$ and $|\mathcal{H}| < \infty$. $\mathcal{H}$ is PAC learnable with $\mathcal{A}$ an ERM learner and sample complexity $m_{\mathcal{H}}(\varepsilon, \delta) = \frac{\log(|\mathcal{H}|/\delta)}{\varepsilon}$

Proof. Let $\varepsilon \in (0,1)$, $\mathcal{D}$ a probability distribution over $\mathcal{X}$ and a labeling function f. By specifying $\varepsilon, \mathcal{D}$ and f we implicitly define a subset of $\mathcal{H}$, which we will denote as $\mathcal{H}_B$ ("B" for "bad"), containing all the "bad" h's. That is, all hypothesis that do not approximate f well:

$$\mathcal{H}_B = \left\{ h \in \mathcal{H} \,\middle|\, L_{\mathcal{D},f}(h) > \varepsilon \right\}$$

By definition

$$\left\{ S \middle| L_{\mathcal{D},f}\left(h_S^{ERM}\right) > \varepsilon \right\} = \left\{ S \middle| h_S^{ERM} \in \mathcal{H}_B \right\}$$

and therefore, to prove theorem 4.3.1, we have to show that

$$\mathcal{D}^m \left(\left\{ S \middle| h_S^{ERM} \in \mathcal{H}_B \right\} \right) < \delta$$

.

Any sample S defines a subset of $\mathcal{H}$, which we will denote as $\mathcal{H}_C^S$ ("C" for "consistent"), containing all h's that are consistent with f on S:

$$\mathcal{H}_c^S = \{h : \, L_S(h) = 0\} = \{h : \, h(\mathbf{x}_i) = f(\mathbf{x}_i), \, i = 1 \ldots, m\}$$

Any sample S, also defines an intersection set $\mathcal{H}_{BC}^S = \mathcal{H}_B \cap \mathcal{H}_C^S$ which contains all h's that are consistent *and* bad (Figure 4.6).

Our algorithm is an ERM learner, $h_S^{ERM} \in \mathcal{H}_C^S$. Therefore all the samples S, in the set $\left\{ S : h_S^{ERM} \in \mathcal{H}_B \right\}$ are such that the intersection set, $\mathcal{H}_{BC}^S$, is not empty since it contains at least one function, namely, h_S^{ERM}. Thus,

$$\left\{ S \middle| h_S^{ERM} \in \mathcal{H}_B \right\} \subseteq \left\{ S \middle| \mathcal{H}_{BC}^S \neq \varnothing \right\}$$

As such it is sufficient to prove that:

$$\mathcal{D}^m \left(\left\{ S : \mathcal{H}_{BC}^S \neq \varnothing \right\} \right) < \delta$$

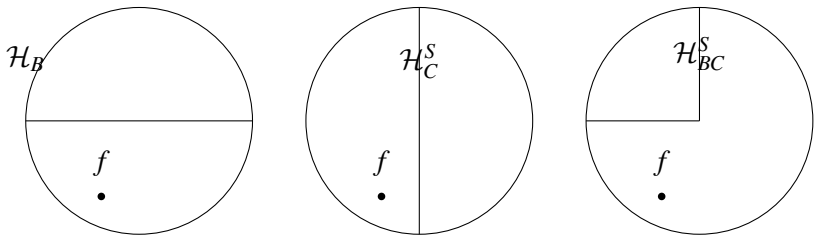


Figure 4.6: *The Bad ($\mathcal{H}_B$) and S-Consistent ($\mathcal{H}_C^S$) subsets of $\mathcal{H}$ and their intersection set ($\mathcal{H}_{BC}^S$)*

Note that

$$\left\{ S \middle| \mathcal{H}_{BC}^S \neq \varnothing \right\} = \bigcup_{h \in \mathcal{H}} \left\{ S \middle| h \in \mathcal{H}_{BC}^S \right\} = \bigcup_{h \in \mathcal{H}_B} \left\{ S \middle| h \in \mathcal{H}_C^S \right\}$$

The first equality is simply a way of saying that instead of checking each S to see if its related intersection set, $\mathcal{H}_{BC}^S$, is non-empty, and then adding it to the above set, we can check each h in $\mathcal{H}$ to see if it appears in any of

the $\mathcal{H}_{BC}^S$'s, and each time it does, we add the S that defined that particular $\mathcal{H}_{BC}^S$ to the set. The second equality results from the definition $\mathcal{H}_{BC}^S = \mathcal{H}_B \cap \mathcal{H}_C^S$. We are therefore left with proving that:

$$\mathcal{D}^m\left(\bigcup_{h\in\mathcal{H}_B}\left\{S\middle|h\in\mathcal{H}_C^S\right\}\right) < \delta$$

Using the union bound we get that

$$\mathcal{D}^m\left(\bigcup_{h\in\mathcal{H}_B}\left\{S\middle|h\in\mathcal{H}_C^S\right\}\right) \leq \sum_{h\in\mathcal{H}_B}\mathcal{D}^m\left(\left\{S\middle|h\in\mathcal{H}_C^S\right\}\right)$$

Given a hypothesis h, $\mathcal{D}^m\left(\left\{S \text{ s.t. } h\in\mathcal{H}_C^S\right\}\right)$ is the probability to pull a sample over which h is perfectly correct (has the right labels for all $\mathbf{x}_i$'s). Since the samples are drawn independently, this probability equals the probability that h will be correct for each $\mathbf{x}_i$ separately. The probability that h will be *incorrect* for a random $\mathbf{x}$ is exactly $L_{\mathcal{D},f}(h)$ and therefore the probability to be correct for m such $\mathbf{x}$'s is $\left(1-L_{\mathcal{D},f}(h)\right)^m$. We therefore have, for any h,

$$\mathcal{D}^m\left(\left\{S\middle|h\in\mathcal{H}_C^S\right\}\right) = \left(1-L_{\mathcal{D},f}(h)\right)^m$$

and in particular, if $h\in\mathcal{H}_B$ then $L_{\mathcal{D},f}(h) > \varepsilon$ which means that

$$\mathcal{D}^m\left(\left\{S\middle|L_S(h)=0\right\}\right) < (1-\varepsilon)^m \quad \forall h\in\mathcal{H}_B$$

Thus we obtain

$$\mathcal{D}^m\left(\bigcup_{h\in\mathcal{H}_B}\left\{S\middle|h\in\mathcal{H}_C^S\right\}\right) \leq \sum_{h\in\mathcal{H}_B}(1-\varepsilon)^m < |\mathcal{H}_B|\cdot(1-\varepsilon)^m \leq |\mathcal{H}|\cdot(1-\varepsilon)^m$$

Finally, using $1-\varepsilon\leq e^{-\varepsilon}$ we conclude:

$$\mathcal{D}^m\left(\left\{S\middle|L_{\mathcal{D},f}(ERM_{\mathcal{H}}(S))>\varepsilon\right\}\right) < |\mathcal{H}|\,e^{-\varepsilon\cdot m}$$

If specifying $m\geq\frac{\log(|\mathcal{H}|/\delta)}{\varepsilon}$, the right-hand side would be smaller than δ which concludes the proof. ■

Theorem 4.3.1 means that, by definitions 4.3.1 and 4.1.5 any *finite* hypothesis class $\mathcal{H}$ is PAC-learnable with a sample complexity not larger than $\log(|\mathcal{H}|/\delta)/\varepsilon$. Moreover, for any m greater or equal to that lower bound, any $ERM_{\mathcal{H}}$ learner is a PAC learner.

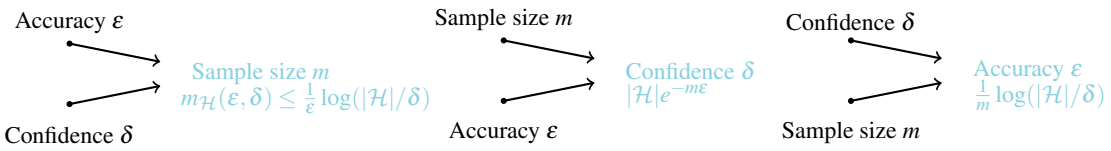


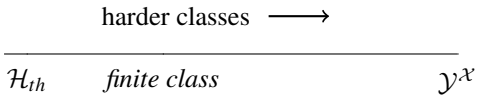
Figure 4.7: *Relation between m, ε and δ*

We have therefore shown that any finite hypothesis class $\mathcal{H}$ is PAC learnable using any ERM learning algorithm, and has a sample complexity $m_{\mathcal{H}}(\varepsilon,\delta)\leq\log(|\mathcal{H}|/\delta)/\varepsilon$. In practical terms, this means that whenever we are given $\varepsilon,\delta\in(0,1)$, a finite hypothesis class $\mathcal{H}$, and a sample of size of at least $\log(|\mathcal{H}|/\delta)/\varepsilon$, we can simply scan $\mathcal{H}$ to find a hypothesis that labels the points in S correctly. With probability of at least $1-\delta$ the generalization error of that hypothesis will not exceed ε.

Several natural questions may now come to mind:

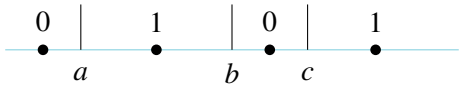
- Is the bound $m_{\mathcal{H}}(\varepsilon,\delta) \leq \frac{\log(|\mathcal{H}|/\delta)}{\varepsilon}$ tight? In other words, can the ERM learner, or an other learner, be Probably Approximately correct (with accuracy ε and confidence δ) using fewer than $\frac{\log(|\mathcal{H}|/\delta)}{\varepsilon}$ samples?
- What happens when noise is present so the y's are not deterministically determined by x?
- What happens when our hypothesis class is infinite?

Consider the last question. As we have seen, the class of threshold functions over $\mathbb{R}$, $\mathcal{H}_{th}$, in spite of being infinite, *is* PAC learnable, with sample complexity $m_{\mathcal{H}_{th}}(\varepsilon,\delta) \leq \frac{\log(1/\delta)}{\varepsilon}$, which is obtained by using an $ERM_{\mathcal{H}_{th}}$ learning rule (recall that the rule output by our algorithm was consistent on any sample). So $\mathcal{H}_{th}$ appears to be simple to learn, even simpler, in terms of the sample complexity, than a finite class. While $\mathcal{Y}^{\mathcal{X}}$ is too complex to learn. Can we explain why? What we need in order to answer the above questions systematically is some sort of a *complexity measure* with which we can order classes by their difficulty along the complexity axis shown in the figure below.



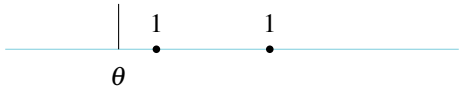
For example, consider the *Two-Intervals* hypothesis class, defined by

$$\mathcal{X} = \mathbb{R}, \quad \mathcal{H} = \left\{ h_{a,b,c} : a < b < c \in \mathbb{R} \right\}, \quad h_{a,b,c} = \mathbb{1}_{x \in [a,b] \vee x \geq c}$$

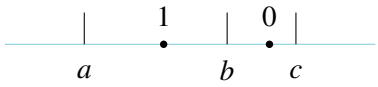


Suppose we would like to learn with the threshold function class, $\mathcal{H}_{th}$, the following sample:

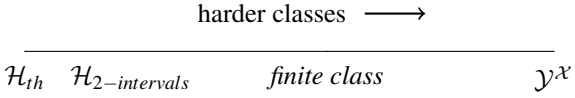
A possible answer would be:



However, samples where $x_1 < x_2$ and $y_1 = 1, y_2 = 0,$, as in the figures below, can not be learned (consistently) by $\mathcal{H}_{th}$, although it can be learned with the 2-intervals class:



Indeed, we somehow feel that the 2-Interval class has a larger complexity than that of $\mathcal{H}_{th}$ but smaller than that of finite classes.



4.3.2 VC Dimension

The VC-Dimension is a *measure of complexity* of hypothesis classes. It is called a *combinatorial* measure because it relies on a certain way of counting the possibilities available in the hypothesis class to label points in $\mathcal{X}$. This measure provides a *precise* test for whether or not a hypothesis class is simple enough to learn in the sense of PAC learnability. It also enables the calculation of clear bounds on the sample complexity of a simple hypothesis class $\mathcal{H}$.

Suppose we receive a training set $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ and were able to fully explain the labels using a hypothesis from a class $\mathcal{H}$, namely, to find a function $h \in \mathcal{H}$ with empirical risk $L_S(h) = 0$. Suppose now, only to see what will happen, we deliberately corrupt our sample S by changing 'by hand' certain labels, and denote the corrupted sample by S'. Suppose that we *also* succeed in explaining S' using a different hypothesis from the same class $\mathcal{H}$, namely, find another function $h' \in \mathcal{H}$ with $L_{S'}(h') = 0$. If we are able to do that, no matter what labels we choose to corrupt and regardless of the sample size, it means that something isn't right. How can we hope to generalize based on a training sample S if, regardless of the labels in S, we can find $h \in \mathcal{H}$ with $L_S(h) = 0$?

Definition 4.3.4 — Restriction. Let $C \subseteq \mathcal{X}$ and $h \in \mathcal{H}$. The *restriction* of h to C is the function $h_C : C \to \mathcal{Y}$, defined as $\forall \mathbf{x} \in C$, $h_C(\mathbf{x}) = h(\mathbf{x})$. The restriction of $\mathcal{H}$ to C is the set $\mathcal{H}_C = \{h_C \,|\, h \in \mathcal{H}\}$.

R Since $\mathcal{Y} = \{\pm 1\}$, we can represent each h_C by the vector $\big(h(\mathbf{x}_1), \ldots, h(\mathbf{x}_{|C|})\big) \in \{\pm 1\}^{|C|}$. The number of possible such vectors is $2^{|C|}$, therefore $|\mathcal{H}_C| \leq 2^{|C|}$.

To further clarify the above point, consider the following important observation. Suppose that $\mathcal{H}$ contains *all* the possible functions over a set $C \subset \mathcal{X}$ of size m, that is, $\mathcal{H}_C = \mathcal{Y}^C$, then there is no Probably Approximately Correct learner that uses $m/2$ or fewer training samples. To understand why, recall example 4.2 that demonstrated the argument behind the No Free Lunch theorem. To play our game against Nature, we choose a learner $\mathcal{A}$ and ask for a training sample size of $m/2$ or less. For points $\mathbf{x} \in \mathcal{X}$ not seen in the training set, we choose to predict $g(\mathbf{x})$, where $g : \mathcal{X} \to \mathcal{Y}$. We only have to make sure that $g(\mathbf{x})$ is such that the resulting h_S will belong to $\mathcal{H}$.

As in example 4.2, Nature plays a distribution $\mathcal{D}$ that is uniform over C and zero elsewhere, and a labeling function f such that $f_C(\mathbf{x}) = -g_C(\mathbf{x})$, that is, $f(\mathbf{x}) = -g(\mathbf{x})$ for any $\mathbf{x} \in C$ (Since $\mathcal{D}$ vanishes outside of C, Nature doesn't really care how her f is defined outside C). Nature can *always* choose such f, since $\mathcal{H}_C = \mathcal{Y}^C$ so in particular $-g_C(\mathbf{x}) \in \mathcal{H}_C$, which means that there is at least one $h \in \mathcal{H}$ with $h_C(\mathbf{x}) = -g_C(\mathbf{x})$. Again, as in example 4.2, our learner fails since the loss will be $1/2$ or more - regardless of the training sample (as long as it is of size $m/2$ or smaller).

As in example 4.2, the argument presented here is not a full proof since we restricted our choice of algorithm only to such that chooses the *same* label $g(\mathbf{x})$, for *all* S's in which $\mathbf{x}$ does not appear. However, the intuition that our argument implies is correct: As long as $\mathcal{H}$ contains *any* set C of size $2m$ with the property that $\mathcal{H}_C = \mathcal{Y}^C$, then we cannot learn with a training sample of size m. It follows that the *maximal size* of such a set C in $\mathcal{H}$ is a *critical quantity*: (i) it gives us a lower bound on $m_{\mathcal{H}}$, the minimal sample size needed, and (ii) if the maximal size is ∞, namely, if for any $m \in \mathbb{N}$, $\mathcal{X}$ contains such as set C with $|C| > m$, $\mathcal{H}$ is not PAC-learnable.

Definition 4.3.5 — Shattering. Let $C = \{\mathbf{x}_1, \ldots, \mathbf{x}_{|C|}\} \subset \mathcal{X}$, $\mathcal{Y} = \{\pm 1\}$ and $\mathcal{H}_C$ be the restriction of $\mathcal{H}$ to C. We say that $\mathcal{H}$ *shatters* C if $|\mathcal{H}_C| = 2^{|C|}$, which is equivalent to saying that $\mathcal{H}_C = \mathcal{Y}^C$.

Definition 4.3.6 — VC-dimension. The *VC-dimension* of the hypothesis class $\mathcal{H}$ is defined as

$$VCdim(\mathcal{H}) = \max\{|C| \ : \ \mathcal{H} \ \text{ shatters } \ C\}$$

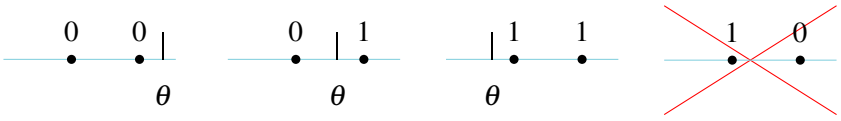
that is, the VC dimension is the maximal size of a set $C \subset \mathcal{X}$ such that $\mathcal{H}_C = \mathcal{Y}^C$.

According to the above definition, in order to show that $VCdim(\mathcal{H}) = d$ we need to show that:

1. There *exists* a set C of size d which is shattered by $\mathcal{H}$.
2. for *any* set C of size greater than d, C is not shattered by $\mathcal{H}$.

■ **Example 4.3 — Threshold class.** Consider the hypothesis class of thereshold function $\mathcal{H}_{th}$. The set $\{0\}$ is shattered by $\mathcal{H}_{th}$ since $x = 0$ can receive both the label 0 and 1, depending on the location of the threshold θ.

In contrast to the above, no two points, x_1 and x_2 can be shattered because if $x_1 < x_2$ then $y_1 \leq y_2$ and therefore the pair of labels $y_1 = 1, y_2 = 0$ is forbidden.



■

■ **Example 4.4 — One-Interval hypothesis class.** Consider the *One-Interval hypothesis class* over $\mathcal{X} = \mathbb{R}$ defined as

$$\mathcal{H} = \left\{h_{a,b} : a < b \in \mathbb{R}\right\}, \quad h_{a,b}(x) = \mathbb{1}_{x \in [a,b]}$$

Now, take for example the two points $\{0,1\}$. We can place the interval over both, over any one of them, or outside $[0,1]$. Therefore, $\{0,1\}$ is shattered. However, any three points cannot be shattered. Let $x_1 < x_2 < x_3$, then no single interval can cover x_1 and x_3 without containing also x_2 and therefore the labeling $y_1 = 1, y_2 = 0, y_3 = 1$ is forbidden. ■

4.3.3 The Fundamental Theorem of Statistical Learning

In the previous sections, we worked hard to understand two fundamental definitions: PAC-Learnability (and sample complexity) of a hypothesis class, and VC-dimension of a hypothesis class.

Along the way, we saw some connections between these two definitions:

- We saw that a finite hypothesis class is PAC-learnable (using the ERM learners) with sample complexity

$$m_{\mathcal{H}}(\varepsilon, \delta) \leq \frac{\log(|\mathcal{H}|) + \log(1/\delta)}{\varepsilon}$$

  and also that in this case $VCdim(\mathcal{H}) \leq \log_2(|\mathcal{H}|)$. These results indicate that there might be a general relation, valid for other hypothesis classes as well, between an *upper bound* on $m_{\mathcal{H}}$ and $VCdim(\mathcal{H})$.
- We saw, using the same argument that led us to the No Free Lunch theorem, that $VCdim(\mathcal{H})$ also gives a *lower bound* on the sample complexity $m_{\mathcal{H}}$ of a hypothesis class $\mathcal{H}$, and that if $VCdim(\mathcal{H})$ is infinite, that class is not PAC-Learnable.

The surprising, wonderful, truth is that VC-dimension gives a complete characterization of PAC-learnability and sample complexity of a hypothesis class, and gives a decisive answer to all the questions we posed at various stages along the way (such as which classes are PAC-learnable, with what sample complexity, and with what algorithm, and is there an algorithm that uses the minimal possible sample size.) These facts result from the *The Fundamental Theorem of Statistical Learning* which states the following:

- The PAC-learnability of a hypothesis class is characterized by its *VC dimension*, a combinatorial property that denotes the maximal size of a sample that can be shattered by the class.
- A hypothesis class is PAC-learnable *if and only if* its VC-dimension is finite.
- When $VCdim(\mathcal{H})$ is finite, $\mathcal{H}$ has a finite sample complexity which is, up to multiplicative constants, given by
$$m_H(\varepsilon,\delta) \sim \frac{VCdim(\mathcal{H}) + \log(1/\delta)}{\varepsilon}$$
- The ERM learning rule is a generic (near) optimal learner, in the sense that when a hypothesis class is PAC-learnable, an ERM learner using
$$m(\varepsilon,\delta) \sim \frac{VCdim(\mathcal{H})\log(1/\varepsilon) + \log(1/\delta)}{\varepsilon}$$
is a Probably Approximately correct learner with accuracy ε and confidence δ.

Theorem 4.3.2 — The Fundamental Theorem of Statistical Learning. Let $\mathcal{H}$ be a hypothesis class of binary classifiers with VC-dimension $d \leq \infty$. Then, $\mathcal{H}$ is PAC-learnable if and only if $d < \infty$. In this case: there are absolute constants C_1, C_2 (that is, they are independent of d, ε and δ) such that the sample complexity of $\mathcal{H}$ satisfies
$$C_1 \frac{d + \log(1/\delta)}{\varepsilon} \leq m_{\mathcal{H}}(\varepsilon,\delta) \leq C_2 \frac{d\log(1/\varepsilon) + \log(1/\delta)}{\varepsilon}$$
Furthermore, the upper bound on sample complexity is achieved by the ERM learner.

As we saw, the intuition behind the lower bound was based on how the VC-dimension $VCdim(\mathcal{H})$ was related to the *minimal* number of training samples needed to PAC-learn the hypothesis class $\mathcal{H}$.

To understand the upper bound, we need to understand how is it that the ERM learner is a *generic learning algorithm* that is able to PAC-learn $\mathcal{H}$ with a training sample size related to the VC-dimension $VCdim(\mathcal{H})$.

Before discussing the upper bound, we shall extend our theoretic framework to make it much more flexible and realistic.

4.4 Agnostic PAC

The theoretical framework we developed so far has several serious limitations when it comes to real-world learning problems.

- *It doesn't model noisy labels*: We have no model for measuring errors in labels. In practice, sometimes even though the label for some $x \in \mathcal{X}$ should have been for example 1, it can be measured as -1 due to measurement mistake, noise, etc. We want the learning framework to allow for the fact that we may, with low probability, observe the point $x \in \mathcal{X}$ twice, and get two different labels.
- *The realizability assumption is unrealistic*: The hypothesis class is, after all, only an intelligent guess, and it is unrealistic to assume that the true label function will belong to it.
- *Limited to misclassification loss*: In the previous sections we measured the performance of a classifier using the misclassification loss (the 0-1 loss). We would like to be able to measure performance using any loss function.

To address these limitations we introduce the *Agnostic PAC* framework. In addition to address these limitations we can also prove that the the fundamental theorem of statistical learning holds in the Agnostic PAC framework as well.

4.4.1 Introducing the Joint Probability Distribution Over $\mathcal{X} \times \mathcal{Y}$

In the PAC framework, $\mathcal{D}$ is a probability distribution over the sample space $\mathcal{X}$ and the labels are determined deterministically using the label function f. In the new, more general framework, $\mathcal{D}$ will be a probability distribution over $\mathcal{X} \times \mathcal{Y}$.

This means that when we draw a new random example $(\mathbf{x}, y)$ - whether for the training sample S or as a test sample - there is randomness in *both* $\mathbf{x}$ and y which are now *dependent* random quantities.

We can factor $\mathcal{D}$ in two ways, conditioning on $\mathbf{x}$ or on y, both are useful for our understanding. Let (X, Y) be a random variable taking valued in $\mathcal{X} \times \mathcal{Y}$ whose distribution is $\mathcal{D}$.

- $\mathbb{P}(X = \mathbf{x}, Y = y) = \mathbb{P}(X = \mathbf{x})\mathbb{P}(Y = y | X = \mathbf{x})$. From this perspective, this is a direct generalization of our previous framework, where $\mathbf{x}$ was random and $y = f(\mathbf{x})$. Indeed, we draw $\mathbf{x}$ from the marginal distribution with probability $\mathbb{P}(X = \mathbf{x})$ - as we did in the previous framework. We then choose a corresponding label according to the conditional probability $\mathbb{P}(Y = y | X = \mathbf{x})$.
  Since the marginal random variable Y describing the label is a Bernoulli random variable, one can think of it as a result of a coin flip of a biased coin, with a probability $p(\mathbf{x})$ to obtain +1, where the function $p : \mathcal{X} \to [0, 1]$ is defined by $\mathbb{P}(Y = +1 | X = \mathbf{x}) = p(\mathbf{x})$. If $p(\mathbf{x}) = 0$ or $p(\mathbf{x}) = 1$ for some $\mathbf{x}$, the label is deterministic and we are back to the label function f. But for other values of $p(\mathbf{x})$, whereas before the label depended deterministically on $\mathbf{x}$, now it is random. This models *measurement noise* - the fact that there may be noise in the labels and that the distribution of the noise may change from one $\mathbf{x}$ to another. This attitude is similar to that of the Logistic Regression classifier, although in that case we did not pay attention to the distribution on $\mathbf{x}$ - instead, we assumed the samples are given and tried to estimate the conditional probability $\mathbb{P}(Y = +1 | X = \mathbf{x}) = p(\mathbf{x})$.
- $\mathbb{P}(X = \mathbf{x}, Y = y) = \mathbb{P}(Y = y)\mathbb{P}(X = \mathbf{x} | Y = y)$. From this perspective, we first draw the label according to a "coin flip" - a Bernoulli random variable. Then, each label has its own distribution for the samples $\mathbf{x}$. We draw the sample $\mathbf{x}$ from $\mathbb{P}(X = \mathbf{x} | Y = +1)$ or from $\mathbb{P}(X = \mathbf{x} | Y = -1)$, according to the label y we obtained. This is a similar approach to the one used in the LDA classifier.

When $\mathcal{D}$ was a distribution over $\mathcal{X}$ alone, we defined the misclassification loss, (4.2), as the probability of obtaining an $\mathbf{x}$ whose *correct* label, $f(\mathbf{x})$, differs from the predicted one, $h(x)$. Now, with $\mathcal{D}$ as a distribution over $\mathcal{X} \times \mathcal{Y}$, we generalize simply by defining the loss as the probability of obtaining an $\mathbf{x}$ whose *measured* label, y, will differ from the predicted one, $h(\mathbf{x})$:

$$L_{\mathcal{D}}(h) = \mathbb{P}_{(\mathbf{x},y)\sim\mathcal{D}}\{h(\mathbf{x}) \neq y\} = \mathcal{D}\{(\mathbf{x}, y) \,|\, h(\mathbf{x}) \neq y\} \tag{4.3}$$

4.4.2 Relaxing Realizability Assumption

Recall that in the case of deterministic labeling, under the realizability assumption, we could reach zero generalization error:

$$\min_{h \in \mathcal{H}} L_{\mathcal{D},f}(h) = L_{\mathcal{D},f}(f) = 0$$

However, according to (4.3), in order to calculate the loss, we now have to compare a deterministic values, $h(\mathbf{x})$, to a random one, the measured y, and therefore we can no longer expect zero loss. In other words, we no longer have a “ground truth” labeling function f, at least not one accessible by measurement. The closest thing we have to f is the conditional probability $\mathbb{P}(Y = y | X = \mathbf{x})$).

So the realizability assumption is no longer practical in the sense that we no longer expect to be able to reach 0 generalization loss. Even if a certain underlying true-label function does exist, and no matter how close our

predicted rule, $h(\mathbf{x})$, resembles it, we may end up with a non-negligible loss due to the noise. This means we have to change also the definition of *accuracy*: we would like the learning algorithm to output a rule which has generalization loss at most ε *above the minimal possible loss* $\min_{h\in\mathcal{H}} L_{\mathcal{D}}(h)$. We now proceed to identify such a lower bound for the loss.

Recall the *Bayes Optimal Classifier* previously defined as $f_{\mathcal{D}}(\mathbf{x}) := \mathbb{1}_{\mathbb{P}(y=1|\mathbf{x})\geq 1/2}$ for $\mathcal{D}$ some a probability distribution over $\mathcal{X}\times\mathcal{Y}$. Note that although $f_{\mathcal{D}}:\mathcal{X}\to\mathcal{Y}$ is a hypothesis, it depends on $\mathcal{D}$, which according to the rules of the game, we don't know. So $f_{\mathcal{D}}$ is what is known as an *Oracle Quantity*: if we had an oracle telling us $\mathcal{D}$, then we could classify with $f_{\mathcal{D}}$. Oracle quantities, like this one, are used to compare the loss of any other rule to the loss of the *best possible* rule.

Definition 4.4.1 Let $\varepsilon > 0$. We say that a rule $h\in\mathcal{H}$ is *Approximately Correct* with *accuracy* ε with respect to the distribution $\mathcal{D}$ on $\mathcal{X}\times\mathcal{Y}$ if $L_{\mathcal{D}}(h)$ is as most ε away from the best possible loss achievable by *any* hypothesis in $\mathcal{H}$:

$$L_{\mathcal{D}}(h) \leq \min_{h'\in\mathcal{H}} L_{\mathcal{D}}(h') + \varepsilon$$

Notice once more that in our previous framework, under the realizability assumption, the minimal loss was simply 0 since we *assumed* the existence of some $h'\in\mathcal{H}$ with $L_{\mathcal{D}}(h') = 0$. Thus, we let go of the realizability assumption: we no longer assume that there exists a "correct" labeling function and in particular no longer assume that Nature plays a labeling function in the chosen hypothesis class $\mathcal{H}$. Nature's strategy consists only of choosing the joint distribution $\mathcal{D}$ over $\mathcal{X}\times\mathcal{Y}$.

4.4.3 General Loss Function

The last extension of the framework developed in the previous sections is to allow the use of a general loss function.

Definition 4.4.2 A *Loss Function* is a function $\ell : \mathcal{H}\times Z\to[0,\infty)$, where $Z = \mathcal{X}\times\mathcal{Y}$.

R Instead of writing $\ell(h,z)$, we shall often write $\ell(h,(\mathbf{x},y))$ or $\ell(h(\mathbf{x}),y)$.

We already used the most common example of a classification loss function - the misclassification loss, also known as the *0-1 loss*:

$$\ell_{0,1}(h,(\mathbf{x},y)) := \begin{cases} 1 & h(\mathbf{x})\neq y \\ 0 & h(\mathbf{x}) = y \end{cases}$$

The definition, (4.3), of the generalization loss we have been using, $L_{\mathcal{D}}(h)$, can be rewritten in terms of $\ell_{0,1}$ as:

$$L_{\mathcal{D}}(h) = \mathbb{E}_{\mathcal{D}}[\ell_{0,1}(h,(\mathbf{x},y))]$$

where $\mathbb{E}_{\mathcal{D}}[\cdot]$ denotes the expected value according to $(\mathbf{x},y)\sim\mathcal{D}$.
Written in its new form, the above definition can be naturally extended to include any loss function:

Definition 4.4.3 — Generalization Loss. Given a distribution $\mathcal{D}$ over $\mathcal{Z} = \mathcal{X}\times\mathcal{Y}$, a hypothesis $h:\mathcal{X}\to\mathcal{Y}$ and a general loss function $\ell:\mathcal{H}\times\mathcal{Z}\to[0,\infty)$, we define the Generalization Loss, $L_{\mathcal{D}}(h)$, of a Hypothesis h induced by ℓ with respect to $\mathcal{D}$, as the expected value (according to $z\sim\mathcal{D}$) of ℓ:

$$L_{\mathcal{D}}(h) = \mathbb{E}_{\mathcal{D}}[\ell(h,z)]$$

Since definition 4.4.1 did not refer explicitly to the choice of a loss function, we shall keep it as our definition of an approximately correct learner and of the accuracy ε, also in the case of a general (not necessarily 0-1) loss function.

4.4.4 Agnostic-PAC Learnability

The new, more general framework we develop is called *Agnostic-PAC*.

Definition 4.4.4 Let $\varepsilon, \delta \in (0,1)$. We say that a learner, $\mathcal{A}$, is *Agnostic Probably Approximately Correct* (Agnostic-PAC) with a confidence δ and an accuracy ε, *with respect to* a loss function ℓ, hypothesis class $\mathcal{H}$ and a distribution $\mathcal{D}$ on $\mathcal{X} \times \mathcal{Y}$, if the probability of obtaining a training sample, S, for which $\mathcal{A}$ will output a prediction rule, $h_S \in \mathcal{H}$, with a loss, $L_{\mathcal{D}}(h_S)$, of at most ε away from the best possible loss achievable by *any* hypothesis in $\mathcal{H}$, is larger or equal to $1-\delta$:

$$\mathcal{D}^m\left(\left\{S \middle| L_{\mathcal{D}}(h_S) \leq \min_{h' \in \mathcal{H}} L_{\mathcal{D}}(h') + \varepsilon\right\}\right) \geq 1-\delta$$

Definition 4.4.5 — Agnostic PAC Learnability. A hypothesis class $\mathcal{H}$ is Agnostic-PAC learnable with respect to loss $\ell : \mathcal{H} \times (\mathcal{X} \times \mathcal{Y}) \to [0,\infty)$ if there exists a function $\tilde{m}_{\mathcal{H}} : (0,1)^2 \to \mathbb{N}$ and a learning algorithm $\mathcal{A} : (\mathcal{X} \times \mathcal{Y})^m \to \mathcal{H}$ with the following property:

- For any $\varepsilon, \delta \in (0,1)$
- For any distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$

when running the learning algorithm $\mathcal{A}$ on $m \geq \tilde{m}_{\mathcal{H}}(\varepsilon, \delta)$ i.i.d samples generated by $\mathcal{D}$, the algorithm returns a hypothesis $h_S = \mathcal{A}(S)$ such that, with probability of at least $1-\delta$:

$$\mathcal{D}^m\left(\left\{S \middle| L_{\mathcal{D}}(h_S) \leq \min_{h' \in \mathcal{H}} L_{\mathcal{D}}(h) + \varepsilon\right\}\right) \geq 1-\delta$$

It can be shown that an Agnostic-PAC learner with $\mathcal{A}$ is a PAC learner (see Ex.5). Using Agnostic-PAC learnability , let us write the "learning game" in the Agnostic PAC framework. Let $\varepsilon, \delta \in (0,1)$ be the desired accuracy and confidence levels, let $\mathcal{H}$ be a hypothesis class $\mathcal{H} \subset \mathcal{Y}^{\mathcal{X}}$ and a loss function ℓ. We play a game against Nature, with random payoff.

- We choose a sample size m and a learner $\mathcal{A} : (\mathcal{X}, \mathcal{Y})^m \to \mathcal{H}$ where m and $\mathcal{A}$ can depend on ε, δ.
- Nature knows our strategy, and, after us, chooses a strategy that consists of a probability distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$. This strategy can depend on $\varepsilon, \delta, \mathcal{H}$ and also on our strategy, m and $\mathcal{A}$.
- A sample $S \in (\mathcal{X} \times \mathcal{Y})^m$ of size m is drawn according to $\mathcal{D}$.
- The sample S is fed into $\mathcal{A}$ to produce a prediction rule $h_S = \mathcal{A}(S)$. Note that $h_S \in \mathcal{H}$.
- The payoff is $L_{\mathcal{D}}(h_S) = \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}}[\ell(h,(\mathbf{x},y))]$. It is random since S is random and therefore h_S is random.
- Nature does her best to win so we look for learners $\mathcal{A}$ that have a **guaranteed maximal loss** $L_{\mathcal{D}}(h)$ **for any** strategy $\mathcal{D}, f$ that Nature might play.
- To determine if we were successful in the game, we play the game many many times (both us and Nature play the same strategies, just the training samples drawn are different). We count and calculate the probability, over the random draws of training samples S, of the event $\{S \sim \mathcal{D}^m \,|\, L_{\mathcal{D}}(h_S) \leq \min_{h' \in \mathcal{H}} L_{\mathcal{D}}(h') + \varepsilon\}$. If this probability is found to be larger than $1-\delta$, that is, if the learner $\mathcal{A}$ we chose was Probably Approximately correct with accuracy ε and confidence δ - against Nature's best strategy - *we say that we've been successful (with regards to the parameters ε, δ) and the hypothesis class $\mathcal{H}$.*

Note that in order to calculate the probability of the event $\{S \sim \mathcal{D}^m \,|\, L_{\mathcal{D}}(h_S) \leq \min_{h' \in \mathcal{H}} L_{\mathcal{D}}(h') + \varepsilon\}$ we have to assume a knowledge of the "Oracle quantity" $\min_{h' \in \mathcal{H}} L_{\mathcal{D}}(h')$ - the best possible loss of any rule in $\mathcal{H}$.

One may ask whether the Agnostic PAC learnability, allowing, for example, more choices for the distribution $\mathcal{D}$, imply PAC-learnability. Not only the answer to this question is positive, but perhaps surprisingly, the inverse claim is also true. In other words, moving to the more general framework of Agnostic-PAC did not

change anything, as expressed by the following theorem:

Theorem 4.4.1 Let $\mathcal{X}$ be a sample space and $\mathcal{H} \subset \mathcal{Y}^{\mathcal{X}}$ a hypothesis class. Then $\mathcal{H}$ is PAC-Learnable if and only if it is Agnostic-PAC learnable.

4.5 The Fundamental Theorem of Statistical Learning

Recall that in the realizable case, an ERM learner was defined as any $h \in \mathcal{H}$ consistent with the training set $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ which in turn implied that it minimized the empirical risk. In the current more general case (where the notion of consistency is no more relevant - even the training set does not have to be consistent with itself since the same point may appear with different labels) we redefine ERM as *any* minimizer of the empirical risk.

Definition 4.5.1 — Empirical Risk and ERM Learner in the Agnostic-PAC framework. Let $h : \mathcal{X} \to \mathcal{Y}$ be a prediction rule. We define the *empirical risk* of h with respect to the loss function ℓ and the sample $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ by

$$L_S(h) = \frac{1}{m} \sum_{i=1}^{m} \ell(h, z_i), \quad z_i = (\mathbf{x}_i, y_i)$$

An *ERM Learning Algorithm*, $\mathcal{A}_{ERM}$, in the Agnostic-PAC framework is defined as an algorithm that outputs a hypothesis that minimizes the empirical risk:

$$\mathcal{A}_{ERM} : S \mapsto h, \quad h \in \left\{ \operatorname*{argmin}_{h \in \mathcal{H}} L_S(h) \right\}$$

Notice that as before the ERM rule may not be unique. There can be many hypotheses in $\mathcal{H}$ that achieve the minimum $\min_{h \in \mathcal{H}} L_S(h)$.
ERM learners are successful and logical due to the Weak Law of Large Numbers (WLLN) which states that if X_i are a series of i.i.d random variables and $\mu = \mathbb{E}(X_i)$, then

$$\lim_{m \to \infty} \frac{1}{m} \sum_{i=1}^{m} X_i = \mu$$

where the convergence is *in probability*. Namely, for any $\delta > 0$

$$\lim_{m \to \infty} \mathbb{P}\left\{ \left| \frac{1}{m} \sum_{i=1}^{m} X_i - \mu \right| > \delta \right\} = 0$$

which is equivalent to require that for any $\varepsilon, \delta > 0$ there is $m_0 \in \mathbb{N}$ such that for $m > m_0$,

$$\mathbb{P}\left\{ \left| \frac{1}{m} \sum_{i=1}^{m} X_i - \mu \right| > \delta \right\} < \varepsilon.$$

Observe now that for any h we have

- $\mathbb{E}_{\mathcal{D}}[L_S(h)] = L_{\mathcal{D}}(h)$
- By WLLN, when S is i.i.d sample of size m, $\lim_{m \to \infty} L_S(h) = L_{\mathcal{D}}(h)$, in probability
- Therefore for any $\delta > 0$ there is $m_0 \in \mathbb{N}$ such that for $m > m_0$,

$$\mathbb{P}\left\{ \left| L_S(h) - L_{\mathcal{D}}(h) \right| > \delta \right\} < \varepsilon.$$

- But does this mean that $\operatorname{argmin}_{h \in \mathcal{H}} L_S(h)$ is close to $\operatorname{argmin}_{h \in \mathcal{H}} L_D(h)$?

Although this is a good start, does this imply that $\operatorname{argmin}_{h \in \mathcal{H}} L_S(h)$ is close to $\operatorname{argmin}_{h \in \mathcal{H}} L_{\mathcal{D}}(h)$?

4.5.1 The Fundamental Theorem

Let us reformulate the Fundamental Theorem using Agnostic-PAC learnability and the generalized notion of ERM we just defined.

Theorem 4.5.1 — The Fundamental Theorem of Statistical Learning. Let $\mathcal{H}$ be a hypothesis class of binary classifiers with VC-dimension $d \leq \infty$. Then, $\mathcal{H}$ is **Agnostic-PAC learnable** if and only if $d < \infty$. In this case:

1. There are absolute constants C_1, C_2 such that the sample complexity of $\mathcal{H}$ satisfies

$$C_1 \frac{d + \log(1/\delta)}{\varepsilon^2} \leq m_{\mathcal{H}}(\varepsilon, \delta) \leq C_2 \frac{d + \log(1/\delta)}{\varepsilon^2}$$

2. Furthermore, the upper bound on sample complexity is achieved by the ERM learner.

Note that the "price" we pay for Agnostic PAC learning is that the sample complexity is proportional to $1/\varepsilon^2$ and not to $1/\varepsilon$ as in the PAC Fundamental theorem.

We conclude this chapter with partially going into the proof of the above theorem (4.5.1), and doing so for the qualitative rather than the quantitative part of the theorem. This will help to understand as to how is it that the VC-dimension characterizes learnability, namely, why is that $\mathcal{H}$ **is Agnostic-PAC learnable if and only if** $VCdim(\mathcal{H}) < \infty$. The first part of the theorem is that learning $\mathcal{H}$ is possible if and only if $\mathcal{H}$ is of finite VC-dimension. If it is not of finite VC-dimension then it is impossible to create an Agnostic-PAC learner $\mathcal{A}$ for $\mathcal{H}$ (with accuracy ε and confidence δ) by **any** learning algorithm and using **any** number of training samples.

We have already gained some intuition when discussing the No-Free Lunch theorem (4.2.1). We saw an informal argument that, if there exists $C \subset \mathcal{X}$ that is **shattered** by $\mathcal{H}$, then no learning algorithm can be a Probably Approximately correct learner if it is based on less than $|C|/2$ samples. Now, the statement $VCdim(\mathcal{H}) = \infty$ just means that there are subsets of $\mathcal{X}$ of arbitrary size that are shattered by $\mathcal{H}$ and therefore, no finite sample size will do.

The second part of Lemma 4.5.1, states that if $\mathcal{H}$ is a hypothesis class with $VCdim(\mathcal{H}) = d < \infty$ then $\mathcal{H}$ is Agnostic-PAC learnable as defined in Definition 4.4.4, using any ERM learner. To prove this we will need to define the Uniform Convergence property of hypothesis classes.

4.5.2 Uniform Convergence property

An ERM learner chooses a rule $ERM_{\mathcal{H}}(S)$ which minimizes $L_S(h)$ for the sample S at hand. We hope that the rule $h_S \in ERM_{\mathcal{H}}(S)$, which has minimal empirical risk, will generalize well. Formally, we have to prove that

$$\mathcal{D}^m \left\{ S \in (\mathcal{X} \times \mathcal{Y})^m \;\middle|\; |L_{\mathcal{D}}(h_S) - L_S(h_S)| \leq \varepsilon \right\} \geq 1 - \delta$$

This can only happen if S is a "special" sample - one for which for any $h \in \mathcal{H}$ the empirical risk $L_S(h)$ is pretty close to the generalization loss $L_{\mathcal{D}}(h)$. This is hard to prove. Note that for any $h \in \mathcal{H}$ we have $\mathbb{E}[L_S(h)] = L_{\mathcal{D}}(h)$, so, as we have seen above, by the weak law of large numbers, $L_S(h)$ converges to $L_{\mathcal{D}}(h)$ in probability as the sample size $m \to \infty$. This means that

$$\forall \mathcal{D}\ \forall h \in \mathcal{H}\ \forall \varepsilon, \delta \in (0,1) \quad \exists m_0 \in \mathbb{N} \quad \text{such that} \quad \mathbb{P}\{|L_S(h) - L_{\mathcal{D}}(h)| < \varepsilon\} > 1 - \delta$$

However m_0 depends on both $\mathcal{D}$ and h. We want m_0 that is **uniform** in the distributions $\mathcal{D}$ and the hypotheses $h \in \mathcal{H}$.

Definition 4.5.2 — Uniform Convergence of Function Sequences. A sequence of functions $f_n : X \to \mathbb{R}$ converges uniformly to $f : X \to \mathbb{R}$ if and only if

$$\forall \varepsilon > 0 \quad \exists m_0 \in \mathbb{N} \quad \text{such that} \quad \forall x \in X \ |f_n(x) - f(x)| < \varepsilon$$

Indeed, what we want is to ensure that $L_S(h)$ converges to $L_{\mathcal{D}}(h)$ **uniformly in** $\mathcal{D}$ **and in** $h \in \mathcal{H}$. This leads to the following definition.

Definition 4.5.3 — ε-representative. A training sample S is called ε-representative for $\mathcal{D}, \mathcal{H}, \ell$ if and only if

$$\forall h \in \mathcal{H} \quad |L_S(h) - L_{\mathcal{D}}(h)| < \varepsilon$$

This condition will ensure that minimizing $L_S(h)$ over $h \in \mathcal{H}$ will be close to minimizing $L_D(h)$ over $h \in \mathcal{H}$, which is approximately what we would like to achieve. Specifically, we can show that if we have an ε-representative training set S, then $ERM_{\mathcal{H}}(S)$ will "almost" achieve $\min_{h \in \mathcal{H}} L_{\mathcal{D}}(h)$.

Lemma 4.5.2 Let S be an $\varepsilon/2$-representative sample for $\mathcal{D}, \mathcal{H}, \ell$. Let h_S be any output of $ERM_{\mathcal{H}}(S)$, namely,$h_S \in \operatorname{argmin}_{h \in \mathcal{H}} L_S(h)$. Then

$$L_{\mathcal{D}}(h_S) \leq \min_{h \in \mathcal{H}} L_{\mathcal{D}}(h) + \varepsilon$$

Definition 4.5.4 — Uniform Convergence Property. A hypothesis class $\mathcal{H}$ is said to have the *uniform convergence property* if and only if there exists a function $m_{\mathcal{H}}^{UC} : (0,1)^2 \to \mathbb{N}$ such that for every $\varepsilon, \delta \in (0,1)$ and every distribution $\mathcal{D}$ on $\mathcal{X} \times \mathcal{Y}$

$$\mathcal{D}^m(\{S \in (\mathcal{X} \times \mathcal{Y})^m \,|\, S \text{ is } \varepsilon\text{-representative}\}) \geq 1 - \delta$$

It can then be shown that if a hypothesis class has the uniform convergence property with function $m_{\mathcal{H}}^{UC}$ then $\mathcal{H}$ is Agnostic-PAC learnable with sample complexity $m_{\mathcal{H}}(\varepsilon, \delta) \leq m^{UC}(\varepsilon/2, \delta)$.

Finite VC-Dimension Implies Uniform Convergence Property

So to show Part Two of the fundamental theorem (that ERM is a universal learner), it is enough to show that if $VCdim(\mathcal{H}) < \infty$ then $\mathcal{H}$ has the uniform convergence property. This means showing that for large enough m, that does not depend on $\mathcal{D}$, an i.i.d sample is ε-representative with probability at least $1 - \delta$, and that this holds for any possible $\mathcal{D}$.
To achieve uniformity across both $\mathcal{D}$ and $h \in \mathcal{H}$ we define the following function $F_m^{\mathcal{D}} : (\mathcal{X} \times \mathcal{Y})^m \to \mathbb{R}$ by

$$F_m^{\mathcal{D}}(S) = \sup_{h \in \mathcal{H}} \left| L_{\mathcal{D}}(h) - L_S(h) \right| \tag{4.4}$$

$F_m^{\mathcal{D}}$ maps a training sample of size m, to a real number measuring its "worse possible confusion" - the maximal difference, over $\mathcal{H}$, between an empirical risk of a hypothesis h and the generalization error of that h. Observe that $F_m^{\mathcal{D}}$ is a function of the random sample S, so it is a random variable, whose distribution depends on the distributions $\mathcal{D}^m$ of training sets of length m.

In essence, we would like to show that with high probability, $F_m^{\mathcal{D}}$ is small. Formally, we would like to show that for every $\varepsilon, \delta \in (0,1)$ there exists $m_{\mathcal{H}}^{UC}(\varepsilon, \delta) \in \mathbb{N}$ such that for every distribution $\mathcal{D}$:

$$\mathcal{D}^m\left\{F_m^{\mathcal{D}}(S) > \varepsilon\right\} < \delta$$

The Case Of Finite $\mathcal{H}$

To understand the key argument, let us first consider the easier case of finite $\mathcal{H}$ and see how Agnostic-PAC learnability can be proven.

Claim 4.5.3 Let $\varepsilon, \delta \in (0,1)$ then there exists $m_0 \in \mathbb{N}$ such that for any $m > m_0$ it holds that

$$\forall \mathcal{D} \text{ over } \mathcal{X} \times \mathcal{Y} \quad \mathcal{D}^m \left\{ F_m^{\mathcal{D}}(S) > \varepsilon \right\} \leq \delta$$

Proof. Directly by definition then

$$\begin{aligned} \mathcal{D}^m \left\{ F_m^{\mathcal{D}}(S) > \varepsilon \right\} &\overset{def.}{=} \mathcal{D}^m \left\{ S \,\middle|\, \exists h \in \mathcal{H}, |L_S(h) - L_{\mathcal{D}}(h)| > \varepsilon \right\} \\ &\overset{union\ bound}{\leq} \sum_{h \in \mathcal{H}} \mathcal{D}^m \left\{ S \,\middle|\, |L_S(h) - L_{\mathcal{D}}(h)| > \varepsilon \right\} \\ &\leq |\mathcal{H}| \cdot \max_{h \in \mathcal{H}} \mathcal{D}^m \left\{ S \,\middle|\, |L_S(h) - L_{\mathcal{D}}(h)| > \varepsilon \right\} \end{aligned}$$

We thus need to bound $\mathcal{D}^m \left\{ S \,\middle|\, |L_S(h) - L_{\mathcal{D}}(h)| > \varepsilon \right\}$ uniformly in $\mathcal{D}$ and h. By the weak law of large numbers (WLLN), since $L_S(h)$ is an empirical mean of i.i.d random variables with expected value $L_{\mathcal{D}}(h)$, we know that

$$\forall \varepsilon > 0 \quad \mathcal{D}^m \left\{ \left| L_S(h) - L_{\mathcal{D}}(h) \right| > \varepsilon \right\} \overset{m \to \infty}{\longrightarrow} 0$$

This is not sufficient as we want the bound to not depend on $\mathcal{D}, h$. What we need is known as a **concentration of measure** inequality: a way to bound the distance between the empirical mean and the expected value. Indeed, recall Hoeffding's Inequality: Let $\theta_1, \ldots \theta_m$ be a sequence of i.i.d random variables and assume that for all i, $\mathbb{E}[\theta_i] = \mu$ and $\mathbb{P}\{a \leq \theta_i \leq b\} = 1$. Then:

$$\forall \varepsilon > 0 \quad \mathbb{P}\left\{ \left| \frac{1}{m} \sum_{i=1}^{m} \theta_i - \mu \right| > \varepsilon \right\} \leq 2 \exp\left(-2 \frac{m\varepsilon^2}{(b-a)^2} \right)$$

Therefore, let us define $\theta_i = \ell(h, (\mathbf{x}_i, y_i))$. Observe that $L_{\mathcal{D}}(h) = \mathbb{E}[\theta_i]$, where the expectation is with respect to $\mathcal{D}$, and that $L_S(h) = \frac{1}{m} \sum_{i=1}^{m} \theta_i$. As such we get that

$$\begin{gathered} \mathcal{D}^m \left\{ S \,\middle|\, |L_S(h) - L_{\mathcal{D}}(h)| > \varepsilon \right\} \leq 2 \exp\left(-2m\varepsilon^2 \right) \\ \Downarrow \\ |\mathcal{H}| \cdot \max_{h \in \mathcal{H}} \mathcal{D}^m \left\{ S \,\middle|\, |L_S(h) - L_{\mathcal{D}}(h)| > \varepsilon \right\} \leq 2|\mathcal{H}| \exp\left(-2m\varepsilon^2 \right) \end{gathered}$$

By choosing that $m \geq \frac{\log(2|\mathcal{H}|/\delta)}{2\varepsilon^2}$ we get that:

$$\mathcal{D}^m \left\{ F_m^{\mathcal{D}}(S) > \varepsilon \right\} \leq 2|\mathcal{H}| \exp\left(-2m\varepsilon^2 \right) \leq \delta$$

■

The Case Of Infinite $\mathcal{H}$

Unfortunately, we are not apply to apply the union bound over an inifinite number of hypotheses. Instead, recall that for every finite $C \subseteq \mathcal{X}$, we write $\mathcal{H}_C$ for the hypotheses in $\mathcal{H}$, all restricted to C. The key to the proof is to understand **how fast** can the restriction $\mathcal{H}_C$ grow with $|C|$. If $|C| \leq VCdim(\mathcal{H})$ it could be that $\mathcal{H}$ shatters $|C|$ and therefore it could be that $|\mathcal{H}_C| = 2^{|C|}$. However, if $|C| > VCdim(\mathcal{H})$ it can't be - by definition - that $|\mathcal{H}_C| = 2^{|C|}$. So the question is how large can $|\mathcal{H}_C|$ be.

Definition 4.5.5 For a hypothesis class $\mathcal{H}$ Define $\tau_{\mathcal{H}}(m)$ by

$$\tau_{\mathcal{H}}(m) = \max \left\{ |\mathcal{H}_C| \,\middle|\, C \subset \mathcal{X}, |C| = m \right\}$$

This definition is a combinatorial property of $\mathcal{H}$: the maximal number of functions that can be obtained by restricting $\mathcal{H}$ to any subset of size m. The larger and more complicated $\mathcal{H}$, the larger we can expect $\tau_{\mathcal{H}}(m)$ to be. In other words, $\tau_{\mathcal{H}}(m)$ measures how fast - at most - $\mathcal{H}_C$ can grow with $|C|$. For example, we saw that if $VCdim(\mathcal{H}) = \infty$, then $\tau_{\mathcal{H}}(m) = 2^m$, namely, $\mathcal{H}_C$ can grow exponentially in $|C|$.

Definition 4.5.6 Let $\mathcal{H} \subset \mathcal{Y}^{\mathcal{X}}$. Suppose there exist $m_0 \in \mathbb{N}$, $b > 0$ and $\beta > 0$ such that: for all

$$\forall m > m_0 \quad \tau_{\mathcal{H}}(m) \leq b \cdot m^{\beta}$$

Then we say that $\mathcal{H}_C$ grows **polynomially** in $|C|$.

So the proof that a finite VC-dimension implies the uniform convergence is based on two parts:

1. If $|\mathcal{H}_C|$ grows polynomially in $|C|$, then $\mathcal{H}$ has the uniform convergence property. Hence it Agnostic-PAC learnable using the ERM rule.
2. If $VCdim(\mathcal{H}) < \infty$, then $|\mathcal{H}_C|$ grows polynomially in $|C|$.

Claim 4.5.4 Let $\mathcal{H}$ be a hypothesis class such that $|\mathcal{H}_C|$ grows polynomially in $|C|$ then $\mathcal{H}$ has the uniform convergence property.

Proof. Recall that we would like to show that

$$\mathcal{D}^m\left(\left\{F_m^{\mathcal{D}}(S) > \varepsilon\right\}\right) < \delta$$

uniformly in $\mathcal{D}$. Since $F_m^{\mathcal{D}}$ is a non-negative random variable, we consider Markov's inequality. We would like to define a sequence of numbers α_m that will depend on $\mathcal{H}$ but *not* on the distribution $\mathcal{D}$, for which it holds that

$$\mathbb{E}_{\mathcal{D}^m}\left[F_m^{\mathcal{D}}(S)\right] \leq \alpha_m \tag{4.5}$$

By doing so we will bound $\mathbb{E}_{\mathcal{D}^m}[F_m^{\mathcal{D}}(S)]$ uniformly across $\mathcal{D}$. If we are successful in doing so then

$$\mathbb{P}_{\mathcal{D}^m}\left\{\sup_{h\in\mathcal{H}}\left|L_{\mathcal{D}}(h) - L_S(h)\right| > \varepsilon\right\} = \mathbb{P}_{\mathcal{D}^m}\left\{F_m^{\mathcal{D}}(S) > \varepsilon\right\} \leq \frac{\mathbb{E}_{\mathcal{D}^m}[F_m^{\mathcal{D}}(S)]}{\varepsilon} \leq \frac{\alpha_m}{\varepsilon}.$$

Which means that with probability at least $1 - \alpha_m/\varepsilon$, a training set of length m is ε-representative. Therefore, we managed to achieve uniformity across both $h \in \mathcal{H}$ (by using $F_m^{\mathcal{D}}$, a supremum over h) and over $\mathcal{D}$ (by bounding the expected value of $F_m^{\mathcal{D}}$ independently of $\mathcal{D}$).

Note, that is we are able to find such as sequence α_m that decreases to 0, then for any ε, δ we can set m_0 to be such that for all $m > m_0$, $\alpha_m/\varepsilon < \delta$. This would imply that $\mathcal{H}$ has the uniform convergence property. To find such a sequence α_m we use the following lemma

Lemma 4.5.5 Let $F_m^{\mathcal{D}}$ be as in (4.4). Then independently of $\mathcal{D}$

$$\mathbb{E}_{\mathcal{D}^m}\left[F_m^{\mathcal{D}}(S)\right] \leq \mathcal{O}\left(\frac{\sqrt{\log(\tau_{\mathcal{H}}(2m))}}{\sqrt{2m}}\right) + o(m)$$

Since we assumed that $|\mathcal{H}_C|$ grows polynomially in $|C|$, we have for all $m > m_0$ (for some m_0) that $\tau_{\mathcal{H}}(m) \leq b \cdot m^{\beta}$ for some $b, \beta > 0$. Hence,

$$\mathbb{E}_{\mathcal{D}^m}[F_m^{\mathcal{D}}(S)] \leq O\left(\frac{\sqrt{\beta \cdot \log(2m)}}{\sqrt{2m}}\right) + o(m) \searrow 0$$

and therefore if $|\mathcal{H}_C|$ grows polynomially in $|C|$ then $\mathcal{H}$ has the uniform convergence property. ■

Claim 4.5.6 Let $\mathcal{H}$ be a hypothesis class with a finite VC-dimension. Then $|\mathcal{H}_C|$ grows polynomially in $|C|$

Proof. By definition, if $m \leq VCdim(\mathcal{H})$ then there exists a set $C \subset \mathcal{X}$, of size m, which is shattered by $\mathcal{H}$. This means that if $m \leq VCdim(\mathcal{H})$ then $\tau_{\mathcal{H}}(m) = 2^m$. It can be shown, see Ex.7, that if $m > VCdim(\mathcal{H})$ then $\tau_{\mathcal{H}}(m) \leq (em/d)^d$. Therefore, while $\tau_{\mathcal{H}}(m)$ grows exponentially in m for $m \leq VCdim(\mathcal{H})$, it only grows **polynomially** in m for $m > VCdim(\mathcal{H})$. ■

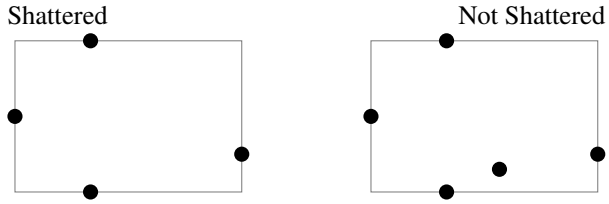
So, that was a taste of the proof of the second part of the fundamental theorem. We proved everything formally, except Lemma 2. This lemma is indeed deep and meaningful: it bounds the expected value of the "worse possible deviation" between empirical risk and generalization error, $\sup_{h\in\mathcal{H}} \left|L_{\mathcal{D}}(h) - L_S(h)\right|$, over a random choice of training sample, uniformly in $\mathcal{D}$. The bound uses $\tau_{\mathcal{H}}(m)$, which bounds how fast the size of a restriction $\mathcal{H}_C$ can grow with $|C|$.

4.6 Summary and Exercises

1. The *Axis-aligned rectangles* hypothesis class over the sample space $\mathcal{X} = \mathbb{R}^2$ is defined as:

$$\mathcal{H} = \left\{h_{(a_1,a_2,b_1,b_2)} : a_1 < a_2 \wedge b_1 < b_2\right\} \quad h_{(a_1,a_2,b_1,b_2)}(x_1,x_2) = \mathbb{1}_{x_1\in[a_1,a_2]\wedge x_2\in[b_1,b_2]}$$

   Note that every function in this hypothesis class defines a finite rectangle aligned with the x and y axes $\mathbb{R}^2$. Show that no set of 5 points can be shattered by the Axis-aligned rectangles class while any 4 points located on 4 different edges (away from the corners) of any given rectangle can:



   Hint: note that the 3 points (x_k,y_k), (x_i,y_i), and $(x_{k'},y_{k'})$ can not be shattered if $x_k \leq x_i \leq x_{k'}$ and $y_k \leq y_i \leq y_{k'}$.
2. Consider the case of a finite hypothesis class.
   - Show that the VC dimension of a finite $\mathcal{H}$ is at most $\log_2(|\mathcal{H}|)$.
   - Show that, over the same sample space, $\mathcal{X}$, VC dimensions can take any value between 1 and $\log_2(|\mathcal{H}|)$ even for hypothesis classes of the same size: Given $n \in \mathbb{N}$, find a sample space, $\mathcal{X}$, and a hypothesis class, $\mathcal{H}^{(n)}$ with $VCdim(\mathcal{H}^{(n)}) = \log_2(|\mathcal{H}^{(n)}|) = n$. Then, for each $k = 1..n-1$ construct a hypothesis class, $\mathcal{H}^{(k)}$, with $|\mathcal{H}^{(k)}| = |\mathcal{H}^{(n)}|$ but with $VCdim(\mathcal{H}^{(k)}) = k$.
3. Let $\mathcal{X} = \mathbb{R}^d$. The *hypothesis class* of half-spaces through the origin is defined as

$$\mathcal{H} = \left\{\mathbf{x} \mapsto sign(\mathbf{x}^\top \mathbf{w}) : \mathbf{w} \in \mathbb{R}^d\right\}$$

   - Show that $\{\mathbf{e}_1,\ldots,\mathbf{e}_d\}$ is shattered by $\mathcal{H}$.
   - Show that any $d+1$ points cannot be shattered (consider the standard basis vectors).
   - What is $VCdim(\mathcal{H})$?
4. Let $|\mathcal{X}| = \infty$. Let $\mathcal{H} \subset \{\pm 1\}^{\mathcal{X}}$ with VCdim($\mathcal{H}$)<∞. Let $\mathcal{H}'$ be the complement hypothesis class, namely, $\mathcal{H}' \equiv \{\pm 1\}^{\mathcal{X}} \setminus \mathcal{H}$. Let $C \subset \mathcal{X}$, $|C| < \infty$, be *any* finite subset of $\mathcal{X}$. Show that $\mathcal{H}'$ shatters C.
5. Let $\mathcal{X}$ be a sample space, $\mathcal{H} \subset \mathcal{Y}^{\mathcal{X}}$ a hypothesis class, and let ℓ_{0-1} be the $0-1$ loss function. Let $\varepsilon, \delta \in (0,1)$. Assume that $h \in \mathcal{H}$ is an Agnostic Probably Approximately correct learner with accuracy ε and confidence δ, with respect to ℓ_{0-1}, $\mathcal{H}$. Show that $\mathcal{A}$ is a Probably Approximately Correct learner (with accuracy ε and confidence δ).

6. Prove Lemma 4.5.2: for S an $\varepsilon/2$-representative sample for $\mathcal{D}, \mathcal{H}, \ell$ any ERM ourput h_S satisfies that

$$L_{\mathcal{D}}(h_S) \leq \min_{h \in \mathcal{H}} L_{\mathcal{D}}(h) + \varepsilon$$

7. Let $\mathcal{H}$ be some hypothesis class of functions $\mathcal{X} \to \{\pm 1\}$ with finite VC-dimension. If $m > VCdim(\mathcal{H})$ then $\tau_{\mathcal{H}}(m) \leq (em/d)^d$.
8. Let $\tilde{\mathcal{D}}$ be a distribution on $\mathcal{X}$ alone, and let $f : \mathcal{X} \to \mathcal{Y}$ be a labeling function, $\mathcal{Y} = \{\pm 1\}$. Construct an equivalent joint distribution $\mathcal{D}(\mathbf{x}, y)$ on $\mathcal{X} \times \mathcal{Y}$, assuming no-noise. Verify the answer by calculating the distribution for the pair $(\mathbf{x}, f(\mathbf{x}))$ and for the pair $(\mathbf{x}, -f(\mathbf{x}))$.
9. Consider again $\tilde{\mathcal{D}}(\mathbf{x})$ and $\mathcal{D}(\mathbf{x}, y)$ defined in Ex.8. Verify that Equation 4.2 gives, for $\tilde{\mathcal{D}}(\mathbf{x})$, the same misclassification loss as Equation 4.3 gives, for $\mathcal{D}(\mathbf{x}, y)$.

5. Ensemble Methods

In previous chapters we discussed different classification algorithms, each of which implements a different learning principle for choosing the learning rule $h_s \in \mathcal{H}$. In this chapter we will not cover new learning algorithms per se, but rather consider several general "meta-algorithms" for the creation of *ensembles* of classifiers. These collections of classifier can be applied to any existing learning algorithm and improve its performance. We will learn about the three B's: *Bootstrapping*, *Bagging* and *Boosting* and understand how they give us better control over the *Bias-Variance Trade-off*. These powerful techniques are broadly used and applicable for many difficult problems even outside the context of the meta-algorithms presented in this chapter.

5.1 Bias-Variance Trade-off

Throughout the book we have stated informally that the "larger" or "more complicated" our chosen hypothesis class, typically our learner will have lower **bias** and higher **variance**. We said informally that bias is part of the generalization error that is incurred by the "best" hypothesis in $\mathcal{H}$. If we think of an unknown labeling function f chosen by nature, then bias measures how well the unknown labeling function f can be decried by the "closest" hypothesis in $\mathcal{H}$. Intuitively, the larger $\mathcal{H}$, the more expressive power it has to describe more complicated functions f - hence a lower bias. We also said informally that variance is the part of the generalization error that is incurred by the fact that the training sample is random, hence our chosen rule h_s is also random. The larger $\mathcal{H}$ will be, the more freedom our learning algorithm has to "chase" random fluctuations in the training sample, which do not represent the underlying labeling we are trying to learn.

The variance part can be further broken down into two parts - one part comes from randomness in the choice of training samples while another part comes from the measurement noise or noise in the labels. For simplicity we will represent the variance as a single component.

As such, the bias-variance **tradeoff** is that: the more complicated the model, the smaller the bias and the larger the variance. Informally, the generalization error is some combination of the two. In cases where we can

tune the model complexity (that is, the size/complexity of the hypothesis class) we would like to look for the "sweet spot" of a model that has "just the right amount of complexity".

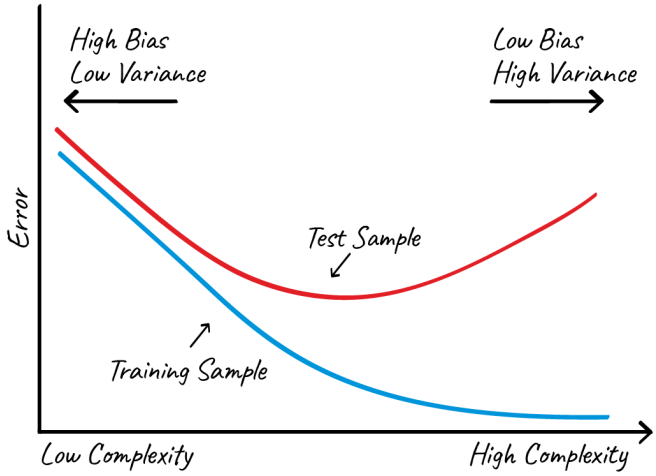


Figure 5.1: The bias-variance tradeoff: *Train- vs. test errors as function of complexity of h*

The methods describe below allow us to escape the tradeoff in some sense. They enable the reduction of the bias or variance of a learner without substantially increasing the other.

5.1.1 Generalization Error Decomposition

We have already encountered the generalization error decomposition when discussing linear regression problems. There we have shown how we can decompose the MSE into the bias and variance components (2.22). Next, let us revisit the decomposition but for some general loss function. Let $h^* = \text{argmin}_{h \in \mathcal{H}} L_{\mathcal{D}}(h)$ and $h_S = \mathcal{A}(S)$ be the output of a learning algorithm, then we can decompose the generalization error of the hypothesis returned by the learner as follows:

$$L_{\mathcal{D}}(h_S) = \underbrace{L_{\mathcal{D}}(h^*)}_{\varepsilon_{approximation}} + \underbrace{L_{\mathcal{D}}(h_S) - L_{\mathcal{D}}(h^*)}_{\varepsilon_{estimation}} \tag{5.1}$$

- The **approximation error** is $L_{\mathcal{D}}(h^*)$. Namely, the error of the hypothesis $h \in \mathcal{H}$ achieving the lowest generalization error. This term does not depend at all on our training sample and its size m. It depends only on the selection of $\mathcal{H}$. As we expand the hypothesis class to become richer we might find a better hypothesis for explaining the data. This error is what we already know as the **bias** error, induced by restricting the hypothesis class.
- The **estimation error** is $L_{\mathcal{D}}(h_S) - L_{\mathcal{D}}(h^*)$. Namely, it is the difference between the generalization error achieved by the selected hypothesis and the best hypothesis in $\mathcal{H}$. This term depends on the training set and its size. This error is what we already know as the **variance** error.

5.2 Ensemble/Committee Methods

> "A collective wisdom of many is likely more accurate than any one." — Aristotle, in *Politics*, circa 300BC

Before exploring specific ensemble methods, let us analyze some mathematical properties of a committee based decision. This will provide insights into how committee based methods manage to improve generalization. Consider a committee of T members, which has to make a "yes"/"no" decision. Each member casts a

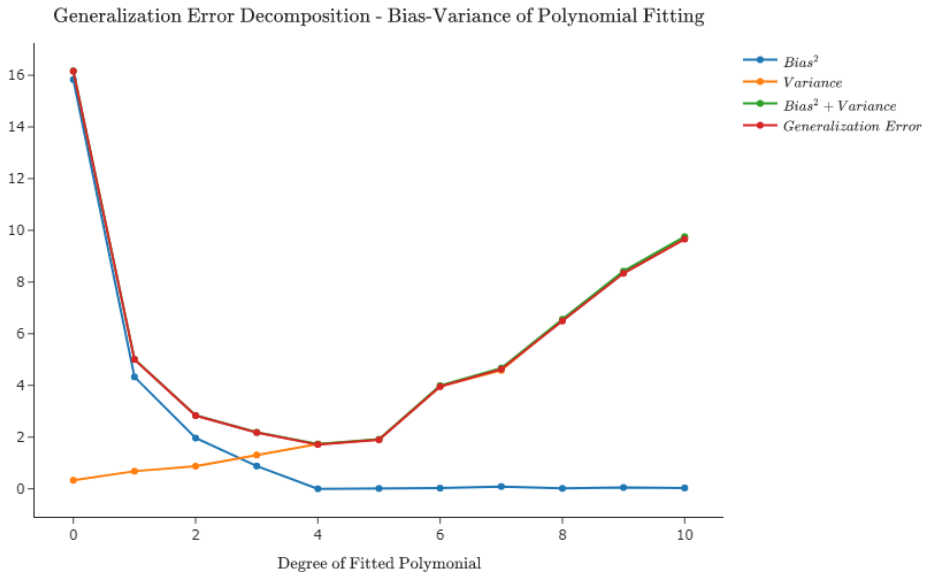


Figure 5.2: Bias-Variance Tradeoff Graph: *for polynomial fitting of true polynomial degree* 4.
Ensemble Methods Examples

vote, which with probability p_i being correct and probability $1-p_i$ being wrong. Let us further assume for simplicity that all members are "equally wise", so that p is the same for all members. After all members vote, the committee's decision is simply the majority vote. For this setup we can ask questions such as:

- What is the probability of the committee deciding the right decision?
- What would a typical decision be? and how consistent is it?
- How does the number of members in the committee influence the measures above?
- If committee members are not independent from one another, and influence each other's decisions, how does it influence the measures above?

We begin answering these questions theoretically starting with the probability of a committee deciding the right decision.

Lemma 5.2.1 Let $X_1,\ldots,X_T \overset{i.i.d}{\sim} Ber(p)$ taking values in $\{\pm 1\}$ and denote $X=\sum X_i$. The probability of the committee making the correct decision is $\mathbb{P}(X>0)$.

Proof. As the committee decides by a majority vote then the probability of the committee deciding right is the same as the probability of having more members deciding right than members deciding wrong:

$$\mathbb{P}(\text{Committee decides right}) = \mathbb{P}(|\text{Decided right}| > |\text{Decided wrong}|)$$

W.o.l.g, suppose the true answer is $+1$. As each random variable takes a value in $\{\pm\}$ we could express the collective vote as $X = sign(\sum X_i)$. If the committee decided right, then there are more members that voted right and $\sum X_i > 0$ which means that $X=1$. On the other hand, if the committee decided wrong, then more members voted wrong and $\sum X_i <= 0$ which means that $X=-1$. So, we conclude that:

$$\mathbb{P}(\text{Committee decides right}) = \mathbb{P}(X>0)$$

■

Lemma 5.2.2 Let $X_1,\ldots,X_T \overset{i.i.d}{\sim} Ber(p)$ taking values in $\{\pm 1\}$ with $p > 0.5$ and denote $X = \sum X_i$. The probability of the committee deciding correctly is bounded below by $1-\exp\left(-\frac{T}{2p}\left(p-\frac{1}{2}\right)^2\right)$.

Proof. W.l.o.g let us assume that the correct answer is $+1$ and denote $X = \sum_{i=1}^{T} X_i$. We are therefore interested in bounding $\mathbb{P}(X > 0)$ from below. We will achieve this by bounding $\mathbb{P}(X \leq 0)$ from above. Notice that for any $a > 0$ it holds that:

$$\mathbb{P}(X \leq 0) = \mathbb{P}(-aX \geq 0) = \mathbb{P}\left(e^{-aX} \geq e^0\right)$$

Now, using Markov's inequality

$$\mathbb{P}(X \leq 0) \leq \mathbb{E}\left[e^{-aX}\right] = \mathbb{E}\left[e^{-a\sum_i X_i}\right] \overset{iid}{=} \mathbb{E}\left[e^{-aX_1}\right]^T$$

Notice that as $X_1 \sim Ber(p)$ over $\{\pm 1\}$ it holds that:

$$\mathbb{E}\left[e^{-aX_1}\right] = pe^{-a} + (1-p)e^{a} = e^{a}\left(1-p+pe^{-2a}\right) \leq e^{a-p+pe^{-2a}}$$

where the last inequality is because $1+x \leq e^x$. Next, we use the inequality $x\ln(x) \geq \frac{x^2}{2} - \frac{1}{2} \quad x \in (0,1)$. For a selection of $a = \frac{1}{2}ln(2p)$ which is positive for $p > 0.5$ we get that:

$$\begin{array}{rcllcl}
\mathbb{P}(X \leq 0) & \leq & \mathbb{E}\left[e^{-aX_1}\right]^T & & \leq & \exp\left(T\left(a-p+pe^{-2a}\right)\right) \\
& = & \exp\left(T\left(\frac{1}{2}ln(2p)-p+\frac{1}{2}\right)\right) & & = & \exp\left(Tp\left(-\frac{1}{2p}ln\left(\frac{1}{2p}\right)-1+\frac{1}{2p}\right)\right) \\
& \leq & \exp\left(Tp\left(\frac{1}{2}-\frac{1}{2(2p)^2}-1+\frac{1}{2p}\right)\right) & & = & \exp\left(-\frac{Tp}{2}\left(\frac{1}{4p^2}-\frac{1}{p}+1\right)\right) \\
& = & \exp\left(-\frac{Tp}{2}\left(\frac{1}{2p}-1\right)^2\right) & & = & \exp\left(-\frac{T}{2p}\left(p-\frac{1}{2}\right)^2\right)
\end{array}$$

Finally, we conclude that:

$$\mathbb{P}(X > 0) = 1 - \mathbb{P}(X \leq 0) \geq 1 - \exp\left(-\frac{T}{2p}\left(p-\frac{1}{2}\right)^2\right)$$

■

Therefore, it turns out that if each member is typically right ($p > 0.5$) then the probability that the committee is right is higher than any individual member. In addition, the probability of the committee being wrong decays with a rate of $\mathcal{O}\left(e^{-T}\right)$. Relating this to our learning scheme, it means that the confidence $1-\delta$ in our prediction increases exponentially as T increases.

Based on (5.2.1) and (5.2.2) we can now simulate different scenarios for different values of T and p and see how committees behave. Figure 5.3 shows the theoretical bound achieved above and the empirical results for committees if increasing size and for different values of p. We can see that:

- The larger the committee size T the higher the probability of the committee being correct.
- The larger the probability of each committee member of being correct p, the fewer committee members are needed for the committee to be correct with probability of 1.

We also see that empirical results agree with the theoretical lower bound devised above.

5.2.1 Uncorrelated Predictors

Next, let us calculate what is a typical decision ($\mathbb{E}(X)$) and how consistent is it ($Var(X)$)? Let $X_1,\ldots,X_T \overset{iid}{\sim} Ber(p)$ taking values of $\{\pm 1\}$ with $p > 0.5$. What is the expectation and variance of $X = \frac{1}{T}\sum_{i=1}^{T} X_i$?

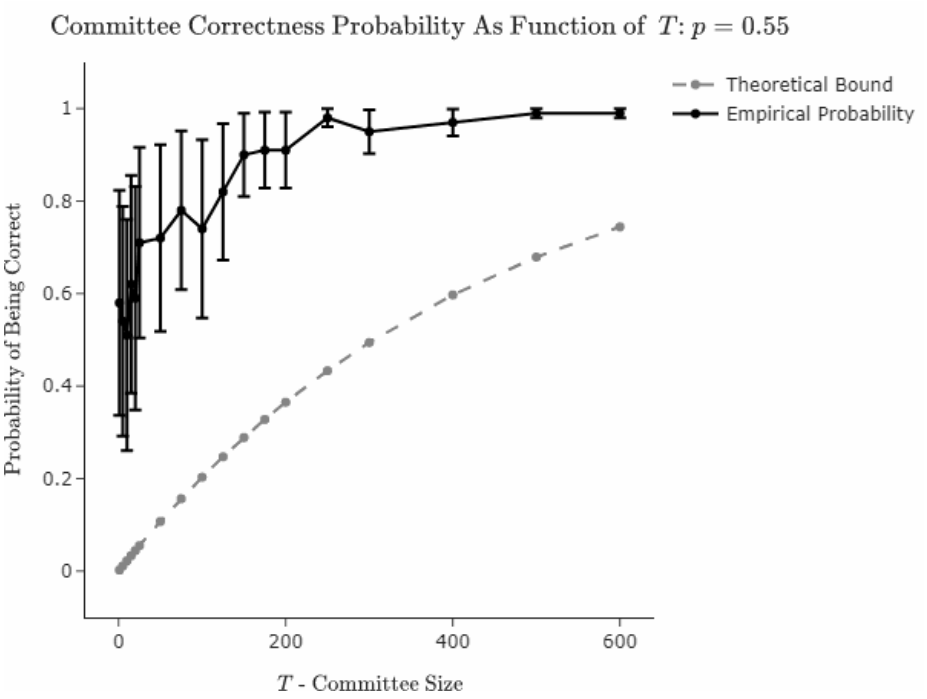


Figure 5.3: ***Committee Decision - Correctness Probability:*** *Theoretical bounds and empirical results as function of T, p.* *Ensemble Methods Examples*

We begin with calculating the expectation and variance of each committee member:

$$\begin{aligned}\mathbb{E}[X_i] &= 1\cdot\mathbb{P}(X_i=1)+(-1)\cdot\mathbb{P}(X_i=-1)\\ &= 2p-1\end{aligned}$$

$$\begin{aligned}Var(X_i) &= \mathbb{E}\left[(X_i-\mathbb{E}[X_i])^2\right]\\ &= p(1-(2p-1))^2+(1-p)(-1-(2p-1))^2\\ &= 4p(1-p)^2+4p^2(1-p)\\ &= 4p(1-p)\end{aligned}$$

Then, for X:

$$\begin{aligned}\mathbb{E}[X] &= \tfrac{1}{T}\textstyle\sum_i \mathbb{E}[X_i] = 2p-1\\ Var(X) &= \tfrac{1}{T^2}Var\left(\textstyle\sum_i X_i\right) \overset{iid}{=} \tfrac{1}{T^2}\textstyle\sum_i Var(X_i) = \tfrac{4}{T}p(1-p)\end{aligned}$$

Therefore, when using a committee of independent members the expectation of decision remains the same while decreasing the variance at a rate of $\mathcal{O}\left(\frac{1}{T}\right)$. In other word, we are able to keep the same accuracy while increasing the confidence.

5.2.2 Correlated Predictors

In practice, however, committee members rarely vote independently. So let us assume that each two members are correlated with equal correlation $\rho \in [0,1]$.

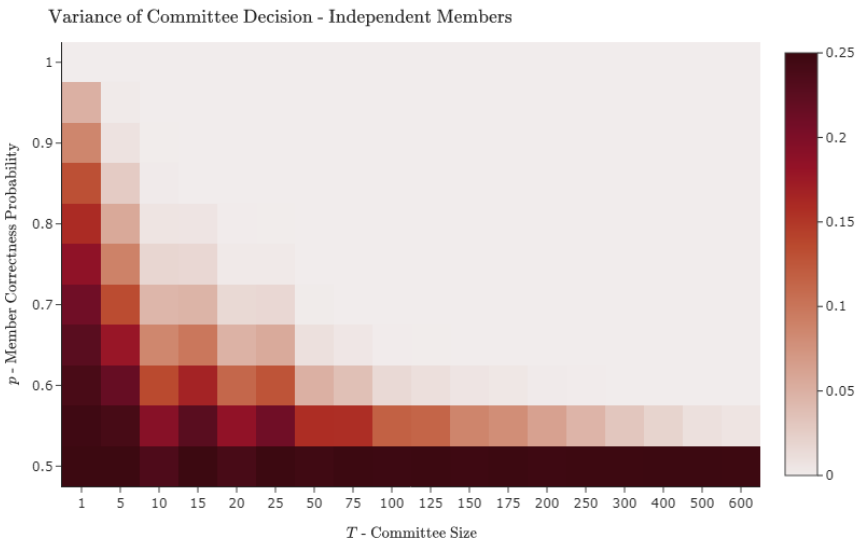


Figure 5.4: Variance of Committee Decision: *with independent members, as function of* T, p.
Ensemble Methods Examples

Lemma 5.2.3 Let $X_1, \ldots, X_T$ be a set of identically-distributed real-valued random variables such that: $Var(X_i) = \sigma^2$ and $corr(X_i, X_j) = \rho$, $i \neq j$. The variance in the commitee's decision is given by $\rho\sigma^2 + \frac{1}{T}(1-\rho)\sigma^2$.

Proof. As X is the average of T identically distributed random variables:

$$Var(X) = Var\left(\frac{1}{T}\sum_i X_i\right) = \frac{1}{T^2}\left[\sum_i Var(X_i) + 2\sum_{i<j} Cov(X_i, X_j)\right]$$

Recall that the correlation between two random variables is defined as:

$$corr(A,B) := \frac{Cov(A,B)}{\sqrt{Var(A)\,Var(B)}}$$

and therefore for any $i \neq j$:

$$Cov(X_i, X_j) = corr(X_i, X_j)\sqrt{Var(X_i)\,Var(X_j)} = \rho\sigma^2$$

Plugging this back into the variance:

$$Var(X) = \frac{1}{T^2}\left[T\sigma^2 + 2\binom{T}{2}\rho\sigma^2\right] = \frac{\sigma^2}{T} + \left(1 - \frac{1}{T}\right)\rho\sigma^2 = \rho\sigma^2 + \frac{1}{T}(1-\rho)\sigma^2$$

■

All together, we have seen analytically and quantitatively that given a committee of members deciding by majority vote, where decisions of members are correlated with correlation ρ and each member is correct with probability p:

- If $p > 0.5$ the decision made by the committee improves with T in two ways: higher probability of being right, and its decision will be more consistent.
- If $\rho > 0$ then increasing T will increase the probability of the decision made by the committee being right up to a certain point.

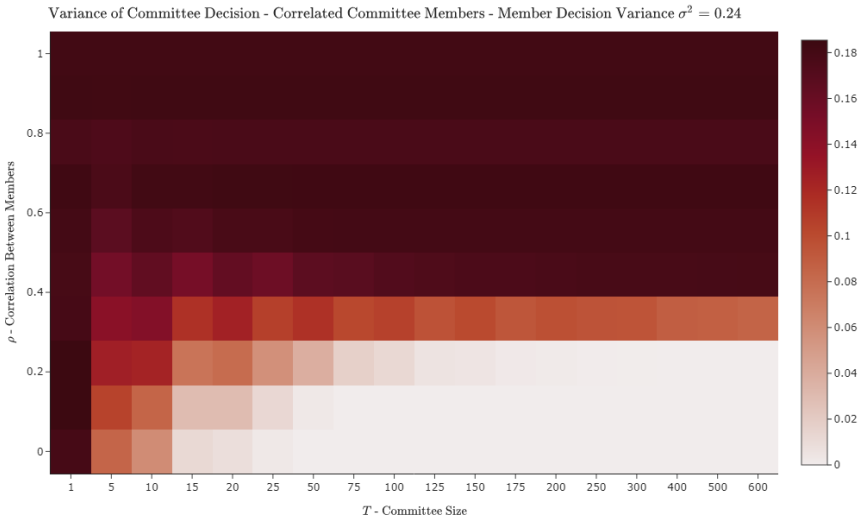


Figure 5.5: Variance of Committee Decision: *with correlated members, as function of* T, ρ. *Ensemble Methods Examples*

5.2.3 Committee Methods In Machine Learning

Let us apply these ideas to machine learning applications. Suppose we have T training samples $S_1, \ldots, S_T$ of size m chosen independently from $\mathcal{X}$ according to some distribution $\mathcal{D}$. Let $\mathcal{A}$ be a learning algorithm and train it on each of the training samples, to obtain $h_{S_1}, \ldots, h_{S_T}$. Let us consider $h_{S_t}(x)$ for some $x \in \mathcal{X}$. As $S_t \overset{i.i.d}{\sim} \mathcal{D}^m$ we can think of the training set as a random variable. This means that the also h_{S_t} obtained by training $\mathcal{A}$ over S_t is a random variable. Lastly, it means that we can think of the prediction $h_{S_t}(x)$ as a random variable, which has some distribution. In addition, notice that as the training samples are chosen independently, predictions of different h_{S_t} over x are also independent random variables. So, if we use $h_{S_1}, \ldots, h_{S_T}$ in a committee we have the situation described above. The generalization loss will be reduced as T grows as the variance of the prediction will decrease as a rate of $1/T$.

However, in batch learning we don't have T training samples, but rather just one, so how would we acquire such different hypotheses? We cannot train $\mathcal{A}$ over S multiple times as we will get identical predictions, which are perfectly correlated. Instead, what we would like to do is to create T training samples from the original one. We would like to find a way by which we could mimic fresh independent draws of new training samples of size m according to $\mathcal{D}$.

Definition 5.2.1 — Committee Methods. Let $\mathcal{A}$ be some learner predicting labels in $\{\pm 1\}$. A committee method over $\mathcal{A}$ is the function:

$$h(x) = \text{sign}\left(\sum_{t=1}^{T} h_t(x)\right)$$

That is, in committee methods (or ensembles) we take an existing "base" learner and apply it on a sequence of T training samples. For the remainder of this chapter we will introduce two very different ideas for building the committee member rules.

5.3 Bagging

5.3.1 The Bootstrap

For the first committee method we begin with introducing a concept from statistics: *The Bootstrap*. This is one of the most groundbreaking ideas of statistics in the 20th century, where we create new "artificial" training samples from the one training sample at hand.

Given a training sample $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ we are going to construct a new training sample, called a *bootstrap sample* S^{*1}. We sample m times from S *with replacement* and denote this "new" sample by:

$$S^{*1} = \left\{ \left(\mathbf{x}_i^{*1}, y_i^{*1}\right) \right\}_{i=1}^m$$

Of course, since we sampled from S with replacements, there might be repeated samples in S^{*1}, even if S itself had no repeated samples. Now we can repeat this process B times, obtaining B training samples, each of length m: $S^{*1}, \ldots, S^{*B}$. The samples in the b-th training sample will be denoted

$$S^{*b} = \left\{ \left(\mathbf{x}_i^{*b}, y_i^{*b}\right) \right\}_{i=1}^m$$

Using the newly created bootstrap samples we can now train our base learner $\mathcal{A}$ over each one separately, obtain B prediction rules and form an ensemble. But how is it that bootstrap actually works? Assume the samples in our learning problem are *i.i.d* samples from an unknown distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$. We are hoping that each Bootstrap from S somehow behaves like a fresh *i.i.d* sample from $\mathcal{D}$ itself. Given a training sample S we can define the *empirical distribution* $\hat{\mathcal{D}}_S$ induced by S on $\mathcal{X} \times \mathcal{Y}$ as the following probability distribution on $\mathcal{X} \times \mathcal{Y}$: for a subset $C \subset \mathcal{X} \times \mathcal{Y}$, define:

$$\hat{\mathcal{D}}_S((X,Y) = (x,y)) := \begin{cases} \frac{1}{m} & (x,y) \in S \\ 0 & (x,y) \notin S \end{cases}$$

or equivalently, for any $C \subset \mathcal{X} \times \mathcal{Y}$:

$$\hat{\mathcal{D}}_S(C) := \frac{|C \cap S|}{m}$$

where for simplicity we assume all samples in S are unique. Observe that this is equivalent to putting a probability mass of $1/m$ on each of the points of S, and zero mass on all other points in $\mathcal{X} \times \mathcal{Y}$. Observe that a bootstrap sample S^{*b} is just an *i.i.d* draw of m points from the *empirical distribution* $\hat{\mathcal{D}}_S$ induced by the one training sample we have, S. As m grows, namely as S becomes larger, the empirical distribution $\hat{\mathcal{D}}_S$ converges in distribution to $\mathcal{D}$. The idea behind the bootstrap is that, if $\hat{\mathcal{D}}_S$ is not so different from $\mathcal{D}$, then m *i.i.d* draws from $\hat{\mathcal{D}}_S$ is a good approximation to m *i.i.d* draws from $\mathcal{D}$.

One way to see the convergence of the empirical distribution to the underlying distribution is on the real line. Figure 5.6 shows the empirical CDF of samples drawn from a standard normal distribution and compares it with the theoretical CDF of the distribution. As the number of samples increases the empirical CDF converges to the CDF of the standard distribution.

5.3.2 Bagging

The idea of Bootstrap samples can be used in any scenario where we wish to create new artificial samples from our only training sample S. It has many uses throughout machine learning, statistics and data science. **Bagging** is a nickname of one such usage where we use Boostrap, to improve the accuracy of an existing supervised machine learning algorithm.

We start with a "base" learning algorithm $\mathcal{A}$ and a training sample S. We then form T bootstrap training samples, $S^{*1}, \ldots, S^{*T}$, each of size m. We then train our learner *separately* on each of the T bootstrap training

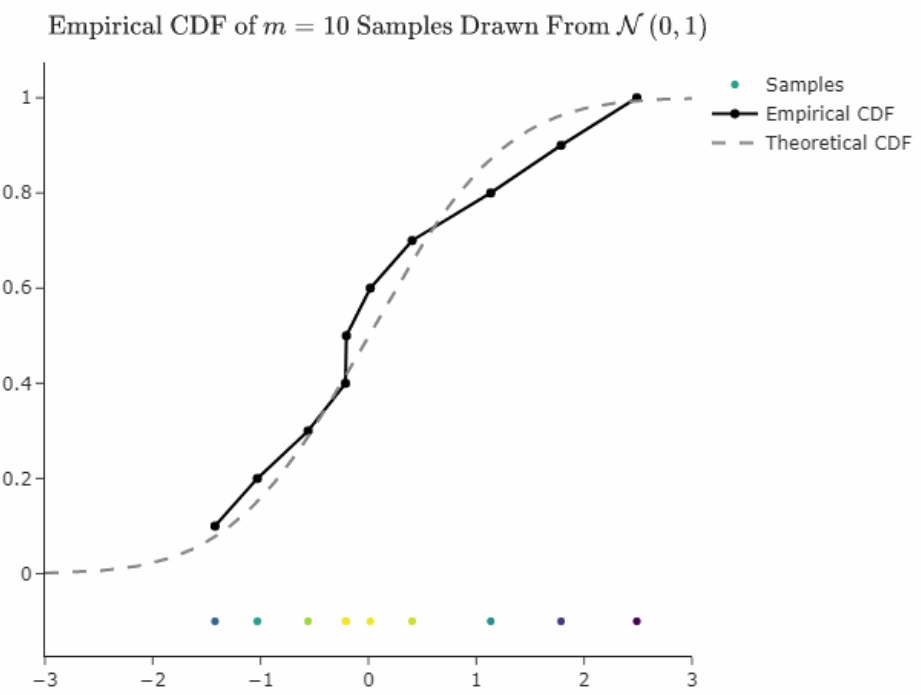


Figure 5.6: Empirical CDF: *of i.i.d samples drawn from a standard normal distribution. Ensemble Methods Examples*

samples. We form the committee $h_{S^{*1}}, \ldots, h_{S^{*T}}$ and store all T trained models. When we need to classify a new test sample $x \in \mathcal{X}$, we run x through all the rules and classify using the majority vote of the committee,

$$h_{bag}(x) := \text{sign}\left(\sum h_{S^{*t}}(x)\right)$$

For example, if we run Bagging on top of the Decision Tree classifier, we'll obtain a committee of decision trees:

Note that our learner $\mathcal{A}$ must know how to handle repeated samples. We may have them anyway in S, but running on a bootstrap sample we are sure to have them. Some learning algorithms suffer when there are repeated samples - as they cause numerical problems (for example, linear and logistic regression), while for others it isn't a problem (for example, decision trees and k-NN).

5.3.3 Bagging Reduces Variance

We saw that a committee majority vote reduces variance, but as seen in Figure 5.6 only to a certain degree. The amount of variance reduced is determined by the correlation between committee members. Therefore we can expect bagging to reduce variance as T increases, which will reduce the generalization error, but only proportionate to the correlation between the bagged prediction rules.

5.3.4 Random Forests - Bagging and De-correlating Decision Trees

As such we would like to find a way to de-correlate the committee members - namely, cause their predictions to somehow be less correlated. One way to do so is by slightly restricting each learner, in a random way, and hope that the performance gain (in bagging them) due to de-correlation is more than the performance loss to each learner by the restrictions. The most well know example of this principle is *Random Forests*. Ex.2

Recall the Decision Tree classification algorithm over $\mathcal{X} = \mathbb{R}^d$. We have a training sample S with m points.

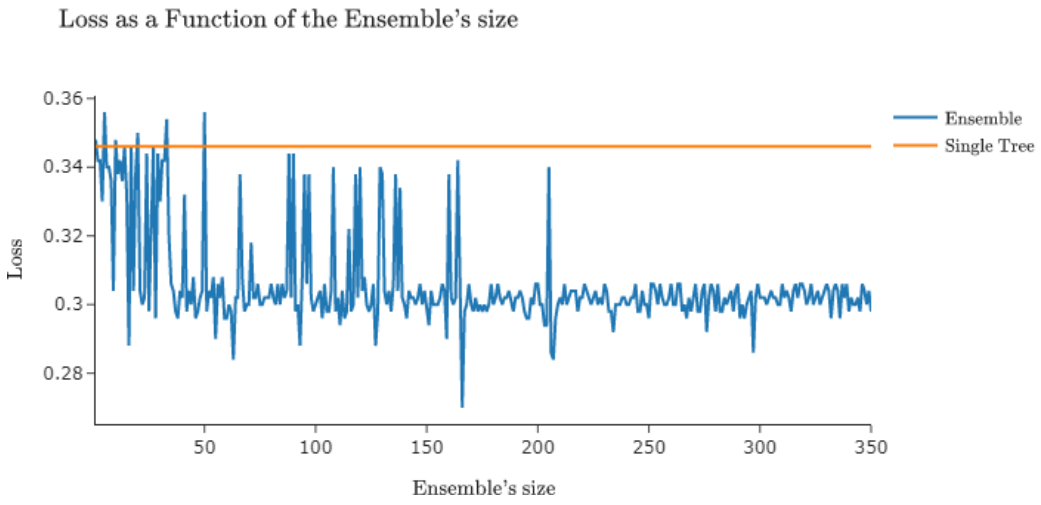


Figure 5.7: *Collection of Bagged Decision Trees. Ensemble Methods Examples*

The Random Forest classifier is obtained by using Bagging on top of the Decision Tree algorithm, *with the important step* that drives the de-correlation: the algorithm has a tuning parameter $k \leq d$. When growing each decision tree, in each split, we choose uniformly at random a subset of k out of the d features. We choose the split only among these k features. Formally:

Algorithm 5 Random Forests

procedure RANDOM-FOREST(training set $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$, maximal tree depth $R \in \mathbb{N}$, minimal training samples in any leaf m_{min}, number of trees T, number of coordinates to choose from in each split k)
  for $t = 1, \ldots, T$ **do**
    Draw a Bootstrap sample S^{*t} from S
    Train a decision tree $h_{S^{*t}}$ on the sample S^{*t} where at each split
    if not reached maximal depth R or minimal number of samples m_{min} **then**
      Select uniformly at random k features from $[d]$
      Choose the best feature to split by from the set of k features
      Split based on selected feature and threshold
    end if
  end for
  return $h_S(\mathbf{x}) = \text{sign}\left(\sum_{t=1}^T h_t(\mathbf{x})\right)$
end procedure

This de-correlation trick works: pretty much on every classification problem you will work on, you will observe something like the next plot: Bagging trees is much better than a single tree, and Random Forest (Bagging with the de-correlation trick) is better than just Bagging trees.

Bagging - Discussion Points

- **Can Bagging harm our prediction?** Always remember that a committee of fools (a committee where each member has probability $p < 0.5$ to make the right decision) makes worse decisions than a single member. So, when our base learner is so poor that its generalization loss is less than 0.5 we shouldn't

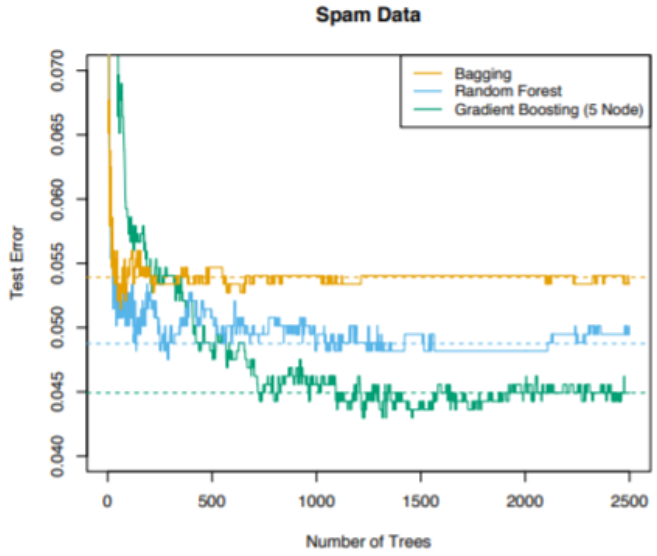


Figure 5.8: *Test error of simple Bagging of decision trees (no de-correlation), Random Forests, and Gradient Boosting of Trees. (Source: ESL)*

use Bagging.

- **What are the disadvantages of Bagging?** As we train not one but T models we need to train all T of them. This directly increases time complexity by a factor proportionate to T. In addition, for predictions, as we need all T models we must store all T of them. Lastly, we loose interpretability as it is much harder to understand why the committee made the decision. We need to understand the decision of each of the T members.
- **Parallelism** From the computational perspective, it is important to note that Bagging in general (and Random Forests in particular) is *embarrassingly parallelizable*. When training a Bagging model with T committee members, we can use T machines in parallel, each using its own random seed to select Bootstrap samples (and random splits, in Random Forest). The machines do not need to interact; when each machine is done, it returns the committee member h_t to the master node.
- **Predicted Class Probabilities:** Can we use the *proportion* of the committee members who voted $+1$ as a predicted class probability? Estimated class probabilities are estimates of $\mathbb{P}\{Y = +1, | X = x\}$. The proportion of members who voted $+1$ estimates $\mathbb{P}\{h_S(x) = +1\}$, which is a different quantity.

5.4 Boosting

Bootstrap and Bagging produce an ensemble of classifiers each trained over a "new" artificial training sample generated from the original one. Boosting on the other hand utilizes the single training sample and produces different classifiers by assuming different distribution over these samples. In Boosting we take a "weak" learning algorithm, an algorithm with better-than-random but possibly not so good accuracy (i.e. generalization error) and *boost* it using a clever committee method to obtain a learning algorithm with good accuracy.

In Bagging, we *pretended* to have fresh training samples $S_1, \ldots, S_T$, and each committee member trained on a different sample. In Boosting on the other hand, we go even further and *pretend* to have *different underlying distributions* $\mathcal{D}$ from which the training sample is drawn. More specifically, in Boosting each committee member h_t is the result of running $\mathcal{A}$ against a training sample S_t that mimics an *i.i.d* sample of size m from a *different distribution* $\mathcal{D}^t$. Whereas in Bagging each committee member is trained independently of all other members, in Boosting the committee members are trained sequentially, one after the other, and each is an improvement, in some sense, on the previous one.

The clever idea behind Boosting is that after we finish training h_t, based on the distribution $\mathcal{D}^t$, we update the distribution in a way that increases the measure of samples where h_t was wrong. This way, we force h_{t+1} to try and do better on those particular samples.

What is meant by "running $\mathcal{A}$ against the training sample S with distribution $\mathcal{D}^t$"? One way to interpret this is to take a *weighted Bootstrap* sample from S, where the probability of selecting $(x,y) \in S$ is proportional to $\mathcal{D}^t(x,y)$. A simpler way to interpret this is as follows. If $\mathcal{A}$ uses the ERM principle, say for standard misclassification ($0-1$ loss), namely, looking to minimize the empirical risk,

$$L_S(h) = \sum_{i=1}^{m} \mathbb{1}_{y_i \neq h(\mathbf{x}_i)}$$

then we can use S itself and have the base learner minimize the *weighted* empirical risk

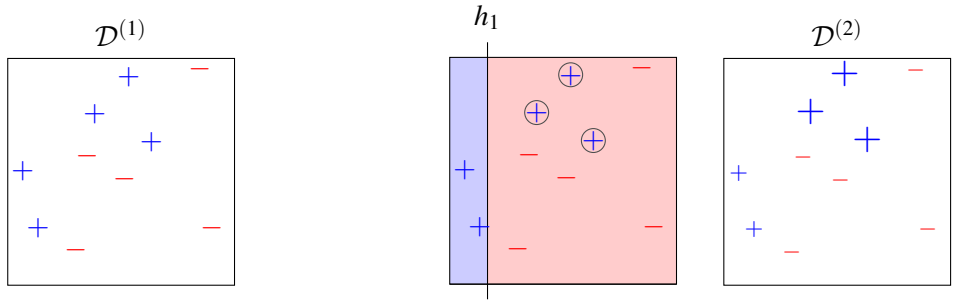
$$L_{S,\mathcal{D}^t}(h) = \sum_{i=1}^{m} \mathcal{D}_i^t \mathbb{1}_{y_i \neq h(\mathbf{x}_i)}$$

where for each $(\mathbf{x}_i, y_i) \in S$ we write $\mathcal{D}_i^t := \mathcal{D}^t(\mathbf{x}_i, y_i)$, so that $\sum_{i=1}^m \mathcal{D}_i^t = 1$.

Observe that these two interpretations are equivalent in expectation. Indeed, the expected number of times for a sample $(\mathbf{x}_i, y_i)$ to appear in the weighted Bootstrap sample is $\mathcal{D}_i^t$, and so it would (in expectation) appear $\mathcal{D}_i^t$ times in the empirical risk sum.

R Note that we usually prefer the second option (using weighted empirical risk) to the first option (using weighted bootstrap). It's more computationally efficient, and does not require worrying about repeated samples. However, the first option (using weighted bootstrap) is always available. The second option (using weighted empirical risk) is not always possible, and is implemented ad-hoc for the particular base learner we are boosting.

To understand this idea consider the following scenario. Suppose we begin with a training sample S of five positive and five negative samples as seen in Figure 5.9. At first we define a uniform distribution over the samples. Then suppose we fit a threshold classifier over S and $\mathcal{D}^{(1)}$ producing h_1. Notice that h_1 misclassified 3 samples. As such we update the distribution of weights over the samples increasing the weights of the misclassified samples and decreasing the weights of the correctly classified samples, forming $\mathcal{D}^{(2)}$.

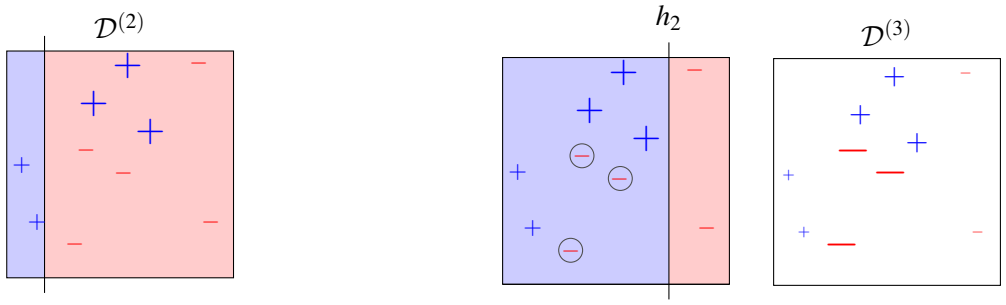


(a) *Initial uniform distribution over training sample*

(b) *Fitting h_1 over $\mathcal{D}^{(1)}$ and producing $\mathcal{D}^{(2)}$*

Figure 5.9: Boosting illustration: *Initial sample distribution and first iteration*

Next, we use $\mathcal{D}^{(2)}$ as the assumed sample distribution over S. By fitting a new threshold function, this time using $\mathcal{D}^{(2)}$ over S and minimizing the weighted misclassification error, we produce h_2. Similar to before based on the results of h_2 we form $\mathcal{D}^{(3)}$ where we increase the weights of the misclassified samples and decrease the weights of the correctly classified samples. This operation is done with respect to $\mathcal{D}^{(2)}$ (and not $\mathcal{D}^{(1)}$) as this is the distribution over which h_2 was produced. Notice that samples where both h_1 and h_2 have correctly classified are now with very low weights in $\mathcal{D}^{(3)}$.

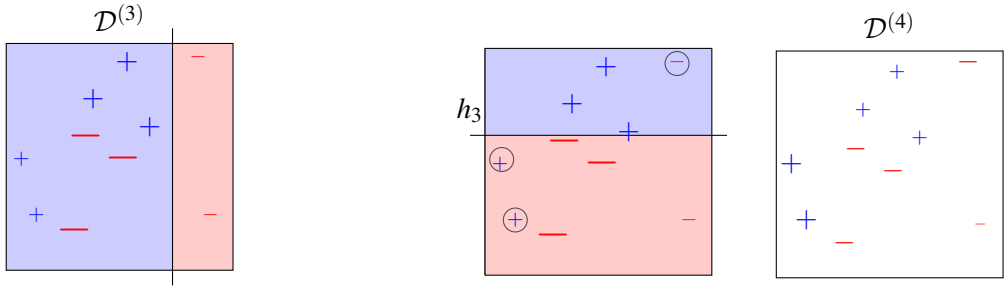


(a) *Current distribution over training sample*

(b) *Fitting h_2 over $\mathcal{D}^{(2)}$ and producing $\mathcal{D}^{(3)}$*

Figure 5.10: Boosting illustration: *second iteration over results of first iteration*

Performing a third iteration the learner this time outputs h_3 which is able to correctly classify the samples misclassified in the previous iteration. Then we update the sample weights distribution based on the classification of h_3. Focusing for example on the bottom right negative sample, notice that all three learners have correctly classified this sample. Therefore, its weight under $\mathcal{D}^{(4)}$ is very low.



(a) *Current distribution over training sample*

(b) *Fitting h_3 over $\mathcal{D}^{(3)}$ and producing $\mathcal{D}^{(4)}$*

Figure 5.11: Boosting illustration: *third iteration over results of second iteration*

Then, if we used the three classifiers produced above h_1, h_2, h_3 and predict based on all of them we achieve a classifier with decision boundaries as in Figure **??**. Even though each single classifier has a simple decision boundary of a threshold function, the ensemble is able to describe much more complex data scenarios. Classifying based on all three classifiers is done by

$$h_{boost}(\mathbf{x}) := sign\left(\sum w_i h_i(\mathbf{x})\right)$$

where w_i are weights given to each classifier and reflect how successful that classifier is.

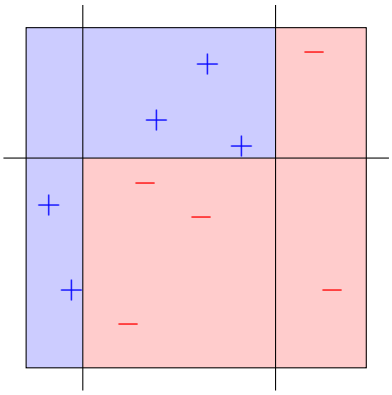


Figure 5.12: Boosting illustraion: *Decision boundaries of the ensemble of classifiers* h_1, h_2, h_3.

5.4.1 AdaBoost Algorithm

The original boosting meta-algorithm is known as **Ada**ptive **Boost**ing. The illustration above follows the operations done by the AdaBoost algorithm.

Algorithm 6 Adaptive **Boost**ing

procedure ADABOOST(training set $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$, base learner $\mathcal{A}$, number of rounds T)
  Set initial distribution to be uniform: $\mathcal{D}^{(1)} \leftarrow \left(\frac{1}{m}, \ldots, \frac{1}{m}\right)$ ▷ Initialize parameters
  for $t = 1, \ldots, T$ **do**
    Invoke base learner $h_t = \mathcal{A}\left(\mathcal{D}^{(t)}, S\right)$
    Compute $\varepsilon_t = \sum \mathcal{D}^{(t)} \mathbb{1}_{y_i \neq h_t(\mathbf{x}_i)}$
    Set $w_t = \frac{1}{2} \ln\left(\frac{1-\varepsilon_t}{\varepsilon_t}\right) = \frac{1}{2} \ln\left(\frac{1}{\varepsilon_t} - 1\right)$.
    Update sample weights $\mathcal{D}_i^{(t+1)} = \mathcal{D}_i^{(t)} \exp\left(-y_i \cdot w_t h_t\left(\mathbf{x}_i\right)\right), \quad i = 1, \ldots, m$
    Normalize weights $\mathcal{D}_i^{(t+1)} = \frac{\mathcal{D}_i^{(t+1)}}{\sum_j \mathcal{D}_j^{(t+1)}} \quad i = 1, \ldots, m$
  end for
  return $h_S\left(\mathbf{x}\right) = sign\left(\sum_{t=1}^T w_t h_t\left(\mathbf{x}\right)\right)$
end procedure

The idea is simple: from iteration t to iteration $t+1$, we want to **increase** the weights of samples misclassified by h_t (where $y_i h_t(\mathbf{x}_i) = -1$) and **decrease** the weights of samples correctly classified by h_t. We want to make the classification problem "maximally hard" in the sense that weighted empirical risk of h_t, with respect to the updated weights $\mathcal{D}^{t+1}$, is the worse possible, namely $1/2$. Finally, the prediction rules vote in the committee with weights w_t.

Claim 5.4.1 For the weighting factor of $w_t = \frac{1}{2} ln\left(\varepsilon_t^{-1} - 1\right)$ and $\varepsilon = \sum_{i=1}^m \mathcal{D}_i^t \cdot \mathbb{1}_{y_i \neq h_t(\mathbf{x}_i)}$ the weighted empirical risk of h_t with respect to $\mathcal{D}^{t+1}$ is $1/2$:

$$\sum_{i=1}^m \mathcal{D}_i^{t+1} \cdot \mathbb{1}_{y_i \neq h_t(\mathbf{x}_i)} = \frac{1}{2}$$

Proof. Directly expressing the weighted empirical risk:

$$\begin{aligned}
\sum_{i=1}^{m} \mathcal{D}_i^{t+1} \cdot \mathbb{1}_{y_i \neq h_i(\mathbf{x}_i)} &= \frac{\sum_{i=1}^{m} \mathcal{D}_i^t \exp\left(-\mathbf{w}_t y_i h_t(\mathbf{x}_i) \mathbb{1}_{y_i \neq h_t(\mathbf{x}_i)}\right)}{\sum_{j=1}^{m} \mathcal{D}_j^t \exp\left(-\mathbf{w}_t y_j h_t(\mathbf{x}_j)\right)} \\
&= \frac{\exp(\mathbf{w}_t \varepsilon_t)}{\exp(\mathbf{w}_t \varepsilon_t) + \exp(-\mathbf{w}_t)(1-\varepsilon_t)} \\
&= \frac{\varepsilon_t}{\varepsilon_t + \exp(-2\mathbf{w}_t)(1-\varepsilon_t)} \\
&= \frac{\varepsilon_t}{\varepsilon_t + \frac{\varepsilon_t}{1-\varepsilon_t}(1-\varepsilon_t)} = \frac{1}{2}
\end{aligned}$$

■

Figure 5.13: Adaboost fitting animation: *Ensemble Methods Examples*

5.4.2 PAC View of Boosting - Weak Learnability

Historically, Boosting appeared as an answer to a fascinating question for which we first need to define **Weak Learnability**:

Definition 5.4.1 — γ-Weak-Learner. A learning algorithm $\mathcal{A}$ is a γ-weak-learner for a hypothesis class $\mathcal{H}$ if there exists a function $m_{\mathcal{H}} : (0,1) \to \mathbb{N}$ such that

- For every $\delta \in (0,1)$
- For every distribution $\mathcal{D}$ over the sample space $\mathcal{X}$
- For every labeling function $f : \mathcal{X} \to \{\pm\}$

if the realizability assumption holds with respect to $\mathcal{H}, \mathcal{D}, f$, then when running $\mathcal{A}$ on a training sample of $m \geq m_{\mathcal{H}}(\delta)$ *i.i.d* samples drawn according to $\mathcal{D}$ and labeled by f, the algorithm returns a hypothesis $h_S = \mathcal{A}(S)$ such that with probability at least $1-\delta$ (with respect to choice of the training sample S), we have $L_{\mathcal{D},f}(h_S) \leq 1/2 - \gamma$.

Definition 5.4.2 A hypothesis class $\mathcal{H}$ is γ-weak-learnable if there exits a γ-weak-learner for $\mathcal{H}$.

How is this different than PAC-learnability? If a hypothesis class $\mathcal{H}$ is PAC-learnable, then for **every** (ε, δ) there exists a learner $\mathcal{A}$. This means that we can learn and generalize a labeling function from $\mathcal{H}$ to any accuracy ε we want. But if $\mathcal{H}$ is γ-weak-learnable, for any δ **and just for** $\varepsilon = 1/2 - \gamma$ there is a learner $\mathcal{A}$. We may not be able to find a learner that has better accuracy (lower ε).

The question that motivated Boosting was the following:

- Suppose that $\mathcal{H}$ is PAC-learnable. Then we know that the rule $ERM_{\mathcal{H}}$ will learn (namely, will be probably approximately correct etc) with a near-minimal number of samples.
- But what if $ERM_{\mathcal{H}}$ is computationally hard? (we've seen examples)
- Assume we can find a **simple** hypothesis class (a "base hypothesis class") $\mathcal{H}_{base}$, such that $ERM_{\mathcal{H}_{base}}$ (choosing the hypothesis in $\mathcal{H}_{base}$ with lowest empirical risk) is computationally efficient, and is γ-weak-learner for $\mathcal{H}$ for some γ.
- This means that we have a computationally efficient way to learn with accuracy $1/2 - \gamma$, for some γ. Maybe we can't find an efficient learner with better γ.
- Is there a way to **boost** $ERM_{\mathcal{H}_{base}}$ in a computationally efficient way, and create a computationally efficient learner $\mathcal{A}$ which is close to minimizing ERM over $\mathcal{H}$?

For example, think about Decision trees. We saw that the ERM learner is not computationally feasible on this hypothesis class. But a small tree may be able to achieve accuracy (over a sample labeled by a larger tree) which is not great, but better than random. Well, as the following theorem shows, Adaboost does just that (for the full proof refer to UML 10.2)

Theorem 5.4.2 Let S be a training set. Assume that at each iteration of Adaboost, the base learner returns a prediction rule (hypothesis h_t) for which the weighted empirical risk satisfies:

$$\sum_{i=1}^{m} \mathcal{D}_i^t \mathbb{1}_{y_i \neq h(\mathbf{x}_i)} \leq \frac{1}{2} - \gamma$$

Then the (standard, non-weighted) empirical risk of the output prediction rule of Adaboost, h_{boost}, (the weighted committee vote) satisfies:

$$L_S(h_{boost}) \equiv \frac{1}{m} \sum_{i=1}^{m} \mathbb{1}_{y_i \neq h_{boost}(\mathbf{x}_i)} \leq \exp\left(-2\gamma^2 T\right)$$

5.4.3 Bias-Variance in Boosting

The hope is, of course, that we are not overfitting, so that low empirical risk will imply low generalization loss. Suppose we run T iterations of Adaboost over a learner $\mathcal{A}_{base}$ that returns a hypothesis from $\mathcal{H}_{base}$. We would like to know what the effective hypothesis class is and how large is it. Adaboost with T iterations will return a

function from the hypothesis class

$$\mathcal{H}_T := \left\{ \mathbf{x} \mapsto \sum_{t=1}^{T} w_t h_t(\mathbf{x}) \,\middle|\, w_t \in [0,\infty),\ \sum_t w_t = 1,\ h_t \in \mathcal{H}_{base} \right\}$$

namely convex combinations of hypotheses from $\mathcal{H}_{base}$. Therefore $\mathcal{H}_T$ becomes larger as T grows. Fortunately it does not grow too fast with T. Suppose for example we have a canonical way to measure the "size" of $\mathcal{H}_T$. It is possible to show that under certain conditions $VCdim(\mathcal{H}_T)$ is roughly $T \cdot VCdim(\mathcal{H}_{base})$. So we can expect Boosting to increase the variance (compared with the base learner) as T increases, but "not too fast".

On the other hand, it is clear that Boosting decreases bias. This is seen from the fact that the empirical risk decreases as T grows, meaning that $\mathcal{H}_T$ is able to come closer and closer to the labeling function on the training set. The fact that empirical risk decreases **exponentially** with T tells us that the bias decreases quite fast. Overall, Boosting typically decreases bias much faster than it increases variance, which is why it typically improves generalization loss quite dramatically. But the question that remains is if we use T too large, will boosting overfit?

Very often we see that boosting ERM over a very simple base hypothesis class is better than boosting ERM over a more complicated class. Figure 5.14 shows the example of the test error of boosting decision stumps (i.e. decision trees with a single split) with Adaboost over number of boosting iterations T, compared with the test errors of a single stump and of a single large decision tree. We are able to see that boosting this very simple base hypothesis class still achieves much better results compared to the very complex hypothesis class of trees with 244 nodes.

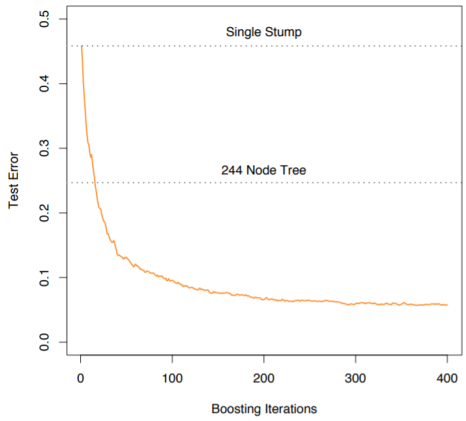


Figure 5.14: *Test error of boosting decision stumps with Adaboost over the number of boosting iterations*

5.5 Summary and Exercises

In this chapter we discussed committee based decisions and saw that a committe of learners using majority vote will achieve better accuracy compared to a single member, given that each member decides better than a random guess. We also saw that this improvement increases as the size of the committee grows but that is bounded by the correlation between the members. To apply this concept we introduced three general methods:

- *Bootstrap:* A method to generate "new" training samples from the one training sample we have.
- *Bagging:* A committee method where we run some base learner against bootstrap samples. Learners are unrelated to each other and all committee members have equal voting weight.

- *Boosting:* A committee method where we run some base learner sequentially on weighted bootstrap samples. Sample weights change between iterations such that samples that the learner misclassified will increase in weight for the next iteration. The committee decision is based on the weighted vote of the members with weights related to their empirical loss.

These methods implement three general principles:

- We can create "artificial" training sets from our one training set S by sampling from S with replacements. This method is known as The Bootstrap. In a typical Bootstrap sample, about a third of the points a left out and others appear more than once.
- We can create a learner with improved accuracy and reduced variance by averaging better than random base learners. When the base learner gets a Bootstrap sample it is called Bagging. Bagging can be done in parallel as each prediction rule is created independently of the others. The prediction accuracy of the Bagging learner improves if the different prediction rules used are as de-correlated as possible. For example Random Forest achieves de-correlation by restricting each split in each tree to a random subset of coordinates.
- We can create a learner with with improved accuracy by boosting a base learner. The key idea behind Boosting is working with a probability distribution over S. Boosting means creating a weighted committee of prediction rules. Rules are created sequentially (not in parallel). Each rule is created from the previous rule by modifying the distribution in such a way that misclassified training samples get an increased weight. For example Adaboost is a Boosting method that uses exponential updates to the probability distribution on S, such that the weighted empirical risk of the previous rule according to the updated distribution is exactly $1/2$ - the worse case scenario.

	Bagging	**Boosting**
Learns committee members	In parallel	Sequentially
Datasets for each member	Bootstrap samples	Weighted bootstrap or original sample with weighted ERM
De-correlation	Recommended	Not necessary
If T is too large	Does not overfit	May overfit
Reduces	Variance	Bias
Committee vote is done	Unweighted	Weighted
Parallel computation implementation	Yes	No
With decision trees use	Deep trees	Shallow trees

Table 5.1: *Comparison of Bagging and Boosting*

Exercises

1. Consider the concept of boosting where we run $\mathcal{A}$ against a training sample with some distribution $\mathcal{D}$. Describe a possible implementation of a decision tree for each of the two interpretations:
   - Using weighted empirical risk: How would you change the CART algorithm to work with a given weight vector $\mathcal{D}^t$ over the training sample S? Hint: what is the best splitting now that we have weights?
   - Using weighted Bootstrap: How would you change the CART algorithm to work with a given weight vector without changing the splitting algorithm, namely, by giving the algorithm a different training sample selected by weighted Bootstrap? Will the algorithm work with repeated samples?
2. Fill out a "Learner ID Card" for the Random Forest classifier specifying the following: hypothesis class; learning principle; computational implementation; making predictions, interpretability; providence of class probabilities; family of models; time complexity of training and predicting; storing the model.

6. Regularization

We have discussed multiple hypothesis classes and learners for choosing an hypothesis $h_s \in \mathcal{H}$ from a given hypothesis class $\mathcal{H}$. In most of these cases choosing h_S was based on minimizing some cost function $\mathcal{F}_S(h)$ over $h \in \mathcal{H}$. This means that $h_S \coloneqq \mathcal{A}_0(S)$ is given by:

$$h_S \coloneqq \underset{h \in \mathcal{H}}{\operatorname{argmin}} \mathcal{F}_S(h)$$

The function $\mathcal{F}$ measures how well h fits the training sample S. We can call $\mathcal{F}_S(h)$ the *fidelity term*. We have seen a few examples for such learners already:

- In linear regression, $\mathcal{F}_S(h)$ measures the *RSS*.
- In logistic regression, $-\mathcal{F}_S(h)$ is the *likelihood* of the model (which we want to maximize).
- In SVM, $-\mathcal{F}_S(h)$ is the *margin* of the hyperplane (which we want to maximize).

With the simplest example for learners that minimize a cost function are of course the *ERM* learners. In any ERM based learner, we define some loss function $\ell(\cdot,\cdot)$ and define the empirical risk induced by ℓ, with respect to the training sample $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ to be:

$$L_S(h) = \frac{1}{m} \sum_{i=1}^{m} \ell(h(\mathbf{x}_i), y_i)$$

For all of these the fidelity term $\mathcal{F}_S(h)$ is the empirical risk $L_S(h)$.

For the different hypothesis classes we have also discussed how their richness and expressiveness governs the bias-variance tradeoff. If the hypothesis class $\mathcal{H}$ is "too large", we are concerned that minimizing $\mathcal{F}$ over $\mathcal{H}$ may lead to over-fitting. Namely, we are concerned our learner will output an hypothesis h_S which will not generalize well, as it is too well adapted to the particular training sample S.

One way to solve this problem would be to restrict $\mathcal{H}$. However, in such a case we are deliberately harming our performance. Instead, we would like to keep $\mathcal{H}$ large, but find a way to tell our learner $\mathcal{A}_0$ "Prefer simpler hypotheses but if you find a complex one that is *really* worth it - take it". To achieve this, we change the

optimization problem that $\mathcal{A}_0$ uses to choose h_S. We introduce an additional term to the problem. We choose $\lambda \geq 0$ and define the learner $\mathcal{A}_\lambda : S \mapsto \mathcal{H}$ by:

$$h_S := \underset{h \in \mathcal{H}}{\operatorname{argmin}} \; \mathcal{F}_S(h) + \lambda \mathcal{R}(h)$$

The term $\mathcal{R}$ is called the *regularization term*. If we choose $\mathcal{R}$ wisely then $\mathcal{R}(h)$ will measure the "complexity" of the hypothesis h. The more complicated the hypothesis h, the larger $\mathcal{R}(h)$ will be.
So we see that in minimizing $\mathcal{F}_S(h) + \lambda \mathcal{R}(h)$ we now have a *trade-off*:

- On one hand, more complicated h, the better it can describe the training sample S, so the fidelity term $\mathcal{F}_S(h)$ will be smaller.
- On the other hand, the more complicated h, the larger $\mathcal{R}(h)$ will be.

And conversely, the simpler h, the larger the fidelity term $\mathcal{F}_S(h)$ (as it won't be able to describe the training sample very well) but the smaller $\mathcal{R}(h)$ will be. So λ is the parameter that governs the trade-off:

- For $\lambda = 0$, we have no regularization and are back to finding a minimizer of $\mathcal{F}_S(h)$.
- For $\lambda \to \infty$, the minimization problem pays no attention to the fidelity term and just wants to find the simplest possible $h \in \mathcal{H}$.
- Any finite value $\lambda \in (0, \infty)$ defines a specific trade-off between the need for fidelity (small $\mathcal{F}_S(h)$) and the need for simplicity of h (small $\mathcal{R}(h)$).

So, when we add regularization to the learner $\mathcal{A}_0 : S \mapsto \operatorname{argmin}_{h \in \mathcal{H}} \mathcal{F}_S(h)$, we won't find a minimizer of $\mathcal{F}_S(h)$ over $\mathcal{H}$. We are hoping that a minimizer of the regularized objective function $\mathcal{F}_S(h) + \lambda \mathcal{R}(h)$ (which does not minimize $\mathcal{F}_S(h)$) will generalize better. This is because, the larger the value of λ, the simpler this minimizer will be.

We therefore get a *family* of learners $\{\mathcal{A}_\lambda\}_{\lambda \in [0,\infty)}$. The regularization parameter λ controls the bias-variance tradeoff: for $\lambda = 0$ we get the most variance and least bias (most complicated h); for $\lambda \to \infty$ we get the least variance and most bias (simplest h).

R We already saw one example for regularization - Soft SVM (subsection 3.3.3). Recall that the Soft-SVM classifier classifies using the half-space $\mathbf{w}^\perp$ where $\mathbf{w}$ is a solution to the optimization problem:

$$\begin{aligned} \text{minimize} \quad & \lambda \|\mathbf{w}\|^2 + \frac{1}{m} \sum_i \xi_i \\ \text{subject to} \quad & y_i \cdot (\langle \mathbf{x}_i, \mathbf{w} \rangle + b) \geq 1 - \xi_i \; \wedge \; \xi_i \geq 0 \end{aligned}$$

Here, the fidelity term is $\|\mathbf{w}\|^2$. The smaller $\|\mathbf{w}\|^2$, the better we fit the training data (in the sense of larger margin). As the total margin is proportional to $1/\|\mathbf{w}\|$, so that minimizing $\|\mathbf{w}\|^2$ means maximizing the margin. The regularization term is $\frac{1}{m}\sum_i \xi_i$. The smaller this term is, the "simpler" the hypothesis since we allow less violations of the margin. Note that here λ is placed on the fidelity term and not on the regularization term.

6.0.1 Regularized Decision Trees

When we presented the decision trees hypothesis class (section 3.7) and the CART algorithm for growing a classification tree we saw how to construct a tree of depth at most k following the ERM learning principle. As we have seen (Figure 3.17) as we increase k and enable deeper trees, though reducing the training error, we unfortunately overfit the training data and increase the generalization error. To avoid overfitting let us apply the concept of regularization to decision trees in the form of pruning: once a tree has been constructed, we remove some of the branches resulting in a less complex decision tree.

Regression Trees

Before introducing pruning as done in CART let us first understand how to adapt the algorithm for *regression trees*. Each classification tree is based on a tree partition of $\mathbb{R}^d = \biguplus_{j=1}^{N} B_j$ into N axis-parallel boxes where each tree $h \in \mathcal{H}_{CT}$ is a function $h : \mathbb{R}^d \to [C]$ defined by:

$$h(\mathbf{x}) = \sum_{j=1}^{N} c_j \mathbb{1}_{\mathbf{x} \in B_j}$$

where $c_j \in [C]$ is the label associated with box j. In the case of regression trees c_j takes values in $\mathbb{R}$ instead of $[C]$.

Just as in the case of classification trees, given a training sample S, we would ideally want to search the hypothesis class of regression trees $\mathcal{H}_{RT}$ for a tree minimizing the empirical risk. As this is computationally hard we must use a heuristic to choose the tree based on the training sample.

Suppose we were given with some tree partition $\mathbb{R}^d = \biguplus_{j=1}^{N} B_j$, then the empirical risk with respect to the squared loss would be minimized by choosing the average response value of training samples in the box: Ex.1

$$c_j \coloneqq \operatorname*{argmin}_{c \in \mathbb{R}} \sum_{i \,|\, \mathbf{x}_i \in B_j} (y_i - c)^2 = \frac{1}{|B_j|} \sum_{i \,|\, \mathbf{x}_i \in B_j} y_i$$

Then, we adjust the CART heuristic (Algorithm 3) for growing regression trees by replacing the misclassification loss by the square loss and the block response assignment to be:

$$\hat{y}_S(B) \coloneqq \frac{1}{|B|} \sum_{i \,|\, \mathbf{x}_i \in B} y_i$$

Pruning a CART Regression Tree

The CART algorithm for growing a regression tree results in a mostly balanced tree. When growing a tree, with each split we do decreases the bias (as we have a more complicated hypothesis, with more expressive power) but also increases the variance (as the labels for prediction in the leaf are averaged over fewer training samples).

The last stage of CART (for both regression- and classification trees) is pruning the grown tree. This means decreasing the tree size by merging together some boxes that were split during the growing stage. By doing so we aim to reach a better bias-variance tradeoff that will lead to better generalization.

Consider a training sample S and regression tree T induced by a partition $\mathbb{R}^d = \biguplus_{j=1}^{N} B_j$. The empirical risk of T on S is given by:

$$L_S(T) = \sum_{j=1}^{N} \sum_{i \,|\, \mathbf{x}_i \in B_j} (y_i - \hat{y}_S(B_j))^2$$

This will be the fidelity term $\mathcal{F}_S(T)$. For the regularization term we define $\mathcal{R}(T)$ to simply be $|T|$ the number of leaves (boxes) in T. In addition, for T_0 a fully grown tree created using the CART heuristic, denote $T \subset T_0$ if the tree T is a sub-tree of T_0 and can be obtained from T_0 by merging boxes in T_0. Then, the regularized optimization problem is given by

$$T^* = \operatorname*{argmin}_{T \subset T_0} L_S(T) + \lambda |T| \tag{6.1}$$

Namely, among all possible sub-trees of T_0 we look for the one optimally balancing empirical risk and hypothesis complexity. Note that we **do not** optimize this objective (6.1) over the entire hypothesis class $\mathcal{H}_{RT}^k$ which would be infeasible. We only consider sub-trees of the tree obtained by CART's greedy splitting, for which there exists efficient algorithms.

The last detail to complete the Random Forest model (subsection 5.3.4) is to incorporate the above pruning strategy into the constructed trees. Each single tree is grown using a heuristic (such as CART) that consists of two stages - growing a full tree and then pruning it. Therefore, we require three tuning parameters: the maximum number of levels, the minimum number of training samples in a leaf (a box) and the regularization parameter λ. When we run bagging on top of the CART algorithm, with the de-correlation trick (restricting each split to a random subset of coordinates), we get a random forest - either for regression or classification. However, in a typical random forest implementation there is no pruning stage. It is simply too computationally expensive to solve the pruning problem (6.1) for each tree if fitting a forest of hundreds of trees or more.

6.0.2 Regularized Regression

6.0.2.1 Subset Selection

Let us revisit linear regression and the least squares estimator. We defined the hypothesis class of linear regression as (2.1):

$$\mathcal{H}_{reg} := \left\{ h(x_1,\ldots,x_d) = w_0 + \sum_{i=1}^{d} x_i w_i \;\middle|\; w_0, w_1, \ldots, w_d \in \mathbb{R} \right\}$$

Then, given a regression problem $\mathbf{X},\mathbf{y}$, we select $h_S \in \mathcal{H}_{reg}$ by finding the least squares estimator, derived using either the ERM learning principle and the squared loss or the Maximum Likelihood learning principle when assuming Gaussian noise. When doing so we assumed that the training sample size m was not smaller than the number of features d, and even preferred that $m \gg d$, since of $m \sim d$ the variance of the least squares estimator can be large.

However, in modern learning problems based on d features, very often we are in a situation where d can be quite large. Linear regression was invented when features were measured and recorded manually. In the last few decades it became very easy to collect features and record them automatically, and a typical regression problem can easily have $d \sim m$ or even $d \gg m$.

When $d \sim m$ or, worse, $d \gg m$, we will have correlated features. The coefficients fitted by linear regression will be poorly determined. For example, a large positive coefficient for some feature can be canceled out by a large negative coefficient for an almost-parallel feature. So the linear regression learner we saw should not be used for $m \sim d$ and cannot be used for $d > m$.

Therefore, we would like to devise a method where we keep only a subset of the features. Let $k \leq d$ be the desired number of features, then we could solve the following optimization problem:

$$\begin{array}{ll} \underset{w_0 \in \mathbb{R}, \mathbf{w} \in \mathbb{R}^d}{\text{minimize}} & ||w_0 \mathbf{1} + \mathbf{X}\mathbf{w} - \mathbf{y}||^2 \\ \text{subject to} & ||\mathbf{w}||_0 \leq k \end{array} \tag{6.2}$$

where $||\mathbf{w}||_0 := \sum_i \mathbb{1}_{\mathbf{w}_i \neq 0}$. That is, find a coefficients vector $\mathbf{w}$ that minimizes the *RSS* under the restriction of using exactly k features. This is exactly what the Best-Subset Selection algorithm does (Algorithm 7). It iterates over all possible combinations of k features, fits for each a least squares model and returns the model (i.e. a vector $\mathbf{w}$) achieving the lowest loss.
As k, the number of features in the model, gives some measure of the complexity of the hypothesis, we can think of the family of learners $\left\{ \mathcal{A}_k^{best-subset} | 0 \leq k \leq d \right\}$. As we change k we move on the bias-variance trade-off. If we restrict k to be small we will find simpler models with higher bias and lower variance and if we restrict k to be large we will find more complex models with lower bias and higher variance. We would like to find a value of k where on the one hand it is large enough to result in a model with low bias (and therefore has more descriptive power) but on the other hand small enough so to avoid high variances.

Unfortunately, even for a single value of k, in order to find the best-subset solution we must go over all

Algorithm 7 Best-Subset Selection

procedure BEST-SUBSET-SELECTION(X, y, k)
  for $S \subset [d]$ where $|S| = k$ **do**
    Fit regression model $\hat{\mathbf{w}}^{(S)}$ and compute RSS $\left(\hat{\mathbf{w}}^{(S)}\right) = \left|\left|\mathbf{y} - \mathbf{X}\hat{\mathbf{w}}^{(S)}\right|\right|^2$
  end for
  return $S^*, \hat{\mathbf{w}}^{(S^*)}$ such that argmin$_{\hat{\mathbf{w}}^{(S)}}$ RSS $\left(\hat{\mathbf{w}}^{(S)}\right)$.
end procedure

$\binom{d}{k}$ subsets of k features. This is an NP-Hard problem and as such cannot be solved efficiently. However, if we change (6.2) to formulate a convex optimization problem with some convex function restricting the complexity of the model in some manner, then perhaps we will be able to find good approximations to the best-subset solution. In order to do so we first change to a related but not equivalent problem that introduces a regularization term over the number of used features. Now we jointly minimize the loss achieved by the model as well as the number of features it effectively contains.

$$\hat{\mathbf{w}}_\lambda^{subset} := \underset{w_0 \in \mathbb{R}, \mathbf{w} \in \mathbb{R}^d}{\operatorname{argmin}} \; ||w_0\mathbf{1} + \mathbf{X}\mathbf{w} - \mathbf{y}||^2 + \lambda \, ||\mathbf{w}||_0 \tag{6.3}$$

where $\lambda \geq 0$. Notice how in both optimization problems (6.2) and (6.3) we do not include the intercept (w_0) in the restriction on the number of features as the intercept is not one of the features per se.

6.0.2.2 Ridge (ℓ_2) Regularization

In addition to the computational challenges of the best-subset selection, it often suffers from a high variance as features are included or excluded based on the single specific training-set we have. To cope with both problems, we use different *shrinkage methods* where we restrict the values of the coefficients, shrinking them (often) towards zero. One such method is the Ridge regression, which imposes a $||\cdot||_2$ penalty on the coefficients:

$$\hat{\mathbf{w}}_\lambda^{ridge} := \underset{w_0 \in \mathbb{R}, \mathbf{w} \in \mathbb{R}^d}{\operatorname{argmin}} \; ||w_0\mathbf{1} + \mathbf{X}\mathbf{w} - \mathbf{y}||^2 + \lambda \, ||\mathbf{w}||_2^2 \quad \lambda \geq 0 \tag{6.4}$$

The regularization parameter $\lambda \geq 0$ is the complexity parameter that controls the amount of shrinkage. For $\lambda = 0$ we get the least squares solution. As $\lambda \to \infty$ we penalize more and more on the size of the coefficients vector, driving the solution $\hat{\mathbf{w}}_\lambda^{ridge} \to 0$. As λ increases the bias increases (we restrict ourselves to specific sets of solutions) and variance decreases (solution is not based solely on training-set) where finally as $\lambda \to \infty$ then $\hat{\mathbf{w}}_\lambda^{ridge} \to 0$.
Notice that the ridge optimization problem is a convex optimization problem and can be solved using some generic QP solver. However, in the case of ridge regression there is no need as we can derive a closed form solution for the minimizer by applying the same strategy as we did for solving the ordinary least squares optimization problem.

Theorem 6.0.1 Let $\mathbf{X}, \mathbf{y}$ be a regression problem and $\lambda \geq 0$. Let $\mathbf{X} = U\Sigma V^\top$ be the SVD of $\mathbf{X}$. The Ridge estimator is given by:

$$\hat{\mathbf{w}}_\lambda^{ridge} = U\Sigma_\lambda V^\top \mathbf{y}, \quad [\Sigma_\lambda]_{ii} = \frac{\sigma_i}{\sigma_i^2 + \lambda}$$

Proof. Let us replace $\mathbf{X}$ with its SVD:

$$\begin{aligned}
\hat{\mathbf{w}}_\lambda^{ridge} &= \left(\mathbf{X}^\top\mathbf{X} + \lambda I\right)^{-1}\mathbf{X}^\top\mathbf{y} &&= \left(\left(U\Sigma V^\top\right)^\top U\Sigma V^\top + \lambda I\right)^{-1}\left(U\Sigma V^\top\right)^\top\mathbf{y} \\
&= \left(V\Sigma U^\top U\Sigma V^\top + \lambda I\right)^{-1}V\Sigma U^\top\mathbf{y} &&= \left(V\Sigma^2 V^\top + \lambda VV^\top\right)^{-1}V\Sigma U^\top\mathbf{y} \\
&= V\left(\Sigma^2 + \lambda I\right)^{-1}V^\top V\Sigma U^\top\mathbf{y} &&= V\Sigma_\lambda U^\top\mathbf{y}
\end{aligned}$$

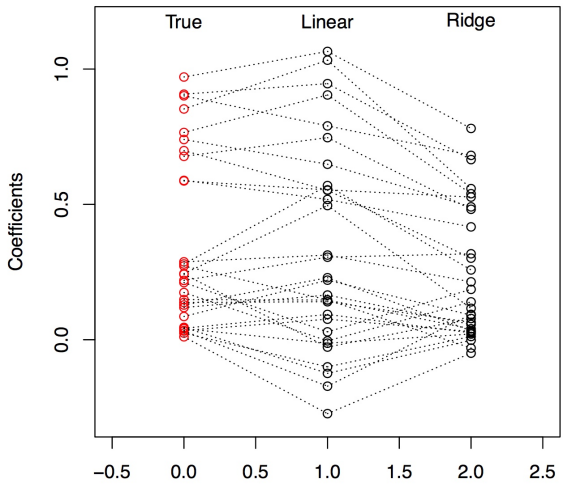


Figure 6.1: *Coefficient shrinkage induced by Ridge regression: Shows the change of the true coefficients of a linear model $y = \mathbf{x}^\top \mathbf{w} + \varepsilon$ between the fitted least squares solution (center) and the ridge solution (right). Source: Ryan Tibshirani Data Mining slides, CMU 36-462/36-662*

where we denote $[\Sigma_\lambda]_{ii} = \frac{\sigma_i}{\sigma_i^2+\lambda}$ and σ_i are the singular values of $\mathbf{X}$. ■

If we choose a strictly positive λ then it can be shown that the Ridge solution takes the form of:

$$\hat{\mathbf{w}}_\lambda^{ridge} := \left(\mathbf{X}^\top \mathbf{X} + \lambda I_d\right)^{-1} \mathbf{X}^\top \mathbf{y}$$

where $\mathbf{X}^\top \mathbf{X} + \lambda I_d$ is a non-singular matrix for any $\lambda > 0$ regardless to $\mathbf{X}^\top \mathbf{X}$ being singular or non-singular. Unlike the least squares estimator which we have seen to be an unbiased estimator (**??**), the ridge estimator is biased. However, it reduces the variance enough such that the overall MSE is lower compared to the least squares solution.

Regularization Path

It is interesting to trace each individual weight $\hat{w}_i$ as λ changes. Figure 6.3 shows the *regularization path* of the ridge solution: it shows how the individual weights change for different values of lambda. Observe how the weights start from the linear regression weights ($\lambda = 0$) and shrink towards zero with a decay roughly like $1/\lambda$ as λ grows. Below the regularization path the losses achieved by the model fitted with each λ are shown separated to the loss of the fidelity term, the regularization term and the joint loss of the fidelity- and regularization terms together.

6.0.2.3 Lasso (ℓ_1) Regularization

By introducing the ridge regularizer we were able to apply linear regression when $d > m$ or when $\mathbf{X}^\top \mathbf{X}$ is not invertible and that it provides a good way to get a simple handle on the bias-variance tradeoff of linear regression. In addition, we saw that even though the ridge estimator is a biased estimator it achieves a lower variance compared to the least squares estimator. However, as evident from the regularization path, the ridge estimator does not do what best-subset does. That is, it cannot **select** a specific subset of features to use for regression.

The *Least Absolute Shrinkage and Selection Operator* (Lasso) attempts to achieve exactly that. This regularization method uses the ℓ_1 norm rather than the ℓ_2 norm:

$$\hat{\mathbf{w}}_\lambda^{lasso} := \underset{w_0 \in \mathbb{R}, \mathbf{w} \in \mathbb{R}^d}{\operatorname{argmin}} \; ||w_0 \mathbf{1} + \mathbf{X}\mathbf{w} - \mathbf{y}||^2 + \lambda \, ||\mathbf{w}||_1 \quad \lambda \geq 0 \tag{6.5}$$

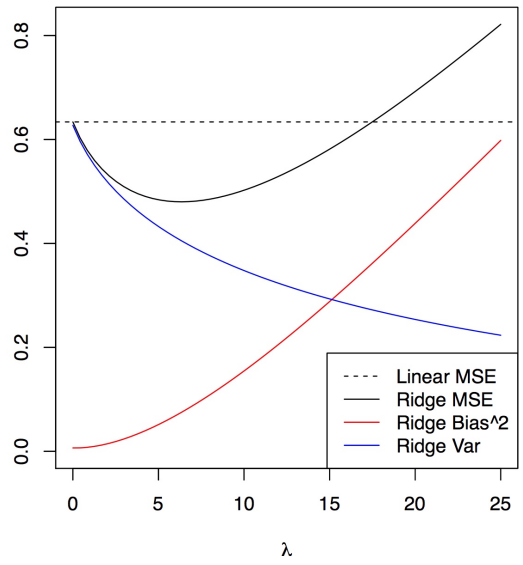


Figure 6.2: Bias, Variance and MSE of Ridge Solution: *as function of λ and compared to the least squares solution. Source: Ryan Tibshirani Data Mining slides, CMU 36-462/36-662*

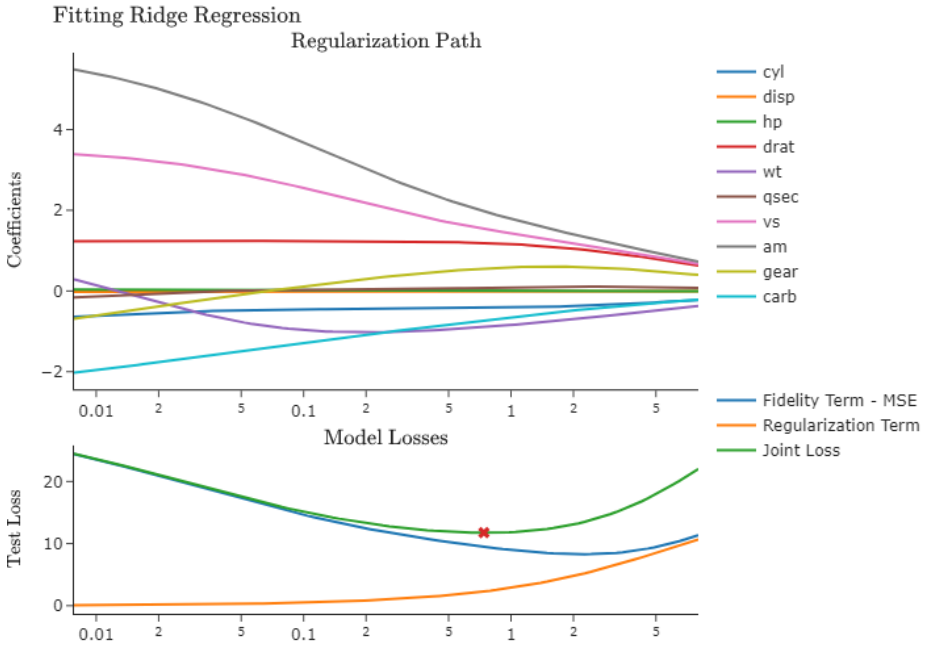


Figure 6.3: Ridge Regularization Path and Losses *as a function of λ over the Motor Trend Car Road Tests dataset. Regularization Examples*

Similar to Ridge, when setting $\lambda = 0$ we get the least squares solution and setting $\lambda \to \infty$ will shrink the coefficients $\hat{\mathbf{w}}_\lambda^{lasso} \to 0$. However the manner of shrinkage is very different between the two problems. The lasso solution is *sparse*. That is, it has zero entries in the vector. Therefore, the larger λ, typically the

more zeros $\hat{\mathbf{w}}_\lambda^{lasso}$ will have, effectively defining a subset of features that are used by the model (a feature corresponding a coefficient of zero is not used by the model). We often refer to this subset of features as the "active set". As we are considering problems where d can be very large, it is hard to interpret the model. Thus, the Lasso has an important advantage in interpretability over Ridge, as it selects for us a subset of variables.

The Lasso optimization problem (6.5) is a convex problem and specifically is a quadratic program. There are however specialized solvers for the Lasso which calculate the entire regularization path at the same computational cost as solving Ridge.

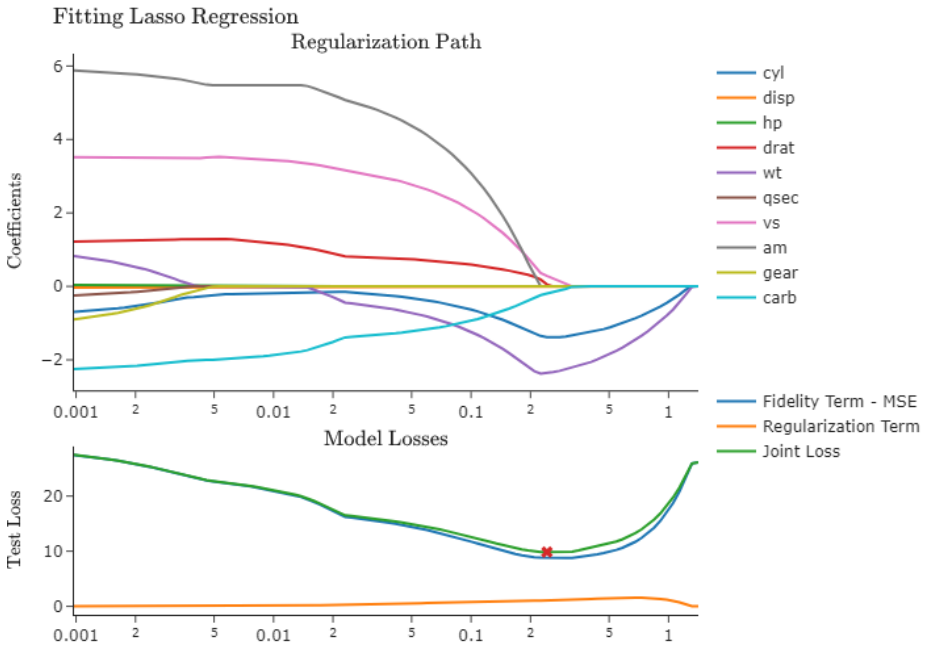


Figure 6.4: Lasso Regularization Path and Losses *as a function of λ over the Motor Trend Car Road Tests dataset. Regularization Examples*

Convexity vs. Sparsity

For the three regression regularization methods using the regularization terms of $||\cdot||_0, ||\cdot||_1, ||\cdot||_2$ (also denoted by ℓ_0, ℓ_1, ℓ_2), we saw that using ℓ_1, ℓ_2 we still have a convex optimization problem whereas for the ℓ_0 term the problem is non-convex. In addition, we stated that unlike ℓ_2, when using ℓ_1 we typically introduce sparsity to the solution. Let us expand this discussion for the L_q family of norms:

$$\text{For } 0 < q \in \mathbb{R}, x \in \mathbb{R}^d \quad ||x||_q := \left(\sum_{i=1}^{d} (x_i)^q \right)^{\frac{1}{q}} \tag{6.6}$$

Specifying any norm in this family as the regression regularization term then:

- Sparsity - For any $q \leq 1$ we typically obtain sparse solutions and therefore our active set of features is smaller than d.
- Convexity - For any $q \geq 1$ the optimization problem (when we use the RSS loss as the fidelity term) is convex. Therefore the problem can be solved efficiently.

The *Lasso* regression uses $q = 1$ and as such achieves **both** sparsity and convexity. But why are $L_{q \leq 1}$ norms obtain their sparsity? This is in fact based on the shapes of their unit balls.

Definition 6.0.1 For a norm $||\cdot||$ on $\mathbb{R}^d$, a ball of radius ρ (around the origin) is the set: $B_\rho := \{\mathbf{w} \in \mathbb{R}^d | ||\mathbf{w}|| \leq \rho\}$. The unit ball of a norm is the ball of radious $\rho = 1$.

For $L_{q>1}$, such as in the case of the Euclidean norm, the unit ball has no corners. In contrast, the unit balls of $L_{q\leq 1}$ have corners. Specifically, the unit ball of the ℓ_1 norm over $\mathbb{R}^d$ has $2d$ corners.

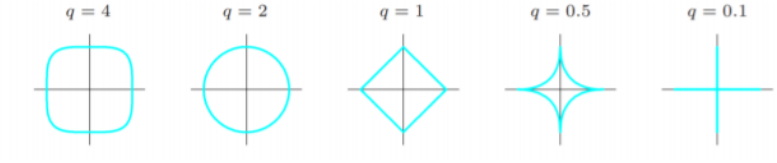


Figure 6.5: Unit Balls of Different L_q Norms: *Source: Elements of Statistical Learning, Figure 3.12*

Now, let us revisit the Ridge and Lasso optimization problems. Instead of the unconstrained forms seen in 6.5 and 6.4 consider the equivalent constraint versions:

minimize $w_0 \in \mathbb{R}, \mathbf{w} \in \mathbb{R}^d$	$\|\|w_0\mathbf{1} + \mathbf{X}\mathbf{w} - y\|\|^2$	minimize $\mathbf{w}_0 \in \mathbb{R}, \mathbf{w} \in \mathbb{R}^d$	$\|\|w_0\mathbf{1} + \mathbf{X}\mathbf{w} - y\|\|^2$
subject to	$\|\|\mathbf{w}\|\|_2^2 \leq \rho$	subject to	$\|\|\mathbf{w}\|\|_1 \leq \rho$

If we fix some $\rho \in \mathbb{R}$ we can ask where in $\mathbb{R}^d$ will we find the $\widehat{\mathbf{w}}_\rho^{ridge}$ and $\widehat{\mathbf{w}}_t^{lasso}$ solutions. As the fidelity term is a quadratic form in $\mathbf{w}$, its level sets are ellipsoids. Suppose ρ is very large, so there is effectively no constrain, the solution of the minimization problem will be the least squares solution. As we restrict the solution more and more, by making ρ smaller and smaller, we are limiting the solution to be inside the norm ball. By definition, the solution will be found where one of the level sets of the fidelity term intersects with the ball of radius ρ. When we consider the $L_{q\leq 1}$ norms (such as the ℓ_1 in Lasso) this intersection typically happens at one of the corners of the norm ball. As these corners take place on the axes they correspond to **sparse** solutions.

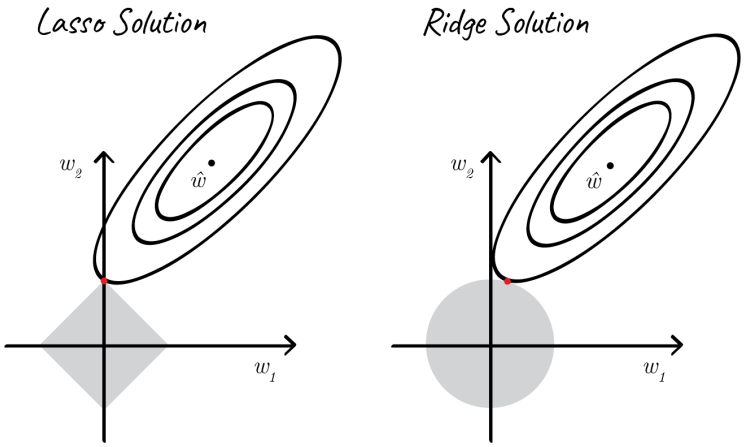


Figure 6.6: The constrained optimization problems in $\mathbb{R}^2$: *With the Lasso problem with the ℓ_1 unit ball and Ridge problem with the ℓ_2 unit ball. The red dots represent the coefficients vector that is both on the unit ball and the objective function.*

6.0.2.4 The Orthogonal Design Case

Here is another way to understand why Lasso solutions are sparse. Consider the case where all features are orthogonal to each other, namely $\mathbf{X}^\top\mathbf{X} = I_d$. In some sense, this is the simplest setup for regression problems, where we can write $\mathbf{y}$ as a linear combination of *orthonormal* vectors. This is also known as an orthogonal design. In such setup, we have a closed form solution for $\hat{\mathbf{w}}_\lambda^{subset}, \hat{\mathbf{w}}_\lambda^{ridge}, \hat{\mathbf{w}}_\lambda^{lasso}$ based on the $\hat{\mathbf{w}}^{LS}$ solution.

Let us define two *thresholding* functions.

Definition 6.0.2 Let the hard- and soft-threshold (at λ) be the functions $\eta_\lambda^{hard}, \eta_\lambda^{soft} : \mathbb{R} \to \mathbb{R}$ defined by:

$$\eta_\lambda^{hard} := \mathbb{1}_{|x|\geq\lambda} \cdot x \qquad \eta_\lambda^{soft} := \text{sign}(x)\,[|x| - \lambda]_+ = \begin{cases} x-\lambda & x \geq \lambda \\ 0 & |x| < \lambda \\ x+\lambda & x \leq -\lambda \end{cases}$$

These functions define a manner of shrinking an input value x towards 0, depending on λ. The hard-thresholding zeros inputs in the range of $(-\lambda, +\lambda)$ and leaves inputs outside of this range unchanged. The soft-thresholding shrinks inputs out of the $(-\lambda, +\lambda)$ range, and zeros inputs that are within the range.

Claim 6.0.2 Let X, y be an orthogonal design matrix and a response vector. Denote $\widehat{w}$ the OLS solution. The lasso regularization optimization problem takes the form of $\widehat{w}^{lasso}(\lambda) = \eta_\lambda^{soft}(\widehat{w})$

Proof. Recall that the least squares solution is $\hat{\mathbf{w}} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}\mathbf{y} = \mathbf{X}\mathbf{y}$. We can re-write the objective function as:

$$\begin{aligned} f_{\ell_1}(\mathbf{w}) = \tfrac{1}{2}\,||\mathbf{y}-\mathbf{X}\mathbf{w}||_2^2 + \lambda\,||\mathbf{w}||_1 &= \tfrac{1}{2}\left(||\mathbf{y}||^2 - 2\mathbf{y}^\top\mathbf{X}\mathbf{w} + \mathbf{w}^\top\mathbf{X}^\top\mathbf{X}\mathbf{w}\right) + \lambda\,||\mathbf{w}||_1 \\ &= \tfrac{1}{2}\left(||\mathbf{y}||^2 + \left(\mathbf{w}^\top - 2\hat{\mathbf{w}}^\top\right)\mathbf{w}\right) + \lambda\,||\mathbf{w}||_1 \\ &= \tfrac{1}{2}\,||\mathbf{y}||^2 + \textstyle\sum_{j=1}^d \left(\tfrac{1}{2}w_j^2 - \hat{w}_j w_j + \lambda\,|w_j|\right) \\ &= \tfrac{1}{2}\,||\mathbf{y}||^2 + \textstyle\sum_{j=1}^d \left(\tfrac{1}{2}w_j - \hat{w}_j + \lambda\,sign(w_j)\right) w_j \end{aligned}$$

Let us minimize the expression for each w_j taking into account the value of λ. So:

$$\frac{\partial\frac{1}{2}||\mathbf{y}-\mathbf{X}\mathbf{w}||+\lambda||\mathbf{w}||_1}{\partial w_j} = \frac{\partial\left(\frac{1}{2}w_j - \hat{w}_j + \lambda sign(w_j)\right)w_j}{\partial w_j} = w_j - \hat{w}_j + \lambda\,sign(w_j) = 0$$
$$\Downarrow$$
$$w_j = \hat{w}_j - \lambda\,sign(w_j)$$

Let us divide into cases:

- If $\left|\hat{w}_j\right| < \lambda$ then $w_j = 0$. That it because:
  - If $w_j < 0$ then $0 > w_j = \hat{w}_j + \lambda$, which $\forall\left|\hat{w}_j\right| < \lambda$ the right-hand side is positive in contradiction.
  - If $w_j > 0$ then $0 < w_j = \hat{w}_j - \lambda$, which $\forall\left|\hat{w}_j\right| < \lambda$ the right-hand side is negative in contradiction.
- If $\hat{w}_j \geq \lambda$ then $w_j = \hat{w}_j - \lambda\,sign(w_j)$ which is valid only for $w_j \geq 0$ which means that $w_j = \hat{w}_j - \lambda$.
- If $\hat{w}_j \leq -\lambda$ then $w_j = \hat{w}_j - \lambda\,sign(w_j)$ which is valid only for $w_j \leq 0$ which means that $w_j = \hat{w}_j + \lambda$.

Putting it all together we got that:

$$w_j(\hat{w}_j, \lambda) = \begin{cases} \hat{\mathbf{w}} - \lambda & \hat{\mathbf{w}} \geq \lambda \\ 0 & |\hat{\mathbf{w}}| < \lambda \\ \hat{\mathbf{w}} + \lambda & \hat{\mathbf{w}} \leq \lambda \end{cases} \implies \hat{w}_j^{lasso}(\lambda) = \eta_\lambda^{soft}(\hat{w}_j)$$

■

Similarly, we can obtain the Best-Subset and Ridge solutions:

$$\begin{aligned}\hat{\mathbf{w}}_{\lambda}^{subset} &:= \eta_{\sqrt{\lambda}}^{hard}\left(\hat{\mathbf{w}}^{LS}\right)\\ \hat{\mathbf{w}}_{\lambda}^{ridge} &:= \hat{\mathbf{w}}^{LS}/\left(1+\lambda\right)\end{aligned}$$

Namely, in the orthogonal design setup we can obtain the different solutions by applying a *univariate shrinkage function* to each coordinate of $\hat{\mathbf{w}}^{LS}$ separately. Observe that in this case:

- The Best-Subset sets some coefficients to zero and leaves the rest untouched.
- The Lasso solution sets some coefficients to zero and shrinks the rest by λ.
- The Ridge solution simply multiplies by a scalar.

Therefore, for the Best-Subset and Lasso solutions, the solutions' sparsity grows as λ grows.

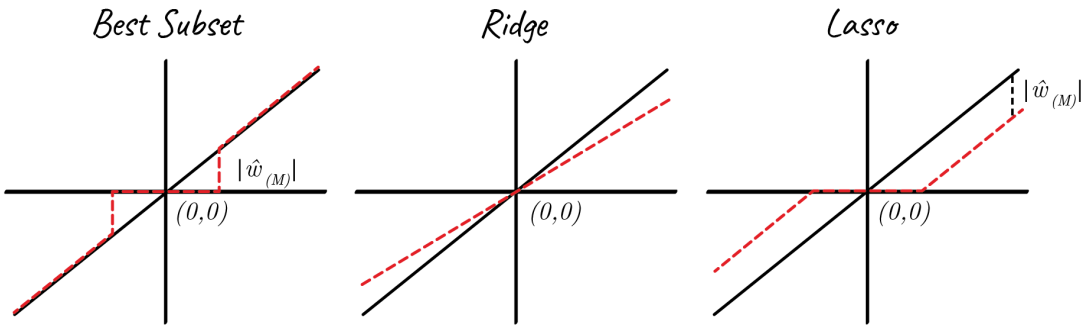


Figure 6.7: Shrinkage of Best-Subset, Ridge and Lasso in the orthogonal design case

6.0.3 Regularized Logistic Regression

We have seen a few different (linear) regression methods using regularization. We can also apply these regularization terms to other regression models such as the logistic regression. Similarly to linear regression, if $m \sim d$ or $d > m$, also for logistic regression optimization will be numerically unstable, might have parallel feature vectors with large coefficients of opposite signs and hard for interpretation. All these problems are alleviated by adding a regularization term.

Recall that the logistic regression classifier find the coefficients vector by solving:

$$\hat{\mathbf{w}} := \underset{w_0\in\mathbb{R},\mathbf{w}\in\mathbb{R}^d}{\operatorname{argmax}} \sum_{i=1}^{m}\left[y_i\left(\langle\mathbf{x}_i,\mathbf{w}\rangle + w_0\right) - log\left(1+e^{w_0+\langle\mathbf{w},\mathbf{x}_i\rangle}\right)\right]$$

We can add the ℓ_1 regularization term to the above fidelity term to obtain the ℓ_1-regularized logistic regression classifier:

$$\hat{\mathbf{w}} := \underset{w_0\in\mathbb{R},\mathbf{w}\in\mathbb{R}^d}{\operatorname{argmax}} \overbrace{\sum_{i=1}^{m}\left[y_i\left(\langle\mathbf{x}_i,\mathbf{w}\rangle + w_0\right) - log\left(1+e^{w_0+\langle\mathbf{w},\mathbf{x}_i\rangle}\right)\right]}^{\mathcal{F}_S(\mathbf{w})} + \lambda\overbrace{||\mathbf{w}||_1}^{\mathcal{R}(\mathbf{w})}$$

This is still a convex optimization problem, and fast specialized solvers are available. The ℓ_1-regularized logistic regression classifier is a very powerful classifier over $\mathcal{X} = \mathbb{R}^d$. It has low variance, we are able to control the bias-variance tradeoff (by choosing λ) and is it very interpretable.

6.1 Summary and Exercises

1. Let $x_1, \ldots, x_m \in \mathbb{R}$ be a set of real values. Show that the minimizer μ of the sum of square distances $\sum (x_i - \mu)^2$ is given by the sample mean:

$$\frac{1}{m}\sum x_i = \operatorname*{argmin}_{\mu \in \mathbb{R}} \sum (x_i - \mu)^2$$

2. Let $\mathbf{X} \in \mathbb{R}^{m \times d}$. Show that for all $\lambda > 0$ then $\mathbf{X}^\top \mathbf{X} + \lambda \cdot I_d$ is non-singular.
3. Let $\mathbf{X}, \mathbf{y}$ be a regression problem with Gaussian errors $\mathbf{y} = \mathbf{X}\mathbf{w} + \varepsilon$, $\varepsilon \sim \mathcal{N}\left(0, \sigma^2 I_m\right)$ and suppose $\mathbf{X}^\top \mathbf{X}$ invertible.
   - Show that $\hat{\mathbf{w}}_\lambda^{ridge} = A_\lambda \hat{\mathbf{w}}$ where $A_\lambda = \left(\mathbf{X}^\top \mathbf{X} + \lambda I_d\right)^{-1} \left(\mathbf{X}^\top \mathbf{X}\right)$. Conclude that $\forall \lambda > 0$ then $\hat{\mathbf{w}}_\lambda^{ridge}$ is a biased estimator for $\mathbf{w}$.
   - Recall that for any non random matrix B and a random vector $\mathbf{z}$ then $Var(B\mathbf{z}) = BVar(\mathbf{z})B^\top$. Show that the variance of the least squares estimator is given by $Var(\hat{\mathbf{w}}) = \sigma^2 \left(\mathbf{X}^\top \mathbf{X}\right)^{-1}$ and the variance of the ridge solution is given by $Var(\hat{\mathbf{w}}_\lambda^{ridge}) = \sigma^2 A_\lambda \left(\mathbf{X}^\top \mathbf{X}\right)^{-1} A_\lambda^\top$.
   - Show that the MSE of the ridge solution is lower than the MSE of the least squares solution.
4. Let $\mathbf{X}, \mathbf{y}$ be a regression problem of orthogonal design and let LS be the least squares solution. Show that the the Ridge and Best-Subset estimator take the form of:

$$\begin{aligned} \hat{\mathbf{w}}_\lambda^{subset} &:= \eta_{\sqrt{\lambda}}^{hard}\left(\hat{\mathbf{w}}^{LS}\right) \\ \hat{\mathbf{w}}_\lambda^{ridge} &:= \hat{\mathbf{w}}^{LS}/(1+\lambda) \end{aligned}$$

7. Model Selection & Evaluation

So far, we have discussed several learning algorithms and meta-algorithms that can be applied over the existing learning algorithms. In the following we will be answering the following questions:

- How to **select** a model? For different algorithms we have described the existance of a family of learners where we defined some "tuning" hyper-parameter. For example, k the number of neighbors used in the k-NN algorithm; the maximal tree depth and pruning regularization in CART; or the regularization lambda in soft-SVM and the different regularized linear- and logistic- regression problems.

  In the case of meta-algorithms such as bagging or boosting, in addition to the base learners' tuning parameters, we need to choose the number of bagging/bootstrapping iterations B. If we are using de-correlation in bagging, there might be additional tuning parameters (e.g. the number k of allowed coordinated for a split in the Random Forest algorithm).

- How to **estimate** the performance of a chosen model? Before applying the chosen model on new samples we would like to estimate how will it perform on a new independent set of samples. If our estimation of the generalization error for the selected model is poor, perhaps we are working with the wrong learning algorithm for our problem, and should therefore select a different candidate. In addition, often we are interested in knowning how our chosen learner will perform before we begin using it.

The performance we would like to estimate, which would also influence the selection of the model, is the generalization error. Assume some loss function $\ell(\cdot,\cdot)$ then we defined the empirical risk of an hypothesis $h : \mathcal{X} \to \mathcal{Y}$ simple as the average loss over the training sample:

$$L_S(h) \coloneqq \frac{1}{m}\sum_{i=1}^{m} \ell\left(h\left(\mathbf{x}_i\right), y_i\right) \quad S = \{(\mathbf{x}_i, y_i)\}_{i=1}^{m}$$

The generalization error (also referred to as *test error*), that is the predicted error over an independent sample set depends on our data-generation model:

- No data-generation model: If we are not assuming any model to describe how the data is generated we simply define the generalization error as the average loss over the test set:

$$L(h) \coloneqq \sum_{i=1}^{|T|} \ell(h(\mathbf{x}_i), y_i)$$

- Probability distribution over $\mathcal{X}$ and unknown labeling function $f : \mathcal{X} \to \mathcal{Y}$: When assuming a model as the PAC model, the generalization error is the expected error over sampling from the distribution:

$$\mathbb{E}_{x \sim \mathcal{D}} [\ell(h(\mathbf{x}), f(\mathbf{x}))]$$

- Probability distribution over $\mathcal{X} \times \mathcal{Y}$: When assuming a model as the Agnostic PAC model, the generalization error is the expected error over sampling a sample-label pair from the distribution:

$$L_{\mathcal{D}}(h) \coloneqq \mathbb{E}_{(\mathbf{x},y) \sim \mathcal{D}} [\ell(h(\mathbf{x}), y)]$$

Once we understand what is correct generalization error, the next step is to find a way to *correctly estimate* it. Suppose we perform the following procedure: given a supervised batch learning setting, where our training sample is $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^{m}$, let us perform model selection and -evaluation over a family of learning algorithms $\{\mathcal{A}_\alpha\}$ simply by using S:

Algorithm 8 Model Selection & Evaluation

1: **procedure** SELECT-AND-EVALUATE(S, $\{\mathcal{A}_\alpha\}$)
2: Train each model over S to obtain $\{h_\alpha \coloneqq \mathcal{A}_\alpha(S)\}$.
3: Choose the best model by using $\alpha^* \coloneqq \text{argmin}_\alpha L_S(h_\alpha)$
4: Estimate generalization error of h_{α^*} over train set $L_{\mathcal{D}}(h_{\alpha^*}) \coloneqq L_S(h_{\alpha^*})$.
5: **return** $\alpha^*, h_{\alpha^*}, L_S(h_{\alpha^*})$
6: **end procedure**

This procedure will yield poor results (Figure 7.1). The selection is based on the empirical error of the model. As we have already seen in the bias-variance trade-off, the generalization error can be represented as the sum of both the bias- and the variance of the model. The model we select α^* will adapt as much as possible (under the restriction to some $\mathcal{H}$) to the training data. The more variance the learner has, the more it will be able to adapt to the particular properties of S. Our generalization error estimator will suffer from *optimism* - which is the technical term for the difference between the empirical risk and generalization error). Thus, we need to devise a different method to estimate the generalization error. One that will suffer less from optimism.

7.0.1 Train-Validation-Test Scheme

The naive approach of using the finite training set S both for training the model and estimating it's future performance yields poor results. Suppose we had infinitely many samples, could we have done better? In such scenario we could use three different sets S (training), V (validation) and T (testing). Then we would perform model selection and evaluation as seen in Algorithm 9.

The introduction of a third set of samples, the validation set, which is used to select the final model decouples the training- from the model selection- stages. This provides us with an unbiased estimator of the generalization error which we can also use to bound the generalization error.

Claim 7.0.1 Let $h_S \coloneqq \mathcal{A}(S)$ be the hypothesis returned by a learner over the training sample S, and let V be a new ("fresh") *i.i.d* samples from $\mathcal{D}$. The empirical risk of h_S over V is an unbiased estimator of the

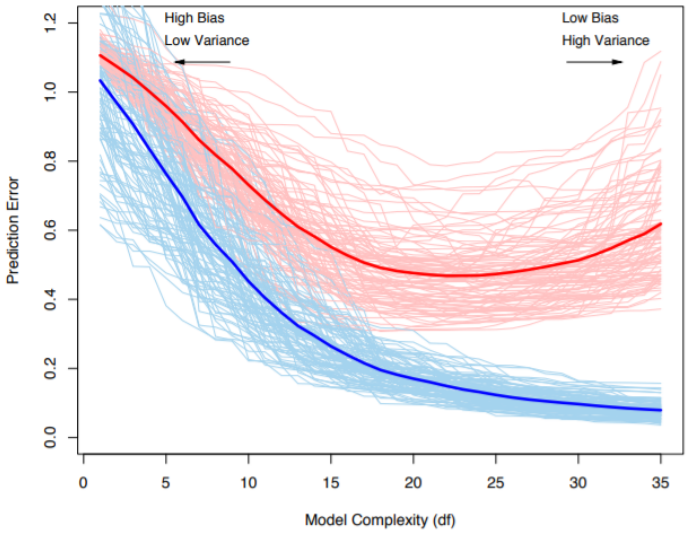


Figure 7.1: Train- and generalization Error: *as function of model complexity. For each pair of random train set and test set the blue line shows the train error as a function of model complexity while the red line shows the test error as a function of model complexity .Source: Elements of Statistical Learning*

Algorithm 9 Model Selection & Evaluation

procedure SELECT-AND-EVALUATE(S, V, T $\{\mathcal{A}_\alpha\}$)
    Train each model over S to obtain $\{h_\alpha := \mathcal{A}_\alpha(S)\}$.
    Choose the model minimizing the loss over the *validation set*: $\alpha^* := \text{argmin}_\alpha L_V(h_\alpha)$
    Estimate generalization error of h_{α^*} over *test set*: $L_{\mathcal{D}}(h_{\alpha^*}) := L_T(h_{\alpha^*})$.
    return $\alpha^*, h_{\alpha^*}, L_T(h_{\alpha^*})$
end procedure

generalization error of h_S.

Proof. Denote $m_V = |V|$. As the sample are identically distributed then:

$$\mathop{\mathbb{E}}_{V\sim\mathcal{D}^{m_V}}[L_V(h_S)] = \frac{1}{m_V}\sum_{i=1}^{m_V}\mathbb{E}_{(\mathbf{x}_i,y_i)\sim\mathcal{D}}[\ell(h_S,(\mathbf{x}_i,y_i))] = L_{\mathcal{D}}(h_S)$$

■

For the following, let us assume that the loss function is *bounded*. If the loss function is unbounded, the generalization error cannot be bounded. Suppose the loss function is bounded by 1.

Corollary 7.0.2 The generalization error of h_S is bound by:

$$\mathbb{P}\left[|L_V(h_S) - L_{\mathcal{D}}(h_S)| \leq \sqrt{\frac{\log(2/\delta)}{2m_V}}\right] \geq 1-\delta$$

Proof. Recall from Hoeffding's inequality that if $X_1,\dots,X_m$ are a set of bounded iid random variables such that $0 \le X_i \le 1$ and $\overline{X} = \frac{1}{m}\sum X_i$, then: $\mathbb{P}\left[\left|\overline{X} - \mathbb{E}\left[\overline{X}\right]\right| \ge \varepsilon\right] \le 2\exp\left(-2m\varepsilon^2\right)$. As V is a set of iid samples denote

$$Z_i \coloneqq (\mathbf{x}_i, y_i)$$

the random variable of the selected pair. Since the samples are iid then $Z_1,\dots,Z_{m_V}$ are iid too. Next, denote

$$X_i \coloneqq \ell\left(h_S, Z_i\right)$$

the random variable of the loss over the Z_i sample. It holds that $X_1,\dots,X_{m_V}$ are a set of iid random variables such that $0 \le X_i \le 1, \quad i = 1,\dots,m_V$. Notice, that for $\overline{X} = \frac{1}{m_V}\sum X_i = L_V\left(h_S\right)$, as $L_V\left(h_S\right)$ is an unbiased estimator of the generalization error, we directly get that:

$$\mathbb{P}\left[\left|L_V\left(h_S\right) - L_{\mathcal{D}}\left(h_S\right)\right| \ge \varepsilon\right] \le 2\exp\left(2m_V\varepsilon^2\right)$$

By setting $\varepsilon = \sqrt{\frac{1}{2m_V}\ln\frac{2}{\delta}}$ we conclude that:

$$\mathbb{P}\left[\left|L_V\left(h_S\right) - L_{\mathcal{D}}\left(h_S\right)\right| \le \sqrt{\frac{\log\left(2/\delta\right)}{2m_V}}\right] \ge 1 - \delta$$

■

7.0.2 Cross Validation

In reality, splitting our finite set of samples into three sets is problematic. In the great majority of cases we are unwilling to decrease the size of our training set. We have already seen that training over a smaller set yields inferior results. Therefore, designating an additional portion of our limited number of samples just for validation isn't feasible. Instead we would like to come up with some method to use S for training but still do proper model selection and -evaluation.

Potentially the simplest method to do so is cross-validation (CV) (Algorithm 10). Instead of thinking of S as a single set, let us think of it as a disjoint union of K equality sized sets, named folds. Then, for each $k = 1,\dots,K$, we fit a model using all samples of S **except** samples belonging to the k'th fold. We then use the k'th fold to calculate the prediction error of the fitted model. Finally we report the estimated generalization error across all K folds. This method is called k-fold Cross Validation. The k'th fold, which is not used for training but only for evaluating the trained model functions as an unbiased estimator for the generalization error.

Algorithm 10 Cross-Validation

procedure k-FOLD-CROSS-VALIDATION(S, k $\mathcal{A}_\alpha$)
  Randomly partition S to k disjoint subsets $S = \biguplus_{i=1}^{k} S_i$.
  for $i = 1,\dots,k$ **do**
    Train the model S except the i'th fold $S \setminus \{S_i\}$.
    Calculate the loss of the model on S_i functioning as a test set.
  end for
  return the estimated mean and standard deviation of the k losses obtained.
end procedure

Cross-Validation is the most popular method for choosing tuning parameters. For each candidate $\mathcal{A}_\alpha$, we train it k times according to the CV procedure, each time leaving out one of the k folds. Then we choose α (and as

such the learner $\mathcal{A}_\alpha$) whose average error (over the k validation sets) was lowest. Once we finished the model selection phase using CV, and have a fully specified learner $\mathcal{A}$, we train $\mathcal{A}$ **again** on the entire training sample S. This way we are still able to train $\mathcal{A}$ on as many points as possible.

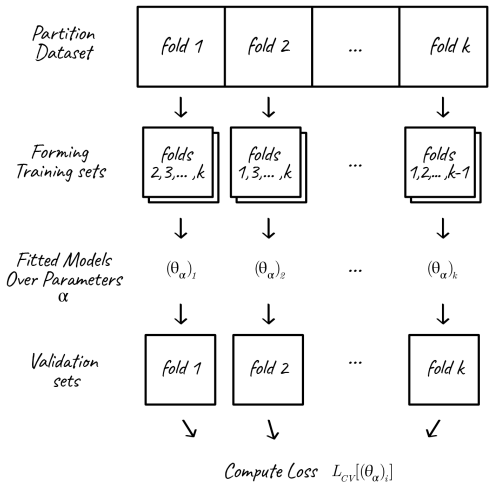


Figure 7.2: Schematics of Cross-Validation: *showing the partitioning of the dataset, training on all folds except the i'th fold to obtain estimators $(\theta_\alpha)_i$ and compute the cross-validation loss over the i'th fold left out.*

Cross validation is also used for model evaluation. Once we have a final, fully specified learner $\mathcal{A}$, we run k-fold CV and report the average error and standard deviation over the k folds. In this manner we supply both an estimation of the generalization error and a measure of accuracy for this estimate.

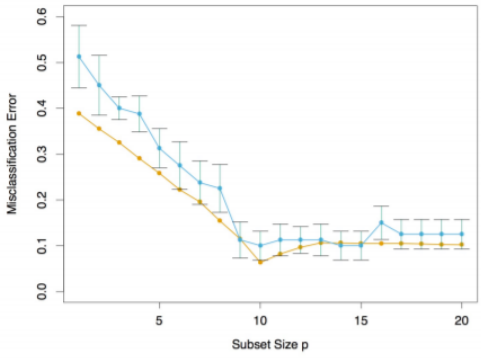


Figure 7.3: Generalization- and Cross-Validation errors as function of complexity: *Generalization error in orange and 10-fold cross-validation with errorbars in blue. Source: Elements of Statistical Learning*

Choosing k, the number of folds

By using CV we have now introduced another "hyper-hyper"-parameter. How should we choose the value for k? If $k = 1$ then there is no CV. If $k = 2$ we split the data into two equally sized folds where we train only one half of the data. Therefore, the reported CV error will be larger than the true generalization error (also known as out-of-sample error). If $k = m$, the sample size, it is called **leave-one-out** CV. Generally, if k is small we may be training on a dataset too small. As such the CV error may be biased upwards. On the other hand, if k is large, the training samples are very similar to each other. As such the trained models are highly correlated, which might introduce high variances.

R Another consideration when using CV is computational. Since we train each model k times, the larger k, the more computations we perform.

7.0.3 Bootstrap For Estimating Generalization Error

In subsection 5.3.1 we introduced the idea of Bootstrap, where by using re-sampling with replacement we can seemingly create new datasets. This method can also be used for estimating the generalization error using just the single training sample S. To do so let $\mathcal{A}_\alpha$ be some learner and $B \in \mathbb{N}$ the number of bootstrap samples to create. Very similar to the CV approach we could now:

- Draw a bootstrap sample $S^{(b)}$ by sampling m samples from S with replacement.
- By sampling $S^{(b)}$ we also obtain an independent test set $T^{(b)} := S \setminus S^{(b)}$ being the samples not chosen in the b-th bootstrap sample. These samples are also called the **out-of-bag** samples.
- Train $\mathcal{A}$ over $S^{(b)}$ to acquire an hypothesis h_S and test it over $T^{(b)}$.
- Finally, report the estimated mean and standard deviation of the generalization error, as measured over the B test sets.

7.0.4 Common Mistakes When Performing Model Selection

There are two very common mistakes when performing model evaluation using either of the methods seen before. The first causes over-estimation of the generalization error while the other causes under-estimation of the generalization error.

7.0.4.1 Over-estimating Generalization Error

Consider a family of learners $\{\mathcal{A}_\alpha\}$ over which cross-validation was used in order to determine α and obtain the fully specified learner $\mathcal{A}$. When each family member was trained it was done using a training set smaller than S. For k the number of folds used and m the number of samples in S, each model was trained using $m(k-1)/k$ samples.

Now, recall that successful learning depends on the number of training samples. When discussing PAC theory we defined the sample complexity as the minimal number of samples required to learn an hypothesis from a given hypothesis class, given some ε, δ. If m samples is a sufficient training size, but $m(k-1)/k$ is not, we cannot guarantee the bounds over the generalization error. In such cases we will estimate the generalization error **too high**, namely over-estimate it.

To better determine the number of folds, we would like to have an hypothetical learning curve (Figure 7.4) for the model in question. Such a curve will capture the essence of the sample complexity function. Given a certain number of training samples the curve shows the expected success rate. Using think curve over a training set S with m samples, we can calculate the effective training set size when using k-fold cross-validation for a certain k. If the effective training size remains in the saturated area of the graph the estimated generalization error using cross-validation would not differ by much from the real generalization error. However, if the effective training size is where the slope of the graph is steep, we do not have a sufficient amount of training samples and will suffer from over-estimating the generalization error.

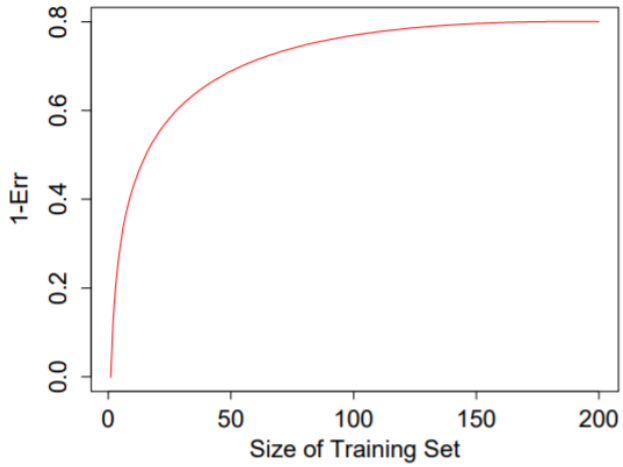


***Figure 7.4: Learning Curve of classifier**: showing the success ($1 - Err$) as a function of the training set size*

7.0.4.2 Under-estimating Generalization Error

A much more common mistake is being too optimistic in estimating the generalization error, causing under-estimation of it. Suppose we are given a learning problem and a dataset with m samples. We begin trying different types of models, each with its own tuning parameters. Perhaps we also alter the training sample S by removing- or adding features and addressing "problematic" samples. Eventually we have a "clean" training sample and a tuned model that works well. Now we perform model evaluation to estimate the generalization error of the chosen learner.

In such a scenario we will end up under-estimating the generalization error due to two mistakes:

- The first is known as **model snooping**. By trying out many learners, tuning the parameters of each and looking for one model that will perform very well, we slowly begin overfitting to the training sample. Each time we discarded some potential hypothesis class in favor of another that performed better we introduced more bias to the procedure. Even in the case where we use a validation set, if we evaluate the performance over it many times, we begin overfitting to it as well.
- The second is known as **data snooping**. Once we evaluate our selected model over a new test sample (or perhaps even when using it in production) this data did not under-go the same level of treatment. It might contain missing features or "problematic" samples that were dealt with in S. Therefore, the cross-validation procedure will under-estimate the generalization error over the new data.

To avoid this problem we should deal with these types of snooping. Limit model- and data- snooping to a small subset of S which will be "contaminated with optimism". Use it to get general understanding of the data and potential models. Only once the snooping stage has completed and a small set of candidate learners is chosen, use the entire training set for model selection- and evaluation. In addition, avoid manual data snooping. **Code the entire pre-processing stage**. At each iteration of Bootstap or cross-validation run the entire pre-processing step over the current sample, just like it would run when predicting over new data.

8. Unsupervised Learning

Up to this point, our learning paradigm was of *supervised batch learning*. Given a sample space $\mathcal{X}$ and a label/response space $\mathcal{Y}$, we were interested in *prediction*: finding some way to predict $y \in \mathcal{Y}$ corresponding to a new unseen sample $x \in \mathcal{X}$, based on a training set of labeled samples $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$. However, there are many learning problems that do not fit into this framework of supervised batch learning. In one such set of problems we still consider a domain set $\mathcal{X}$ but there is no label/response space. Namely, the data is of the form $S = \{\mathbf{x}_i\}_{i=1}^m$ where $\mathbf{x}_i \in \mathcal{X}$. Such problems are called *unsupervised* learning problems, and the goal is to infer different properties of $\mathcal{X}$. A few examples are:

- *Uncovering low-dimensional structures:* In some cases we have reason to believe that some given data, though represented in high dimension, could actually be represented in a lower dimension. Consider for example the MNIST database of handwritten digits. This coprus of 28-by-28 pixels grey-scaled images shows scans of people handwriting of the digits $0, \ldots, 9$. Though each image (sample) is represented in $\mathbb{R}^{28\times 28}$, there are very few variations on how people write digits. In Figure 8.1 we can see that in most images a big area in the surrounding of the image remains constant. This means that these pixels are not informative for predicting the digit. In addition, even though two images of the same digit (for example the digit 3 as seen below) show differences, they are still mostly the same. Therefore, it might be possible to *reduce the dimension* of the samples into a more compact, of lower dimension, representation.

- *Clustering:* Often, when we are given a dataset, we are interested in grouping (or segmenting) it into subsets such that those samples within each cluster are in some way more "closely related" to each other than to samples in other subsets. We refer to these subsets as *clusters*. Consider once more the MNIST digits dataset. In general, it seems logical that images of the same digit resemble each other more than images of different digits. If we would to try to cluster it (that is, segment its samples into different subsets) we would hope to get 10 separate subsets, each containing images of a specific digit.

- *Anomaly detection:* Suppose we are interesting in detecting when some given system is not behaving

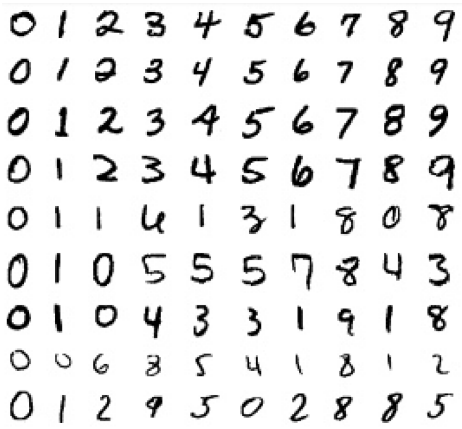


Figure 8.1: Handwritten digits*: Liu et al., Handwritten digit recognition, Pattern Recognition 37(2) 2004*

"as usual". Consider an air conditioning system or power plant. We place sensors in the system and would like to get a warning when something is behaving "strange". We do not know what exactly a "strange" behavior would look like, but it might be caused due to some mechanical malfunction, a software problem or even a cyber attack. We monitor the state of the system at all times. If we have installed d sensors, each time we take a reading of the system we get a sample $\mathbf{x}_i \in \mathbb{R}^d$. We train our learning system on the readings $\mathbf{x}_1, \ldots, \mathbf{x}_m \in \mathbb{R}^d$ where we believe these represent the normal state of the system. Now, given a new reading $\mathbf{x} \in \mathbb{R}^d$, is it "normal"? or is there something wrong? We are required to make a decision without having seen the system in the "wrong"/"abnormal"/"strange" state before.

8.1 Dimensionality Reduction

Dimensionality reduction is the process of mapping some high dimensional space (the ambient space) to some new low dimensional space (the intrinsic space). Given a dataset $\mathbf{x}_1, \ldots, \mathbf{x}_m \in \mathbb{R}^d$ we want to find some mapping $f : \mathbb{R}^d \to \mathbb{R}^k$ for $k \ll d$, where $f(\mathbf{x}_1), \ldots, f(\mathbf{x}_m)$ still resembles $\mathbf{x}_1, \ldots, \mathbf{x}_m$ in some sense. Different dimensionality reduction techniques are usually separated to linear- and non-linear techniques depending on the selected f.

There are different reasons to study and apply dimension reduction:

- **Learnability:** Some learning algorithms work better when the dimension d is small compared with the training sample size m, and might even fail if d is too large. Recall for example the case of linear regression. Though in both cases $m < d$ and $d < m$ we have a closed form solution, we have seen that if $m < d$ (i.e. less samples than features) the results are numerically unstable.

- **Computation:** For many algorithm the time- and space complexity depends on the dimension d. By reducing the data dimensionality we can use fewer computational resources to perform the learning task we intended.

- **Visualization:** Visualization is an extremely useful tool both for explaining and exploring our data. If we can reduce dimension to $k \leq 4$ then we can plot the data (possibly on a 3 dimensional axis, with the forth dimension represented by color).

8.1.1 Principal Component Analysis (PCA)

Let $\mathbf{x}_1,\ldots,\mathbf{x}_m \in \mathbb{R}^d$ be some dataset and suppose that it approximately resides in some lower k-dimensional linear subspace of $\mathbb{R}^d$. We would like to find some linear transformation of the data to project it on that lower dimension subspace. To do so we need to solve two challenges:

1. There are infinitely many subspaces we could consider. Which subspace should we choose and how should we acquire it?

2. Even if we knew the subspace, projecting the data-points $\mathbf{x} \in \mathbb{R}^d$ onto the subspace does not yet reduce the dimension. We would want to find a way to represent each sample by the k coordinates of that sample *in* the subspace. This is also known as the *embedding* of the samples.

In the case of Principal Component Analysis (PCA), given a dataset $\mathbf{x}_1,\ldots,\mathbf{x}_m \in \mathbb{R}^d$ we are searching for some linear transformation $f : \mathbb{R}^d \to \mathbb{R}^d$ whose range is of dimension k, such that we minimize the squared error between the original samples and their transformation:

$$f^* \coloneqq \underset{f}{\operatorname{argmin}} \sum_{i=1}^{m} ||\mathbf{x}_i - f(\mathbf{x}_i)||^2 \tag{8.1}$$

This problem can be formulated in four different ways:

1. Finding the closest affine subspace to the points.
2. Finding the affine subspace that retains most of the variation seen in the data (sometimes referred to as *signal*).
3. Finding the affine subspace that minimizes the distortion of the pairwise distances between points in the ambient- compared to the intrinsic spaces.
4. Generalization of linear regression with Gaussian noise both in the explanatory variables directions and in the response direction. This interpretation is also known as probabilistic PCA.

Here, we will discuss the first two, most common, formulations.

8.1.1.1 Closest Affine Subspace

To simplify the problem of finding the closest affine subspace, let us assume that the data is centered around the origin and as such instead of searching for some affine subspace, we are searching for an "ordinary" subspace. Therefore, we are searching for some linear map $W : \mathbb{R}^d \to \mathbb{R}^k$ and an "inverse" linear map $U : \mathbb{R}^k \to \mathbb{R}^d$

$$W^*, U^* = \underset{W \in \mathbb{R}^{k\times d}, U \in \mathbb{R}^{d\times k}}{\operatorname{argmin}} \sum_{i=1}^{m} ||\mathbf{x}_i - UW\mathbf{x}_i||^2 \tag{8.2}$$

Lemma 8.1.1 Let (U,W) be a solution to (8.2). Then U's columns are orthonormal and $W = U^\top$.

Proof. Let U,W be two matrices and consider the mapping $\mathbf{x} \mapsto (UW)\mathbf{x}$. The matrix (UW) is a d-by-d matrix whose image is a k dimensional subspace of $\mathbb{R}^d$. Denote $S \coloneqq \operatorname{Im}(UW)$. As $\mathbf{x} \in \mathbb{R}^d$ and $(UW)\mathbf{x} \in S$, it holds that the projection that minimizes $||\mathbf{x} - UW\mathbf{x}||_2$ is the orthogonal projection onto the subspace. In addition, the point in S closest to $\mathbf{x}$, namely the orthogonal projection of $\mathbf{x}$ onto S, is given by $VV^\top\mathbf{x}$ where the columns of V are an orthonormal basis of S:

$$\forall \mathbf{u} \in S \quad ||\mathbf{x} - \mathbf{u}||_2 \geq \left|\left|\mathbf{x} - VV^\top\mathbf{x}\right|\right|_2$$

Thus, the solution to 8.2 is U with orthonormal columns and $W \coloneqq U^\top$. ■

Based on the above lemma, we can now write an equivalent problem to the PCA problem presented in 8.2. Observe that:

$$\begin{aligned}\left|\left|\mathbf{x}-UU^\top\mathbf{x}\right|\right|^2 &= ||\mathbf{x}||^2 - 2\mathbf{x}^\top UU^\top\mathbf{x} + \mathbf{x}^\top UU^\top UU^\top\mathbf{x} \\ &= ||\mathbf{x}||^2 - \mathbf{x}^\top UU^\top\mathbf{x} \\ &= ||\mathbf{x}||^2 - \left(U^\top\mathbf{x}\right)^\top U^\top\mathbf{x} \\ &\overset{(*)}{=} ||\mathbf{x}||^2 - \text{trace}\left(U^\top\mathbf{x}\left(U^\top\mathbf{x}\right)^\top\right) \\ &= ||\mathbf{x}||^2 - \text{trace}\left(U^\top\mathbf{x}\mathbf{x}^\top U\right)\end{aligned}$$

where $(*)$ is because $\forall \mathbf{v},\mathbf{u}\in\mathbb{R}^k$ $trace\left(\mathbf{u}\mathbf{v}^\top\right)=\mathbf{v}^\top\mathbf{u}$. As the trace is a linear operator we can re-write an equivalent problem:

$$U^* = \underset{U\in\mathbb{R}^{d\times k}, U^\top U=I}{\text{argmax}} \sum_{i=1}^m \text{trace}\left(U^\top\mathbf{x}_i\mathbf{x}_i^\top U\right) = \underset{U\in\mathbb{R}^{d\times k}, U^\top U=I}{\text{argmax}} \text{trace}\left(U^\top\sum_{i=1}^m\mathbf{x}_i\mathbf{x}_i^\top U\right) \tag{8.3}$$

Theorem 8.1.2 Let $A=\sum\mathbf{x}_i\mathbf{x}_i^\top$ and let $\mathbf{u}_1,\ldots,\mathbf{u}_k$ be the k leading eigenvectors of A. Then, the solution to the PCA problem is given by $U\in\mathbb{R}^{d\times k}$ whose columns are $\mathbf{u}_1,\ldots,\mathbf{u}_k$.

Proof. Denote $A=\sum\mathbf{x}_i\mathbf{x}_i^\top$. Then, we need to solve

$$\underset{U\in\mathbb{R}^{d\times k}, U^\top U=I}{\text{argmax}} \; trace\left(U^\top AU\right)$$

Notice that A is a square symmetric matrix so let $A=VDV^\top$ be the EVD of A, where the diagonal of D are the eigenvalues of A in decreasing order $\lambda_1\geq\ldots\geq\lambda_d$ and the columns of V are the corresponding eigenvectors. So:

$$\begin{aligned}\text{trace}\left(U^\top AU\right) &= \text{trace}\left(U^\top VDV^\top U\right) \overset{(*)}{=} \text{trace}\left(B^\top DB\right) \\ &= \text{trace}\left(DBB^\top\right) = \textstyle\sum_{j=1}^d\left[DBB^\top\right]_{jj} \\ &= \textstyle\sum_{j=1}^d[D]_j\left[BB^\top\right]_{\cdot,j} = \sum_{j=1}^d\lambda_j\cdot\left[BB^\top\right]_{jj} \\ &= \textstyle\sum_{j=1}^d\lambda_j\cdot\sum_{i=1}^k B_{ji}^2\end{aligned}$$

where in $(*)$ we denote $B=V^\top U\in\mathbb{R}^{d\times k}$. Notice that

$$\begin{gathered}B^\top B = U^\top VV^\top U = I_d \\ \Downarrow \\ \mathbb{1}_{j\neq i} = \left\langle\left[B^\top\right]_j,[B]_{\cdot,i}\right\rangle = \left\langle[B]_{\cdot,j},[B]_{\cdot,i}\right\rangle\end{gathered}$$

meaning that the columns of B are orthonormal, and therefore $\sum_{j=1}^d\sum_{i=1}^k B_{ji}^2=k$. Based on B, define $\widetilde{B}=\left[\begin{array}{c|c} B & M\end{array}\right]\in\mathbb{R}^{d\times d}$ with $M\in\mathbb{R}^{d\times(d-k)}$ completing an orthonormal basis in R^d. If so, then $\forall j$ $\sum_{i=1}^d\widetilde{B}_{ji}^2=1$, which implies that $\sum_{i=1}^k B_{ji}^2$. As such

$$\text{trace}\left(U^\top AU\right) = \sum_{j=1}^d\lambda_j\cdot\sum_{i=1}^k B_{ji}^2 \leq \max_{\beta\in[0,1]^d,||\beta||_1\leq k}\sum_{j=1}^d\lambda_j\beta_j = \sum_{j=1}^k\lambda_k$$

To conclude the proof, let U be the matrix whose columns are the k leading eigenvectors of A then

$$U^\top AU = \begin{bmatrix}\mathbf{u}_1^\top\\ \vdots\\ \mathbf{u}_k^\top\end{bmatrix} A\begin{bmatrix}\mathbf{u}_1 & \cdots & \mathbf{u}_k\end{bmatrix} = \begin{bmatrix}\mathbf{u}_1^\top\\ \vdots\\ \mathbf{u}_k^\top\end{bmatrix}\begin{bmatrix}\lambda_1\mathbf{u}_1 & \cdots & \lambda_k\mathbf{u}_k\end{bmatrix} = \text{diag}\left(\lambda_1,\ldots,\lambda_k,\overbrace{0,\ldots,0}^{d-k}\right)$$

and $\text{trace}\left(U^\top AU\right)=\text{trace}\left(\text{diag}\left(\lambda_1,\ldots,\lambda_k,0,\ldots,0\right)\right)=\sum^k\lambda_i$, as requested. ■

Generalizing For Affine Subspaces

In the above theorem we in fact found the closest subspace- and not the closest affine subspace to the data. To find the closest affine subspace, we would like to allow the mapping W to be *affine*. To achieve this we generalize the above by considering $W : \mathbb{R}^d \to \mathbb{R}^k$ to be of the form

$$W(\mathbf{x}) := \widetilde{W}(\mathbf{x} - \mu), \quad \mu \in \mathbb{R}^d, \widetilde{W} : \mathbb{R}^d \to \mathbb{R}^k$$

This allows us to "shift" the data before applying the linear map $\widetilde{W}$. When adding μ to the optimization problem above, we find that the minimizer μ is given by: $\mu = \frac{1}{m}\sum \mathbf{x}_i$. This is the empirical mean of the data, often denoted by $\overline{\mathbf{x}}$. So, in order to find the closest affine subspace to the data we center the matrix A:

$$A := \sum_{i=1}^{m} (\mathbf{x}_i - \overline{\mathbf{x}})(\mathbf{x}_i - \overline{\mathbf{x}})^\top$$

yielding the following pseudo-code for the PCA algorithm:

Algorithm 11 PCA

procedure PCA($\mathbf{X}$, k) ▷ $\mathbf{X}$ The design matrix of m samples and d features
  Compute $A \leftarrow \sum_{i=1}^{m} (\mathbf{x}_i - \overline{\mathbf{x}})(\mathbf{x}_i - \overline{\mathbf{x}})^\top$
  Let $\mathbf{u}_1, \ldots, \mathbf{u}_k$ be the eigenvectors of A corresponding the largest eigenvalues.
  return $\mathbf{u}_1, \ldots, \mathbf{u}_k$
end procedure

The eigenvalues of A, $\lambda_1 \geq \ldots \geq \lambda_d$ are referred to as the *Principal Values* of $\mathbf{X}$. The eigenvectors of A, $\mathbf{u}_1, \ldots, \mathbf{u}_d$ are referred to as the *Principal Components*. Notice that the matrix A is a d-by-d positive semi-definite matrix. Therefore the PCA algorithm is in fact a case of a matrix diagonalization algorithm where we simply try to find the eigenvalues and eigenvectors of some target matrix.

8.1.1.2 Maximum Retained Variance

Another way to think about PCA is as a dimensionality reduction technique that maintains the maximum amount of variance of the data possible in a $k < d$ dimensional subspace. To prove so we will have to solve a constraint maximization problem using Lagrange Multipliers.

Theorem 8.1.3 Let $\mathbf{X}$ be an m-by-d design matrix. The projection of $\mathbf{X}$ onto a k dimensional linear subspace that retains maximum of the variance in $\mathbf{X}$ is given by the matrix $U \in \mathbb{R}^{d \times k}$ whose columns are the k eigenvectors with leading eigenvalues of the sample covariance matrix S.

Proof. We begin with considering the projection onto a one-dimensional subspace. Without loss of generality let $\mathbf{v} \in \mathbb{R}^d$ be a unit vector used to project the data onto. The expected value of the projected data is

$$\mathbb{E}_{\mathbf{x}}\left[\mathbf{v}^\top \mathbf{x}\right] = \frac{1}{m}\sum \mathbf{v}^\top \mathbf{x}_i = \mathbf{v}^\top \overline{\mathbf{x}}$$

Therefore, the variance of the projection is given by:

$$\begin{aligned}
Var\left(\mathbf{v}^\top \mathbf{x}\right) &= \mathbb{E}_{\mathbf{x}}\left[\left(\mathbf{v}^\top \mathbf{x}_i - \mathbb{E}_{\mathbf{x}}\left[\mathbf{v}^\top \mathbf{x}\right]\right)^2\right] &&= \frac{1}{m}\sum\left(\mathbf{v}^\top \mathbf{x}_i - \mathbf{v}^\top \overline{\mathbf{x}}\right)^2 \\
&= \frac{1}{m}\sum\left[\mathbf{v}^\top (\mathbf{x}_i - \overline{\mathbf{x}})\right]^2 &&= \frac{1}{m}\sum\left[\mathbf{v}^\top (\mathbf{x}_i - \overline{\mathbf{x}})\right]\left[\mathbf{v}^\top (\mathbf{x}_i - \overline{\mathbf{x}})\right]^\top \\
&= \frac{1}{m}\sum \mathbf{v}^\top (\mathbf{x}_i - \overline{\mathbf{x}})(\mathbf{x}_i - \overline{\mathbf{x}})^\top \mathbf{v} &&= \mathbf{v}^\top\left[\frac{1}{m}\sum (\mathbf{x}_i - \overline{\mathbf{x}})(\mathbf{x}_i - \overline{\mathbf{x}})^\top\right]\mathbf{v} \\
&= \mathbf{v}^\top S \mathbf{v}
\end{aligned}$$

So now let us maximize the projected variance with respect to $\mathbf{v}$:

$$\hat{\mathbf{v}} = \underset{||\mathbf{v}||=1}{\text{argmax}}\ \mathbf{v}^\top S\mathbf{v}$$

To solve this optimization problem we will use the Lagrange multipliers method with the constraint $g(\mathbf{v}) = 1 - \mathbf{v}^\top\mathbf{v}$. So the lagrangian is given by:

$$\begin{gathered}\mathcal{L} = \mathbf{v}^\top S\mathbf{v} + \lambda g(\mathbf{v}) \\ \Downarrow \\ \frac{\partial}{\partial \mathbf{v}}\mathcal{L} = 2S\mathbf{v} - 2\lambda\mathbf{v} = 0\end{gathered}$$

Therefore the maximizer of $\mathbf{v}^\top\mathbf{v}$ must be an eigenvector of S. Notice that by left-multiplying the derivative by $\mathbf{v}^\top$ we find that the retained variance itself is given by:

$$\mathbf{v}^\top S\mathbf{v} = \lambda\mathbf{v}^\top\mathbf{v} \overset{||\mathbf{v}||=1}{=} \lambda$$

Thus, the maximal retained variance is the largest eigenvalue λ_1, achieved by $\mathbf{v} := \mathbf{u}_1$.

Next, let us find the direction that retains the second largest amount of variance. As we are looking for an orthogonal projection we add an additional constraint that this direction is orthogonal to $\mathbf{u}_1$:

$$\hat{\mathbf{v}} = \underset{||\mathbf{v}||=1, \mathbf{v}^\top\mathbf{u}_1=0}{\text{argmax}}\ \mathbf{v}^\top S\mathbf{v}$$

As before, when solving the constraint optimization problem for $\mathbf{v}$ we find that the direction retaining the second largest amount of variance is $\mathbf{v} := \mathbf{u}_2$, with the amount of variance being λ_2. Proving by induction we get that $\forall k \leq d$, the k dimensional subspace that retains maximal variance of projecting $\mathbf{X}$ onto it, is given by the k leading eigenvectors of S. ■

8.1.1.3 Link Between Closest Subspace and Maximum Variance

In the above we have seen two interpretations of PCA: one as closest subspace and one as maximizing variance. To understand how the two are connected consider the dataset seen in Figure 8.2 and specifically the $\mathbf{x}_i$ data-point. Notice that by orthogonally projecting $\mathbf{x}_i$ onto $\mathbf{u}_1$ a right-angle triangle is formed where:

- The edge denoted by a is the size of the projection of $\mathbf{x}_i$ onto $\mathbf{u}_1$: $a := \left|\left|\mathbf{x}_i^\top\mathbf{u}_1\right|\right|$. This is the measure maximized in the maximum variance interpretation.
- The edge denoted by b is the distance between the original data-points $\mathbf{x}_i$ and its orthogonal projection onto $\mathbf{u}_1$: $b := ||\mathbf{x}_i - \mathbf{x}_i\top\mathbf{u}_1||$. This is the measure minimized in the closest subspace interpretation.
- The edge denoted by c is the size of $\mathbf{x}_i$: $c := ||\mathbf{x}_i||$.

As this is a right-angle triangle, we know from the Pythagorean theorem that $c^2 = a^2 + b^2$. Therefore, if we find a PCA solution that minimizes b (the closest subspace), we in fact find a solution that maximizes a. Similarly by finding a solution that maximizes a (maximal projected variance), we find a solution that minimizes b.

8.1.1.4 Projection- vs. Coordinates of Data-Points

When considering PCA (or many other dimensionality reduction algorithms), an important but often overlooked point is the difference between projection and embedding (i.e coordinates) of the data-points. Suppose $\mathbf{X} \in \mathbb{R}^{m\times d}$ and we run the PCA algorithm to find a lower k dimensional linear subspace. The optimal PCA solution is the subspace that minimizes the sum of squared distances between each data-point $\mathbf{x}_i$ and its orthogonal projection onto the subspace. As we have seen above (8.1.2, 8.1.3), this subspace is spanned by the k leading eigenvectors of the $d \times d$ sample covariance matrix $S = \frac{1}{m}\sum(\mathbf{x}_i - \bar{\mathbf{x}})(\mathbf{x}_i - \bar{\mathbf{x}})^\top$.

This matrix $U \in \mathbb{R}^{d\times k}$ enables the *projection* of the points in $\mathbb{R}^d$ onto the k dimensional subspace, spanned by these leading eigenvectors. However, we would also like to actually reduce the dimension. Namely, find the

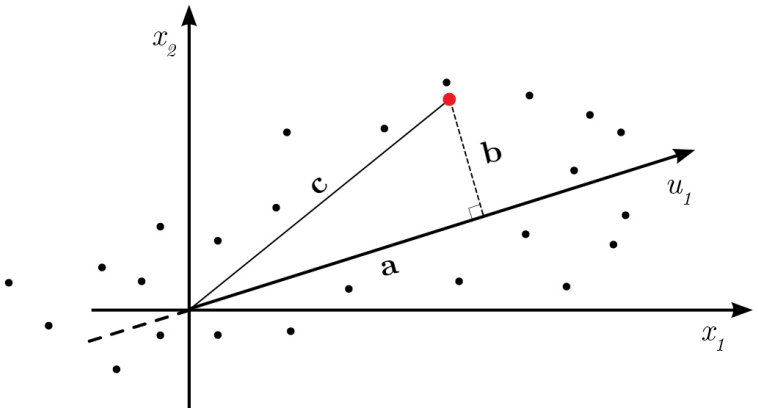


***Figure 8.2: Link Between PCA Interpretations**: Projection of $\mathbf{x}_i$ onto $\mathbf{u}_1$ forming a right-angle triangle.*

map $W : \mathbb{R}^d \to \mathbb{R}^k$ and work with the dimension-reduced dataset $W(\mathbf{x}_1), \ldots, W(\mathbf{x}_m)$. As proven above, it is in fact $W = U^\top$ the map that provides the *embedding* of the data into the low dimension space. The vector $U^\top \mathbf{x}_i$ is a k dimensional vector, laying within the found k dimensional subspace. These are the *coordinates* of the original vector $\mathbf{x}_i$ according to the orthonormal set of k leading eigenvalues of S.

Let $\mathbf{u}_1, \ldots, \mathbf{u}_d$ be the d eigenvectors of S numbered in ascending order by the associated eigenvalues. As these vectors form a basis in $\mathbb{R}^d$, the vector $\mathbf{x}_i$ can be decomposed as $\mathbf{x}_i = \sum_{j=1}^{d} \langle \mathbf{x}_i, \mathbf{u}_j \rangle \mathbf{u}_j$.

- The *projecting* of $\mathbf{x}_i$ on the optimal k-dimensional subspace is given by $\mathbf{x}_i = \sum_{j=1}^{k} \langle \mathbf{x}_i, \mathbf{u}_j \rangle \mathbf{u}_j$.
- The *coordinates* (i.e. embedding) of $\mathbf{x}_i$ according to the k leading principal vectors are given by: $(\langle \mathbf{x}_i, \mathbf{u}_1 \rangle, \ldots, \langle \mathbf{x}_i, \mathbf{u}_k \rangle)^\top$.

In Figure 8.3 we illustrate the difference between the projection and the coordinates (embedding) of the data-points. This dataset was generated as follows: 1000 points were sampled from the ℓ_2 unit ball in $\mathbb{R}^2$. That is, $\{((x_i)_1, (x_i)_2)\}_{i=1}^{1000}$ where $(x_i)_1^2 + (x_i)_2^2 = 1$. Then Gaussian noise was added in a third axis:

$$\{((x_i)_1, (x_i)_2, \varepsilon_i)\}_{i=1}^{1000} \quad (x_i)_1^2 + (x_i)_2^2 = 1 \quad \varepsilon_1, \ldots, \varepsilon_{1000} \overset{i.i.d}{\sim} \mathcal{N}(0, 0.1)$$

Figure 8.3 (left) shows this dataset. Then, using PCA we first project the data onto the subspace defined by the 3 PCs (Figure 8.3, center). At this point each data-point is still represented in $\mathbb{R}^3$, but we can indeed see that it is mainly the first 2 axes that capture the signal in the data, with the third axis showing only minor variations (the added noise). Lastly, when embedding the data into a 2 dimensional subspace (Figure 8.3, right), we are left with the true signal of the data (i.e. samples drawn from a 2 dimensional ℓ_2 unit ball), and data-points are represented using only 2 coordinates.

8.1.1.5 Principal Components As "Typical Data-Points"

The principal components found by PCA are vectors in the ambient space $\mathbb{R}^d$. This means that they share the same dimension as the data-points $\mathbf{x}_1, \ldots, \mathbf{x}_m$. As they are orthonormal vectors chosen such that the first k provide the best linear approximation of dimension k to the dataset, we can look at them in an interesting way. They are, in this sense, the "typical" data-points, maximally different from each other (as they are an orthonormal set). Thus, it is often interesting to see what they represent as data-points: what part of the "signal" do they capture.

Returning to the MNIST Digits dataset of Figure 8.1 we can look at each principal component as a 28-by-28 image. Then, we can think of the representation of each image by the linear combination of these images:

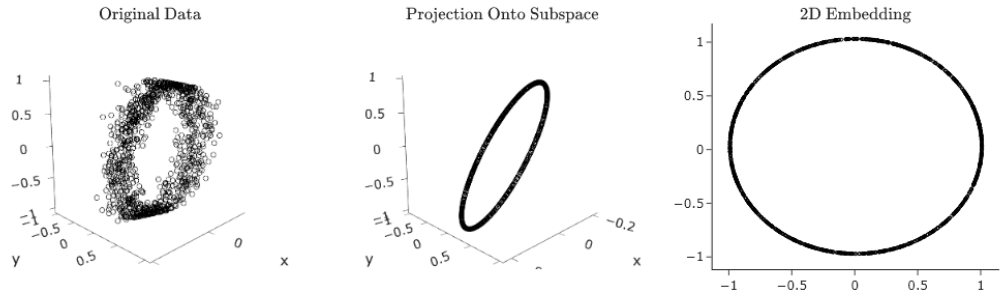


***Figure 8.3: Projection vs. Embedding**: Dataset of sampels taken on the ℓ_2 unit ball with Gaussian noise in $\mathbb{R}^3$, and then rotated in a random direction. Unsupervised PCA Examples*

$$\hat{f}(\mathbf{x}) = \sum_{i=1}^{k} \alpha_i \mathbf{u}_i + \bar{\mathbf{x}}, \quad \alpha_1, \ldots, \alpha_k \in \mathbb{R}$$

where $\mathbf{u}_1, \ldots, \mathbf{u}_k$ are the leading k principal components and $\hat{\mathbf{x}}$ is the centering vector of the data. Notice that as this is a linear combination we are reconstructing the sample $\mathbf{x}$ via adding and subtracting (weighted) principal components.

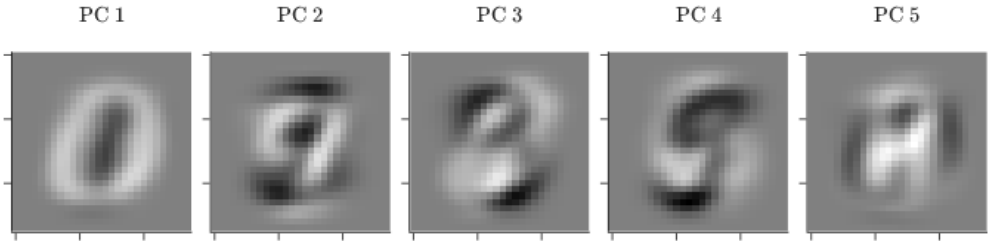


***Figure 8.4: Top Principal Components** of PCA fitted over the MNIST Digits dataset. Unsupervised PCA Examples*

Figure 8.5 shows the reconstruction process of an image of digit 4 as the linear combination of the principal components. In each frame of the animation the restored image (center) is calculated as the restored image of the previous iteration plus $\alpha_i \mathbf{u}_i$ for $\mathbf{u}_i$ the principal component of the current iteration and α_i the loadings (coordinates) of the image for the i'th principal component.

8.2 Clustering

A very useful set of learning problems is of clustering. Often, either as part of data exploration or as part of the main analysis, we are interested in partitioning our data into *meaningful* groups. For example, given a corpus of images we might want to divide them into different groups such as nature/urban/people/etc. photographs; or, given the set of genes in the human genome, we might want to group together genes associated with specific diseases. In both these examples we are only given the samples (images or genes) but we do not have any *ground truth* (labels). We are not given the information of what is seen in each image, nor for each disease what genes are associated with it.

We would therefore want to define some measure of *similarity* between samples of a given domain. Using this

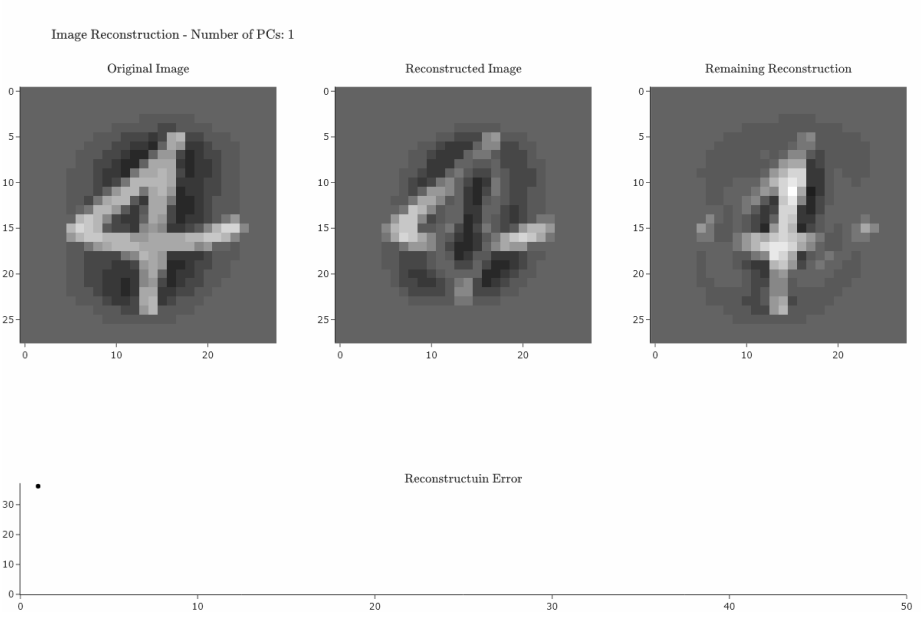


Figure 8.5: ***PCA Reconstruction:*** *Constructing image from principal components. Unsupervised PCA Examples*

similarity we could split the data into different subsets, where intuitively samples within a given set are *more similar* to one another compared to samples of different sets.

R This form of clustering, where we partition the data into distinct non-overlapping groups is also referred to as "hard assignment" as each sample is assigned to one specific subset. Sometimes, rather than assigning to some specific cluster we are interested in "partly assigning" to multiple clusters. This is referred to as "soft assignment/clustering".

8.2.1 K-Means

Though there are many different approaches to perform clustering, common to many is the notion of defining some representative data-point of the cluster. Then, clustering of data-points give the given sample is perform with respect to these cluster representatives.

Definition 8.2.1 A partition of a dataset $\{\mathbf{x}_i\}_{i=1}^m$ is a set $C_1,\ldots,C_k$ such that $\{\mathbf{x}_i\}_{i=1}^m = \biguplus_{j=1}^k C_j$

Given some partition over the data and the representative data-points, we can define a *cost function* for the partition. For some metric (i.e. distance function) over the data $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}_+$ we define:

$$G_d(C_1,\ldots,C_k,\mu_1,\ldots,\mu_k) \coloneqq \sum_{j=1}^{k} \sum_{\mathbf{x}\in C_j} d(\mathbf{x},\mu_j) \tag{8.4}$$

where $\mu_1,\ldots,\mu_k$ are the representatives of clusters $C_1,\ldots,C_k$. Using this function, the goal is to find the partitioning that minimizes the following:

$$\{C_1,\ldots,C_k\}^* = \underset{\{C_j,\mu_j\}_{j=1}^k}{argmin}\ G_d(C_1,\ldots,C_k,\mu_1,\mu_k) = \underset{\{C_j,\mu_j\}_{j=1}^k}{argmin} \sum_{j=1}^{k} \sum_{\mathbf{x}\in C_j} d(\mathbf{x},\mu_j) \tag{8.5}$$

In the case of the k-means algorithm the cluster representatives are chosen to be:

$$\mu_j(C_j) \coloneqq \underset{\mu \in \mathcal{X}}{argmin} \sum_{\mathbf{x} \in C_j} d(\mathbf{x}, \mu)$$

In order to find the minimizers of 8.5 we would have to navigate the space of all possible partitions of m objects into k subsets. As there are an exponential amount of such subsets, minimizing the cost function G is NP-hard, and we must resort to *heuristics*. The most famous heuristic for minimizing G, when d is the Euclidean distance, is k-means.

Algorithm 12 K-Means

procedure K-MEANS($\mathbf{X}$, k) ▷ $\mathbf{X}$ The design matrix of m samples and d features
  Choose initial centroids $\mu_1, \ldots, \mu_k$ randomly.
  while Not converged **do**
    Assignment: Assign each point $\mathbf{x}_i$ to the centroid closest to it:

$$C_j^{(t)} \coloneqq \left\{ \mathbf{x} | \mu_j = argmin_\mu \, d(\mathbf{x}, \mu), \mathbf{x} \in \mathbf{X} \right\}$$

    Update: Adjust cluster centroids by: $\forall j \in [k] \quad \mu_j^{(t+1)} \coloneqq \left| C_j^{(t)} \right|^{-1} \sum_{\mathbf{x} \in C_j^{(t)}} \mathbf{x}$
  end while
  return $C_1, \ldots, C_k$
end procedure

This clustering approach uses *Lloyd's Algorithm* to minimize G using an iterative strategy where each time we *alternate* between minimizing two terms. We first define a partition of the dataset, associating points to subsets by their nearest centroid. These subsets are called Voronoi cells. Then we re-calculate the cluster's centroid.

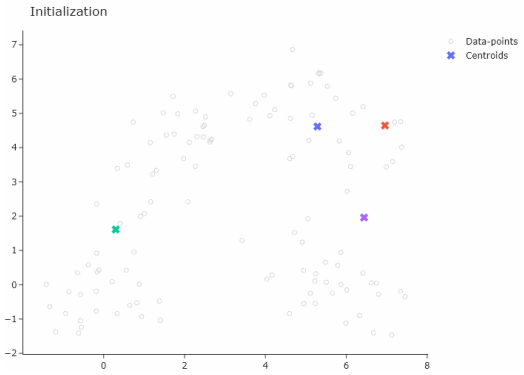


Figure 8.6: ***K-Means Run*** *Unsupervised K-Means Examples*

8.2.1.1 Convergence to Multiple- and Sub- Optimal Solutions

As clustering problems are NP-Hard, the K-Means algorithm uses the Lloyd's algorithm heuristic to find a good partition of the data. Being a heuristic, the optimality of the algorithm assignment to clusters isn't guaranteed. In fact, due to the nature of the random initialization of centroids, the K-Means algorithm might

converge into a sub-optimal solution or to there exists more than a single possible optimal solution.

Sub-optimality means that given a dataset $\{\mathbf{x}_i\}_{i=1}^m$ the K-Means algorithm returned some partitioning $C_1,\ldots,C_k$ such that there exists a different partitioning of the data $C'_1,\ldots,C'_k$ with a lower objective value: $G(C_1,\ldots,C_k) > G\left(C'_1,\ldots,C'_k\right)$. This is the product of the objective function not being convex, which means there could be several local minima, each achieving a different cost. In Figure 8.7 we see 4 groups of data-points and two initial centroids, positioned in such a way that forces the algorithm to converge into a sub-optimal solution. Optimal assignment is achieved for final centroids $\mu_i = (0,5)\,, \mu_j = (20,5)\,,\ i \neq j$.

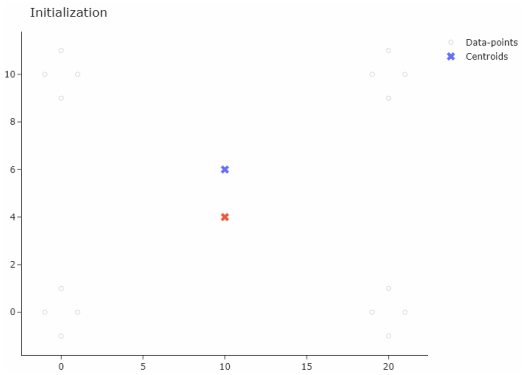


Figure 8.7: ***Suboptimal Solution:*** *Data-points and initial centroids leading to a suboptimal solution. Unsupervised K-Means Examples*

Multiple optimal solution means that given a dataset $\{\mathbf{x}_i\}_{i=1}^m$ there are more than a single possible partitioning of the data that will yield the lowest objective value. It is important to note, that whenever discussing clustering assignments, we always look at the partitioning up to a permutation of the partition names. That is, suppose we are given the samples $1,2,3,4,5$ the partitioning of $C_1 = \{1,2,3\}, C_2 = \{4,5\}$ is identical to $C_1 = \{4,5\}, C_2 = \{1,2,3\}$, and clearly, both will achieve the same objective value. When referring to multiple optimal solution we mean:

$$\begin{array}{ll} \text{Given } \{\mathbf{x}_i\}_{i=1}^m & \exists \{C_1\ldots,C_k\}, \left\{C'_1,\ldots,C'_k\right\} \\ \text{For which} & G(C_1,\ldots,C_k) = G\left(C'_1,\ldots,C'_k\right) \\ \text{It holds that} & \exists j \in [k]\ \ \forall l \in [k] \quad C_j \neq C'_l \end{array}$$

and that they achieve an objective value lower (or equal) to any other partitioning of the dataset. An example for such dataset and initialization of centroids is seen in Figure 8.7. In this case both centroids are initialized in the center of all data points.

8.2.1.2 Selection of k

As in different algorithms previously seen, in K-Means too we are required to provide a value for the hyper-parameter k. As we do not know how many clusters we really have in the dataset this is not a simple task. Notice that as long as we have more data-points than clusters, for any optimal solution with k clusters, we can achieve a solution with a lower objective for $k+1$. As such, simply running over different values of k and selecting the minimal value isn't a viable solution. Instead, we will apply a similar technique to the one used in PCA. We will plot the objective achieved for different values of k and select the value after which the improvement is less drastic. In Figure 8.9 we apply this strategy over the dataset seen in Figure 8.6. As seen in the elbow-plot once we reach $k = 4$, the incremental improvement in score is much lower. Since the data was generated from 4 different Gaussians, this approach managed to find the correct value.

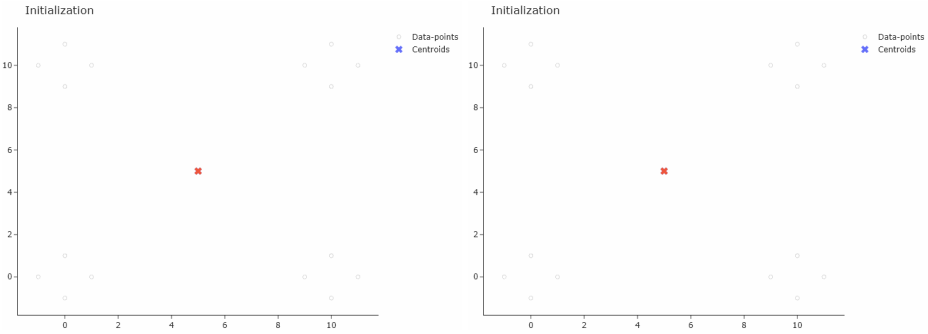


Figure 8.8: Multiple Optimal Solutions: *Data-points and initial centroids leading to two different optimal solutions.* *Unsupervised K-Means Examples*

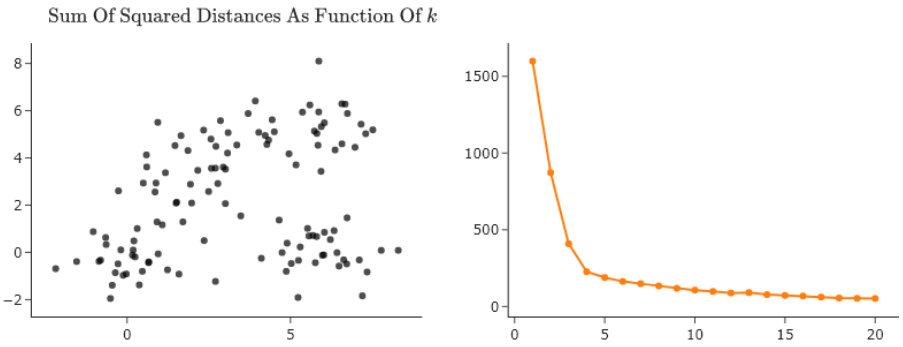


Figure 8.9: Selecting k hyper-parameter in K-Means *Unsupervised K-Means Examples*

8.2.2 Spectral Clustering

The K-Means algorithm discussed above has several limitations. The first is the implicit assumption that the structure of the data is of separate "clouds" of points, with each cloud representing a different cluster. If that is so, it makes sense to assign membership of samples to clusters by the distance to the cluster's center. However, what shall we do if the cluster structure is instead of shapes similar to those seen in Figure 8.10. In these cases we do not wish to cluster based on the global distances between samples, but rather by the *small* distances between *nearby* samples. For example, in the circles dataset of Figure 8.10 we would have liked to find a strategy for which distances between samples of different circles to be infinity, while the distances between samples in the same circle to be small.

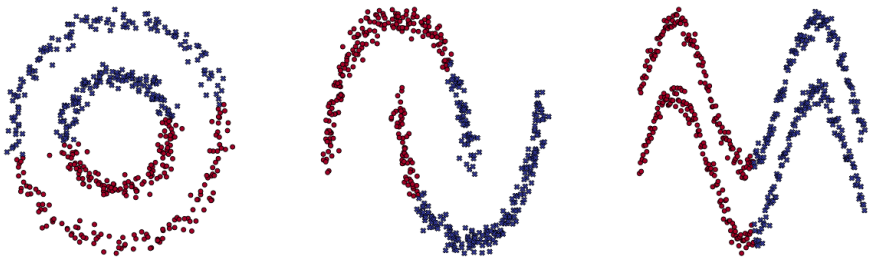
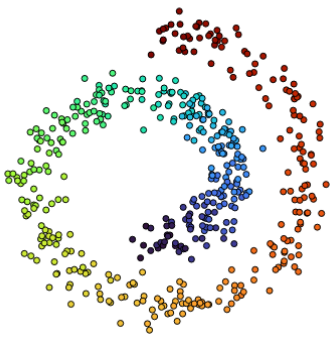


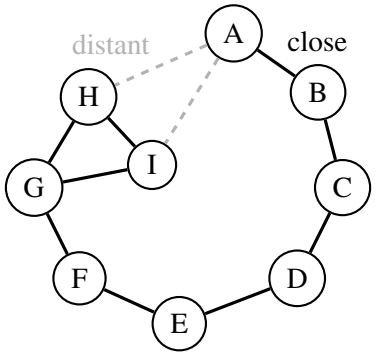
Figure 8.10: Scenarios unfit for K-Means *Unsupervised Spectral Clustering Examples*

A second limitation of K-Means relates to the curse of dimensionality. Suppose we cluster high dimensional data like images of different animals, with each image being represented by 1000-by-1000 RBG pixels. In this case the data is therefore embedded in an Euclidean space with dimension of $3 \cdot 10^6$. However, when we run the K-Means algorithm we are only interested in the *pairwise distances* between the samples and centroids. Perhaps we do not actually have to work with such high dimensional representation of the data.

The family of Spectral Clustering algorithms solve both these limitations and achieve so by combining the following ideas. First, we wish to consider *only* small pairwise distances between samples, while ignoring large possibly less informative distances. Then, based on these small (local) connections between nearby samples, derive the global structure of the data. Second, we could consider representing the samples using an undirected graph. Each sample is represented as nodes in a *similarity graph*, and weighted edges are added between pairs of samples that are *similar* enough. Thus, the problem of clustering the data is rephrased as a graph-partition problem.



(a) *Swiss-Roll dataset illustrating one-dimensional manifold within the two-dimensional euclidean space*



(b) *Node-graph of datapoints showing close datapoints linked*

Figure 8.11: *Distances and traversal according to similarity graph*

Put together, the general scheme of the Spectral Clustering algorithm is therefore to compute a pairwise similarity/affinity matrix capturing the local neighborhood relationships between datapoints, and then to find a good graph partitioning. If the clusters are properly separated, we intuitively expect each cluster to be captures by a different connected component. if that is the case, we could then find the different connected components using the DFS algorithm. However, if we construct a graph over noisy data, when the separation of different clusters is less clear, we might treat the entire sample as a single connected component and fail to separate the different clusters.

Thus, we need an algorithm that can find the connected components of a graph in a manner that is *robust* to the influences of random noise. The algorithm that will manage to do so is that of matrix diagonalization.

8.2.2.1 Graph Laplacian Matrices

Let us begin with exploring the connection between a graph and the eigenvectors of its corresponding affinity matrix. Consider a set of datapoints $\mathbf{x}_1, \ldots, \mathbf{x}_m$ and a value $0 < \varepsilon \in \mathbb{R}$. We construct an ε-neighborhood graph where each datapoint corresponds to a node and there exists an edge between every two datapoints whose

Euclidean distance is at most ε. Formally:

$$G \coloneqq \langle V, E \rangle \quad \text{where} \quad \begin{aligned} V &\coloneqq \{1, \ldots, m\} \\ E &\coloneqq \left\{ \{i, j\} \mid \left|\left| \mathbf{x}_i - \mathbf{x}_j \right|\right| < \varepsilon,\ i \neq j \right\} \end{aligned}$$

To conceptualize this neighborhood graph fix $\varepsilon > 0$ and consider the graph constructed over a dataset as seen in Figure 8.12. Notice that if the *local density* around a given point is low there are few points an a ball of radius ε around the point. This means that its ε-neighborhood contains few points and it has few edges in G. If however the local density around the point is high then there are many points in its ε-neighborhood and many edges in G.

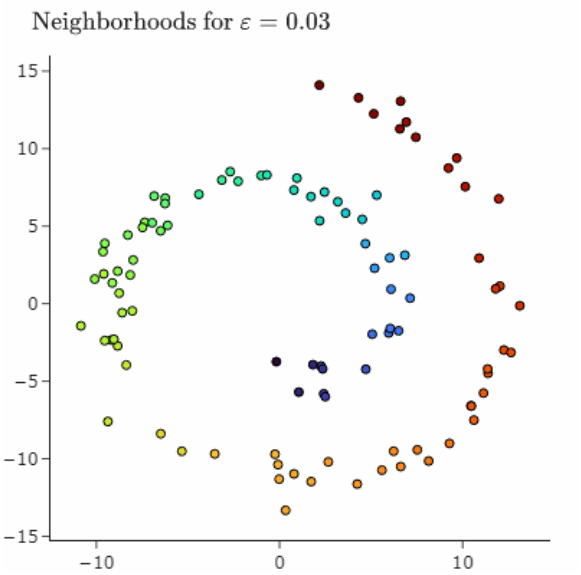


Figure 8.12: *Construction of ε-neighborhoods for different values of ε. Unsupervised Spectral Clustering Examples*

Denote by $A \in \mathbb{R}^{m \times m}$ the weighted *adjacency matrix* (also refereed to as the affinity matrix) of G. By construction, A is a symmetric matrix where:

$$A_{ij} \coloneqq \begin{cases} 1 & \left|\left| \mathbf{x}_i - \mathbf{x}_j \right|\right| < \varepsilon \wedge i \neq j \\ 0 & otherwise \end{cases}$$

Using the adjacency matrix A, we define the *degree* matrix D, whose values are the sums of the rows of A:

$$D_{ii} \coloneqq \sum_{j=1}^{m} A_{ij}, \quad D \in \mathbb{R}^{m \times m}$$

and consider the matrix L defined as:

$$L \coloneqq D - A \tag{8.6}$$

This is known as the *unnormalized Graph Laplacian* matrix, which has several properties useful for the task of partitioning the data. It is a symmetric and PSD matrix. As such it has non-negative, real-valued eigenvalues, with the smallest being zero. The multiplicity eigenvalue 0 is equal to the number of connected components in G. Furthermore, the eigenspace of eigenvalue 0 is spanned by the indicator vectors $\mathbb{1}_{C_1}, \ldots, \mathbb{1}_{C_k}$ for $C_1, \ldots, C_k$ denoting the k connected components in G.

From the unnormalized Graph Laplacian we further derive the *normalized Graph Laplacian* matrix:

$$L_{sym} \coloneqq D^{-\frac{1}{2}} L D^{-\frac{1}{2}} \tag{8.7}$$

Since L and L_{sym} are similar matrices they share the same eigenvalues and the eigenspace of 0 is spanned by the eigenvectors $D^{\frac{1}{2}}\mathbb{1}_{C_1}, \ldots, D^{\frac{1}{2}}\mathbb{1}_{C_k}$. Thus, given a sample $S \coloneqq \{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$, $\mathbf{x}_i \in \mathbb{R}^d$ with k clusters, if we correctly construct the adjacency matrix A we can solve the clustering problem by solving an eigenvalues problem of L_{sym} and then use the indicator eigenvectors to conclude the k connected components and assign membership to clusters.

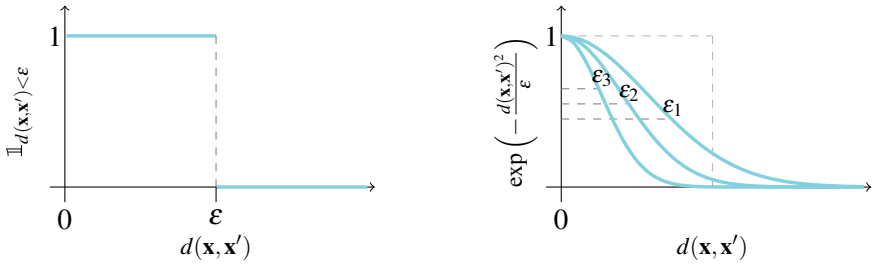
Measuring Affinity Between Samples

Next, we focus on how to determine if to add an edge between two samples and its weight. In the graph constructed above, this decision is simply made by checking if the (Euclidean) distance between the samples is smaller than some fixed $0 < \varepsilon$. This decision corresponds to a step function where all "close enough" samples are connected with an edge of equal weight, while further samples are not connected Figure 8.13a.

We could replace this function with some non-negative decreasing function such that instead of adding or not adding an edge we simply decrease the weight of an edge as the distance between the samples increases. One commonly used function is the Gaussian function (also referred to as Gaussian *kernel*):

$$A_{ij} \coloneqq \begin{cases} \exp\left(-\frac{||\mathbf{x}-\mathbf{x}'||^2}{\varepsilon}\right) & i \neq j \\ 0 & i = j \end{cases}$$

Seen in Figure 8.13b, this function assigns edges a high weight if the samples are very close (i.e. high affinity) and decays exponentially fast as the distance between the samples increases. The $\varepsilon > 0$ parameter controls how fast the decay is. As ε *decreases* the faster the decay. This in turn, will result in an affinity matrix where close samples are even closer while further samples are pushed further away, influencing the number of connected components (and thus clusters) in the graph.



(a) *ℓ_ε for edge weight* ***(b)*** *Gaussian decay for edge weight*

Figure 8.13: *Weight of edge between* $\mathbf{x}$ *and* $\mathbf{x}'$ *as function of the distance of* $\mathbf{x}'$ *from* $\mathbf{x}$

There exists a wide family of functions we could consider for the weights function. Though the choice (and that of the tuning parameters) depend on the problem in hand, it is important that the properties of the derived matrix L of the normalized Graph Laplacian are kept.

8.2.2.2 Normalized Spectral Clustering Algorithm

And so, using the translation from a data-partitioning problem to solving an eigenvalue problem, we derive the following algorithm of spectral clustering.

Algorithm 13 Normalized Spectral Clustering (Ng, Jordan and Weiss (2002))

procedure SPECTRAL-CLUSTERING(S, k) ▷ For S a pairwise distance matrix and k number of clusters
    Construct a similarity graph and let A be its weighted adjacency matrix.
    Compute the normalized Graph Laplacian L_{sym}.
    Compute the eigenvectors $\mathbf{u}_1, \ldots, \mathbf{u}_k$ corresponding the first k eigenvalues with multiplicities (ascending order).
    From $U \in \mathbb{R}^{m \times k}$ the matrix whose columns are the found eigenvectors derive the renormalized matrix

$$Y_{ij} = X_{ij} / \sqrt{\sum_l X_{il}^2}$$

    Treating each row of Y as a point in $\mathbb{R}^k$, cluster them into k clusters via K-means (or any other algorithm that attempts to minimize distortion).
    return assignment of original datapoints s_i to cluster j if and only if row i of the matrix Y was assigned to cluster j.
end procedure

8.3 Summary and Exercises

1. Let G be an undirected graph with non-negative weights and let L be the unnormalized graph Laplacian matrix as constructed in (8.6). Prove the following properties of L:
    - L is a symmetric PSD matrix.
    - The value 0 is an eigenvalue of L.
    - The multiplicity k of eigenvalue 0 corresponds to the number of connected components in G with the eigenspace of eigenvalue 0 spanned by the indicator vectors $\mathbb{1}_{C_1}, \ldots, \mathbb{1}_{C_k}$ where $C_1, \ldots, C_k$ denote all connected components in G.

9. Kernel Methods

Recall the hypothesis classes of linear regression and classification:

$$\begin{aligned} \mathcal{H}_{reg} &:= \left\{ \mathbf{x} \mapsto \mathbf{w}^\top \mathbf{x} + w_0 \;\middle|\; \mathbf{w} \in \mathbb{R}^d, w_0 \in \mathbb{R} \right\} \\ \mathcal{H}_{reg} &:= \left\{ \mathbf{x} \mapsto \text{sign}\left(\mathbf{w}^\top \mathbf{x} + w_0\right) \;\middle|\; \mathbf{w} \in \mathbb{R}^d, w_0 \in \mathbb{R} \right\} \end{aligned} \tag{9.1}$$

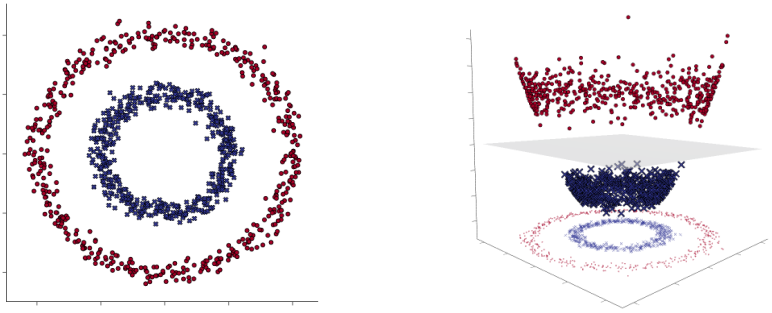
These hypothesis classes are of limited power. Consider the case where $\mathcal{X} \subset \mathbb{R}^2$ with samples as seen in Figure 9.1a. In this dataset the samples are not linearly separable and therefore algorithms matching the hypothesis class above will fail. Notice however, that if instead of using the original feature representation of the data, we embed the samples in a new feature space, such as mapping each sample $\mathbf{x} \mapsto \left(x_1^2, x_2^2, x_1^2 + x_2^2\right)^\top$, we get a linearly separable sample in $\mathbb{R}^3$ (Figure 9.1b). Over this new representation of the data we are able to separate the two labels using the SVM algorithm (found hyperplane represented in grey).

So, to increase the expressiveness of an hypothesis class, we can consider embedding the data into another (potentially of higher dimension) feature space, over which we then learn some predictor. Schematically:

- Given some domain set $\mathcal{X} \subseteq \mathbb{R}^d$ and a learning algorithm $\mathcal{A}$, select an embedding $\psi : \mathcal{X} \to \mathcal{F}$ for some feature space $\mathcal{F}$. In many cases we will want $\mathcal{F} \subseteq \mathbb{R}^k$ for some $k \gg d$ (and possibly $k = \infty$).
- Train $\mathcal{A}$ over a training set $\{(\psi(\mathbf{x}_i), y_i)\}_{i=1}^m$ using the hypothesis classes:

$$\begin{aligned} \mathcal{H}_\psi &:= \left\{ \mathbf{x} \mapsto \mathbf{w}^\top \psi(\mathbf{x}) + w_0 \;\middle|\; \mathbf{w} \in \mathbb{R}^k, w_0 \in \mathbb{R} \right\} \\ \mathcal{H}_\psi &:= \left\{ \mathbf{x} \mapsto \text{sign}\left(\mathbf{w}^\top \psi(\mathbf{x}) + w_0\right) \;\middle|\; \mathbf{w} \in \mathbb{R}^k, w_0 \in \mathbb{R} \right\} \end{aligned} \tag{9.2}$$

This approach seems very attractive as the assumption that our data can be described using some linear function (either for regression or classification) in *some* feature space does not look very limiting. The fact is that we have already encountered an example of such approach. In the case of polynomial fitting (section 2.4) given a sample in $x \in \mathbb{R}$ we first embedded the data in some new feature space where $\psi(x)_j = x^j$ and then learn using the linear regression hypothesis class. Later in this chapter we will expand this mapping to use multivariate polynomials Lemma 9.3.2.



(a) *Two class dataset that is not linearly separable*

(b) *Dataset mapped to* $\mathbb{R}^3$ *using the mapping* $(x_1,x_2) \to \left(x_1^2, x_2^2, x_1^2 + x_2^2\right)^\top$

Figure 9.1: Illustration of mapping to Feature Space*: Originally linearly inseparable dataset is linearly separable in mapped feature space.* Kernel Methods Examples

In general, in order to apply this strategy, we must address three challenges. The first, is to understand to which learning algorithms could we apply such an approach. The second, is that we must find an appropriate mapping ψ under which it makes sense to learn using a linear predictor. The third is that we want our algorithm to be computationally efficient. When we map the samples to $\mathcal{F}$, as we are often interested in an embedding where $k \gg d$ and as ψ could be very complicated, computing $\psi(\mathbf{x})$ might be computationally expensive. We would like to find a way to select $h_S \in \mathcal{H}$ without explicitly evaluating the higher-dimensional features.

9.1 An Altered Learning Problem

To understand which learning algorithms could use this approach, we begin with noticing that many of the optimization problems presented in earlier chapters can be formulated as

$$\mathbf{w}^* \coloneqq \underset{\mathbf{w}\in\mathcal{F}}{\operatorname{argmin}} \; f\left(\langle \mathbf{w}, \psi(\mathbf{x}_1)\rangle, \ldots, \langle \mathbf{w}, \psi(\mathbf{x}_m)\rangle\right) + R(||\mathbf{w}||) \tag{9.3}$$

for f an arbitrary function and $R : \mathbb{R}_+ \to \mathbb{R}$ a monotonically non-decreasing function. For example, in the case of the LS optimization problem f was the RSS function that given a prediction $\hat{y}_i$ returned $\sum(y_i - \hat{y}_i)^2$. In the case of the Soft-SVM optimization problem f calculated the mean hinge loss. For optimization problems taking the form seen in (9.3) there exists an optimal solution that can be represented as a linear combination of the given samples $\psi(\mathbf{x}_i)$.

Theorem 9.1.1 — The Representer Theorem. Let $\mathcal{X}$ be a non-empty domain set and $\psi : \mathcal{X} \to \mathcal{F}$ be a mapping into some Hilbert space. Consider an optimization problem over $\mathbf{x}_1, \ldots, \mathbf{x}_m \in \mathcal{X}$ of the form seen in (9.3). Then, there exists $\alpha \in \mathbb{R}^m$ such that $\mathbf{w}^* = \sum_{i=1}^m \alpha_i \psi(\mathbf{x}_i)$ is a minimizer of the optimization problem.

Proof. Let $\mathbf{w}^* \in \mathcal{F}$ be an optimal solution for the optimization problem. As $\mathcal{F}$ is a Hilbert space we can represent $\mathbf{w}^*$ as:

$$\mathbf{w}^* = \sum_{i=1}^m \alpha_i \psi(\mathbf{x}_i) + \mathbf{u}, \quad \operatorname{span}(\psi(\mathbf{x}_1), \ldots, \psi(\mathbf{x}_m)) \otimes \mathbf{u} = \mathcal{F}$$

Denote $\mathbf{w} = \mathbf{w}^* - \mathbf{u}$ and notice that

$$\langle \mathbf{w}^*, \psi(\mathbf{x}_i)\rangle = \langle \mathbf{w}^* - \mathbf{u}, \psi(\mathbf{x}_i)\rangle = \langle \mathbf{w}, \psi(\mathbf{x}_i)\rangle \quad i = 1, \ldots, m$$

Therefore both $\mathbf{w}^*$ and $\mathbf{w}$ obtain the same value for the fidelity term. Notice that since $||\mathbf{w}^*||^2 = ||\mathbf{w}||^2 + ||\mathbf{u}||^2 \geq ||\mathbf{w}||^2$ and R a monotonically non-decreasing function, then $R(||\mathbf{w}||^*) \geq R(||\mathbf{w}||)$. Thus, put together, the objecting of $\mathbf{w}$ is bound from above by the objective of $\mathbf{w}^*$ and as such $\mathbf{w}$ is also an optimal solution. ■

Based on the representer theorem we can reformulate (9.3) with respect to α. By replacing $\mathbf{w}$ with the found representation then

$$\langle \mathbf{w}, \psi(\mathbf{x}_i) \rangle = \left\langle \sum_{j=1}^{m} \alpha_j \psi(\mathbf{x}_j), \psi(\mathbf{x}_i) \right\rangle = \sum_{j=1}^{m} \alpha_j \left\langle \psi(\mathbf{x}_j), \psi(\mathbf{x}_i) \right\rangle \quad i = 1, \ldots, m \tag{9.4}$$

and

$$||\mathbf{w}||^2 = \left\langle \sum_{i=1}^{m} \alpha_i \psi(\mathbf{x}_i), \sum_{i=1}^{m} \alpha_i \psi(\mathbf{x}_i) \right\rangle = \sum_{i,j=1}^{m} \alpha_i \left\langle \psi(\mathbf{x}_i), \psi(\mathbf{x}_j) \right\rangle \alpha_j \tag{9.5}$$

By denoting $G \in \mathbb{R}^{m \times m}$ the Gram matrix whose entries are $G_{i,j} = \left\langle \psi(\mathbf{x}_i), \psi(\mathbf{x}_j) \right\rangle$ we rewrite (9.3) as

$$\boldsymbol{\alpha}^* := \underset{\boldsymbol{\alpha} \in \mathbb{R}^m}{\operatorname{argmin}} \; f(G\boldsymbol{\alpha}) + R\left(\boldsymbol{\alpha}^\top G \boldsymbol{\alpha}\right) \tag{9.6}$$

This *dual* representation of the former (primal) optimization problem searches for a solution $\boldsymbol{\alpha} \in \mathbb{R}^m$ (i.e. the number of training samples) while the former optimization problem (9.3) searches for a solution $\mathbf{w} \in \mathcal{F}$ which can be arbitrary large (or as mentioned even of infinite dimension). In addition, notice that the dual problem is a quadratic problem in $\boldsymbol{\alpha}$ and therefore, if G can be computed efficiently, then the entire problem can be solved efficiently. Once we have solved (9.6) for $\boldsymbol{\alpha}$ we can derive the optimal solution for the primal optimization problem and predict the response of a new sample by:

$$\hat{y}(\mathbf{x}) = \langle \mathbf{w}^*, \psi(\mathbf{x}) \rangle = \left\langle \sum_{j=1}^{m} \alpha_j \psi(\mathbf{x}_j), \psi(\mathbf{x}) \right\rangle = \sum_j \alpha_j \left\langle \psi(\mathbf{x}_j), \psi(\mathbf{x}) \right\rangle = \boldsymbol{\alpha}^\top \mathbf{k} \tag{9.7}$$

where $k_i = \left\langle \psi(\mathbf{x}_i), \psi(\mathbf{x}) \right\rangle$, $i = 1, \ldots, m$.

9.2 Kernelized Algorithms

Based on the Representer Theorem (9.1.1), if an optimization problem is in the specified form, we can derive a kernelized version of the algorithm by replacing $\mathbf{x}$ with $\psi(\mathbf{x})$, $\mathbf{w}$ with $\boldsymbol{\alpha}$ and solve the optimization problem with respect to $\boldsymbol{\alpha}$. Below are a few examples of algorithms covered in previous chapters but adapted for a *kernelized* version.

9.2.1 Kernel SVM

Recall the optimization problem of the homogenous Hard-SVM (3.16)

$$\mathbf{w}^* := \underset{\mathbf{w} \in \mathbb{R}^d}{\operatorname{argmin}} \; ||\mathbf{w}||^2 \quad \text{s.t.} \quad y_i \langle \mathbf{w}, \mathbf{x}_i \rangle \geq 1 \quad \forall i \in [m]$$

We derive the dual problem by replacing $\mathbf{x}$ with $\psi(\mathbf{x})$, $\mathbf{w}$ with $\boldsymbol{\alpha}$ and solving for $\boldsymbol{\alpha}$. So for the:

$$\begin{aligned}
&\text{Objective:} && ||\mathbf{w}||^2 &&= \left(\textstyle\sum_{i=1}^m \alpha_j \psi(\mathbf{x}_i)\right)^2 \\
& && &&= \textstyle\sum_{i,j=1}^m \alpha_i \left\langle \psi(\mathbf{x}_i), \psi(\mathbf{x}_j) \right\rangle \alpha_j \\
&\text{Constraints:} && y_i \langle \mathbf{w}, \psi(\mathbf{x}_i) \rangle &&= y_i \left\langle \textstyle\sum_j \alpha_j \psi(\mathbf{x}_j), \psi(\mathbf{x}_i) \right\rangle \\
& && &&= y_i \textstyle\sum_j \alpha_j \left\langle \psi(\mathbf{x}_j), \psi(\mathbf{x}_i) \right\rangle \quad \forall i \in [m]
\end{aligned}$$

which in matrix notation yields the kernelized Hard-SVM optimization problem:

$$\boldsymbol{\alpha}^* := \underset{\boldsymbol{\alpha}\in\mathbb{R}^m}{\operatorname{argmin}} \ \boldsymbol{\alpha}^\top G\boldsymbol{\alpha} \quad \text{s.t.} \quad y_i\left(G\boldsymbol{\alpha}\right)_i \geq 1 \quad \forall i \in [m] \tag{9.8}$$

where $G_{ij} = \left\langle \psi(\mathbf{x}_i), \psi(\mathbf{x}_j) \right\rangle$. Notice that this is a quadratic optimization problem with linear constraints and can therefore be solved efficiently.

Similarly, let us derive a kernelized Soft-SVM algorithm. The homogenous Soft-SVM optimization problem is (**??**):

$$\mathbf{w}^* := \underset{\mathbf{w}\in\mathbb{R}^d}{\operatorname{argmin}} \ \lambda \left|\left|\mathbf{w}\right|\right|^2 + \frac{1}{m}\sum_i \max\left\{0, 1 - y_i \left\langle \mathbf{w}, \mathbf{x}_i \right\rangle\right\}$$

As before, we replace $\mathbf{x}$ by $\psi(\mathbf{x})$ and solve for $\boldsymbol{\alpha}$ to obtain the kernel Soft-SVM optimization problem:

$$\boldsymbol{\alpha}^* := \underset{\boldsymbol{\alpha}\in\mathbb{R}^m}{\operatorname{argmin}} \ \lambda \boldsymbol{\alpha}^\top G\boldsymbol{\alpha} + \frac{1}{m}\sum_i \max\left\{0, 1 - y_i \left(G\boldsymbol{\alpha}_i\right)\right\} \tag{9.9}$$

9.2.2 Kernel Ridge Regression

Recall the Ridge Regression optimization problem (6.0.2.2) which jointly minimizes the RSS and the ℓ_2 norm of the coefficients vector $\mathbf{w}$:

$$\mathbf{w}^* := \underset{\mathbf{w}\in\mathbb{R}^d}{\operatorname{argmin}} \ \left|\left|\mathbf{y} - \mathbf{X}\mathbf{w}\right|\right|^2 + \lambda \left|\left|\mathbf{w}\right|\right|^2$$

Let us derive the dual representation of the optimization problem by replacing $\mathbf{x}$ with $\psi(\mathbf{x})$, $\mathbf{w}$ with $\boldsymbol{\alpha}$ and solving for $\boldsymbol{\alpha}$. For the fidelity term then:

$$\begin{aligned}
\textstyle\sum_{i=1}^m \left(y_i - \left\langle \psi(\mathbf{x}_i), \mathbf{w} \right\rangle\right)^2 &= \textstyle\sum_{i=1}^m \left(y_i - \left\langle \psi(\mathbf{x}_i), \sum_{j=1}^m \alpha_j \psi(\mathbf{x}_j) \right\rangle\right)^2 \\
&= \textstyle\sum_{i=1}^m \left(y_i - \sum_{j=1}^m \alpha_j \left\langle \psi(\mathbf{x}_i), \psi(\mathbf{x}_j) \right\rangle\right)^2 \\
&= \textstyle\sum_{i=1}^m \left(y_i^2 - 2y_i \sum_{j=1}^m \alpha_j \left\langle \psi(\mathbf{x}_i), \psi(\mathbf{x}_j) \right\rangle + \left(\sum_{j=1}^m \alpha_j \left\langle \psi(\mathbf{x}_i), \psi(\mathbf{x}_j) \right\rangle\right)^2\right)
\end{aligned}$$

and for the regularization term, as seen in the case of the kernel Hard-SVM then $\lambda \left|\left|\mathbf{w}\right|\right|^2 = \lambda \boldsymbol{\alpha}^\top G\boldsymbol{\alpha}$. In matrix notation we obtain the following optimization problem:

$$\boldsymbol{\alpha}^* := \underset{\boldsymbol{\alpha}\in\mathbb{R}^m}{\operatorname{argmin}} \ \left|\left|\mathbf{y}\right|\right|^2 - 2y^\top G\boldsymbol{\alpha} + \boldsymbol{\alpha}^\top G^2\boldsymbol{\alpha} + \lambda \boldsymbol{\alpha}^\top G\boldsymbol{\alpha}$$

Denote the function by f. We equate the gradient of f with respect to α to zero:

$$\begin{gathered}
\nabla_{\boldsymbol{\alpha}} f(\boldsymbol{\alpha}) = 2G^2\boldsymbol{\alpha} - 2G\mathbf{y} + 2\lambda G\boldsymbol{\alpha} = 0 \\
\Downarrow \\
G(G + \lambda I_m)\boldsymbol{\alpha} = G\mathbf{y}
\end{gathered}$$

Since G is a PSD matrix then $G + \lambda I_m$ is a PD matrix and is invertible. Thus, the minimizer of the dual problem is given by

$$\hat{\boldsymbol{\alpha}} = \left(G + \lambda I_m\right)^{-1}\mathbf{y}$$

Using this minimizer we can now express the prediction over a new sample $\mathbf{x}$ by:

$$\begin{aligned}
\hat{y}(\mathbf{x}) &= \left\langle \mathbf{w}, \psi(\mathbf{x}) \right\rangle \\
&= \textstyle\sum_{i=1}^m \alpha_i \left\langle \psi(\mathbf{x}_i), \psi(\mathbf{x}) \right\rangle \\
&\overset{(*)}{=} \mathbf{k}^\top \boldsymbol{\alpha} \\
&= \mathbf{k}^\top \left(G + \lambda I_m\right)^{-1}\mathbf{y}
\end{aligned}$$

where $k_i = \left\langle \psi(\mathbf{x}_i), \psi(\mathbf{x}) \right\rangle$. Notice that both G and the expression derived in the prediction only access the samples though inner-products of their mappings. Thus, we can simply replace the inner-products with the easy-to-compute PSD kernel function and therefore solve this optimization problem efficiently.

9.2.3 Kernel Regularized Logistic Regression

Similar to the Kernel Ridge Regression algorithm, we can derive a kernelized version for the regularized Logistic Regression classifier. For simplicity let the regularization function be the ℓ_2 norm of $\mathbf{w}$. Then, the objective of the ℓ_2-regularized Logistic Regression is

$$f(\mathbf{w}) = \sum_{i=1}^{m} \left[\log\left(1 + e^{\langle \mathbf{x}_i, \mathbf{w} \rangle}\right) - y_i \langle \mathbf{x}_i, \mathbf{w} \rangle \right] + \lambda ||\mathbf{w}||^2$$

By replacing $\mathbf{x}$ with $\psi(\mathbf{x})$ and $\mathbf{w}$ with α we derive the following objective function:

$$\begin{aligned}
f(\alpha) &= \sum_{i=1}^{m} \left[\log\left(1 + e^{\left\langle \psi(\mathbf{x}_i), \sum_j \alpha_j \psi(\mathbf{x}_j) \right\rangle}\right) - y_i \left\langle \psi(\mathbf{x}_i), \sum_j \alpha_j \psi(\mathbf{x}_j) \right\rangle \right] + \lambda \alpha^\top G \alpha \\
&= \sum_{i=1}^{m} \left[\log\left(1 + e^{\sum_j \alpha_j \langle \psi(\mathbf{x}_i), \psi(\mathbf{x}_j) \rangle}\right) - y_i \sum_j \alpha_j \left\langle \psi(\mathbf{x}_i), \psi(\mathbf{x}_j) \right\rangle \right] + \lambda \alpha^\top G \alpha \\
&= \sum_{i=1}^{m} \log\left(1 + e^{[G\alpha]_i}\right) - \sum_{i=1}^{m} y_i [G\alpha]_i + \lambda \alpha^\top G \alpha \\
&= \sum_{i=1}^{m} \log\left(1 + e^{[G\alpha]_i}\right) - \mathbf{y}^\top G \alpha + \lambda \alpha^\top G \alpha
\end{aligned}$$

To find the minimizer we equate the derivative with respect to each coordinate of α to zero:

$$\begin{aligned}
\frac{\partial f(\alpha)}{\partial \alpha_j} &= \sum_{i=1}^{m} \frac{\partial \log\left(1 + e^{[G\alpha]_i}\right)}{\partial \alpha_j} - \frac{\partial \mathbf{y}^\top G \alpha}{\partial \alpha_j} + \lambda \frac{\partial \alpha^\top G \alpha}{\partial \alpha_j} \\
&= \sum_{i=1}^{m} \frac{1}{1 + \exp(-[G\alpha]_i)} \cdot \frac{\partial e^{[G\alpha]_i}}{\partial \alpha_j} - [G\mathbf{y}]_j + 2\lambda [G\alpha]_j \\
&= \sum_{i=1}^{m} \frac{\exp([G\alpha]_i)}{1 + \exp(-[G\alpha]_i)} \cdot \frac{\partial [G\alpha]_i}{\partial \alpha_j} - [G\mathbf{y}]_j + 2\lambda [G\alpha]_j \\
&= \sum_{i=1}^{m} G_{ij} \cdot \frac{\exp([G\alpha]_i)}{1 + \exp(-[G\alpha]_i)} - [G\mathbf{y}]_j + 2\lambda [G\alpha]_j \quad = \quad 0
\end{aligned}$$

Which can be written as follows:

$$\sum_{i=1}^{m} G_{ij} \left(\frac{1}{1 + \exp\left(-[G\alpha]_i\right)} + 2\lambda \alpha_i \right) = \sum_{i=1}^{m} G_{ij} y_i$$

Therefore the minimizers $\alpha_1, \ldots, \alpha_m$ are the solutions to the following transcendental equations

$$\frac{1}{1 + \exp\left(-[G\alpha]_i\right)} + 2\lambda \alpha_i = y_i$$

9.2.4 Kernel PCA

Another algorithm that can be kernalized is the PCA algorithm and is known as the Kernel PCA (Schölkopf *et al.,* 1998). In this algorithm we solve the PCA problem in some feature space $\mathcal{F}$, rather than using the original coordinates. To show we can apply the Kernel Trick to the PCA algorithm we must first show that we can re-write PCA such that accessing the data is done only through inner products of the samples. In addition we would need to show that projecting samples onto the found subspace can be done only through inner products. Then, if we are able to do so, we can replace $\mathbf{x}$ with $\psi(\mathbf{x})$ substitute the inner product with the associated PSD kernel function.

Let $\mathbf{X} \in \mathbb{R}^{m \times d}$ be a *centered* design matrix. We saw that when we solve the PCA optimization problem we are in fact solving an eigenvalues problem for the sample covariance matrix $C = \mathbf{X}^\top \mathbf{X} = \sum_i \mathbf{x}_i \mathbf{x}_i^\top$ (Lemma 8.1.2). We would like to represent this sum of outer-products via inner-products.

Let $\mathbf{v}$ be an eigenvector of C corresponding to eigenvalue $\lambda \neq 0$ so $\lambda \mathbf{v} = C\mathbf{v}$. This means that $\mathbf{v} \in Im(C)$ and

therefore is in the span of $\mathbf{x}_1,\ldots,\mathbf{x}_m$. As such, solving an eigenvalue problem for C is equivalent to finding a vector $\mathbf{v}$ for which :

$$\lambda \langle \mathbf{x}_i, \mathbf{v}\rangle = \langle \mathbf{x}_i, C\mathbf{v}\rangle \quad i = 1,\ldots,m \tag{9.10}$$

In addition, by the definition of C notice that $\lambda\mathbf{v} = C\mathbf{v} = \sum_i \mathbf{x}_i\mathbf{x}_i^\top \mathbf{v} = \sum_i \langle \mathbf{x}_i, \mathbf{v}\rangle \mathbf{x}_i$ and therefore $\mathbf{v} = \sum_i \frac{1}{\lambda}\langle \mathbf{x}_i, \mathbf{v}\rangle \mathbf{x}_i$. By denoting $\alpha_i = \frac{1}{\lambda}\langle \mathbf{x}_i, \mathbf{v}\rangle$ we get that $\mathbf{v} = \sum_{i=1}^m \alpha_i \mathbf{x}_i = \mathbf{X}^\top \alpha$. Now, rather than solving the equations in (9.10) notice the following. Fix some $i \in [m]$ and then:

$$\begin{aligned}
\lambda \left\langle \mathbf{x}_i, \textstyle\sum_j \alpha_j \mathbf{x}_j \right\rangle &= \left\langle \mathbf{x}_i, C\left(\textstyle\sum_j \alpha_j \mathbf{x}_j\right)\right\rangle \\
&= \textstyle\sum_j \alpha_j \left\langle \mathbf{x}_i, C\mathbf{x}_j \right\rangle \\
&= \textstyle\sum_j \alpha_j \left\langle \mathbf{x}_i, \sum_l \mathbf{x}_l\mathbf{x}_l^\top \mathbf{x}_j \right\rangle \\
&= \textstyle\sum_j \alpha_j \sum_l \left\langle \mathbf{x}_l, \mathbf{x}_j \right\rangle \left\langle \mathbf{x}_i, \mathbf{x}_l \right\rangle \\
&\Downarrow \\
\lambda \textstyle\sum_j \alpha_j \left\langle \mathbf{x}_i, \mathbf{x}_j \right\rangle &= \textstyle\sum_j \alpha_j \sum_l \left\langle \mathbf{x}_l, \mathbf{x}_j \right\rangle \left\langle \mathbf{x}_i, \mathbf{x}_l \right\rangle
\end{aligned}$$

Which in matrix notation is $\lambda G\alpha = G^2\alpha$ where we define G to be the Gram matrix over $\mathbf{x}_1,\ldots,\mathbf{x}_m$: $G_{ij} = \mathbf{x}_i^\top \mathbf{x}_j$, or equivalently $G = \mathbf{X}\mathbf{X}^\top$. For eigenvectors of G, not corresponding to zero eigenvalues the solution of $\lambda G\alpha = G^2\alpha$ is equivalent to that of $\lambda\alpha = G\alpha$.

Therefore, once we find $\alpha^{(1)},\ldots,\alpha^{(m)}$ the eigenvectors of G we can then:

- Obtain the eigenvectors of C by $\mathbf{v}^{(l)} = \sum_{i=1}^m \alpha_i^{(l)}\mathbf{x}_i$.
- Project the data-points on the low dimension subspace by:

$$\widetilde{\mathbf{x}}_l := \left\langle \mathbf{v}^{(l)}, \mathbf{x} \right\rangle = \sum_i \alpha_i^{(l)} \langle \mathbf{x}_i, \mathbf{x}\rangle$$

As we were able to represent the original PCA problem and the projection operating using inner-products of samples we can now apply the Kernel Trick to the PCA and substitute the inner-products with some PSD kernel function. Thus, for a feature map ψ where $C = \sum_i \psi(\mathbf{x}_i)\psi(\mathbf{x}_i)^\top$:

- Solve the eigenvalue problem $\lambda\alpha = G\alpha$, for $G_{ij} = \left\langle \psi(x_i), \psi(\mathbf{x}_j)\right\rangle$.
- Project the data-points by:

$$\widetilde{\mathbf{x}}_l := \left\langle \mathbf{v}^{(l)}, \psi(\mathbf{x}) \right\rangle = \sum_i \alpha_i^{(l)} \langle \psi(\mathbf{x}_i), \psi(\mathbf{x})\rangle = \sum_i \alpha_i^{(l)} k(\mathbf{x}_i, \mathbf{x})$$

Centering Matrix In Feature Space

The above derivation assumes the given design matrix $\mathbf{X}$ is centered. Unlike in the case of PCA, in the Kernel PCA we cannot compute the mean and reduce it as it would require to compute $\forall i\ \phi(\mathbf{x}_i)$. Therefore, we would need to use a different approach:

Lemma 9.2.1 Let G be the Gram matrix of ψ over the dataset $\mathbf{X}$. The centered Gram matrix to be used in the Kernel PCA algorithm is given by:

$$\widetilde{G} = G - \mathbf{1}_m G - G\mathbf{1}_m + \mathbf{1}_m G\mathbf{1}_m$$

Proof. Denote $\widetilde{\psi}(\mathbf{x})$ the projected data-point after centering:

$$\widetilde{\psi}(\mathbf{x}) = \psi(\mathbf{x}) - \frac{1}{m}\sum_l \psi(\mathbf{x}_l)$$

where $\mathbf{1}_m \in \mathbb{R}^{m\times m}$, $[\mathbf{1}_m]_{ij} = 1/m$. So the elements of the centered Gram matrix are given by:

$$\begin{aligned}
\widetilde{G}_{ij} &= \langle \widetilde{\psi}(\mathbf{x}_i), \widetilde{\psi}(\mathbf{x}-j)\rangle \\
&= \left\langle \psi(\mathbf{x}_i) - \tfrac{1}{m}\textstyle\sum_l \psi(\mathbf{x}_l), \psi(\mathbf{x}_j) - \tfrac{1}{m}\sum_k \psi(\mathbf{x}_k)\right\rangle \\
&= \psi(\mathbf{x}_i)^\top \psi(\mathbf{x}_j) - \tfrac{1}{m}\textstyle\sum_k \psi(\mathbf{x}_i)^\top \psi(\mathbf{x}_k) - \tfrac{1}{m}\sum_l \psi(\mathbf{x}_l)^\top \psi(\mathbf{x}_j) + \tfrac{1}{m^2}\sum_{l,k} \psi(\mathbf{x}_l)^\top \psi(\mathbf{x}_k) \\
&= G_{ij} - \tfrac{1}{m}\textstyle\sum_l G_{il} - \tfrac{1}{m^2}\sum_k G_{jk} + \tfrac{1}{m}\sum_{l,k} G_{lk}
\end{aligned}$$

which in matrix notation is $\widetilde{\psi}(\mathbf{x}) = \psi(\mathbf{x}) - \frac{1}{m}\sum_l \psi(\mathbf{x}_l)$. ■

Put all together, the pseudo-code for the Kernel PCA algorithm is:

Algorithm 14 Kernel-PCA

procedure KERNEL-PCA($\mathbf{X}$, k, l) ▷ k the kernel computing ϕ and l the dimension to reduce to
Compute the centered Gram matrix (9.2.1):

$$\widetilde{K} = K - \mathbf{1}_m K - K\mathbf{1}_m + \mathbf{1}_m K \mathbf{1}_m, \quad K_{ij} \coloneqq k(\mathbf{x}_i, \mathbf{x}_j)$$

Let $\alpha^{(1)}, \ldots, \alpha^{(l)}$ be the eigenvectors of $\widetilde{K}$ corresponding the largest eigenvalues.
Compute the corresponding eigenvectors of A by: $\mathbf{v}^{(j)} \coloneqq \sum_{i=1}^{m} \alpha_i^{(j)} \mathbf{x}_i$
return $\mathbf{v}^{(1)}, \ldots, \mathbf{v}^{(l)}$
end procedure

R The PCA algorithm seen in Algorithm 11 diagonalizes the sample covariance matrix, which is a d-by-d matrix and has a time complexity of $\mathcal{O}(d^3)$. If $d \gg m$ this can be computationally expensive. Though the proof of the Kernel PCA algorithm we have actually shown that we can solve the "normal" PCA by computing $\mathbf{X}\mathbf{X}^\top$ instead of using $\mathbf{X}^\top\mathbf{X}$. This means that time complexity is reduced to $\mathcal{O}(m^3)$ for computing $\mathbf{X}\mathbf{X}^\top$ and then $\mathcal{O}(m^2 \cdot d)$ for adjusting the eigenvalues of $\mathbf{X}\mathbf{X}^\top$ to those of $\mathbf{X}^\top\mathbf{X}$.

9.3 Characterizing Kernel Functions

In the section above we have shown that given some map $\psi : \mathcal{X} \to \mathcal{F}$ where $\mathcal{F}$ some Hilbert space we can solve the dual optimization problem with respect to $\alpha \in \mathbb{R}^m$ and then use α to predict the response of a given new sample.

However, as the mapped feature space might be of an arbitrary large dimension (or even of infinite dimension), computing the Gram matrix G or the predicted value of a new sample can be computationally expensive. To address this problem we will next apply the idea of *kernel substitution*. We will decide to use a (wide) specific family of functions, called *PSD kernel functions*, that even though might map to a very high dimension, can still be computed efficiently.

Definition 9.3.1 Let $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ be some function. k is called a *kernel function* if k is symmetric, i.e. $\forall \mathbf{x}, \mathbf{x}' \in \mathcal{X}\ k(\mathbf{x}, \mathbf{x}') = k(\mathbf{x}', \mathbf{x})$.

Definition 9.3.2 Let $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ be a kernel function. k is called a *Positive Semi-Definite kernel* if and only if for any $m \in \mathbb{N}$ and for any $\mathbf{x}_1, \ldots, \mathbf{x}_m \in \mathcal{X}$ the matrix $K \in \mathbb{R}^{m \times m}$ whose entries are $K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$ is a PSD matrix. K is referred to as the associated kernel matrix of k over $\mathbf{x}_1, \ldots, \mathbf{x}_m$.

To check if a given function is a kernel function we could: (1) show the associated Gram matrix is PSD or (2) find the transformation ψ that satisfies that $\langle \psi(\mathbf{x}), (\mathbf{x}') \rangle$.

■ **Example 9.1** Consider the following function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ for $\mathcal{X} = \mathbb{R}^d$ defined as $k(\mathbf{x}, \mathbf{x}') = 1,\ \mathbf{x}, \mathbf{x}' \in \mathcal{X}$. Let us show that this is a PSD-kernel function as find a possible mapping function ψ such that $k(\mathbf{x}, \mathbf{x}') = \langle \psi(\mathbf{x}), \psi(\mathbf{x}') \rangle$.

Let $\mathbf{x}_1,\dots\mathbf{x}_m \in \mathcal{X}$. We begin with showing that the associated Gram matrix of k over $\mathbf{x}_1,\dots,\mathbf{x}_m$ is a PSD matrix. The Gram matrix of k over this set is: $G_{ij} = k(\mathbf{x}_i,\mathbf{x}_j) = 1$. Notice that as k is symmetric then G is a symmetric matrix. In addition, the eigenvalues of G are $m, 0 \geq 0$. Thus, $G \in PSD$ and k a valid PSD-kernel function.

As for finding a mapping function ψ, it must satisfy that $\langle \psi(\mathbf{x}), (\mathbf{x}') \rangle = 1$ for any $\mathbf{x},\mathbf{x}' \in \mathbb{R}^d$. So we could choose $\psi : \mathbb{R}^d \to \mathbb{R}^k$ that always returns some unit vector $\mathbf{v}$. Then $\langle \psi(\mathbf{x}), (\mathbf{x}') \rangle = \mathbf{v}^\top \mathbf{v} = ||\mathbf{v}|| = 1$. Notice that we have not specified the dimension of the feature space, which could be of any size we desire. ■

■ **Example 9.2** Consider the standard inner product in $\mathbb{R}^d$. Let us show that this is a PSD-kernel function. Let $\mathbf{x}_1,\dots,\mathbf{x}_m \in \mathbb{R}^d$ and the matrix G whose entries are $G_{ij} = \langle \mathbf{x}_i, \mathbf{x}_j \rangle$. Notice that as we can write G as $G = \mathbf{X}^\top \mathbf{X}$ where the i'th row of $\mathbf{X}$ is the i'th sample. Then G is a PSD matrix and thus k a PSD-kernel function. For this PSD-kernel function a possible mapping function can be the identity function $\psi(\mathbf{x}) \equiv \mathbf{x}$. ■

Interestingly, we are able to tie between the mapping functions ψ discussed in the previous section and PSD-kernel functions using the following (simplified) condition.

Theorem 9.3.1 — Mercer's Condition. Let $\psi : \mathcal{X} \to \mathcal{F}$ where $\mathcal{F}$ some Hilbert space. Then there exists a symmetric function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ implementing an inner product in $\mathcal{F}$ if and only if k is a PSD-kernel; namely, for all $\mathbf{x}_1,\dots,\mathbf{x}_m$ $G_{ij} := k(\mathbf{x}_i,\mathbf{x}_j)$ and $k(\mathbf{x}_i,\mathbf{x}_j) = \langle \psi(\mathbf{x}_i), \psi(\mathbf{x}_i) \rangle$, the matrix G is PSD.

This condition tells us how we should choose ψ. We expressed the dual optimization problem (9.6) and the prediction (9.7) as *inner products* of the mappings of samples $\langle \psi(\mathbf{x}), \psi(\mathbf{x}') \rangle$. Therefore, if we choose ψ such that the Gram matrix $G_{ij} = \langle \psi(\mathbf{x}_i), \psi(\mathbf{x}_j) \rangle$ is a PSD matrix then there exists a PSD-kernel function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ which yields the same outputs but is agnostic to the (high dimensional) feature space $\mathcal{F}$ and can be calculated efficiently.

9.3.1 The Polynomial- and Gaussian Kernel Functions

One commonly used kernel function is the polynomial kernel. It maps a given sample to the feature space whose coordinates correspond to all the monomials of degree at most k. This function extends the transformation seen in polynomial fitting (section 2.4).

■ **Example 9.3** Before proving the general case let us consider the polynomial kernel in $\mathbb{R}^2$. Let $\mathbf{x},\mathbf{x}' \in \mathcal{X} = \mathbb{R}^2$ with $x_0, x_0' = 1$ and $k = 2$ so:

$$\begin{aligned}(1+\langle \mathbf{x},\mathbf{x}' \rangle)^2 &= (1+x_1x_1'+x_2x_2')^2 \\ &= 1+2x_1x_1'+2x_2x_2'+(x_1x_1')^2+(x_2x_2')^2+2x_1x_1'x_2x_2'\end{aligned}$$

For ψ defined by $\psi(\mathbf{x}) = \left(1, 1\cdot x_1, x_1\cdot 1, 1\cdot x_2, x_2\cdot 1, x_1^2, x_2^2, x_1\cdot x_2, x_2\cdot x_1\right)^\top$ we achieve that $k(\mathbf{x},\mathbf{x}') = \psi(\mathbf{x})^\top \psi(\mathbf{x}')$. ■

Claim 9.3.2 — Polynomial Kernels. Let $\mathcal{X} = \mathbb{R}^d$ and consider the function

$$k(\mathbf{x},\mathbf{x}') := (1+\langle \mathbf{x},\mathbf{x}' \rangle)^k$$

This is a valid PSD-kernel function corresponding the mapping:

$$\mathbf{x} \mapsto \left(1,\dots,x_i,\dots,\mathbf{x}_i\cdot\mathbf{x}_j,\dots,\prod_{\substack{i\in J \\ J\subset[d],|J|=k}} x_i\right)$$

Proof. To show this is a valid kernel, we will find some mapping ψ such that $K(\mathbf{x},\mathbf{x}')$ equals to $\psi(\mathbf{x})^\top \psi(\mathbf{x}')$. For simplicity let $x_0 = 1 = x_0'$. Then:

$$\begin{aligned}(1+\langle \mathbf{x},\mathbf{x}'\rangle)^k &= \left(\sum_{i=0}^d x_i x_i'\right)^k \\ &= \sum_{J\in\{0,\dots,d\}^k} \prod_{i=1}^k x_{J_i} x'_{J_i} \\ &= \sum_{J\in\{0,\dots,d\}^k} \prod_{i=1}^k x_{J_i} \prod_{i=1}^k x'_{J_i} \\ &= \sum_{J\in\{0,\dots,d\}^k} \psi(\mathbf{x})_J \, \psi(\mathbf{x}')_J \\ &= \psi(\mathbf{x})^\top \psi(\mathbf{x}')\end{aligned}$$

Where we define $\psi : \mathbb{R}^d \to \mathbb{R}^{(d+1)^k}$ such that each coordinate of $\psi(\mathbf{x})$ corresponds to some $J \in \{0,\dots,d\}^k$ and is equal to $\prod_{i=1}^k x_{J_i}$. Therefore we obtained that $k(\mathbf{x},\mathbf{x}') = \langle \psi(\mathbf{x}), \psi(\mathbf{x}')\rangle$ ■

Using the polynomial kernel and the associated mapping function ψ, let us return to the question of the cost of computing the optimization problem. If we were to solve the optimization problem as presented in (9.6) we would have to compute the Gram matrix G by calculating for every cell i,j the value $G_{ij} = \left\langle \psi(\mathbf{x}_i), \psi(\mathbf{X}_j)\right\rangle$. Even without considering computation times of the mappings themselves, the computation of the inner product in the mapped feature space would take $\mathcal{O}((d+1)^k)$ compared to $\mathcal{O}(d)$ in the optimization problem prior to the mapping. However, if we indeed do apply the *kernel substitution* idea and rather than explicitly compute the mapping (and thus the inner product in the feature space) only compute the kernel function then computing $(1+\langle \mathbf{x},\mathbf{x}'\rangle)^k$ for $\mathbf{x},\mathbf{x}' \in \mathbb{R}^d$ is an $\mathcal{O}(d)$ operation. This improvement in computation time holds also when performing predictions.

R The polynomial kernel function presented above maps to all monomials of degree *at most* k. It can be shown that the function $k(\mathbf{x},\mathbf{x}') = \langle \mathbf{x},\mathbf{x}'\rangle^k$ is also a valid PSD-kernel and maps to the space of all monomials of degree *exactly* k.

Another powerful and commonly used kernel function is the Gaussian kernel.

Claim 9.3.3 — Gaussian Kernels. The following is a valid kernel function:

$$k(\mathbf{x},\mathbf{x}') := \exp\left(-\frac{1}{2\sigma^2}\left|\left|\mathbf{x}-\mathbf{x}'\right|\right|^2\right), \quad \sigma^2 \in \mathbb{R}_+$$

with the mapping function ψ defined such that:

$$\forall n \in \mathbb{N} \quad \psi(\mathbf{x})_n := \frac{1}{\sqrt{n!}} \exp\left(-x^2/2\sigma^2\right) x^n$$

Proof. Let us begin with showing that $k(\mathbf{x},\mathbf{x}') = \psi(\mathbf{x})^\top \psi(\mathbf{x}')$:

$$\begin{aligned}\psi(\mathbf{x})^\top \psi(\mathbf{x}') &= \sum_{n=0}^\infty \left(\tfrac{1}{\sqrt{n!}} \exp\left(-x^2/2\sigma^2\right) x^n\right)\left(\tfrac{1}{\sqrt{n!}} \exp\left(-\left(\mathbf{x}'\right)^2/2\sigma^2\right)\left(\mathbf{x}'\right)^n\right) \\ &= \exp\left(-\tfrac{\mathbf{x}^2+\left(\mathbf{x}'\right)^2}{2\sigma^2}\right)\sum_{n=0}^\infty \tfrac{\left(\mathbf{x}\mathbf{x}'\right)^n}{n!} \\ &= \exp\left(-\tfrac{1}{2\sigma^2}\left|\left|\mathbf{x}-\mathbf{x}'\right|\right|^2\right)\end{aligned}$$

■

Consider the feature space of the primal optimization problem for which we use the Gaussian kernel. Notice that as ψ maps to a space of infinite dimension, the primal optimization problem is one over infinitely many parameters. Using the dual representation however, we are able to find an equivalent optimal solution over

a space with m parameters. In terms of computing the elements seen in the dual representation, by kernel substitution we understand that we do not have to explicitly compute $\psi(\mathbf{x}), \psi(\mathbf{x}')$ and there inner product in the feature space but only calculate the kernel function - an operation performed in $\mathcal{O}(d)$.

R The Gaussian kernel function is a specific case of the wider Radial Basis Function (RBF) kernel $k(\mathbf{x},\mathbf{x}') := \exp\left(-\beta ||\mathbf{x}-\mathbf{x}'||^2\right)$.

9.3.2 Closure Properties For PSD-Kernels

Since a function is a valid PSD-kernel function if the associated Gram matrix is PSD we can derive several closure properties for PSD-kernel functions. Here are some of these properties.

Claim 9.3.4 Let k be some valid kernel function then the following are valid kernel functions:
1. $k(\mathbf{x},\mathbf{y}) = \mathbf{x}^\top A\mathbf{y}$, where $A \in PSD$.
2. $c \cdot k_1(\mathbf{x},\mathbf{y})$ where $c > 0$.
3. $exp(k_1(\mathbf{x},\mathbf{y}))$
4. $f(\mathbf{x}) k_1 f(\mathbf{y})$

Using these closure properties we are able to construct new PSD-kernel functions and derive a very rich family of functions.

■ **Example 9.4** Using the closure properties above, let us show that the Gaussian kernel function is indeed a valid PSD-kernel.

$$\begin{aligned} k(\mathbf{x},\mathbf{x}') &= \exp\left(-||\mathbf{x}-\mathbf{x}'||^2/2\sigma^2\right) \\ &= \exp\left(-||\mathbf{x}||^2/2\sigma^2\right)\cdot\exp\left(\mathbf{x}^\top\mathbf{x}'/\sigma^2\right)\cdot\exp\left(-||\mathbf{x}'||^2/2\sigma^2\right) \\ &\overset{(*)}{=} f(\mathbf{x})\exp\left(\mathbf{x}^\top\mathbf{x}'/\sigma^2\right)f(\mathbf{x}') \end{aligned}$$

where we define $f(\mathbf{x}) = \exp\left(-||\mathbf{x}||^2/2\sigma^2\right)$. Now notice that this is a valid kernel as: (1) $\mathbf{x}^\top\mathbf{x}'$ is a valid kernel. (2) scaling by $\frac{1}{\sigma^2}$ is a valid kernel. (3) exponent of a valid kernel is a valid kernel. Finally, (4) multiplying by $f(\mathbf{x}), f(\mathbf{x}')$ from left and right is a valid kernel. ■

9.4 Summary and Exercises

Kernelization is a powerful learning concept through which, rather than defining more expressive (i.e. complex) hypothesis classes, we find a new embedding of the data. Through the use of the Representer Theorem (9.1.1) we have seen that given an algorithm formulated as (9.3):

$$\mathbf{w}^* := \underset{\mathbf{w}\in\mathcal{F}}{\operatorname{argmin}}\ f\left(\langle\mathbf{w},\psi(\mathbf{x}_1)\rangle,\ldots,\langle\mathbf{w},\psi(\mathbf{x}_m)\rangle\right) + R(||\mathbf{w}||)$$

there exists a *dual* optimization problem (9.6) equivalent to the original:

$$\alpha^* := \underset{\alpha\in\mathbb{R}^m}{\operatorname{argmin}}\ f(G\alpha) + R\left(\alpha^\top G\alpha\right)$$

and that this problem is a convex quadratic optimization problem which can be solved efficiently. The equivalence between the problems is in the sense that the optimal solution for the primal problem can be obtained from the optimal solution of the dual problem: $\mathbf{w}^* = \sum\alpha_j^*\psi(\mathbf{x}_j)$. We further noticed that using the dual representation, in both training and predicting, we never access the samples (or their embeddings) directly, but only by the inner products of the embeddings:

$$\begin{aligned} &\text{Training:} && G_{ij} &&= \left\langle\psi(\mathbf{x}_i),\psi(\mathbf{x}_j)\right\rangle \quad \forall i,j\in[m] \\ &\text{Predicting:} && \hat{y}(\mathbf{x}) &&= \alpha^{*\top}\boldsymbol{k} = \textstyle\sum_j \alpha_j^*\left\langle\psi(\mathbf{x}_j),\psi(\mathbf{x})\right\rangle \end{aligned}$$

Thus, we learned from Mercer's Condition (9.3.1) that we do not have to explicitly specify ψ. Instead we can simply specify a PSD-kernel function k for which there exists a mapping ψ to some Hilbert space. We know that such kernel satisfies that

$$k(\mathbf{x},\mathbf{x}') \equiv \left\langle \psi(\mathbf{x}), \psi(\mathbf{x}') \right\rangle \quad \forall \mathbf{x},\mathbf{x}' \in \mathcal{X}$$

By doing so, and substituting the inner-products of the mappings with k we are able to compute the kernel's output without explicitly evaluating the function in the high dimensional feature space $\mathcal{F}$. This means that we are able to efficiently solve an optimization problem for which the dimension of the primal representation is of arbitrary size.

Exercises

1. Prove Mercer's condition for $\mathcal{X} \coloneqq \mathbb{R}^d, \mathcal{F} \coloneqq \mathbb{R}^k, d,k \in \mathbb{N}$. That is, let $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ and let $\psi : \mathbb{R}^d \to \mathbb{R}^k$. k is a PSD-kernel function if and only if $k(\mathbf{x},\mathbf{x}') = \langle \psi(\mathbf{x}), \psi(\mathbf{x}') \rangle$.
2. Let $k(\mathbf{x},\mathbf{x}') = \langle \mathbf{x},\mathbf{x}' \rangle^k$. Show that k is a valid PSD-kernel function mapping to the feature space of all monomials of degree at most k. In addition what is the dimension of such feature space?

10. Descent Methods

In general, optimization problems are hard to solve computationally. We therefore take special interest in *convex* optimization problems since they have a unique solution, and that solution can be found in *computationally tractable* ways. Throughout the book we have seen multiple optimization problems that took the form of - or that were cast to the form of - a *convex optimization problem*. That is, an optimization problem of following the form

$$\mathbf{w}^* := \underset{\mathbf{w}\in D}{\operatorname{argmin}}\ f_{\mathbf{X},\mathbf{y}}(\mathbf{w}) \quad \text{subject to} \quad f_i(\mathbf{w}) \leq b_i,\ i = 1,\ldots,n$$

where f and $f_1,\ldots,f_n$ are all convex functions, and $D = \operatorname{dom} f \cap (\bigcap_i \operatorname{dom} f_i)$ the domain over which the problem is defined. f is called the *objective function* and $f_1,\ldots,f_n$ are called the *inequality constraint* functions. Any point $\mathbf{w} \in D$ that satisfies all constraints $f_i(\mathbf{w}) \leq b_i$ is called a *feasible point* or *feasible solution*. The optimum of f over all feasible solutions is called the *optimal value* and is denoted by f^*. $\mathbf{w}$ achieving the optimal value. i.e. $f(\mathbf{w}) = f^*$ is called the *optimal point/solution* or *minimizer*. We often denote by $C \subseteq \mathbb{R}^d$ the set of feasible solutions and rewrite the optimization problem simply as minimize $f(\mathbf{w})$ subject to $\mathbf{w} \in C$.

For example, the in RSS minimization problem discussed in chapter 2 is a convex optimization problem (with no constraints) and specifically a quadratic optimization problem.

$$\underset{\mathbf{w}\in\mathbb{R}^d}{\operatorname{argmin}}\ f(\mathbf{w}), \quad f(\mathbf{w}) := \tfrac{1}{m}\,||\mathbf{X}\mathbf{w}-\mathbf{y}||^2 = \tfrac{1}{m}\sum_i (y_i - \langle \mathbf{w}, \mathbf{x}_i\rangle)^2$$

As previously stated, instead of minimizing the squared loss, we could have minimized the Absolute Value Loss, which would still yield a convex optimization problem Ex.1

$$\underset{\mathbf{w}\in\mathbb{R}^d}{\operatorname{argmin}}\ f(\mathbf{w}), \quad f(\mathbf{w}) := \tfrac{1}{m}\,||\mathbf{X}\mathbf{w}-\mathbf{y}||_1 = \tfrac{1}{m}\sum_i |y_i - \langle \mathbf{w}, \mathbf{x}_i\rangle| \tag{10.1}$$

In both cases, $f(\mathbf{w})$ is a convex function of $\mathbf{w}$. Note though that the Squared Loss function is differentiable with respect to $\mathbf{w}$ while the Absolute Value Loss function it is not. Similarly, Ridge regression and Lasso regression are also convex optimization problems. In the case of finding a separating hyperplane that maximizes

the margin, both Hard-SVM and Soft-SVM are convex optimization problems, and specifically *Quadratic Programming* problems.

$$\begin{array}{cc} \text{Hard-SVM} & \text{Soft-SVM} \\ \underset{\mathbf{w}}{\operatorname{argmin}} \ ||\mathbf{w}||^2 & \underset{\mathbf{w}}{\operatorname{argmin}} \ \lambda \, ||\mathbf{w}||^2 + L_S^{hinge}(\mathbf{w}) \\ \text{s.t. } y_i \langle \mathbf{w}, \mathbf{x}_i \rangle \geq 1 \ \forall i & \end{array}$$

R In general, the constraint functions do not have to be neither linear nor even convex - as long as the intersection of the sets of points which satisfy each constraint, with each other and with $\dim f$ is convex, the problem is still a convex optimization problem.

In the following chapter we discuss how to learn a convex optimization problem beginning from the theoretical aspects and followed by introducing one of the most fundamental algorithms to solve such problems.

10.1 Learnability of Convex Learning Problems

Since many of the learning problems that we have discussed throughout the book were reduced to a convex optimization problem, does it mean that these problems are in some sense “not too hard”? That is, are we able to connect between the PAC framework for learnability and convex learning problems.

The hypothesis classes we used for Linear regression, Logistic Regression, Linear Regression with regularization such as Lasso and Ridge, as well as Half-Spaces, each were equivalent to a set of vectors in $\mathbb{R}^d$. For example, in the case of the halfspaces hypothesis class $HS_d \coloneqq \left\{ \mathbf{x} \mapsto \text{sign}(\langle \mathbf{w}, \mathbf{x} \rangle) : \mathbf{w} \in \mathbb{R}^d \right\}$ we identified each hypothesis with a vector $\mathbf{w} \in \mathbb{R}^d$. In some cases, such as the hypothesis class of tree partitions cannot be identified with a set of vectors in $\mathbb{R}^d$.

In order to utilize algorithms of convex optimization to learning problems, we shall first assume that there exists a *parameterization* of the hypothesis class, $\mathcal{H}$, so that it can indeed be mapped into a subset of $\mathbb{R}^d$ and only then check if the problem at hand is actually convex. We shall often identify our hypothesis class with that subset, i.e., talk about "properties of $\mathcal{H}$" while actually referring to properties of the set of vectors in $\mathbb{R}^d$ that parameterize $\mathcal{H}$.

Once we identified the hypothesis class with $\mathbb{R}^d$ (i.e map each $h \in \mathcal{H}$ with some $\mathbf{w} \in \mathbb{R}^d$) we can begin connecting learning problems and convex optimization.

Definition 10.1.1 Let $\mathcal{Z} = \mathcal{X} \times \mathcal{Y}$. A learning problem, $(\mathcal{H}, \mathcal{Z}, \ell)$, is called a *Convex Learning Problem* if and only if:

- The hypothesis class $\mathcal{H}$ is a convex set.
- For all $z \in \mathcal{Z}$ the loss function $\ell(\cdot, z)$ is a convex function of z, where for any z, $\ell(\cdot, z)$ denotes the function $f : \mathcal{H} \to \mathbb{R}$ defined by $f(\mathbf{w}) = \ell(\mathbf{w}, z)$.

A key observation regarding this definition is that in a convex learning problem, using the ERM principle for choosing an hypothesis is reduced to solving a convex optimization problem. Consider the $ERM_{\mathcal{H}}$ problem, find $\min_{\mathbf{w} \in \mathcal{H}} \frac{1}{m} \sum_i \ell(\mathbf{w}, z_i)$. If we have a convex learning problem, that is, if $\mathcal{H}$ is a convex set and the loss function, $\ell(\mathbf{w}, z)$, is a convex function of $\mathbf{w}$, then the empirical risk, being the sum of convex functions, is also convex. As an example, in the case of the Least Squares problem $\mathcal{H} = \mathbb{R}^d$, $\mathcal{Z} = \mathbb{R}^d \times \mathbb{R}$ and $\ell(\mathbf{w}, (\mathbf{x}, y)) = \left(\mathbf{x}^\top \mathbf{w} - y \right)^2$ where indeed $\mathcal{H}$ is identified by a convex set of vectors and ℓ is convex in $\mathbf{w}$. Thus, this is a convex learning problem.

Recall that according to the fundamental theorem of learning, ERM allows us to learn with approximately the minimal-possible number of samples. So is it a *sufficient condition* for a learning problem to be convex for it to be PAC-learnable? Unfortunately not, but by addition the conditions of $\mathcal{H}$ being bounded (in the sense of a bounded convex set) and that the loss function is *Lipschitz* we obtain PAC-learnability.

10.1.1 Convex-Lipchitz-Bounded Learning Problems

In the context of optimization, we are dealing in most cases with functions which are quite regular in the sense of being continuous, smooth at most points and limited in their rate of variation. However, as these functions are not necessarily differentiable (e.g hinge-loss in Soft-SVM or Absolute Value loss regression) we still need to define a sense in which a function is "well-behaved". We define this by Lipschitz continuity.

Definition 10.1.2 A function $f : C \to \mathbb{R}$ is called *Lipschitz continuous* with a *Lipschitz constant* ρ, or simply, ρ-*Lipschitz*, if for every $\mathbf{w}_1, \mathbf{w}_2 \in C$ it holds that $|f(\mathbf{w}_1) - f(\mathbf{w}_2)| \leq \rho \, ||\mathbf{w}_1 - \mathbf{w}_2||$.

That is, Lipschitz continuity bounds the rate in which a function can change. Let $f : \mathbb{R} \to \mathbb{R}$ be a ρ-Lipschitz function. By definition, this means that $|f(w_1) - f(w_2)| \leq \rho \, |w_1 - w_2|$ for every $w_1, w_2 \in \mathbb{R}$. Fix $w_1 = 0$ then for any $w_2 \in \mathbb{R}$ it holds that $|f(w_2)| \leq \rho \, |w_2|$, namely f is bound between the two lines passing through $f(w_1)$ and with a slope of $\pm\rho$. Generalizing for $\mathbb{R}^d$, f is bound within a cone in $\mathbb{R}^{d+1}$.

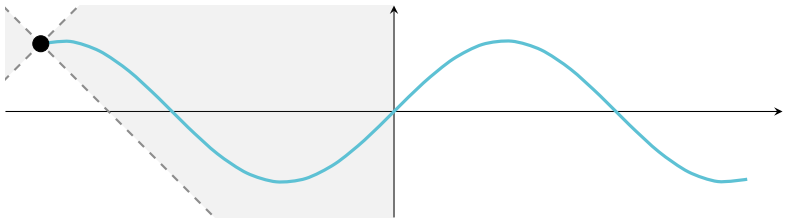


Figure 10.1: *Rate of change of Lipschitz function is bounded by a cone at every point*

Of note, ρ-Lipschitz continuity implies continuity but does not imply differentiability (e.g., $|w|$). Moreover, a function may be continuous, differentiable and convex but still not be ρ-Lipschitz because the value of its derivatives isn't bounded (e.g., w^2). Thus, ρ-Lipschitzness can be violate because a function has a local jump (e.g., discontinuity at the origin) or because it becomes steeper and steeper smoothly so that the norm of its gradient has no upper bound.

Given a continuous function $f : \mathbb{R}^d \to \mathbb{R}$, then f Lipschitz if and only if there exists $M \in \mathbb{R}_+$ such that $||\nabla f(\mathbf{x})|| \leq M$, $\forall \mathbf{x} \in \text{dom} f$. Namely, the graident of f is bound. Furthermore, a Lipschitz function is differentiable almost everywhere, that is, except in a set of measure zero. Using the notion of a Lipschitz function, we now focus on a subset of convex learning problems - the convex-Lipschitz-bounded learning problems.

Definition 10.1.3 A learning problem, $(\mathcal{H}, Z, \ell)$, is called *Convex-Lipschitz-Bounded*, with parameters ρ, B if:

- $\mathcal{H}$ is a convex and bounded, i.e $\exists B < \infty, \ \max_{\mathbf{w} \in \mathcal{H}} \|\mathbf{w}\| < B$.
- For all $z \in Z$, the loss function, $\ell(\mathbf{w}, z)$, is convex and ρ-Lipschitz in $\mathbf{w}$.

Lemma 10.1.1 Let $(\mathcal{H}, Z, \ell)$ be a Convex-Lipschitz-Bounded learning problem, then $(\mathcal{H}, Z, \ell)$ is PAC learnable with sample complexity that depends only on $\varepsilon, \delta, B, \rho$.

10.2 Gradient Descent

With the knowledge that Convex-Lipschitz-Bounded learning problem are PAC learnable, let us discuss algorithms that can be used to solve such problems. Quite often, a minimum of a function $f : \mathbb{R}^d \to \mathbb{R}$ can not be found analytically and one has to resort to numerical algorithms, many of which, especially in the context of machine learning, take the following general form

Given a starting point $\mathbf{x} \in \mathrm{dom} f$
Repeat
  Determine a step direction $\Delta\mathbf{x}$
  Choose a step size $\eta > 0$
  Update $\mathbf{x} := \mathbf{x} + \eta\Delta\mathbf{x}$
Until stopping criterion is satisfied (10.2)

The different algorithms vary in the manner in which the initialization, step direction, step size and stopping criterion are defined. This general convex optimization "solver" is used to find the minimizer of a convex function. Using this general algorithm we can implement various learning algorithms of previously discussed problems such as Hard- and Soft-SVM, logistic regression and Lasso regression. In addition, if faced with a learning problem with no "off the shelf" solution, the GD algorithm can be tailored for a new and "strange" objective function or for dealing with large scales of samples or features.

R There are specialized solvers for specific types, or families, of convex optimization problems, e.g. Lasso regression can be solved efficiently for every value of λ at once. A specialized solver is typically preferred, as it leverages some particular structure of the problem to solve it more efficiently, using less space, etc.

Before defining the actual algorithm let us consider a very naive approach over $\mathbb{R}$. Let $f : \mathbb{R} \to \mathbb{R}$ be some continuous function (at this point not necessarily convex or differentiable) for whom we wish to find its minimizer $x^* \in \mathbb{R}$. In this approach, depicted in Figure 10.2, at each step we try to step in a direction that will lead to a decrease in the value of the function. We randomly start at some point $x^{(t)} \in \mathrm{dom} f$ for $t = 1$. Then, we evaluate f at $f(x^{(t)})$ and at $f(x^{(t)} \pm \eta)$ for $\eta > 0$. η is referred to as the *step size* or the *learning rate*. If f increases in both directions, stop and output $x^{(t)}$. If $f(x^{(t)} + \eta) > f(x^{(t)} - \eta)$ set a variable $\Delta x^{(t)} = -1$. Otherwise set $\Delta x^{(t)} = 1$. Update $x^{(t+1)} = x^{(t)} + \eta \cdot \Delta x^{(t)}$ and repeat. Halt once the function no longer decreases, i.e. when $f(x^{(t+1)}) \geq f(x^{(t)})$, and output $x^{(t)}$. This naive approach suffers from several limitations.

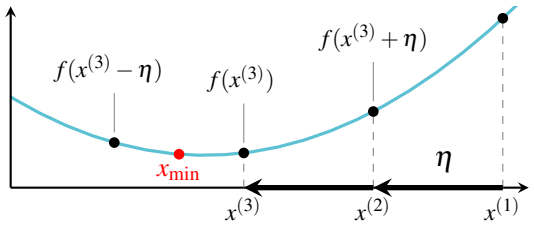


Figure 10.2: *Illustration of naive minimum-search approach over* $\mathbb{R}$

Incorrect halting point This algorithm may halt at a plateau or at a *local minimum*, missing a deeper global one. Moreover, if the hard halting condition $f(x^{(t+1)}) \geq f(x^{(t)})$ is replaced by a softer one such as $f(x^{(t+1)}) \geq f(x^{(t)}) + \varepsilon$, the algorithm may halt at a saddle point (consider $f(x) = x^3$).

To address this limitation let us assume from now on that f is a convex function defined over a convex domain, thus resolving this limitation - there are no saddle-points, any local minimum, $x_{\min}$, will then be a global minimum and any point on the segment between $x^{(1)}$ and $x_{\min}$ will belong to $\mathrm{dom} f$.

Fixed step size The algorithm proceeded each time by a *fixed* step size η, and therefore its ability to find the minimum of f is sensitive to our choice of η. If η is too small the algorithm will take a very long time to converge. If η is too large the algorithm might leap far over the minimum (referred to as *over-shooting*).

To (partially) resolve this matter, we need to allow the step size to automatically vary, so that initial steps will be large and will shrink the closer we get to $x_{\min}$. We could for example define η to be a function of t such that $\eta_t := \eta(t) = -a \cdot t + b$ for some $a, b > 0$. This is referred to as a *linear decay*. We could also define $\eta_t = a \cdot \exp(-b \cdot t + c)$ for some $a, b, c > 0$ to achieve an *exponential decay*. Furthermore, we might want the step size to be adaptive in the sense of changing it with respect to the change in the function. One approach for automatically varying η is discussed in subsection 10.2.2.

Infinite possible descent directions A more serious limitation is determining the *descent direction* - a direction in which the value of the function decreases. In the case of $d = 1$, at any point we only had to choose if to go "left" or "right", i.e. to evaluate the function at $x^{(t)} \pm \eta$, and through a series of steps (approximately) reach the function's minimum. For $d > 1$, we have an infinite number of directions to evaluate (consider for example the Euclidean unit ball) and potentially an infinite number of them are descent directions. We therefore must find a computationally efficient way to decide which descent direction to choose.

To resolve these issues consider the following simplifying proposition. The function we wish to minimize f might be complicated, and therefore difficult to find its minimum. Let us make the assumption that the function is in fact linear or quadratic and thus simple to minimize. We will *replace* f with the *approximation* of f at $\mathbf{x}^{(t)}$, and *minimize the approximation* instead. The hope is then that stepping right into the minimum of the *approximation* of f, which is easy to compute, will land us not too far from the minimum of f itself.

Let $f : \mathbb{R}^d \to \mathbb{R}$ be a continuous, convex and differentiable function with a local (and being convex, also global) minimum at, $\mathbf{x}_{\min}$. Being at $\mathbf{x}^{(t)}$ we wish to find a descent direction $\mathbf{x} \in \mathrm{dom} f$. From the Taylor expansion of f around $\mathbf{x}^{(t)}$, for a small enough $\|\mathbf{x}^{(t)} - \mathbf{x}\|$, it holds that

$$f(\mathbf{x}) \approx f(\mathbf{x}^{(t)}) + \left\langle \nabla f(\mathbf{x}^{(t)}), \mathbf{x} - \mathbf{x}^{(t)} \right\rangle$$

That is, f can be replaced by a linear function given by its first order Taylor's expansion at $\mathbf{x}^{(t)}$ (Figure 10.3). Then, for a step size of $\eta \equiv \|\mathbf{x}^{(t)} - \mathbf{x}\|$ the linear approximation achieves a minimum at $\eta \cdot \nabla f(\mathbf{x}^{(t)})$, yielding the update rule $\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \eta \nabla f(\mathbf{x}^{(t)})$. This type of update rule is referred to as an *additive* update rule.

By following this update rule we are indeed manage to step in a descent direction of f:

$$f(\mathbf{x}^{(t+1)}) = f\left(\mathbf{x}^{(t)} - \eta \nabla f(\mathbf{x}^{(t)})\right) \approx f(\mathbf{x}^{(t)}) - \eta \left\langle \nabla f(\mathbf{x}^{(t)}), \nabla f(\mathbf{x}^{(t)}) \right\rangle \leq f(\mathbf{x}^{(t)}) \tag{10.3}$$

Furthermore, for f a differentiable function at $\mathbf{x} \in \mathrm{dom} f$ it holds that $\frac{\nabla f(\mathbf{x})}{\|\nabla f(\mathbf{x})\|}$ is the vector that points to the direction of steepest ascent. Likewise, $-\frac{\nabla f(\mathbf{x})}{\|\nabla f(\mathbf{x})\|}$ is the vector that point to the direction of steepest descent. Ex.2

Next, using the above update rule, when should the algorithm halt? That is, how will the algorithm know

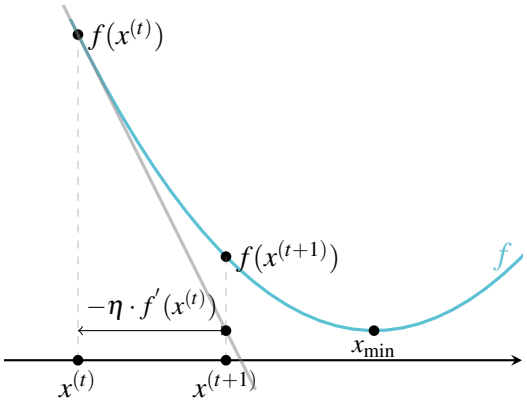


Figure 10.3: *Minimizing f by minimizing a first-order approximation of f at $x^{(t)}$*

that it has reached (approximately) the minimum. For this, let us derive the *first-order condition for optimality*.

Claim 10.2.1 Given a convex problem of minimizing $f(\mathbf{x})$ subject to $\mathbf{x} \in C$, where f is differentiable, then a feasible solution $\mathbf{x}^*$ is optimal if and only if

$$\langle \nabla f(\mathbf{x}^*), \mathbf{x} - \mathbf{x}^* \rangle \geq 0 \quad \forall \mathbf{x} \in C$$

Namely, $\mathbf{x}^*$ is a minimizer if for any feasible vector pointing from $\mathbf{x}^*$ in the direction $\mathbf{h}$, $\mathbf{x}^* + \mathbf{h}$, we ascend in f. That is, there is no component in the descent direction $-\nabla f(\mathbf{x}^*)$. Observe that if $\mathbf{x}^*$ is *not optimal* then there exists a feasible $\mathbf{x}^* + \mathbf{h}$ for which $\langle \nabla f(\mathbf{x}^*), \mathbf{h} \rangle$. $\mathbf{h}$ is then a *descent direction* from $\mathbf{x}^*$.

Notice that in the special case of $C = \mathbb{R}^d$ (no constraints) the optimality condition implies that $\mathbf{x}^*$ is optimal if and only if $\nabla f(\mathbf{x}^*)$ is orthogonal to any $\mathbf{x} \in \mathbb{R}^d$, namely if and only if $\nabla f(\mathbf{x}^*) = 0$. Thus, we half when $\|\nabla f(\mathbf{x})\| = 0$ meaning that $\mathbf{x}$ is a local minimum - and in a convex function a global minimum.

■ **Example 10.1** Consider the following unconstrained quadratic program where

$$\mathbf{x}^* = \underset{\mathbf{x} \in \mathbb{R}^d}{\operatorname{argmin}} f(\mathbf{x}), \quad f(\mathbf{x}) := \tfrac{1}{2}\mathbf{x}^\top Q\mathbf{x} + \mathbf{b}^\top \mathbf{x} + c$$

for a symmetric $Q \in \mathbb{R}^{d \times d}, \mathbf{b} \in \mathbb{R}^d$ and $c \in \mathbb{R}$, for which we wish to find the minimizers. Since f is a differentiable function, with $\nabla f(\mathbf{x}) = Q\mathbf{x} + \mathbf{b}$, we can use the first-order optimality condition (10.2.1) to find the minimizer. If $Q \succ$ (i.e. strictly PD) then, as Q is non-singular Q^{-1} exists and by equating the derivative to zero we obtain that $\mathbf{x}^* = -Q^{-1}\mathbf{b}$. If Q is singular and $\mathbf{b} \notin \mathrm{Im}(Q)$ then as f is unbounded from below, there are no optimal solutions. Lastly, if Q is singular but $\mathbf{b} \in \mathrm{Im}(Q)$ then any point of the form $\mathbf{x}^* = Q^\dagger \mathbf{b} + \mathbf{z}$, with $Q^\dagger$ is the Moore-Penrose pseudoinverse and $\mathbf{z} \in \mathrm{Ker}(Q)$, is optimal.

Notice that the above problem resembles the Least-Squares linear regression problem (2.5) for which $Q = \mathbf{X}^\top \mathbf{X}$, $\mathbf{b} = -2\mathbf{y}^\top \mathbf{X}$ and $c = ||\mathbf{y}||^2$:

$$\mathrm{RSS}(\mathbf{w}) = ||\mathbf{X}\mathbf{w} - \mathbf{y}||^2 = \mathbf{w}^\top \mathbf{X}^\top \mathbf{X}\mathbf{w} - 2\mathbf{y}^\top \mathbf{X}\mathbf{w} + ||\mathbf{y}||^2$$

Furthermore, it can be shown that the Least-Squares solution $\hat{\mathbf{w}}^{LS}$ is a specific case of the first-order optimality condition for a linear program. ■

Thus, we obtain the Gradient Descent algorithm. Starting from some initial point $\mathbf{x}^{(1)} \in \mathbb{R}^d$ at each iteration t

we update according the the minimization of the first-order approximation

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \eta_{t+1} \nabla f\left(\mathbf{x}^{(t)}\right) \tag{10.4}$$

and stop once $\|\nabla f(\mathbf{x}^{(t)})\|$ is close enough to zero. Namely, we step in the steepest direction $-\nabla f(\mathbf{x})$ with a *gradient step size of* η_{t+1} until we are close enough to $\|\nabla f(\mathbf{x}^{(t)})\| = 0$ that is a minimum point. Once the algorithm halts we can consider several output vectors. For T the number of iterations performed we can output different vectors: the last vector $\mathbf{x}^{(T)}$, the best performing vector $\mathbf{x}^{(t^*)}$ where $t^* := \text{argmin}_{t\in[T]} f(\mathbf{x}^{(t)})$, the average of the stepwise vectors $\bar{\mathbf{x}} = \frac{1}{T}\sum \mathbf{x}^{(t)}$, etc. By returning the average vector $\bar{\mathbf{x}}$ and not just any of $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(T)}$ we reduce the effect of individual fluctuations of the path around the minimum.

Algorithm 15 Gradient Descent

procedure GRADIENT DESCENT(f, $\{\eta_t\}$)
  Initialize $\mathbf{x}^{(1)} \in \mathbb{R}^d$
  for $t = 1, \ldots, T$ **do**
    $\mathbf{x}^{(t+1)} \leftarrow \mathbf{x}^{(t)} - \eta_{t+1} \nabla f\left(\mathbf{x}^{(t)}\right)$
  end for
  return $\bar{\mathbf{x}} = \frac{1}{T}\sum \mathbf{x}^{(t)}$
end procedure

Figure 10.4 illustrates the GD algorithm for the Least-Squares optimization problem. In each iteration of the algorithm we see the traverse in the parameter space (left) and the regression line the current solution yields (right). In this example the step size η remains constant.

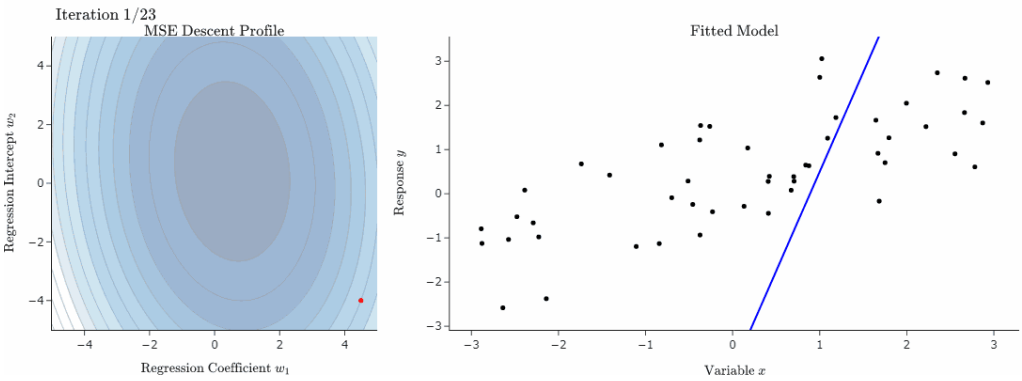


Figure 10.4: ***Solving Least-Squares using GD*** *Descent Methods Examples*

10.2.1 Convergence Analysis

Given the gradient descent algorithm (Algorithm 15) are we able to prove that it does indeed converge to an optimal solution. The simplest *convergence analysis* theorem for gradient descent is the following. Let f be a convex and differentiable function with $\text{dom} f = \mathbb{R}^d$. Assume that ∇f is ρ-Lipschitz such that $||\nabla f(\mathbf{x}) - \nabla f(\mathbf{x}')||_2 \leq \rho\, ||\mathbf{x} - \mathbf{x}'||_2$ for any $\mathbf{x}, \mathbf{x}' \in \text{dom} f$. Then, the gradient descent algorithm, with a fixed step size $\eta > 0$ satisfies that

$$f\left(\mathbf{x}^{(t)}\right) - f(\mathbf{x}^*) \leq \frac{1}{2\eta t} \left|\left|\mathbf{x}^{(0)} - \mathbf{x}^*\right|\right|_2^2 \tag{10.5}$$

Namely, the gradient descent algorithm converges to the optimum as $t \to \infty$. This theorem further provides us with a sufficient number of iterations T such that for every accuracy level $\varepsilon > 0$ the solution returned by the algorithm $\mathbf{w}^{(T)}$ satisfies $f(\mathbf{x}^{(T)}) - f(\mathbf{x}^*) \geq \varepsilon$. By isolating t in the expression above

$$f\left(\mathbf{x}^{(t)}\right) - f(\mathbf{x}^*) \leq \frac{1}{2\eta t}\left|\left|\mathbf{x}^{(0)} - \mathbf{x}^*\right|\right|_2^2 \leq \varepsilon \quad \Rightarrow \quad t \geq \frac{\left|\left|\mathbf{x}^{(0)} - \mathbf{x}^*\right|\right|^2}{2\eta\varepsilon}$$

By bounding the set of feasible solutions we obtain an expression that depends on the size of the bound, the learning rate and the desired accuracy. Note that this bound is not tight and even under the assumptions above a tighter bound on T can be derived.

10.2.2 Adaptive Learning Rate - Backtracking Line Search

A question remains regarding the gradient descent algorithm which is how to correctly choose the learning rates $\{\eta_t\}$. In the original naive approach (Figure 10.2) we use a fixed learning rate, $\eta_t \equiv \eta > 0$. This approach might perform poorly as we are assuming a single step size to fit the whole range of the function. Generally, we have argued that if the learning rate is too small the algorithm will require many iterations to converge, while if the learning rate is too large the algorithm might diverge and fail to find the minimizer.

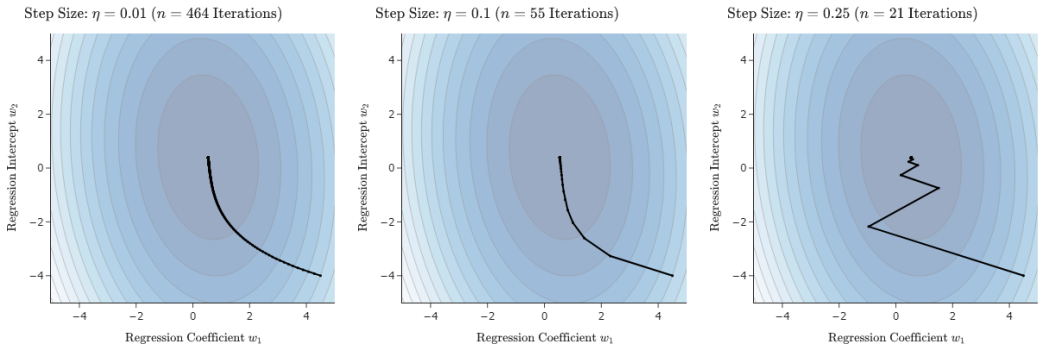


Figure 10.5: GD iterations for different learning rates *Descent Methods Examples*

We have also suggested specifying $\{\eta_t\}$ using some decay function such as a linear decay $\eta_t = a \cdot t + b$ for $a < 0$, an exponential decay $\eta_t = a \cdot \exp(-b \cdot t + c)$ for some $a, b, c > 0$, etc. These methods will provide a "cooling down" effect where initially the algorithm takes large steps in f and then, as the algorithm begins to converge so do the step sizes decrease allowing better local fits and convergence to the minimum. A challenge with this approach is how to choose the "correct" decay strategy: what function to use to model the decay? with what parameters? should we consider a mixture of decay functions? etc.

An approach that yields better results is to choose the learning rate in an adaptive manner. That is, adjusting the value of η_{t+1} to the manner in which the function behaves at the current location. We do so by using some form of a *line search* algorithm. Namely, evaluating the function $f(\mathbf{x}^{(t)} - \eta \nabla f(\mathbf{x}))$ for different values of η and choosing the one that minimizes f the most. Denote $\mathbf{x} \in \mathrm{dom} f$ the current step and $\Delta\mathbf{x}$ the next step direction (e.g $\Delta\mathbf{x} := -\nabla f(\mathbf{x})$). Since we assume f to be convex and differentiable we know that the linear approximation $f(\mathbf{x}) + \eta \langle \nabla f(\mathbf{x}), \Delta\mathbf{x} \rangle$ lies below the function graph. Now, consider the ray starting from $f(\mathbf{x})$ that is given by

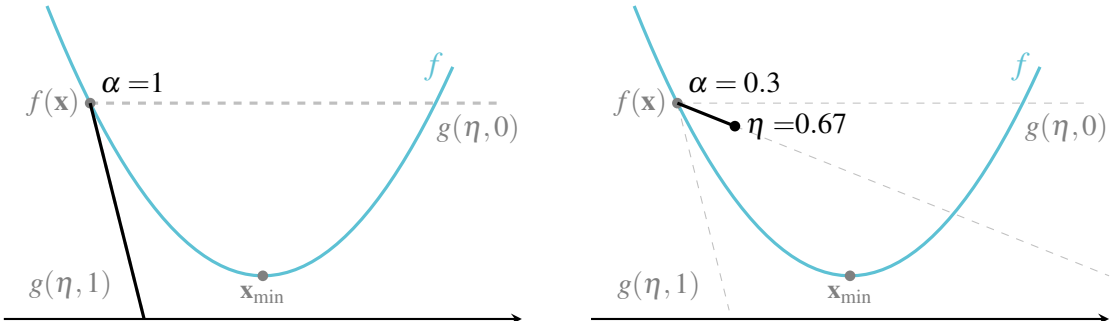
$$g(\eta, \alpha) = f(\mathbf{x}) + \alpha\eta \langle \nabla f(\mathbf{x}), \Delta\mathbf{x} \rangle, \quad \eta > 0, \alpha \in (0,1)$$

As a function of η, notice that fixing $\alpha = 1$, the ray $g(\eta, 1)$ coincides with the the linear approximation of f at $\mathbf{x}$, while when fixing $\alpha = 0$ the ray is the constant function $g(\eta, 0) = f(\mathbf{x})$. Therefore, α functions as a tuning parameter for the angle of the ray, and for which it holds that the minimum of f lies between these two extremes. Values $\alpha \notin (0,1)$ will result in stepping in ascent directions.

Next, fix $\alpha = \frac{1}{2}$ and consider g as a function of η. Notice that since f is convex:

- For small enough $\eta > 0$ then $f(\mathbf{x}) + \alpha\eta \langle \nabla f(\mathbf{x}), \Delta\mathbf{x} \rangle > f(\mathbf{x} + \eta\Delta\mathbf{x})$.
- For large enough $\eta > 0$ then $f(\mathbf{x}) + \alpha\eta \langle \nabla f(\mathbf{x}), \Delta\mathbf{x} \rangle < f(\mathbf{x} + \eta\Delta\mathbf{x})$.

As we also assume that f is differentiable (and thus continuous), from the intermediate value theorem there must be some value $\eta_0 > 0$ for which $f(\mathbf{x}) + \alpha\eta \langle \nabla f(\mathbf{x}), \Delta\mathbf{x} \rangle = f(\mathbf{x} + \eta\Delta\mathbf{x})$.



(a) *Evaluation with respect to* $\alpha \in (0,1)$ ***(b)*** *Evaluation with respect to* $\eta > 0$

Figure 10.6: *Evaluating ray* $g(\eta, \alpha)$ *with respect to each of its parameters*

The *Backtracking Line Search* algorithm approximates η_0 for which the ray intersects with the graph of the function. To do so we perform the following. We initialize $\eta = 1$. Then, in an iterative manner we set $\eta \mapsto \beta\eta$ from some fixed $\beta \in (0,1)$ and repeat while $f(\mathbf{x}) + \alpha\eta \langle \nabla f(\mathbf{x}), \Delta\mathbf{x} \rangle < f(\mathbf{x} + \eta\Delta\mathbf{x})$. We take the first value of η that violates the condition and use it as the chosen step size. Namely, we repeat as long as we are below the function's graph.

Algorithm 16 Gradient Descent With Backtracking Line Search

procedure LINE-SEARCH($\mathbf{x}$)
  Initialize $\eta \leftarrow 1$
  while $f(\mathbf{x}) + \alpha\eta \langle \nabla f(\mathbf{x}), \Delta\mathbf{x} \rangle < f(\mathbf{x} + \eta\Delta\mathbf{x})$ **do**
    $\eta \leftarrow \beta\eta$
  end while
  return η
end procedure

procedure GRADIENT DESCENT(f)
  Initialize $\mathbf{x}^{(1)} \in \mathbb{R}^d$
  for $t = 1, \ldots, T$ **do**
    $\eta \leftarrow$ LINE-SEARCH($\mathbf{x}^{(t)}$)
    $\mathbf{x}^{(t+1)} \leftarrow \mathbf{x}^{(t)} - \eta \nabla f(\mathbf{x}^{(t)})$
  end for
  return $\overline{\mathbf{x}} = \frac{1}{T} \sum \mathbf{x}^{(t)}$
end procedure

This is a very crude procedure. Suppose $\beta = 0.1$ then we begin with evaluating $\eta = 1$, then $\eta = 0.1$, then $\eta = 0.01$, etc. Through very crude, this procedure works very well even compared to much more sophisticated approaches, and with high robustness to the selection of the hyper-parameters α and β.

10.2.3 Projected Gradient Descent

In the scenarios above, of minimizing $f(\mathbf{x})$ subject to $\mathbf{x} \in C$, we assumed the convex optimization problems were unconstrained (i.e $C = \mathbb{R}^d$). As such, for any $\mathbf{x}^{(t+1)}$ obtained in the gradient descent step then $\mathbf{x}^{(t+1)}$ is a feasible solution: $\mathbf{x}^{(t+1)} \in C$. In the case of a constrained convex optimization problem (i.e $C \neq \mathbb{R}^d$) we must verify that the proposed solutions $\mathbf{x}^{(t+1)}$ are feasible solutions. One way of doing so is the *Projected Gradient Descent*.

Denote P_C the *projection operator* on the feasible set C defined by $P_C(\mathbf{x}) \coloneqq \text{argmax}_{\mathbf{x}' \in C} ||\mathbf{x} - \mathbf{x}'||_2$. Notice that this itself is a convex optimization problem. The projected gradient descent iteration is then defined as

$$\mathbf{x}^{(t+1)} = P_C\left(\mathbf{x}^{(t)} - \eta_{t+1} \nabla f\left(\mathbf{x}^{(t)}\right)\right) \tag{10.6}$$

Namely, after each gradient step we project the result $\mathbf{x}^{(t)} - \eta_{t+1}\nabla f(\mathbf{x}^{(t)})$ back onto the feasible set C. Since the projection, which involves solving a convex problem, in each iteration of the gradient descent it is important that we are able to calculate $P_C(\mathbf{x})$ efficiently.

10.2.4 Second-Order Approximations

Being negative or positive, the slope, $\nabla f(\mathbf{x}^{(t)})$, of that linear approximation, indicates in which *direction* we should step to get closer to the minimum. However, if C is not bounded the minimum of a linear function with a non-zero slope is at $\pm\infty$ (in Figure 10.3 we used η as a way to bound the domain). Therefore $f_{\text{approx}}(\mathbf{x})$ does not tell us much about the *distance* from $\mathbf{x}_{\text{min}}$, since this distance depends on how fast f "bends away" from its linear approximation, i.e. it depends on higher derivatives of f.

To estimate the distance from $\mathbf{x}^{(t)}$ to $\mathbf{x}_{\text{min}}$, we can approximate f with its quadratic (second order) Taylor's expansion:

$$f\left(\mathbf{x}^{(t)} + \mathbf{x}\right) \approx f\left(\mathbf{x}^{(t)}\right) + \nabla f\left(\mathbf{x}^{(t)}\right)^{\top} \mathbf{x} + \tfrac{1}{2}\mathbf{x}^{\top} H\left(\mathbf{x}^{(t)}\right)\mathbf{x} \tag{10.7}$$

for $H(\mathbf{x}^{(t)})$ the Hessian of f at $\mathbf{x}^{(t)}$. If before we approximated and minimized the linear (first-order) approximation of f, here we approximate f by a paraboloid.

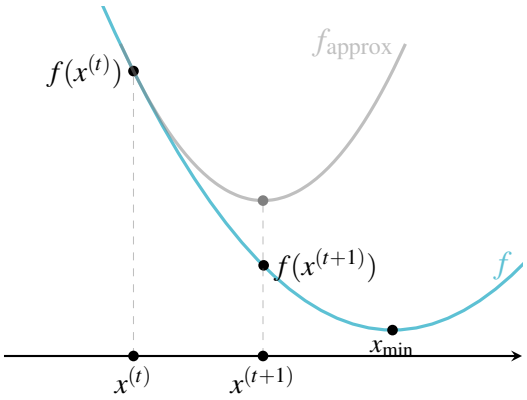


Figure 10.7: *Minimizing f by minimizing a second-order approximation of f at $x^{(t)}$*

We now wish to find $\mathbf{x}$ that minimizes (10.7). By taking the first derivative with respect to $\mathbf{x}$ and equating it with zero we find that:

$$\begin{aligned} \frac{\partial}{\partial \mathbf{x}}\left(f\left(\mathbf{x}^{(t)}\right) + \nabla f\left(\mathbf{x}^{(t)}\right)^{\top}\mathbf{x} + \tfrac{1}{2}\mathbf{x}^{\top} H\left(\mathbf{x}^{(t)}\right)\mathbf{x}\right) &= \nabla f(\mathbf{x}^{(t)}) + H\left(\mathbf{x}^{(t)}\right)\mathbf{x} = 0 \\ &\Downarrow \\ \mathbf{x} &= -\left[H\left(\mathbf{x}^{(t)}\right)\right]^{-1} \cdot \nabla f(\mathbf{x}^{(t)}) \end{aligned}$$

And derive the following update rule:

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \left[H\left(\mathbf{x}^{(t)}\right)\right]^{-1} \cdot \nabla f(\mathbf{x}^{(t)}) \tag{10.8}$$

Minimizing f using this update rule is known as *Newton's Method*. As in Gradient Descent, at each step we move in the steepest descent direction $-\nabla f(\mathbf{x}^{(t)})$ (where the Hessian is non-negative since f is convex). Unlike Gradient Descent, here we do not control the step size through the use of η but instead by $\left[H\left(\mathbf{x}^{(t)}\right)\right]^{-1}$. For sufficiently small $\|\mathbf{x}^{(t)}-\mathbf{x}\|$ the Taylor expansion is sufficiently accurate and this choice of $\mathbf{x}^{(t+1)}$ yields a very fast convergence rate. However, if f around $\mathbf{x}^{(t)}$ is very flat the second derivatives are close to zero, resulting in their inverse being very large and to too large step sizes. This is known as *divergence*. One approach to avoid divergence of Newton's method is by beginning with Gradient Descent and finishing using Newton's Method.

Algorithm 17 Newton's-Method

procedure NEWTON'S-METHOD(f) ▷ for f twice differentiable
  Initialize $\mathbf{x}^{(1)} \in \mathbb{R}^d$
  for $t = 1,\ldots,T$ **do**
    $\mathbf{x}^{(t+1)} \leftarrow \mathbf{x}^{(t)} - \left[H\left(\mathbf{x}^{(t)}\right)\right]^{-1} \cdot \nabla f(\mathbf{x}^{(t)})$
  end for
  return $\bar{\mathbf{x}} = \frac{1}{T}\sum \mathbf{x}^{(t)}$
end procedure

10.3 Sub-Gradients

When deriving the gradient descent algorithm we required from the objective f to be a convex and differentiable function. Then, for any $\mathbf{x} \in \mathrm{dom} f$ reached in the algorithm's run we can calculate $\nabla f(\mathbf{x})$ and deduce the next step. What should we do however in the case of a convex function which is not differentiable across all its domain? Suppose we wish to minimize the ℓ_1 norm such as in the case of the Absolute Value Loss optimization problem (10.1). This function is not differentiable at $\mathbf{x} = \mathbf{0}$ and therefore $\nabla f(\mathbf{0})$ does not exist. We know that for a differentiable and convex function, the tangent at any point on the function graph lies below the graph. The gradient vector $\nabla f(\mathbf{w})$ defines the slope of the tangent at $\mathbf{x}$. When f is convex but not differentiable at $\mathbf{x}$, even though the gradient does not exists, it is intuitively clear (Figure 10.8) that there exist at least a single hyperplane that lie fully below the graph of f, except for the point where it touches it $(\mathbf{x}, f(\mathbf{x}))$. Each of these hyperplanes is defined by a vector which is called a *sub-gradient*.

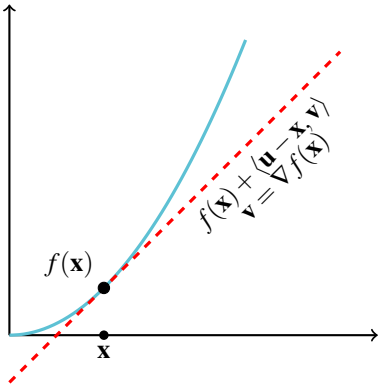
Definition 10.3.1 Let $f : \mathbb{R}^d \to \mathbb{R}$. A vector $\mathbf{v} \in \mathbb{R}^d$ is said to be a *sub-gradient* of f at $\mathbf{x} \in \mathrm{dom} f$ if

$$f(\mathbf{u}) \geq f(\mathbf{x}) + \langle \mathbf{v}, \mathbf{u}-\mathbf{x} \rangle \quad \forall \mathbf{u} \in \mathrm{dom} f$$

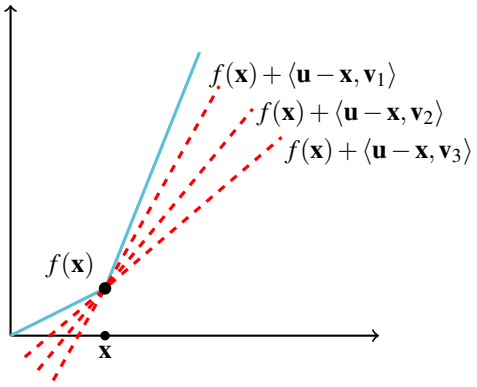
The set of all sub-gradients of f at $\mathbf{x}$ is called the *sub-differential* of $f(\mathbf{x})$ and is denoted by $\partial f(\mathbf{x})$.

■ **Example 10.2** Consider the ℓ_1 norm over $\mathbb{R}$, $f(x) := |x|$. This convex function is differentiable in $\mathrm{dom} f \backslash \{0\}$. At any point $x \neq 0$ the only element in its sub-differential is $\partial f(x) = \{\mathrm{sign}(x)\}$. For $x = 0$ the set of lines which lie under the function graph are those passing through $(0, f(0))$ and whose slope is between -1 and 1. Therefore the sub-differential at $x = 0$ is $\partial f(x) = [-1, 1]$. Generalizing to the multivariate case, for $\mathbf{x} \neq \mathbf{0}$, the sub-differential at every point is $\partial f(\mathbf{x}) = \{\nabla f(\mathbf{x})\} = \{(\mathrm{sign}(x_1), \ldots, \mathrm{sign}(x_d))^\top\}$. For $\mathbf{x} = \mathbf{0}$ then $\partial f(\mathbf{x}) = [-1, 1]^d$. ■

■ **Example 10.3** Let $f : \mathbb{R}^d \to \mathbb{R}$ be the ℓ_2 norm defined as $f(\mathbf{x}) = ||\mathbf{x}||_2 = \sqrt{\sum x_i^2}$. This function is



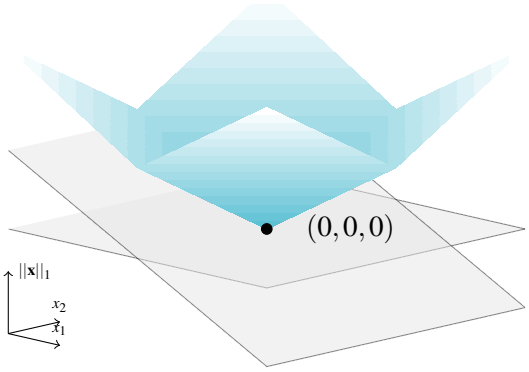
(a) *Convex and differentiable at* $\mathbf{x}$*. The sub-differential is* $\partial f(\mathbf{x}) = \{\nabla f(\mathbf{x})\}$



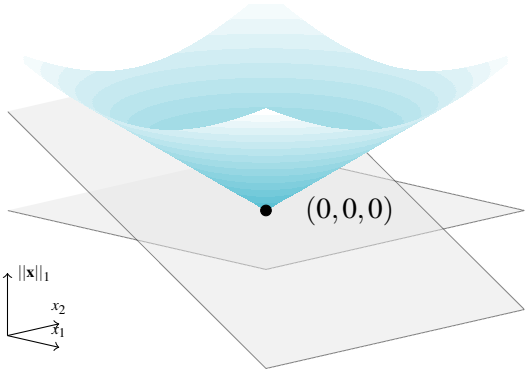
(b) *Convex but non-differentiable at* $\mathbf{x}$*. The sub-differential has an infinite number of sub-gradients*

Figure 10.8: *Gradients vs. sub-gradients*

differentiable in all its domain except at $\mathbf{x} = \mathbf{0}$. For $\mathbf{x} \neq \mathbf{0}$ the sub-differential is $\partial f(\mathbf{x}) = \{\frac{\mathbf{x}}{||\mathbf{x}||_2}\}$ and for $\mathbf{x} = \mathbf{0}$ the sub-differential is $\partial f(\mathbf{x}) = \{\mathbf{u} \,|\, ||\mathbf{u}||_2 \leq 1\}$ ∎



(a) ℓ_1 *norm over* $\mathbb{R}^2$ *with two sub-gradients of* $\ell_1(\mathbf{0})$



(b) ℓ_2 *norm over* $\mathbb{R}^2$ *with two sub-gradients of* $\ell_2(\mathbf{0})$

10.3.1 Properties of the sub-differential

The sub-differential $\partial f(\mathbf{x})$ is a closed set. It is also a convex set, regardless to f being convex or non-convex. As such, it is closed under the following transformations:

- Non-negative scalar multiplication: $\partial(\alpha f) = \alpha \cdot \partial f = \{\alpha \mathbf{u} \,|\, \mathbf{u} \in \partial f\}$ for $\alpha > 0$.
- Additivity: $\partial(f_1 + f_2) = \partial f_1 + \partial f_2 = \{\mathbf{u} + \mathbf{v} \,|\, \mathbf{u} \in \partial f_1, \mathbf{v} \in \partial f_2\}$.
- Affine composition: for $g(\mathbf{x}) = f(A\mathbf{x} + b)$ then $\partial g(\mathbf{x}) = A^\top \partial f(A\mathbf{x} + b)$.

Furtherfore, based on the examples above we can begin observing a connection between convexity, differentiability and the sub-differential. It holds that f is differentiable at $\mathbf{x}$ if and only if its sub-differential is a singleton $\{g\}$ for $g := \nabla f(\mathbf{x})$. In addition, f is convex if and only if $\partial f(\mathbf{x}) \neq \emptyset$ for every $\mathbf{x} \in \text{dom} f$.

■ **Example 10.4** Recall the unconstrained form of the Soft-SVM objective $f(\mathbf{w}) = \lambda ||\mathbf{w}||^2 + \sum_i \ell^{hinge}(\mathbf{w}, (\mathbf{x}_i, y_i))$ for $\ell^{hinge}(\mathbf{w}, (\mathbf{x}, y)) := \max\{0, 1 - y\langle \mathbf{w}, \mathbf{x}\rangle\}$. Focusing on the hinge loss, it is a convex function being the maximum of two convex (and differentiable) functions $f_1(\mathbf{w}) := 1 - y\langle \mathbf{w}, \mathbf{x}\rangle$ and $f_2(\mathbf{w}) = 0$.

For every $\mathbf{w}$ such that $y\langle \mathbf{w}, \mathbf{x}\rangle < 1$ then $\ell^{hinge}(\mathbf{w}) = f_1(\mathbf{w})$. For such $\mathbf{w}$, as the function is differentiable, then $\partial \ell^{hinge}(\mathbf{w}) = \{\nabla f_1(\mathbf{w})\} = \{-y\mathbf{x}\}$. For every $\mathbf{w}$ such that $y\langle \mathbf{w}, \mathbf{x}\rangle > 1$ then $\ell^{hinge}(\mathbf{w}) = f_2(\mathbf{w}) = 0$. Here too the function is differentiable and therefore $\partial \ell^{hinge}(\mathbf{w}) = \{\nabla f_2(\mathbf{w})\} = \{\mathbf{0}\}$. Lastly, in the case of $\mathbf{w}$ such that $y\langle \mathbf{w}, \mathbf{x}\rangle = 1$, though the function is not differentiable, it holds that $\mathbf{0}, -y\mathbf{x} \in \partial \ell^{hinge}(\mathbf{w})$. Put together for every $\mathbf{w} \in \text{dom} f$ a subgradient of the hinge loss at $\mathbf{w}$ is:

$$\mathbf{v} = \begin{cases} \mathbf{0} & y\langle \mathbf{w}, \mathbf{x}\rangle \geq 1 \\ -y\mathbf{x} & y\langle \mathbf{w}, \mathbf{x}\rangle < 1 \end{cases}$$

Thus, for $\mathbf{w} \in \mathbb{R}^d$, a vector v defined as $\mathbf{v} = 2\lambda\mathbf{w} - \sum_{i|y_i\langle \mathbf{w}, \mathbf{x}_i\rangle < 1} y_i \mathbf{x}_i$ is a subgradient of the Soft-SVM objective at $\mathbf{w}$. Note that for $y\langle \mathbf{x}, \mathbf{w}\rangle = 1$ we could have chosen any vector $\mathbf{v}$ such that $y\langle \mathbf{x}, \mathbf{v}\rangle = 1$ and with a slope in $[-1, 0]$.

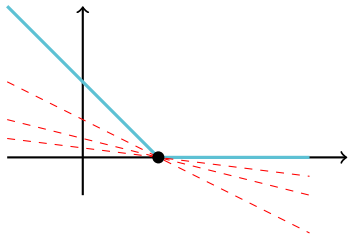


Figure 10.10: *ℓ^{hinge} with potential subgradients at $\mathbf{w}$ such that $y\langle \mathbf{x}, \mathbf{w}\rangle = 1$*

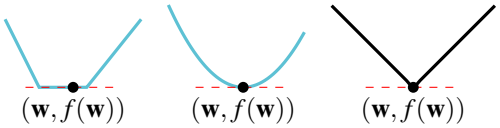
■

Claim 10.3.1 Let $f_1, \ldots, f_k$ be a set of convex differentiable functions and $f(\mathbf{w}) = \max_i f_i(\mathbf{w})$. Given $\mathbf{w} \in \text{dom} f$ then $\nabla f_j(\mathbf{w}) \in \partial g(\mathbf{w})$ for $j = \text{argmax}_i g_i(\mathbf{w})$.

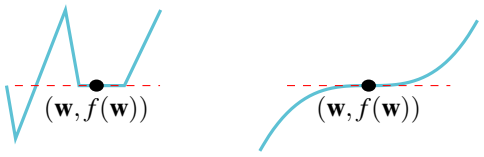
By substituting $\mathbf{v}$ as $\mathbf{0}$ in the sub-gradient definition (10.3.1) is we notice the following interesting observation.

Lemma 10.3.2 Let $f : \mathbb{R}^d \to \mathbb{R}$ be a convex function. If $\mathbf{0} \in \partial f(\mathbf{w})$ then $\mathbf{w}$ is a global minimum of f.

We further say that if f is also differentiable at $\mathbf{w}$ we say that f is "locally flat" around $\mathbf{w}$. In each of the following cases $\mathbf{0}$ is a sub-gradient at $\mathbf{w}$. In the blue graphs f is also differentiable at $\mathbf{w}$ and therefore "locally flat" around it.



For *non-convex* f, a point with a zero-slope tangent does not have to be a global (or a local) minimum and in particular may not have any sub-gradient, even if f is differentiable and locally flat around it:



10.3.2 Sub-Gradient Descent

Using sub-gradients, let us derive an algorithm for minimizing a convex function f, which might not be differentiable for any $\mathbf{x} \in \text{dom} f$. To do so we need to decide on a strategy for choosing the step direction and halting criterion. In the sub-gradient descent algorithm we chose to step in the direction of one of the sub-gradients $\mathbf{v} \in \partial f(\mathbf{x})$, which since f is a convex function then $\partial f(\mathbf{x}) \neq \emptyset$ for any $\mathbf{x} \in \text{dom} f$. Notice that for $\mathbf{x} \in \text{dom} f$, if f is differentiable at $\mathbf{x}$ then $\partial f(\mathbf{x}) = \{\nabla f(\mathbf{x})\}$. This means that for a differentiable point we are still able to step in the direction specified by $\Delta \mathbf{x} := -\nabla f(\mathbf{x})$.

Next, we wish to know when should the algorithm halt. By the definition of sub-gradients it holds that $\mathbf{v} \in \partial f(\mathbf{x})$ if and only if $\forall \mathbf{u} \in \text{dom} f \ f(\mathbf{u}) \geq f(\mathbf{x} + \langle \mathbf{v}, \mathbf{u} - \mathbf{x} \rangle))$. Notice that if $\mathbf{0} \in \partial f(\mathbf{x})$ it implies that $f(\mathbf{u}) \geq f(\mathbf{x})$ for all $\mathbf{u} \in \text{dom} f$. Namely, any direction is an *ascend* direction. Thus, the *sub-gradient optimality condition* is that $\mathbf{x}^*$ is a local minimizer of f if and only if $\mathbf{0} \in \partial f(\mathbf{x}^*)$, and as f is convex then $\mathbf{x}^*$ is a global minimizer.

■ **Example 10.5** Using the sub-gradient optimality condition let us find a necessary and sufficient condition for a point to be a minimizer of the Lasso problem. Let $\mathbf{X} \in \mathbb{R}^{m \times d}$ and $\mathbf{y} \in \mathbb{R}^m$ be a regression problem and consider

$$\min_{\mathbf{w} \in \mathbb{R}^d} \frac{1}{2} ||\mathbf{y} - \mathbf{X}\mathbf{w}||_2^2 + \lambda ||\mathbf{w}||_1, \quad \lambda \geq 0$$

This objective is convex but not differentiable. By the sub-gradient optimality condition then:

$$\mathbf{0} \in \partial \left(\frac{1}{2} ||\mathbf{y} - \mathbf{X}\mathbf{w}||_2^2 + \lambda ||\mathbf{w}||_1 \right) \iff \mathbf{0} \in -\mathbf{X}^\top (\mathbf{y} - \mathbf{X}\mathbf{w}) + \lambda \partial ||\mathbf{w}||_1$$

Therefore, $\mathbf{0}$ is a sub-gradient if and only if there exists $\mathbf{v} \in \partial ||\mathbf{w}||_1$ such that $\mathbf{X}^\top (\mathbf{y} - \mathbf{X}\mathbf{w}) = \lambda \mathbf{v}$. Generalizing Example 10.2 to $\mathbb{R}^d$ it can be shown that if $\mathbf{v} \in \partial ||\mathbf{w}||_1$ then

$$v_i = \begin{cases} 1 & w_i > 0 \\ -1 & w_i < 0 \\ [-1, 1] & w_i = 0 \end{cases}$$

Denote φ_i the i-th column of $\mathbf{X}$. Then, the sub-gradient optimality condition for the Lasso is that $\mathbf{w}$ is an optimal solution if and only if

$$\begin{array}{ll} \langle \varphi_i, \mathbf{y} - \mathbf{X}\mathbf{w} \rangle = \lambda \text{sign}(w_i) & w_i \neq 0 \\ |\langle \varphi_i, \mathbf{y} - \mathbf{X}\mathbf{w} \rangle| \leq \lambda & w_i = 0 \end{array}$$

Observing these conditions, if $\lambda = 0$ then we recover the condition seen for the Least Squares solution, where $\mathbf{w}$ is an optimal solution if and only if $\langle \varphi_i, \mathbf{y} - \mathbf{X}\mathbf{w} \rangle = 0$, $\forall i \in [d]$. That is, if the features φ_i are orthogonal to the residual vector $\mathbf{y} - \mathbf{X}\mathbf{w}$. For $\lambda > 0$, these conditions imply that $\mathbf{w}$ is an optimal solution if the features φ_i in the active set (i.e those where $w_i \neq 0$) form a constant angle θ with the residual vector:

$$||\varphi_i|| \, ||\mathbf{y} - \mathbf{X}\mathbf{w}|| \, cos\theta = \langle \varphi_i, \mathbf{y} - \mathbf{X}\mathbf{w} \rangle = \lambda \text{sign}(w_i) \quad \forall i \in [m]$$

Notice, that in the case of an orthogonal design $\mathbf{X}^\top \mathbf{X} = I$ the vector $\eta_\lambda^{soft}(\mathbf{w}^{LS})$ satisfies the Lasso sub-gradient optimality conditions derived above. ■

And so, the sub-gradient descent is a first-order convex optimization algorithm suitable for convex problems with non-differentiable objectives. We replace the gradient descent iteration with the following update rule.

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \eta_{t+1}\mathbf{v}^{(t)}, \quad \mathbf{v}^{(t)} \in \partial f\left(\mathbf{x}^{(t)}\right) \tag{10.9}$$

When stepping in the direction of some arbitrary sub-gradient there is no guarantee that this is in fact a descent direction. Therefore, it makes no sense to return the last vector $\mathbf{x}^{(T)}$. We will return either the best-performing or average vector.

Algorithm 18 Sub-Gradient Descent

procedure SUB-GRADIENT DESCENT(f)
  Initialize $\mathbf{x}^{(1)} \in \mathbb{R}^d$
  for $t = 1, \ldots, T$ **do**
    $\eta \leftarrow$ LINE-SEARCH($\mathbf{x}^{(t)}$)
    Pick $\mathbf{v}^{(t)} \in \partial f\left(\mathbf{x}^{(t)}\right)$
    $\mathbf{x}^{(t+1)} \leftarrow \mathbf{x}^{(t)} - \eta_{t+1}\mathbf{v}^{(t)}$
  end for
  return $\bar{\mathbf{x}} = \frac{1}{T}\sum \mathbf{x}^{(t)}$
end procedure

R Recall the Perceptron algorithm (1) and notice that this is in fact a specific case of the sub-gradient descent algorithm.

Just as in the case of gradient descent, the specification of a step size is an important issue. However, as the sub-gradient of f at $\mathbf{x}$ is not necessarily a descent direction we do not have adaptive methods such as line each to determine the appropriate step size. Instead we use some decay function (as the linear or exponential examples previously seen) which converges to zero but does so "not too fast". For example some function for which $\sum_{t=1}^{\infty} \eta_t^2 < \infty$ but $\sum_{t=1}^{\infty} \eta_t = \infty$.

10.3.2.1 Convergence Analysis

In the case of gradient descent, where we assume f to be convex and differentiable, we have seen that by stepping in the direction $\Delta\mathbf{x} := -\nabla f(\mathbf{x})$ and with size $\eta > 0$ we are guaranteed to converge to the minimum. In the case of sub-gradient descent however, as the direction at time t is not necessarily a descent direction, it is a-priori unclear whether the algorithm converges to the optimal solution.

To guarantee convergence to the optimal solution we assume the set of feasible solutions C is bounded and that f does not vary "too wildly". Namely, that f is Lipschitz continuous for some ρ. In this setup of f is convex-Lipchitz-bounded and therefore, as seen in Lemma 10.1.1, is PAC learnable and will converge to the optimal solution. If f is Lipchitz but not bounded, that is $\mathrm{dom} f = \mathbb{R}^d$, the sub-gradient descent with a diminishing step-size is still guaranteed to converge $\lim_{t\to\infty} f(\mathbf{x}_{best}^{(t)}) = f^*$, for $\mathbf{x}_{best}^{(t)}$ the best-performing point among $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(t)}$ and f^* the optimal value of f.

10.4 Stochastic Gradient Descent

The Gradient Descent algorithm is not a learning algorithm but an *optimization algorithm* used in order to implement certain learning algorithms. If the specific application of a learning algorithm to the training data

happens to require the solution of a convex optimization problem, we can use Gradient Descent (or more generally, Sub-Gradient Descent) to solve it.

The types of learning algorithms or more generally learning principles such as ERM or Maximum Likelihood, were applied to the full set of training data in order to choose one hypothesis h out of a hypothesis class, $\mathcal{H}$. Realistic learning problems can often involve large number of variables and samples making it unreasonable to process, access or even store, in a single stage of the calculation. Moreover, as we try to process one batch of training data, new training data may be piling up. Stochastic Gradient Descent is a *learning algorithm* based on the ideas of Gradient Descent which allows us to cope with these problems.

Definition 10.4.1 Let $G \in \mathbb{R}^d$ be a vector-valued random variable. The iteration

$$\mathbf{w}^{(t+1)} = \mathbf{w}^t - \eta_{t+1}\mathbf{g}^{(t)}$$

is called a *stochastic gradient descent* (SGD) iteration for f if and only if for each iteration t it holds that $\mathbf{g}^{(t)} \overset{ind.}{\sim} G^{(t)}$ is a random vector with $\mathbb{E}\left[G^{(t)}\right] \in \partial f\left(\mathbf{w}^{(t)}\right)$.

That is, for an iterative algorithm with the update rule above the algorithm proceeds in each step in a random direction which falls in $\partial f(\mathbf{w}^{(t)})$ in *expectation*. Notice that if f is differentiable at $\mathbf{w}^{(t)}$ then $\mathbb{E}\left[G^{(t)}\right] = \nabla f\left(\mathbf{w}^{(t)}\right)$ and therefore in *expectation* the algorithm proceeds in the steepest direction that minimizes f. As we will formulate below, the application of the SGD algorithm will be over small random subsets of the training set, each time stepping in the direction minimizing the loss function over the random (stochastic) subset.

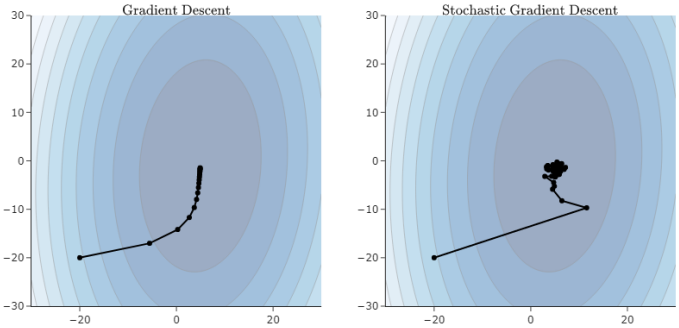


Figure 10.11: *GD- (left) and SGD (right) descent profiles.* Descent Methods Examples

■ **Example 10.6** Consider the minimization problem of the convex objective f of the form $f(\mathbf{w}) = \sum_{i=1}^{m} f_i(\mathbf{w})$ where f_i are all convex functions. Since $\partial \sum_i f_i(\mathbf{w}) = \sum_i \partial f_i(\mathbf{w})$ a *sub-gradient* iteration of f would be

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta_{t+1} \sum_i \mathbf{g}_i^{(t)} \quad \text{where} \quad \mathbf{g}_i^{(t)} \in \partial f_i(\mathbf{w}^{(t)})$$

Now, suppose at iteration t we select k_t uniformly at random according to $k_t \overset{i.i.d}{\sim} \text{Unif}(1,\ldots,m)$ and then perform the iteration

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta_{t+1}\mathbf{g}_{k_t}^{(t)}$$

The vector $\mathbf{g}_{k_t}^{(t)}$ is a random vector drawn from the probability space $\mathbf{g}_1^{(t)},\ldots,\mathbf{g}_m^{(t)}$, each with probability of $\frac{1}{m}$. The expectation of the random vector is the sum $\sum_i \mathbf{g}_i^{(t)}$ which is the gradient of f. Thus this is a *stochastic* gradient descent iteration.

We could go further an generalize the above to sample a *mini-batch* of samples of some fixed size. For $K_t \subset \{1,\ldots,m\}$ a uniformly distributed random subset (or a bootstrap sample) we can perform the iteration

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \frac{\eta_{t+1}}{|K_t|}\sum_{j\in K_t}\mathbf{g}_j^{(t)}$$

■

How would the bias-variance properties change depending on our sampling strategy? If we perform SGD with a single sample at each iteration the directions in which the algorithm will move will vary greatly between iterations. As we increase the size of the mini-batch, the step direction is averaged over more and more samples reducing the variance. For this reason, when outputting the trained model we will return the model being the average of all steps $\frac{1}{T}\sum_t \mathbf{w}^{(t)}$.

In addition to specifying the batch size we are able to describe many different strategies for selecting samples in the different SGD iterations. For example, instead of choosing a random sample (or mini-batch) we can simply cycle through the training set. This is termed cyclic SGD. The number of SGD iterations required to use the entire training set once is called an epoch.

Notice that when using SGD we are no longer following the batch learning paradigm. We are no longer restricted to the framework of receiving a set of samples, training a model over these samples and outputting a "final" model. Instead, at each iteration the learner is only concerned of a smaller set of samples. It might be the case that we have trained a model $\mathbf{w}_{S_1}$ over a given sample S_1 (using some number of iterations). Then, after we have "finished" training $\mathbf{w}_{S_1}$ we have received another traning sample S_2. We can now simply "continue" training $\mathbf{w}_{S_1}$ by performing more SGD iterations using S_2 to produce our "final" model. This enables us to consider a *streaming* model where the model is *continuously* trained using SGD as data continuously comes in.

10.4.1 Optimization for $L_{\mathcal{D}}$

By learning using SGD we are in fact defining a new learning principle. Consider a convex learning problem with loss function ℓ and assume the agnostic PAC framework that $z = (\mathbf{x}, y) \overset{i.i.d}{\sim} \mathcal{D}$ for some probability distribution $\mathcal{D}$ over $\mathcal{X}\times\mathcal{Y}$. Our goal is to (probably approximately) solve

$$\mathbf{w}^* \coloneqq \operatorname*{argmin}_{\mathbf{w}\in\mathcal{H}} L_{\mathcal{D}}(\mathbf{w}) \quad \text{where} \quad L_{\mathcal{D}}(\mathbf{w}) = \mathop{\mathbb{E}}_{z\sim\mathcal{D}}\left[\ell(\mathbf{w},z)\right]$$

As we do not have a direct way to optimize the generalization error $L_{\mathcal{D}}(\mathbf{w})$ we argued in favor of the ERM learning principle where we optimized for the empirical error $L_S(\mathbf{w})$ as a proxy to the generalization error. We developed the theory and found conditions under which we could expect good generalization.

SGD however, provides us with a method to *directly* optimize for the generalization error. If we would have known $L_{\mathcal{D}}$, we could have used GD iterations starting from $\mathbf{w}^{(1)} = 0$ and updating $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta\nabla L_{\mathcal{D}}(\mathbf{w})$. Unfortunately as we know, we can not assume to know $\mathcal{D}$. Notice however that

$$\nabla L_{\mathcal{D}}(\mathbf{w}) = \nabla \mathop{\mathbb{E}}_{z\sim\mathcal{D}}\left[\ell(\mathbf{w},z)\right] = \mathop{\mathbb{E}}_{z\sim\mathcal{D}}\left[\nabla\ell(\mathbf{w},z)\right] \tag{10.10}$$

(and similarly in the case fof sub-gradients). In other words, $\nabla\ell(\mathbf{w},z)$ is an *unbiased estimator* of $\nabla L_{\mathcal{D}}(\mathbf{w})$. As such, by selecting $z_t \sim \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ and updating according to

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta_{t+1}\nabla\ell\left(\mathbf{w}^{(t)}, z_t\right)$$

we perform an SGD iteration for the generalization error in spite the fact that we do not know $\mathcal{D}$. The following lemma tells us that for a sufficient number of iterations we will indeed converge to the minima.

Lemma 10.4.1 Consider a convex-Lipschitz-bounded learning problem with parameters ρ, B. Then, for every $\varepsilon > 0$, if we run the SGD method for minimizing $L_{\mathcal{D}}(\mathbf{w})$ with a number of iterations $T \geq \frac{B^2\rho^2}{\varepsilon^2}$ and $\eta = \sqrt{\frac{B^2}{\rho^2 T}}$ then the output of the SGD, $\mathbf{w}'$, satisfies

$$\mathbb{E}\left[L_{\mathcal{D}}(\mathbf{w}')\right] \leq \min_{\mathbf{w}\in\mathcal{H}} L_{\mathcal{D}}(\mathbf{w}) + \varepsilon$$

10.5 Summary, Labs & Exercises

Exercises

Theoretical Questions

1. Show that the Absolute Value Loss optimization problem is a convex optimization problem.

2. Let $f : \mathbb{R}^d \to \mathbb{R}$ be differential at $\mathbf{x} \in \text{dom} f$. Show that $\frac{\nabla f(\mathbf{x})}{\|\nabla f(\mathbf{x})\|}$ is the vector that points to the direction of steepest ascent in $f(\mathbf{x})$. Similarly, show that $-\frac{\nabla f(\mathbf{x})}{\|\nabla f(\mathbf{x})\|}$ is the vector that points to the direction of steepest descent in $f(\mathbf{x})$.
3. Let $f : \mathbb{R}^d \to \mathbb{R}$ then f is differentiable at $\mathbf{x} \in \text{dom} f$ if and only if $\partial f(\mathbf{x}) = \{\nabla f(\mathbf{x})\}$.

4. Let $f : \mathbb{R}^d \to \mathbb{R}$ then f is convex if and only if $\partial f(\mathbf{x}) \neq \emptyset$ for every $\mathbf{x} \in \text{dom} f$.

5. Prove Lemma 10.3.1: Let $f_1, \ldots, f_k$ be a set of convex differentiable functions and $f(\mathbf{w}) = \max_i f_i(\mathbf{w})$. Given $\mathbf{w} \in \text{dom} f$ then $\nabla f_j(\mathbf{w}) \in \partial g(\mathbf{w})$ for $j = \text{argmax}_i g_i(\mathbf{w})$.

6. Let f be a differentiable function and consider the optimization problem of minimizing $f(\mathbf{x})$ subject to $\mathbf{x} \in C$. Derive the first-order optimality condition from the sub-gradient optimality condition for the equivalent problem of minimizing $f(\mathbf{x}) + \mathbb{1}_{\mathbf{x}\in C}$.

7. Show that the sub-differential of the ℓ_1 norm is:

$$\partial \|\mathbf{x}\| = \left\{ (v_1, \ldots, v_d) \in \mathbb{R}^d \,\middle|\, v_i = \begin{cases} 1 & x_i > 0 \\ -1 & x_i < 0 \\ [-1, 1] & x_i = 0 \end{cases} \right\}$$

8. Let $\mathbf{X}, \mathbf{y}$ be an orthogonal design regression problem and $\eta_\lambda^{soft}(\mathbf{w}^{LS})$ the entry-wise soft threshold (with parameter λ) of the Least Squares solution $\mathbf{w}^{LS}$. Show by direct calculation that $\eta_\lambda^{soft}(\mathbf{w}^{LS})$ satisfies the Lasso sub-gradient optimality conditions shown in Example 10.5.

9. Prove Example 10.1: Given an unconstrained quadratic program

$$\mathbf{x}^* = \underset{\mathbf{x}\in\mathbb{R}^d}{\text{argmin}}\, f(\mathbf{x}), \quad f(\mathbf{x}) := \tfrac{1}{2}\mathbf{x}^\top Q\mathbf{x} + \mathbf{b}^\top\mathbf{x} + c$$

for a symmetric $Q \in \mathbb{R}^{d\times d}, \mathbf{b} \in \mathbb{R}^d$ and $c \in \mathbb{R}$, use the first-order optimality condition to show that:
   - If $Q \succ$ the optimal solution is unique and is given by $\mathbf{x}^* = Q^{-1}\mathbf{b}$.
   - If Q is singular and $\mathbf{b} \notin \text{Im}(Q)$ there are no optimal solutions.
   - If Q is singular and $\mathbf{b} \in \text{Im}(Q)$ any point of the form $\mathbf{x}^* = Q^\dagger\mathbf{b} + \mathbf{z}$, for $\mathbf{z} \in \text{Ker}(Q)$ is an optimal solution.

11. Deep Learning & Neural Networks

Neural networks have revolutionized the world of machine learning and though the hype around the topic is high, the essence of neural networks and deep learning is fairly simple. We begin with explaining the structure of a neural network (referred to as *architecture*). Once the structure is clear we discuss how to efficiently train a neural network.

11.1 Neural Network Architecture

A neural network describes a computational model for computing complex functions using *basic units* termed "neurons". Each neuron is a simple computational unit $f : \mathbb{R}^d \to \mathbb{R}$ with two parts:

- First, for a given input $\mathbf{x} = (x_1, \ldots, x_d)^\top$ we take the weighted sum (inner-product) with respect to a set of weights $\mathbf{w} = (w_1, \ldots, w_d)^\top$ $a \coloneqq \langle \mathbf{w}, \mathbf{x} \rangle = \sum_{j=1}^{d} x_j w_j$.
- Then, we pass the calculated value through some (non-linear) *activation function* to acquire some output (*activation*) $o \coloneqq \sigma(a)$.

Thus, a single neuron can be described as a function $f : \mathbb{R}^d \to \mathbb{R}$ defined by $f(\mathbf{x}) \coloneqq \sigma(\langle \mathbf{w}, \mathbf{x} \rangle)$. Schematically a neuron is represented as in Figure 11.1.

R The terminology of neurons and neuronal network originates from brain science where the basic computation unit in the brain is of a type of cell called neuron. These cells receive multiple inputs from other neurons. The inputs are integrated (summation and negation) and if pass a certain threshold "fire", outputting a signal to downstream connected neurons. If we consider the activation function in Figure 11.1 to be for example $\sigma(a) = \mathbb{1}_{a \geq \theta}$ then the output o will be 1 (that is "fired") if and only if the weighted sum of the inputs is sufficiently large.

Previously in this book we have already encountered a simple form of neural network when we discussed logistic regression (section 3.4). The hypothesis class of the logistic regression model is of functions $\mathbf{x} \to \sigma(\langle \mathbf{w}, \mathbf{x} \rangle)$ where σ is the *logit* (sigmoidal) function $\sigma(a) = e^a / (1 + e^a)$. The learning principle we used

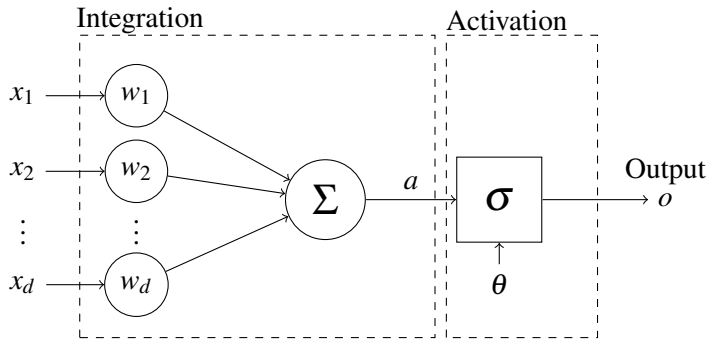


Figure 11.1: *Schematic diagram of a single neuron computation unit*

to select an hypothesis from the hypothesis class was of maximum likelihood and the loss function derived was

$$\ell(\mathbf{w}, S) = \sum_{i=1}^{m} \left[\mathbb{1}_{y_i=1} \cdot \log\left(\sigma\left(\langle \mathbf{w}, \mathbf{x}_i \rangle\right)\right) + \mathbb{1}_{y_i=0} \cdot \log\left(1 - \sigma\left(\langle \mathbf{w}, \mathbf{x}_i \rangle\right)\right) \right]$$

Since the negative log-likelihood is a convex function we can using SGD to train a logistic model. By doing so, we are in fact training what is known as a *feed-forward* neural network for binary classification with no *hidden layers*. Another example for such a neural network is of the Perceptron algorithm (section 3.2).

Using single neurons as building blocks we can now compose them to compute more complex functions. Suppose we fit a logistic regression model to obtain the weights vector $\mathbf{w}_1$ and predict by $f_1(\mathbf{x}) = \sigma\left(\langle \mathbf{w}_1, \mathbf{x} \rangle\right)$. Now, we can fit a second model to obtain the weights vector $\mathbf{w}_2$ and predict by $f_2(\mathbf{x}) = \sigma\left(\langle \mathbf{w}_2, \mathbf{x} \rangle\right)$. We continue doing so for k times obtaining the weights vectors $\mathbf{w}_1, \ldots, \mathbf{w}_k \in \mathbb{R}^d$. We can arrange these k neurons into a single *layer* and consider the hypothesis class that consists of functions $f : \mathbb{R}^d \to \mathbb{R}^k$ defined by

$$f(\mathbf{x}) := \begin{bmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ \vdots \\ f_k(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} \sigma\left(\langle \mathbf{w}_1, \mathbf{x} \rangle\right) \\ \sigma\left(\langle \mathbf{w}_2, \mathbf{x} \rangle\right) \\ \vdots \\ \sigma\left(\langle \mathbf{w}_k, \mathbf{x} \rangle\right) \end{bmatrix} = \sigma(\mathbf{W}\mathbf{x}) \tag{11.1}$$

where for the last equality we define $[\sigma(\mathbf{a})]_i = \sigma(a_i)$. Then, the hypothesis class is the set of functions $f_{\mathbf{W}}(\mathbf{x}) = \sigma(\mathbf{W}\mathbf{x})$ for $\mathbf{W} \in \mathbb{R}^{k \times d}$.

$$\mathcal{H} := \left\{ \mathbf{x} \to \sigma(\mathbf{W}\mathbf{x}) \;\middle|\; \mathbf{W} \in \mathbb{R}^{k \times d} \right\} \tag{11.2}$$

It is important to notice that the structure (i.e. *architecture*) of the networks in this hypothesis class is identical. They all take the form of linearly transforming the input vector, $\mathbf{W}\mathbf{x}$, and then passing the result through the activation function φ. What changes between different networks (hypotheses) in this class are the values of the model parameters, i.e the values in $\mathbf{W}$.

Schematically a layer of a neural network with d input values and k neurons can be represented as in Figure 11.2 where the left column represents the input values (each node is a single scalar), the right column represents the output values (each node is a single scalar) and the edges are the weights $\mathbf{W}$. Notice that each node in the output values is a different neuron. Thus, this diagram depicts 3 different neurons, each with its own set of weights and performing the inner-product and activation function independent to the other neurons.

We can now go a step further and use the output values of one layer as input values for an additional layer. By composing two such layers we describe hypotheses of the form

$$(f_2 \circ f_1)(\mathbf{x}) = \sigma_2\left(\mathbf{W}_2 \cdot \sigma_1\left(\mathbf{W}_1 \mathbf{x}\right)\right) \tag{11.3}$$

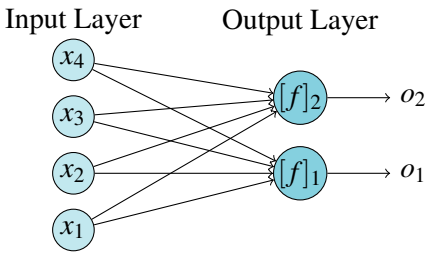


Figure 11.2: *Schematic diagram of a neural network layer*

where $\mathbf{W}_1, \mathbf{W}_2$ are the weights of the first and second layers, and σ_1, σ_2 the activations of the first and second layers. Thus, we compose one layer on top of another forming a deeper network. For network with T layers the composition of the layers computes

$$\mathbf{x} \to f_1(\mathbf{x}) \to f_2(f_1(\mathbf{x})) \to \ldots \to f_T(f_{T-1}(\ldots f_1(\mathbf{x})\ldots)) \tag{11.4}$$

where $f_1, \ldots, f_T$ are each a function of the form $f_t(\mathbf{o}_{t-1}) = \sigma_t(\mathbf{W}_{t-1}\mathbf{o}_{t-1})$ for some selection of activation functions $\sigma_1, \ldots, \sigma_T$. We term $f_1, \ldots, f_{T-1}$ the *hidden layers* of the network.

By following (11.4) we can derive very interesting networks fitted for different sorts of tasks. Notice that the only difference between performing a regression- or classification task is in the form of the output layer. If the output function returns a continuous value (i.e. $\sigma_T \equiv \mathrm{Id}$), the network is such of a regression task. If the output function returns a value from a discrete set, the network is such of a classification task. Further, if the output layer consists of multiple output neurons (i.e $\sigma_T : \mathbb{R}^d \to \mathbb{R}^k$ where $k > 1$) the network performs a task of multiple outputs being either multiple-regression or multi-classification.

This type of neural network architecture, in which each layer's output is fed *directly* to the following layer's input is called a *feed-forward* neural network. Formally,

Definition 11.1.1 The hypothesis class of a *fully connected feedforward network* with $T \in \mathbb{N}$ layers, taking input $\mathbf{x} \in \mathbb{R}^d$ and produces k outputs is

$$\mathcal{H}_{G,\{\sigma\},\phi} \coloneqq \left\{ \langle G, w, \{\sigma\}, \phi \rangle \;\middle|\; w : E \to \mathbb{R} \right\}$$

where:

- $G = \langle V, E \rangle$ is a *Directed Acyclic Graph* where $V = \biguplus_{t=0}^{T} V_t$ are the T layers
- $w : E \to \mathbb{R}$ is the weight function, given as the matrices $\mathbf{W}_0, \ldots, \mathbf{W}_{T-1}$
- $\sigma_1, \ldots, \sigma_T : \mathbb{R} \to \mathbb{R}$ are the activation functions
- ϕ the network's output function $\phi : \mathbb{R}^{|V_T|} \to \mathbb{R}^{|V_T|}$ where $|V_T| = k$

and the output of the layer t is given by:

$$\mathbf{o}_t = \begin{cases} \mathbf{x} & t = 0 \\ \sigma_t(\mathbf{W}_{t-1} \cdot \mathbf{o}_{t-1}) & t \in [T] \end{cases}$$

Thus, Definition 11.1.1 defines a *family* of hypothesis classes where the architecture is set by the hyper-parameters $G, \{\sigma\}, \phi$ and the different networks by the different weight functions.

Similar to the approach seen in linear regression, if we wish to include an intercept to a given layer we add an additional neuron to the preceding layer outputting some constant value. Figure 11.3 shows an example for a

4-layered feed-forward neural network. It receives as input a vector $\mathbf{x} \in \mathbb{R}^3$. Layer 1 integrates these inputs while including an intercept calculating a function $f_1 : \mathbb{R}^3 \to \mathbb{R}^5$. Layer 2 uses the output from layer 1 and integrates them while including an intercept calculating a function $f_2 : \mathbb{R}^5 \to \mathbb{R}^5$. The next layer calculates a function $f_3 : \mathbb{R}^5 \to \mathbb{R}^3$. All together this network describes the computation of a complex function $f : \mathbb{R}^3 \to \mathbb{R}^3$.

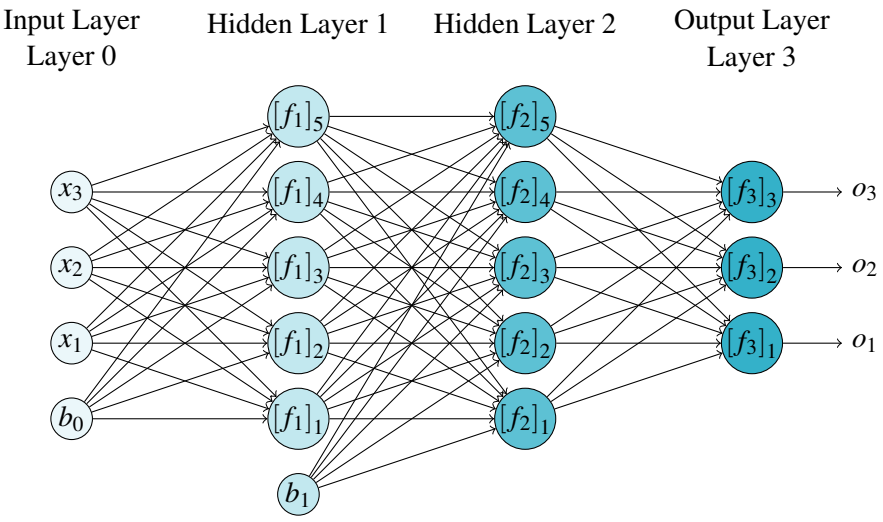
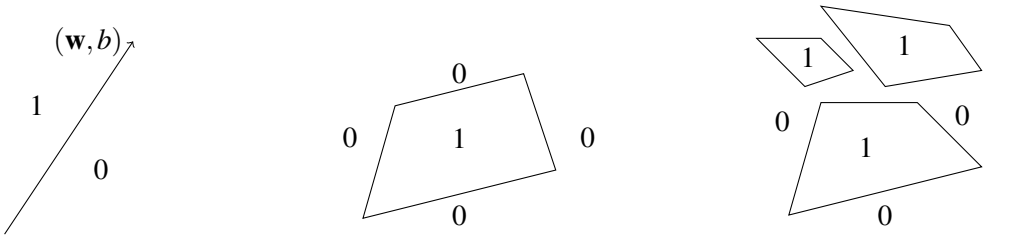


Figure 11.3: *Schematic diagram of a 4-layered feed-forward neural network*

11.1.1 Expressive Power of Neural Networks

Given Definition 11.1.1 of a neural network, the next question is how rich is this hypothesis class. Let us consider networks with a single output neuron, with the output function being $\phi(o) := \mathbb{1}_{o>0}$ and activation $\sigma(a) := \mathbb{1}_{a>0}$. Even with such a restricted scenario, we can describe a wide range of functions. If the network has no hidden layers (i.e. $\mathbf{x} \to \phi(\sigma(\langle \mathbf{w}, \mathbf{x} \rangle))$) it implements a halfspace classifier in $\mathbb{R}^d$ (similar to the Perceptron algorithm). If we now enable a single hidden layer with k neurons, the network can learn the intersections of $k-1$ halfspace (i.e. all polytops in $\mathbb{R}^d$ with $k-1$ faces). We can further extend this by adding a second hidden layer including k_2 neurons. Such a network can therefore learn unions of k_2 $k-1$ -faced polytops in $\mathbb{R}^d$.



(a) *No hidden layers*

(b) *Single hidden layer with $k = 5$ neurons*

(c) *Two hidden layers with $k_1 = 5$ and $k_2 = 3$ neurons*

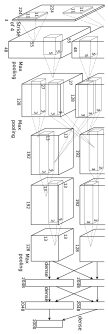
Figure 11.4: *Expressiveness for increasing number of layers*

Thus, by either increasing the depth of the network or the number of neurons at any given layer, we are able to express very complex functions. Specifically, if we focus on input vectors $\mathbf{x} \in \{0,1\}^d$ (boolean vectors),

for any $d \in \mathbb{N}$, we can construct a network with a single hidden layer such that for every boolean function $f : \{0,1\}^d \to \{0,1\}$ there exists a weight assignment of the network implementing such function. However, it holds that the minimal number of neurons required in the hidden layer is exponential in d. We therefore conclude that even with a single hidden layer the hypothesis class of neural networks is extremely rich.

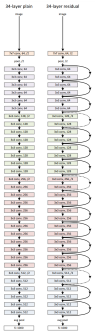
11.1.2 Deep Networks

As composition of layers, one after another, increases the richness of the hypothesis class and enables us to describe very elaborated function (Figure 11.4) we can start defining "blocks" of *multiple layers* and compose these blocks one after the other. Below are known stepping stone networks from recent years. In both the GoogLeNet and ResNet each of the blocks seen is not a single layer but rather a set of layers repeated multiple times throughout the network architecture. Further notice that these are not feed-forward networks as some layers are feeding their output beyond the scope of the layer directly after.



(a) AlexNet: *60 million parameters, Krizhevsky A. et al., 2012*

(b) GoogLeNet: *6.8 million parameters, Szegedy C. et al., 2014*



(c) ResNet: *60 million parameters, He. K. et al., 2015*

11.2 Training The Network

Once we defined the hypothesis class (i.e. the network architecture) the next step is to select a single hypothesis from it. As we have previously seen, we do so by defining a learning principle and loss function by which we select the optimal hypothesis. Training a neural network using the ERM learning principle is NP-hard and therefore we resort to using a different principle. Instead, we use the maximum likelihood principle.

Notice that the hypothesis class $\mathcal{H}_{G,\{\sigma\},\phi}$ of networks with a specified architecture is the set of functions $w : E \to \mathbb{R}$ for E the edges in the network graph. By specifying an order over the edges (e.g first by layer, then by in-going neuron and then by out-going neuron) we can represent each hypothesis as a vector in $\mathbf{w} \in \mathbb{R}^{|E|}$ and the hypothesis class as a whole takes an Euclidean structure. As such, let $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ be a training set and denote $\ell_{\mathbf{w}}(\mathbf{x}, y)$ the negative log-likelihood, for $\mathbf{w}$ representing weights assignment of a neural network. We therefore would like to minimize

$$L(\mathbf{w}; S) \coloneqq \sum_{(\mathbf{x},y)\sim S} \ell_{\mathbf{w}}(\mathbf{x}, y)$$

In the notation $L(\mathbf{w}; S)$ we aim to explicitly distinguish the above from the empirical risk (whose minimization is NP-hard). Note, that while in the case of a logistic regression model the objective $L(\mathbf{w}; S)$ is convex in the weights $\mathbf{w}$, for a neural network with even a single hidden layer the objective is *highly non-convex* in $\mathbf{w}$.

Since the objective is not convex it is not clear how to minimize it. The *big* surprise in machine learning of recent years is that using Gradient Descent to maximize the likelihood for choosing the network weights

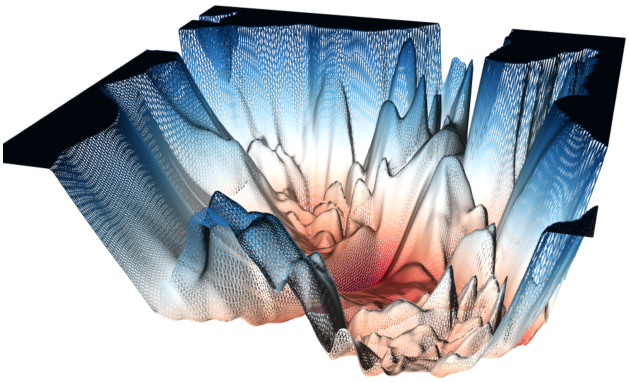


Figure 11.6: *Loss landscape visualization of ResNet-56 shows highly non-convex function*

yields a very powerful learning algorithm even though the target function is non-convex. Implementing GD means begin able to calculate the following update rule

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta_{t+1} \nabla L\left(\mathbf{w}^{(t)}; S\right)$$

for L the negative log-likelihood. This however is not a simple task. Calculating the gradient means calculating the partial derivatives with respect to each of the parameters of the model - that is, each one of the edges in the network. As we construct larger and deeper networks we have to calculate the partial derivative with respect to more parameters and the complexity of this calculation increases. For example, how would you calculate the partial derivative of L with respect to $w_{0,1,1}$, $\partial L_{w_{0,1,1}}(S)$, illustrated in Figure 11.7? Now suppose the network consists of millions if not billions of parameters. Further, as we increase the number of parameters in the model we must increase the training set size. And so the question is how to compute $\nabla L(\mathbf{w}, S)$ *efficiently*?

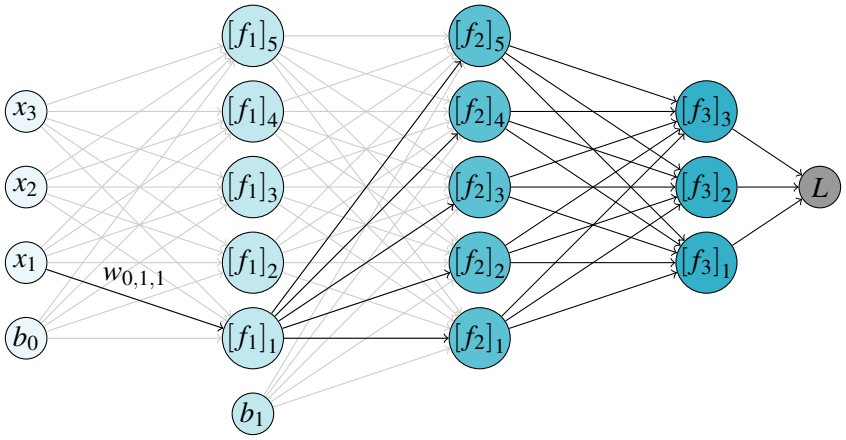


Figure 11.7: *Illustration of partial derivative*

To do so we will use two computational tricks:

- Rather than using GD we use SGD. By doing so, in each iteration of the algorithm instead of computing the gradient $\nabla_{\mathbf{w}} L(\mathbf{w}, S)$ over the entire training set we calculate $\partial_{\mathbf{w}} \ell(\mathbf{w}, z)$ for a single training sample (or a small mini-batch).
- Use a fast numerical algorithm for calculating $\partial_{\mathbf{w}} \ell(\mathbf{w}, z)$ known as *back-propagation*.

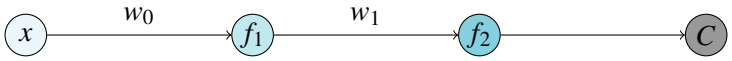
11.2.1 Insights From The Chain-Rule

The back-propagation algorithm relies on two simple yet smart observations regarding the calculations made using the chain-rule (subsection B.2).

■ **Example 11.1** Consider the following simple neural network with two single-neuron hidden layers without output evaluated under the MSE loss function where for each hidden layer i denote:

- The pre-activation by $a_i = \langle o_{i-1}, w_{i-1} \rangle$.
- The activation by $o_i = \sigma(a_i)$ and $o_0 = x$.

so $f_i(o_{i-1}) = \sigma(\langle o_{i-1}, w_{i-1} \rangle)$ and the loss function $C(o_2) = (o_2 - y)^2$.

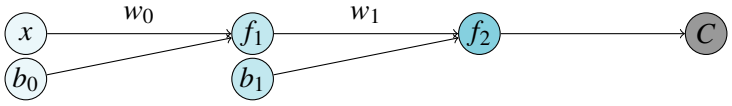


We would like to calculate the gradient of the composition $C \circ f_2 \circ f_1$ with respect to all of the parameters w_0, w_1. Using the chain rule then:

$$\frac{\partial C}{\partial w_2} = \frac{\partial C}{\partial o_2} \cdot \frac{\partial o_2}{\partial a_2} \cdot \frac{\partial a_2}{\partial w_2} = 2(o_2 - y) \cdot \sigma'(a_2) \cdot o_1$$

$$\frac{\partial C}{\partial w_1} = \frac{\partial C}{\partial o_2} \cdot \frac{\partial o_2}{\partial a_2} \cdot \frac{\partial a_2}{\partial o_1} \cdot \frac{\partial o_1}{\partial a_1} \cdot \frac{\partial a_1}{\partial w_1} = 2(o_2 - y) \cdot \sigma'(a_2) \cdot w_2 \cdot \sigma'(a_1) \cdot o_0$$

Let us update the network above and include a bias node for each of the hidden layers:



Notice that for the derivative with respect to the bias parameters we only need to replace $\frac{\partial a_i}{\partial w_i}$ with $\frac{\partial a_i}{\partial b_i}$ which equals to 1. Therefore, the full gradient is:

$$\nabla C = \left(\frac{\partial C}{\partial b_0}, \frac{\partial C}{\partial w_0}, \frac{\partial C}{\partial b_1}, \frac{\partial C}{\partial w_1} \right)^\top$$

■

Looking closely at the expressions derived using the chain rule we gain several insights. The first is that by applying the chain-rule we break down the dependencies between the functions into a linear chain of independent expressions where each can be calculated separately. These expressions take one of two forms:

- Inner-product derivatives $\frac{\partial a_i}{\partial w_{i-1}}$ - the values of these expressions are simply the outputs of the previous layers o_{i-1}. Obtaining these values can be done by calculating the output of the network function up to the current layer.
- Activation-function derivatives $\sigma'(a_i)$ - as these derivatives are only with respect to the inputs passed to the activation function it does not matter how complex the network is and we can choose scalar activation functions whose derivative is simple to calculate (e.g. ReLU, sigmoid, etc.)

A second insight is that chain-rules of different parameters share different expressions. In the example above $\frac{\partial C}{\partial o_2}$ and $\frac{\partial o_2}{\partial a_2}$ are shared across all four parameters b_0, w_0, b_1, w_1. In addition, if we look at b_0, w_1 we see that their chain rule also shares $\frac{\partial a_2}{\partial o_1}$ and $\frac{\partial o_1}{\partial a_1}$. Both of these insights suggest that we can derive an algorithm to compute the gradient step in a smart and efficient manner, regardless to the depth of the network.

Before using these insights to derive the back-propagation algorithm let us consider a more realistic feed-forward neural network $\langle G, w, \{\sigma\}, \phi \rangle$ with $T \in \mathbb{N}$ layers and in each layer $k_t \in \mathbb{N}$ neurons. Let us denote L_t the layer function that receives the output from the previous later and computes the current layer $L_t(\mathbf{o}_{t-1}) =$

$\sigma_t(\langle \mathbf{o}_{t-1}, \mathbf{W}_{t-1} \rangle)$. In addition let us consider the pre-activation $\mathbf{a}_t = \langle \mathbf{o}_{t-1}, \mathbf{W}_{t-1} \rangle$ as the function that receives $\mathbf{o}_{t-1}$ and $\mathbf{W}_{t-1}$ and calculates their inner product. Similarly let us consider the activation $\mathbf{o}_t = \sigma_t(\mathbf{a}_t)$ the function that receives $\mathbf{a}_t$ as input and calculates the activation.

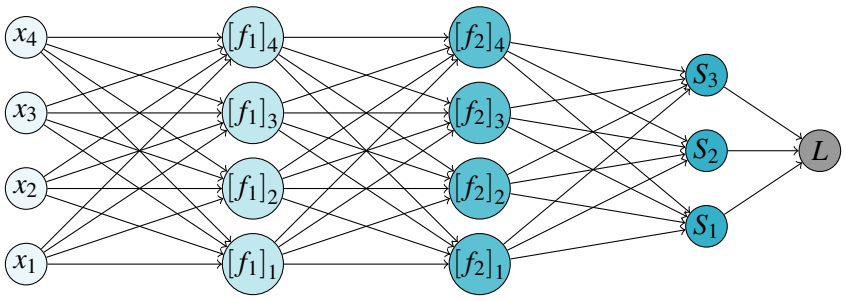
To calculate the gradient of the entire network We wish to ask how does the network's output change with respect to each of the network parameters (Figure 11.7). This time, as we are dealing with multivariate functions with multiple input values and multiple parameters at each layer, the partial derivatives at each layer form a Jacobian matrix. Therefore, from the chain-rule for multivariate functions, if $f : \mathbb{R}^d \to \mathbb{R}^m$ and $g : \mathbb{R}^k \to \mathbb{R}^d$ then the Jacobian of the composition $(f \circ g) : \mathbb{R}^k \to \mathbb{R}^m$ at $\mathbf{x}$ is

$$J_{\mathbf{x}}(f \circ g) = J_{g(\mathbf{x})}(f) J_{\mathbf{x}}(g)$$

In the case of the neural network, which is the composition $(L_T \circ L_{T-1} \circ \ldots \circ L_1)(\mathbf{x})$, the Jacobian with respect to the parameters of layer $t-1 \in [T]$ is:

$$\begin{aligned} \frac{\partial}{\partial \mathbf{W}_{t-1}} (L_T \circ L_{T-1} \circ \ldots \circ L_1) &= J_{\mathbf{o}_{t-1}}(L_t) \cdot \prod_{i=T}^{t+1} J_{\mathbf{o}_{i-1}}(L_i) \\ &= J_{\mathbf{a}_t}(\mathbf{o}_t) \cdot J_{\mathbf{W}_{t-1}}(\mathbf{a}_t) \cdot \prod_{i=T}^{t+1} \left[J_{\mathbf{a}_i}(\mathbf{o}_i) \cdot J_{\mathbf{o}_{i-1}}(\mathbf{a}_i) \right] \end{aligned} \tag{11.5}$$

■ **Example 11.2** As an example, suppose we wish to train the following network where S is the Softmax function outputting the probability of being assigned one of three labels and L the Cross-Entropy function. For simplicity suppose all activation functions are the identity function.



So the function computed by the network computes is $f : \mathbb{R}^3 \to \mathbb{R}$ such that:

$$\mathbb{R}^4 \xrightarrow{f_1} \mathbb{R}^4 \xrightarrow{f_2} \mathbb{R}^4 \xrightarrow{\text{Softmax}} [0,1]^3 \xrightarrow{\text{Cross-Entropy}} \mathbb{R}$$

Now, the partial derivative of the network with respect to its parameters is:

$$\begin{aligned} \frac{\partial}{\partial \mathbf{W}_1} (L \circ S \circ L_2)(\mathbf{x}) &= J_S(L) \cdot J_{\mathbf{o}_2}(S) \cdot J_{\mathbf{a}_2}(\mathbf{o}_2) \cdot J_{\mathbf{W}_1}(\mathbf{a}_2) \\ \frac{\partial}{\partial \mathbf{W}_0} (L \circ S \circ L_2 \circ L_1)(\mathbf{x}) &= J_S(L) \cdot J_{\mathbf{o}_2}(S) \cdot J_{\mathbf{a}_2}(\mathbf{o}_2) \cdot J_{\mathbf{o}_1}(\mathbf{a}_2) \cdot J_{\mathbf{a}_1}(\mathbf{o}_1) \cdot J_{\mathbf{W}_0}(\mathbf{a}_1) \end{aligned}$$

Notice how the insights gained in the univariate case hold for the multivariate case:

- Expressions take one of two forms:
  - Derivatives of activation functions (e.g. $J_{\mathbf{a}_2}(\mathbf{o}_2)$).
  - Derivatives of inner products with respect to each of the inputs, which simply equals to the other input. e.g. $J_{\mathbf{W}_2}(\mathbf{a}_2) = \mathbf{o}_1$ or $J_{\mathbf{W}_0}(\mathbf{a}_1) = \mathbf{o}_0$.
- Expressions repeat across the derivation chains of different sets of parameters (e.g. $J_{\mathbf{o}_2}(S)$ or $J_{\mathbf{a}_2}(\mathbf{o}_2)$).

In the case of this specific network, the Jacobians of the Cross-Entropy, Softmax and identity function are:

$$J_S(L) = \nabla_S L^\top = \left[-\frac{\mathbf{y}}{\hat{\mathbf{y}}}\right]^\top, \quad J_{\mathbf{o}_2}(S) = diag(S) - SS^\top, \quad J_{\mathbf{a}_a}(\mathbf{o}_2) = I_4$$

The Jacobian of $J_{\mathbf{W}_2}(\mathbf{o}_a)$ is more complex. As $\mathbf{a}_2$, as a function of $\mathbf{W}_2$, is a function receiving a 4-by-4 matrix and outputting a vector in $\mathbb{R}^4$, the Jacobian is a *tensor* in $\mathbb{R}^{4\times(4\times4)}$. Looking at the i'th coordinate of the function's output then:

$$\mathbf{a}_2 = \begin{bmatrix} \vdots \\ [\mathbf{z}_2]_i \\ \vdots \end{bmatrix} = \begin{bmatrix} \vdots \\ \sum_{j=1}^5 [\mathbf{W}_2]_{ij} x_j \\ \vdots \end{bmatrix}$$

So the partial derivative of the of the t'th coordinate is:

$$\frac{\partial}{\partial [\mathbf{W}_2]_{ij}} [\mathbf{a}_2]_t = \begin{cases} x_j & i = t \\ 0 & i \neq t \end{cases} \quad \Rightarrow \quad \frac{\partial [\mathbf{a}_2]_t}{\partial \mathbf{W}_2} = \begin{bmatrix} 0 & \cdots\cdots & 0 \\ \vdots & & \vdots \\ x_1 & \cdots\cdots & x_4 \\ \vdots & & \vdots \\ 0 & \cdots\cdots & 0 \end{bmatrix} \overset{i=t}{\leftarrow}$$

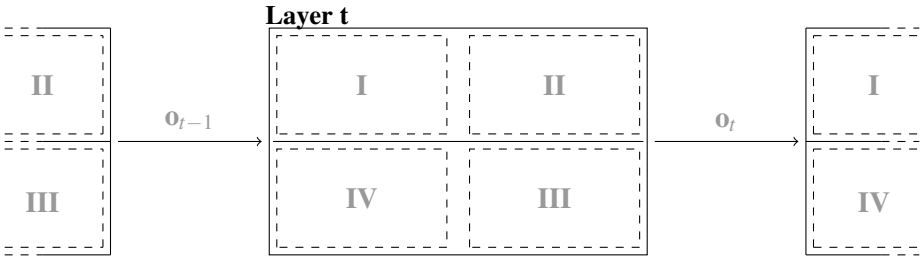
■

11.2.2 The Back-Propagation Algorithm

We now utilize the two insights above (i.e repeated use of neuron outputs and activation function derivatives) to derive an efficient algorithm for computing the gradient of the network with respect to *all* of the parameters. When doing so keep in mind the derivation chain

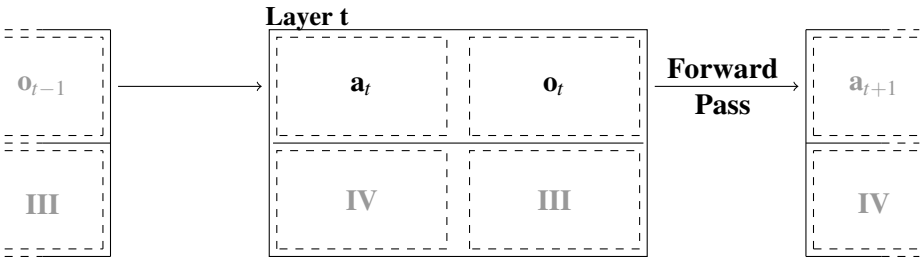
$$\frac{\partial}{\partial \mathbf{W}_{t-1}} (L_T \circ L_{T-1} \circ \ldots \circ L_1) = J_{\mathbf{a}_t}(\mathbf{o}_t) \cdot J_{\mathbf{W}_{t-1}}(\mathbf{a}_t) \cdot \prod_{i=T}^{t+1} \left[J_{\mathbf{a}_i}(\mathbf{o}_i) \cdot J_{\mathbf{o}_{i-1}}(\mathbf{a}_i) \right]$$

and consider the following representation of a layer:

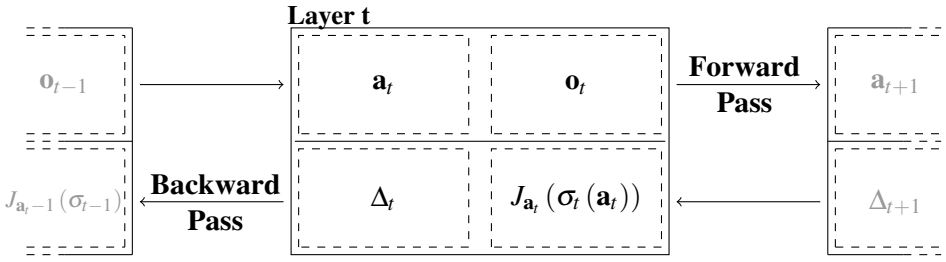


We begin with "feeding" the network a single sample and "push" it through the network graph calculating the pre-activations and activations. In each layer t, given the previous layer's output $\mathbf{o_{t-1}}$ we compute and *store* the pre-activation values $\mathbf{a}_t = \mathbf{W}_{t-1}\mathbf{o}_{t-1}$. Then we compute and store *store* the activation values $\mathbf{o}_t = \sigma_t(\mathbf{a}_{t-1})$. We denote this as the *forward pass* through the network graph.

Next, we begin from the output layer and define $\Delta_T \coloneqq \nabla_{\mathbf{o}_T} L$. If the loss function is for example the squared-loss then $Delta_t = \mathbf{o}_T - \mathbf{y}$ for $\mathbf{y}_i = y$ the response of the sample fed to the network. We then iterate from layer T backwards to layer 1 and compute the derivatives of the activation functions with respect to their input $[J_{\mathbf{a}_t}(\sigma_t(\mathbf{a}_t))]_i = \sigma_t'([\mathbf{a}_t]_i)$ for $i = 1, \ldots, k_t$. By defining the multivariate activation functions to be the



element-wise univariate activation function $\sigma_t(\mathbf{a}_t)_i = \sigma_t(a_i)$ and by selecting functions whose derivative can be computed efficiently (such as sigmoidal, tangent, ReLU and absolute value) this calculation can be done efficiently. We then *store* $\Delta_t \coloneqq J_{\mathbf{W}_{t-1}}(\mathbf{a}_t) \cdot J_{\mathbf{a}_t}(\mathbf{o}_t) \cdot \Delta_{t+1}$. We denote this as the *backward pass* through the network graph.



Finally we can conclude that the partial derivatives of the network with respect to each of its parameters are:

$$\begin{aligned} \frac{\partial}{\partial \mathbf{W}_{t-1}}(L_T \circ L_{T-1} \circ \ldots \circ L_t) &= J_{\mathbf{a}_t}(\mathbf{o}_t) \cdot J_{\mathbf{W}_{t-1}}(\mathbf{a}_t) \cdot \prod_{i=T}^{t+1} \left[J_{\mathbf{a}_i}(\mathbf{o}_i) \cdot J_{\mathbf{o}_{i-1}}(\mathbf{a}_i) \right] \\ &= J_{\mathbf{a}_t}(\sigma_t(\mathbf{a}_t)) \cdot \mathbf{o}_{t-1} \cdot \Delta_t \end{aligned} \tag{11.6}$$

all of which are calculated and stored throughout the forward- and backward passes. Put altogether, the back-propagation algorithm is as follows:

11.2.3 Tweaks & Tricks

The objective function of the neural network is highly complex and non-convex, and yet we use SGD to minimize it. As such we do not have a guarantee that we will converge into the global minimum of the function (which might not even be unique). Thus, there are many different tweaks and tricks which we can use when training the network and that will dramatically change the quality of the output model. Below we briefly discuss some of these options.

SGD With Momentum

The convergence rate of SGD might be slow dur to the high variance in SGD step directions. One way to adjust for this is to add *Nesterov's momentum* where in each iteration we take the place of the previous iteration and update it according to the vector of velocity. Now, the meaning of the gradient of this function is the acceleration.

$$\begin{aligned} \mathbf{v}^{(t+1)} &= \alpha \mathbf{v}^{(t)} - \eta_t \sum_{(\mathbf{x},y) \in B_t} \nabla L \left(\theta^{(t)} + \alpha \mathbf{v}^{(t)}; B_t \right) \\ \theta^{(t+1)} &= \theta^{(t)} + \mathbf{v}^{(t)} \end{aligned}$$

for $\theta^{(t)}$ the location at iteration t, $\mathbf{v}^{(t)}$ the velocity, α an hyper-parameter representing friction and the acceleration given by $\eta \nabla L$. Thus, the movement at each iteration is a linear combination of the previous movement and of current gradient.

Algorithm 19 Back-propagation

procedure BACK-PROPAGATION($\langle G, \{\mathbf{W}_t\}, \{\sigma_t\}, \phi \rangle$, $(\mathbf{x}, y)$)
  Denote $N := L_T \circ L_{T-1} \circ \ldots \circ L_1$
  FORWARD PASS
    Denote $\mathbf{o}_0 \leftarrow \mathbf{x}$
    for $t = 1, 2, \ldots, T$ **do**
      Compute pre-activations $\mathbf{a}_t \leftarrow \langle \mathbf{W}_{t-1}, \mathbf{o}_{t-1} \rangle$
      Compute activations $\mathbf{o}_t \leftarrow \sigma_t(\mathbf{a}_t)$
    end for

  BACKWARD PASS
    Set $\Delta_T = \phi'(\mathbf{o}_T)$
    for $t = T-1, T-2, \ldots, 1$ **do**
      Set derivation chain $\Delta_t \leftarrow \Delta_{t+1} \cdot J_{\mathbf{a}_t}(\mathbf{o}_t) \cdot \mathbf{W}_{t-1}$
      Set partial derivatives $\nabla_{\mathbf{W}_{t-1}} N \leftarrow \Delta_{t+1} \cdot J_{\mathbf{a}_t}(\mathbf{o}_t) \cdot \mathbf{o}_{t-1}$
    end for

  return ∇N
end procedure

Weight Initialization

Unlike in a convex function, the location at which we start the descent in a non-convex function is very important. Therefore there are many ways to initialize the network weights. We can initialize them using i.i.d draws of a random variable. We can initialize a sparse network where in each layer we only set a fixed few weights to be non zero.

Adaptive Learning Rate

As we have already seen we can devise different algorithms for choosing an adaptive learning rate. In the context of deep learning there exists a wide array of methods for choosing the step size (even individually for every entry of $\mathbf{w}$) such as AdaGrad, RMSProp and ADAM.

Regularization

Recall the use of regularization (such as ℓ_1 or ℓ_2) in the classical methods discussed earlier in the book. This was used to reduce the variance and improve the performance. Similarly, we can add a regularization term to the network's objective. For example, instead of minimizing the objective $L(\mathbf{w})$ we can add an ℓ_2 (Ridge) regularization to the weights and obtain the objective $L(\mathbf{w}) + \frac{\lambda}{2} ||\mathbf{w}||_2^2$. Then the gradient is $\nabla L(\mathbf{w}) + \lambda \mathbf{w}$ instead of $\nabla L(\mathbf{w})$.

Except the ideas above there are many other wild ideas that turn out to improve performance significantly such as attention, dropout, batch normalization, gradient clipping, pooling, data augmentation and more. Each of these refers to a family of ideas on how to improve the training process of a network.

11.3 Summary and Exercises

Exercises

Theoretical Questions

1. Find, in matrix notation, the Jacobian of the Softmax function $S(\mathbf{x})_i = e_i^x / \sum_{l=1}^k e^{x_l}$.
2. Calculate the Jacobian of the Cross-Entropy function $CE(\mathbf{y}, \mathbf{y}') = -\sum y_i \cdot \ln(y_i')$.

3. Calculate the sub-gradient of the ReLU function: $ReLU(x,a) = \max(x,a)$.
4. Let $f : \mathbb{R}^d \to \mathbb{R}^{k-1} \to \mathbb{R}^{k-2} \to \{0,1\}$ be a neural network with a single output neuron, two hidden layers (with k_1, l_2 neurons respectively) and with activation- and output functions being the indicator function $\sigma(x) = \mathbb{1}_{x \geq 0}$. Prove that functions learned by this network are of unions of k_2 polytops in $\mathbb{R}^d$, each with $k-1$ faces.

Appendices

A Linear Algebra

A.1 Norms & Inner Products

A *metric* (or distance function) is a function defined over an arbitrary set X that associates each pair of items in the set with a non-negative scalar quantity. Formally a function $d : X \times X \to \mathbb{R}$ is called a *metric* if and only if for all $x, y, z \in X$ the following properties hold:

- Identity of indiscernibles: $d(x,y) = 0 \iff x = y$
- Symmetry: $d(x,y) = d(y,x)$
- Triangle inequality: $d(x,z) \leq d(x,y) + d(y,z)$.

These conditions imply that a metric is a non-negative function returning values in $[0,\infty)$. Some common metric functions are the Euclidean distance, graph distance and string edit distance.

■ **Example .3** Consider the vector space $\mathbb{R}^d$. Let us show that the absolute distance, defined as the sum of absolute element-wise subtraction between the vectors $d(\mathbf{v},\mathbf{u}) := \sum_{i=1}^{d} |v_i - u_i|$, is a metric function.

- Identity of indiscernibles: Notice that from the properties of the absolute value over $\mathbb{R}$, for any two scalars $a, b \in \mathbb{R}$ it holds that $|a-b| = 0$ if and only if $a = b$. Therefore d, being a sum of non-negative elements, equals to zero if and only if all summed elements are zero. This takes place if and only if $\mathbf{v} = \mathbf{u}$. So $d(\mathbf{v},\mathbf{u}) = 0 \iff \mathbf{v} = \mathbf{u}$.
- Symmetry: The symmetry of d is achieved through symmetry of the absolute value function.
- Triangle inequality: Let $\mathbf{v}, \mathbf{u}, \mathbf{w} \in \mathbb{R}^d$. Then:

$$d(\mathbf{v},\mathbf{u}) = \sum |v_i - u_i| = \sum |v_i - w_i + w_i - u_i| \leq \sum |v_i - w_i| + \sum |w_i - u_i| = d(\mathbf{v},\mathbf{w}) + d(\mathbf{w},\mathbf{u})$$

■

A *norm* on vector space $\mathbb{R}^d$ is a function $||\cdot|| : \mathbb{R}^d \to \mathbb{R}_+$ that satisfies that for all $\alpha \in \mathbb{R}$ and for all $\mathbf{v}, \mathbf{u} \in \mathbb{R}^d$ the following properties hold:

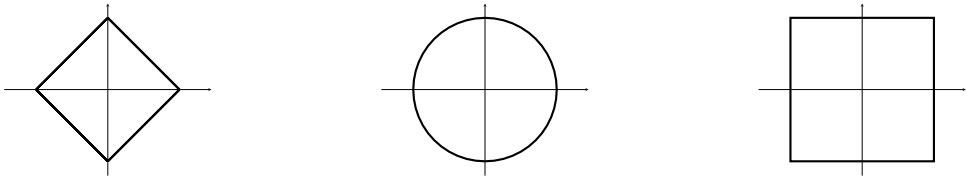
- Positive definiteness: $||\mathbf{v}|| \geq 0$, and $||\mathbf{v}|| = 0$ if and only if $\mathbf{v}$ is the zero vector
- Absolute homogeneity: $||\alpha\mathbf{v}|| = |\alpha| \cdot ||\mathbf{v}||$
- Triangle inequality: $||\mathbf{v} + \mathbf{u}|| \leq ||\mathbf{v}|| + ||\mathbf{u}||$

It is helpful to think of a norm as the distance of a vector from the origin. A few commonly used norms are:

- Absolute norm (ℓ_1): $||\mathbf{v}||_1 \coloneqq \sum |v_i|$.
- Euclidean norm (ℓ_2): $||\mathbf{v}||_2 \coloneqq \sqrt{\sum v_i^2}$.
- Infinity norm (ℓ_∞): $||\mathbf{v}||_\infty \coloneqq \max_i |v_i|$.

R These norms are part of a wider family of norms called the L_p norms, defined as $||\mathbf{v}||_p \coloneqq \left(\sum |v_i^p|\right)^{1/p}$ for $1 \leq p \leq \infty$. The infinity norm is obtained by taking the limit as $p \to \infty$.

Given a norm on a vector space, i.e. a normed vector space $(V, ||\cdot||)$, we specify the *unit ball* of $||\cdot||$ as the set of vectors such that: $B_{||\cdot||} = \{\mathbf{v} \in V : ||\mathbf{v}|| \leq 1\}$. The use of different norms, and thus different shapes of their unit ball, influence the outcome of optimization algorithms using them (subsection 6.0.2).



(a) $||\cdot||_1$ ***(b)*** $||\cdot||_2$ ***(c)*** $||\cdot||_\infty$

Figure 8: *Unit balls of norms on* $\mathbb{R}^2$

Similar to a metric, an *inner product* provides a scalar quantity associated with two given vectors. An inner product space is a vector space V over $\mathbb{R}$ together with a map $\langle \cdot, \cdot \rangle : V \times V \to \mathbb{R}$, satisfying that $\forall \mathbf{v}, \mathbf{u}, \mathbf{w} \in V$, $\alpha, \beta \in \mathbb{R}$ the following properties hold:

- Symmetry: $\langle \mathbf{v}, \mathbf{u} \rangle = \langle \mathbf{u}, \mathbf{v} \rangle$
- Linearity: $\langle \alpha \mathbf{v} + \beta \mathbf{w}, \mathbf{u} \rangle = \alpha \langle \mathbf{v}, \mathbf{u} \rangle + \beta \langle \mathbf{w}, \mathbf{u} \rangle$
- Non-negativity : $\langle \mathbf{v}, \mathbf{v} \rangle \geq 0$, and $\langle \mathbf{v}, \mathbf{v} \rangle = 0 \iff \mathbf{v} = \mathbf{0}$

An example for an inner product is the *standard inner product* $\langle \mathbf{v}, \mathbf{u} \rangle \coloneqq \sum_i v_i u_i$. These definitions of a norm and an inner product are very similar. In fact, given an inner product space, we are also given a norm over this space. Given an inner product space H, the function $||\cdot|| : H \to \mathbb{R}_+$ that is defined by $||\mathbf{v}|| = \langle \mathbf{v}, \mathbf{v} \rangle^{\frac{1}{2}}$ for all $\mathbf{v} \in H$ is a norm on H. It is called the *induced norm*.

Using inner products we can further formulate other intuitive geometrical notions such as describing the angle between two as $\cos\theta = \langle \mathbf{v}, \mathbf{u} \rangle / ||\mathbf{v}|| \cdot ||\mathbf{u}||$. Consider the norm of the vector $\mathbf{v} - \mathbf{u}$. Being the induced norm of some inner product then:

$$||\mathbf{v} - \mathbf{u}||^2 = \langle \mathbf{v} - \mathbf{u}, \mathbf{v} - \mathbf{u} \rangle = \langle \mathbf{v}, \mathbf{v} \rangle - 2\langle \mathbf{v}, \mathbf{u} \rangle + \langle \mathbf{u}, \mathbf{u} \rangle = ||\mathbf{v}||^2 + ||\mathbf{u}||^2 - 2\langle \mathbf{v}, \mathbf{u} \rangle$$

On the other hand, by the Law of Cosines for the triangle defined by $\mathbf{v}, \mathbf{u}, \mathbf{v} - \mathbf{u}$ then

$$||\mathbf{v} - \mathbf{u}||^2 = ||\mathbf{v}||^2 + ||\mathbf{u}||^2 - 2||\mathbf{v}|| \cdot ||\mathbf{u}|| \cdot \cos\theta$$

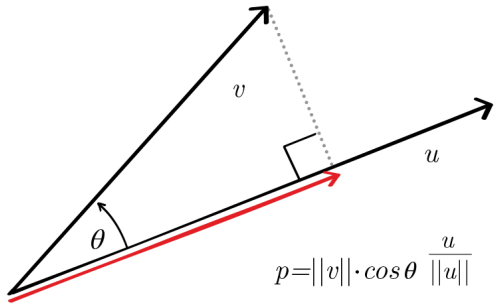
Taken together we obtain that:

$$\langle \mathbf{v}, \mathbf{u} \rangle = ||\mathbf{v}|| \cdot ||\mathbf{u}|| \cdot \cos\theta \quad \Rightarrow \quad \cos\theta = \langle \mathbf{v}, \mathbf{u} \rangle / ||\mathbf{v}|| \cdot ||\mathbf{u}||$$

A special case if when the angle between two vectors is of 90° for which the result of the inner product between the vectors equals to zero: $\theta = 90^\circ \iff \langle \mathbf{v}, \mathbf{u} \rangle = 0$. In this case we say the vectors are *orthogonal* to each other and denote it as $\mathbf{v} \perp \mathbf{u}$.

Another geometric notion is that of *vector projection*. The vector projection of $\mathbf{v}$ onto $\mathbf{u}$ is the orthogonal projection of $\mathbf{v}$ onto a line parallel to the vector $\mathbf{u}$. It is defined as $\mathbf{p} := p \cdot \hat{\mathbf{u}}$ for $p := \langle \mathbf{v}, \hat{\mathbf{u}} \rangle \hat{\mathbf{u}}$ and $\hat{\mathbf{u}} := \mathbf{u}/||\mathbf{u}||$. p is called the *scalar projection* of $\mathbf{v}$ onto $\mathbf{u}$, and $\hat{\mathbf{u}}$ is the unit vector in the direction of $\mathbf{u}$. The vector $\mathbf{v} - \mathbf{p}$ that is perpendicular to $\mathbf{u}$ and completes a right-angle triangle is called the *vector rejection* of $\mathbf{v}$ from $\mathbf{u}$. Using the angle θ between the two vectors, we can write the vector projection as:

$$\mathbf{p} = \langle \mathbf{v}, \hat{\mathbf{u}} \rangle \hat{\mathbf{u}} = \langle \mathbf{v}, \mathbf{u} \rangle \cdot \frac{\mathbf{u}}{||\mathbf{u}||^2} = ||\mathbf{v}|| \, ||\mathbf{u}|| \cos\theta \cdot \frac{\mathbf{u}}{||\mathbf{u}||^2} = ||\mathbf{v}|| \cos\theta \cdot \hat{\mathbf{u}}$$



A.2 Matrices of Linear Transformations

Given two vector spaces V and W over a field $\mathbb{F}$, a *linear transformation* $T : V \to W$ is a function satisfying that $\forall \mathbf{v}, \mathbf{u} \in V$ and $\forall \alpha \in \mathbb{F}$:

- Additivity: $T(\mathbf{u} + \mathbf{v}) = T(\mathbf{u}) + T(\mathbf{u})$
- Scalar multiplication: $T(\alpha \mathbf{v}) = \alpha \cdot T(\mathbf{v})$

In the case where V, W are of finite dimensions d and m, any linear map $T : V \to W$ is of the form $T(\mathbf{v}) = A\mathbf{v}$ for $\mathbf{v} \in V$ and some matrix $A \in \mathbb{R}^{m \times d}$. The matrix A is called the *representing matrix* of T. Extending linear transformations, an *affine transformation* is a transformation of the form $T(\mathbf{v}) = A\mathbf{v} + \mathbf{w}$ for $\mathbf{v} \in V, \mathbf{w} \in W$ and A the matrix associated with a linear transformation from V to W. Notice that an affine transformation is not a linear transformation as it does not map $\mathbf{0}_V$ to $\mathbf{0}_W$.

Using the representing matrix A of a linear transformation $T : V \to W$ we define four *fundamental subspaces*:

- Kernel- (or null-) space of A as $Ker(A) := \{\mathbf{v} \in V | A\mathbf{v} = 0\}$. Also denotes as $\mathcal{N}(A)$.
- Image- (or column/range-) space of A as $Im(A) := \{\mathbf{w} \in W | \mathbf{w} = A\mathbf{v}, \mathbf{v} \in V\}$. Also denotes as $Col(A)$ or $\mathcal{R}(A)$.
- Row space of A as $Im\left(A^\top\right) := \left\{\mathbf{v} \in V | \mathbf{v} = A^\top \mathbf{w}, \mathbf{w} \in W\right\}$. Equivalently it can be defined as the column space of $A^\top$ and therefore denoted as $Col\left(A^\top\right)$.
- Null space of $A^\top$ as $Ker\left(A^\top\right) := \left\{\mathbf{w} \in W | A^\top \mathbf{w} = 0\right\}$. This space is also referred to as the left null space of A.

Note that by definition, $Ker(A), Row(A) \subseteq V$ and $Im(A) \subseteq W$. Another quantity associated with matrices is the *rank* of a matrix. For $A \in \mathbb{R}^{m \times d}$ the rank of A is the maximum number of linearly independent rows of A and denoted by $rank(A)$. It holds that the rank of A equals to both the dimension of the columns space and of the row space of A. As such, we refer to A being of *full rank* if and only if $rank(A) = \min\{m, d\}$. Otherwise we say that A is *rank deficient*.

For the case of a square matrix we define the notion of invertability. For $A \in \mathbb{R}^{d \times d}$ a square matrix, we say

that A is invertible (or non-singular) if there exists a matrix $B \in \mathbb{R}^{d\times d}$ such that $AB = I_d = BA$. We denote the inverse by A^{-1}. Then, the following are equivalent:

- A is invertible (non-singular)
- A is full-rank
- $Det(A) \neq 0$
- $Im(A) = \mathbb{R}^d$ (i.e. the image is the whole space)
- $ker(A) = \vec{0}$ (i.e. the kernel is trivial)

A.2.1 Orthogonal Projection Matrices

We can extend the notion of vector projections to projecting a given vector onto a subspace of arbitrary dimension.

Definition A.1 Let V be a k-dimensional subspace of $\mathbb{R}^d$, and let $\mathbf{v}_1, \ldots, \mathbf{v}_k$ be an orthonormal basis of V. The matrix $P := \sum_{i=1}^k \mathbf{v}_i \otimes \mathbf{v}_i = \sum_{i=1}^k \mathbf{v}_i \mathbf{v}_i^\top$ is an *orthogonal projection matrix* onto the subspace V.

where the operation performed on the vectors is the *outer product*. For two vectors $\mathbf{v} \in \mathbb{R}^n, \mathbf{u} \in \mathbb{R}^m$, the outer product of $\mathbf{v}$ and $\mathbf{u}$, which is denoted by $\mathbf{v} \otimes \mathbf{u}$ or $\mathbf{v}\mathbf{u}^\top$ is an $n \times m$ matrix with entries:

$$[\mathbf{v} \otimes \mathbf{u}]_{ij} = v_i \cdot u_j, \quad \mathbf{v} \otimes \mathbf{u} = \begin{bmatrix} v_1u_1 & v_1u_2 & \cdots & v_1u_m \\ \vdots & \vdots & \ddots & \vdots \\ v_nu_1 & v_nu_2 & \cdots & v_nu_m \end{bmatrix}$$

Of note, an outer product of two non-zero vectors yields a matrix of rank 1. We therefore express P as the sum of k rank 1 matrices each being a one-dimensional orthogonal projection matrix. Another way to write P is as $P = AA^\top$ where the columns of A are $\mathbf{v}_1, \ldots, \mathbf{v}_k$ an orthonormal basis of V. The following lemma summarizes some useful properties of orthogonal projection matrices.

Lemma A.1 Let P be an orthogonal projection matrix, then P has the following properties:

- P is symmetric
- $P^2 = P$
- The eigenvalues of P are either 0 or 1. $\mathbf{v}_1, \ldots, \mathbf{v}_k$ are the eigenvectors of P which correspond to the eigenvalue 1.
- $(I - P)P = 0$
- $\forall \mathbf{x} \in \mathbb{R}^d$ and $\forall \mathbf{v} \in V$, $||\mathbf{x} - \mathbf{v}|| \geq ||\mathbf{x} - P\mathbf{x}||$
- $\mathbf{v} \in V \Rightarrow P\mathbf{v} = \mathbf{v}$

A.2.2 Positive (Semi-) Definiteness

For square symmetric matrices we define the notion of definiteness and semi-definiteness.

Definition A.2 Let $A \in \mathbb{R}^{d\times d}$ be a symmetric matrix. A is a *positive semi-definite* (PSD) matrix if and only if

$$\forall \mathbf{x} \in \mathbb{R}^d \quad \mathbf{x}^\top A\mathbf{x} \geq 0$$

and denote so by $A \succcurlyeq 0$. Further, A is *positive definite* (PD) if and only if

$$\forall \mathbf{x} \in \mathbb{R}^d \quad \mathbf{x}^\top A\mathbf{x} > 0$$

and denote so by $A \succ 0$.

In addition, for a symmetric matrix A the following are equivalent:

- A is a PSD matrix
- For all $\mathbf{x} \in \mathbb{R}^d$ then $\mathbf{x}^\top A\mathbf{x} \geq 0$
- For λ an eigenvalue of A then $\lambda \geq 0$

- A can be written as the product $A = B^\top B$ for some matrix $B \in \mathbb{R}^{k\times d}$

Similarly, these conditions can be defined for PD matrices restricting that $\mathbf{x}^\top A\mathbf{x} > 0$, strictly positive eigenvalues and $A = B^*B$ for B^* being the conjugate transpose of B. There are many useful properties for PSD matrices such as that for $\alpha \geq 0$ also αA is a PSD matrix; the sum of PSD matrices is a PSD matrix; and for M, N two PSD matrices also the products MNM and NMN are PSD matrices.

A.3 Matrix Factorizations

Matrix factorization/decomposition is a strong tool with many theoretical as well as practical usages. The core idea is to find a representation of a given matrix as the product of several different matrices which have certain desired properties. For example, in the case of a PSD matrix $A \in \mathbb{R}^{d\times d}$ we have seen that there exists a matrix $B \in \mathbb{R}^{k\times d}$ such that $A = B^\top B$. Here we were able to decompose A into the product of the matrices $B^\top$ and B. We could now try and find if there exists a matrix B with some property. For example, if B is a lower triangular matrix, i.e. $B_{ij} = 0\ \forall j > i$, it is called the *Cholesky decomposition*.

A.3.1 Eigenvalue Decomposition

Let A be a square matrix. We say that a vector $\mathbf{0} \neq \mathbf{v} \in \mathbb{R}^d$ is an *eigenvector* of A corresponding to an *eigenvalue* $\lambda \in \mathbb{R}$ if and only if $A\mathbf{v} = \lambda\mathbf{v}$. To find the eigenvectors and eigenvalues of A we therefore wish to solve the linear system of $(A - \lambda I)\,\mathbf{v} = \mathbf{0}$. This system has a non-zero solution if and only if $\det(A - \lambda I) = 0$. These solutions if exist, are the roots of the *characteristic polynomial* of A: $p_A(\lambda) := |A - \lambda I|$. From the fundamental theorem of algebra we learn that the characteristic polynomial can be factored as the product

$$|A - \lambda I| = (\lambda_1 - \lambda) \cdot \ldots \cdot (\lambda_d - \lambda)$$

for λ_i the roots of the characteristic polynomial and the eigenvalues of A. Given λ and eigenvalue of A we define the set of eigenvectors corresponding to λ as $E_\lambda := \{\mathbf{v} \mid (A - \lambda I)\,\mathbf{v} = \mathbf{0}\}$. This linear subspace is the *eigenspace* of A associated with the eigenvalue λ. The dimension of E is termed the *geometric multiplicity* of λ.

The idea of matrix decomposition relates to that of *matrix diagonalization*. For a square matrix $A \in \mathbb{R}^{d\times d}$, we say that A is diagonalizable if there exists an invertible matrix P such that $P^{-1}AP$ is diagonal. In the case of a symmetric matrix, the Eigenvalue Decomposition (EVD) provides a diagonalization of a matrix using its eigenvalues and eigenvectors.

Theorem A.2 Let $A \in \mathbb{R}^{d\times d}$ be a real symmetric matrix. Then, there exist an orthogonal matrix $U \in \mathbb{R}^{d\times d}$ and a diagonal matrix $D \in \mathbb{R}^{d\times d}$ such that $A = UDU^\top$. Furthermore, it holds that

- The diagonal entries of D are the eigenvalues of A (with their multiplicities).
- The columns of U are eigenvectors for A corresponding to the eigenvalues on the diagonal of D. These form an orthonormal basis of $\mathbb{R}^d$.

This decomposition is called the *Eigenvalue Decomposition* (EVD) or the *Spectral Decomposition* of A.

Namely, $A\mathbf{u}_i = D_{ii}\mathbf{u}_i,\ i \in [d]$. Without loss of generality, we arrange the elements of D (and respectively, the columns of U) such that the eigenvalues are in decreasing order $D_{ii} \geq D_{i+1,i+1}$.

■ **Example .4** Consider the following symmetric matrix

$$A = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

To find the EVD of A we solve its eigenvalue problem by finding λ such that $\det(A - \lambda I) = 0$, yielding the equation $(2-\lambda)^2 - 1 = 0$ with the solutions $\lambda = 3, 1$. Now, to find the eigenvectors we search for $\mathbf{v} \neq 0$ such that $(A - \lambda I)\mathbf{v} = 0$ for $\lambda = 3, 1$. We find that the eigenvector corresponding to eigenvalues $\lambda = 3$ is $\mathbf{v}_3 = (1,1)^\top$ and the eigenvector corresponding to eigenvalue $\lambda = 1$ is $\mathbf{v}_1 = (1,-1)^\top$. We normalize the

eigenvectors and obtain the EVD of A:

$$A = UDU^\top \quad U = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix}, D = \begin{bmatrix} 3 & 0 \\ 0 & 1 \end{bmatrix}$$

By multiplying $UDU^\top$ we validate that indeed we achieve the matrix A and the correctness of the decomposition. ∎

To gain a deeper understanding into the EVD, consider the linear transformations represented by the matrix A and the matrices U and D. It holds that for a square matrix, the eigenspace corresponding to eigenvalue zero spans the kernel space of A, while the eigenspace corresponding to non-zero eigenvalues spans the range of A. Thus, for $rank(A) = k \leq d$:

$$\mathcal{R}(A) = \text{span}\{\mathbf{u}_1, \ldots, \mathbf{u}_k\}, \quad \mathcal{N}(A) = \text{span}\{\mathbf{u}_{k+1}, \ldots, \mathbf{u}_d\}$$

As these vectors are orthogonal to each other, we have an orthonormal basis of eigenvectors of A to $\mathbb{R}^d$. Now consider a unit vector $\mathbf{x} \in \mathbb{R}^d$, $||\mathbf{x}|| = 1$. The image of $\mathbf{x}$ under A is

$$A\mathbf{x} = (UDU^\top)\mathbf{x} = UD \begin{bmatrix} \langle \mathbf{x}, \mathbf{u}_1 \rangle \\ \vdots \\ \langle \mathbf{x}, \mathbf{u}_d \rangle \end{bmatrix} = U \begin{bmatrix} \lambda_1 \langle \mathbf{x}, \mathbf{u}_1 \rangle \\ \vdots \\ \lambda_d \langle \mathbf{x}, \mathbf{u}_d \rangle \end{bmatrix} = \sum_{i=1}^{d} \lambda_i \langle \mathbf{x}, \mathbf{u}_i \rangle \mathbf{u}_i$$

Namely, A provides a representation of $\mathbf{x}$ in terms of the orthonormal basis $\mathbf{u}_1, \ldots, \mathbf{u}_d$, and dialites or contracts components (i.e directions in space) according to the magnitude of the eigenvalues. If the matrix is not of full rank, in some of the directions (those corresponding to the null space) we are left with the zero vector.

The spectral decomposition has many useful properties. Just by observing the decomposition itself we are able to "read" the spectrum of A - the unique values along the diagonal of D - with their algebraic multiplicity, and the range- and null-spaces of A. In addition, the determinant and trace of A are obtained from D by:

$$det(A) = \prod_i D_{ii}, \quad tr(A) = \sum_i D_{ii}$$

Furthermore, taking A to the power is simply done by raising each of the eigenvalues to that power. And in the case of an invertible matrix A, the inverse is simply

$$A^{-1} = (UDU^\top)^{-1} = UD^{-1}U^\top, \quad D_{ii}^{-1} = (D_{ii})^{-1}$$

A.3.2 Singular-Values Decomposition

The *Singular-Values Decomposition* is a decomposition related to the EVD for an arbitrary real matrix $A \in \mathbb{R}^{m \times d}$. We say that $\mathbf{u} \in \mathbb{R}^m$ and $\mathbf{v} \in \mathbb{R}^d$ are left and right singular vectors of A, corresponding to singular value $\sigma \in \mathbb{R}_+$ if and only if $A\mathbf{v} = \sigma\mathbf{u}$.

Theorem A.3 Let $A \in \mathbb{R}^{m \times d}$ be a real matrix. Then, there exists:

- An orthogonal matrix $U \in \mathbb{R}^{m \times m}$ whose columns are the left singular vectors of A.
- An othrogonal matrix $V \in \mathbb{R}^{d \times d}$ whose columns are the right singular vectors of A.
- A diagonal matrix $\Sigma \in \mathbb{R}^{m \times d}$ whose diagonal entries are the singular values of A such that $A\mathbf{v}_i = \Sigma_{ii}\mathbf{u}_i$.

$$A = U\Sigma V^\top = \overbrace{\begin{bmatrix} | & & | \\ \mathbf{u}_1 & \cdots & \mathbf{u}_m \\ | & & | \end{bmatrix}}^{m \times m} \overbrace{\begin{bmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_d \\ \hline & \mathbf{0}_{(m-d) \times d} & \end{bmatrix}}^{m \times d} \overbrace{\begin{bmatrix} - & \mathbf{v}_1^\top & - \\ & \vdots & \\ - & \mathbf{v}_d^\top & - \end{bmatrix}}^{d \times d}$$

This is called the *Singular-Value Decomposition* of A.

Without loss of generality, we arrange the elements of Σ (and respectively, the columns of U, V) such that the singular values are in decreasing order $\Sigma_{ii} \geq \Sigma_{i+1,i+1}$.

Geometric Interpretation

As we have done for the EVD, let us consider the linear transformations represented by A. In this case the transformation takes $\mathbb{R}^d$ to a different space $\mathbb{R}^m$. When represented using the decomposition, similar to the EVD, the nature of the transformation becomes clear. For V, U whose columns are orthonormal basis of the domain $\mathbb{R}^d$ and range $\mathbb{R}^m$ of A it can be shown that Ex.7

$$\begin{aligned} \mathcal{R}(A) &= \text{span}\{\mathbf{u}_1, \ldots, \mathbf{u}_k\}, \quad \mathcal{N}(A^\top) = \text{span}\{\mathbf{u}_{k+1}, \ldots, \mathbf{u}_m\} \\ \mathcal{R}(A^\top) &= \text{span}\{\mathbf{v}_1, \ldots, \mathbf{v}_k\}, \quad \mathcal{N}(A) = \text{span}\{\mathbf{v}_{k+1}, \ldots, \mathbf{v}_d\} \end{aligned}$$

where $rank(A) = k \leq \min\{m, d\}$. Then, for a general element $\mathbf{x} \in \mathbb{R}^d$, represented in this basis $\mathbf{x} = \sum_i \langle \mathbf{x}, \mathbf{v}_i \rangle \mathbf{v}_i$, we see that:

$$A\mathbf{x} = A \sum_i \langle \mathbf{x}, \mathbf{v}_i \rangle \mathbf{v}_i = \sum_i \langle \mathbf{x}, \mathbf{v}_i \rangle U\Sigma V^\top \mathbf{v}_i = \sum_i \langle \mathbf{x}, \mathbf{v}_i \rangle U\Sigma \mathbf{e}_i = \sum_i \sigma_i \langle \mathbf{x}, \mathbf{v}_i \rangle U\mathbf{e}_i = \sum_i \sigma_i \langle \mathbf{x}, \mathbf{v}_i \rangle \mathbf{u}_i$$

That is, the SVD simply scales (expands or contracts) some of the components according to the magnitude of the singular values, and discards components corresponding to the null-spaces. Now, to understand the manner in which A deforms the space consider its action on the unit sphere in $\mathbb{R}^d$. Suppose $\mathbf{x}$ is on the unit sphere, i.e $\mathbf{x} = x_1\mathbf{v}_1 + \ldots + x_d\mathbf{v}_d$, $||\mathbf{x}||_2^2 = \sum x_i^2 = 1$. The image of the unit sphere under A is therefore $A\mathbf{x} = y_1\mathbf{u}_1 + \ldots + y_k\mathbf{u}_k$ for $y_i := \sigma_i x_i$ where

$$||A\mathbf{x}||_2^2 = \frac{y_1^2}{\sigma_1^2} + \ldots + \frac{y_k^2}{\sigma_k^2} = \sum_{i=1}^{k} x_i^2 \leq 1$$

Namely, A maps the unit k-sphere in $\mathbb{R}^d$ into an ellipsoid with axes in the directions of $\mathbf{u}_i$ and with magnitudes σ_i (collapsing $d-k$ dimensions of the domain) and then embeds the ellipsoid in $\mathbb{R}^m$.

The connection between SVD and EVD

The natural question at this point is how to choose the basis $\{\mathbf{v}_1, \ldots, \mathbf{v}_d\}$ and $\{\mathbf{u}_1, \ldots, \mathbf{u}_m\}$. Since $A\mathbf{v}_i = \sigma_i\mathbf{u}_i$ it is simple to obtain the diagonal representation of A. Let $\{\mathbf{v}_1, \ldots, \mathbf{v}_d\}$ be some orthonormal basis of $\mathbb{R}^d$ such that the first k elements span the row space of A while the remaining $d-k$ elements span the null space of A. We can now obtain $\mathbf{u}_i$ as a unit vector parallel to $A\mathbf{v}_i$, and extend this to a basis of $\mathbb{R}^m$. Relative to these basis we have achieved a diagonalization of A:

$$U^\top AV = U^\top U\Sigma V^\top V = \Sigma \tag{7}$$

In general however, even for orthonormal v's there is no guarantee for orthogonality to be preserved under A. Therefore the key point is in finding the vectors $\mathbf{v}_1, \ldots, \mathbf{v}_k$ for which $\mathbf{u}_1, \ldots, \mathbf{u}_k$ are orthonormal. We find such a basis using the EVD of the $d \times d$ symmetric matrix $A^\top A$. Let $A^\top A = VDV^\top$ be the EVD of $A^\top A$, where the columns of V, $\{\mathbf{v}_1, \ldots, \mathbf{v}_d\}$, (i.e the eigenvectors of $A^\top A$) are an orthonormal basis of $\mathbb{R}^d$. Then

$$\langle A\mathbf{v}_i, A\mathbf{v}_j \rangle = (A\mathbf{v}_i)^\top (A\mathbf{v}_j) = \mathbf{v}_i^\top \left(A^\top A\mathbf{v}_j\right) = \mathbf{v}_i^\top (\lambda_j\mathbf{v}_j) = \lambda_j\mathbf{v}_i^\top\mathbf{v}_j = 0$$

Thus, the image set of applying A over the basis of eigenvectors of $A^\top A$, $\{A\mathbf{v}_1, \ldots, A\mathbf{v}_d\}$, is orthogonal with the nonzero vectors forming a basis for the range of A. Namely, the images under A of the eigenvectors of $A^\top A$ provide an orthogonal bases allowing the diagonalization of A as seen in (7). By normalizing the non-zero vectors we obtain $\{\mathbf{u}_1, \ldots, \mathbf{u}_k\}$

$$\mathbf{u}_i := \frac{A\mathbf{v}_i}{||A\mathbf{v}_i||} = \frac{A\mathbf{v}_i}{\sqrt{\lambda_i\mathbf{v}_i^\top\mathbf{v}_i}} = \frac{1}{\sqrt{\lambda_i}} A\mathbf{v}_i, \quad i \in [k]$$

completing the construction of the orthonormal bases for $\mathbb{R}^d$ and $\mathbb{R}^m$. By setting $\sigma_i = \sqrt{\lambda_i}$ we also achieve that $A\mathbf{v}_i = \sigma_i \mathbf{u}_i$ or in matrix notation $A = U\Sigma V^\top$.

Conversely, given the SVD of $A = U\Sigma V^\top$ we can recover the EVD of $A^\top A$ or of $AA^\top$:

$$A^\top A = \left(U\Sigma V^\top\right)^\top \left(U\Sigma V^\top\right) = V\Sigma^\top U^\top U\Sigma V^\top = V\Sigma^\top \Sigma V^\top$$
$$AA^\top = \left(U\Sigma V^\top\right) \left(U\Sigma V^\top\right)^\top = U\Sigma V^\top V\Sigma^\top U^\top = U\Sigma\Sigma^\top U^\top$$

where $\Sigma^\top\Sigma$ and $\Sigma\Sigma^\top$ are both square matrices whose first k diagonal entries are σ_i^2. We therefore conclude that the left singular vectors of A are the eigenvectors of $AA^\top$; the right singular vectors of A are the eigenvectors of $A^\top A$; and the singular values of A are the square root of the eigenvalues of $AA^\top$ and $A^\top A$. Furthermore, it can be shown that up to orthogonal transformations of $A^\top A$ and $AA^\top$ the SVD of A is uniquely determined.

Compact SVD Form

When writing the SVD of a matrix A, with rank $rank(A) = k < min\{m,d\}$, it becomes evident that we can express the SVD in a more compact way. Without loss of generality, suppose $d \leq m$. Notice that Σ consists of zero rows and columns as $\sigma_{k+1}, \ldots, \sigma_d$ are all zero.

$$A = \left[\begin{array}{ccc|ccc} \mathbf{u}_1 & \cdots & \mathbf{u}_k & \mathbf{u}_{k+1} & \cdots & \mathbf{u}_m \end{array}\right] \left[\begin{array}{ccc|c} \sigma_1 & & & \\ & \ddots & & \mathbf{0} \\ & & \sigma_k & \\ \hline & \mathbf{0} & & \mathbf{0} \end{array}\right] \left[\begin{array}{c} \mathbf{v}_1^\top \\ \vdots \\ \mathbf{v}_k^\top \\ \hline \mathbf{v}_{k+1}^\top \\ \vdots \\ \mathbf{v}_d^\top \end{array}\right]$$

When we partition the multiplication as follows it becomes evident that the left and right singular vectors $\mathbf{u}_{k+1}, \ldots, \mathbf{u}_m$ and $\mathbf{v}_{k+1}, \ldots, \mathbf{v}_d$ do not make any contribution to A. Their purpose is to expand the leading left and right singular vectors into orthonormal bases of $\mathbb{R}^m$ and $\mathbb{R}^d$ respectively.

$$A = \left[\begin{array}{ccc} \mathbf{u}_1 & \cdots & \mathbf{u}_k \end{array}\right] \left[\begin{array}{ccc} \sigma_1 & & \\ & \ddots & \\ & & \sigma_k \end{array}\right] \left[\begin{array}{c} \mathbf{v}_1^\top \\ \vdots \\ \mathbf{v}_k^\top \end{array}\right] + \left[\begin{array}{ccc} \mathbf{u}_{k+1} & \cdots & \mathbf{u}_m \end{array}\right] \left[\begin{array}{c} \mathbf{0} \end{array}\right] \left[\begin{array}{c} \mathbf{v}_{k+1}^\top \\ \vdots \\ \mathbf{v}_d^\top \end{array}\right]$$

Thus, we eliminate left and right singular vectors corresponding to zero singular values to obtain the *Compact SVD* form.

$$A = \overbrace{\left[\begin{array}{ccc} | & & | \\ \mathbf{u}_1 & \cdots & \mathbf{u}_k \\ | & & | \end{array}\right]}^{m\times k} \overbrace{\left[\begin{array}{ccc} \sigma_1 & & \\ & \ddots & \\ & & \sigma_k \end{array}\right]}^{k\times k} \overbrace{\left[\begin{array}{ccc} - & \mathbf{v}_1^\top & - \\ & \vdots & \\ - & \mathbf{v}_k^\top & - \end{array}\right]}^{k\times d} \tag{8}$$

A.4 Exercises

Theoretical Questions

1. Let $\mathbf{v}_1, \ldots, \mathbf{v}_k \in V \subset \mathbb{R}^d$ be an orthonormal basis of V. Let A be the matrix whose columns are $\mathbf{v}_1, \ldots, \mathbf{v}_k$. Prove that $\sum \mathbf{v}_i \mathbf{v}_i^\top = AA^\top$.

2. Let $V \subset \mathbb{R}^d$ be a vector space of dimension k and P an orthogonal projection matrix onto V. Prove the properties of an orthogonal projection matrix described in Lemma A.1.

3. Let $A \in \mathbb{R}^{d\times d}$ be a square matrix of rank $k \leq d$. Let $\mathbf{u}_1,\ldots,\mathbf{u}_k$ be eigenvectors of A corresponding to non-zero eigenvalues and $\mathbf{u}_{k+1},\ldots,\mathbf{u}_d$ be eigenvectors of A corresponding to eigenvalue zero. Show that $\mathcal{R}(A) = \text{span}\{\mathbf{u}_1,\ldots,\mathbf{u}_k\}$ and that $\mathcal{N}(A) = \text{span}\{\mathbf{u}_{k+1},\ldots,\mathbf{u}_d\}$.

4. Let $A \in \mathbb{R}^{d\times d}$ be a symmetric matrix.
   - Show that $det(A) = \prod D_{ii}$.
   - Show that $tr(A) = \sum D_{ii}$.

5. Let $A = UDU^\top$ be the EVD of A. Provide an expression for A^k for $k \in \mathbb{N}$, using the eigenvalues of A.

6. Let $U \in \mathbb{R}^{d\times d}$ be an orthogonal matrix. Show that U is isometric, that is $||U\mathbf{x}||_2 = ||\mathbf{x}||_2 \ \forall \mathbf{x} \in \mathbb{R}^d$.

7. Let $A \in \mathbb{R}^{m\times d}$ be a real matrix of rank k and $A = U\Sigma V^\top$ an SVD of A. Show that the four fundamental subspaces of A are given by:

$$\begin{array}{rclcrcl} \mathcal{N}(A) & = & \text{span}\{\mathbf{v}_{k+1},\ldots,\mathbf{v}_d\}, & & \mathcal{R}(A) & = & \text{span}\{\mathbf{u}_1,\ldots,\mathbf{u}_k\} \\ \mathcal{N}(A^\top) & = & \text{span}\{\mathbf{u}_{k+1},\ldots,\mathbf{u}_m\}, & & \mathcal{R}(A^\top) & = & \text{span}\{\mathbf{v}_1,\ldots,\mathbf{v}_k\} \end{array}$$

B Calculus

B.1 Gradients, Jacobians & Hessian

The *derivative* of a function measures the degree in which the function changes in a small area around a point of interest. For a scalar function $f : \mathbb{R} \to \mathbb{R}$, the derivative of f at point $x \in \mathbb{R}$ is defined as

$$\frac{d}{dx} f(x) := \lim_{a\to 0} \frac{f(x+a) - f(x)}{a}$$

■ **Example .5** Consider the **Re**ctified **L**inear **U**nit function defined as the positive part of its argument: $f(x) = \max(0,x)$. The derivative of this function is:

$$\frac{\partial}{\partial x} f(x) = \mathbb{1}_{x>1}$$

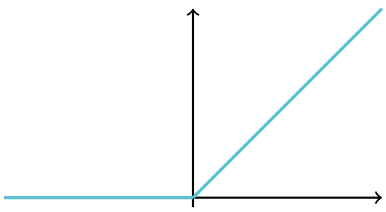


Figure 9: *ReLU Function*

Note that at $x = 0$ the derivative of ReLU is undefined. To deal with such cases we will later define sub-gradients. ■

In the case of multivariate functions where $f : \mathbb{R}^d \to \mathbb{R}$, the notion of derivative is generalized to the degree in which a function changes in each of the input coordinates. The *partial derivative* of f at point $\mathbf{x} \in \mathbb{R}^d$ with

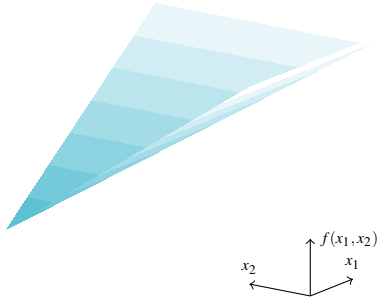
respect to x_i is defined as

$$\begin{aligned}\frac{\partial}{\partial x_i} f(\mathbf{x}) &:= \lim_{a\to 0} \frac{f(\mathbf{x}+a\mathbf{e}_i) - f(\mathbf{x})}{a} \\ &= \lim_{a\to 0} \frac{f(x_1,\ldots,x_i+a,\ldots,x_d) - f(x_1,\ldots,x_i,\ldots,x_d)}{a}\end{aligned}$$

where $\mathbf{e}_i$ is the i-th standard basis vector. Namely, a partial derivative of a function is its derivative with respect to one of its variables, while all others are kept constant.

■ **Example .6** For $f(\mathbf{x}) = \max_i(x_1,\ldots,x_d)$ the partial derivatives of f at x_i are:

$$\frac{\partial}{\partial x_i} f(\mathbf{x}) = \begin{cases} 1, & x_i = \max_i(x_1,\ldots,x_d) \\ 0, & \text{o.w} \end{cases} = \mathbb{1}_{i=\text{argmax}\,(x_1,\ldots,x_d)}$$



■

■ **Example .7** For $f(x,y) = x^2 + xy + y^2$ the partial derivatives of f at (x_0,y_0) are

$$\frac{\partial}{\partial x} f(x_0,y_0) = 2x_0 + y_0, \quad \frac{\partial}{\partial y} f(x_0,y_0) = 2y_0 + x_0$$

■

Then the notion of *gradient* of $f : \mathbb{R}^d \to \mathbb{R}$ at $\mathbf{x}$ is simply the vector of all partial derivatives. It is as a convention that we define the gradient as a column vector.

$$\nabla f(\mathbf{x}) := \left(\frac{\partial f(\mathbf{x})}{\partial x_1}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_d}\right)^\top$$

■ **Example .8** Let $\mathbf{w} \in \mathbb{R}^d$ and $f : \mathbb{R}^d \to \mathbb{R}$ be a linear functional defined by $f(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle = \mathbf{w}^\top \mathbf{x}$. To calculate the gradient of f at point $\mathbf{x}$ we derive the partial derivatives of f. From linearity of the derivative:

$$\frac{\partial}{\partial x_j} f(\mathbf{x}) = \frac{\partial}{\partial x_j} \sum_i w_i x_i = \sum_i \frac{\partial}{\partial x_j} w_i x_i = w_j$$

Therefore, in vector notation $\nabla f(\mathbf{x}) = \mathbf{w}$. ■

■ **Example .9** Let $f : \mathbb{R}^d \to \mathbb{R}$ be defined by $f(\mathbf{x}) = ||\mathbf{x}||^2$. Using linearity of the derivative then

$$\frac{\partial}{\partial x_j} f(x) = \sum_i \frac{\partial}{\partial x_j} x_i^2 = 2x_j$$

which in vector notation can be written as $\nabla f(\mathbf{x}) = 2\mathbf{x}$. ■

The Jacobian is the generalization of the gradient for multivariate vector-valued functions, that is, functions that receive and output vectors. So for $f : \mathbb{R}^d \to \mathbb{R}^m$ where $f(\mathbf{x}) = (f_1(\mathbf{x}), \ldots, f_m(\mathbf{x}))^\top$, the *Jacobian* of f is the $m \times d$ matrix of all partial derivatives:

$$J_{\mathbf{x}}(f) \coloneqq \begin{bmatrix} \frac{\partial f_1(\mathbf{x})}{\partial x_1} & \cdots & \frac{\partial f_1(\mathbf{x})}{\partial x_d} \\ \vdots & & \vdots \\ \frac{\partial f_m(\mathbf{x})}{\partial x_1} & \cdots & \frac{\partial f_m(\mathbf{x})}{\partial x_d} \end{bmatrix}$$

■ **Example .10** Let $f : \mathbb{R}^2 \to \mathbb{R}$ be defined as $f(\mathbf{x}) = x_1^2 + x_2^2$. The Jacobian of f is:

$$J_{\mathbf{x}}(f) = \begin{bmatrix} \frac{\partial f_1(\mathbf{x})}{\partial x_1} & \frac{\partial f_1(\mathbf{x})}{\partial x_2} \end{bmatrix} = [2x_1, 2x_2] = \nabla f(\mathbf{x})^\top$$

■

Notice that for any function where $m = 1$ the Jacobian is in fact the transposed gradient vector: $J_{\mathbf{x}}(f) = \nabla f(\mathbf{x})^\top$.

■ **Example .11** Let $f : \mathbb{R}^d \to \mathbb{R}^m$ defined as $f(\mathbf{x}) = A\mathbf{x}$ where $A \in \mathbb{R}^{m \times d}$ and denote the rows of A as $\mathbf{a}_1, \ldots, \mathbf{a}_m$. To find the Jacobian of f, $J_{\mathbf{x}}(f)$, define the set of functions computing each coordinate in the output vector $\forall i \in [m] \quad f_i(\mathbf{x}) = \langle \mathbf{a}_i, \mathbf{x} \rangle$. Then the Jacobian of f is comprised of the gradients of $f_1, \ldots, f_m$ as rows. Notice that we have already computed the gradient of linear functionals in Example .8 so:

$$J_{\mathbf{x}}(f) = \begin{bmatrix} \nabla f_1(\mathbf{x})^\top \\ \vdots \\ \nabla f_d(\mathbf{x})^\top \end{bmatrix} = \begin{bmatrix} -\mathbf{a}_1 - \\ \vdots \\ -\mathbf{a}_m - \end{bmatrix} = A$$

■

Similar to the gradient we can consider the second derivative of a function with respect to each of its partial first derivatives. Let $f : \mathbb{R}^d \to \mathbb{R}$ be a twice differential function. The *Hessian* matrix H of f is the square matrix of second derivative:

$$H[f] \coloneqq \begin{bmatrix} \frac{\partial^2 f}{\partial^2 x_1} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_d \partial x_1} & \cdots & \frac{\partial^2 f}{\partial^2 x_d} \end{bmatrix}$$

■ **Example .12** Let us calculate the Hessian of the polynomial function $f(x_1, x_2) = x_1^2 + x_1 x_2 + x_2^2$. We begin with calculating the first partial derivatives of f: $\frac{\partial f(x_1, x_2)}{\partial x_i} = 2x_i + x_j$ for $i \in \{1, 2\}$ and $j \neq i$. Next, we calculate the derivative a second time with respect to each of the parameters:

$$\frac{\partial^2 f(x_1, x_2)}{\partial x_i \partial x_j} = \begin{cases} 2 & x_i = x_j \\ 1 & x_i \neq x_i \end{cases}$$

We therefore conclude that the Hessian of f is :

$$H = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

■

B.2 Chain Rules

Many times a given function of interest is the composition of other functions. In that case, when we calulate the derivatives we apply the chain rule. In the case of a scalar function, let $f : \mathbb{R} \to \mathbb{R}$ and $g : \mathbb{R} \to \mathbb{R}$ be two differential functions, then the derivative of the composite $f \circ g$ is:

$$(f \circ g)' := (f' \circ g) \cdot g'$$

Namely, for $h(x) = f(g(x))$ then $\forall x \in \mathbb{R}\ h'(x) = f'(g(x)) \cdot g'(x)$. We can extend this to the multivariate case where the composite receives a vector and outputs another.

Theorem B.1 Let $f : \mathbb{R}^d \to \mathbb{R}^m$ and $g : \mathbb{R}^k \to \mathbb{R}^d$. The Jacobian of the composition $(f \circ g) : \mathbb{R}^k \to \mathbb{R}^m$ at $\mathbf{x}$ is

$$J_{\mathbf{x}}(f \circ g) = J_{g(\mathbf{x})}(f) J_{\mathbf{x}}(g) := \begin{bmatrix} \frac{\partial f_1(g(\mathbf{x}))}{\partial g_1(\mathbf{x})} & \cdots & \frac{\partial f_1(g(\mathbf{x}))}{\partial g_d(\mathbf{x})} \\ \vdots & & \vdots \\ \frac{\partial f_m(g(\mathbf{x}))}{\partial g_1(\mathbf{x})} & \cdots & \frac{\partial f_m(g(\mathbf{x}))}{\partial g_d(\mathbf{x})} \end{bmatrix} \begin{bmatrix} \frac{\partial g_1(\mathbf{x})}{\partial x_1} & \cdots & \frac{\partial g_1(\mathbf{x})}{\partial x_k} \\ \vdots & & \vdots \\ \frac{\partial g_d(\mathbf{x})}{\partial x_1} & \cdots & \frac{\partial g_d(\mathbf{x})}{\partial x_k} \end{bmatrix}$$

In element-wise notation:

$$J_{\mathbf{x}}(f \circ g)_{i,j} := \sum_l \frac{\partial f_i(g(\mathbf{x}))}{\partial g_l(\mathbf{x})} \frac{\partial g_l(\mathbf{x})}{\partial x_j}$$

■ **Example .13** Next, let us calculate the gradient of the following function: $h(\mathbf{x}) = \frac{1}{2}||A\mathbf{x} - \mathbf{y}||^2$. Let's define $g(\mathbf{x}) = A\mathbf{x} - \mathbf{y}$ and $f(\mathbf{x}) = \frac{1}{2}||\mathbf{x}||^2$ and notice that $h = f \circ g$. As g is an affine transformation, we have seen in Example .11 that $J_{\mathbf{x}}(g) = A$. Notice, that as $Im(h) \subseteq \mathbb{R}$, the Jacobian of h equals to the transpose of its gradient. As seen in Example .9, $J_{g(\mathbf{x})}(f) = \frac{1}{2}(2g(\mathbf{x}))^\top = (g(\mathbf{x}))^\top$. Now, applying the chain rule:

$$J_{\mathbf{x}}(h) = J_{\mathbf{x}}(f \circ g) = J_{g(\mathbf{x})}(f) J_{\mathbf{x}}(g) = (A\mathbf{x} - \mathbf{y})^\top A$$

$$\nabla_{\mathbf{x}}(h) = J_{\mathbf{x}}(h)^\top = A^\top (A\mathbf{x} - \mathbf{y})$$

■

■ **Example .14** The softmax function defined over $S : \mathbb{R}^d \to [0,1]^d$ returns a vector that its coordinates sum up to 1. It is defined by

$$S(\mathbf{a})_j = \frac{e^{a_j}}{\sum_{k=1}^d e^{a_k}}$$

Let us calculate the derivative of the softmax function. Denote $g_i(\mathbf{a}) := e^{a_i}$ and $h(\mathbf{a}) := \sum_k g_k(\mathbf{a})$. So:

$$\frac{\partial S_i}{\partial a_j} = \frac{\partial}{\partial a_j} \frac{e^{a_i}}{\sum_d e^{a_k}} = \frac{\partial}{\partial a_j} \frac{g_i(\mathbf{a})}{h(\mathbf{a})}$$

Note that for any a_j the derivative of h is e^{a_j}. In the case of g_i, when deriving with respect to a_j we get that the derivative is e^{a_j} only if $i = j$. Otherwise, the derivative is 0. Therefore, the derivative of S_i in the case where $i = j$ is:

$$\frac{\partial}{\partial a_j} \frac{e^{a_i}}{\sum_k e^{a_k}} = \frac{e^{a_i}(\sum_k e^{a_k}) - e^{a_i} e^{a_j}}{(\sum_k e^{a_k})^2} = \frac{e^{a_i}}{(\sum_k e^{a_k})} \cdot \frac{(\sum_k e^{a_k}) - e^{a_j}}{(\sum_k e^{a_k})} = S_i(1 - S_j)$$

It is left to show the derivative in the case where $i \neq j$. Intuitively, the softmax function is a "soft" version of the argmax function. Instead of just selecting one maximal element, the softmax breaks the vector into segments with the maximal input element getting a proportionally larger portion, but the other elements getting some of it as well. ∎

B.3 Function Approximations

Consider a function $f : \mathbb{R} \to \mathbb{R}$ and recall the definition of the Taylor series of f at x_0 near x:

$$T(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x-x_0)}{n!} x^n = f(x_0) + f'(x_0)(x-x_0) + \frac{1}{2} f''(x_0)(x-x_0)^2 + \ldots$$

A linear approximation (or first order approximation) is an approximation of a general function using a linear function. For a differentiable function $f : \mathbb{R} \to \mathbb{R}$, Taylor's theorem implies that for a close enough x then

$$f(x) \approx f(x_0) + f'(x_0)(x-x_0)$$

We can now extend this theorem to define linear approximations of multivariate functions.

Definition B.1 Let $f : \mathbb{R}^d \to \mathbb{R}$ and $\mathbf{x}_0 \in \mathbb{R}^d$. The *linear approximation* of f for every $\mathbf{x} \in \mathbb{R}^d$ near $\mathbf{x}_0$ is defined as

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + \langle \nabla f(\mathbf{x}_0), \mathbf{x} - \mathbf{x}_0 \rangle$$

So if for example, we consider a bivariate function $f : \mathbb{R}^2 \to \mathbb{R}$, the linear approximation of f near the point (x_0, y_0) is:

$$f(x_0+x, y_0+y) \approx f(x_0, y_0) + x \cdot \frac{\partial f(x_0, y_0)}{\partial x} + y \cdot \frac{\partial f(x_0, y_0)}{\partial y}$$

Now, if f is a linear function, intuition dictates that the linear approximation of f would be the function itself. Let $\mathbf{b} \in \mathbb{R}^d$ and let $f : \mathbb{R}^d \to \mathbb{R}$ be defined by $f(\mathbf{x}) = \mathbf{b}^\top \mathbf{x}$. Then, the linear approximation of f near $\mathbf{x}_0 \in \mathbb{R}^d$ is

$$\begin{aligned} f(\mathbf{x}) &\approx f(\mathbf{x}_0) + \langle \nabla f(\mathbf{x}_0), \mathbf{x} - \mathbf{x}_0 \rangle \\ &= \mathbf{b}^\top \mathbf{x}_0 + \langle \mathbf{b}, \mathbf{x} - \mathbf{x}_0 \rangle \\ &= \mathbf{b}^\top (\mathbf{x}_0 + \mathbf{x} - \mathbf{x}_0) \\ &= f(\mathbf{x}) \end{aligned}$$

∎ **Example .15** Let $f : \mathbb{R}^2 \to \mathbb{R}$ be defined by $f(x,y) = \sqrt{x^2+y^2}$. Let us calculate the linear approximation of f near $(3,4)$. We begin with expressing the gradient of f. So, the partial derivative of f with respect to first argument at point (x_0, y_0) is:

$$\frac{\partial}{\partial x} f(x_0, y_0) = 2x_0 \cdot \frac{1}{2\sqrt{x_0^2+y_0^2}}$$

Therefore the gradient of f is $\nabla f(\mathbf{x}) = \left(\frac{x_0}{\sqrt{x_0^2+y_0^2}}, \frac{y_0}{\sqrt{x_0^2+y_0^2}} \right)^\top$. So for a point (x,y) in the vicinity of $(3,4)$ the linear approximation is:

$$f(3+x, 4+y) \approx 5 + \frac{3}{5}x + \frac{4}{5}y$$

If for example $x = 0.1, y = 0.2$ then $f(3+0.1, 4+0.2) = 5.2201.. \approx 5.22 = 5 + \frac{3}{5} \cdot 0.1 + \frac{4}{5} \cdot 0.2$ ∎

Using the gradient of a function we are able to provide a first order (linear) approximation of a function. Similarly, we can also provide a second order approximation of a function. The second order Taylor expansion of f near $\mathbf{x}$ is given by:

Definition B.2 Let $f:\mathbb{R}^d\to\mathbb{R}$ be a twice differentiable function and $\mathbf{x}_0\in\mathbb{R}^d$. The *second order approximation* (also referred to as quadratic approximation) of f for every $\mathbf{x}\in\mathbb{R}^d$ near $\mathbf{x}_0$ is defined as

$$f(\mathbf{x})\approx f(\mathbf{x}_0)+\langle\nabla f(\mathbf{x}_0),\mathbf{x}-\mathbf{x}_0\rangle+\tfrac{1}{2}(\mathbf{x}-\mathbf{x}_0)^\top H[f(\mathbf{x}_0)](\mathbf{x}-\mathbf{x}_0)$$

■ **Example .16** Returning to the second order polynomial function defined in Example .12, let us find the first- and second-order approximations of f near point $\mathbf{x}_0=(x_0,y_0)^\top$. The first order approximation is given by:

$$\begin{aligned}f(\mathbf{x}_0+\mathbf{x}) &\approx f(\mathbf{x}_0)+\langle\nabla f(\mathbf{x}_0),\mathbf{x}\rangle\\ &= x_0^2+x_0y_0+y_0^2+\begin{bmatrix}2x_0+y_0\\2y_0+x_0\end{bmatrix}^\top\begin{bmatrix}x\\y\end{bmatrix}\\ &= x_0^2+x_0y_0+y_0^2+2x_0x+y_0x+2y_0y+x_0y\end{aligned}$$

The second order approximation is given by

$$\begin{aligned}f(\mathbf{x}_0+\mathbf{x}) &\approx x_0^2+x_0y_0+y_0^2+2x_0x+y_0x+2y_0y+x_0y+\tfrac{1}{2}\begin{bmatrix}x\\y\end{bmatrix}^\top\begin{bmatrix}2&1\\1&2\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}\\ &= x_0^2+x_0y_0+y_0^2+2x_0x+y_0x+2y_0y+x_0y++x^2+yx+y^2\\ &= (x_0+x)^2+(x_0+x)(y_0+y)+(y_0+y)^2\end{aligned}$$

Notice that since f is a second order polynomial then the calculated value is the exact value of the function at $\mathbf{x}_0+\mathbf{x}$. ■

B.4 Convexity

A set $C\subseteq\mathbb{R}^d$ is called a *convex set* if and only if $\forall\mathbf{v},\mathbf{u}\in C$, $\forall\alpha\in[0,1]$ then $\alpha\mathbf{v}+(1-\alpha)\mathbf{u}\in C$. That is, if the line segment between any two vectors $\mathbf{v},\mathbf{u}\in C$, is contained in C. The vector $\alpha\mathbf{v}+(1-\alpha)\mathbf{u}$ is called a *convex combination* of $\mathbf{v}$ and $\mathbf{u}$.

(a) *convex sets examples* ***(b)*** *Non-convex sets examples*

Figure 10: *Examples of convex and non-convex sets*

■ **Example .17** Let V be a vector space and $U\subseteq V$. U is a convex set as for every $\mathbf{v},\mathbf{u}\in U$ and $\alpha\in[0,1]$ $\alpha\mathbf{v}+(1-\alpha)\mathbf{u}$ is a linear combination of vectors in U and therefore is also in U. ■

■ **Example .18** Consider the unit ball of some norm $B_{||\cdot||}=\{\mathbf{v}\in V:||\mathbf{v}||\leq 1\}$. This is a convex set as for any $\mathbf{v},\mathbf{u}\in B$ and $\alpha\in[0,1]$, the triangle inequality imples that:

$$\begin{aligned}\|\alpha\mathbf{v}+(1-\alpha)\mathbf{u}\| &\leq \|\alpha\mathbf{v}\|+\|(1-\alpha)\mathbf{u}\|\\ &= \alpha\|\mathbf{v}\|+(1-\alpha)\|\mathbf{u}\|\\ &\leq \alpha+1-\alpha=1\end{aligned}$$

■

■ **Example .19** Let $(\mathbf{w},b)$ be a hyperplane for $\mathbf{w}\in\mathbb{R}^d$ and $b\in\mathbb{R}$. The closed halfspace $W:=\{\mathbf{v}:\mathbf{w}^\top\mathbf{v}\leq b\}$ is a convex set. For any $\mathbf{v},\mathbf{u}\in W$ and $\alpha\in[0,1]$ it holds that:

$$\langle\mathbf{w},\alpha\mathbf{v}+(1-\alpha)\mathbf{u}\rangle=\alpha\langle\mathbf{w},\mathbf{v}\rangle+(1-\alpha)\langle\mathbf{w},\mathbf{u}\rangle\leq\alpha b+(1-\alpha)b=b$$

■

Convexity is preserved under several operations. The claim below demonstrates a few such operations.

Claim B.2 Convexity is preserved under the following operations:
1. The intersection $C := \bigcap_{i\in I} C_i$ for $\{C_i : i \in I\}$ a collection of convex sets.
2. The vector sum $C_1 + C_2 := \{c_1 + c_2 : c_1 \in C_1, c_2 \in C_2\}$ of two convex sets.
3. The set $\lambda C := \{\lambda c : c \in C\}$ is convex, for any convex set C, and every scalar λ.

Proof. Proving directly from definition:

1. Let $\mathbf{v}, \mathbf{u} \in C$, and let $\alpha \in [0,1]$. As C is the intersection of $\{C_i\}$ it holds that $\mathbf{v}, \mathbf{u} \in C_i$ for any $i \in I$. Since $\{C_i\}$ are convex sets then for any $\alpha \in [0,1]$ it holds that $\alpha\mathbf{v} + (1-\alpha)\mathbf{u} \in C_i$. Thus $\alpha\mathbf{v} + (1-\alpha)\mathbf{u} \in \bigcap_{i\in I} C_i = C$.

2. Let $\mathbf{v}, \mathbf{u} \in C_1 + C_2$, then there exists $\mathbf{v}_1, \mathbf{v}_2$ and $\mathbf{u}_1, \mathbf{u}_2$ for which

$$\mathbf{v} = \mathbf{v}_1 + \mathbf{v}_2 \text{ s.t. } \mathbf{v}_1 \in C_1, \mathbf{v}_2 \in C_2, \quad \mathbf{u} = \mathbf{u}_1 + \mathbf{u}_2 \text{ s.t. } \mathbf{u}_1 \in C_1, \mathbf{u}_2 \in C_2$$

   Let $\alpha \in [0,1]$, then:

$$\begin{aligned} \alpha\mathbf{v} + (1-\alpha)\mathbf{u} &= \alpha(\mathbf{v}_1 + \mathbf{v}_2) + (1-\alpha)(\mathbf{u}_1 + \mathbf{u}_2) \\ &= [\alpha\mathbf{v}_1 + (1-\alpha)\mathbf{u}_1] + [\alpha\mathbf{v}_2 + (1-\alpha)\mathbf{u}_2] \end{aligned}$$

   where, from convexity of C_1, C_2 it holds that:

$$\alpha\mathbf{v}_1 + (1-\alpha)\mathbf{u}_1 \in C_1, \quad \alpha\mathbf{v}_2 + (1-\alpha)\mathbf{u}_2 \in C_2$$

   and therefore $\alpha\mathbf{v} + (1-\alpha)\mathbf{u} \in C_1 + C_2$.

3. Let $\mathbf{v}, \mathbf{u} \in \lambda C$, then there exists $\mathbf{x}, \mathbf{y} \in C$ such that $\mathbf{v} = \lambda\mathbf{x}$ and $\mathbf{u} = \lambda\mathbf{y}$. Let $\alpha \in [0,1]$. Then:

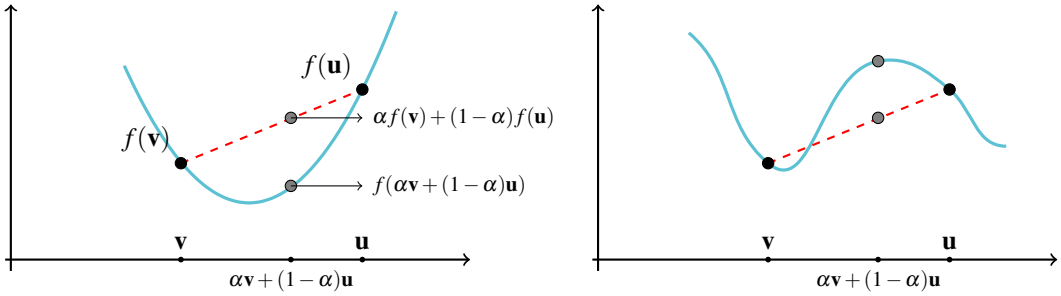
$$\alpha\mathbf{v} + (1-\alpha)\mathbf{u} = \lambda(\alpha\mathbf{x} + (1-\alpha)\mathbf{y}) \in \lambda C.$$

■

In a similar manner we can show that the vector sum of a finite set of convex sets is convex, giving a convex combination of arbitrary length.

Given a convex set, we can define the notion of a *convex function*. A function $f : C \to \mathbb{R}$ is convex if and only if $C \equiv \text{dom} f$ is convex and for every $\mathbf{u}, \mathbf{v} \in C$ and every $\alpha \in [0,1]$ then $f(\alpha\mathbf{v} + (1-\alpha)\mathbf{u}) \leq \alpha f(\mathbf{v}) + (1-\alpha) f(\mathbf{u})$.

This means that the line segment connecting any two points on the curve of a convex function f lies fully above that curve. In other words, the value of a convex function f at any convex combination $\mathbf{v}$ and $\mathbf{u}$ is always smaller than the convex combination of the values of f at $\mathbf{v}$ and $\mathbf{u}$.



(a) *Convex function* ***(b)*** *Non-convex function*

Figure 11: *Illustrating convex vs. non-convex function*

■ **Example .20** Let $||\cdot|| : \mathbb{R}^d \to \mathbb{R}$ be a norm, $\mathbf{v}, \mathbf{u} \in \mathbb{R}^d$ and $\alpha \in [0,1]$. From, the triangle inequality it holds that:

$$\|\alpha\mathbf{v} + (1-\alpha)\mathbf{u}\| \leq \|\alpha\mathbf{v}\| + \|(1-\alpha)\mathbf{u}\| = \alpha\|\mathbf{v}\| + (1-\alpha)\|\mathbf{u}\|$$

Hence, the norm is a convex function. ■

■ **Example .21** Let $f : \mathbb{R}^d \to \mathbb{R}$ be an affine transformation, that is $f(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle + b$ for $\mathbf{w} \in \mathbb{R}^d$ and $b \in \mathbb{R}$. Let $\mathbf{v}, \mathbf{u} \in \mathbb{R}^d$ and $\alpha \in [0,1]$, then:

$$\begin{aligned} f(\alpha\mathbf{v} + (1-\alpha)\mathbf{u}) &= \langle \mathbf{w}, \alpha\mathbf{v} + (1-\alpha)\mathbf{u} \rangle + b \\ &= \alpha(\langle \mathbf{w}, \mathbf{v} \rangle + b) + (1-\alpha)(\langle \mathbf{w}, \mathbf{u} \rangle + b) \\ &= \alpha f(\mathbf{v}) + (1-\alpha) f(\mathbf{u}) \end{aligned}$$

■

■ **Example .22** Let $f(\mathbf{w}) := ||X\mathbf{w} - \mathbf{y}||^2$ for $X \in \mathbb{R}^{m \times d}, \mathbf{y} \in \mathbb{R}^m$ and $\mathbf{w} \in \mathbb{R}^d$. Let us show that f is convex in $\mathbf{w}$. Let $\mathbf{v}, \mathbf{u} \in \mathbb{R}^d$ and $\alpha \in [0,1]$ then:

$$\begin{aligned} f(\alpha\mathbf{v} + (1-\alpha)\mathbf{u}) &= ||X[\alpha\mathbf{v} + (1-\alpha)\mathbf{u}] - \mathbf{y}|| \\ &= ||\alpha(X\mathbf{v} - \mathbf{y}) + (1-\alpha)(X\mathbf{v} - \mathbf{y})|| \\ &\leq \alpha||X\mathbf{v} - \mathbf{y}|| + (1-\alpha)||X\mathbf{u} - \mathbf{y}|| \\ &= \alpha f(\mathbf{v}) + (1-\alpha) f(\mathbf{u}) \end{aligned}$$

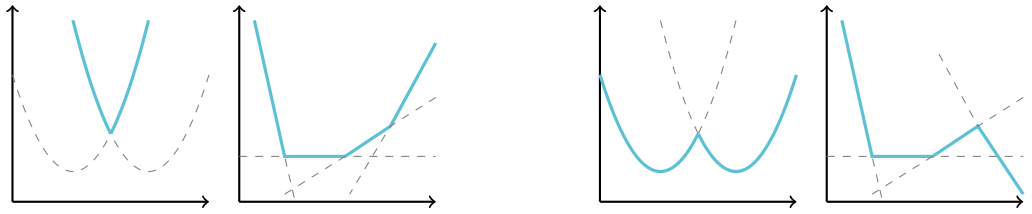
■

Some commonly used convex functions are the exponent ($x \to e^{ax}\ \forall a$), $x \to x^a\ \forall a \notin (0,1)$, and negative logarithm ($x \to -\log(x)$). Furthermore, as can be seen from Example .21 and from Example .22, affine transformations $\mathbf{x} \to \mathbf{w}^\top\mathbf{x} + b$ are convex as well as quadratic transformations are convex $\mathbf{x} \to \mathbf{x}^\top A\mathbf{x} + \mathbf{w}^\top\mathbf{x} + \alpha$ for $A \succcurlyeq 0$. The family of ℓ_p norms (Example A.1) defined as $||\mathbf{x}||_p := \left(\sum x_i^p\right)^{1/p}$ is convex for $p \geq 1$.

On the basis of some known convex functions, we can define a set of closure properties of convexity. The following are a few of such operations that preserve convexity:

- Non-negative linear combinations preserve convexity. That is, for $g(\mathbf{x}) = \sum \alpha_i f_i(\mathbf{x})$, then g is convex if f_i are convex functions and α_i are non-negative.
- The function $g(\mathbf{x}) = \sup_i f_i(\mathbf{x})$ is convex if f_i are convex functions.
- Partial minimization of a function preserves convexity. That is, for a convex function $g : \mathbb{R}^{d+k} \to \mathbb{R}$ defined by $(\mathbf{x}_1|\mathbf{x}_2) \to g(\mathbf{x}_1|\mathbf{x}_2)$ for $\mathbf{x}_1 \in \mathbb{R}^d, \mathbf{x}_2 \in \mathbb{R}^k$, then the partial minimization function $h : \mathbb{R}^d \to \mathbb{R}$ defined by $h(\mathbf{x}_1) = \min_{\mathbf{x}_2 \in C} g(\mathbf{x}_1, \mathbf{x}_2)$ (where $C \subset \mathbb{R}^k$ is a convex set) is a convex function.
- For $g : \mathbb{R}^d \to \mathbb{R}$ convex function and $h : \mathbb{R} \to \mathbb{R}$ convex and nondecreasing, the composition $h \circ g$ is a convex function.

- For $h : \mathbb{R}^k \to \mathbb{R}$ convex and nondecreasing in each of its coordinates and $g_1, \ldots, g_k : \mathbb{R}^d \to \mathbb{R}$ convex functions, then $f(\mathbf{x}) = h(g_1(\mathbf{x}), \ldots, g_k(\mathbf{x}))$ is a convex function.



(a) *Convex functions* ***(b)*** *Non-convex functions*

Figure 12: *Examples of convex- and non-convex functions illustrating closure properties*

Properties of Convex Functions Convex functions have many useful properties, making them simple to optimize, i.e. finding the input that minimizes them.

Claim B.3 — First order characterization. Let $f : \mathbb{R}^d \to \mathbb{R}$ be a differential function, then f is convex if and only if $\mathrm{dom} f \subseteq \mathbb{R}^d$ is a convex set and $f(\mathbf{u}) \geq f(\mathbf{w}) + \langle \mathbf{u} - \mathbf{w}, \nabla f(\mathbf{w}) \rangle \;\; \forall \mathbf{u}, \mathbf{w} \in \mathrm{dom} f$.

Proof. Without loss of generality, assume $\mathbf{w} = \mathbf{0}$ since otherwise we could simply shift the axis. Therefore, it is enough to show that for any $\mathbf{u}$, $f(\mathbf{u}) \geq f(\mathbf{0}) + \langle \nabla f(\mathbf{0}), \mathbf{u} \rangle$. As f is convex it holds that

$$\begin{aligned} f(\alpha \mathbf{u}) \;&\leq\; (1-\alpha) f(\mathbf{0}) + \alpha f(\mathbf{u}) \\ &\Downarrow \\ \frac{f(\alpha \mathbf{u}) - f(\mathbf{0})}{\alpha} \;&\leq\; f(\mathbf{u}) - f(\mathbf{0}) \end{aligned}$$

This holds for any $\alpha \in [0,1]$ and thus taking the limit of both sides, with $\alpha \to 0$, and recalling that $\lim_{\alpha \to 0} \frac{f(\alpha \mathbf{u}) - f(\mathbf{0})}{\alpha} = \langle \mathbf{u}, \nabla f(\mathbf{0}) \rangle$ concluding the proof. ■

Namely, at any point on the graph of a convex (and differential) function f, the tangents lie below the graph of the function.

■ **Example .23** Let $f : \mathbb{R}^d \to \mathbb{R}$ be a linear functional defined by $f(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle$ for $\mathbf{w} \in \mathbb{R}^d$. Let us show that f is convex using the characterization above. For any $\mathbf{x} \in \mathbb{R}^d$ it holds that $\nabla f(\mathbf{x}) = \mathbf{w}$. Therefore, for a point $\mathbf{y} \in \mathbb{R}^d$ it holds that

$$f(\mathbf{y}) = \langle \mathbf{w}, \mathbf{y} \rangle = \langle \mathbf{w}, \mathbf{x} \rangle + \langle \mathbf{y} - \mathbf{x}, \mathbf{w} \rangle = f(\mathbf{x}) + \langle \mathbf{y} - \mathbf{x}, \nabla f(\mathbf{x}) \rangle$$

■

Notice that the affine transformation of $\mathbf{u}$ that is given by $f(\mathbf{w}) + \langle \mathbf{u} - \mathbf{w}, \nabla f(\mathbf{w}) \rangle$ is the first-order Taylor approximation of f near $\mathbf{w}$. Suppose we wish to estimate (i.e approximate) the value of f near a point of interest. We therefore learn that for a convex f the first-order Taylor approximation is a *global underestimator*. Furthermore it shows that in the case of a convex function, the *local* information about the function (i.e the gradient at $f(\mathbf{w})$) allows us to derive *global* information about it. An example for such information is seen in the following claim.

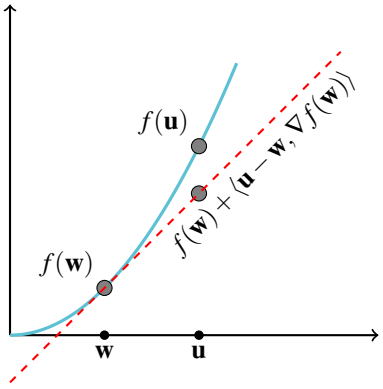


Figure 13: *Tangent of Convex Function*

Claim B.4 Let $f : \mathbb{R}^d \to \mathbb{R}$ be a convex function. If $f(\mathbf{u})$ is a local minimum then it is a global minimum.

Proof. Let $f : C \to \mathbb{R}$. and denote $B(\mathbf{u}, r)$ the intersection of C and a sphere of radius r around the point $\mathbf{u} \in C$:

$$B(\mathbf{u}, r) \coloneqq \{\mathbf{v} : \mathbf{v} \in C, \|\mathbf{v} - \mathbf{u}\| \leq r\}$$

Let $\mathbf{u}$ be a local minimizer of f (i.e. $f(\mathbf{u})$ is a local minimum). Therefore, there exists $r > 0$ such that for any $\mathbf{v} \in B(\mathbf{u}, r)$ it holds that $f(\mathbf{u}) \leq f(\mathbf{v})$. Note that since f is a convex function, its domain, C, is a convex set - otherwise, the convex combination of two vectors in C may not belong to C and the convexity condition in definition would not be well-defined. Let $\mathbf{v} \in C$ (not necessarily in B) and take $\alpha \in [0, 1]$. Since C is convex, $\alpha\mathbf{u} + (1-\alpha)\mathbf{v}$ is in C. It follows that to ensure that $\alpha\mathbf{u} + (1-\alpha)\mathbf{v}$ is in B it is enough to choose an α close enough to 1 so that the distance between $\alpha\mathbf{u} + (1-\alpha)\mathbf{v}$ and $\mathbf{u}$ is smaller than r. Since $\alpha\mathbf{u} + (1-\alpha)\mathbf{v}$ is inside B and C is convex, by definition one has

$$\begin{aligned} f(\mathbf{u}) \leq f(\alpha\mathbf{u} + (1-\alpha)\mathbf{v}) \;&\leq\; \alpha f(\mathbf{u}) + (1-\alpha) f(\mathbf{v}) \\ &\Downarrow \\ (1-\alpha) f(\mathbf{u}) \;&\leq\; (1-\alpha) f(\mathbf{v}) \end{aligned}$$

Hence, $f(\mathbf{u}) \leq f(\mathbf{v})$. This holds for every $\mathbf{v}$, hence $f(\mathbf{u})$ is also a global minimum of f. ■

Claim B.5 — Second order characterization. Let $f : \mathbb{R}^d \to \mathbb{R}$ be a twice differentiable function, then f is convex if and only if $\mathrm{dom} f \subset \mathbb{R}^d$ is a convex set and $\nabla^2 f(\mathbf{x}) \succcurlyeq 0$, $\forall \mathbf{x} \in \mathrm{dom} f$.

■ **Example .24** Let $A \in \mathbb{R}^{d \times d}$ be a symmetric matrix and let $f : \mathbb{R}^d \to \mathbb{R}$ defined as $f(\mathbf{x}) = \mathbf{x}^\top A\mathbf{x}$. Let us show that f is convex using the characterization above. To do so we find the first and second derivatives of f. Denote $g(\mathbf{x}) = A\mathbf{x}$ and $h(\mathbf{x}) = \mathbf{x}$ so we can write f as $f \equiv h^\top g$. Using the product rule then:

$$\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} = \frac{\partial h(\mathbf{x})^\top}{\partial \mathbf{x}} \cdot g(\mathbf{x}) + \frac{\partial g(\mathbf{x})}{\partial \mathbf{x}} \cdot h(\mathbf{x})$$

As the derivative of g by $\mathbf{x}$ is A then:

$$\nabla f(\mathbf{x}) = A^\top \mathbf{x} + A\mathbf{x} = \left(A + A^\top\right)\mathbf{x}$$

Since A is symmetric then $\nabla f(\mathbf{x}) = 2A\mathbf{x}$. Next, let us compute the second derivative:

$$\nabla^2 f(\mathbf{x}) = \frac{\partial^2 f}{\partial^2 \mathbf{x}} = \frac{\partial 2A\mathbf{x}}{\partial \mathbf{x}} = 2A$$

Thus, by the characterization above f is convex if and only if $A \succcurlyeq 0$. That is, if and only if A is a PSD matrix. ■

■ **Example .25** Recall the function $f(\mathbf{w}) := ||X\mathbf{w} - \mathbf{y}||^2$ for $X \in \mathbb{R}^{m\times d}, \mathbf{y} \in \mathbb{R}^m$ and $\mathbf{w} \in \mathbb{R}^d$ seen in Example .22. Let us show that f is convex using the characterization above. As we have shown in Example .13, the gradient of $f = h \circ g$ for $g(\mathbf{w}) := X\mathbf{w} - \mathbf{y}$ and $h(\mathbf{w}) := ||\mathbf{w}||^2$ is:

$$\nabla_{\mathbf{w}} f(\mathbf{w}) = J_{\mathbf{w}}(f(\mathbf{w}))^\top = \left(J_{g(\mathbf{w})}(h) J_{\mathbf{w}}(g)\right)^\top = 2X^\top (X\mathbf{w} - \mathbf{y})$$

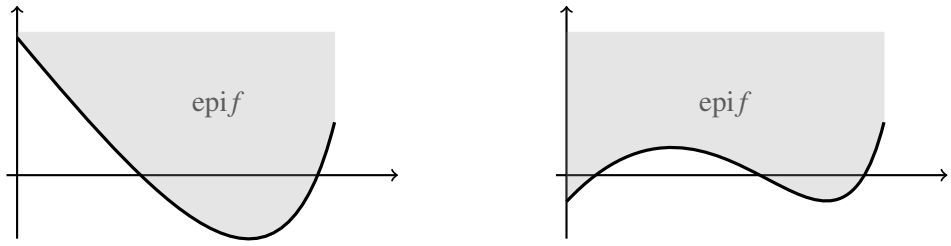
Next, we calculate the second derivative of f, namely, we calculate the partial derivatives of the graident of f with respect to each of its coordinates.

$$H_{\mathbf{w}}[f(\mathbf{w})] = \frac{\partial}{\partial \mathbf{w}} \nabla_{\mathbf{w}} f(\mathbf{w}) = 2X^\top X$$

Since $X^\top X \succcurlyeq 0$ we conclude that the hessian of f is PSD and thus f is a convex function. ■

R Notice that the requirement of $\mathrm{dom} f$ to be a convex set is necessary in both the first- and second-order characterizations.

Given a function $f : \mathbb{R}^d \to \mathbb{R}$, the *graph* of the function is defined as the set $\{(\mathbf{x}, f(\mathbf{x})) \,|\, \mathbf{x} \in \mathrm{dom} f\} \subseteq \mathbb{R}^{d+1}$. The *epigraph* of f is the set of all points above f, $\mathrm{epi} f := \{(\mathbf{x}, \beta) \,|\, \mathbf{x} \in \mathrm{dom} f, f(\mathbf{x}) \leq \beta\}$. The connection between convex sets and convex functions is through the epigraph of the function. A function $f : \mathbb{R}^d \to \mathbb{R}$ is convex if and only if $\mathrm{epi} f$ is a convex set.



(a) *Convex function & convex epigraph*

(b) *Non-convex function & non-convex epigraph*

B.5 Exercises

Theoretical Questions

1. Let $S : \mathbb{R}^d \to [0,1]^d$ be the softmax function defined by $S(\mathbf{a})_j = \frac{e^{a_j}}{\sum_{k=1}^d e^{a_k}}$. In Example .14 we have shown that the derivative $\frac{\partial}{\partial a_j} S(\mathbf{a})_i$ in the case where $i = j$ is $S_i(1 - S_j)$. Calculate the derivative in the case where $i \neq j$.

2. Show that the set of real PSD matrices $V \coloneqq \{A \in \mathbb{R}^{d\times d} | A \in PSD\}$ is a convex set.

3. Show that the image and pre-image of an affine transformation are convex sets.

4. Show that the image and pre-image of the linear-fraction function are convex sets. The linear-fraction function is defined as $f(\mathbf{x}) = \frac{A\mathbf{x}+\mathbf{b}}{\mathbf{c}^\top \mathbf{x}+\mathbf{d}}$

5. Show that quadratic functions $\mathbf{x} \to \mathbf{x}^\top A\mathbf{x} + \mathbf{w}^\top \mathbf{x} + \alpha$ are convex if and only if $A \succcurlyeq 0$.

6. Show that for any $p \geq 1$ the norm $||\mathbf{x}||_p \coloneqq \left(\sum x_i^p\right)^{1/p}$ is a convex function.

7. Show that the log-sum-exp function, defined as $f(\mathbf{x}) = \log\left(\sum_{i=1}^k \exp\left(\mathbf{w}_i^\top \mathbf{x} + b_i\right)\right)$ is a convex function. Hint: use the fact that affine compositions preserve convexity and the second order characterization.

8. Show that the following function is a convex function: $f(\mathbf{w}) \coloneqq \frac{1}{m}\sum_i \log(1+\exp(-y_i \langle \mathbf{w}, \mathbf{x}_i\rangle))$ for $\mathbf{w}, \mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$. This function is the loss function used in the logistic regression optimization problem (3.28).

9. Let $f(\mathbf{w})$ be a differentiable convex function. Show that, beside the tangent at $\mathbf{w}$, no other line passing through the point $(\mathbf{w}, f(\mathbf{w}))$ lies fully below $f(\mathbf{w})$. In other words, show that if $\forall \mathbf{u} \in \mathbb{R}^d$, $f(\mathbf{u}) \geq f(\mathbf{w}) + \langle \mathbf{u}-\mathbf{w}, \mathbf{a}\rangle$ then $\mathbf{a} = \nabla f(\mathbf{w})$.

10. Show that $f(\mathbf{w}) \coloneqq \min_{i\in[m]} \left|\mathbf{w}^\top \mathbf{x}_i\right|$ for $\mathbf{x}_1, \ldots, \mathbf{x}_m$ is a concave function. Namely, that $-f$ is a convex function. This function is the margin which maximized in the Hard-SVM optimization problem (3.13).

11. Show that in both the first- and second-order characterization of a convex function, the requirement that dom$f$ is a convex set is necessary. That is, describe non-convex functions f whose domain is not a convex set and that $\nabla f \geq 0$ or $\nabla^2 f \succcurlyeq 0$.