EN.601.475: Machine Learning

Lecture Summary

James Guo

Fall 2024

Contents

I Supervised Learning

I.1 Learning a Function

Definition I.1.1. Function.

A function $f : X \mapsto Y$ maps inputs in its domain to a specific output in its range Y. ⌟

Computer scientists are concerned to construct functions with useful properties via algorithms, and the **fundamental problem** in Machine Learning is learning a function given a set of input/output examples.

Setup I.1.2. Input and Output.

We let a set of n input examples (each taking k inputs) as:

$$[X] := \begin{pmatrix} \vec{X}_1 \\ \vec{X}_2 \\ \vdots \vec{X}_n \end{pmatrix} = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1k} \\ x_{21} & x_{22} & \cdots & x_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nk} \end{pmatrix},$$

in which it is a k-by-n matrix (or a two dimensional array).
Y is a one dimensional output. ⌟

The goal of the learning is to learn an **unknown function** $f : \vec{X} \mapsto Y$.

Remark I.1.3. Assumptions of the Data.

The machine learning process needs to abide to the following assumptions:

(i) Examples in $[X]$ are *representative* and not picked to try to foll us.
In particular, we want **randomness** to be drawn from the sample. Otherwise, it will introduce **biases** into the model.

(ii) We have the access to *true outputs*, *i.e.*, $Y_i = f(\vec{X}_i)$.
True outputs are key in **supervised learning**, they are necessary to train the data.

(iii) A *hypothesis class*: $\mathcal{H} := \{\tilde{f} \mid \tilde{f} : \vec{X} \mapsto Y\}$.
This is the set of *all morphisms* from the input to output. If $f \notin \mathcal{H}$, the function is outside the hypothesis class. It is out of supervision for the learning. ⌟

Below is a simple example of learning a function with a **decision tree**.

I.2 Decision Tree

Decision trees are a special case for learning a function. In particular, it enforces the input and output to endorse binary results. Inputs can be multi-dimensional, though.

Example I.2.1. Defining Decision Trees.

Let input x_{ij} in $\vec{X}_i = (x_{i1}, \cdots, x_{ik})$ be binary (0 or 1), and the output Y also binary.
Here, $\mathcal{H}$ is all possible decision trees, whereas decision trees can be represented as flow chart data structure that decides on the output by sequentially considering question of input.

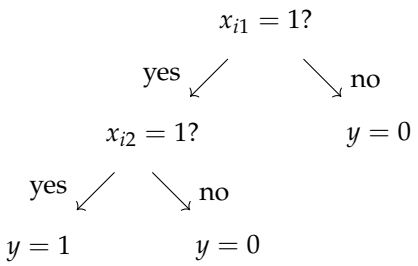


Figure I.1. Example of a binary decision tree.

Here, the size of $\mathcal{H}$ is $|\mathcal{H}| = 2^{2^k}$, which is *very big* but finite. In reality, when Y is not finite, $|\mathcal{H}|$ can be infinite. When $Y = \mathbb{R}$, $|\mathcal{H}|$ is (uncountably) infinite, which is not possible to deal with. ⌟

In order to judge how well a tree does on $[X]$ and $[Y]$, we need to penalize $\tilde{f} \in \mathcal{H}$ if it gets the output wrong. Hence, we need a quantitative analysis method.

Definition I.2.2. Badness of a Prediction Function.

The **loss function** checks how well a function guesses on a particular input $\vec{X}_i$ with the actual output Y_i, and is denoted $L\big(\tilde{f}(\vec{X}_i, Y_i)\big)$. The **badness** of a guess function $\tilde{f}$, or the **risk on data set** $[D]$, denoted $R_{[D]}(\tilde{f})$, is the sum of loss functions over the data set:

$$R_{[D]}(\tilde{f}) = \frac{1}{n} \sum_{i=1}^{n} L\big(\tilde{f}(\vec{X}_i, Y_i)\big).$$

One way to penalize is to apply a penalty of 1 *point* when $\tilde{f}(\vec{X}_i) \neq Y_i$, this is called a **0-1 loss function**. When using the 0-1 loss function, the badness of a function is:

$$R_{[D]}(\tilde{f}) = \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}\big(\tilde{f}(\vec{X}_i) \neq Y_i\big),$$

where $\mathbf{1}$ is the **indicator function**. ⌟

The goal is to pick a good decision tree, that is minimizing the badness of the function $\tilde{f}$. If we have $\tilde{f}_1, \tilde{f}_2 \in \mathcal{H}$ such that $R_{[D]}(\tilde{f}_1) < R_{[D]}(\tilde{f}_2)$, then $\tilde{f}_1$ is better.

Remark I.2.3. Picking a Good Decision Tree is a Search Problem.

There are a lot of possible trees. Therefore, picking a good decision tree becomes a **search problem**.

- A lot of algorithms in Machine learning solve search problems.
- In general, the search problem is **intractable**, *i.e.*, NP-hard, to find an element of $\mathcal{H}$ that minimizes common measured of badness. The **optimization theory** searches for cleaver algorithms.
- In practice, **greedy heuristics** and **local search methods** are often used.

These algorithms might not give the best result, but they are good enough for a problem. ⌟

To look for a good decision tree, we use the **ID3 Algorithm**, which is both local and greedy.

I.3 The ID3 Algorithm and Entropy of Data

To account for the efficiency, we want to know the **quality** of the data. Here, we use the definition from **information theory** to quantify.

Definition I.3.1. Entropy.

Let $[D]$ be a binary data set with outputs $[Y]$, with $\#1^{Y}_{[D]}$ denoting the number of Y that is 1, and $\#0^{Y}_{[D]}$ denoting the number of Y that is 0, the entropy of Y on data set $[D]$ is:

$$H([D]) := -\frac{\#1^{Y}_{[D]}}{|[D]|}\log_2\left(\frac{\#1^{Y}_{[D]}}{|[D]|}\right) - \frac{\#0^{Y}_{[D]}}{|[D]|}\log_2\left(\frac{\#0^{Y}_{[D]}}{|[D]|}\right).$$ ⌟

Specifically, we can make $[D]$ into conditional data set, *i.e.*, selecting conditional data from $[D]$.

Algorithm I.3.2. The ID3 Algorithm.

Let $[D]$ represent the binary data set with input $[X_j]$ and output $[Y]$. The algorithm lies as follows:

(i) Given a list of examples $[D] := ([X], [Y])$, for every variable X_j, we calculate the **quality** of X_j, that is, how much easier it is to predict Y in each branch if we split on X_j.

$$\text{quality of } X_j = H([D]) - \frac{\#0^{X_j}_{[D]}}{|[D]|}H([D|X_j = 0]) - \frac{\#1^{X_j}_{[D]}}{|[D]|}H([D|X_j = 1]).$$

(ii) Then, we create the root of the tree and split on X_j with the highest quality.

(iii) Recursively repeat step (i) with the left and right branches using the subset of data where $X_j = 1$ and $X_j = 0$, that is, $[D|X_j = 1]$ and $[D|X_j = 0]$, respectively.

Eventually, we stop splitting *at some point*. ⌟

Here, we need to think of when to stop the decision tree. Here are some cases with the ID3 algorithm of when to stop:

- If $[D|X_i = x_i, X_j = x_j, \cdots]$ has all the same outcome values for Y, then we should stop splitting, since what we have is as well as possible on the current subset of the data.
- If the remaining attributes $[D|X_i = x_i, X_j = x_j, \cdots]$ all have the same values, then we should stop splitting, since there is nothing left to split on.
- If there is no information gain for any potential split, we may need to keep going. We might want to choose one attribute and go on for the split. *Example:* `XOR` *is a half-half split*.

Remark I.3.3. Notes on the ID3 Algorithm.

- Decision trees can represent any functions from $(x_1, \cdots, x_k)$ to y, as it acts as a truth table.
- The ID3 search does not necessary minimize the 0-1 loss, but it tries to find trees with low 0-1 loss.
- During the ID3 construction, we need to choose the split with the most information gain.
- Different decisions trees could represent the same function.
- The ID3 search i snot perfectly accurate:
  - If $[D]$ has a lot of columns, it is unrealistic to contain them all.
  - A good fit on the data does not imply combinations of input that are not seen.
  - The true f might not be in $\mathcal{H}$, necessarily, and this is a **universal** problem in Machine Learning.

In general, perfect accuracy is hopeless for the ID3 search. ⌟

Speaking of that, the data set is not including all input/output combinations, so there are **biases** included as we train the model.

I.4 Inductive Bias and Overfitting

When picking simple trees, it might not capture the trends in the data set. However, when picking very complicated trees, even if we have **perfect accuracy** we still have problems with it.

Here, at perfection, it is just regurgitating the data from inputs, but it might not account for the patterns when there are no existing data.

Definition I.4.1. Inductive Bias.

The **inductive bias** of a learning algorithm is the set of assumptions that the learner uses to predict outputs of given inputs that it has not encountered. It is on how we want to generalize our model. ⌟

In particular, when people train models, a common preference is to pick trees that are neither *too simple* nor *too complicated*, and such selection is **inductive bias**, since it impacts on the data that are not encountered.

To check for inductive bias, we typically hold some data in $[D]$ in reserve, and test them respectively.

Setup I.4.2. Training and Test Data.

The part of $[D]$ that we show to the algorithm is the **training data**, denoted $[D]_{\text{train}}$.
The part of $[D]$ that we test the algorithm on is the **test data**, denoted $[D]_{\text{test}}$. ⌟

The reservation of the **test data** allows us to verify if we have overfitting.

Definition I.4.3. Overfitting.

For some $f \in \mathcal{H}$, using the **risk** of training data, *i.e.*, the average loss of f, we say f is overfitting if:

$$R_{[D]_{\text{train}}} < R_{[D]}.$$ ⌟

For decision trees, the accuracy for both training and test data grows at first, where as the size of the tree got larger, the accuracy on training data keep increasing while the accuracy on test data drops. This is because the data is no longer **representative**.

Due to the nature of training, we would want to address the overfitting issue.

Remark I.4.4. Some Solutions to Overfitting.

Some possible deals with overfitting are:

(i) Try to stop splitting early when constructing the tree.
Terminating early would prevent the complicated model that remembers the input data.

(ii) Use another validation set $[D]_{\text{validation}}$ as a guide. These data are also unseen to the algorithm.
For example, we can merge two leaves where the resulting smallest tree yields the largest improvement on $[D]_{\text{validation}}$.

It is worth mentioning that no inductive bias will work on all problems well. ⌟

The above arguments are on overfitting, which is over the assumption that we have perfect data set and unlimited computational power. The following issue account for the reality.

Example I.4.5. The Curse of Dimensionality.

As the number of features for $\vec{X}$ grows, the **number of possible feature combinations** grows exponentially ($2^{|X|}$ for binary data, worse for discrete data, even worse for continuous data). The problem is that we need **exponentially growing amount of data** and **large trees** to approximate the function.
In general, things get increasingly difficult as problem dimension increases. Some algorithms perform poorly, and others stop working entirely. ⌟

This problem is a **general phenomenon**, it applies to other learning problem, and there is no simple solution. The typical solution is to put **restrictions** on $\mathcal{H}$.

I.5 Generalization of Supervised Learning

The supervised learning can be extended to more cases from the binary decision tree.

- For **discrete inputs and outputs**, we use the same algorithm but do splits into *more than* 2 branches.
- For **continuous inputs**, we instead pick a **threshold** to split on. The feature is if it is larger than ($<$) or no greater ($\leq$) than the threshold.

- For **streaming inputs**, there will be improvements on data with more data points, we learn a tree on an initial $[D]$, and then modify the tree as new examples come in.
  This is an example of an **online algorithm**.
- For **missing outcomes and features**, this becomes a **semi-supervised learning problem**. It will be difficult if the variables are missing systematically.
  At larger, literature in statistics on missing data problems deals with this situation.
- For **corrupted entries** in $[D]$, it becomes a measurement error problem. It will be very hard to deal with.
- For **adversarial or dependent entries** in $[D]$, often in a network, where the samples there are *not necessarily true*. It will also be hard to deal with.

II Probability Theory

II.1 Preliminaries of Probability

Probability theory was first invented for analyzing games of chance. Later, it was applied to statistics/-machine learning to address the **inherent uncertainty** about values.

Definition II.1.1. Random Variable.

Denotes X with values $x \in X$, a random variable represents outcomes that are random, whereas events are when random variable X assumes value x, *i.e.*, $X = x$. ⌟

We often use randomness to model our ignorance of information. In particular, we use probability to encode **uncertainty** about events.

Definition II.1.2. Probability.

Probability are real numbers, and $\mathbb{P}(X = x)$ denotes the probability of event $X = x$. It abides to the following *Kolmogorov's axiomatization of probability*:

(i) $0 \leq \mathbb{P}(A) \leq 1$,

(ii) $\mathbb{P}(\text{True, certain event}) = 1$,

(iii) $\mathbb{P}(\text{False, impossible event}) = 0$, and

(iv) $\mathbb{P}(A \text{ or } B) = \mathbb{P}(A) + \mathbb{P}(B) - \mathbb{P}(A, B)$, where $\mathbb{P}(A, B)$ is the probability of both A and B. ⌟

We can always represent probability by **Venn Diagrams**, where event probabilities are represented by areas. The entire area is 1 and areas cannot be less than zero.

Probability theory has a variety of properties from the axioms, and they are important to construct the understanding of probability.

Remark II.1.3. Important Results in Probability.

- The conditional probability of A given B is: $\mathbb{P}(A|B) = \mathbb{P}(A,B)/\mathbb{P}(B)$.
- A joint probability distribution describes probabilities for a set of **mutually exclusive**, **exhaustive** events. These probabilities always sum to 1 and they partition the entire event space.
  The joint distribution gives the probability of *any event combination*.
- For binary random variables A, B, we have:
$$\mathbb{P}(A) = 1 - \mathbb{P}(\neg A) \text{ and } \mathbb{P}(A) = \mathbb{P}(A,B) + \mathbb{P}(A,\neg B).$$
- For discrete A, B of finite values, assume A has k values:
$$\mathbb{P}(A = a_1 \text{ or } A = a_2 \text{ or } \cdots \text{ or } A = a_k) = \sum_{j=1}^{k} \mathbb{P}(A = a_j),$$
$$\mathbb{P}(B) = \mathbb{P}\big(B, (A = a_1 \text{ or } \cdots \text{ or } A = a_k)\big) = \sum_{j=1}^{k} \mathbb{P}(B, A = a_j),$$
$$\sum_{j=1}^{k} \mathbb{P}(A = a_j) = 1.$$
- Marginal probability: We can express marginal probability by sum of joint probability:
$$\mathbb{P}(X_1 = x_1, \cdots, X_m = x_m) = \sum_{x_{m+1}, \cdots, x_k} \mathbb{P}(X_1 = x_1, \cdots, X_m = x_m, X_{m+1} = x_{m+1}, \cdots, X_k = x_k).$$
  Then, we can express the conditional probability by the joint and marginal probability:
$$\mathbb{P}(X_1 = x_1, \cdots, X_m = x_m | X_{m+1} = x_{m+1}, \cdots, X_k = x_k) = \frac{\mathbb{P}(X_1 = x_1, \cdots, X_k = x_k)}{\mathbb{P}(X_{m+1} = x_{m+1}, \cdots, X_k = x_k)},$$
  which is equivalently $\mathbb{P}(A|B) = \sum_C \mathbb{P}(A,B,C) / \sum_{C,A} \mathbb{P}(A,B,C)$. ⌟

Despite the effectiveness of probability theory, it exhibits certain problems implementation-wise.

- The joint distribution is very big, it grows exponentially, so it might not fit in a computer.
- It is hard to specify lots of probabilities from experts.
- It is hard to do the probabilistic calculations, such as summing exponentially many probabilities.

Alias, here are some additional rules that can account for the situation.

Algorithm II.1.4. Chain Rule and Bayes' Rule.

The **chain rule** applies as:
$$\mathbb{P}(X_1, \cdots, X_k) = \prod_{i=1}^{k} \mathbb{P}(X_i | X_{i-1}, \cdots, X_1).$$
The **Bayes' rule** applies as:
$$\mathbb{P}(A|B) = \mathbb{P}(B|A) \cdot \frac{\mathbb{P}(A)}{\mathbb{P}(B)}.$$ ⌟

The **chain rule** uses the multiplications of conditional so it avoids the large joint distribution.

The **Bayes' rule** allows us to easily manipulate the data, especially in the following circumstances.

- Bayes' rule allows us to update our degree of belief in response to more evidence, and
- Bayes' rule allows us to express hard to elicit probabilities in terms of easy to elicit ones. There are cases when one direction is *easier* compared to the other direction.

II.2 Independence Relationship

Then, we discuss another tool to help, which is **independence**. It is an equality restriction on joint distributions that encodes irrelevance.

Definition II.2.1. Marginally Independent.

Random variables A and B are **marginally independent** if:

$$\mathbb{P}(A, B) = \mathbb{P}(A) \cdot \mathbb{P}(B).$$

It is written as $A \amalg B$. ⌟

More than marginally independent, there are also independence established on the conditional sense.

Definition II.2.2. Conditionally Independent.

Random variables A and B are **conditionally independent** given C if:

$$\begin{cases} \mathbb{P}(A|B, C) = \mathbb{P}(A|C), \\ \mathbb{P}(B|A, C) = \mathbb{P}(B|C), \end{cases}$$

or equivalently:

$$\mathbb{P}(A, B|C) = \mathbb{P}(A|C) \cdot \mathbb{P}(B|C).$$

It is denoted as $A \amalg B|C$. ⌟

The conditional independence can be generalized to sets of variables in the same manner. Additionally, it follows the below properties with events A, B, C, D:

- Symmetry: $A \amalg B|C \iff B \amalg A|C$.
- Chain rule: $A \amalg B \cup D|C \iff A \amalg D|C$ and $A \amalg B|C \cup D$.

The independence is useful as it makes some probabilistic reasoning **tractable**.

Example II.2.3. Independence make Joint Distribution more Tractable.

When we have k binary variables $X_1, \cdots, X_k$, the size of the joint distribution is $2^k - 1$ (where the minus 1 is due to their sum is 1).
However, if every variable is marginally independent of the others, that is:

$$X_1 \amalg \{X_2, \cdots, X_k\}, X_2 \amalg \{X_3, \cdots, X_k\}, \cdots, X_{k-1} \amalg X_k,$$

we then have:

$$\mathbb{P}(X_1, X_2, \cdots, X_k) = \mathbb{P}(X_1) \cdot \mathbb{P}(X_2, \cdots, X_k) = \cdots = \mathbb{P}(X_1) \cdot \mathbb{P}(X_2) \cdots \mathbb{P}(X_k),$$

and since every $\mathbb{P}(X_i)$ must sum to 1, so we need specify 1 probability for each i, summing to a total of size k for the joint distribution, which is more tractable. ⌟

The independence of features becomes an important tool to combat with the **curse of dimensionality**.

Definition II.2.4. Expectation and Variance.

For a discrete random variable X, the expected value is the average weighted by probabilities:

$$\mathbb{E}[X] := \sum_{i=1}^{m} x_i \cdot \mathbb{P}(X = x_i) \text{ if } X \text{ has } m \text{ values.}$$

The variance of X is then:

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

Although X is random, $\mathbb{E}[X]$ and $\text{Var}[X]$ are fixed. ⌟

In particular, $\mathbb{E}[X]$ describes the **average** value of X and $\text{Var}[X]$ describes the **spread** of X. These are important concepts for random variables.

II.3 Real-Valued Distributions

Many features and outcomes in machine learning are real-valued, so we extend probability to real-valued variables using **density functions**.

Setup II.3.1. Real-Valued Random Variables.

For a real-valued random variable X, the **density function** $f(x)$ is defined to be:

$$f(x) := \lim_{h \to 0} \mathbb{P}\big(X \in (x, x+h)\big),$$

where $f(x)$ is the limit of the probability that X has a value in a little line segment around every point. For a set of real valued $\vec{X}$, we can define the joint density function $f(\vec{x})$ to be:

$$f(\vec{x}) = \lim_{\vec{h} \to \mathbf{0}} \mathbb{P}\big(\vec{X} \in (\vec{x}, \vec{x} + \vec{h})\big).$$

Here, $f(\vec{x})$ becomes the limit of the probability $\vec{X}$ in a little hyper sphere around every point $\vec{x}$. ⌟

An important example of a real-valued random variable is the **Gaussian distribution**.

Definition II.3.2. Gaussian Distribution.

The density of the Gaussian distribution is:

$$f\big(x; \{\mu, \sigma^2\}\big) = \mathcal{N}(\mu, \sigma^2) = \frac{1}{\sqrt{2\sigma^2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right).$$

Here, X has density $\mathcal{N}(\mu, \sigma^2)$, with $\mathbb{E}[X] = \mu$ and $\text{Var}[X] = \sigma^2$. ⌟

A lot of the real-valued variables are approximately Gaussian, hence its applications are wide.

Remark II.3.3. Properties of Density Functions.

- Non-negativity: $f(\vec{x}) \geq 0$.
- Normalization: $\int f(\vec{x})d\vec{x} = 1$.
- Positivity: although $\mathbb{P}(\vec{X}) \leq 1$, we may have $f(\vec{x}) > 1$. However, the area under f corresponding to the line segments of the x-axis must be ≤ 1.

⌟

Additionally, most properties and concepts for probabilities carry over to density functions.

- We can joint densities: $f(a, b)$, conditional densities: $f(a|b) = f(a, b)/f(b)$, and marginal densities: $f(a) := \int f(a, b)db$.
- Chain rule, Bayes' rule, conditional independence hold similarly as for probabilities.

Definition II.3.4. Expectation and Variance.

For a real-valued variable X, the expected value is integral of values over its support measure:

$$\mathbb{E}[X] := \int xf(x)dx.$$

The variance of X is still:

$$\mathrm{Var}[X] = \mathbb{E}\big[(X - \mathbb{E}[X])^2\big].$$

⌟

We also have a concept of **covariance** as a measure of joint variability of two variables:

Definition II.3.5. Covariance.

Let X and Y be random variables, then:

$$\mathrm{Cov}[X, Y] = \mathbb{E}\big[(X - \mathbb{E}[X]) \cdot (Y - \mathbb{E}[Y])\big].$$

⌟

If $\mathrm{Cov}[X, Y] = 0$, then X and Y are **uncorrelated**, which is not the same **independence**, in general.

For the single random variable X, we have $\mathrm{Var}[X] = \mathrm{Cov}[X, X]$.

Now, we discuss the conditional expectations, variances, and covariances. Let X, Y, Z be random variables, then:

$$\mathbb{E}[Y|X = x] := \begin{cases} \sum_{i=1}^{m} y_i \cdot \mathbb{P}(y_i|x), & \text{if } Y \text{ has } m \text{ values,} \\ \int y \cdot f(y|x), & \text{if } Y \text{ is real-valued.} \end{cases}$$

$$\mathrm{Var}[Y|X = x] := \mathbb{E}\big[(Y - \mathbb{E}[Y|X = x])^2 \big| X = x\big].$$

$$\mathrm{Cov}[Y, Z|X = x] := \mathbb{E}\big[(Y - \mathbb{E}[Y|X = x]) \cdot (Z - \mathbb{E}[Z|X = x]) \big| X = x\big].$$

Again, we note that the above functions have fixed output even if the variables are random.

II.4 Data as Samples

The **supervised learning** task is to learn a function $f : \vec{X} \to Y$ from a data set with input $[X]$ and output $[Y]$ of a function.

Setup II.4.1. Underlying Observed Data Distribution.

We postulate an **underlying observed data distribution** as $\mathbb{P}(x_1, \cdots, x_k, y)$ with **sample** data elements:

$$[D] := ([X], [Y]) = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1k} & y_1 \\ x_{21} & x_{22} & \cdots & x_{2k} & y_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nk} & y_n \end{pmatrix}$$

Here our study is the **ontological status** of the distribution. There could be two different possibilities:

(i) $\mathbb{P}$ is the true distribution representing uncertainty in the underlying phenomena in the world.

- This view is more on the statistics, looking for the truth.
- Typically some aspects of p, such as single parameter like $\mathbb{E}[Y]$.
- Goal is to learn about the mapping from $X_1, \cdots, X_k$ to Y from the data set $[D]$.
- The **true distribution**, denoted p_0, could naturally choose $\mathbb{E}[Y|X]$.
- The measure of success is the extent of recovering the truth.

(ii) $\mathbb{P}$ is an assumption of convenience, with is something good enough.

- Typically used in highly complex, high dimensional setting. It is unrealistic to get the truth.
- Often in machine learning and Bayesian statistics. It is not **restrictive** with assumption on p.
- The measure of success is the success in updating degrees of belief on Bayesian, or performance on a task in Machine Learning.

Typically, we do not know the distribution exactly, we only have access to the **samples**.

Definition II.4.2. Models and Hypothesis Class.

We consider a set of possible **distributions**, and we want to find either $\mathbb{P}(x_1, \cdots, x_k, y)$ itself or some part of it (such as $\mathbb{E}[Y]$, $\mathbb{E}[Y|X]$, etc.). The set of distributions is called a **statistical model**.
If the set of distribution allows any distributions, the model is **unrestricted** or **non-parametric** saturated.
If the set of distributions has up to k finite parameters, the model is a **parametric model of dimension k**.
If the model is neither unrestricted nor parametric, it is **semi-parametric**.

For example, a parametric model of dimension 2 can be:

$$\left\{ f(x) = \frac{1}{2\sigma^2\pi} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2} \right) : \text{ for all } \mu, \sigma^2 \right\}.$$

This distribution is the Gaussian distribution with **mean** and **variance** being the parameters.

Now, we consider the model $f : \vec{X} \mapsto Y$ with data $[D]$. If we are learning $\mathbb{E}[Y|X]$ from $[D]$, then our hypothesis class $\mathcal{H}$ degenerates to restrictions of $\mathbb{E}[Y|\vec{X}]$ with $\mathbb{P}(\vec{x}, y)$, which leads to semi-parametric model.

If our model is restricting $p(Y|\vec{X})$, then $\mathbb{E}[Y|\vec{X}]$ becomes a **decision tree**.

Setup II.4.3. Parametric Model.

Suppose we are learning $f : \vec{X} \to Y$ and think of the function as $\mathbb{E}[Y|\vec{X}]$ from $\mathbb{P}(\vec{x}, y)$ with a parameter of dimension k, we denote it as $\vec{\beta}$.
Then, we may denote our distribution with such parameter indexed in:

$$\mathbb{P}(\vec{x}, y; \vec{\beta}).$$

In such case, we want to choose the best $\vec{\beta}$ using the existing data in $[D]$. ⌟

The process of using data to choose model parameters is called **statistical inference**. We now want a evaluation function to tell us how **surprised** we are when seeing the data based on the current model.

II.5 Likelihood Function and Maximum Likelihood Equation

Our main goal is to have a **likelihood** function of seeing a particular row in data set $[D]$ given the current distribution model. Throughout the process, we assume that the rows in $[D]$ were independently and randomly drawn from the same distribution $\mathbb{P}(x_1, \cdots, x_k, y)$, which is **independently, identically distributions** (i.i.d.).

Definition II.5.1. Likelihood Function.

Given distribution $\mathbb{P}(\vec{x}, y; \vec{\beta})$, the probability of seeing every row of $[D]$, or the **likelihood function**, is:

$$\mathcal{L}_{[D]}(\vec{\beta}) := \prod_{i=1}^{n} \mathbb{P}(x_{i1}, \cdots, x_{ik}, y_i; \vec{\beta}).$$ ⌟

Note that we used a product since we assumed independent selections of samples, so if $A \amalg B$, then $\mathbb{P}(A, B) = \mathbb{P}(A) \cdot \mathbb{P}(B)$.

Basically, the likelihood function models the probability that we are seeing each row in $[D]$ given the parameter β, and we want to increase this number so we are as **unsurprised** as possible.

Remark II.5.2. Joint Distribution and Likelihood Function.

The **joint distribution** and **likelihood function** are different. **Joint distribution** is a function mapping $\vec{X}$ and Y to a real number, but the **likelihood function** maps the parameter value β to a real number with fixed data points. ⌟

With such property, our goal is to maximize the likelihood function (less surprise) by manipulating the values of $\vec{\beta}$. Here, we discuss the special case in which $\mathcal{L}(\vec{\beta})$ is twice differentiable with respect to $\vec{\beta}$.

Algorithm II.5.3. Second Derivative Test.

Suppose that $\mathcal{L}(\vec{\beta})$ is twice differentiable with respect to $\vec{\beta}$. From **second derivative test**, we have:

- Whenever $\frac{\partial \mathcal{L}(\vec{\beta})}{\partial \vec{\beta}} = 0$, $\mathcal{L}(\vec{\beta})$ has a critical point, *i.e.*, local maximum, local minimum, or saddle point.
- Whenever $\frac{\partial^2 \mathcal{L}(\vec{\beta})}{\partial \vec{\beta}^2} < 0$, the critical point is a maximum, *i.e.*, $\mathcal{L}(\vec{\beta})$ attains the maximum value. ⌟

Such strategy of find maximum likelihood is called a **Likelihood Based Learning**.

As an interlude, we recall the derivatives with respect to vectors in multivariable calculus.

Definition II.5.4. Gradient and Hessian Matrix.

Let f be a real-valued function that takes a vector input $\vec{x}$ of n entries, its **gradient** is defined as:

$$\nabla f = \frac{\partial f(\vec{x})}{\partial \vec{x}} = \begin{pmatrix} \partial f/\partial x_1 \\ \partial f/\partial x_2 \\ \vdots \\ \partial f/\partial x_n \end{pmatrix}.$$

Its second derivative, or the **Hessian matrix** is:

$$\mathrm{H}_f(\vec{x}) = \frac{\partial^2 f(\vec{x})}{\partial \vec{x}^2} = \begin{pmatrix} \partial^2 f/\partial x_1^2 & \partial^2 f/\partial x_1 \partial x_2 & \cdots & \partial^2 f/\partial x_1 \partial x_n \\ \partial^2 f/\partial x_2 \partial x_1 & \partial^2 f/\partial x_2^2 & \cdots & \partial^2 f/\partial x_2 \partial x_n \\ \vdots & \vdots & \ddots & \vdots \\ \partial^2 f/\partial x_n \partial x_1 & \partial^2 f/\partial x_n \partial x_2 & \cdots & \partial^2 f/\partial x_n^2 \end{pmatrix}.$$ ⌟

When having doing the **second derivative test** with vector inputs, the maximum occurs when we have the gradient being $\vec{0}$, and the Hessian matrix is **positive definite**, *i.e.*, all eigenvalues are negative.

To facilitate computation, we use the logarithm strategy. This is valid since log is a **strictly monotonic** function, *i.e.*, if $L_1 > L_2$, then $\log L_1 > \log L_2$ for any $L_1, L_2 \in \mathbb{R}$.

Definition II.5.5. The Score Equation.

By introducing the logarithmic function, we can make the optimization problem into solving the following **score equation**:

$$\frac{\partial \log \mathcal{L}_{[D](\vec{\beta})}}{\partial \vec{\beta}} = 0.$$

Then, we can also do the second derivative test for obtaining a **maximum**. ⌟

For simple models, we are able to solve the **score equation** in closed form. The solution to the score equation is called the **maximum likelihood estimate** (MLE).

Example II.5.6. Maximum Likelihood for Coin Flip.

Here, we let $\mathbb{P}(x;q)$ be a model containing a single binary variable X (0 for tails and 1 for heads), and parameter q representing $q := \mathbb{P}(X = 1)$.
Here, we suppose a set of data of n coin flips as $[D] = (x_1, x_2, \cdots, x_n)$.
Here, the log likelihood function is:

$$\log \mathcal{L}_{[D]}(q) = \log \prod_{i=1}^{n} q^{x_i} \cdot (1-q)^{1-x_i} = \sum_{i=1}^{n} \left(x_i \log q + (1-x_i) \log(1-q) \right).$$

Here, we can take the derivative in **closed form**, that is:

$$\frac{\partial \log \mathcal{L}_{[D]}(q)}{\partial q} = \sum_{i=1}^{n} \left(\frac{x_i}{q} - \frac{1-x_i}{1-q} \right) = \sum_{i=1}^{n} \left(\frac{x_i}{q} - \frac{1}{1-q} + \frac{x_i}{1-q} \right) = \frac{\#1^x_{[D]}}{q} - \frac{n}{1-q} + \frac{\#1^x_{[D]}}{1-q} = 0.$$

Therefore, we solve for q, which gives the maximum likelihood as:

$$q = \frac{\#1^x_{[D]}}{n},$$

which is exactly the possibility of heads over the sample, which is reasonably the best distribution. ⌟

Note that the *coin flipping* example is very simple and has only one parameter. For the general cases, we will experience many difficulties:

- In general, when there are many parameters, the closed form solutions will not exist.
  In these cases, we need to think of **iterative** and **local** search methods.
- For the likelihood function, it might have many maximums, and the derivative methods can only find local maximums, not identifying the **global maximums**.

Here, if we entirely believe the parametric model, we may use the maximized likelihood estimate to form $\mathbb{E}[Y|X]$.

II.6 Bayesian Inference

Rather than just obtaining the true model, the **Bayesian inference** starts with an **opinion**, and then use the data to **revise** the opinion systematically.

Setup II.6.1. Prior and Posterior Distribution.

The starting opinion is the **prior distribution**. It can be thought of as assigning probabilities to elements in a statistical model. Here, we can have the model $\{\mathbb{P}(\vec{x}, y; \vec{\beta}); \vec{\beta}\}$, which is a prior distribution of $\mathbb{P}_{\text{prior}}(\vec{\beta})$ over $\vec{\beta}$.
Given data $[D]$ represented by likelihood function $\mathcal{L}_{[D]}(\vec{\beta})$, we use **Bayes' rule** to yield the **posterior distribution**:

$$\mathbb{P}_{\text{posterior}}(\vec{\beta}|[D]) = \begin{cases} \dfrac{\mathcal{L}_{[D]}(\vec{\beta}) \cdot \mathbb{P}_{\text{prior}}(\vec{\beta})}{\sum_{\vec{\beta}} \mathcal{L}_{[D]}(\vec{\beta}) \cdot \mathbb{P}_{\text{prior}}(\beta)}, & \text{for a finite set of } \vec{\beta}, \\ \dfrac{\mathcal{L}_{[D]}(\vec{\beta}) \cdot \mathbb{P}_{\text{prior}}(\vec{\beta})}{\int \mathcal{L}_{[D]}(\vec{\beta}) \cdot \mathbb{P}_{\text{prior}}(\beta) d\vec{\beta}}, & \text{for an infinite set of } \vec{\beta}, \end{cases}$$

⌟

A **Bayesian inference** will generally return the posterior, which revises the opinion on how likely each one is. However, for most problems, we want to **particular discrete decision**.

Algorithm II.6.2. Maximum a Posteriori.

The common principle to choose a model with parameters $\vec{\beta}$ is by **maximum a posteriori** (MAP), that is the parameter that maximize the posterior:

$$\arg\max_{\beta} \mathbb{P}_{\text{posterior}}(\vec{\beta}|[D]).$$ ⌟

In particular, if we calculate $\mathbb{P}_{\text{posterior}}(\vec{\beta}|[D])$ explicitly, we can choose $\vec{\beta}$ by taking derivatives $\frac{\partial \mathbb{P}_{\text{posterior}}(\vec{\beta}|[D])}{\partial \vec{\beta}}$. When there are finitely many $\vec{\beta}$, we simply choose the one with the largest probability.

Remark II.6.3. Conjugate Priors.

Suppose that we are in infinitely many cases, the prior and posterior will both be density functions. Since we are reusing posteriors as new priors, it would be helpful if both prior and posterior had the same form. This is called **conjugate prior**, which examines the form of the distribution in likelihood. ⌟

Likewise, we can consider the example of flipping a coin as a conjugate prior.

Example II.6.4. Flipping a Coin with Conjugate Prior.

Suppose a coin X is flipped n times, in which the probability of head if q, *i.e.*, $\mathbb{P}(X = 1) = q$. Here, the probability of having k heads is:

$$\mathbb{P}(k) = \binom{n}{k} q^k (1-q)^{n-k} = \frac{n!}{k!(n-k)!} q^k (1-q)^{n-k}.$$

Therefore, the prior distribution, after normalization is:

$$\mathbb{P}_{\text{prior}}(q) = \frac{q^{\alpha-1}(1-q)^{\beta-1}}{B(\alpha,\beta)} = \frac{q^{\alpha-1}(1-q)^{\beta-1}}{\int_0^1 q^{\alpha-1}(1-q)^{\beta-1} dq}.$$

Suppose that there are k heads and $n-k$ tails, then the likelihood function is:

$$\mathcal{L}_{[D]}(q) = \binom{n}{k} q^k (1-q)^{n-k}.$$

Hence, we then update the posterior, so we have:

$$\mathbb{P}_{\text{posterior}}(q) = \frac{\mathcal{L}_{[D]}(q) \cdot \mathbb{P}_{\text{prior}}(q)}{\int \mathcal{L}_{[D]}(q) \cdot \mathbb{P}_{\text{prior}}(q) dq} = \frac{q^{k+\alpha-1}(1-q)^{n-k+\beta-1}}{B(k+\alpha, n-k+\beta)}.$$

By this update, it is assuming that we started with $\alpha - 1$ heads and $\beta - 1$ tails. ⌟

Here, we also observe that the conjugate prior have **one larger dimension** compared to the likelihood model. Conjugate prior works with many parametric models, thus allowing **closed form calculation** of the posterior.

Of course, when the modeling has no close form, we can still grasp some data of the posterior, such as $\mathbb{E}_{\mathbb{P}_{\text{posterior}}}[Y]$, the median values, or even MAP values of $\vec{\beta}$.

II.7 Gibbs Sampling

A powerful set of techniques is called **Markov Chain Monte Carlo** (MCMC), which is about drawing samples from complex distributions that cannot be evaluated directly.

Setup II.7.1. Optimize Using Independence.

Suppose we have a set of random variables, if we have:

$$X_i \amalg X_{j_1}, X_{j_2}, \cdots, X_{j_m} | X_{k_1}, X_{k_2}, \cdots, X_{k_\ell},$$

we can reduce the conditional distribution as:

$$\mathbb{P}(x_i | x_{j_1}, \cdots, x_{j_m}, x_{k_1}, \cdots, x_{k_\ell}) = \mathbb{P}(x_i | x_{j_1}, \cdots, x_{j_m}).$$ ⌟

A more simple algorithm of MCMC is the **Gibbs sampling**, which draws samples from $p(x_1, \cdots, x_n)$.

Algorithm II.7.2. Gibbs Sampling Algorithm.

First, we initialize $X_1, \cdots, X_n$ to arbitrary starting values $x_{10}, x_{20}, \cdots, x_{n0}$.
We repeat for $i = 1, 2, \cdots, s$ times, where we:

- Pick X_j in which we denote X_{-j} as all other variables.
- Let x_{ji} be a sample from $\mathbb{P}(x_j | x_{-j,i-1})$, where $x_{-j,i-1}$ are the values of X_{-j} at the previous step.

This above algorithm can be repeated for long enough so we get a sample from the joint distribution. ⌟

One advantage is that by accounting for independence, we can reduce the joint data size as $2^n - 1$ into a smaller number as the **sum of the conditionals**. This prevents the model from getting into **too high a dimension**.

Moreover, it is sometimes hard to represent the joint but the conditionals are easier.

Even when we do not have access to $\mathbb{P}_{\text{posterior}}(\vec{\beta})$, as long as we know the ratio $\mathbb{P}_{\text{posterior}}(\vec{\beta}_1)/\mathbb{P}_{\text{posterior}}(\vec{\beta}_2)$, we can pick a proposal distribution $Q(\vec{\beta}_1 | \vec{\beta}_2)$, which is easier to work with. This is a more generalized **Metropolis-Hastings Algorithm**. The **Gibbs Sampling Algorithm** is a special case of this.

III Regression Models, Selections, and Regularization

III.1 General Regression Models

The **regression models** are different from the learning function, we are trying to minimize the squared loss function, that is $\mathbb{E}[(Y - f(\vec{X}))^2]$, and the minimum choice is the regression function $\mathbb{E}[Y | \vec{X}]$.

Overall, the model is **semi-parametric**, since there are no restrictions on $p(\vec{x})$, but we want the model to be finitely monitored. Two special cases are **linear regression** for real-valued Y and **logistic regression** on

binary Y.

Theorem III.1.1. Property of True Distribution.

Suppose $\mathbb{P}_0$ is the **true distribution**. Then, let $S_{\mathbb{P}}(\vec{\beta})$ denote the model score at $\mathbb{P}$, that is:

$$S_{\mathbb{P}}(\beta) := \frac{\partial \log \mathcal{L}_{\mathbb{P}}(\vec{\beta})}{\partial \vec{\beta}}.$$

The expectation of the score is zero, *i.e.*:

$$\mathbb{E}_{\mathbb{P}_0}[S_{\mathbb{P}_0}(\vec{\beta_0})] = 0.$$

Such result is a direct consequence of Leibniz integral rule and arithmetic manipulations. ⌟

Such property hold true only if we have the true $\mathbb{P}_0$, this underlies why solving the score equation for MLE works. When we don't have the true $\mathbb{P}_0$, we need to approximate it using **empirical distributions**.

The general class of regressions is embedded in the **restricted moment model**.

Definition III.1.2. Restricted Moment Model.

The **restricted moment model** is on $\mathbb{P}(\vec{X}, Y)$ such that $\mathbb{E}[Y|X] = g(\vec{X}; \vec{\beta})$ is a fixed function. We define the true result $Y = g(\vec{X}; \vec{\beta}) + \epsilon$, where we restrict ϵ to be:

$$\mathbb{E}[\epsilon|\vec{X}; \vec{\eta}] = 0,$$

where ϵ is characterized as the disturbance, and $\vec{\eta}$ are some parameters that do not overlap with $\vec{\beta}$ that we do not care about (called **nuisance parameters**). There could be as many of them. ⌟

III.2 Linear Regression Model

A special case is the **linear regression**.

Setup III.2.1. Linear Regression Model.

Here, we assume $[D]$ has real-valued outcome Y. Moreover, the model assumes $Y = g(\vec{x}, \vec{\beta}) + \epsilon$, where:

$$g(\vec{x}; \vec{\beta}) := \beta_0 + \sum_{j=1}^{k} x_j \beta_j \qquad \text{and} \qquad \epsilon \sim \mathcal{N}(0, \sigma^2).$$

This model is known as a restricted moments model, since ϵ is independent of $\vec{x}$, so we have:

$$\mathbb{E}[\epsilon|\vec{X}] = \mathbb{E}[\epsilon] = 0.$$ ⌟

Remark III.2.2. Likelihood of Linear Regression Model.

Here, the model likelihood is:

$$\mathcal{L}_{[D]}(\vec{\beta}) = \prod_{i=1}^{n} \mathbb{P}(\vec{x_i}) \cdot \frac{1}{\sqrt{2\sigma^2\pi}} \exp\left(- \frac{\left(y_i - (\beta_0 + \sum_{j=1}^{k} \beta_j x_{ij})\right)^2}{2\sigma^2} \right).$$

Moreover, we calculate the log likelihood as:

$$\log \mathcal{L}_{[D]}(\vec{\beta}) = \sum_{i=1}^{n} \log \frac{\mathbb{P}(\vec{x}_i)}{\sqrt{2\sigma^2\pi}} + \sum_{i=1}^{n} \left(-\frac{\left(y_i - (\beta_0 + \sum_{j=1}^{k} \beta_j x_{ij})\right)^2}{2\sigma^2} \right). \qquad \lrcorner$$

Recall that our goal is to maximize the log likelihood with respect to $\vec{\beta}$, which is equivalently minimizing $\sum_{i=1}^{n} \left(y_i - (\beta_0 + \sum_{j=1}^{k} \beta_j x_{ij})\right)^2$.

To deal with β_0, we add a columns of 1 to the left of $[X]$, which yields $[\tilde{X}]$, and the model is then:

$$\begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} 1 & x_{11} & \cdots & x_{1k} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \cdots & x_{nk} \end{pmatrix} \cdot \begin{pmatrix} \beta_0 \\ \vdots \\ \beta_k \end{pmatrix} + \begin{pmatrix} \epsilon_1 \\ \vdots \\ \epsilon_n \end{pmatrix},$$

which is expressed as:

$$\begin{array}{ccccccc} [Y] & = & [\tilde{X}] & \cdot & \vec{\beta} & + & \vec{\epsilon}. \\ \cap & & \cap & & \cap & & \cap \\ \mathbb{F}^{n\times 1} & & \mathbb{F}^{k\times(k+1)} & & \mathbb{F}^{(k+1)\times 1} & & \mathbb{F}^{n\times 1}. \end{array}$$

Then, we choose $\vec{\beta}$ to minimize:

$$([Y] - [\tilde{X}] \cdot \vec{\beta})^\intercal \cdot ([Y] - [\tilde{X}] \cdot \vec{\beta}) = \vec{\epsilon}^\intercal \cdot \vec{\epsilon} = \|\vec{\epsilon}\|^2.$$

Here, we are minimizing the square error, which is nonnegative. In fact, we may find the closed form as:

$$\begin{aligned} ([Y] - [\tilde{X}] \cdot \vec{\beta})^\intercal \cdot ([Y] - [\tilde{X}] \cdot \vec{\beta}) &= ([Y]^\intercal - \vec{\beta}^\intercal \cdot [\tilde{X}]^\intercal) \cdot ([Y] - [\tilde{X}] \cdot \vec{\beta}) \\ &= [Y]^\intercal [Y] - 2[Y]^\intercal [\tilde{X}]\vec{\beta} + \vec{\beta}^\intercal [\tilde{X}]^\intercal [\tilde{X}]\vec{\beta}. \end{aligned}$$

Here, we do the derivative with respect the $\vec{\beta}$ so we can solve for $\vec{\beta}$ as long as the inverses exists, which is:

$$0 = -2[\tilde{X}]^\intercal [Y] + 2[\tilde{X}]^\intercal [X]\vec{\beta},$$

which is $[\tilde{X}]^\intercal [\tilde{X}]\vec{\beta} = [\tilde{X}]^\intercal [Y]$.

Proposition III.2.3. Optimal Parameter for Linear Regression.

The MLE for square error for linear regression is with:

$$\vec{\beta} = ([\tilde{X}]^\intercal [\tilde{X}])^{-1} [\tilde{X}]^\intercal [Y]. \qquad \lrcorner$$

Note that the linear regression is inherently a line, and we can technically achieve **polynomial regressions** as well, which is **higher order**.

For example, we may trivially form the quadratic as:

$$Y = \beta_{\text{int}} + \vec{\beta}_X X + \beta_{X^2} \vec{X}^2.$$

Setup III.2.4. Polynomial Regression.

In terms of polynomial regression, we can add columns of $\vec{X}^2$ vertically on our data, so the new matrix is:

$$\tilde{X} = \begin{pmatrix} X & X^2 & \cdots \end{pmatrix}.$$

And the model is still a linear regression model. $\lrcorner$

The above model seems to handle the case, but it incurs issues with **interaction effects**, *i.e.*, we cannot assume that X and X^2 are independent, and they are in fact not independent.

Therefore, we want to have a simple modification to add the columns of the multiplication of data, so the quadratic model becomes:

$$Y = \beta_{\text{int}} + \vec{\beta}_X X + \beta_{X^2} X^2 + \beta_{XX^2} X X^2 + \epsilon.$$

In particular, this model can be generalized into any model with interacting factors.

Setup III.2.5. Regression Model with Interacting Factor.

Suppose X_1 and X_2 are factor with interactions, and let $X_1 X_2$ denote the multiplication of the data, the model becomes:

$$Y = \beta_{\text{int}} + \vec{\beta}_1 X_1 + \beta_2 X_2 + \beta_{12} X_1 X_2 + \epsilon.$$

Here, β_1 and β_2 are **main effects**, and β_{12} is the **interaction effect**. ⌟

In practice, the interaction effects are often smaller than the main effects, known as the **sparsity of effects principle**.

The main concern is to build complex models with linear pieces, there are multiple ways of approaching that:

(i) We may modify the input end by adding interesting or complicated features, such as **interactions** or **general pre-processing**.
Hence, we will be able to craft complex surfaces by a piecewise linear model. The way of achieving this is by having **indicator features** that corresponds to intervals over the x-axis.

(ii) On the output end, we may add non-line transformation, *i.e.*, link functions.
For example, we can feed a **linear function** into an **exponential**, namely $\exp(\beta_{\text{int}} + \beta_X X)$.

(iii) Also, we can feed the output of one into the input of another, which is an application of the **neural network**.

One specific non-linear transformation of a linear model is the **sigmoid function**.

III.3 Logistic Regression and the Newton-Raphson Method

For logistic regression, we use the **sigmoid function** to create the distribution. Here, we assume $[D]$ has binary outcome Y.

Definition III.3.1. Sigmoid Function.

The model assume that the probability of having 1 as output is:

$$\mathbb{P}(Y = 1 | \vec{X}; \vec{\beta}) = \frac{1}{1 + \exp\left[-\beta_{\text{int}} - \left(\sum_{i=1}^{k} x_i \cdot \beta_i\right)\right]} = \frac{1}{1 + \exp\left(-\sum_{i=0}^{k} x_i \cdot \beta_i\right)}.$$ ⌟

Note that we are given a conditional expectation, it is a **regression model**.

Moreover, the model is **restricted** since we assume that $\mathbb{E}[\epsilon|X] = 0$.

Remark III.3.2. Likelihood Function.

Note that the likelihood function is:

$$\mathcal{L}_{[D]}(\vec{\beta}) = \left(\prod_{i=1}^{n} \mathbb{P}(\vec{x}_i)\right) \left(\prod_{i:y=1} \frac{1}{1+\exp\left(-\sum_{j=1}^{k} x_i \cdot \beta_j\right)}\right) \left[\prod_{i:y_i=0} \left(1 - \frac{1}{1+\exp\left(-\sum_{j=0}^{k} x_i \cdot \beta_j\right)}\right)\right].$$

Correspondingly, the log likelihood is:

$$\begin{aligned}
&\log \mathcal{L}_{[D]}(\vec{\beta}) \\
&= \sum_{i=1}^{n} \log \mathbb{P}(\vec{x}_i) + \sum_{i:y_i=1} \log\left(\frac{1}{1+\exp\left(-\sum_{j=0}^{k} x_i \cdot \beta_j\right)}\right) + \sum_{i:y_i=0} \log\left(1 - \frac{1}{1+\exp\left(-\sum_{j=0}^{k} x_i \cdot \beta_j\right)}\right) \\
&= \sum_{i=1}^{n} \log \mathbb{P}(\vec{x}_i) - \sum_{i:y_i=1} \log\left[1+\exp\left(-\sum_{j=0}^{k} x_i \cdot \beta_j\right)\right] + \sum_{i:y_i=0} \log\left[\frac{\exp\left(-\sum_{j=0}^{k} x_i \cdot \beta_j\right)}{1+\exp\left(-\sum_{j=0}^{k} x_i \cdot \beta_j\right)}\right].
\end{aligned}$$

Here, we are trying to maximize $\log \mathcal{L}_{[D]}(\vec{\beta})$, which is solving for zero for the derivative. We note that the derivative with respect to each entry is:

$$\frac{\partial \log \mathcal{L}_{[D]}(\vec{\beta})}{\partial \beta_j} = \sum_{i=1}^{n} x_{ij} \cdot \left(y_i - \mathbb{P}(y_i = 1|\tilde{x}_i; \vec{\beta})\right) = 0.$$

Note that this equation is **transcendental**, so it cannot be solved in **closed form**. Thus, we move to **iterative methods** using local search.

Here, we make the assumption of **smooth function** ($\mathcal{C}^\infty$), in which we can apply the **Taylor expansion**.

Theorem III.3.3. The Taylor Expansion.

Let a function $g(x)$ to be defined with all derivative at x_0, *i.e.*, $g(x) \in \mathcal{C}^\infty(x_0)$, it may be written as an infinite polynomial:

$$g(x) \sim \sum_{n=0}^{\infty} \frac{g^{(n)}(x_0)}{n!}(x-x_0)^n = g(x_0) + \frac{g'(x_0)}{1!}(x-x_0) + \frac{g''(x_0)}{2!}(x-x_0)^2 + \cdots.$$

Such function is very useful as **polynomials** have nice behavior. Often $g(x)$ could be complicated, but approximating to the first few terms could be a good **approximation**. One application of the **Taylor expansion** is finding the root of a smooth function $g(x)$.

Algorithm III.3.4. The Newton-Raphson Method.

Let $g(x)$ be a function that is smooth, *i.e.*, $g(x) \in \mathcal{C}^\infty(\mathbb{R})$. We denote x^* to be such that $g(x^*) = 0$, which is our goal of finding.

- We start with a guess x_0, which is almost certain that $g(x_0) \neq 0$.

- Then, we form the Taylor expansion around x_0 to approximate $g(x)$ near x_0:
$$g(x) \sim g(x_0) + g'(x_0) \cdot (x - x_0) + \frac{g''(x_0)}{2} \cdot (x - x_0)^2 + \cdots .$$
In particular, we can take the first two terms to approximate $g(x)$ by:
$$\tilde{g}(x) = g(x_0) + (x - x_0) \cdot g'(x_0).$$
- Then, we *hope* that the root of $\tilde{g}(x)$ will be a better guess than x_0 for the root of $g(x)$. We solve for the root of $\tilde{g}(x)$, that is:
$$x = x_0 - \frac{g(x_0)}{g'(x_0)}.$$

Then, we repeat the above steps with $x_1, x_2, \cdots$. If we have $x_i = x_0$ at step i, we stop the process, else we stop as long as $g(x_i) \simeq 0$. ⌟

With such the **Newton-Raphson Method**, we can apply it onto the process of maximizing $\log \mathcal{L}_{[D]}(\vec{\beta})$. Here, we are looking for the root for $\frac{\partial \log \mathcal{L}_{[D]}(\vec{\beta})}{\partial \vec{\beta}}$.

In particular, the derivative $\frac{\partial \log \mathcal{L}_{[D]}(\vec{\beta})}{\partial \vec{\beta}}$ is telling us the **direction** in which we need to update the guess , whereas $\frac{1}{\frac{\partial^2 \log \mathcal{L}_{[D]}(\vec{\beta})}{\partial^2 \beta_j}}$ is telling us the magnitude of the step we need to take in the chosen direction. Hence, such method updates the **guess on the parameter** using the derivative of the function that we are **optimizing**.

Recall that we have define the equivalent of derivatives and second derivatives in multivariable calculus. Hence, we can also use the **Newton-Raphson Method** to approximate the root.

Algorithm III.3.5. Newton-Raphson Method for Multivariable Case.

Here, we use H as the score function that we are trying to maximize, whose gradient is ∇h and Hessian matrix is H_h, then we denote $\vec{\beta}^*$ as the **prior guess**, the **updated guess** is then:
$$\vec{\beta} = \vec{\beta}^* - \mathrm{H}_h(\vec{\beta}^*)^{-1} \cdot \nabla h(\vec{\beta}^*).$$
Hence, it is the same process for multivariate variables:

(i) Make a guess with $\vec{\beta}_0$, which is unlikely to be the root.

(ii) Update $\vec{\beta}_i$ from $\vec{\beta}_{i-1}$ using the update rule, and

(iii) Stop when $\vec{\beta}_i$ and $\vec{\beta}_{i+1}$ are **sufficiently closed**. ⌟

Just to note, the **gradient** for Multivariate Logistic Regression is:
$$\nabla h = \frac{\partial h(\vec{\beta})}{\partial \vec{\beta}} = \begin{pmatrix} \partial h/\partial \beta_1 \\ \vdots \\ \partial h/\partial \beta_k \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^n \vec{x_{i1}} \cdot \left(y_i - \mathbb{P}(y_i = 1 | \vec{x_i}; \vec{\beta})\right) \\ \vdots \\ \sum_{i=1}^n \vec{x_{ik}} \cdot \left(y_i - \mathbb{P}(y_i = 1 | \vec{x_i}; \vec{\beta})\right) \end{pmatrix}$$

Likewise, the **Hessian matrix** is:

$$\mathrm{H}_h(\vec{\beta}) = \frac{\partial^2 h(\vec{\beta})}{\partial \vec{\beta}^2} = \begin{pmatrix} \partial^2 h/\partial x_1^2 & \cdots & \partial^2 h/\partial x_1 \partial \beta_k \\ \vdots & \ddots & \vdots \\ \partial^2 h/\partial x_n \partial x_1 & \cdots & \partial^2 h/\partial \beta_k^2 \end{pmatrix}$$
$$= \begin{pmatrix} \sum_{i=1}^n x_{i1}x_{i1}\mathbb{P}(y_i=1|\tilde{x}_i;\vec{\beta})\mathbb{P}(y_i=0|\tilde{x}_i;\vec{\beta}) & \cdots & \sum_{i=1}^n x_{i1}x_{ik}\mathbb{P}(y_i=1|\tilde{x}_i;\vec{\beta})\mathbb{P}(y_i=0|\tilde{x}_i;\vec{\beta}) \\ \vdots & \ddots & \vdots \\ \sum_{i=1}^n x_{ik}x_{i1}\mathbb{P}(y_i=1|\tilde{x}_i;\vec{\beta})\mathbb{P}(y_i=0|\tilde{x}_i;\vec{\beta}) & \cdots & \sum_{i=1}^n x_{ik}x_{ik}\mathbb{P}(y_i=1|\tilde{x}_i;\vec{\beta})\mathbb{P}(y_i=0|\tilde{x}_i;\vec{\beta}) \end{pmatrix}.$$

Remark III.3.6. Limitations with Newton-Raphson Method.

The Newton-Raphson Method, at the same time it incurs the following limitations:

- Such methods are **local**, so it will only find a root of the score equation h.
- The result from the algorithm could be a maximum, minimum, or saddle point since the we only had the first derivative as zero. Otherwise, we would need to do the second derivative test.
- Moreover, since the search is local, even if it is local maximum, it is not necessarily a global maximum.

Hence, the algorithm could find a very **bad solution**m but there are no general solutions. ⌟

However, there are some tricks that could address for this.

Remark III.3.7. Properties for Convex and Concave Function.

A function $g(x)$ is **convex** if for ay x_1, x_2 and $0 \leq t \leq 1$, we have:

$$g\big(tx_1 + (1-t)x_2\big) \leq tg(x_1) + (1-t)g(x_2).$$

A function $g(x)$ is **concave** if $-g(x)$ is convex.

Figure III.1. Convex function (left) and concave function (right).

A **concave** function has a unique maximum and a **convex** function has a unique minimum. This makes them nice to maximize or minimize, in particular for the local search process. ⌟

Some likelihood functions are convex, so it might not lead to a good local search. The **logistic regression** likelihood is **concave** in every parameter, so it will work well.

III.4 Vapnik's Principle and Loss Function

Previously, we have had the assumption that we have a **parametric model** on $\mathbb{P}(x_1, \cdots, x_k, y; \vec{\beta})$ and $\mathcal{L}_{[D]}(\vec{\beta})$. Consequently, we assume that minimizing square loss $\mathbb{E}[\big(Y - f(\vec{X})\big)^2]$ is the equivalent to learning $\mathbb{E}[Y|\vec{X}]$ by maximizing the **likelihood** in this case. However, it is unrealistic to expect a **parametric**

likelihood.

At the same time, we conform to **Vapnik's Principle**.

Proposition III.4.1. Vapnik's Principle.

When solving a problem of interest, do not solve a more general problem as an intermediate step. ⌟

Hence, we want to think about **function learning** directly as a problem. Basically, our goal is:

$$\text{Learn function } f : \vec{X} \mapsto Y \text{ by minimizing a loss } L\big(Y, f(\vec{X})\big).$$

Therefore, the main problem is how to pick a loss, and picking the **right loss function** is the key for solving the problem.

For example, when we use the loss as $|Y - f(\vec{X})|$, it is the conditional median models, which is robust to the **outliers**, and we can think of **other losses** that are easier to work with. Some desirable properties are **differentiability**, **tractability to minimize**, etc.

Now our problem becomes, what are possible loss functions that we shall use. Previously, we had the square error $\big(Y - f(\vec{X})\big)^2$, 0-1 loss, approximations, or negative log likelihood as a loss function. The **loss function** is the key of telling how good the job the **learning function** does.

Remark III.4.2. Ability to Generalize for Model.

Moreover, we want to check how the learned function $\hat{f}$ generalizes, *i.e.*, how it does on data that it has not seen before. ⌟

For any reasonable loss, there are infinitely many $\hat{f}$ that minimize the average loss:

$$\frac{1}{n}\sum_{i=1}^{n} L(Y_i, \hat{f}(\vec{X}_i)) \text{ on } [D],$$

and they all generalize differently. Recall that we can have a **almost zero** model by simply remembering all data $[D]$. Hence, the focus shall be how will they **generalizes**.

Remark III.4.3. Inductive Biases from Generalization.

Give our situation that there are infinitely many $\hat{f}$ that minimize the average loss, **inductive bias** is a choice among which possible f commits to dealing with unseen data in a particular way.
To see how effective the **inductive bias choice** is, we estimate the **test error**:

$$\text{Err}_{\text{test}} := \mathbb{E}_{\mathbb{P}_0(\vec{X},Y)}\big[L\big(Y, \hat{f}(\vec{X})\big)\big|[D]\big].$$

Here, $[D]$ and $\hat{f}$ are fixed, so the expectation is with respect to $\mathbb{P}(\vec{x}, y)$, which is the **unseen** instances of the data. ⌟

Confusingly, we note that the test error is defined holding a particular set of new inputs x_0, y_0, and averaging over possible $[D]$ as an impact. To conduct such analysis, we use the **mean squared error** (MSE).

Definition III.4.4. Mean Squared Error for Prediction.

Let $\hat{f}_{[D]}$ be the predicted function with data $[D]$, the definition of test error is the **mean squared error**, that is:

$$\text{MSE} = (\vec{x_0}, y_0) = \mathbb{E}_{\mathbb{P}([D])}\left[\left(y_0 - \hat{f}_{[D]}(\vec{x_0})\right)^2\right].$$

Recall that it is only evaluating the **unseen** data set. ⌟

The MSE tried to tell what might fo wrong when a function learner ha a specific goal (such as predicting the **ground truth** y_0) given different sorts of data.

Proposition III.4.5. Bias Variance Decomposition of MSE.

For a prediction model, two type of things can go wrong. We might be **systematically wrong** (bias) or we might be paying attention to random **noise in the data** rather than a true patter (variance). Statistically, we can decompose MSE as follows:

$$\mathbb{E}_{\mathbb{P}([D])}\left[\left(y_0 - \hat{f}_{[D]}(\vec{x_0})\right)^2\right] = \underbrace{\text{Var}_{[D]}\left[\hat{f}_{[D]}(\vec{x_0})\right]}_{\text{variance}} + \underbrace{\left(\mathbb{E}_{\mathbb{P}([D])}\left[y_0 - \hat{f} + [D](\vec{x_0})\right]\right)^2}_{\text{square bias}}.$$

⌟

The *proof* involves quadratic formula, linearity of expectation, fixed y_0, and $\text{Var}(A) = \mathbb{E}[A^2] - \mathbb{E}[A]$. It is omitted for its lengthiness.

In particular, the effective generalization to unseen data involves picking a **hypothesis class** that is flexible but not too flexible. If we are **not flexible** enough, we will get excessive MSE due to **bias**, if we are **too flexible**, we will get excessive MSE due to **variance**.

Recall the performance on training data is not effective at all for the nature that models can become (almost) perfect. Instead, we must estimate the **measure the accuracy** of generalization and minimize it by making the trade off between **bias** and **variance**.

III.5 Function Learning and Optimism

Setup III.5.1. Process of Function Learning.

In describing a learning problem, it is typically in three distinct phases:

(i) **Model selection**: Make a trade off in **model complexity**, that is, use **domain insights grounds**, **data**, or both as guides to select a model (and corresponding hypothesis class $\mathcal{H}$).

(ii) **Statistical inference**: Given a model, we learn its parameters with data.

(iii) **Test check**: Given a learned element of the model that minimizes some objectives, we check how well it does on **unseen data**.

⌟

Especially when the amount of data is abundant, we split the data set into $[D]$, the **training set** for step (ii), $[V]$, the **validation set** for step (i), and $[T]$, the **test set** for step (iii).
When there are not so many data, we can combine $[D]$ and $[V]$ in the first two steps, what we end up with is to **penalize the flexibility** of the model so we may obviate the need for $[V]$.

For the last criterion, we measure the accuracy of **generalization**. In particular, a good estimation is:

$$\mathrm{Err} := \mathbb{E}[\mathrm{Err}_{\text{test}}] = \mathbb{E}_{\mathbb{P}([D])}\big[\mathbb{E}_{\mathbb{P}_0(\vec{X},Y)}\big[L\big(Y,\hat{f}(\vec{X})\big)\big|[D]\big]\big].$$

It seems that the natural choice is to use the error in $[D]$. However, it is not the case, since we cannot use the same data for both **training** and **testing data**. It is trivial to construct functions that will perform (almost) perfectly on the training data. However, most of them does not **generalize** well.

Moreover, we can develop a process to **simultaneously** process the generalization issues.

Definition III.5.2. In-Sample Error.

As a intermediate step, we define the **in-sample error** as:

$$\mathrm{Err}_{\text{in}} = \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}\big[L\big(y_i^0,\hat{f}(\vec{x}_i)\big)\big|[D]\big].$$

In other words, we draw a new outcome from each $\vec{x}_i$ in $[D]$ and compare it to the predicted output. ⌟

In this definition, we have Y_i^0 not necessarily identical with Y_i in $[D]$, and we consider Y_i^0 as random quantities. Since $\vec{x}_i$ are fixed, the expectation is with respect to all Y_i^0.

Definition III.5.3. Optimism of Data Set.

Therefore, we define the **optimism** at $[D]$ as the difference between training error and in-sample error for a particular training set $[D]$:

$$\mathrm{Op}_{[D]} := \mathrm{Err}_{\text{in}} - \mathrm{Err}_{\text{train}} = \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}\big[L\big(y_i^0,\hat{f}(\vec{x}_i)\big)\big|[D]\big] - \frac{1}{n}\sum_{i=1}^{n}L\big(y_i,\hat{f}(\vec{x}_i)\big).$$ ⌟

Optimism tries to quantify how much the model tries to take credit for the accident of seeing particular y_i for each $\vec{x}_i$ at every row (versus the other outcomes it could have seen).

In practice, it is much easier to **estimate** the **average optimism**, which averages over possible training sets we may see:

$$\omega := \mathbb{E}[\mathrm{Op}_{[D]}] = \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}\big[\mathbb{E}\big[L\big(y_i^0,\hat{f}(\vec{x}_i)\big)\big|[D]\big]\big] - \frac{1}{n}\sum_{i=1}^{n}L\big(y_i,\hat{f}(\vec{x}_i)\big).$$

For this loss, we can represent it as the sum of covariance.

Proposition III.5.4. Average Optimism as Sum of Covariance.

The **average optimism** can be shown to be related as average of covariance of the **prediction** and **truth**:

$$\omega = \frac{2}{n}\sum_{i=1}^{n}\mathrm{Cov}\left(\hat{f}(x_i),y_i\right).$$

Hence, by having that:

$$\mathbb{E}[\text{Err}_{\text{in}}] = \mathbb{E}[\text{Err}_{\text{train}}] + \mathbb{E}[\text{Op}_{[D]}],$$

which leads to the conclusion that:

$$\frac{1}{n}\sum_{i=1}^{n}\mathbb{E}_{\mathbb{P}([D])}\left[\mathbb{E}\left[L(Y_i^0, \hat{f}(\vec{x_i}))\middle|[D]\right]\right] = \frac{1}{n}\sum_{i=1}^{n}L\big(y_i, \hat{f}(\vec{x_i})\big) + \frac{2}{n}\sum_{i=1}^{n}\text{Cov}\left(\hat{f}(x_i), y_i\right). \qquad \lrcorner$$

With such model, we may attempt estimate the **in-sample error** through the covariance.

Example III.5.5. Estimating Average Optimism and In-sample Error.

Suppose $Y = f(X) + \epsilon$, where $f(X)$ is a linear function with d inputs, we have:

$$\frac{n}{2}\omega = \frac{n}{2}\mathbb{E}\left[\text{Op}_{[D]}\right] = \sum_{i=1}^{n}\text{Cov}\left(\hat{f}(x_i), y_i\right) = d\sigma_\epsilon^2 := 2\,\text{Var}(\epsilon).$$

Therefore, we have:

$$\mathbb{E}[\text{Err}_{\text{in}}] = \mathbb{E}[\text{Err}_{\text{train}}] + 2\cdot\frac{d}{2}\sigma_\epsilon^2. \qquad \lrcorner$$

If we are interested in a particular loss, we can select a model by minimizing the estimated in-sample error as:

$$\frac{1}{n}\sum_{i=1}^{n}L\big(y_i, \hat{f}(\vec{x_i})\big) + 2\cdot\frac{d}{n}\hat{\sigma_\epsilon}^2.$$

This can be thought of as modifying the loss by a factor having to do with the dimension of the model, *i.e.*, the number of parameters, d.

III.6 Information Criteria for Model Selection

The same idea applies to the negative log likelihood loss, which leads to the **information criteria** for model selection.

Definition III.6.1. Information Criterion.

The **Akaike information criterion** is given by:

$$AIC = -2\log\mathcal{L}_{[D]}(\hat{\vec{\beta}}) + 2\cdot d.$$

where $\hat{\vec{\beta}}$ is the maximum likelihood estimate.
Moreover, if we choose to choose complex penalty parameter d using the data **adaptively**, it yields:

$$AIC(\alpha) = -2\log\mathcal{L}_{[D]}(\hat{\vec{\beta}}) + 2d(\hat{\alpha}).$$

Likewise, we can define the **Bayesian information criterion** with different penalization weights.

$$BIC = -2\log\mathcal{L}_{[D]}(\hat{\vec{\beta}}) + (\log n)\cdot d. \qquad \lrcorner$$

Here, BIC penalizes model dimension more heavily, *i.e.*, by a log of the sample size. In particular, BIC is motivated by **Bayesian considerations**, that is, it is proportional to a model posterior with a particular simplicity prior.

Moreover, BIC is **consistent** for model section, *i.e.*, as sample size $n \to \infty$, it will select the true model from a set as long as the set contains the truth.

Remark III.6.2. Generalizing Optimism.

Recall that **closed form** notation of optimism relied on the assumption that there is a **fixed number of parameters**. It is not as easy to generalize the complex predictor models.

This measure is called the **Vapnik-Chervonenkis dimension** of a hypothesis class. ⌟

Therefore, we think of some ways that we can **estimate test error**. The easiest is by **sample splitting**.

Setup III.6.3. Cross-Validation for Estimating Test Error.

We partition $[D]$ into k subsets of size n/k called **folds**. We train the model on all but the ith hold, to yield $\hat{f}^{(-i)}$. There, we estimate the generalization treating all but ith fold as test error:

$$\text{Err}_i = \frac{1}{n/k} \sum_{j=1}^{n/k} L\big(y_j, \hat{f}^{(-i)}(\vec{x}_j)\big) \text{ where } j \text{ ranges over rows of the } i\text{th fold.}$$

The final estimate would be the average of all results, *i.e.*:

$$\hat{\text{Err}} = \frac{1}{k} \sum_{i=1}^{k} \text{Err}_i.$$ ⌟

Here, if we let $k = n$, $\hat{\text{Err}}$ will be **unbiased**, since we consider the possibility of error for all missing rows.

However, we are adding up *a lot of data points*, and we are using most of the data each time to train. Since most training sets will be similar (except for one point), so we are beholden to peculiarities of $[D]$, *i.e.*, having high variance. When k is relatively smaller, like $k = 5$, we are cutting down on the **process of training data**. This would cause **bias in the estimator** and affect **model performance** since we have a lot less training data.

The practical guidance suggest $k = 5$ to 10 for cross validation. But the essential point is that both **model selection** and **model training** must be done in each fold, and cannot *peek* at the omitted fold.

On the other hand, we may tackle on the selection of **features**. In considering the maximizing the likelihood, these methods could be biased due to model **misspecification**, and high variance due to **too many features** being used.

Setup III.6.4. Feature Selection for Function Learning.

Imagine the true function with the form:

$$Y = \beta_0 + \sum_{i=1}^{k} \beta_i X_i + \epsilon,$$

where β_i are close to zero (or are small relative to the variance of X_i).
Such model may not do a great job in predicting Y, even if we have a linear model, it is too much **impacted**

by the noise. In particular, we may not be table to tell apart the **noise** from the **variability** of X_i and the **signal** of β_i.
Here the strategy is to select a subset of features to use in the model. ⌟

Another reason for feature selection is **model interpretability**, a regression with less features is easier to understand than a regression with a lot.

There is many ways that we can select a subset of the **features**.

(i) We can select by hand by taking the feedback from the experts (this is surprisingly often).

(ii) Alternatively, we use data for model selection.
This is a **search problem** as the number of subsets is exponential. at the same time, we need some information criteria above on the training data.

- For modern computer, it is possible to do **exhaustive search** with a moderate set of features.
- Otherwise, we must use a linear search, often with **greedy heuristics**.
  - **Forward selection**: Starts with a predictor that only uses an **intercept**, and sequentially adds more features, one by one, which is the one with the **most improvement**.
  - **Backwards selection**: Starts with a full model with all features, and sequentially removes features that **affect the criterion** the least.

From another perspective, we can consider modifying the model by trading off bias and variance.

III.7 Regularization

Regularization or **shrinkage** is imposing a **penalty** to the loss that restricts model flexibility in some way.

Definition III.7.1. Regularization Term.

For the same model, we add a penalty term $J(\vec{\beta})$ to discourage the size and number of parameters for some $\lambda \geq 0$:

$$L\big(Y, f(\vec{X}, \vec{\beta})\big) + \lambda J(\vec{\beta}).$$

⌟

The method is similar to the motivation of **Lagrange multiplier** for restricting with additional condition.

Example III.7.2. Some Regularization Loss.

Following are some popular regularized loss terms:

- **Rigid regression**: $\sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{k} \beta_j x_{ij} \right)^2 + \lambda \sum_{j=1}^{k} \beta_j^2$.
- **The lasso**: $\sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{k} \beta_j x_{ij} \right)^2 + \lambda \sum_{j-1}^{k} |\beta_j|$.

- **Bayes prior**: $\sum_{i=1}^{n}\left(y_i-\beta_0-\sum_{j=1}^{k}\beta_j x_{ij}\right)^2+\lambda\sum_{j-1}^{k}|\beta_j|^q$ for $q\geq 0$.
- **Elastic net**: $\sum_{i=1}^{n}\left(y_i-\beta_0-\sum_{j=1}^{k}\beta_j x_{ij}\right)^2+\lambda\sum_{j-1}^{k}\left[\alpha\beta_j^2+(1-\alpha)|\beta_j|\right]$. ⌟

In particular, the **ridge regression** may be solved in closed form for linear models, that is:

$$\hat{\vec{\beta}}=\left([X]^\intercal[X]+\lambda\,\mathrm{Id}\,\right)^{-1}[X]^\intercal[Y].$$

The other methods have **no closed form solutions** and thus would require methods such as **quadratic programming**.

The **regularization** of parametric regression is very popular in very high dimensional problems, in particular in the $\boldsymbol{p\gg n}$ **problems**, where the number of features greatly exceeds the number of data points. (The lasso would kill the extra parameters). In this case, the model is unlikely to work, so the models must be **restricted further** or remain **non-identified**.

Remark III.7.3. Non-identified Data for Shrinkage.

Model parameters are **non-identified** from the observed data if different parameter settings can yield the same observed data distribution.
There, no function from the observed data distribution to the parameters can exists.
Likewise, we say the model parameters as **non-identified** from the observed data if different parameter settings can yield the same observed data. ⌟

Likewise, there are various regularization methods using shrinking.

Example III.7.4. Ways of Shrinking the Coefficients.

Here, we let λ be a constant in some shrinking processes:

- **Best subset**: $\hat{\beta}_j\cdot\mathbb{1}(|\hat{\beta}_j|\geq|\hat{\beta}_{\min}|)$, where $\hat{\beta}_{\min}$ is the smallest estimated coefficient among m subsets.
- **Ridge**: $\hat{\beta}_j/(1+\lambda)$, which enforces a minimum coefficient value of 0, so this throws nothing away.
- **Lasso**: $\mathrm{sgn}(\hat{\beta}_j)\cdot(|\hat{\beta}_j|-\lambda)\cdot\mathbb{1}(|\hat{\beta}_j|\geq\lambda)$.

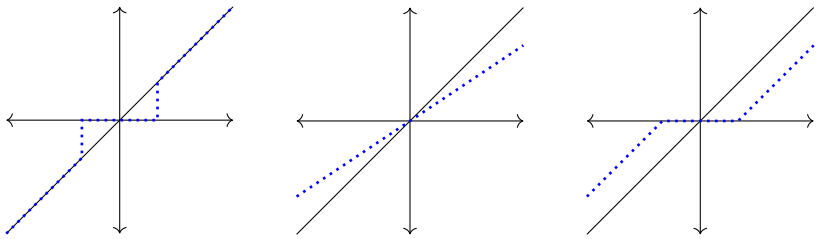


Figure III.2. Shrinking process for Best Subset (left), Ridge (middle), and Lasso (right).

⌟

When comparing the regularization methods, for coefficients β_1 and β_2, we let $\hat{\beta}$ to be the minimizer of the squared loss, so for $v > 0$, the loss is:

$$\frac{(\beta_1 - \hat{\beta_1})^2}{c_1} + \frac{(\beta_2 - \hat{\beta_2})^2}{c_2} = v,$$

which is a set of elliptical curves. The lasso constraint region uses $|\beta_1| + |\beta_2| < t$, which is a square of diamond. The ridge constraint region has $\beta_1^2 + \beta_2^2 < t^2$ which is a ball at the center.

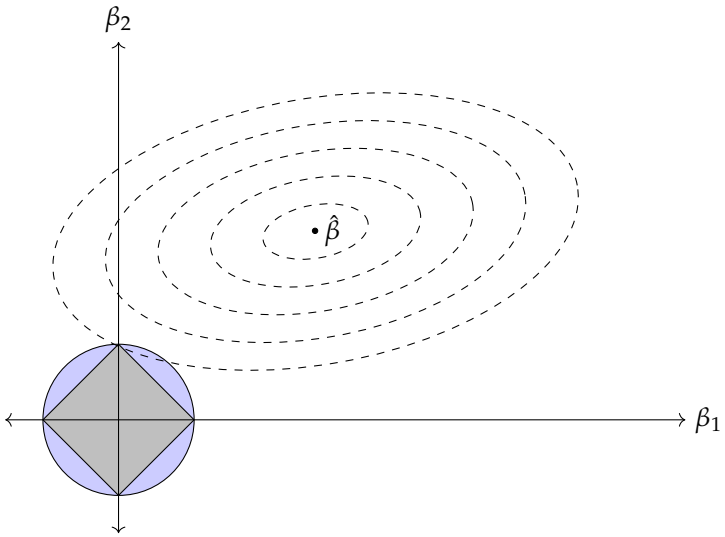


Figure III.3. The goal is to let the level curves of the loss function to intersect the boundary of the constraint region of regularization terms.

IV Classification Problem

IV.1 Decision Boundary Problem

In **regressions**, we aim to study the **loss** from the **function** $f : \vec{X} \mapsto Y$. In **classification**, the natural loss is 0-1, and the relevant **hyperplanes** partitions of the range of f into regions where inputs are mapped to specific values of Y.

Definition IV.1.1. Decision Boundaries.

The borders, typically hyperplanes of manifolds, between the regions are called **decision boundaries**. ⌟

As long as we find all boundaries, we solve the **classification problem**. The concern is to find the **optimal way** of doing so.

Setup IV.1.2. Using Decision Tree Model to Classify.

Say we have m classes $y_1, y_2, \cdots, y_m$. If we consider the **truth** as $\mathbb{P}(\vec{x}, y)$, then it may be possible for a set of features $\vec{x_j}$ to produce different outcome labels, according to the probability $\mathbb{P}(y|\vec{x_j})$. ⌟

Clearly, this case is working. However, we need to think of **optimizing** for the average and worst cases.

IV.2 Bayes Risk and Minimax Risk

First, we consider the **risk**, or **expected loss** for $\hat{f}$ at a particular input $\vec{x}$ as:

$$\mathbb{E}\left[L\big(Y,\hat{f}(\vec{x})\big)\big|\vec{x}\right].$$

Definition IV.2.1. The Bayes Risk.

For the **average performance**, we use the Bayes risk ($R_b(\hat{f})$), which averages over possible inputs, and **distributions** of inputs using a prior π:

$$R_b(\hat{f}) = \mathbb{E}_{\mathbb{P}(\vec{x})\sim\pi}\big[\mathbb{E}[L\big(Y,\hat{f}(\vec{x})\big)|\vec{x}]\big] = \mathbb{E}\left[\sum_{i=1}^{m} L\big(Y=y_i,\hat{f}(\vec{X})\big)\mathbb{P}(y_i|\vec{X})\right].$$

Here, for the 0-1 loss, it is:

$$R_b(\hat{f}) = \mathbb{E}\big[1-\mathbb{P}\big(\hat{f}(\vec{X})|\vec{X}\big)\big].$$

⌟

Definition IV.2.2. The Minimax Risk.

For the worst case performance, we use the minimax risk ($R_f(\hat{f})$). For a predictor $\hat{f}$ at $\vec{x}_j$ we have:

$$R_f(\hat{f}) = \sup_{\mathbb{P}(\vec{x})} \mathbb{E}_{\mathbb{P}(\vec{x})}\big[L\big(Y,\hat{f}(\vec{x})\big)\big|\vec{x}\big] = \inf_{\hat{f}}\sup_{\mathbb{P}(\vec{x})}\sum_{i=1}^{m} L\big(Y=y_i,\hat{f}(\vec{x})big\big)\mathbb{P}(y_i|\vec{x}).$$

Likewise, for the 0-1 loss, it is:

$$R_f(\hat{f}) = \inf_{\hat{f}}\sup_{\mathbb{P}(\vec{x})}\big(1-\mathbb{P}(\hat{f}(\vec{x})|\vec{x})\big).$$

⌟

In particular, the **minimax risk** will give the worst possible distribution over $\vec{x}$ for any m$\hat{f}$ that is picked. The job is to pick $\hat{f}$ that minimizes how badly it performs.

IV.3 Classification Models

For the sake of this part, we use the **Bayes risk**. We want a classifier that minimizes Bayes risk chooses, *i.e.*, for every $\vec{x}$, the class y_i that maximizes $\mathbb{P}(y_i|\vec{x})$. Such model is called the **optimal Bayes classifier**, denoted f^*.

Proposition IV.3.1. Optimal Bayes Classifier minimizes Bayes Risk.

Let $\tilde{f}$ be another classifier that picks a class in a different way than f^* for some $\vec{x}_i$, and other otherwise agrees with f^*, then:

$$R_b(f^*) \leq R_b(\tilde{f}).$$

⌟

Here, the above result can be verified by having the following inequalities by dividing the **Bayes risk** into

the parts in which the **classifiers disagree** and the parts in which the **classifiers agrees**:

$$\begin{aligned}
R_b(f^*) &= \mathbb{E}\big[1 - \mathbb{P}\big(f^*(\vec{x})|\vec{x}\big)\big] = \int \big(1 - \mathbb{P}(f^*(\vec{x})|\vec{x})\big)\mathbb{P}(\vec{x};\theta)\pi(\theta)d\vec{x}\theta \\
&= \int \big(1 - \mathbb{P}(f^*(\vec{x})|\vec{x})\big)\mathbb{P}(\vec{x};\theta)\pi(\theta) \cdot \big(\mathbb{1}(\vec{x} = \vec{x}_i) + \mathbb{1}(\vec{x} \neq \vec{x}_i)\big)d\vec{x}\theta \\
&= \int \big(1 - \mathbb{P}(f^*(\vec{x})|\vec{x})\big)\mathbb{P}(\vec{x};\theta)\pi(\theta) \cdot \mathbb{1}(\vec{x} = \vec{x}_i)d\vec{x}\theta + \int \big(1 - \mathbb{P}(f^*(\vec{x})|\vec{x})\big)\mathbb{P}(\vec{x};\theta)\pi(\theta) \cdot \mathbb{1}(\vec{x} \neq \vec{x}_i)d\vec{x}\theta \\
&\leq \int \big(1 - \mathbb{P}(\tilde{f}(\vec{x})|\vec{x})\big)\mathbb{P}(\vec{x};\theta)\pi(\theta) \cdot \mathbb{1}(\vec{x} = \vec{x}_i)d\vec{x}\theta + \int \big(1 - \mathbb{P}(\tilde{f}(\vec{x})|\vec{x})\big)\mathbb{P}(\vec{x};\theta)\pi(\theta) \cdot \mathbb{1}(\vec{x} \neq \vec{x}_i)d\vec{x}\theta \\
&= \mathbb{E}\big[1 - \mathbb{P}\big(\tilde{f}(\vec{x})|\vec{x}\big)\big] = R_b(\tilde{f}).
\end{aligned}$$

Here, we note that f^* and $\tilde{f}$ agree anywhere $\vec{x} \neq \vec{x}_i$, so the second terms are equal. Moreover, it is not possible to do better than f^*, for average.

Here, it seems like that we have established the **optimal solution** for classification problem, all we need to do is learn $\mathbb{P}(Y = y_i|\vec{x})$ for each i. However, we already know from regression problems to know that it is hopeless to learn $\mathbb{P}(y|\vec{x})$, or even $\mathbb{E}[y|\vec{x}]$. Hence, we need to make inductive bias assumptions to make progress.

- A **classification of a point** gives you information about **classification of points** around it.
- **Decision boundaries** have a **simple form**.
- Learn a **discriminating function** $g_i : \vec{X} \mapsto Y$ for each value y_i of Y (hopefully in **simple form**), and choose a class with highest values.
- Change the 0-1 loss into a simpler loss and minimize that instead.

From our experiences, the **simplest** and very **important** type of decision boundary is **linear**.

Remark IV.3.2. Generative versus Discriminative Models.

An important modeling choice when aiming to minimize Bayes risk is to have $\mathbb{P}(y|\vec{x})$ as a discriminative model or use $\mathbb{P}(y, \vec{x})$ as a generative model.
For **discriminative models**, the advantages are:

- It assumes less things about $\mathbb{P}(\vec{x}, y)$, but only about $\mathbb{P}(y|\vec{x})$, which is nothing with $\mathbb{P}(\vec{x})$. Therefore, it is more likely to be right.
- It is directly relevant for what we are doing (*i.e.*, **Vapnik's principle**).

For **generative models**, the advantages are:

- It has substantive assumptions, *i.e.*, assumptions that work well in practice, which is sometimes easier to phrase on $\mathbb{P}(\vec{x}, y)$.
- It does not ignore the fact about $\mathbb{P}(\vec{x}, y)$, which is not expressible when discussing $\mathbb{P}(y|\vec{x})$. Such facts should be exploited, not discarded. ⌟

Both **generative models** and **discriminative models** are effective models, and the arguing of which is better is ongoing.

IV.4 Linear Decision Boundaries

Recall that the simplest decision boundary is **linear**. Here, we have a **point** for one feature, a **line** for two features, a **plane** with three features, and a **hyperplane** for more features.

Remark IV.4.1. Not All Boundaries will Work.

Not all boundaries will work to be the boundaries.
For some hyperplanes, they might not be able to separate the data via their feature.
In fact some data are not separable in the current dimension, such as `XOR`. ⌟

Hence, our main goal is to find the linear model from learning.

Setup IV.4.2. Learning a Linear Boundary.

With m values $y_1, \cdots, y_m$ of Y, we can learn m linear regression models, namely:

$$f_i(\vec{x}) = \beta_{i0} + \sum_{k=1}^{K} \beta_{ik}\vec{x}_k,$$

with $\mathbb{1}(Y = y_i)$ as an outcome for $i = 1, \cdots, m$.
Therefore, the boundary between y_i and y_j will be whenever $\hat{f}_i(\vec{x}) = \hat{f}_j(\vec{x})$. Here, the **linear discriminant functions** lead to a **hyperplane**:

$$\{\vec{x} : (\beta_{i0} \hat{-} \beta_{j0}) + (\hat{\beta}_i - \hat{\beta}_j)^\intercal \cdot \vec{x} = 0\}.$$

This setup is **true** for any pair of classes. Moreover, any monotone transformation of a **linear discriminant function** will also lead to a **linear decision boundary**. ⌟

The **linearity** is preserved during **monotone transformations**.

Example IV.4.3. Decision Boundary for Logistic Regression.

We consider the logistic regression for two classes, namely:

$$\mathbb{P}(Y = 1|\vec{x}) = \frac{\exp(\beta_0 + \vec{\beta}^\intercal \cdot \vec{x})}{1 + \exp(\beta_0 + \vec{\beta}^\intercal \cdot \vec{x})}, \qquad \mathbb{P}(Y = 0|\vec{x}) = \frac{1}{1 + \exp(\beta_0 + \vec{\beta}^\intercal \cdot \vec{x})}.$$

Hence, this implies that:

$$\log \frac{\mathbb{P}(Y = 1|\vec{x})}{\mathbb{P}(Y = 0|\vec{x})} = \beta_0 + \vec{\beta}^\intercal \cdot \vec{x}.$$

Here, $\log\big(p/(1-p)\big)$ is a monotone function, so it yields to the following decision boundary:

$$\{\vec{x} : \beta_0 + \vec{\beta}^\intercal = 0\}.$$ ⌟

It should be noted that linear boundaries can be transformed in a **feature space**. For example, we can add square terms or interaction terms, such as $X_1 \cdot X_2$ or X_1^2 as added features.

Even if we do **transformations**, the boundary is still **linear** in the transformed feature space with respect to the parameters.

The fundamental idea is that the very complex transformation and still use linear methods.

IV.5 Multivariate Gaussian Distribution

A straightforward generalization exists for m classes $y_1, \cdots, y_m$. Here, we have: We consider the **logistic regression** for two classes, namely:

$$\mathbb{P}(Y=1|\vec{x}) = \frac{\exp(\beta_0 + \vec{\beta}^\intercal \cdot \vec{x})}{1+\exp(\beta_0 + \vec{\beta}^\intercal \cdot \vec{x})}, \qquad \mathbb{P}(Y=0|\vec{x}) = \frac{1}{1+\exp(\beta_0 + \vec{\beta}^\intercal \cdot \vec{x})}.$$

We may fit this by **Newton-Raphson**, as in the simple case where $m = 2$. We can show that the separation is linear model, still.

Remark IV.5.1. Optimal Classification and Specification.

Optimal classification requires correct specification of $\mathbb{P}(Y|\vec{X})$. Sometimes it is easier to achieve by modeling $\mathbb{P}(Y)$ and simple model of $\mathbb{P}(\vec{X}|Y)$. This yields the **Bayes rule**, is:

$$\mathbb{P}(Y|\vec{X}) = \frac{\mathbb{P}(Y)\mathbb{P}(\vec{X}|Y)}{\sum_y \mathbb{P}(y)\mathbb{P}(y)\mathbb{P}(\vec{X}|Y=y)}.$$ ⌟

Therefore, a useful method is the **multivariate Gaussian distribution**.

Definition IV.5.2. Multivariate Gaussian Distribution.

A **multivariate normal distribution** $\mathbb{P}(\vec{x})$ with k variables $X_1, \cdots, X_k$ has the form:

$$\mathbb{P}(\vec{x}) = \frac{1}{(2\pi)^{k/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(\vec{x}-\vec{\mu})^\intercal \Sigma^{-1}(\vec{x}-\vec{\mu})\right),$$

where $\vec{\mu}$ is a vector of k **means** and Σ is a k-by-k **covariance** matrix, and $|\Sigma|$ is the determinant of matrix Σ. In particular, the covariant matrix is:

$$\Sigma := \begin{pmatrix} \mathrm{Var}(X_1) & \mathrm{Cov}(X_1, X_2) & \cdots & \mathrm{Cov}(X_1, X_k) \\ \mathrm{Cov}(X_2, X_1) & \mathrm{Var}(X_2) & \cdots & \mathrm{Cov}(X_2, X_k) \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{Cov}(X_k, X_1) & \mathrm{Cov}(X_k, X_2) & \cdots & \mathrm{Var}(X_k) \end{pmatrix}.$$ ⌟

Such distribution will induce a bell shaped curve in higher dimensions.

IV.6 Linear Discriminant Analysis and Quadratic Discriminant Analysis

Setup IV.6.1. Linear Discriminant Analysis.

In **linear discriminant analysis** (LDA), it assumes that:

$$\mathbb{P}(Y|\vec{X}) = \frac{\mathbb{P}(Y)\mathbb{P}(\vec{X}|Y)}{\sum_{y_i} \mathbb{P}(y_i)\mathbb{P}(\vec{X}|Y=y_i)}$$
$$\mathbb{P}(\vec{X}|y_i) = \frac{1}{(2\pi)^{k/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(\vec{x}-\vec{\mu_i})^\intercal \Sigma^{-1}(\vec{x}-\vec{\mu_i})\right).$$

In other words, $\vec{\mu}_i$ are class specific, but the covariant matrix Σ is the same shared across all classes. ⌟

Here, we note that this is a linear boundary because:

$$\begin{aligned}\log \frac{\mathbb{P}(Y=y=y_i|\vec{x})}{\mathbb{P}(Y=y_m|\vec{x})} &= \log \frac{\mathbb{P}(y_i)}{\mathbb{P}(y_m)} + \log \frac{\mathbb{P}(\vec{x}|y_i)}{\mathbb{P}(\vec{x}|y_m)} \\ &= \underbrace{\log \frac{\mathbb{P}(y_i)}{\mathbb{P}(y_m)} - \frac{1}{2}(\vec{\mu}_i + \vec{\mu_m})^\intercal \Sigma^{-1}(\vec{\mu}_i - \vec{\mu_m})}_{\text{constant}} + \underbrace{\vec{x}^\intercal \Sigma^{-1}(\vec{\mu}_i - \vec{\mu_m})}_{\text{linear in } \vec{x}}.\end{aligned}$$

Implementation-wise, we need to learn $\mathbb{P}(Y)$ and $\mathbb{P}(\vec{X}|Y=y_i)$ for every y_i.

Remark IV.6.2. MLE for LDA Setup.

Since $\mathbb{P}(Y)$ is just a **discrete distribution**, its MLE is $\#^i/n$, whichi s the number of times a class ovvurs over the number of rows.
For $\mathbb{P}(\vec{X}|y_i)$, it is class-specific, and we use the **multivariate Gaussian**, so it can be specified in closed form:

$$\begin{aligned}\hat{\vec{\mu}}_i &= \left(n_i^{-1}\sum_{j=1}^{n_i} x_{1j} \quad n_i^{-1}\sum_{j=1}^{n_i} x_{2j} \quad \cdots \quad n_i^{-1}\sum_{j=1}^{n_i} x_{kj}\right)^\intercal, \\ \hat{\Sigma} &= n^{-1}\sum_{i=1}^{m}\sum_{j=1}^{n_i}(\vec{x}_j - \vec{\mu}_i)(\vec{x}_j - \vec{\mu}_i)^\intercal/(n-m).\end{aligned}$$

where n_i is a y_i-specified subset of $[D]$, and j ranges over rows of that subset. ⌟

Note that the **LDA setup** gives the same decision boundary compared to **logistic regression**. However, the LDA is a generative model, meaning it has assumptions on $\mathbb{P}(Y, \vec{X})$, whereas logistic regression is discriminative, since it has assumptions on $\mathbb{P}(Y|\vec{X})$, where $\mathbb{P}(\vec{X})$ is left unrestricted.

If the **assumptions on joint** holds, then LDA high higher efficiency (or reduced variance), but if the assumption does not hold, it will gain more bias.

An **alternative representation** of the decision boundary in LDA is the class-specific discriminant function, in which we evaluate this **function per class** and classify based on which function gives the **largest output**.

Setup IV.6.3. Quadratic Discriminant Analysis.

If we do not wish to assume Σ constant across classes, but each class y_i has its own Σ_i, we end up with a **quadratic discriminant function** instead:

$$\frac{1}{2}\log|\Sigma_i| - \frac{1}{2}(\vec{x} - \vec{\mu}_i)^\intercal \Sigma^{-1}(\vec{x} - \vec{\mu}_i) + \log \mathbb{P}(Y = y_i).$$

⌟

When having LDA with **quadratic features**, we will see the non-linear boundaries, it is similar to using QDA. Some differences are on the inductive biases of selecting the boundary for some points.

When we generalize the linear classifiers, we get the Naive Bayes, as there are many ways to impose on

smoothness (LDA uses the **multivariate Gaussian**).

Setup IV.6.4. Naive Bayes.

A popular approach to achieve conditional independence is by **naive Bayes**, which assumes that:

$$X_i \amalg X_1, \cdots, \widehat{X_i}, \cdots, X_k | Y \text{ for all } X_i.$$

I.e., conditional on the outcome of all features are independent, which simplifies to:

$$\mathbb{P}(\vec{X}|y_i) = \prod_{i=1}^{k} \mathbb{P}(X_i|y_i),$$

thus giving that:

$$\mathbb{P}(y_i|\vec{X}) = \frac{\mathbb{P}(y_i)\prod_{i=1}^{k}\mathbb{P}(X_i|y_i)\}}{\sum_{y_i}\mathbb{P}(y_i)\prod_{i=1}^{k}\mathbb{P}(X_i|y_i)}.$$ ⌟

Note that the assumption in which **all features are independent** is almost certainly no true in practice. However, the setup allows many advantages:

- It is easy to incorporate **discrete and real-valued features** as well as **missing values**.
- It can fit model pieces $\mathbb{P}(y_i)$ and $\mathbb{P}(x_j|y_i)$ separately, and since the models are **univariate**, it is more efficient.
- The performance is quite good in practice.

Again, as we consider the decision boundary, we have:

$$\log\frac{\mathbb{P}(y_i|\vec{x})}{\mathbb{P}(y_m|\vec{x})} = \underbrace{\log\frac{\mathbb{P}(y_i)}{\mathbb{P}(y_m)}}_{\text{constant}} + \sum_{i=1}^{k} \underbrace{\log\frac{\mathbb{P}(X_i|y_i)}{\mathbb{P}(X_i|y_m)}}_{\text{single feature transformations } g_{im}}.$$

Again, since $\mathbb{P}(Y)$ is discrete, we have the MLE being $\#^i/n$, and for $\mathbb{P}(\vec{X}|y_i)$, we learn parameter of each pieces altogether.
If X_i is discrete, the MLE is by counting values. If X_i is real valued, we postulate a univariate parametric model and find its MLE.

IV.7 Perceptrons

In general, we can view the linear decision boundaries are a hyperplane of the form:

$$\{x : \beta_0 + \sum_{i=1}^{k} \beta_i \cdot x_i = 0\},$$

where the decision is derived based on the sign of the outcome $\text{sgn}(\beta_0 + \sum_{i=1}^{k}\beta_i \cdot x_i)$.

Remark IV.7.1. MLE Solution is Not Necessarily Perfect.

The MLE solution is not necessarily **perfect separation**, this is because that MLE does not account for 0-1 loss, hence it would be inclined to also minimize the **furtherest squared distance**. ⌟

There are **infinitely many choices** for a decision boundary, the problem is on how to pick the best one.

Remark IV.7.2. Criterion of Good Decision Boundary.

We want to consider about the distance of points to classify the decision boundary.

- If there are **misclassified points** that are far away from boundary, we want it to be closer.
- If there are **correctly classified points** that is close to the boundary, we want to push the points away from the boundary.

In either case, we want the distance to be appropriate for the **classified** points, correct or not. ⌟

One such algorithm developed to find the **optimal** decision boundary is by **Rosenblatt's Perceptron Algorithm**.

Algorithm IV.7.3. Rosenblatt's Perceptron Algorithm.

Suppose we have a set of k features $\vec{X}$ and a binary outcome Y (with values 1 or -1). Suppose we have a linear boundary that classifies all points perfectly, we use the **decision rule** that $\text{sgn}(\beta_0 + \vec{x_i}^\intercal \cdot \vec{\beta})$.
Here, we define a **generalized 0-1 loss** to account for the concept of **distance** for the points:

$$L(y_i, \hat{f}(\vec{x_i})) = \begin{cases} -y_i \cdot \hat{f}(\vec{x_i}), & \text{if } y_i \text{ is misclassified,} \\ 0, & \text{if } y_i \text{ is correct,} \end{cases}$$

where $\hat{f}(\vec{x_i}) = \beta_0 + \vec{x_i}^\intercal \cdot \vec{\beta}$. ⌟

The **penalization** is proportional to how far off the classifications are, so it will manage to optimize the boundary when there are points that are misclassified.

Moreover, since it is a **linear function**, it is **differentiable**. Thus, we can minimize $R_{[D]}(\hat{f}) := \sum_{i=1}^{n} L(y_i, \vec{x_i})$, and by taking the derivative, we have:

$$\frac{\partial R_{[D]}(\hat{f})}{\partial \vec{\beta}} = -\sum_{y_i \neq \hat{f}(\vec{x_i})} y_i \vec{x_i}, \text{ and } \frac{\partial R_{[D]}(\hat{f})}{\partial \beta_0} = -\sum_{y_i \neq \hat{f}(\vec{x_i})} y_i.$$

For the Perceptron, it is possible to do **online learning**, which is to accommodate the data one by one.

Algorithm IV.7.4. Newton-Raphson Algorithm in Perceptron / Gradient Descent.

We consider each step updates the parameter, using some step size ρ, we have the update rule as:

$$\begin{pmatrix} \vec{\beta} \\ \beta_0 \end{pmatrix} \leftarrow \begin{pmatrix} \vec{\beta} \\ \beta_0 \end{pmatrix} + \rho \begin{pmatrix} \sum_{y_i \neq \hat{f}(\vec{x_i})} y_i \vec{x_i} \\ \sum_{y_i \neq \hat{f}(\vec{x_i})} y_i \end{pmatrix}.$$

Here, the ρ is a **step size** or the **learning rate**. ⌟

Such algorithms can be better fitted to the **Perceptron online learning**, namely by having the **stochastic**

gradient descent that updates only using the derivative at the current point we see, so it becomes:

$$\begin{pmatrix}\vec{\beta}\\ \beta_0\end{pmatrix} \leftarrow \begin{pmatrix}\vec{\beta}\\ \beta_0\end{pmatrix} + \mathbb{1}(y_i \neq \hat{f}(\vec{x}_i))\rho \begin{pmatrix}y_i\vec{x}_i\\ y_i\end{pmatrix}.$$

The **perceptron algorithm** is very simple and efficient.

Proposition IV.7.5. Existence of Solution $\Longrightarrow$ Finite Steps in Perceptron.

Suppose that we have a set of samples from $\mathbb{P}(\vec{x}, y)$ which gives either all at one, or one at a time, such that there exists a linear decision boundary separating classes $y_i = 1$ and $y_i = -1$, the the **perceptron algorithm** can find such boundary in finite number of steps. ⌟

At the same time, such algorithm is having some problems with its correspondence.

- Finite time is **finite**, but it could still be long. In particular, if the true gap between two points is very small, then it could take **very long** to separate them.
- If there are linear boundaries, there are generally **lots of them**, and the algorithm can find different ones by the **order** that the data were fed in.
- If there are no linear boundary, the algorithm will **never terminate** and go around, and this is **hard to determine** in advance.

One way to address for this is **maximizing the margin**. The key idea is that:

> If a point is classified as **one type**, perhaps the nearby points are also classified as the **same type**.

Therefore, we want to keep points as far away as possible, which leads to keeping the points as far away from the boundary as possible.

Setup IV.7.6. Modified Rosenblatt's Loss.

We are having M as the margin, and:

$$\begin{aligned}&\max_{\vec{\beta},\beta_9,\|\vec{\beta}\|=1} M,\\ &\text{subject to } y_i(\vec{x}_i^{\intercal}\vec{\beta} + \beta_0) \geq M \text{ for all } i = 1, \cdots, n.\end{aligned}$$

Meanwhile, we can **normalize** $\vec{\beta}$, and since the magnitude does not matter, we can let $\|\vec{\beta}\| = 1/M$, which leads to:

$$\begin{aligned}&\min_{\vec{\beta},\beta_0} \frac{1}{2}\|\vec{\beta}\|^2\\ &\text{subject to } y_i(\vec{x}_i^{\intercal}\vec{\beta} + \beta_0) \geq 1 \text{ for all } i = 1, \cdots, n.\end{aligned}$$ ⌟

IV.8 Constrained Optimization

Here, we are free to use the **Lagrange method** to solve for the constraint maximization or minimization, that is:

$$\Lambda(\vec{\beta}, \lambda) := f(\vec{\beta}) - \lambda g(\vec{\beta}).$$

Checking on the **stationary points** correspond to the **optima** of the original problem with the Hessian:

$$H(\Lambda) = \begin{pmatrix} \frac{\partial^2 \Lambda}{\partial \lambda^2} & \frac{\partial^2 \Lambda}{\partial \lambda \partial \vec{\beta}} \\ \left(\frac{\partial^2 \Lambda}{\partial \lambda \partial \vec{\beta}}\right)^\intercal & \frac{\partial^2 \Lambda}{\partial \vec{\beta}^2} \end{pmatrix}.$$

When there are more constrains, we can add more terms $g_i(\vec{\beta})$ with more λ_i parameters.

Theorem IV.8.1. The Karush-Kuhn-Tucker Theorem.

Say we want to maximize $f(\vec{\beta})$ subject to $g(\vec{\beta}) = 0$ and $h(\vec{\beta}) \leq 0$, we can rephrase the problem as finding stationary points of the Lagrange function:

$$\Lambda(\vec{\beta}, \lambda, \mu) := f(\vec{\beta}) - \lambda g(\vec{\beta}) - \mu h(\vec{\beta}).$$

The following conditions should hold for such a point to solve the original problem:

- **Stationarity**: Derivatives of Λ with respect to $\vec{\beta}$ are 0.
- **Primal feasibility**: $g(\vec{\beta}) = 0$ and $h(\vec{\beta}) \leq 0$.
- **Dual feasibility**: $\mu \geq 0$, and
- **Complementary slackness**: $\mu h(\vec{\beta}) = 0$.

⌟

Here, recall the optimization criterion as:

$$\min_{\vec{\beta}, \beta_0} \frac{1}{2} \|\vec{\beta}\|^2$$
$$\text{subject to } y_i(\vec{x_i}^\intercal \vec{\beta} + \beta_0) \geq 1 \text{ for all } i = 1, \cdots, n.$$

We may use **Lagrange multiplier** using the **Karush-Kuhn-Tucker** theirem as a primal minimization criterion, that is:

$$L_P := \frac{1}{2} \|\vec{\beta}\|^2 - \sum_{i=1}^{n} \alpha_i \big(y_i(\vec{x_i}^\intercal \vec{\beta} + \beta_0) - 1\big).$$

By setting the partial derivatives of L_P as zero, we have:

$$\vec{\beta} = \sum_{i=1}^{n} \alpha_i y_i \vec{x_i} \text{ at } \frac{\partial L_P}{\partial \vec{\beta}} = 0,$$
$$\sum_{i=1}^{n} \alpha_i y_i = 0 \text{ at } \frac{\partial L_P}{\partial \beta_0} = 0.$$

As we substitute in the zero conditions, we have:

$$L_P := \frac{1}{2} \|\vec{\beta}\|^2 - \sum_{i=1}^{n} \alpha_i \big(y_i(\vec{x_i}^\intercal \vec{\beta} + \beta_0) - 1\big),$$

and by adding the **Karush-Kuhn-Tucker** conditions, it leads to:

$$\max L_D = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \vec{x_i}^\intercal \vec{x}_j,$$

subject to

$$\text{Dual feasibility: } \alpha_i \geq 0,$$
$$\text{Partial derivative condition: } \sum_{i=1}^{n} \alpha_i y_i = 0, \text{ and}$$
$$\text{Complementary slackness: } \alpha_i\big(y_i(\vec{x}_i^{\,\intercal}\vec{\beta} + \beta_0) - 1\big) = 0 \text{ for all } i.$$

Maximizing L_D is the **dual equivalent** to minimizing L_P, but is an easier convex optimization problem using **quadratic programming**.

Remark IV.8.2. Notes on Dual Equivalence.

- Observed realizations $\vec{x}$ only enter into the problem as a k-by-k matrix $\vec{x}^{\intercal}\vec{x}$, which allows us to easily **insert** feature transformations.
- Primal parameters $\vec{\beta}$ match the number of **features**, but dual parameters $\vec{\alpha}$ match the number of **data points**. ⌟

With such considerations, we have address the issues with the issue that there could be more than 1 boundary, by finding the maximum margin separating the hyperplane.

IV.9 Support Vector Machine

Note that to find the margin, we are looking for the vectors that satisfies:

$$\alpha_i\big(y_i(\vec{x}_i^{\,\intercal} + \beta_0) - 1\big) = 0,$$

and this is possible in only two ways: $\alpha_i = 0$, or $\big(y_i(\vec{x}_i^{\,\intercal} + \beta_0) - 1\big) = 0$.

We shall want to focus on the second case, so the point $\vec{x}_i$ is exactly 1 away from the boundary, and so is **precisely on the margin**. *In the first case, it is further away*.

Remark IV.9.1. Support Vectors.

Recall that $\vec{\beta} = \sum_{i=1}^{n} \alpha_i y_i \vec{x}_i$, we see that only points $\vec{x}_i$ **precisely on the margin** have $\alpha_i > 0$, and are the only ones that matter for finding $\vec{\beta}$. These vectors are called **support vectors**. ⌟

Now, our strategy is to accommodate for the **slack variables**.

If there are no hyperplane that can separate classes, we can assume that there are only a few **outliers** that will lie on the wrong side of the boundary.

Setup IV.9.2. Slack Variables.

Here, we can define **slack variables** ξ_i for $i = 1, \cdots, n$. Here, we allow some **slack** in having points lie beyond the margin:

$$y_i(\vec{x}_i^{\,\intercal}\vec{\beta} + \beta_0) \geq M(1 - \xi_i),$$

with $\xi_i \geq 0$ and $\sum_{i=1}^n \xi_i$ bounded by a constant C, which leads to the following **minimization problem**:

$$L_P := \frac{1}{2}\|\vec{\beta}\|^2 + C\sum_{i=1}^n \xi_i - \sum_{i=1}^n \alpha_i\big(y_i(\vec{x_i}^\intercal\vec{\beta}+\beta_0) - (1-\xi_i)\big) - \sum_{i=1}^n \mu_i\xi_i,$$

and we minimize this with respect to $\vec{\beta}$, β_0, and ξ_i. ⌟

Similarly, we obtain the stationary point conditions, that is:

$$\vec{\beta} = \sum_{i=1}^n \alpha_i y_i \vec{x}_i, \qquad 0 = \sum_{i=1}^n \alpha_i y_i, \qquad \alpha_i = C - \mu_i.$$

Thus, we can obtain the **dual** by substituting into the primal and adding the **Karush-Kuhn-Tucker** conditions.

Algorithm IV.9.3. Dual Formulation with Slack Variable.

$$\begin{aligned}
&\max_{\alpha} \sum_{i=1}^n \alpha_i - \frac{1}{2}\sum_{i=1}^n\sum_{j=1}^n \alpha_i\alpha_j y_i y_j \vec{x}_i^T\vec{x}_j \\
&\text{subject to } 0 \leq \alpha_i \leq C, \quad i = 1,\dots,n \\
&\qquad \sum_{i=1}^n \alpha_i y_i = 0, \text{ and} \\
&\qquad \alpha_i\big(y_i(\vec{x_i}^\intercal\vec{\beta}+\beta_0) - (1-\xi_i)\big) = 0 \text{ and } \mu_i\xi_i = 0 \text{ for all } i.
\end{aligned}$$ ⌟

As before, the solution has the form:

$$\beta = \sum_{i=1}^n \alpha_i y_i \vec{x}_i, \text{ where } \alpha_i \geq 0.$$

Unlike before, some points will lie exactly on the margin ($\xi_i = 0$), and others will be misclassified or lie within the margin ($\xi_i > 0$ and $\alpha_i = C$), the decision rule is still $\operatorname{sgn}(\vec{x}^\intercal\vec{\beta} + \beta_0)$.

IV.10 Kernel Tricks

Up to now, we are assuming that the classification problem is **almost linear**. However, if the problem is highly non-linear, the **slack variable** approach breaks down and leads to poor classification.

In addressing this, the two most natural ways are:

- We may expand the feature space, so the non-linear problem can become linear.
  The explicit example of this is having **polynomial terms** and **interactions** in parametric regression models.
- Otherwise, we consider that what enters the SVM clarifier is the **inner product** $\vec{x}^\intercal\vec{x}$, which can be replaced.
  Here, the inner product is just:
  $$K(\vec{x_1},\vec{x_2}) = \langle\vec{x_1},\vec{x_2}\rangle,$$
  but we just need the output, so we can use another **kernel** as $K(\vec{x_1},\vec{x_2})$.

Example IV.10.1. Typical Kernels In Use.

There are a few **kernels** that are typically used.

- **Linear Kernel** (Default): $K(\vec{x_1}, \vec{x_2}) = \langle \vec{x_1}, \vec{x_2} \rangle$,
- **Degree d Polynomial**: $K(\vec{x_1}, \vec{x_2}) = (1 + \langle \vec{x_1}, \vec{x_2} \rangle)^d$,
- **Radial Basis**: $K(\vec{x_1}, \vec{x_2}) = \exp(-\gamma \|\vec{x_1} - \vec{x_2}\|^2)$,
- **Neural Network Activation**: $K(\vec{x_1}, \vec{x_2}) = \tanh(\kappa_1 \langle \vec{x_1}, \vec{x_2} \rangle + \kappa_2)$.

⌟

In fact, with a proper inner product, the number of m does not need to be finite, which generalizes to a **infinite dimension Hilbert space** ($\mathcal{H}$).

Here, we note that the SVM is minimizing the **hinge loss**.

Proposition IV.10.2. SVM Minimizes the Hinge Loss.

The SVM algorithm is equivalent to:

$$\min_{\vec{\beta}, \beta_0} \sum_{i=1}^{n} \max\left((1 - y_i (h(\vec{x})^\intercal \vec{\beta} + \beta_0)), 0 \right) + \frac{\lambda}{2} \|\vec{\beta}\|^2,$$

which is known as the **hinge loss**. ⌟

The other classifications uses different types of losses:

Classifier	Loss	Loss Expression	Minimizing Function of $\mathbb{P}(\vec{x}, y)$
optimal Bayes	0-1	$\mathbb{1}\big(y = f(\vec{x})\big)$	$f(x) = \mathbb{P}(y \vert \vec{x})$
logistic regression	deviance	$\log(1 + e^{-yf(\vec{x})})$	$f(x) = \log \frac{\mathbb{P}(y=+1 \vert \vec{x})}{\mathbb{P}(y=-1 \vert \vec{x})}$
SVM	hinge	$\max(1 - yf(\vec{x}), 0)$	$f(x) = \operatorname{sgn}\left(\mathbb{P}(y = 1 \vert \vec{x}) - \frac{1}{2}\right)$
LDA	squared	$\big(y - f(\vec{x})\big)^2 = \big(1 - f(\vec{x})\big)^2$	$f(x) = 2\mathbb{P}(y = 1 \vert \vec{x}) - 1$
AdaBoost	exponential	$\exp\big(-yf(\vec{x})\big)$	

The graphs of the loss can be shown as follows.

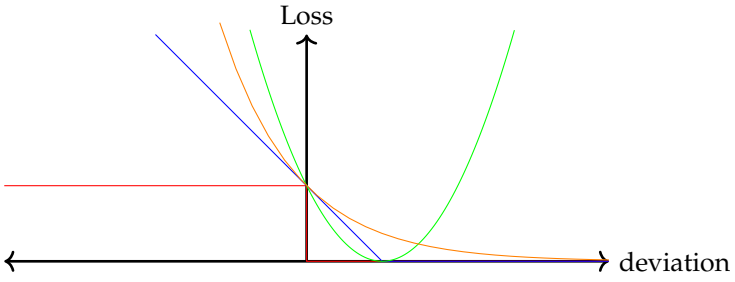


Figure IV.1. Loss functions for 0-1(red), hinge(blue), squared(green), and exponential(orange). Deviance similar to exponential.

V Ensemble Methods

Ensemble methods aim to combine **multiple predictors** to achieve **better performance** than each **individual predictor** could on its own.

V.1 Bootstrap and Bagging

Bootstrap aggregating improves the predictions in high variance, low bias data set. It is a general method for assessing reliability of a procedure that uses data.

Setup V.1.1. Bootstrap Assessment.

Let $\mathbb{P}(\vec{x}, y)$ be an observed data distribution, and $[D]$ be a data set of independent samples from this distribution. We want to assess any function $g : [D] \mapsto \hat{\theta}$.
Ideally, we want to give g lots of datasets drawn from $\mathbb{P}(\vec{x}, y)$. However, we do not have access to $\mathbb{P}(\vec{x}, y)$.
⌟

To account for the loss of full distribution, we assign the data that do not appear with measure 0, hence, we can develop a **pick without replacements**.

Algorithm V.1.2. Bootstrap Approach.

(i) Sample with m datasets $[D]_1, \cdots, [D]_m$ of the same size as $[D]$ by sampling rows of $[D]$ **with replacement**.

(ii) Then calculate $\hat{\theta_1} = g([D]_1), \cdots, \hat{\theta_m} = g([D]_m)$.

(iii) Use the above step to calculate anything involving the distribution of $g([D])$ for $[D] \sim \mathbb{P}(\vec{x}, y)$. ⌟

Although we do not have access to $\mathbb{P}(\vec{x}, y)$, we can view the **original data set** $[D]$ as an approximation of $\mathbb{P}(\vec{x}, y)$.
Hence, instead of sampling from $\mathbb{P}(\vec{x}, y)$, we sample from $[D]$, and this is often used to calculate **confidence intervals**, as being used to assess uncertainty in parameter estimation due to sampling variability in $[D]$.

Moreover, **bootstrap** is an alternative to cross-validation to estimate **generalization error**.

Example V.1.3. Bootstrap Aggregating with Regression.

We consider the regression problem first. Instead of using predictor $\hat{f}$ learned by a function g that uses $[D]$, we use:

$$\hat{f}_{\text{bag}}(\vec{x}) := \frac{1}{m} \sum_{i=1}^{m} \hat{f}_i(\vec{x}),$$

where we use $[D]_i$, the i-th bootstrap dataset to train $\hat{f}_i := g([D]_i)$. ⌟

With the above example, we take the **majority vote** for **classification**. Here, we want to analyze if **bagging** helps.

Remark V.1.4. Notes on Classification and Regression of Bagging.

In terms of **classification**, we note that:

- Instead of sampling $[D]_1, \cdots, [D]_m$ from $[D]$, we sampled from $\mathbb{P}(\vec{x}, y)$ directly.
- If all $[D]_i$ are independent, so are the learners $\hat{f}_i = g([D]_i)$.
- If each learner has an error probability of $0.5 - \epsilon$, then the majority vote have error probability $\to 0$ as $m \to \infty$, which is known as **the wisdom of the crowds** effect.
- The key is that the learners **have to be independent**. If we sample $[D]_i$ from $[D]$m they are not independent.

In terms of **regression**, we note that:

- Suppose we are doing a regression with a squared loss $\mathbb{E}\big[Y - \big(\hat{f}_{[D]}(\vec{x})\big)^2\big]$. Consider an idealized bagging regression model that averages over all datasets $[D]$ drawn from $\mathbb{P}(\vec{x}, y)$:
$$f_{\text{bag}}(\vec{x}) := \mathbb{E}_{[D]\sim\mathbb{P}(\vec{x},y)}\big[\hat{f}_{[D]}(\vec{x})\big].$$
- Then, we have the expectation of squared loss such that:
$$\begin{aligned}
&\mathbb{E}\big[\big(Y - \hat{f}_{[D]}(\vec{x})\big)^2\big] \\
&= \mathbb{E}\big[\big(Y - f_{\text{bag}}(\vec{x})\big)^2\big] + \underbrace{2\mathbb{E}\big[\big(Y - f_{\text{bag}}(\vec{x})\big)\big(f_{\text{bag}}(\vec{x}) - \hat{f}_{[D]}(\vec{x})\big)\big]}_{0} + \mathbb{E}\big[\big(\hat{f}_{[D]}(\vec{x}) - f_{\text{bag}}(\vec{x})\big)^2\big] \\
&\geq \mathbb{E}\big[\big(Y - f_{\text{bag}}(\vec{x})\big)^2\big].
\end{aligned}$$
Therefore, the aggregation process does not increase the mean square error, and would often decrease it by reducing the variance.

Although these arguments do not necessarily carry over to sampling from $[D]$, the bagging has found to help in $[D]$ with **low bias** but **high variance**. ⌟

V.2 Super Learner

Super Learner uses **cross-validation** to create as combined learner that does as well as the best learner in an ensemble.

The basic idea of **super learner** is the weighted combination of **learners** that are trained differently. The process lies as follows.

Algorithm V.2.1. Super Learner Training.

Our problem is to have a **regression model**, minimizing $\mathbb{E}\big[L\big(Y, \hat{f}(\vec{X})\big)\big]$ by choosing parameters of $\hat{f}(\vec{x})$ using $[D]$ of size N.
Additionally, we assume that we have a library of functions $f_1, \cdots, f_M$ of possible predictors, and we want to combine them in a useful way, which is by **cross-validation**.

(i) Fit each $\hat{f}_{1,[D]}, \cdots, \hat{f}_{M,[D]}$ using all of $[D]$.

(ii) Split $[D]$ into training $[T]_k$ and validation $[V]_k$ for $k = 1, \cdots, K$ pieces using ***K*-fold cross validation**.

(iii) For each $1, \cdots, K$ fold, fit each $f_1, \cdots, f_M$ using $[T]_k$, and save the predictions on $[V]_k$.

(iv) Create a $N \times M$ matrix of **validation predictions**, where each $\hat{f_{km}}$ is the mth learner trained using $[T]_k$. We use $\hat{f_{km}}$ to predict $[V]_k$.

(v) Consider a weighted average of all predictors:

$$f^*_{k,\hat{\alpha}}(\vec{x}) := \sum_{m=1}^{M} \alpha_m \hat{f_{km}}(\vec{x})$$

for every **validation region** $[V]_k$ in $[D]$, where $\sum_{m=1}^{M} \alpha_m = 1$.

(vi) Then, we choose $\hat{\vec{\alpha}}$ to minimize $\sum_{i=1}^{N} \left(y_i - f^*_{\vec{\alpha}}(\vec{x}_i)\right)^2$.

With all of above, we can create a final predictor:

$$f_{\text{sl}}(\vec{x}) = \sum_{m=1}^{M} \hat{\alpha_m} \hat{f_m, [D]}(\vec{x}). \qquad \lrcorner$$

The **super learner** algorithm is quite involved, and it is **computationally challenging**.

Proposition V.2.2. Super Learner as Best Possibly Weight Combination.

The **super learner** does as well as the best possibly weighted combination in $\mathcal{L}$ as $N \to \infty$.
I.e., if $\mathcal{L}$ contains the true $\mathbb{E}[Y|\vec{X}]$, then **super learner** will do as well as if we used that. $\lrcorner$

Super learner will add the technological distinctiveness of predictor to its own.

Remark V.2.3. Practical Problem with Super Learner.

However, the super learner has a few practical problems:

- Every predictor in $\mathcal{L}$ must run on $[D]$ and $[T]$, and the different implementations have different requirements in practice.
- It might not be able to have **good properties** at the available sample size.
- The weighted predictor might not run on $[D]$ in practice.
  For example, a weighted combination of parametric linear models may lead to a non-invertible design matrix $[X]^\intercal[X]$.

$\lrcorner$

V.3 Random Forests

Random Forest combines small decision trees into a high performance learner. It leads to a **random forest** family of models.

Algorithm V.3.1. Random Forest Algorithm.

For $b = 1, \cdots, B$, we do:

(i) Draw a **bootstrap dataset** $[D]_b$ by sampling rows with replacement from $[D]$.

(ii) Learn a tree T_b using $[D]_b$ as follows:

- Pick a subset of variables $\vec{S} \subset v$ (which are **columns** in $[D]_b$),
- Pick the best split variable in M$\vec{S}$ (such as via IDS heuristic),
- Split the tree, create two split datasets, and recurse until at a **stopping criterion**.

(iii) For **regressions**, we have output:

$$\hat{f_{\text{rf}}} = \frac{1}{B} \sum_{b=1}^{B} T_b(\vec{x}).$$

For **classification** model, we have output as majority vote among $T_b(\vec{x})$. ⌟

If the subsets $\vec{S}$ are correlated, variance reduction by averaging over T_b will be limited by the extent of correlation. Hence, we may pick variables up randomly at step (ii).

There area various other types of **random forests** based on different principles.

Example V.3.2. Bayesian Auto-Regressive Trees.

The **Bayesian auto-regression trees** (BART) use Gibbs sampling to generate posterior samples (sample 1 tree out of 200 given the others are fixed). Prior favors small trees. ⌟

There is no clear idea why random forests work so well. Roughly, trees are a **high variance estimator**, and bagging is a **variance reduction method** in some cases, hence they fir well.

V.4 Boosting and AdaBoost

Boosting combines *weak learners* into a strong learner.

Remark V.4.1. Weak Learners should be Better than Guessing.

Boosting improves **weak classifiers** (probability of correct classification is $0.5 + \epsilon$) to yield a **strong classifier**. It works by recursively challenging with the part of the data where the classifier constructed so far is not doing well. ⌟

The idea of **boosting** is from the proposition about the learners.

Proposition V.4.2. Existence of Learners.

Given a random example from an unknown, arbitrary $\mathbb{P}(\vec{x}, y)$:

(i) There exists a **strong** probably approximately correct (PAC) learner means for any distribution, with high probability, given a polynomial sized data set and polynomial times, we can create a classifier with arbitrarily small generalization error.

(ii) There exists a **weak** probably approximately correct (PAC) leaner means the same condition above, but classifiers only does slightly better than random guessing. ⌟

It is clear that the (ii) is easier to approach than (i), but we want to achieve (i).

The existence of a boosting algorithms that shows (ii) implies (i) was by Schapire in 1989, and the first practical and effective algorithm was **AdaBoost** by Freund and Shapire in 1995.

Algorithm V.4.3. AdaBoost.

For $t = 1, \cdots, T$, we do:

(i) Construct a distribution $\mathbb{P}_{t,[D]}$ from $[D]$.

(ii) Find a classifier $f_t : \vec{x} \mapsto y \in \{-1, 1\}$ with small error ϵ_t on instances from $\mathbb{P}_{t,[D]}$.

Eventually, we construct the **final classifier** f_{final} from f_t (where $t = 1, \cdots, T$). ⌟

When constructing $\mathbb{P}_{t,[D]}$, we have:

- All distributions will be **re-weighted versions** of the empirical distribution given by $[D]$.
- For the ith row, we have $\mathbb{P}_{1,[D]}(\vec{x}_i, y_i) = 1/n$.
- For the ith row, we update the distribution as:
$$\mathbb{P}_{t+1,[D]} = \frac{\mathbb{P}_{t,[D]}(\vec{x}_i, y_i)}{Z_t} \cdot \exp\left(-\alpha_t y_i f_t(\vec{x}_i)\right),$$
which is add **more weights** to the incorrect prediction and **normalize** all.
- In particular, we want Z_t to be the normalizing constant. As long as the algorithm is running:
$$\alpha_t = \frac{1}{2} \log\left(\frac{1-\epsilon_t}{\epsilon_t}\right) > 0.$$
When it is exactly 0, the update stops, and when it is less than 0, AdaBoost behaves incorrectly.

In particular, the final classifier is:

$$f_{\text{final}}(\vec{x}) = \operatorname{sgn}\left(\sum_{t=1}^{T} \alpha_t f_t(\vec{x})\right).$$

Remark V.4.4. Training Error for AdaBoost.

Let $\epsilon_t = 0.5 - \gamma_t$, we can show that:

$$\sum_{i=1}^{n} L\big(y_i, f_{\text{final}}(\vec{x}_i)\big) \leq \prod_{t=1}^{T} 2\sqrt{\epsilon_t(1-\epsilon_t)} = \prod_{t=1}^{T} \sqrt{1-4\gamma_t^2} \leq \exp\left(-2\sum_{t=1}^{T} \gamma_t^2\right).$$

Hence, if all γ_t are **positive and bounded below** by a small constant γ, we have the sum of the loss bounded above by $\exp(-2\gamma^2 T)$, which gets **exponentially small**. ⌟

Typically, having the **training error** converges to zero is well but the **real test** might fails to generalize due to **overfitting**.

Remark V.4.5. AdaBoost has Less Impact from Overfitting.

For AdaBoost, it minimizes the exponential loss, *i.e.*:

$$L\big(y, f(\vec{x})\big) = \exp\big(-yf(\vec{x})\big).$$

The minimizer of this loss is:

$$f^*(\vec{x}) = \frac{1}{2}\log\frac{\mathbb{P}(y=1|\vec{x})}{\mathbb{P}(y=-1)|\vec{x}} \text{ or } \mathbb{P}(y=1|\vec{x}) = \frac{1}{1+\exp\left(-2f^*(\vec{x})\right)}.$$

For **exponential loss** in particular, the model still penalizes after being classified to the correct side. The model need to push the boundary **far away from the boundary** in order to have a smaller loss, hence it enforces the model to improve even if we have correct classification, but to get **away from all points**. ⌟

VI Neural Network Models

In previous discussions, we have been constraint to having a **model** ready and discuss their features by update. However, we do not have to decide on what the **pre-processed features** are in advance. We can use data to determine.

VI.1 Projection Pursuit Model

Setup VI.1.1. Simple Unrestricted Function.

We pick m different **linear combinations** of the features $\vec{x}$, namely $\vec{\beta}_m^\intercal \cdot \vec{x}$, where we learn a complex non-linear link function and add the results into:

$$f(\vec{x}) = \sum_{m=1}^{M} g_m\big(\vec{\beta}_m^\intercal \cdot \vec{x}\big)$$

for a set of **unrestricted** $g_m(\bullet)$. ⌟

Note that the output is not a weighted sum of g_m, since the **weights may be absorbed into g_m**. This model is more general than a pure linear model, as it inserts non-linearities via g_m.

Proposition VI.1.2. Universal Approximator.

Suppose that M is large enough, we can learn **any** function with derivatives as closely as desired. Such a function learner is called a **universal approximator**. ⌟

It seems that the existence of **universal approximator** is a contradiction to the **no free lunch** principle, however, they are compatible.

Remark VI.1.3. Universal Approximator is Compatible with No Free Lunch Principle.

While for any true f_0 and a desired degree of accuracy, we can pick a large enough M to approximate, but we do not know f_0 in advance. If we choose poorly (too many or too few M), we will not get good generalization. The assumption to pick M correctly is to know f_0, in which learning is not necessary if we know f_0. ⌟

Now, our goal is to minimize error in a **projection pursuit model**. For simplicity, we let $M = 1$, and we want to minimize:

$$\frac{1}{n}\sum_{i=1}^{n} L\big(y_i, g_1(\vec{\beta}^\intercal \vec{x}; \vec{\eta})\big) = \frac{1}{n}\big(y_i - g_1(\vec{\beta}^\intercal \vec{x}; \vec{\eta})\big)^2,$$

where $\vec{\eta}$ parameterizes our choice of g_1. The natural choice here is to optimize $\vec{\beta}$ while holding $\vec{\eta}$ fixed, and vice versa, which leads to two **update rules** based on derivatives.

Algorithm VI.1.4. Update Rule for Projection Pursuit Model.

Given $\vec{\beta}$ fixed, we create a new data set with features $\tilde{x}_i := \vec{\beta}^\intercal \cdot \vec{x}_i$ for $i = 1, \cdots, n$.

- We fit a univariate regression $Y = g_1(\tilde{x}; \vec{\eta})$.
  For example, if g is differentiable, we solve the score equation for $\vec{\eta}$.
- Given fixed $\vec{\eta}$, we can set up a **Newton-Raphson procedure** using the **Taylor expansion**:
  $$g(\vec{\beta}^\intercal \vec{x}) \simeq g(\vec{\beta}_0^\intercal \cdots \vec{x}) + g'(\vec{\beta}_0^\intercal \cdots \vec{x})(\vec{\beta} - \vec{\beta}_0)^\intercal \vec{x}.$$
- Then, consider the following trick:
  $$\sum_{i=1}^{n} \big(y_i - g(\vec{\beta}^\intercal \vec{x}_i)\big)^2 \simeq \sum_{i=1}^{n} g'(\vec{\beta}_0 \vec{x}_i)^2 \Bigg(\underbrace{\beta_0^\intercal \vec{x}_i + \frac{y_i - g(\vec{\beta}_0^\intercal \vec{x}_i)}{g'(\vec{\beta} 0^\intercal \vec{x}_i)}}_{\text{new pseudo-outcome } \tilde{y}_i} - \vec{\beta}^\intercal \vec{x}_i \Bigg)^2.$$
- This implies that we can solve a weighted linear regression with features $\vec{x}$ and outcome:
  $$\tilde{y} := \vec{\beta}_0^\intercal + \frac{y_i - g(\vec{\beta_0}^\intercal \vec{x})}{g'(\vec{\beta}_0^\intercal \vec{x})},$$
  with each row weighted by $g'(\vec{\beta}_0^\intercal \vec{x}_i)^2$.

We may iterate these steps for $\vec{\beta}$ and $\vec{\eta}$ until convergence. ⌟

VI.2 Neural Network Models

Neural networks are generalization of **linear models** and related to the projection pursuit models. It aims to model the human brain.

The model architecture is by **nesting** the simple linear models (with a link function) together for output of one model serves as input of one another, in **layers**.

Setup VI.2.1. Activation Function and Layers.

The typical form of jth output at the ith layer is to have a function of all outputs from $i-1$th layer:

$$x_{ij} = g(\beta_0 + \vec{\beta}_{i-1}^{\intercal} \vec{x_{i-1}}; \vec{\eta}).$$

Here, the β_0 term is a **bias term** in the neural network model, and $g(\bullet)$ is the **activation function**. ⌟

In the **neural network** model, the first layer are the **input** features and the last layer forms the **output**.

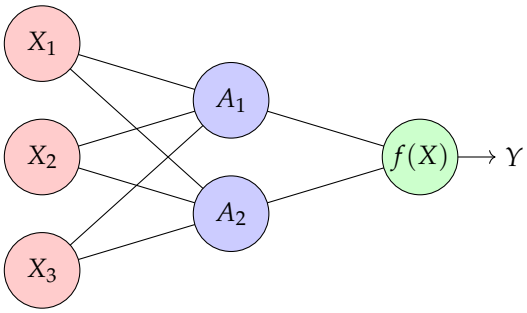


Figure VI.1. A neural network with 3 input features, a second layer with 2 outputs, and the final outcome layer with 1 outcome.

Example VI.2.2. Activation Function.

There are a lot of **activation functions**, the most typical two are:

- **Sigmoid**: $\dfrac{1}{1+e^{-v}}$,
- **ReLU**, Rectified linear unit: $\max\{0, v\}$.

These functions tend to exhibit a **phase transition** from a low to high value after the put increases past a critical points, which is just like how **neurons** works in brain.

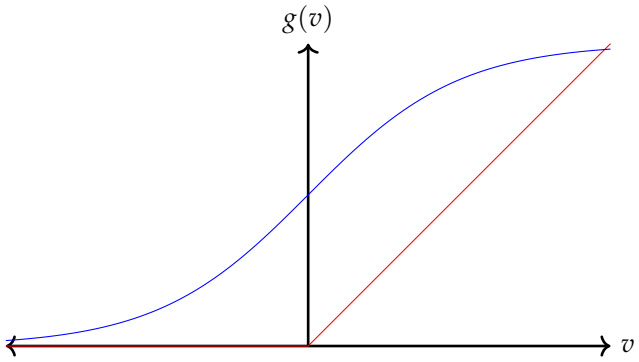


Figure VI.2. Activations functions: ReLU(red) and sigmoid(blue).

⌟

Again, for the neural network, we need a **loss function** and an **optimization algorithm**.

Setup VI.2.3. Loss Function for Neural Network.

For the **regression problems**, we typically use the squared loss:

$$\sum_{i=1}^{n} \left(y_i - f(\vec{x_i})\right)^2.$$

For the **classification problems** with k classes, there are typically k output layer vertices or a since vertex with a **softmax function** $\exp(x_i)/\sum_{j=1}^{k}\exp(x_j)$, with the loss given by:

$$\sum_{i=1}^{n} y_i \log f(\vec{x_i}).$$

Assume that we have a **single output** and squared loss, then the **neural network** model loss with M layers (excluding the last layer, and assuming K_m vertices in the mth layer) is:

$$R(\vec{\beta}) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{2}\big(y_i - f(\vec{x_i})\big)^2 = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{2}\big(y_i - g_M(\underbrace{\vec{\beta}_M^{\mathsf{T}}}_{\text{size }=K_m})\cdot x_{\tilde{i,M}}\big),$$

where:

$$\tilde{x}_{i,m} = \vec{x_i}; \tilde{x}_{i,m} = \Big(g_{1,m}(\vec{\beta}_{1,m}^{\mathsf{T}}\cdot\tilde{x}_{i,m-1}) \quad \cdots \quad g_{K_m,m}(\vec{\beta}_{K_m,m}^{\mathsf{T}}\cdot\tilde{x}_{i,m-1})\Big)^{\mathsf{T}}.$$ ⌟

An obvious approach is to solve $\partial R(\vec{\beta}/\partial\vec{\beta}) = 0$. It will not be solvable in **closed form**, but we may use the **gradient descent**.

VI.3 Backpropagation

Taking derivatives for neural network models is not that bad, since it has regular structure:

- New layer outputs are **linear functions** of the previous layer, and linearity is preserved through **composition**.
- The **activation function** is, hopefully, **differentiable**.

Remark VI.3.1. Chain Rule in Derivatives.

Thus, we will be able to apply the chain rule of differentiation to decompose the overall derivatives into pieces associated with parts of the network diagram.

$$R(\vec{\beta}) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{2}\big(y_i - g_M(\vec{\beta}_M^{\mathsf{T}}\cdot x_{\tilde{M}})\big)^2,$$

$$\frac{\partial R(\vec{\beta})}{\partial\vec{\beta_M}} = -\frac{1}{n}\sum_{i=1}^{n}\underbrace{\big(y_i - g_M(\vec{\beta}_M^{\mathsf{T}}\cdot\tilde{x}_{i,M})\big)g'_M(\vec{\beta}_M^{\mathsf{T}}\cdot\tilde{x}_{i,M})}_{\text{term from chain rule applied at M, denoted }\Delta_M}\cdot\tilde{x}_{i,M}.$$ ⌟

Now, we may consider the partial derivatives with respect to some entries, that is:

$$\frac{\partial R(\vec{\beta})}{\partial\vec{\beta}_{k,m}} = \underbrace{\sum_j \Delta_{j,m+1}\beta_{j,m+1}g'_{k,m}(\vec{\beta}_{1,m}\cdot\tilde{x}_{i,m})}_{\text{terms from chain rule applied at }m\text{, denoted }\Delta_{k,m}}\quad \tilde{x}_{i,k,m}.$$

Therefore, to **calculate derivatives** for every $\vec{\beta}_{k,m}$, we need to calculate $g'_{k,m}$ and Δ_{m+1}.

Strategically, we can store these locally in a **graph data structure** mirroring the network graph. To have this algorithm, we need to decide upon:

(i) Step size. (There are a lot of choices, some steps decay over time.)

(ii) Starting parameter values. (Often randomized around **linear values** of the activation function.)

VI.4 Issues and Improvements on Models

Now, the neural network has **a lot of** parameters, which causes various issues.

Remark VI.4.1. Issues with Neural Network.

(i) **Gradient descent** is very slow.

(ii) The loss surface is **highly non-convex**, so there are a lot of **saddle points** rather than maximum/minimum.

(iii) May **overfit**.

(iv) Some activation functions may not be **differentiable** everywhere.
For example, ReLU is non-differentiable at $v = 0$.

(v) Good solutions to these problems led to the development of **deep learning** based on neural network with many layer, and they are currently the best performing predictors (such as in *vision*, *speech recognition*, etc.).

⌟

Algorithm VI.4.2. Solution to Time Complexity: Stochastic Gradient Descent.

Considering about time efficiency, we use the **stochastic gradient descent** (SGD), which update based on a random small subset of the data (maybe even a single row).
Here, SGD will pick the right direction in **expectation**, since we assumed randomness and it will descent towards the correct direction. The steps could be noisy at first, but it would be faster. ⌟

For **non-convex surfaces**, there is no easy answer, we can try to **restart** the model or use **optimization tricks**.

For differentiability, although ReLU is **not entirely differentiable**, it is faster to compute since it is **piecewise linear**.

For overfitting issues, consider **regularization**, **dropout learning**, and **network tuning**.

Setup VI.4.3. Regularization in Neural Network.

There are many regularization schemes for **neural networks**. A common one is one shared with parametric regression models, by modifying the risk as:

$$R(\vec{\beta}) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{2}\left(y_i - g_M(\vec{\beta}_M^{\mathsf{T}} \cdot \tilde{x}_{i,m})\right)^2 + \lambda \sum_j \beta_j^2.$$

For the multiple layers, we can also regularize parts of the risk separately. ⌟

Note that the penalty term is also **differentiable**, so we can still use GD or SGD.

Setup VI.4.4. Dropout Learning.

During GD or SGD, we randomly choose a fraction ϕ of inputs to ignore, while upweight the other parameters by $1/(1-\phi)$.

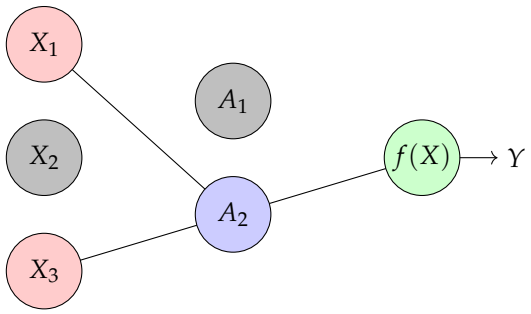


Figure VI.3. Dropping out some inputs and nodes in the layers to ignore.

Note that the dropouts are random, so the dropout learning can be viewed as a **random forest** and **bagging**, since we then combine the results together. ⌟

One application of strategy is to use **data augmentation**.

Setup VI.4.5. Data Augmentation.

One way to avoid **overfitting** is to artificially augment a training data set with additional examples with a **known class**.
Especially for graphic, this can be easily done with images by applying **affine transformations** (rotations, translations, etc.) This will force the model to learn some **other presentations** that are invariant to differences irrelevant to the class. ⌟

Another strategy is to use **network tunning**, since there are lots of **meta-parameters**, that acts like **black arts**. In particular, we want to think of these parameters:

- **Regularization** meta-parameter λ,
- **Network architecture**, *i.e.*, number of hidden layers, nodes per layer, activation functions, etc.
- Choice for **SGD**, which is algorithm for choosing the **subset of data**, **size of the subset**, etc.
- Choices for **data augmentation**.

VI.5 Interpolation and Double Descent

As people try to explain the performances of **deep learning neural network** models, they, according to the theory, should overfit.

Remark VI.5.1. Double Descent.

For the **neural network** models, there is a phenomenon as **double descent**. In particular, the **generalization error** decreases as training fit improves, then increases as model **overfits**, but can then decrease again

as the model is overparameterized. ⌟

One tentative explanation to this phenomenon is that the model matches the **degree of freedom** in the data. There is only **one way** to achieve 0 in training error.
When the model is **very overparameterized**, there are many ways to achieve training error of 0, and by applying regularization, we can get better generalization by picking an **appropriately smooth** way.

VI.6 Convolution Neural Network

Convolution neural networks (CNNs) are a particular type of neural network that specializes **classification tasks** using **images** as inputs.

They work by extracting a set of **low level features** (such as lines separating large color changes, spots, etc.) and combine them into **high level features** that corresponds to parts of objects, and eventually combining into objects.

Setup VI.6.1. Convolution and Pooling.

Convolution elements implement **recognizing** of a feature, whereas pooling elements implement **summarizing** what was found so far. ⌟

These steps can be thought of as doing **feature extractions** before feeding into a regular **neural network** in the final layers.

Definition VI.6.2. Convolution Elements.

A convolution is an element-wise multiplication of sub-matrices in an input matrix by another sub-matrix caled a **convolution filter**. ⌟

Given an original image $\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \\ j & k & l \end{pmatrix}$ and the convolution filter $\begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix}$, their convoluted image becomes:

$$\begin{pmatrix} a\alpha + b\beta + d\gamma + e\delta & b\alpha + c\beta + e\gamma + f\delta \\ d\alpha + e\beta + g\gamma + h\delta & e\alpha + f\beta + h\gamma + i\delta \\ g\alpha + h\beta + j\gamma + k\delta & h\alpha + i\beta + k\gamma + l\delta \end{pmatrix}.$$

If we want to build a **binary recognizer** based on convolution, we can use ReLU unit (or other **activation functions** that get the **convoluted image** as input).

Example VI.6.3. Pooling Elements.

A pooling aims to summarize a **large image** into a **smaller one**.

There are lots of approximation, a common one is by max operator, so it summarizes a sub-matrix by the largest element. It could be useful to recognize a **large value** in a region of an image. ⌟

In general, the CNN architecture has lots of meta-parameter choices, and has convolution and pooling layers **alternating for multiple times**, before the summarized output is fed into a regular (often shallow) **neural network**.

Here, we mention about the special structure of the **recurrent neural network**. It is used for prediction problems where input is a **temporal sequence**.

Setup VI.6.4. Recurrent Neural Networks.

We want a recipe for constructing a **family of unrolled network**, where parameters in each time-step are shared. The parameter learning may be viewed as done on the unrolled network via the usual method, but updating the same parameters at multiple layers.

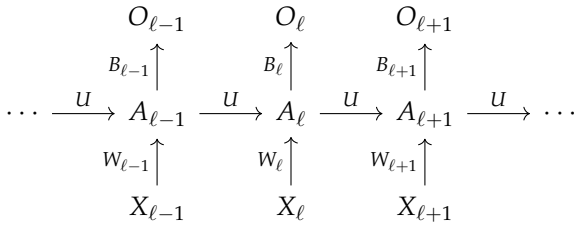


Figure VI.4. Reuse of the data A_ℓ as the neural network unrolls in a chain of complex. ⌟

In particular, the recurrent neural networks can be helpful in **Healthcare data**, **image classification**, **text classification**, **videos**, **time series forecasting**, etc.

Remark VI.6.5. General Notes for Neural Network.

The deep learning neural network are popular but very **complicated models**, they do not work well out of the box. The training is with **enormous data sets** and **long running times** for training to reach the state of the good performance.
It is important to think of many practical problems with simple predictors that often perform just as well. The **simple logistic regression classifier** are performing well in many settings.
Hence, try the simple algorithms first. ⌟

VII Graphical Models

The **Naïve Bayes** assumes that all features are independent given a class label, but the **Markov** model is defined by **conditional independence restrictions**, and is a general and powerful class of models.

Such models are often represented as **graphical models**, which bring visual intuitions to the system. The graphical models are helpful in various fields:

- Input the domains of knowledge into predictive model.
- Learning complex network of associations.
- Causal reasoning and inference.
- Unsupervised learning models.
- Missing data problem.
- Interpretation of existing models.

In particular, **probability** is for reasoning consistently about uncertainty, but is hard to visualize. Here, the **graphs** can represent **independence/irrelevance** in probability distribution, which help with **efficiency**.

VII.1 Markov Random Field

Ernst Ising first developed **undirected graph** to model the spin states in metals.

Setup VII.1.1. Undirected Model for State of Atoms.

Iron atoms have states **up** and **down**, and the magnetized irons has most states pointing in the same direction, while they want to be **like their neighbors**. Hence, in the setup:

- **Vertices** are atoms and **edges** connect the neighbors.
- The corresponding model is that atoms states only depends on **immediate neighbors**.
- The model could be 1D/2D/3D, while higher dimensions means higher complexity.

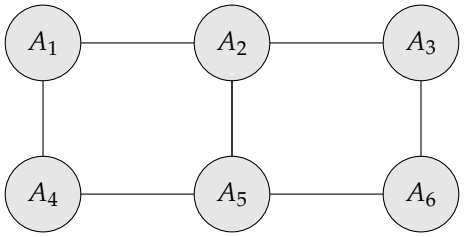


Figure VII.1. Vertices and edges representing the probability model.

With such setup, we can use graphs to represent the relationship of the models rather than using a table.

Definition VII.1.2. Markov Random Field.

Markov random fields (MRFs) are undirected models, which gives probability of any configuration in terms of **local factors**. For Ising and a graph $\mathcal{G}$, we have:

$$\mathbb{P}(a_1, a_2, \cdots, a_k) = \frac{1}{Z} \prod_{A_i \text{ is a node in } \mathcal{G}} \phi(a_i) \prod_{(A_i, A_j) \text{ is an edge in } \mathcal{G}} \phi(a_i, a_j),$$

where ϕ maps A_i to numbers, representing propensity of neighbors to have the same value (this is not probabilities).

Aside, Z is the normalizing factor to make the sum of all products to be 1, which is related to amount of energy in system for Ising. ⌟

At high energy, atoms are disordered, where $\mathbb{P}$ is close to uniform distribution. At low energy, the atoms are arranges as all "up" or all "down".

VII.2 Factorization and Markov Properties

In particular, **distributions** forming a graphical model may be defined in different ways, but a distribution $\mathbb{P}(\vec{x})$ of a graph $\mathcal{G}$ can generally be factorized, or expressed as a product of smaller pieces defined by $\mathcal{G}$. The factorization helps to define a tractable likelihood.

Definition VII.2.1. Global Markov Property.

A distribution $\mathbb{P}(\vec{x})$ in a graphical model of a graph $\mathcal{G}$ obeys the **global Markov property** obeys independence restrictions that can be read off from $\mathcal{G}$ via a criterion involving paths. ⌟

Definition VII.2.2. Local Markov Property.

A distribution $\mathbb{P}(\vec{x})$ in a graphical model of a graph $\mathcal{G}$ obeys the **local Markov property** if it can be defined by a small list of constraints defined by $\mathcal{G}$. ⌟

Note that the **local Markov properties** are helpful for proving a **distribution** is in a model.

Definition VII.2.3. Clique and Maximum Clique.

A **clique** is a set of vertices that are pairwise connected.
A **maximum clique** $\vec{C}$ in a graph $\mathcal{G}$ is a subset of pairwise connected vertices, where no *proper* superset of $\vec{C}$ is a clique. Here, we denote the set of all maximum cliques by $\mathcal{C}$. ⌟

For notation simplicity, we typically call a **maximum clique** a **clique**.

Remark VII.2.4. Clique can Intersect.

Cliques can intersect. For example, with graph $A - B - C$, the cliques are $\{A, B\}$ and $\{B, C\}$. ⌟

Proposition VII.2.5. Factorization of Graph.

Given a graph $\mathcal{G}$, a distribution $\mathbb{P}(\vec{x})$ in the MRF model associated with $\mathcal{G}$ factorizes as:

$$\mathbb{P}(\vec{x}) = \frac{1}{Z} \prod_{\vec{C} \in \mathcal{C}(\mathcal{G})} \phi_{\vec{C}}(\vec{c}),$$

where the choice $\vec{c}$ on edge $\vec{C}$ is arbitrary, and Z is the the normalization factor, *i.e.*:

$$Z = \int \prod_{\vec{C} \in \mathcal{C}(\mathcal{G})} \phi_{\vec{C}}(\vec{c}) d\vec{x},$$

to ensure that $\mathbb{P}(\vec{x})$ sum to 1 for all combinations of $\vec{x}$. ⌟

Again, $\phi_{\vec{C}}(\vec{c})$ is not in general probability distribution and does not necessarily sum to 1.

Example VII.2.6. MRF Factorization of a Graph.

Let a $\mathcal{G}$ graph be defined as:

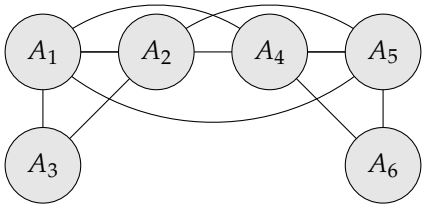


Figure VII.2. Undirected graphical model for $\mathcal{G}$.

Here, the maximum cliques are:

$$\{A_1, A_2, A_3\}, \quad \{A_1, A_2, A_4, A_5\}, \quad \text{and } \{A_4, A_5, A_6\}.$$

Therefore, the probability model is:

$$\mathbb{P}(a_1, a_2, a_3, a_4, a_5, a_6) = \frac{1}{Z} \phi_{A_1, A_2, A_3}(a_1, a_2, a_3) \phi_{A_1, A_2, A_4, A_5}(a_1, a_2, a_4, a_5) \phi_{A_4, A_5, A_6}(a_4, a_5, a_6).$$ ⌟

Definition VII.2.7. Path Separation.

Given three disjoint sets $\vec{A}$, $\vec{B}$, and $\vec{C}$, $\vec{A}$ is **path-separated** from $\vec{B}$ given $\vec{C}$ in $\mathcal{G}$ if for every $A \in \vec{A}$ and $B \in \vec{B}$, every path from A to B is intersected by one or elements of $\vec{C}$, and it is denoted as $\vec{A} \amalg_{\mathcal{G}} \vec{B} \mid \vec{C}$. ⌟

The **path-separation** is an indication of **global Markov property**.

Proposition VII.2.8. Path Separation $\implies$ Conditional Independence.

For every disjoint $\vec{A}$, $\vec{B}$, and $\vec{C}$, $\vec{A} \amalg_{\mathcal{G}} \vec{B} \mid \vec{C}$ implies $\vec{A} \amalg_{\mathbb{P}(\vec{x})} \vec{B} \mid \vec{C}$. ⌟

Note that the converse is not necessarily true.

Example VII.2.9. Example of Path Independence.

In graph demonstrated by *Figure VII.2*:

- $A_1 \amalg A_6 \mid A_2, A_4$ is false, since we have $A_1 - A_5 - A_6$.
- $A_1 \amalg A_6 \mid A_2, A_4, A_5$ is true, although A_2 is unnecessary.

⌟

The **local Markov property** depends only on neighbors.

Proposition VII.2.10. Non-neighbor Independent w.r.t. Neighbors.

Suppose $\mathbb{P}(\vec{x})$ is the MRF model of $\mathcal{G}$, every variable A is independent of **non-neighbors** given all **neighbors**. ⌟

Example VII.2.11. Example of Local Markov Property.

Again, in graph demonstrated by *Figure VII.2*, the following are all **local Markov properties**:

$$A_1 \amalg A_6 \mid A_2, A_3, A_4, A_5,$$
$$A_2 \amalg A_6 \mid A_1, A_3, A_4, A_5,$$
$$A_3 \amalg A_4, A_5, A_6 \mid A_1, A_2,$$
$$A_4 \amalg A_3 \mid A_1, A_2, A_5, A_6,$$
$$A_5 \amalg A_3 \mid A_1, A_2, A_4, A_6,$$
$$A_6 \amalg A_1, A_2, A_3 \mid A_4, A_5.$$ ⌟

In fact, **factorization** and the two **Markov properties** are equivalent.

Theorem VII.2.12. Hammersley-Clifford Theorem.

Given a positive distribution $\mathbb{P}(\vec{x})$, *i.e.*, for all $\vec{x}$, $\mathbb{P}(\vec{x}) > 0$, the following are equivalent:

(i) $\mathbb{P}(\vec{x})$ factorizes with respect to $\mathcal{G}$,

(ii) $\mathbb{P}(\vec{x})$ obeys the **global Markove property** with respect to $\mathcal{G}$, and

(iii) $\mathbb{P}(\vec{x})$ obeys the **local Markove property** with respect to $\mathcal{G}$. ⌟

VII.3 Directed Models

Sewall Wright developed **directed models** for pedigree analysis for animals. Animals inherit traits from parents probabilistically. The goal is to formalize the degree of inbreeding.

Setup VII.3.1. Directed Model and Degree of Inbreeding.

In the model, we have various arrows:

- $\to$ means **causal**, that is, the parent trait cause child traits.
- $\leftrightarrow$ means **correlation** of noise, and it makes the model more complex.

Moreover, for the graph to work, we enforce that there must be no cycles, *i.e.*, one cannot be their own ancestor. ⌟

In Wright's model of related variables Y and X such that $X \to Y$ is in $\mathcal{G}$, then using a linear regression:

$$Y = w_0 + \sum_{X_i} w_i \cdot X_i + \epsilon_Y.$$

Here, the model allows **correlation between noise terms**, which is represented as:

$$\operatorname{cov}(\epsilon_{Y_i}, \epsilon_{Y_k}) \neq 0,$$

which is represented as $Y_i \leftrightarrow Y_j$.

Setup VII.3.2. Path Analysis.

The inbreeding coefficient of two variables is the **measure of dependence** in the model, which is the sum of contributions from relevant paths in graph. The **paths** must satisfy that:

- Must trace **back** then **forward**, and not the other way around.
- Must not pass a **variable** twice.
- Can pass at most one **bi-directed edge** per path.

Therefore, the **contribution** of a path is the product of all coefficients along the path. ⌟

Example VII.3.3. Path Analysis Example.

Here, we define a directed graph $\mathcal{G}$ as follows:

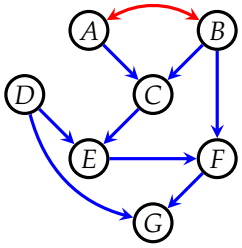


Figure VII.3. Directed graphical model for $\mathcal{G}$.

Here, to evaluate the inbred factor between x_E and x_F, we find all relevant paths:

(i) $E \leftarrow C \leftarrow A \leftrightarrow B \rightarrow F$,

(ii) $E \leftarrow C \leftarrow B \rightarrow F$, and

(iii) $E \rightarrow F$.

Hence, the inbreeding coefficient is:

$$f = \psi(C,E)\psi(A,C)\psi(A,B)\psi(B,F) + \psi(C,E)\psi(B,C)\psi(B,F) + \psi(E,F),$$

where $\psi(X,Y)$ represents the association factor of the **causal** relationship of X to Y. ⌟

The **Structural Equation Models** (SEMs) has applications on social sciences and economics, and the computer programs has **iterative fitting**. It applies to areas that may not necessarily be causal.

Example VII.3.4. Hidden Markov Model.

An example of directed model is the **hidden Markov model**, which was developed for speech processing. The model has a hidden state of known complexity evolving in discrete time. The model can be represented as:

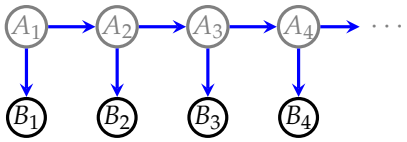


Figure VII.4. Model of hidden Markov model for speech recognition.

The model can be modeled by the conditional probability as:

$$\mathbb{P}(A_i \mid A_{i-1}) \text{ and } \mathbb{P}(B_i \mid A_i).$$

In descriptive language, A_i are the words said and B_i are the speech sound wave.
The **Viterbi** is an efficient algorithm for such model. ⌟

There are also many other **special cases**. The **directional models** are different from MRFs, and the models could be either statistical causal.

VII.4 Bayesian Networks

Setup VII.4.1. Terminologies of Graph.

Given a directed graph, we have:

- $\mathrm{pa}_{\mathcal{G}}(\bullet)$ representing the set of parents,
- $\mathrm{ch}_{\mathcal{G}}(\bullet)$ representing the set of children,
- $\mathrm{an}_{\mathcal{G}}(\bullet)$ representing the set of ancestors, and
- $\mathrm{de}_{\mathcal{G}}(\bullet)$ representing the set of descendants. ⌟

One special model is the **Bayesian network**.

Example VII.4.2. Sample Bayesian Network.

Here, we define a Bayesian network graph $\mathcal{G}$ as follows:

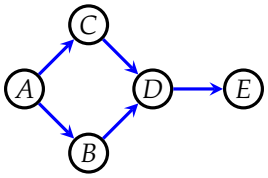


Figure VII.5. Bayesian network model for $\mathcal{G}$.

In the graph, according to the path $A \to C \to D \to E$, we have:

- $A, C, D \in \mathrm{an}_{\mathcal{G}}(E)$, $D \in \mathrm{pa}_{\mathcal{G}}(E)$,
- $C, D, E \in \mathrm{de}_{\mathcal{G}}(A)$, and $C \in \mathrm{ch}_{\mathcal{G}}(A)$. ⌟

Definition VII.4.3. Bayesian Network.

A **Bayesian network** model links probability distributions and a **directed acyclic graph** (DAG), which is similar to MRFs. It must satisfy that:

- **Directed**: It must only contain $\to$ edges.
- **Acyclic**: If $X \in \mathrm{de}_G(Y)$, then $Y \notin \mathrm{ch}_G(X)$.

⌟

Different from the **Directed graph**, $\to$ in **Bayesian model** is not causal.

Proposition VII.4.4. Factorization in Bayesian Network.

Given a distribution, the factorization has a term for a conditional distribution of a variable given its parents. ⌟

Example VII.4.5. Bayesian Network of Factorization.

In graph demonstrated by *Figure VII.5*, we have the factorization as:

$$\mathbb{P}(A, B, C, D, E) = \mathbb{P}(E \mid D)\mathbb{P}(D \mid B, C)\mathbb{P}(C \mid A)\mathbb{P}(B \mid A)\mathbb{P}(A).$$ ⌟

Note that by using such approach, if the graph has fewer edges, it needs fewer parameters.

Remark VII.4.6. Reduce Use of Parameters.

In the above example, assume it is binary model, then the total distribution (*left*) needs $2^5 - 1 = 31$ parameters, but the factorized model (*right*) needs only $2^1 + 2^2 + 2^1 + 2^1 + 1 = 11$ parameters. ⌟

Bayesian network models also has **Markov properties** as of **undirected graph**, *i.e.*, graph implies a small list of independences that imply the rest.

Proposition VII.4.7. Local Markov Property.

Every X is independent of non-parental, non-descendants, conditional on parents. ⌟

Example VII.4.8. Local Markov Property on Bayesian Model.

In graph demonstrated by *Figure VII.5*, for the node C, we have:

$$C \amalg B \mid A.$$ ⌟

Definition VII.4.9. Observational Equivalence.

Two graphs are called **observational equivalent** if their **local Markov property** gives the same independence. ⌟

For example, the graphs $A \to B \to C$ and $A \leftarrow B \leftarrow C$ are **observational equivalent**, as they have $A \amalg C \mid B$ as the only independence.

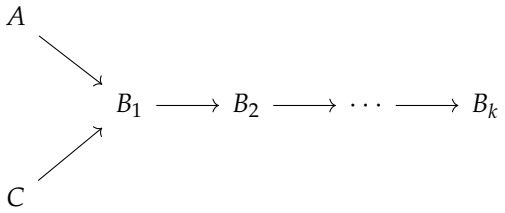
Definition VII.4.10. Blocking of Triplets.

For the **blocking of triplets**, we need to distinguish by cases:

- **Directed triplet**:
$$A \to B \to C,$$
if unobserved, we have $A \not\amalg C$, if observed, then $A \amalg C \mid B$.
In causal intuition, if B is a noisy version of A and C is a noisy version of B, then C is a noisy version of A.

- **Split triplet**:
$$A \leftarrow B \to C,$$
if unobserved, we have $A \not\amalg C$, if observed, then $A \amalg C \mid B$.
In a causal intuition, if A and C share a common cause, they become dependent.

- **Collider triplet**:
$$A \to B \leftarrow C,$$
if unobserved, we have $A \amalg C$, if observed, then $A \not\amalg C \mid B$.
In a causal intuition, A and C are two independent causes of an effect, but knowing a shared effect creates the dependence of cause.

- Note on **Descendants of Colliders**:


as long as B_k is observed, we have $A \not\amalg C \mid B_k$ down the chain. ⌟

Here, we want to use the triplets to form **path blocking** and **d-separation**.

Definition VII.4.11. Path Blocking and d-separation.

A path from A to B is **blocked** by $\vec{C}$ if there is a blocking triplet on the path. They are said to be **d-separated** given $\vec{C}$ if all paths from A to B are block by $\vec{C}$. ⌟

Example VII.4.12. d-separation in Bayesian Network.

Again, from the graph demonstrated by *Figure VII.5*, we have:

- $A \amalg E \mid C$ is false, since $A \to B \to D \to E$ is not blocked.
- $C \amalg B \mid A$ is true, since $C \leftarrow A \to B$ and $C \to D \leftarrow B$ are blocked.

Here, we denote $(A \amalg_d B \mid \vec{C})_{\mathcal{G}}$ as A being **d-separated** from B given $\vec{C}$ in $\mathcal{G}$, and this can be extended to $(\vec{A} \amalg_d \vec{B} \mid \vec{C})_{\mathcal{G}}$ if $(A \amalg_d B \mid \vec{C})_{\mathcal{G}}$ for all $A \in \vec{A}$ and $B \in \vec{B}$.

Proposition VII.4.13. Global Markov Property.

Suppose $\mathcal{G}(\vec{V})$ is a Bayesian network with DAG, then $(A \amalg_d B \mid \vec{C})_{\mathcal{G}(\vec{V})}$ implies $A \amalg_{\mathbb{P}(\vec{V})} B \mid \vec{C}$.

Note that the converse is also not necessarily true, since there could be extra independences in $\mathbb{P}(\vec{V})$.

Theorem VII.4.14. Verma and Peral.

Given a DAG $\mathcal{G}(\vec{V})$, the following are equivalent:

(i) the distribution $\mathbb{P}(\vec{V})$ factorizes according to $\mathcal{G}$,

(ii) the distribution obeys the **local Markov property** according to $\mathcal{G}$, and

(iii) the distribution obeys the **global Markov property** according to $\mathcal{G}$.

The **Bayesian network** is a generalization of **Naïve Bayes**, so we can consider the following.

Remark VII.4.15. Naïve Bayes is a Special Bayesian Network.

The Naïve Bayes may be viewed as a DAG model, that is:

$$\mathbb{P}(\vec{x}, y) = \mathbb{P}(y)\mathbb{P}(x_1 \mid y) \cdots \mathbb{P}(x_k \mid y).$$

Graphically, it is:

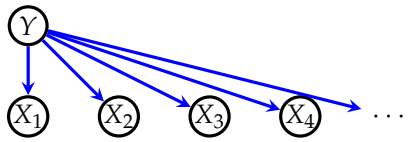


Figure VII.6. Model of hidden Markov model for speech recognition.

VII.5 Computation of Marginal Likelihood

Assume that we are using $\mathbb{P}(\vec{x}, y)$ in a graphical model form some graph $\mathcal{G}$ for prediction, it can be done using maximizing the likelihood, that is $\mathbb{E}[y \mid \vec{x}]$.

Example VII.5.1. Likelihood for MRF is Computationally Intense.

Consider the MRF with graph $\mathcal{G}$, its likelihood would be:

$$\prod_{i=1}^{n} \mathbb{P}(\vec{x}_i, y_i) = \prod_{i=1}^{n} \frac{1}{Z} \prod_{\vec{C} \in \mathcal{C}(\mathcal{G})} \phi_{\vec{C}}(\vec{c}_i).$$

The problem with this computation is its **computation complexity**. To compute Z, we need to sum over all values of $\vec{x}$ and y, and given k number of binary features and binary outcome, it requires w^{201} operations, and this also occurs when computing conditional expectations. ⌟

Now, if we consider the computation of marginals in a chain, we may take advantage if conditional independence of the model.

Example VII.5.2. Computing Marginal in Chain for DAG by Conditional Independence.

Consider the chain of the following DAG:

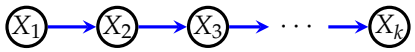


Figure VII.7. DAG of a chain.

Therefore, we may consider the constraints of the model as:

$$x_i \amalg x_1, \cdots, x_{i-2} \mid x_{i-1}.$$

Therefore, we may represent the **marginal probability** as:

$$\begin{aligned}
\mathbb{P}(x_k) &= \sum_{x_1,\cdots,x_{k-1}} \mathbb{P}(x_1, \cdots, x_k) \\
&= \sum_{x_1,\cdots,x_{k-1}} \mathbb{P}(x_1) \prod_{i=2}^{k} \mathbb{P}(x_i \mid x_1, \cdots, x_{i-1}) && \text{(By chain rule.)} \\
&= \sum_{x_1,\cdots,x_{k-1}} \mathbb{P}(x_1) \prod_{i=2}^{k} \mathbb{P}(x_i \mid x_{i-1}) && \text{(By model constraits.)} \\
&= \sum_{x_2,\cdots,x_{k-1}} \underbrace{\left(\sum_{x_1} \mathbb{P}(x_1)\mathbb{P}(x_2 \mid x_1) \right)}_{\mathbb{P}(x_2)} \prod_{i=3}^{k} \mathbb{P}(x_i \mid x_{i-1}) = \sum_{x_2,\cdots,x_{k-1}} \mathbb{P}(x_2) \prod_{i=3}^{k} \mathbb{P}(x_i \mid x_{i-1}) \\
&= \sum_{x_3,\cdots,x_{k-1}} \underbrace{\left(\sum_{x_2} \mathbb{P}(x_2)\mathbb{P}(x_3 \mid x_2) \right)}_{\mathbb{P}(x_3)} \prod_{i=4}^{k} \mathbb{P}(x_i \mid x_{i-1}) = \cdots = \mathbb{P}(x_k).
\end{aligned}$$

Therefore, by the approach, the sum of each marginal probability is a constant of operations repeated for $\mathcal{O}(k)$ times, so the computation is **linear** (compared to $\mathcal{O}(2^k)$ for naïve approach). ⌟

Similarly, this also applies to **MRF chains**, *i.e.*, for the undirected graphs.

Example VII.5.3. Computing Marginal in MRF Chain by Conditional Independence.

Consider the chain of the following DAG:

Figure VII.8. MRF of a chain.

Therefore, we may consider the constraints of the model as:

$$x_i \amalg x_1, \cdots, x_{i-2}, x_{i+2}, \cdots, x_k \mid x_{i-1}, x_{i+1}.$$

Therefore, we may form the model as:

$$\begin{aligned}
Z \cdot \mathbb{P}(x_k) &= Z \cdot \sum_{x_1,\cdots,x_{k-1}} \mathbb{P}(x_1,\cdots,x_k) \\
&= \sum_{x_1,\cdots,x_{k-1}} \prod_{i=1}^{k-1} \phi_{X_i,X_{i+1}}(x_i,x_{i+1}) && \text{(By model factorization)} \\
&= \sum_{x_2,\cdots,x_{k-1}} \underbrace{\left(\sum_{x_1} \phi_{X_1,X_2}(x_1,x_2)\right)}_{\phi^*_{X_2}(x_2)} \prod_{i=2}^{k-1} \phi_{X_i,X_{i+1}}(x_i,x_{i+1}) = \sum_{x_2,\cdots,x_{k-1}} \phi^*_{X_2}(x_2) \prod_{i=2}^{k-1} \phi_{X_i,X_{i+1}}(x_i,x_{i+1}) \\
&= \sum_{x_3,\cdots,x_{k-1}} \underbrace{\left(\sum_{x_2} \phi_{X_2,X_3}(x_2,x_3)\right)}_{\phi^*_{X_3}(x_3)} \prod_{i=3}^{k-1} \phi_{X_i,X_{i+1}}(x_i,x_{i+1}) = \cdots = \psi^*_{X_k}(x_k).
\end{aligned}$$

Therefore, we have that:

$$\mathbb{P}(x_k) = \frac{\phi^*_{X_k}(x_k)}{Z} = \frac{\phi^*_{X_k}(x_k)}{\sum_{x_k} \phi^*_{X_k}(x_k)}.$$

Here, we also simply into the sum of each marginal probability being a constant of operations repeated for $\mathcal{O}(k)$ times, so the computation is **linear** (compared to $\mathcal{O}(2^k)$ for naïve approach). ⌟

VII.6 Message Passing

It is noteworthy to consider finding the **marginal probability** at arbitrary node, here, we consider the same formula as:

$$Z \cdot \mathbb{P}(x_i) = \sum_{x_{i-1},x_{i+1}} \phi^*_{X_{i-1}}(x_{i-1}) \phi_{X_{i-1},X_i}(x_{i-1},x_i) \phi^*_{X_{i+1}}(x_{i-1}) \phi_{X_i,X_{i+1}}(x_i,x_{i+1}),$$

which implies that:

$$\begin{aligned}
\mathbb{P}(x_i) &= \frac{\sum_{x_{i-1},x_{i+1}} \phi^*_{X_{i-1}}(x_{i-1}) \phi_{X_{i-1},X_i}(x_{i-1},x_i) \phi^*_{X_{i+1}}(x_{i-1}) \phi_{X_i,X_{i+1}}(x_i,x_{i+1})}{Z} \\
&= \frac{\sum_{x_{i-1},x_{i+1}} \phi^*_{X_{i-1}}(x_{i-1}) \phi_{X_{i-1},X_i}(x_{i-1},x_i) \phi^*_{X_{i+1}}(x_{i-1}) \phi_{X_i,X_{i+1}}(x_i,x_{i+1})}{\sum_{x_{i-1},x_i,x_{i+1}} \phi^*_{X_{i-1}}(x_{i-1}) \phi_{X_{i-1},X_i}(x_{i-1},x_i) \phi^*_{X_{i+1}}(x_{i-1}) \phi_{X_i,X_{i+1}}(x_i,x_{i+1})}.
\end{aligned}$$

Remark VII.6.1. Calculating Margin for Arbitrary Node.

Such procedure is universal for any X_i, hence we may calculate all marginals in a single pass by finding the left and right of the chain. We consider the things passed as **messages**, where the **message** from left is $\phi_{\vec{X}_i}(x_i)$ and the **message** from right is $\phi_{\overleftarrow{X}_i}(x_i)$.
Such method is called **message passing**. ⌟

Algorithm VII.6.2. Message Passing for Chain.

The message passing works for marginal likelihood as:

$$\mathbb{P}(x_1) = \frac{\phi_{\overleftarrow{X}_1}(x_1)}{\sum_{x_1} \phi_{\overleftarrow{X}_1}(x_1)}, \qquad \mathbb{P}(x_i) = \frac{\phi_{\overleftarrow{X}_i}(x_i) \phi_{\vec{X}_i(x_i)}}{\sum_{x_i} \phi_{\overleftarrow{X}_i}(x_i) \phi_{\vec{X}_i(x_i)}}, \qquad \mathbb{P}(x_k) = \frac{\phi_{\vec{X}_k}(x_k)}{\sum_{x_k} \phi_{\vec{X}_k}(x_k)}.$$

Note that by this algorithm, **message passing** allows us to calculate Z and all univariate margins in **linear time and space**. ⌟

Also, such algorithm works for a **tree** as a generalized case than a **chain**.

Algorithm VII.6.3. Message Passing for Tree.

Consider a fragment of a undirected tree (with root X) as:

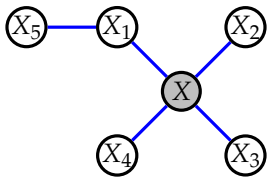


Figure VII.9. Part of the MRF of a tree.

Now, suppose that we are computing $\mathbb{P}(x_r)$, we consider the **inbound message** from all leaves of the tree towards the root, and the **outbound message** away from the root towards the leaves.
Here, we consider X as the root, so we have the following example of messages:

- $\phi_{X_1}^{X_5 \to X_1}(x_1)$ is an **inbound message** from X_1 to X.
- $\phi_{X_5}^{X_5 \leftarrow X_1}(x_5)$ is an **outbound message** from X to X_1.

For a particular **inbound** message, we may recursive define it (from leaf to root) as:

$$\phi_{X_m}^{X_k \to X_m}(x_m) = \sum_{x_k} \prod_{\substack{Xi \neq X_m \\ X_i \text{ is a neighbor of } X_k}} \phi_{X_k}^{X_i \to X_k}(x_k) \phi_{X_k, X_m}(x_k, x_m).$$

Correspondingly, the margin of the root is:

$$\mathbb{P}(x_r) = \frac{\prod_{X_n \text{ is a neighbor of } X_r} \phi_{X_r}^{X_n \to X_r}(x_r)}{\sum_{x_r} \prod_{X_n \text{ is a neighbor of } X_r} \phi_{X_r}^{X_n \to X_r}(x_r)}.$$

For a particular **outbound** message, we may recursive define it (from root to leaf) as:

$$\phi_{X_k}^{X_k \leftarrow X_m}(x_k) = \sum_{x_m} \phi_{X_k, X_m}(x_k, x_m) \phi_{X_m}^{X_m \leftarrow X_d}(x_m) \prod_{\substack{Xi \neq X_k, X_i \neq X_d \\ X_i \text{ is a neighbor of } X_m}} \phi_{X_m}^{X_i \to X_m}(x_m),$$

where X_d is the neighbor of X_m that is closer to the root X_r than X_m (*exists uniquely since the graph is a tree*).
Correspondingly, the margin of any node (X_j, that is not root) is:

$$\mathbb{P}(x_j) = \frac{\left(\prod_{\substack{X_i \neq X_d \\ X_i \text{ is a neighbor of } X_j}} \phi_{X_j}^{X_i \to X_j}(x_j) \right) \phi_{X_j}^{X_i \leftarrow X_d}(x_j)}{\sum_{x_j} \left(\prod_{\substack{X_i \neq X_d \\ X_i \text{ is a neighbor of } X_j}} \phi_{X_j}^{X_i \to X_j}(x_j) \right) \phi_{X_j}^{X_i \leftarrow X_d}(x_j)}.$$

⌟

In this case, we may compute all conditionals in a univariate tree for MRF.

For binary variables, we form a table of size $2^{|\vec{X}|}$ and we consider the case for $\mathbb{P}(\vec{X} \setminus \vec{E}, \vec{E} = \vec{e})$, in which we need a table of size $2^{|\vec{X} \setminus \vec{E}|}$ as the set of probabilities, then we set the probabilities of rows incompatible

with $\vec{e}$ to zero, and normalize the other probabilities.

Example VII.6.4. Binary Univariate Tree MRF.

Consider the original table as:

x_1	x_2	$\mathbb{P}(x_1, x_2)$
-1	-1	p_1
-1	1	p_2
1	-1	p_3
1	1	p_4

and with the given condition that $x_2 = 1$, we have:

x_1	x_2	$\mathbb{P}(x_1, x_2 = 1)$
-1	-1	0
-1	1	p_2
1	-1	0
1	1	p_4

In particular, we have:

- $\mathbb{P}(x_2 = 1) = p_2 + p_4$,
- $\mathbb{P}(x_1 = -1 \mid x_2 = 1) = \frac{p_2}{p_2+p_4}$, and $\mathbb{P}(x_1 = 1 \mid x_2 = 1) = \frac{p_4}{p_2+p_4}$. ⌟

Setup VII.6.5. General Setup for Univariate Tree MRF.

Assume that $\mathbb{P}(\vec{X})$ is an MRF w.r.t $\mathcal{G}$ that is a univariate tree, then all ϕ are 2 tables of $\phi_{X_i,X_j}(x_i, x_j)$ in such MRF. To calculate $\mathbb{P}(\vec{X} \setminus \vec{E} \mid \vec{E} = \vec{e})$, we follow that:

- Find all **clique** terms $\phi_{X_i,X_j}(x_i, x_j)$ where $X_i \in \vec{E}$ or $X_j \in \vec{E}$ as **incompatible rows**,
- set all **incompatible rows** with $\vec{e}$ in the table to 0, and
- Calculate all univariate margins as before.

In such circumstances, this will yield $\mathbb{P}(X_i \mid \vec{E} = \vec{e})$. Note that if $X_i \in \vec{E}$, the value is 1 for $\mathbb{P}(X_i = x_i)$. ⌟

VII.7 Clique Tree and Chordal Graphs

Definition VII.7.1. Chordal.

An undirected graph is called a **chordal** if any cycle of size 4 or more have an edge not in the cycle connecting vertices in the cycle.

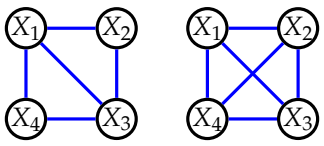


Figure VII.10. Examples of Chordals in 4 cycle.

Chordal graphs are nice since they allow us to construct a **clique tree** allowing us to perform **message passing** efficiently. ⌟

Algorithm VII.7.2. Clique Tree Decomposition for MRF.

Given a undirected graph $\mathcal{G}$ with vertices $\vec{V}$, a **tree decomposition** is a tree with vertices $\mathbb{V}$ and edges $\mathbb{E}$ such that:

(i) Every $\vec{S} \in \mathbb{V}$ is a subset of $\vec{V}$,

(ii) The union of all $\vec{S} \in \mathbb{V}$ is $\vec{V}$, *i.e.*, $\bigcup_{\vec{S} \in \mathbb{V}} \vec{S} = \vec{V}$,

(iii) If there is an edge $\epsilon \in \mathbb{E}$ connecting $\vec{S_1}$ and $\vec{S_2}$, then there are vertices in $\vec{V}$ in common to both $\vec{S_1}$ and $\vec{S_2}$ (recall that $\vec{S_1}, \vec{S_2} \subset \vec{V}$), and

(iv) It follows **running intersection property**: for any $\vec{S_1}$ and $\vec{S_2}$, any vertices in $\vec{V}$ they have in common also occur in $\vec{S_i} \in \mathbb{V}$ on the unique path in the tree from $\vec{S_1}$ to $\vec{S_2}$.

⌟

Example VII.7.3. Forming Clique Tree from Chordal Graph.

Consider the following **chordal graph**:

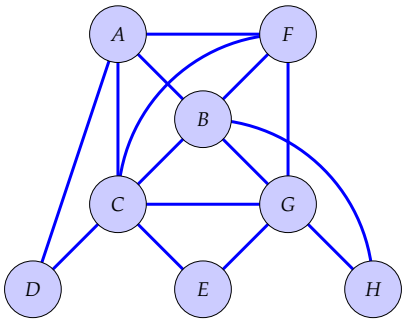


Figure VII.11. Chordal graph with 5 maximum cliques.

The **clique tree** it forms is:

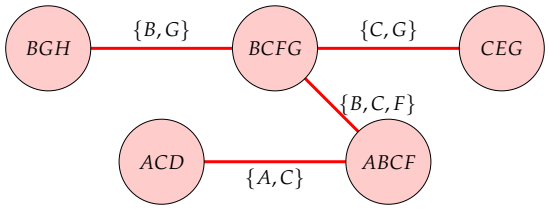


Figure VII.12. Clique tree formed from the chordal graph.

⌟

In particular, **chordal graphs** has nice properties as we can construct a tree decomposition efficiently as a clique tree. Moreover, we may define a **partial order** on the set of vertices.

Definition VII.7.4. Prefect Ordering of Vertices.

A perfect ordering $\prec$ of the vertices satisfies that for every V, every neighbor is bigger than V according to $\prec$ if it forms a clique with V. ⌟

Algorithm VII.7.5. Construction of the Chordal Graph.

We use the following steps:

(i) Find a perfect ordering of $\mathcal{G}$,

(ii) Use the perfect ordering to find all maximal clique in $\mathcal{G}$, and these will be vertices in $\mathbb{V}$, and

(iii) Use a spanning tree algorithm to construct $\mathbb{E}$ connection $\mathbb{V}$, where the edge weight for ϵ connecting $\vec{S}_1$ and $\vec{S}_2$ as the number of nodes in $\vec{V}$ that is shared by $\vec{S}_1$ and $\vec{S}_2$. ⌟

Similar to the previous goal, we still want to compute **margins or conditionals** efficiently. In particular, we calculate $\mathbb{P}(\vec{s})$ for every $\vec{S} \in \mathcal{C}(\mathcal{G})$.

Definition VII.7.6. Incoming and Outgoing Messages.

We can compute the **incoming messages** from $\vec{S_k}$ to neighbor $\vec{S_m}$ closer to the root $\vec{S_k}$ as:

$$\phi_{\vec{S_m}\cap\vec{S_k}}^{\vec{S_k}\to\vec{S_m}}(\vec{s_m}\cap\vec{s_k}) = \sum_{\vec{S_k}\setminus\vec{S_m}} \phi_{\vec{S_k}}(\vec{s_k}) \left(\prod_{\substack{\vec{S_i}\neq\vec{S_m} \\ \vec{S_i} \text{ is a neighbor of } \vec{S_k}}} \phi_{\vec{S_k}\cap\vec{S_i}^{\vec{S_i}\to\vec{S_k}}}(\vec{s_k}\cap\vec{s_i}) \right).$$

Similarly, we can compute the **outgoing messages** from $\vec{S_m}$ to neighbor $\vec{S_k}$ further from the root $\vec{S_k}$ as:

$$\phi_{\vec{S_m}\cap\vec{S_k}}^{\vec{S_k}\leftarrow\vec{S_m}}(\vec{s_m}\cap\vec{s_k}) = \sum_{\vec{S_m}\setminus\vec{S_k}} \phi_{\vec{S_m}}(\vec{s_m})\phi_{\vec{S_m}\cap\vec{S_d}}^{\vec{S_m}\leftarrow\vec{S_d}}(\vec{s_m}\cap\vec{s_d}) \left(\prod_{\substack{\vec{S_i}\neq\vec{S_k},\vec{S_i}\neq\vec{S_d} \\ \vec{S_i} \text{ is a neighbor of } \vec{S_m}}} \phi_{\vec{S_m}\cap\vec{S_i}^{\vec{S_i}\to\vec{S_m}}}(\vec{s_m}\cap\vec{s_i}) \right). \quad ⌟$$

Proposition VII.7.7. Computing Margins in a Clique Tree MRF.

Here, let $\mathcal{G}$ be a **clique tree** MRF, for each leaf node $\vec{S_i}$ with a neighbor node $\vec{S_j}$, the margin is:

$$\mathbb{P}(\vec{S_i}) = \frac{\phi_{\vec{S_j}\setminus\vec{S_i}}^{\vec{S_i}\leftarrow\vec{S_j}}\phi_{\vec{S_i}}}{\sum_{\vec{S_i}}\phi_{\vec{S_j}\setminus\vec{S_i}}^{\vec{S_i}\leftarrow\vec{S_j}}\phi_{\vec{S_i}}} = \frac{\sum_{\vec{V}\setminus\vec{S_i}}\prod_{C\in\mathcal{C}(\mathcal{G})}\phi_C}{\sum_{\vec{V}}\prod_{C\in\mathcal{C}(\mathcal{G})}\phi_C}.$$

For each non-leaf note $\vec{S_i}$ with a neighbor $\vec{S_j}$ closer to the root, and neighbors $\vec{S}_1,\ldots,\vec{S}_m$ further from the root, the margin is:

$$\mathbb{P}(\vec{S_i}) = \frac{\phi_{\vec{S_j}\setminus\vec{S_i}}^{\vec{S_i}\leftarrow\vec{S_j}}\phi_{\vec{S_i}}\left(\prod_{k=1}^m\phi_{\vec{S_k}\setminus\vec{S_i}}^{\vec{S_k}\to\vec{S_i}}\right)\phi_{\vec{S_i}}}{\sum_{\vec{S_i}}\phi_{\vec{S_j}\setminus\vec{S_i}}^{\vec{S_i}\leftarrow\vec{S_j}}\phi_{\vec{S_i}}\left(\prod_{k=1}^m\phi_{\vec{S_k}\setminus\vec{S_i}}^{\vec{S_k}\to\vec{S_i}}\right)\phi_{\vec{S_i}}} = \frac{\sum_{\vec{V}\setminus\vec{S_i}}\prod_{C\in\mathcal{C}(\mathcal{G})}\phi_C}{\sum_{\vec{V}}\prod_{C\in\mathcal{C}(\mathcal{G})}\phi_C}.$$

For the root node $\vec{S}_i$ with neighbors $\vec{S}_1, \ldots, \vec{S}_m$, the margin is:

$$\mathbb{P}(\vec{S}_i) = \frac{\left(\prod_{k=1}^{m} \phi_{\vec{S}_k \setminus \vec{S}_i}^{\vec{S}_k \to \vec{S}_i}\right) \phi_{\vec{S}_i}}{\sum_{\vec{S}_i} \left(\prod_{k=1}^{m} \phi_{\vec{S}_k \setminus \vec{S}_i}^{\vec{S}_k \to \vec{S}_i}\right) \phi_{\vec{S}_i}} = \frac{\sum_{\vec{V} \setminus \vec{S}_i} \prod_{C \in \mathcal{C}(\mathcal{G})} \phi_C}{\sum_{\vec{V}} \prod_{C \in \mathcal{C}(\mathcal{G})} \phi_C}.$$

The algorithm is **linear** in the tree decomposition graph, but is **exponential** in the size of the cliques (since only the largest clique matters, and the size of the largest clique is called the **treewidth**).

Also, if the graph has no missing edges, then the entire graph is a big **maximum clique**, and the algorithm will be **exponential**.

Remark VII.7.8. Edge Cases in Chordal Trees.

There are some special cases where we want to make the graph into a **Chordal tree**.

- If the graph is **not chordal**, we can always add edges to make it chordal. (*Potentially many ways to make it chordal.*)
  We want to pick the optimal way so that **treewidth** is as small as possible. (However, this is NP-hard).
- If the graph is a DAG, we can form a **moralized** or **augmented** undirected graph $\mathcal{G}^a$ by adding an undirected edge to any non-adjacent vertices with a child in common, and then drop all the edge orientations.
  Then, we assign the distributions $\mathbb{P}\big(V \mid \mathrm{pa}_{\mathcal{G}}(V)\big)$ to any clique in $\mathcal{G}^a$ containing V and $\mathrm{pa}_{\mathcal{G}}(V)$. (*If any clique has no factors, it will be assigned with a trivial factor 1.*)

VIII Semi-Supervised Learning and Unsupervised Learning

Recall from **supervised learning**, we consider it as learning a function, whereas in **semi-supervised learning**, there are some feedback, but there are less of them than supervised learning. Some examples are:

- **Missing outcomes**: learning a function from partial examples, where $\vec{x_o}$ are always observed but y_i are sometimes observed and sometimes not.
- **Missing data**: learning from $\vec{x_i}$ and y_i where some entries (either feature or outcomes) are censored.
- **Reinforcement learning**: where agents acting in an environment, where the environment signals a reward/punishment. (*Agents aiming to learn which actions lead to high rewards.*)
- **Causal machine learning**: use observed data to learn cause-effect relationships.

Unsupervised learning looks for patterns in the data without any supervision, some examples are:

- Clustering data points into groups.
- Learning a graphical model from data.
- Dimension reduction.

VIII.1 Clustering Problem

In a **clustering problem**, we have a set of points $\vec{x}$ without **class labels**. We want to cluster points into a set of k **similar groups**.

We cannot set up a loss using class labels, as they do not exist. What we can do is to define how similar $\vec{x_i}$ and $\vec{x_j}$ are, and put **similar points** into one group. The question is how to define similarity and how to make this an optimization problem.

Definition VIII.1.1. Metric Space.

Assume $\mathbf{x} \in \mathbb{R}^k$, we define a distance metric $d : \mathbb{R}^k \times \mathbb{R}^k \to \mathbb{R}_{>0}$ so that for all $\vec{x_i}, \in \mathbb{R}^k$, d satisfies:

- **Positivity**: $d(\vec{x_1}, \vec{x_2}) \geq 0$,
- **Definiteness**: $d(\vec{x_1}, \vec{x_2}) = 0$ if and only if $\vec{x_1} = \vec{x_2}$,
- **Symmetry**: $d(\vec{x_1}, \vec{x_2}) = d(\vec{x_2}, \vec{x_1})$, and
- **Triangular Inequality**: $d(\vec{x_1}, \vec{x_3}) \leq d(\vec{x_1}, \vec{x_2}) + d(\vec{x_2}, \vec{x_3})$.

⌟

Typically, a natural choice of distance metric is the **Euclidean distance**, that is:

$$d_e(\vec{x_1}, \vec{x_2}) := \|\vec{x_1} - \vec{x_2}\| = \sqrt{(x_{11} - x_{12})^2 + (x_{21} - x_{22})^2 + \cdots + (x_{k1} - x_{k2})^2}.$$

Remark VIII.1.2. Conventions in Euclidean Distance.

There are some conventions in **Euclidean distance** to simply the computations:

- Sometimes, we use the squared distance $\|\vec{x_1} - \vec{x_2}\|^2$ as a **monotone transformation** for nonnegative quantities. It does not change the order but simplifies calculations.
- Also, we need to **normalize** all dimensions so variations is on the same scale for each dimension, otherwise, one or a few dimensions will dominate the distance, which is not desiring.

⌟

To solve **clustering** problem, suppose that we have a set of points $\vec{x_1}, \cdots, \vec{x_n}$ forming the data set $[D]$ and a distance metric d. We want to have k clusters.

The intuitive idea is to represent each cluster by some **exemplar elements**, called **centroids**, then we may assign each point to the cluster based on which **exemplar point** it is closest to.

Setup VIII.1.3. Forming Exemplar Points.

We want to have exemplars as $\mu_1, \cdots, \mu_k$, and for each point $\vec{x_i}$ and each cluster j, in which $j = 1, \cdots, k$, we define an indicator parameter r_{ij}, for whether the point is clustered into the jth cluster.

As long as we know all the exemplars, we just assign the point to the closest one:

$$r_{ij} = \begin{cases} 1, & \text{if } j = \operatorname{argmin}_j \|\vec{x}_i - \mu_j\|^2, \\ 0, & \text{otherwise.} \end{cases}$$

Then, as we know which cluster each point is in, we can pick μ_j to be the mean of the points in jth cluster:

$$\mu_j = \frac{\sum_{i=1}^n r_{ij}\vec{x}_i}{\sum_{i=1}^n r_{ij}},$$

which is the average of all the points on the Euclidean space. ⌟

The setup suggests a **natural iterative procedure**, which is guessing and keep iterating.

Algorithm VIII.1.4. The K-Means Algorithm.

Following the setup, we initialize at $t = 0$ by having $\mu_{10}, \cdots, \mu_{k0}$ to be k distinct random points in $[D]$. Then, we iteratively apply that:

- For every $\vec{x}_i$, we set:
$$r_{ij} = \begin{cases} 1, & \text{if } j = \operatorname{argmin}_j \|\vec{x}_i - \mu_j\|^2, \\ 0, & \text{otherwise.} \end{cases}$$
- Then, set $t = t + 1$, and for every cluster j, we calculate:
$$\mu_{jt} = \frac{\sum_{i=1}^n r_{ij}\vec{x}_i}{\sum_{i=1}^n r_{ij}},$$

The **K-means algorithm** learns μ_j and r_{ij} to minimize the loss:

$$\sum_{i=1}^n \sum_{j=1}^k r_{ij}\|\vec{x}_i - \mu_j\|^2.$$ ⌟

Remark VIII.1.5. Notes on K-Means Algorithm.

The **k-means algorithm** has the following properties:

- The **k-means algorithm** has each parameter update minimizes the loss, so the algorithm will eventually reach a minimum.
- Since the problem is **not convex** in general, so it might not be a global minimum.
  In fact, it is **NP-hard** to find the global minimum for Euclidean distances.
- The starting values matter for the clustering.
  In practice, the algorithm is fast, so many starting values may be tried, and the **best clusters** were kept.

⌟

The clustering **do not have to use geometric distance**, so the distance can be a specific notion of **similarity**. An application of clustering is to cluster similar pixels of an image together (using RGB values rather than location).

Algorithm VIII.1.6. KNN as Distance-Based Prediction.

Given the notion of distance between points, a simple algorithm for learning functions given labeled data $[D]$ with features $\vec{X}$ and outcome Y can be simply storing $[D]$.
Given a new point $\vec{x}_i$, we can compute the k nearest closest points $\vec{x_1}, \vec{x_2}, \cdots, \vec{x_k}$ in $[D]$ with known labels, and take a vote among them to decide on the outcome of $\vec{x}_i$ in terms of classification or take some sort of average for regression. The method is known as ***k*-nearest neighbor** (KNN). ⌟

The **KNN** algorithm is effective is some problem, but it has some issues:

- It does not work well with high dimensional $\vec{X}$.
- Finding the closest few points to $\vec{x}_i$ may be computationally intense given large datasets $[D]$.
- It is sensitive to the precise choice of **distance**.

VIII.2 EM Algorithm

The **K-means** algorithm is based on minimizing a distance-based loss, but it has no probabilities involved, and in some cases, **an explicit probabilistic model** for data is more reasonable.

Setup VIII.2.1. Mixture Model for Learning Clusters.

Consider still with k clusters, we may describe a cluster probabilistically as points varying around an exemplar point according to some **probability distribution**, which leads to **Gaussian mixture model**. The **mixture model** has general form:

$$\mathbb{P}(\vec{x}) = \sum_c \mathbb{P}(\vec{x} \mid c)\mathbb{P}(c),$$

where c is the unobserved ones. Here, $\mathbb{P}(c)$ is of some **unobserved variable *C***, which is typically categorical. Thus, we can use C to represent **cluster membership**, and a common choice for $\mathbb{P}(\vec{x} \mid c)$ is a **conditional multivariate Gaussian distribution**, modeled as:

$$\mathbb{P}(\vec{x} \mid c) = \frac{1}{(1\pi)^{k/2}|\Sigma_c|^{1/2}} \exp\left[-\frac{1}{2}(\vec{x} - \vec{\mu}_c)^\intercal \Sigma_c^{-1}(\vec{x} - \vec{\mu}_c)\right].$$

Moreover, we assign discrete probability distribution $\mathbb{P}(c)$ with parameters κ_i for $i = 1, \cdots, k-1$ to govern the **probability of belonging to a particular cluster**. ⌟

The **Mixture model** is different from k-means, as now each point is not assigned to a cluster but rather has a probability of being in a cluster (*and probability should sum to 1*).

Consequently, this gives use an observed data likelihood that we can maximize to solve for the parameters for the model:

$$\mathcal{L}_{[}X]\big(\{\kappa_i, \vec{\mu}_i, \Sigma_i : i = 1, \cdots, k\}\big) = \prod_{i=1}^{n} \sum_c \mathbb{P}(\vec{x}_i \mid c)\mathbb{P}(c).$$

With this likelihood, we can use **gradient descent** to maximize it, but there happened to be a better algorithm.

Remark VIII.2.2. Gaussian Mixture Model as Graphical Model.

Fundamentally, the **Gaussian mixture model** is a graphical model, represented as a DAG with hidden variable:

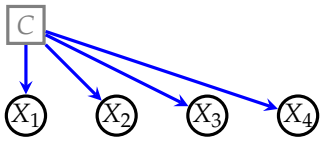


Figure VIII.1. Graphical model for Gaussian mixture model.

In particular, the gray square vertex is a hidden category cluster variables, and the circles are **observed data features**. ⌟

Algorithm VIII.2.3. The EM Algorithm.

Here, recall that our given likelihood is:

$$\mathcal{L}_{[}X]\big(\{\kappa_i, \vec{\mu}_i, \Sigma_i : i = 1, \cdots, k\}\big) = \prod_{i=1}^{n} \sum_{c} \mathbb{P}(\vec{x}_i \mid c)\mathbb{P}(c).$$

If we could observe C, then we have the full data likelihood, that is:

$$\mathcal{L}_{[}X, C]\big(\{\kappa_i, \vec{\mu}_i, \Sigma_i : i = 1, \cdots, k\}\big) = \prod_{i=1}^{n} \mathbb{P}(\vec{x}_i \mid c_i)\mathbb{P}(c_i).$$

Here, we may treat the full data likelihood as fixed in $[X]$ but random in C, and take on the expectations given C, that is:

$$\mathbb{E}\big[\mathcal{L}_{[X,C]}(\vec{\beta}) \mid [X]\big] \text{ w.r.t the distribution } \mathbb{P}(c \mid [X]).$$

Then for parameter $\vec{\beta}^{(t)}$, we can maximizing the expectation recursively, by:

$$\vec{\beta}^{(t+1)} = \operatorname{argmax}_{\vec{\beta}} \mathbb{E}\big[\mathcal{L}_{[X,C]}(\vec{\beta}) \mid [X]; \vec{\beta}^{(t)}\big].$$

As usual, we use the log likelihood, which is fundamentally the same story:

$$\vec{\beta}^{(t+1)} = \operatorname{argmax}_{\vec{\beta}} \mathbb{E}\big[\log \mathcal{L}_{[X,C]}(\vec{\beta}) \mid [X]; \vec{\beta}^{(t)}\big].$$

Then, note that the probability is:

$$\mathbb{P}(C = j \mid \vec{x}_i; \vec{\beta}^{(t)}) = \frac{\mathbb{P}(\vec{x}_i \mid C = j)\mathbb{P}(C = j)}{\sum_j \mathbb{P}(\vec{x}_i \mid C = j)\mathbb{P}(C = j)} = \frac{\kappa_j^{(t)} \mathcal{N}(\vec{x}_i; \vec{\mu}_j^{(t)}, \Sigma_j^{(t)})}{\sum_{\ell=1}^{k} \kappa_\ell^{(t)} \mathcal{N}(\vec{x}_i; \vec{\mu}_\ell^{(t)}, \Sigma_\ell^{(t)})}.$$

Hence, we can show that the parameters that maximize the expectation of log likelihood can be estimated as $1/n$:

$$\kappa_j^{(t+1)} = \mathbb{P}(C = j) = \frac{1}{n} \sum_{i=1}^{n} \mathbb{P}(C = J \mid \vec{x}_i; \vec{\beta}^{(t)}),$$

$$\vec{\mu}_j^{(t+1)} = \mathbb{E}[\vec{x} \mid C = j] = \frac{\sum_{i=1}^{n} \mathbb{P}(C = j \mid \vec{x}_i; \vec{\beta}^{(t)}) \vec{x}_i}{\sum_{i=1}^{n} \mathbb{P}(C = j \mid \vec{x}_i; \vec{\beta}^{(t)})},$$

$$\Sigma_j^{(t-1)} = \operatorname{Cov}[\vec{x} \mid C = j] = \frac{\sum_{i=1}^{n} \mathbb{P}(C = j \mid \vec{x}_i; \vec{\beta}^{(t)})(\vec{x}_i - \vec{\mu}_j^{(t+1)})(\vec{x}_i - \vec{\mu}_j^{(t+1)})^\intercal}{\sum_{i=1}^{n} \mathbb{P}(C = j \mid \vec{x}_i; \vec{\beta}^{(t)})}.$$

Note that these updates are in the closed form, so the algorithm can run **iteratively**. ⌟

Note that maximizing the **log likelihood** is the equivalent of maximizing **likelihood**, and as we think about each row separately, one can proof that:

$$\log \mathbb{P}(\vec{x}_i; \vec{\beta}) - \log \mathbb{P}(\vec{x}_i; \vec{\beta}^{(t)}) \geq Q^i(\vec{\beta}; \vec{\beta}^{(t)}) - Q^i(\vec{\beta}^{(t)}; \vec{\beta}^{(t)}),$$

where Q^i is the ith row part of what an EM step maximizes, hence each step for each row is an improvement.

Remark VIII.2.4. Notes about EM Algorithm as a Family.

The **EM algorithms** is a family of algorithms, so we have:

- Since the previous argument is independent of **mixtures of Guassian**, hence it works generically for observed data and missing data.
- The EM is a generic family of algorithms for fitting likelihoods in **hidden variable models** or **missing data**.
- The advantage of EM is that:
  - No need to worry about choosing step size, it is iterative automatically.
  - Optimizing $\mathbb{E}\left[\log \mathcal{L}_{[X,C]}(\vec{\beta}) \mid [X]; \vec{\beta}^{(t)}\right]$ is easier than $\log \mathcal{L}_{[X]}(\vec{\beta})$, and
  - It applies to a wide range of problems.
- Some disadvantages are that:
  - It has no control over step size, so the overall algorithm may be slow.
  - There could be blow ups by the singularities.

⌟

Recall that in the algorithm, we are updating $\vec{\mu}_j$ and Σ_j with divisions, in case the denominator is zero or very small, it is possible that the algorithm fails to converge, and the results might not be smooth everywhere.

VIII.3 Missing Data

Missing data values are fact of life, there are dropout in studies, survey sampling, issues with data collection, censoring by death, coarsened values, scarce outcome labeling, etc.

For **model missing values**, we can:

- **Directly** fit the full data likelihood (by EM, optimization, etc.), and can be better identified from observed data.
- **Imputation methods**, but not always clear about the model and what we are imputing.

For model **missingness status**, we want to:

- Inverse weighting/propensity score methods.
- Use **semi-parametric** methods, which need additional modeling.

Here, we think of the missing data **counterfactually**.

Definition VIII.3.1. Missing Variable.

We define a **missing variable** by $X_i^{(1)}$, which is the variable X_i if we could hypothetically see it. Every $X_i^{(1)}$ has an **indicator** R_i and a **factual proxy** variable X_i. In particular:

$$X_i = \begin{cases} X_i^{(1)}, & \text{if } R_i = 1, \\ \text{(Undefined)}, & \text{if } R_i = 0. \end{cases}$$

Thus, the missing variable is a counterfactual with R_i as treatment. ⌟

Now the problem turns to how R_i and $X_i^{(1)}$ are related, and we can use graphs to model their relationship.

The procedures are:

$$\begin{array}{ccccc} \theta & & & & \\ \uparrow & & & & \\ \mathbb{P}(\vec{X}^{(1)}) & \xrightarrow[\text{(sampling)}]{\sim} & \left.\begin{matrix} x_{11}^{(1)} & x_{21}^{(1)} & x_{31}^{(1)} \\ x_{12}^{(1)} & x_{22}^{(1)} & x_{32}^{(1)} \\ \vdots & \vdots & \vdots \end{matrix}\right\} & \xrightarrow[\text{problem}]{\text{full data}} & \hat{\theta} \\ \downarrow cor & & & & \\ \mathbb{P}(\vec{R}, \vec{X}) & \xrightarrow{\sim} & \left.\begin{matrix} 0 & 0 & 1 & ? & ? & x_{31} \\ 1 & 1 & 0 & x_{12} & x_{22} & ? \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{matrix}\right\} & \xrightarrow[\text{problem}]{\text{missing data}} & \hat{\vartheta} \end{array}$$

There, Donald Rubin developed on the causal inference with missing data.

Definition VIII.3.2. Modern Missingness Hierarchy.

There are different **level of missingness of the data**:

- **Missing Completely at Random** (MCAR): The presence of ? can be determined by an independent coin flip.
- **Missing at Random** (MAR): The presence of ? can be determined by coin flip independent of underlying variable given observed data.
- **Missing not at Random** (MNAR): Neither of above applies. ⌟

The MCAR case is easy to deal with, while most missing data work assumes MAR, although it is *unrealistic*.

Here, we do want to consider the missing case by case:

- For **Missing Completely at Random**, we have the events that lead to missingness occur independently of observed and unobserved data, so we can translate it into:
$$X_1^{(1)} \amalg R_1,$$
which is a collider relationship, so graphically it is:

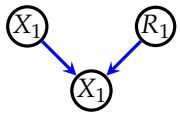


Figure VIII.2. Collider model for MCAR.

Therefore, we have that:
$$\mathbb{P}(X_1^{(1)}) = \mathbb{P}(X_1^{(1)} \mid R_1 = 1) = \mathbb{P}(X_1 \mid R_1 = 1).$$
Under MCAR, we can use observed case analysis, *i.e.*, we can do the analysis on fully observed rows only.

- For **Missing at Random**, the event that lead to missingness occur independently of unobserved data given observed data, so it translates into:
$$X_2^{(1)} \amalg R_2 \mid X_1,$$
that is we have X_1 fully observed, so this is a split triplet.

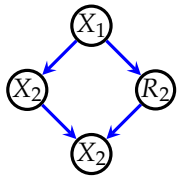


Figure VIII.3. Split model for MAR.

Therefore, we have that:
$$\mathbb{P}(X_1, X_2^{(1)}) = \mathbb{P}(X_2^{(1)} \mid X_1)\mathbb{P}(X_1) = \mathbb{P}(X_2^{(1)} \mid X_1, R_2 = 1)\mathbb{P}(X_1) = \mathbb{P}(X_2 \mid R_2 = 1, X_1)\mathbb{P}(X_1).$$
Then it is no longer sufficient to just look at observed rows, so we use two estimators for $\mathbb{E}[X_2^{(1)}]$, that is:

  - **Likelihood based inference**: $\mathbb{E}[X_2^{(1)}] = \frac{1}{N}\sum_i \mathbb{E}[X_2 \mid R_2 = 1, X_1^i]$, or
  - **Propensity based inference** by Horvitz-Thompson: $\mathbb{E}[X_2^{(1)}] = \frac{1}{N}\sum_i \frac{\mathbb{1}(R_2 = 1)}{\mathbb{P}(R_2 = 1 \mid X_1^i)} X_2^i$.

Without MAR, $\mathbb{P}(\vec{X}^{(1)})$ is not always a function of $\mathbb{P}(\vec{X}, \vec{R})$, and a simple example could be variable causes its own missingness status:

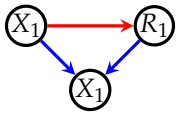


Figure VIII.4. Cycle, variable causes its own missingness.

Moreover, there will be no unique map from observed to full data even for binary $X^{(1)}$ since the full law has a higher dimension of the observed law. Thus, there does not exist a **two-sided inverse**.

Remark VIII.3.3. Notes on Non-Identification.

In typical **machine learning** problems, quantities of interest were all function of the observed data distribution, but it is not necessarily true, especially in **missing data** problems.
If the object we are estimating is not a function of the observed data distribution, given the model, we could:

- Shrink the model (since model is a set), *i.e.*, make more assumptions.
- Enlarge the parameter (give the parameters with **bounds** on it).
  In particular, some parameters such as **worst case risk** is bounded for the computation complexity, but here, computing a parameter is **impossible**.
- Change the parameter you want to estimate something easier.
- Give up and seek for better data. ⌟

Meanwhile, there ate various strategies to deal with missing data:

(i) **Inverse weighting**:

- We check that under the missing data model, $\mathbb{P}(R = 1 \mid \vec{X}^{(1)})$ is identified as $f\big(\mathbb{P}(\vec{R}, \vec{X})\big)$.
- Then, we can fit $\hat{f}\big(\mathbb{P}(\vec{R}, \vec{X})\big)$ by **observed data**.
- Thus, we create a new dataset representing a **pseudo-population** by weighting each row of the original data by $1/\hat{f}\big(\mathbb{P}(\vec{R}, \vec{X})\big)$.
- Alternatively, we can modify the score equation by the weighting, *i.e.*, replace $\mathbb{E}\big[g(\vec{X})\big(Y - \mathbb{E}[Y \mid X]\big)\big]$ for regression by $\mathbb{E}\big[\mathbb{1}(R = 1)\mathbb{P}(R \mid \vec{X})g(\vec{X})\big(Y - \mathbb{E}[Y \mid \vec{X}]\big)\big]$ if Y is MAR given $\vec{X}$.

(ii) **Imputation**:

- For every row with observed data $\vec{X}_{\vec{r}}$ (governed by $\vec{R} = \vec{r}$), we check that under the missing data model that $\mathbb{P}(\vec{X}^{(1)} \setminus \vec{X}^{(1)}_{\vec{r}} \mid \vec{X}_{\vec{r}})$ is identified as $f_{\vec{r}}\big(\mathbb{P}(\vec{X}, \vec{R})\big)$.
- Then, we fir $\hat{f}_{\vec{r}}\big(\mathbb{P}(\vec{R}, \vec{X})\big)$ using observed data for every patter $\vec{r}$.
- Then, we create a new dataset representing a pseudo-population by imputing all missing data given a pattern $\vec{r}$ using $\hat{f}_{\vec{r}}\big(\mathbb{P}(\vec{R}, \vec{X})\big)$, that is sample all missing values.
- Eventually, we do the problem in the **new dataset** and **repeat** as many times as desired.

(iii) **Maximum likelihood inference**:

- Here, we check that $\mathbb{P}(\vec{R}, \vec{X}^{(1)})$ is identified as $f\big(\mathbb{P}(\vec{R}, \vec{X})\big)$.
- Then, we fit $\hat{f}\big(p(\vec{R}, \vec{X})\big)$ using EM or GD algorithms.
- Then do the problem given parameters of the full data law.

Definition VIII.3.4. Missing Data Graphs.

We can construct the missing data as graphs. These graphs follow that:

- Indicators cannot cause underlying counterfactuals.
- Each observed proxy only has two parents.
- There are no **cycles**.
- Use a **single graph**, with both observed and counterfactural variables on the same graph.
- In missing data only one counterfactual world matters, where we observe everything of $\vec{R} = 1$.

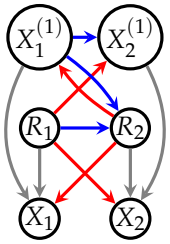


Figure VIII.5. Valid (blue), and invalid (red) of missing data graph.

Proposition VIII.3.5. Laws of Data Models of DAG.

There are **laws** associated with data models of a DAG:

- **Target law**: $\mathbb{P}(\vec{X}^{(1)}, \vec{O})$ over:
  - Potentially missing random variables $\{\vec{X_1}^{(1)}, \cdots, \vec{X_k}^{(1)}\}$, and
  - Observed random variables $\{O_1, \cdots, O_m\}$.
- **Nuisance law**: $\mathbb{P}(\vec{R} \mid \vec{X}^{(1)}, \vec{O})$ over:
  - Missingness indicators $\vec{R} \equiv \{R_1, \cdots, R_k\}$.
- **Coarsening law** (deterministic): $\mathbb{P}(\vec{X} \mid \vec{X}^{(1)}, \vec{R})$, with:
  - $X_i \equiv X_i^{(1)}$ if $R_i = 1$ and $X_i \equiv ?$ if $R_i = 0$.

Hence, by the **chain rule of probability**, we have:

$$\mathbb{P}(\vec{X}^{(1)}, \vec{O}) = \frac{\mathbb{P}(\vec{X}^{(1)}, \vec{O}, \vec{R} = 1)}{\mathbb{P}(\vec{R} = 1 \mid \vec{X}^{(1)}, \vec{O})} = \frac{\mathbb{P}(\vec{X}, \vec{O}, \vec{R} = 1)}{\mathbb{P}(\vec{R} = 1 \mid \vec{X}^{(1)}, \vec{O})}.$$

Therefore, $\mathbb{P}(\vec{X}^{(1)}, \vec{O})$ is identifiable if and only if $\mathbb{P}(\vec{R} = 1 \mid \vec{X}^{(1)}, \vec{O})$ is identifiable.

Theorem VIII.3.6. Bhattacharya, Nabi, and S.

Any missing data DAG $\mathcal{G}$ on $\vec{R}$, $\vec{X}^{(1)}$, and $\vec{O}$ without:

- Self-censoring edges ($X_i^{(1)} \to R_i$), or

- Colliders ($X_i^{(1)} \to R_j \leftarrow R_i$)

is identified. Any $\mathcal{G}$ with at least one of those structures is not. ⌟

VIII.4 Maximum Likelihood Inference

The **maximum likelihood inference** can be applied to a DAG model when **identified**.

Proposition VIII.4.1. Identified DGA Model.

If a DAG model is identified, then:

- Both $\mathbb{P}(\vec{X}^{(1)})$ and $\mathbb{P}(\vec{R} \mid \vec{X}^{(1)})$ are **functionals** of $\mathbb{P}(\vec{R}, \vec{X})$.
- $\mathbb{P}(\vec{X}^{(1)}, \vec{R})$ factorizes w.r.t a DAG, and
- Each factor (a **conditional** in the full law) is an **algebraic functional** of $\mathbb{P}(\vec{R}, \vec{X})$.

Note that the functional is not a conditional in the observed law in general, and is typically more complicated. Therefore, under mild assumption of having **smooth parameter map**, we can maximize the observed likelihood directly as:

$$\mathcal{L}_n(\vec{R}, \vec{X}, \vec{O}; \eta) \equiv \prod_{i=1}^{n} \sum_{x_{ji}^{(1)} : r_{ji}=0} \prod_{v_k \in \vec{x}, \vec{r}, \vec{x}^{(1)}} \mathbb{P}(v_{ki} \mid \{v_{mi} \in \mathrm{pa}_{\mathcal{G}}(v_k)\}).$$

⌟

Recall when we maximize the likelihood using EM algorithm, say we have counts $\#\big(v_i, \mathrm{pa}_{\mathcal{G}}(v_i)\big)$ of all value combinations of v_i, $\mathrm{pa}_{\mathcal{G}}(v_i)$ for every v_i, including those with missing values at particular rows. IN this case, we can obtain the MLE in closed form, via:

$$\frac{\#\big(v_i, \mathrm{pa}_{\mathcal{G}}(v_i)\big)}{\#\,\mathrm{pa}_{\mathcal{G}}(v_i)}.$$

If we have missing data, we don't have access to count of some v_i and $\mathrm{pa}_{\mathcal{G}}(v_i)$. However, we can calculate the **expected counts** given the observed data.

VIII.5 Structure Learning

Setup VIII.5.1. Structural Learning Algorithm.

Suppose that there is a graph $\mathcal{G}(\vec{V})$ with k vertices and a distribution $\mathbb{P}(\vec{V})$ factorizing relative to $\mathcal{G}$.

- The input is a data set $[X]$, assuming that it is sampled **independently** from $\mathbb{P}(\vec{V})$.
- The output is a set of graphs consistent with what we know about $[X]$, and hopefully it will include $\mathcal{G}$.

The **structural learning** is an **unsupervised learning** problem, and we want to find a causal description of the data. ⌟

Recall that **Global Markov property** is one way. This works when we known the DAG and want to know what is implied by the data, but now we want to know that the data implies about the DAG.

Definition VIII.5.2. Faithfulness.

For any $\vec{A}$, $\vec{B}$, and $\vec{C}$ in a **faithful** DAG, then $\vec{A}$ is d-separated from $\vec{B}$ given $\vec{C}$ in $\mathcal{G}$ if and only if $\vec{A} \amalg \vec{B} \mid \vec{C}$ in $\mathbb{P}(\vec{V})$. ⌟

The assumption for **faithfulness** is somewhat controversial. It can be thought of as **an additional property** that $\mathbb{P}(\vec{V})$ factorization according to $\mathcal{G}$ may satisfy.

Proposition VIII.5.3. Most Distributions will be Faithful.

Most $\mathbb{P}(\vec{V})$ will be faithful, and the unfaithful distribution form a set of measure zero. ⌟

However, there are various caveats:

- If there are finite samples, we many not be able to tell unfaithful from nearly unfaithful $\mathbb{P}(\vec{V})$, and there are many more of those.
- Nature does not pick distributions at random, there may evolve unfaithful situations for evolutionary reasons, such as **homeostasis**.

Hence, in practice, we should justify why **faithfulness** is **sensible** for the problem.

Recall by **Definition VII.4.9**, some graphs are observational equivalence, *i.e.*, the **local Markov property** gives the same independence. Hence, the structure learning can only learn graphs **up to equivalent classes** (quotient of equivalent classes $\{\mathcal{G}\}/\sim$).

Theorem VIII.5.4. Verma and Pearl.

Two distinct DAGs $\mathcal{G}_1$ and $\mathcal{G}_2$ represent the same statistical DAG model if and only if they **share skeletons** and **unshielded colliders** (that is $A \to C \leftarrow B$ where A and B are not adjacent). ⌟

Example VIII.5.5. Equivalent Class of DAG.

Consider the following graphs:

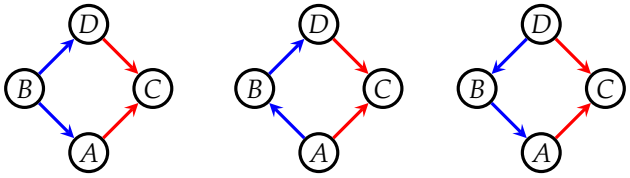


Figure VIII.6. Equivalent classes of DAGs.

All three DAGs gives that the model that $B \amalg C \mid A, D$ and $A \amalg D \mid B$, and by **Theorem VIII.5.4 Verma and Pearl**, they share the same skeletons and the unshielded colliders. ⌟

Therefore, we can think of different types of **structural learning**.

Setup VIII.5.6. Constraint Based Learning.

The constraint based learning finds the constraints that hold in $[X]$. Then, we rule out graphs that are inconsistent with the constraints we found, and we return what is left. ⌟

VIII.6 Score Based Learning

Another method to learn a **graph** is through the score based learning, which is to assign a **quantitative likelihood** to the function.

Setup VIII.6.1. Score Based Learning.

- We assign a **score** to any graphical model.
- **Scores** typically reward fit.
  At the same time, we also **regularize**, since we want a concise description of the data.
- Eventually, we do search (**model selection**) for high scoring models given $[X]$. ⌟

The **score based learning** is asymptotically equivalent to **constraint based learning**, but over **finite iterations**, they behave differently.

For **parametric** structure learning, we:

- Make strong additional assumptions on $\mathbb{P}(\vec{V})$, and
- Orient edges using the assumptions.

Some examples are additive noise models or having structural learning as classification.

Example VIII.6.2. Intuition of Score Based Learning.

Consider the following DAGs:

Figure VIII.7. Two DAGs, true (left) and proposed (right).

By assuming **faithfulness**, we have in the true graph that:

$$A \amalg C \text{ and } A \not\amalg C \mid B,$$

whereas for the proposed graph, we have:

$$A \not\amalg C \text{ and } A \amalg C \mid B,$$

Therefore, we may conclude that the proposed graph does not fit the data well. ⌟

Hence, we want to assess the data fit using the **log likelihood for DAG** models.

Recall that the DAG factorization is:

$$\mathbb{P}(\vec{V}) = \prod_{V_i \in \vec{V}} \mathbb{P}\big(V_i \mid \mathrm{pa}_{\mathcal{G}}(V_i)\big).$$

Setup VIII.6.3. Decomposable Likelihood.

We assume that the models were on $\mathbb{P}(V_i \mid \mathrm{pa}_{\mathcal{G}}(V_i); \alpha_i)$ for each $V_i \in \vec{V}$, then for a dataset $[D]_{n\times k}$ and $\vec{\alpha} = \{\alpha_i \mid V_i \in \vec{V}\}$, the **log likelihood** is:

$$\log \mathcal{L}_{[D]}(\vec{\alpha}) = \sum_{j=1}^{n}\sum_{i=1}^{k} \log \mathbb{P}\big(V_i^j \mid \mathrm{pa}_{\mathcal{G}}^j(V_i); \alpha_i\big).$$

Then we can fit each α_i separately on part of data that involves $\mathbb{P}\big(V_i \mid \mathrm{pa}_{\mathcal{G}}(V_i)\big)$. ⌟

Example VIII.6.4. Decomposable Likelihood for Score Based Learning.

Referring to the situation in *Figure VIII.7*, the left is the true graph and the right is the proposed, their likelihoods are respectively:

$$\mathcal{L}^l_{[D]}(\alpha) = \sum_{j=1}^{n}\Big[\log \mathbb{P}(A^j; \beta_A) + \log \mathbb{P}(B^j \mid A^j, C^j; \beta_B) + \log \mathbb{P}(C^j; \beta_C)\Big],$$

$$\mathcal{L}^r_{[D]}(\alpha) = \sum_{j=1}^{n}\Big[\log \mathbb{P}(A^j \mid B^j; \beta_A) + \log \mathbb{P}(B^j \mid C^j; \beta_B) + \log \mathbb{P}(C^j; \beta_C)\Big].$$

Here, we pick $\vec{\alpha}$ and $\vec{\beta}$ to maximize **log likelihood**, and we may compare the result. The **true graph** will have a higher likelihood value. ⌟

VIII.7 Bayesian Information Criterion Score

However, there are issues with only using the **likelihood**, there could be **overfitting** issues.

Example VIII.7.1. Counterexample of Overfitting in Score Based Learning.

Alternatively, we consider the following DAGs:

Figure VIII.8. Two DAGs, true (left) and proposed (right).

Here, their respective likelihoods are:

$$\mathcal{L}^l_{[D]}(\alpha) = \sum_{j=1}^{n}\Big[\log \mathbb{P}(A^j; \beta_A) + \log \mathbb{P}(B^j \mid A^j, C^j; \beta_B) + \log \mathbb{P}(C^j; \beta_C)\Big],$$

$$\mathcal{L}^r_{[D]}(\alpha) = \sum_{j=1}^{n}\Big[\log \mathbb{P}(A^j \mid B^j, C^j; \beta_A) + \log \mathbb{P}(B^j \mid C^j; \beta_B) + \log \mathbb{P}(C^j; \beta_C)\Big].$$

Here, the right model will fit the data better.
Although the left model is the true one, it is $\{\mathbb{P}(A, B, C) \mid A \amalg C\}$, but the right model is $\{\mathbb{P}(A, B, C)\}$

without constraints. The left model is a **submodel** of the right so it will be strictly more restrictive and fit the data less. ⌟

If simply using the likelihood to score the models, the highest scoring model is always the complete graph, which is similar to **overfitting** in **machine learning**. Suppose the true graph is **sparse**, the complete graph would have **too many parameters**. The fit would be well but it is **not** capturing the true structure of the problem. Hence, we want to **punish too many parameters**.

Definition VIII.7.2. Bayesian Information Criterion Score.

The **Bayesian Information Criterion** (BIC) score is a number assigned to a DAG and parameterization $\vec{\alpha} = \{\alpha_i \mid V_i \in \vec{V}\}$ for all Markov factors:

$$\text{BIC} = -2 \log \mathcal{L}_{[D]}(\vec{\alpha}) + m \log n,$$

where n is the number of data points in $[D]$ and m is the number of parameters. ⌟

Since we have the minus sign for the likelihood, so **low BIC** is good, *i.e.*, we prefer models with low scores. This is similar to a **regularization**, but over graph structures.

Also, we are using **BIC** to select models since as $n \to \infty$, **BIC** does sensible things.

Proposition VIII.7.3. Properties of BIC.

Suppose $\mathcal{G}^*$ is the true DAG, BIC has the following properties:

(i) If $\mathcal{G}_1$ is an edge supergraph of $\mathcal{G}^*$ and $\mathcal{G}_2$ is not. Then $\mathcal{G}^*$ is a submodel of $\mathcal{G}_1$ model but not submodel of $\mathcal{G}_\in$, then **BIC** gives a better score to $\mathcal{G}_1$ than $\mathcal{G}_2$.

(ii) If $\mathcal{G}_1$ and $\mathcal{G}_2$ are both edge supergraph of $\mathcal{G}^*$, but $\mathcal{G}_1$ has fewer parameters than $\mathcal{G}_2$, then **BIC** gives a better score to $\mathcal{G}_1$ than $\mathcal{G}_2$. ⌟

With the above properties, if we choose the correct parameter families for $\mathbb{P}(V_i \mid \text{pa}_\mathcal{G}(V_i); \alpha_i)$, then **BIC** will find the class of true graph (up to **equivalence**) eventually.

Note that using **BIC** requires computation of the following:

- List all DAGs on k vertices.
- Compute BIC score for each DAG.
- Pick the best scoring DAGs.

Although it guarantees accuracy, it is on the assumption of unbounded computing resources. The number of DAGs are $\mathcal{O}(2^k)$, and the problem becomes **intractable** for larger DAGs.

In general, the problem is **NP-hard**, so we need to make further assumptions.

VIII.8 Local Search of DAG Structure

Of course, we may assume that the true DAG is **sparse**, but the space of all sparse DAGs is still large, so we need to think of ways to **search locally**.

Remark VIII.8.1. Intuitions to Local Search.

The idea of **local search** is just like making **game-playing programs**. Here we have a big undirected graph (as **search space**), the vertices are states (**game positions**), and the edges are neighboring states (**game position reachable** from current one by one move).
Here, each state has a value, and in order to explore the space, we look for a **good state**. ⌟

Just liking playing chess, the computer is performing better since:

- Uses discrete search and clever ways of assigning values to positions,
- Has big opening database and endgame database, and
- Has powerful computers to search quickly.

Setup VIII.8.2. Local Search in DAG.

For us, the search space states are all DAGs.

- We define **local moves** to be able to reach any DAG from any other DAG.
- We want a clever algorithm to find the best state (DAG) using a **(polynomial) sequence** of moves.

A key for local move is to **not** move to a graph that is **observational equivalent**, rather we want the move to be in the **quotient space**. ⌟

Algorithm VIII.8.3. Greedy Equivalence Search Algorithm.

(i) We start with a graph with no edges.

(ii) Then we pick the **best** new class after **adding a single edge** to current class.
Note that the addition of the edge should move the class to be a different class.

(iii) Repeat the previous step until there are no score improvement.

(iv) Then, pick the **best** new class after **removing a single edge** to the current class.

(v) Repeat the previous step until there are no score improvement.

Then the class we have is the result to return. ⌟

Theorem VIII.8.4. GES Completeness, Chickering.

Under **faithfulness**, as $n \to \infty$, GES cirrectly identifies the **true equivalence class**. ⌟

Hence, as we have indefinite computing resources, GES is proven to be correct.

Remark VIII.8.5. Some Issues with Local Search Algorithm.

Still **BIC** and **GES** are large sample results, they exhibits that:

- With finite data, all bets are all.
- Also, the local search is still intractable if the score cannot be evaluated in **polynomial time**.
- If we get the models $\mathbb{P}(V_i \mid \mathrm{pa}_{\mathcal{G}}(V_i); \alpha_i)$ wrong, all the bests are off.
- The model cannot be used when $p \gg n$, that is there are way more variables than the number of samples. (*In particular, there are many $p \gg n$ problems for genomics and social networks.*)
- There needs to be **sparsity ideas** to make progress, and it is also assuming that there is no **hidden variables**.

⌟

IX Dimension Reduction

Previously, we discussed about the **curse of dimensionality** which leads to **problems for prediction and estimation methods**.

In previous attempts, we have tried the following to reduce dimensions:

- **Regularization**: assume that **simpler models** can do well even in high dimensional data, and
- **Model selection**: select a parsimonious hypothesis class.

Another approach is to explicitly transform high dimensional data into a lower dimension representation that preserves features **salient** for analysis.

IX.1 Principal Component Analysis

When transforming data, we want to use an **affine** transformation so it is zero centered (this is *trivially* by moving the ellipse of points so its is centered at origin). For ellipses, we can ignore the shorter of ellipse axes and compress the data with a lower dimension.

Here, we may want to use a matrix to handle the data, such as **stretching**, **rotating**, or **reflecting** a vector in the Euclidean space.

Definition IX.1.1. Eigenvalues and Eigenvectors.

Suppose $[A]$ is a linear map (or k-by-k *matrix* in this context), for some nonzero vector $\vec{v}$, it stretches or compresses the vector, *i.e.*:

$$[A].\vec{v} = \lambda\vec{v},$$

and here $\vec{v}$ is an **eigenvector** of $[A]$ and λ is its corresponding **eigenvalue**. ⌟

Theorem IX.1.2. Diagonalization.

For a k-by-k matrix $[A]$, it has at most k eigenvalues, and a classical decomposition is **diagonalization**, or **eigendecomposition**, where we have:

$$[A] = [Q][\Lambda][Q]^{-1},$$

where $[Q]$ as the columns as eigenvectors and $[\Lambda]$ is a diagonal matrix with eigenvalues, that is:

$$[A] = \begin{pmatrix} \vec{v_1} & \vec{v_2} & \cdots & \vec{v_k} \end{pmatrix} \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_k \end{pmatrix} \begin{pmatrix} \vec{v_1} & \vec{v_2} & \cdots & \vec{v_k} \end{pmatrix}^{-1}.$$

⌟

Here, we may consider $[A]$ as a function that moves vectors, and the eigenvectors are the **coordinate axes** for what $[A]$ is doing.

Algorithm IX.1.3. Principal Component Analysis.

Given a dataset $[D]$ of (n rows and k columns) on variables $\vec{X} = (X_1, \cdots, X_k)$, we do the following:

(i) Estimate the vector of means:

$$\vec{\mu} = \left(\frac{1}{n}\sum_i x_{i1}, \frac{1}{n}\sum_i x_{i2}, \cdots, \frac{1}{n}\sum_i x_{ik} \right).$$

(ii) Then, we **zero-center** the data, that is to calculate a new dataset $[\tilde{D}]$ by subtracting μ_j from each realization of X_j in $[D]$.

(iii) Calculate the empirical covariance matrix:

$$\hat{C} = \frac{1}{n-1}[\tilde{D}]^*[\tilde{D}].$$

*We are using **conjugate transpose**, but for the simplicity of real numbers, it becomes the transpose only.*

(iv) Then, we compute the **eigendecomposition** of $\hat{C}$, that is:

$$\hat{C} = [V][\Lambda][V]^{-1}.$$

(v) Then, we sort the matrix $[\Lambda]$ and the corresponding vectors in $[V]$ in decreasing order of magnitude of eigenvalues.

(vi) Pick the $m < k$ largest eigenvalues, and construct a k-by-m submatrix $[V]_m$ of $[V]$ with the corresponding eigenvectors.

Eventually, we can project the center data as $[D'] = [\tilde{D}][V]_m$, so we reduce a n by k dataset to n by m dataset. ⌟

Here, we can consider the covariance matrix $\hat{C}$ as a map – it moves vectors around. In particular, it gives a **linear map** view of data variability. Moreover, the **eigenvectors** of $\hat{C}$ tell us the axes of this variability with corresponding eigenvalues telling us how big the movement is in a particular axis direction.

As PCA picks m largest eigenvalues, it is picking the **largest directions** of movements and applies a **linear transformation** of the data so those directions are preserved (like a **projection**).

A side note is that we cannot map any one feature in the transformed dataset $[D']$ back to one feature of the original centered data $[\tilde{D}]$ since each new feature is now a **linear combination** of all the original features.

Example IX.1.4. Eigenfaces.

An example that eigendecomposition PCA works is the **eigenfaces** example, PCA was applied to a set of black and white images of faces. The original data has:

$$\underbrace{256}_{\text{grayscale}} \times \underbrace{64 \times 64}_{\text{pixels}},$$

and a good subsequent performance could be achieved even in dimension 16. ⌟

Remark IX.1.5. PCA as Data Preprocessing Tool.

PCA is often used as a **preprocessing** step for other methods that rely on distances, such as clustering ***k*-nearest neighbors**, which does not work with high dimensions. The PCA often significantly improves performance of such method with **reduced features**. ⌟

The **PCA** method also has some disadvantages:

- There is not clear **cutoff** of how to choose discarding eigenvalues.
- The PCA, based on the **covariance matrix**, is a **fairly limited view of the data** in general.
- It is capturing every interesting thing only in special cases, such as **multivariated Gaussian data**. In general, **simple methods** like PCA cannot capture everything about the data.
  Analogically, in **classification problems** with linear classifiers, such as perceptron or SVM do not work well when it is not linear. *But in SVM, there are kernel tricks to get around.*

IX.2 Kernelizing PCA

Recall the **dual form** in SVMs, it allows us to extend to nonlinear cases, and we want to apply the same case to PCA.

Algorithm IX.2.1. Kerneling PCA.

The PCA algorithm finds all eigenvalues at once and apply **diagonalization**, but in fact, modern efficient methods find the eigendecomposition iteratively by finding the largest eigenvalue and project out the corresponding eigenvector and repeats.

(i) Start with a guess $\vec{v_0}$, we apply the following update rule:

$$\vec{v_{t+1}} = \frac{\hat{C}\vec{v_t}}{\|\hat{C}\vec{v_t}\|}.$$

(ii) We repeat the update rule until no change is detected. This will find an eigenvector $\hat{v_{\lambda_1}}$ corresponding to λ_1.

(iii) Then, we **project out** $\vec{v_{\lambda_1}}$ by replacing each row $\vec{x_i}$ in $[D]$ by:

$$\vec{x_i}^{(t+1)} = \vec{x_i}^{(t)} - \vec{v_{\lambda_1}}(\vec{x_i}^{\intercal}\vec{v_{\lambda_1}})$$

to yield dataset $[D]_2$ with a new empirical covariance matrix $\hat{C_2}$.

(iv) Then we iterate to find the next largest eigenvalue and continue. ⌟

Also, instead of original features $\vec{x}$ (1-by-k vector), we want to sue a high dimensional transformation $\phi(\vec{x})$ (1-by-k_ϕ vectors, where $k_\phi \gg k$) like with the SVMs.

Then, we construct a mean centered dataset $[\tilde{D}]$ as before, and calculate $\hat{C_\phi}$ as before:

$$\hat{C_\phi} = \frac{1}{n-1}\sum_{i=1}^{n}\phi(\vec{x_i})^{\intercal} \otimes \phi(\vec{x_i}).$$

Then, we need to solve the eigenvalue problem, that is:

$$\hat{C_\phi}\vec{v_\lambda} = \lambda\vec{v_\lambda} \text{ for the largest } \lambda.$$

The issue is that $\hat{C_\phi}$ and $\vec{\lambda}$ will match the dimension k_ϕ of $\phi(\vec{x})$, so we need a better representation.

Algorithm IX.2.2. Optimizing KPCA.

Recall that every eigenvector is a linear combination of transformed observed points, *i.e.*:

$$\vec{v_{\lambda_m}} = \sum_{i=1}^{n}\alpha_{mi}\phi(\vec{x_i})^{\intercal}.$$

Hence, finding $\vec{v_{\lambda_m}}$ is equivalent to finding λ_{m_i} for every i. Thus, we define the kernel as:

$$K(\vec{x_i}, \vec{x_j}) = \phi(\vec{x_i})\phi(\vec{x_j})^{\intercal},$$

which is n-by-n matrix, and if we substitute $\vec{v_{\lambda_m}}$, we have:

$$\begin{aligned}
\underbrace{\lambda_m \sum_{i=1}^{n}\alpha_{mi}\phi(\vec{x_i})^{\intercal}}_{\vec{v_{\lambda_m}}} &= \underbrace{\frac{1}{n-1}\sum_{j=1}^{n}\phi(\vec{x_j})^{\intercal}\phi(\vec{x_j})}_{\hat{C_\phi}}\underbrace{\sum_{i=1}^{n}\alpha_{mi}\phi(\vec{x_i})^{\intercal}}_{\vec{v_{\lambda_m}}} \\
&= \frac{1}{n-1}\sum_{j=1}^{n}\phi(\vec{x_j})^{\intercal}\sum_{i=1}^{n}a_{mi}K(\vec{x_i}, \vec{x_j}), \\
\lambda_m \sum_{i=1}^{n}\alpha_{mi}\phi(\vec{x_l})\phi(\vec{x_i})^{\intercal} &= \frac{1}{n-1}\sum_{j=1}^{n}\phi(x_l)\phi(\vec{x_j})^{\intercal}\sum_{i=1}^{n}a_{mi}K(\vec{x_i}, \vec{x_j}), \\
\lambda_m \sum_{i=1}^{n}\alpha_{mi}K(\vec{x_l}, \vec{x_i}) &= \frac{1}{n-1}\sum_{j=1}^{n}K(\vec{x_l}, \vec{x_j})\sum_{i=1}^{n}\alpha_{mi}K(\vec{x_i}, \vec{x_j}).
\end{aligned}$$

It turns out that we can write:

$$\lambda_m(n-1)\vec{\alpha_m} - K\vec{\alpha_m},$$

and this is another version of the eigenvalue problem with alternative representation of $\vec{v_{\lambda_m}}$ and $\hat{C_\phi}$.
In matrix form, we have:

$$\tilde{K} = K - \frac{2}{n}[1]K + \frac{1}{n}[1]K\frac{1}{n}[1],$$

where $[1]$ is a matrix with all entries as 1. ⌟

In summary, we do the following steps:

(i) Choose a kernel $K(\vec{x_i}, \vec{x_j})$.

(ii) Then, construct a centered kernel matrix of the data:
$$\tilde{K} = K - \frac{2}{n}[1]K + \frac{1}{n}[1]K\frac{1}{n}[1],$$
where $[1]$ is a matrix with all entries as 1.

(iii) Find the eigenvalues and eigenvectors of $\tilde{K}$ by solving the eigenvalue problem:
$$\tilde{K}\vec{\alpha_m} = \lambda_m\vec{\alpha_m}.$$

(iv) Scale each eigenvector by multiplying by the square root of the corresponding eigenvalue.

(v) Project the data onto the space spanned by the eigenvectors of $\tilde{K}$.

IX.3 Probabilistic PCA

Previously, we mentioned that an issue with PCA is that it does not have a **clear choice** of how many eigenvalues to keep, and a way of making this choice is to express PCA in terms of a **probabilistic mod**el and apply **model selection methods**.

Remark IX.3.1. Viewing the PPCA.

A **probabilistic PCA** may be viewed as:

- A hidden variable graphical model,
- A clustering method,
- A density estimation method,
- A dimension reduction method, and
- A missing data model.

⌟

Setup IX.3.2. Full Data Likelihood for PPCA.

Given a set of k variables $\vec{X}$, the PPCA assumes the following **full data likelihood**:
$$\vec{L} \sim \mathcal{N}(\vec{0}, \mathrm{Id}),$$
$$\vec{X} \sim \mathcal{N}(\vec{L} \cdot [W]^\intercal + \vec{\mu}, \sigma^2 \mathrm{Id}),$$
where $\vec{L}$ is a set of q latent variables ($q < k$), Id is the identity matrix of the appropriate size, $[W]$ is the matrix of weights, $\vec{\mu}$ is a mean offset, and σ^2 is the variance of every $X_i \in \vec{X}$. ⌟

In other words, the observed variables $\vec{X}$ are assumed to be:

- Gaussian,
- Independent conditional on a smaller set of **latent Gaussian** $\vec{L}$, which are themselves **mutually independent**, and
- $\vec{X}$ are all linear combinations of $\vec{L}$ via the weight matrix $[W]$.

All the parameters of this model are $[W]$, $\vec{\mu}$, and σ^2, so an alternative representation of $\vec{X}$ is:

$$\vec{X} \sim \mathcal{N}(\vec{\mu}, [W][W]^\intercal + \sigma^2 \operatorname{Id}).$$

Example IX.3.3. PPCA as Graphical Model.

Consider the PPCA with 5 observed features and 2 latent feature, we can represent it graphically as:

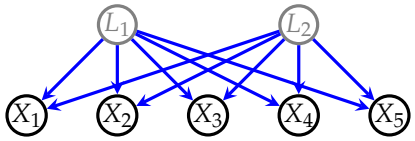


Figure IX.1. PPCA model represented graphically, with latent Gaussian variables (gray) and observed Gaussian variables (black).

The **latent Gaussian variables** are independent of each other, and the **observed Gaussian variables** are independent given latent state variables.
This is very similar to a **mixture model**, except that the latent states are real numbers rather than categorical values. ⌟

Then, we would want to find the **maximum likelihood estimation** for PPCA.

Algorithm IX.3.4. Closed Form Iterative Approach by EM.

Consider the log likelihood for observed data:

$$\log \mathcal{L}_{[D]}(\vec{\mu}, \sigma^2, [W]) = -\frac{n}{2}\left[k \log(2\pi) + \log\left(\det\left([W][W]^\intercal + \sigma^2 \operatorname{Id}\right)\right) + \operatorname{tr}\left(([W][W]^\intercal + \sigma^2)^{-1}\hat{C}\right)\right],$$

and the **closed form solution** for **MLE parameter** values are:

$$\hat{\vec{\mu}} = \frac{1}{n}\sum_{i=1}^{n} \vec{x}_i,$$

$$\hat{\sigma}^2 = \frac{1}{k-q}\sum_{j=q+1}^{k} \lambda_j,$$

$$[\hat{W}] = [U]_q([\Lambda]_q - \sigma^2 \operatorname{Id})^{1/2} \operatorname{Id},$$

where $[U]_q$ is a k-by-q matrix with principal eigenvectors of $\vec{C}$, and $[\Lambda]_q$ is a q-by-q matrix with eigenvalues $\lambda_1, \cdots, \lambda_q$. ⌟

Algorithm IX.3.5. EM Algorithm for PPCA.

Let the full data likelihood being:

$$\log \mathcal{L}_{\text{full}}(\vec{\mu}, \sigma^2, [W]) = \sum_{i=1}^{n} \log \mathbb{P}(\vec{x}_i, \vec{l}_i),$$

which is equal to:

$$\sum_{i=1}^{n}(2\pi\sigma^2)^{-k/2}\exp\left[-\frac{\|\vec{x}_j-[W]\vec{I}_i-\hat{\vec{\mu}}\|^2}{w\sigma^2}\right](2\pi)^{-q/2}\exp\left[-\frac{\|\vec{I}_i\|^2}{2}\right].$$

Here, we have:

- The E step as taking the expectation of each row term w.r.t $\mathbb{P}(\vec{I}_i \mid \vec{x}_i; [W]^{(t)}, (\sigma^2)^{(t)})$, and dropping terms that do not depend on the model parameters.
- In the M step, we maximize $\mathbb{E}\left[\log \mathcal{L}_{[D]}(\vec{\mu}, \sigma^2, [W]) \mid [D]; \vec{\beta}^{(t)}\right]$ w.r.t σ^2.

As usual, we repeat until there are no parameter improvements. ⌟

The general EM result earlier applies here, that is, each step will **improve the value of the log likelihood**.

Remark IX.3.6. Missing Data Problem.

Since we have a model of the full data distribution, we will be able to deal with missing entries if the data is **missing at random** (MAR).

- For every row $\vec{x}_i$ containing a set of missing entries $\vec{z}_i$, simply calculate $\mathbb{P}(\vec{z}_i \mid \vec{x}_i)$ from the full data distribution $\mathbb{P}(\vec{L}, \vec{X})$, which is Gaussian at current parameter guess $\vec{\beta}^{(t)}$, and
- Proceed as before to impute new values at each step. ⌟

Remark IX.3.7. Computational Complexity for PCA.

Consider the approaches to solve PCA, we compute the **sample covariance**, which has runtime $\mathcal{O}(nk^2)$, and it is a problem for large k, so we should think of using PPCA.
With the update rule for PPCA, it is ending up as $\mathcal{O}(nkq)$, and for a not too large k, the PPCA should be **faster**. ⌟

X Reinforcement Learning

Reinforcement learning (RL) is a semi-supervised problem and we want to build an agent that learns to act by interacting with the environment.

In the process, the **agent gets feedback** after acting, hence there are supervision, but is often delayed.

For a turn-based game:

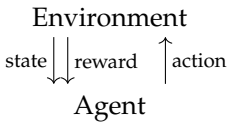
- **Task**: choose the best move given a board position.
- **Feedback**: win/lose/draw at the end of the game.
- **Training**: an agent can play a lot of games with copies of itself.

- **Results**: a proficient player (matching or exceeding the best human).

For acting in real time environment:

- **Task**: choose an **action at a particular moment** in time, given a representation of the environment.
- **Feedback**: good or bad outcomes.
- **Training**: agent playing with itself or in different environments.
- **Results**: Successes in automated machinery.

In general, the **schematic of reinforcement learning** is to have:



X.1 Markov Decision Processes

A common model for RL is a **Markov Decision Process** (MDP).

Setup X.1.1. Setup for Markov Decision Process.

The MDPs contain:

- **A set of states** $\vec{S}$: represent possible situations in the environment an agent might encounter.
  This could be a board position in a chess game, namely, $s_i \in \vec{S}$.
- **A set of actions** $\vec{A}$: represent possible actions the agent may take at each state.
  Sometimes, only a subset of $\vec{A}$ is allowed in a particular state s_i.
  This could be a legal move in a board game.
- **A state transition distribution** $\mathbb{P}\big(s^{(t+1)} \mid s^{(t)}, a^{(t)}, \cdots, s^{(0)}, a^{(0)}\big)$: governs how likely a sequence of state visits and actions up to time t are to lead to a particular new state at time $t+1$.
  In an MDP, we make a crucial simplification that:
  $$\mathbb{P}\big(s^{(t+1)} \mid s^{(t)}, a^{(t)}, \cdots, s^{(0)}, a^{(0)}\big) = \mathbb{P}\big(s^{(t+1)} \mid s^{(t)}, a^{(t)}\big),$$
  i.e., state transitions obeys **Markov property** and only depend on the last state and last action taken, not any prior history.
- **A reward function** $R_{a_k}(s_i, s_j)$: maps a state transition (s_i, s_j) if action a_k was taken to a reward r_{ij}.
  There could be no reward to some (if not most) state transition and action triplets.

Also, note that MDPs assume **discrete time**, *i.e.*the agent starts in some state $s^{(0)} \in \vec{S}$ at time 0, and moves to other states after choosing to take an action $a^{(0)}$.
The agent eventually finds itself in state $s^{(t)}$ at time t, and chooses to take an action $a^{(t)}$. ⌟

The MDP is an ideal model, it assume that state transition is *forgetful*, and time is discrete. This might **not be possible** in reality.

Example X.1.2. Represent MDP using Graphical Model.

Consider the following MDP:

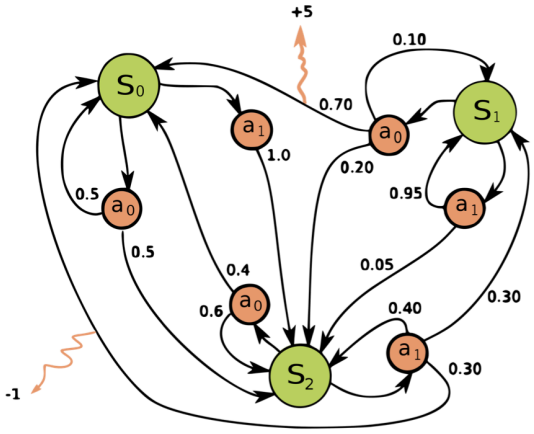


Figure X.1. MDP represented as a DAG model.

Here, it has 3 states, 2 possible actions, and the reward functions are only defined for $R_{a_0}(s_1, s_0) = 5$ and $R_{a_1}(s_2, s_0) = -1$. There are **no rewards** for any other triplets. ⌟

Definition X.1.3. Absorbing States.

Assume that we start at some state $s^{(0)} \in \vec{S}$, in modeling applications like games, we can consider **absorbing state** as a special state in which once the agent reaches such as state, it stays there forever. *This is like the end of a game like chess*. ⌟

Here, assume the upper bound T as the maximum number of time steps to reach absorbing state, the agent could reasonably maximize expected total reward given a sequence of actions $\{a_i\}_{i=0}^{T}$, and we have the expected reward as:

$$\mathbb{E}\left[\sum_{t=0}^{T} R_{at}\left(s^{(t)}, s^{(t+1)}\right)\right].$$

This expectation is w.r.t:

$$\mathbb{P}_{a_0,\cdots,a_T}\left(s^{(1)}, \cdots, s^{(T)}\right) = \prod_{t=1}^{T} \mathbb{P}\left(s^{(t)} \mid s^{(t-1)}, a_t\right).$$

Setup X.1.4. Infinite Horizon Problem.

To model applications where agents act continuously, there is no specific stopping points, we consider such as the **infinite horizon problem**. Then, we have the expected reward as:

$$\mathbb{E}\left[\sum_{t=0}^{\infty} R_{at}\left(s^{(t)}, s^{(t+1)}\right)\right].$$

However, this raise the problem that the expectation could be divergent, *i.e.*, we have **infinite rewards**.

Then, we introduce a **discount factor** $0 \leq \gamma \leq 1$ such that the reward we maximize is:

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_{at}\left(s^{(t)}, s^{(t+1)}\right)\right],$$

where having smaller γ favors **instant gratification** over **delayed gratification**. ⌟

In an MDP the reward depends on the current state, the action taken and the next state we transition to. At any time t, the agent sees $s^{(t)}$.

Definition X.1.5. Optimal Policies.

The learning problem is to learn a policy $\pi : s^{(t)} \to a_i$ that maximizes **expected reward** (or **discounted expected reward**). The optimal policy would be:

$$\pi^* = \text{argmax}_\pi \, \mathbb{E}\left[\sum_{t=0}^{T} R_{at}\left(s^{(t)}, s^{(t+1)}\right)\right], \text{ or}$$

$$\pi^* = \text{argmax}_\pi \, \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_{at}\left(s^{(t)}, s^{(t+1)}\right)\right].$$

⌟

This is **not** a **supervised learning problem**, since we do not see $(s^{(t)}, a_i)$ pairs as training data. Hence, we have to learn indirectly by somehow **associating actions with subsequent rewards**.

Remark X.1.6. Existence of Optimal Policy.

An optimal policy π^* is not necessarily unique, but at least one always exists. ⌟

X.2 Value Function and Iteration

As of right now, we consider the **infinite horizon problem** where we know that state transition probabilities $\mathbb{P}(s^{(t+1)} \mid s^{(t)}, a_t)$.

Definition X.2.1. Value Function.

Given a fixed policy π, we can define the value function for any state s_i as:

$$V^\pi(s_i) = \sum_{t=0}^{\infty} \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_{at}\left(s^{(t)}, s^{(t+1)}\right)\right],$$

where $s^{(0)} = s_i$. ⌟

In other words, if we are at state s_i, the value function $V^\pi(s_i)$ will tell us the **expected reward** of following the policy π from the point on. If we we know the **value function** for a fixed policy π, we could choose the π that gives us the largest expected reward.

Then, we want to learn how we can learn value functions, we let $V^*(s_0)$ denote the value function for an

optimal policy π^* staring at s_0, then we have the following recursive definition:

$$V^*(s_0) = \underbrace{\mathbb{E}\left[R_{a_0=\pi^*(s_0)}(s_0, s^{(1)})\right]}_{\text{expected reward from taking optimal action at } s_0} + \underbrace{\gamma \mathbb{E}_{\mathbb{P}\left(s^{(1)} | s_0, a_0=\pi^*(s_0)\right)}\left[V^*(s^{(1)})\right]}_{\text{discounted expected reward of following } \pi^* \text{ afterwards}}.$$

Algorithm X.2.2. Value Iteration.

Hence, this suggests a recursive algorithm for learning $V^*(s)$ for $s \in \vec{S}$.

(i) We set $V^{(0)}(s) = 0$ for all $s \in \vec{S}$.

(ii) For $t = 1, 2, \cdots$, for each $s \in \vec{S}$, and for each $a \in \vec{S}$, we compute:

$$Q^{(t)}(s, a) = \sum_{s^{(t)} \in \vec{S}} R_a(s, s^{(t)}) \mathbb{P}(s^{(t)} \mid s, a) + \gamma \sum_{s^{(t)} \in \vec{S}} V^{(t-1)} \mathbb{P}(s^{(t)} \mid s, a).$$

Then, we assign $\max_a Q^{(t)}(s, a)$ to $V^{(t)}(s)$.

Then, as $t \to \infty$, we have $V^{(t)}(s)$ converging to $V^*(s)$ for each $s \in \vec{S}$. ⌟

One note of the **iterative method** is that it involves **dynamic programming**, *i.e.*, memorizing sums of rewards over many possible trajectories by a state/action index, assuming we know:

$$\mathbb{P}(s^{(t+1)} = s_j \mid s^{(t)} = s_i, a_k) \text{ and } R_{a_k}(s_i, s_j) \text{ for all } s_i, s_j \in \vec{S}, a_k \in \vec{A}.$$

If there are a lot of states, it would be difficult to specify in advance what all the transition probabilities and rewards are.

Alternatively, we can learn them by repeatedly navigating the environment.

Setup X.2.3. Motivation to Learning Q Function.

Recall that we have learning with the Q function:

$$Q^{(t)}(s, a) = \underbrace{\sum_{s^{(t)} \in \vec{S}} R_a(s, s^{(t)}) \mathbb{P}(s^{(t)} \mid s, a)}_{\text{expected reward for doing action } a_k \text{ in state } s_i} + \underbrace{\gamma \sum_{s^{(t)} \in \vec{S}} V^{(t-1)} \mathbb{P}(s^{(t)} \mid s, a)}_{\text{expected discount reward of using the optimal policy } \pi^* \text{ after}}.$$

Therefore, we want to learn $Q(s_i, a_k)$ directly, **without knowing transition probabilities and rewards in advance**, provided the environment supplies them as we explore it.
The key of this is the relationship between the **Q function** and the **value function**. ⌟

If we know $Q(s, a)$ for every $s \in \vec{S}$ and $a \in \vec{A}$, we could choose the optimal policy as $\pi^*(s) = \text{argmax}_a Q(s, a)$. Assume that all state transitions are deterministic, we can learn $Q(s, a)$ directly by noting that:

$$\begin{aligned} Q(S_i, a_k) &= \underbrace{R_{a_k}(s_I, s_j)}_{\text{determinism}} + \sum_{s_j \in \vec{S}} \gamma V^*(s_j) \mathbb{P}(s_j \mid s_i, a_k) \\ &= R_{a_k}(s_I, s_j) + \sum_{s_j \in \vec{S}} \gamma \left(\max_a Q(s_j, a)\right) \mathbb{P}(s_j \mid s_i, a_k). \end{aligned}$$

This suggests a similar algorithm for **learning Q function** $Q(s, a)$ as we had for value functions $V^*(s)$.

Algorithm X.2.4. Learning Q Function for Deterministic Case.

We have the following algorithm on **Q learning** with deterministic case:

(i) Initialize $\hat{Q}(s,a) = 0$ for all $s \in \vec{S}$ and $a \in \vec{A}$, and let the discount factor γ be fixed.

(ii) For $t = 1, 2, \cdots$:

- In current state s_i, somehow pick an action a.
- Execute action a, deterministically transition to state s_j and receive reward $R_a(s_i, s_j)$.
- Update $\hat{Q}(s_i, a)$ as:
$$R_a(s_i, s_j) + \gamma \max_{\tilde{a}} \hat{Q}(s_j, \tilde{a}).$$
- Set the current state to s_j.

⌟

It turns out that this simple algorithm is enough, *i.e.*, $Q(\hat{s}, a)$ converges to $Q(s,a)$ for all $s \in \vec{S}$ and $a \in \vec{A}$ provided each state and action pair occurs infinitely often.

It is possible to show that $\max_{\tilde{a}} \hat{Q}(s_j, \tilde{a})$ converges to $V^*(s_j)$, which suffices to show that $\hat{Q}(s_i, a)$ converges to $Q(s_i, a)$ by definition.

Algorithm X.2.5. Q Learning for Non-deterministic Case.

We have the following algorithm on **Q learning** with non-deterministic case:

(i) Initialize $\hat{Q}(s,a) = 0$ for all $s \in \vec{S}$ and $a \in \vec{A}$, and let the discount factor γ be fixed.

(ii) For $t = 1, 2, \cdots$:

- In current state s_i, somehow pick an action a.
- Execute action a, non-deterministically transition to state s_j and receive reward $R_a(s_i, s_j)$.
- Update $\hat{Q}(s_i, a)$ as:
$$(1 - \alpha^{(t)})(s_i, a) + \alpha^{(t)} \left(R_a(s_i, s_j) + \gamma \max_{\tilde{a}} \hat{Q}(s_j, \tilde{a}) \right),$$
where $\alpha^{(t)} = \frac{1}{1 + \#\text{ times } (s,a) \text{ was visited}}$.
- Set the current state to s_j.

⌟

It turns out that this simple algorithm also result in $Q(\hat{s}, a)$ converges to $Q(s,a)$ for all $s \in \vec{S}$ and $a \in \vec{A}$ provided each state and action pair occurs infinitely often.

Remark X.2.6. Tradeoffs on Q Learning.

Recall that in the **Q learning algorithm**, given the current state s_i, we **somehow picked an action** a.
If the agent has been exploring the environment for a while, there comes to be tradeoffs: we may know some **actions lead to sure thing rewards**, but we may also have some actions that **lead to underexplored regions** of the space which could contain greater rewards.
This is known as the **exploration/exploitation tradeoff**:

- **ϵ-greedy exploration** (model-free):
  We set a parameter ϵ that goes to 0 as the agent explores. In every state s, choose a random action with probability ϵ, and choose the best currently known action with probability $1-\epsilon$, so it is likely to explore the unexplored actions.
- **Thompson sampling** (model-based):
  We learn a probability model that estimates the probability $\mathbb{P}^*_{a,s}$ that the action a maximizes expected reward in state s. Then, at each state s, we choose the action a with highest probability $\mathbb{P}^*_{a,s}$. ⌟

If the set of (s,a) is very large, they can no longer be represented as a table. Here, we would try to **incrementally learn a model**, using f implemented as a **big neural network**.

Algorithm X.2.7. Model Based RL.

We have the following algorithm modified from **Q learning**, with implementation of model $f : s \mapsto \big(a, \hat{Q}(s,a)\big)$:

(i) Initialize the model parameters $\hat{\vec{\beta}}$ in f, and let the discount factor γ be fixed.

(ii) For $t = 1, 2, \cdots$:

- In current state s_i, evaluate $f(s_i; \hat{\vec{\beta}})$ to yield a set of action/Q function pairs:
$$\big(a_1, \hat{Q}(s_i, a_1)\big), \cdots, \big(a_k, \hat{Q}(s_i, a_k)\big).$$
- Then, we somehow pick an action a.
- Execute action a, non-deterministically transition to state s_j and receive reward $R_a(s_i, s_j)$.
- We here modify $\hat{\vec{\beta}}$ in g by providing the outcome $\big(am\hat{Q}^{(t+1)}(s_i, a)\big)$.
- Set the current state to s_j.

⌟

The algorithm is fundamentally the same as **Q Learning**, but its uses tricks to yield powerful models.