

Machine Learning 1

Machine Learning 1

1. Introduction to Machine Learning

Useful probability and statistics facts

Inductive bias

Formulation of ML

Prediction vs. inference

Common Tasks

Setting up the tasks

Optimization view

Statistics view

General form of a linear model

On data pre-processing

2. Linear Data Visualization

Dimensionality reduction

Principal Components Analysis

Fisher's Discriminant Analysis

Counterparts

3. Theory for regression and linear models (I).

The regression framework

Bias-Variance analysis

4. Regression theory and linear regression models (II)

Quality of the fit

Leaping forward: basis functions

Singular Value Decomposition

SVD for least squares

Regularized least squares

LASSO Regression

Conclusions

5. Classification theory and linear classification models (I). Bayesian decision theory.

Introduction: Bayes' formula

Decision rules

The Bayes classifier

The notion of risk

0/1 losses

Discriminant functions

The Gaussian Distribution

Properties

6. Classification theory and linear classification models (II).

Generative Bayesian classifiers

Discriminant functions for the Gaussian density

A numerical example

Computations in practice

Discussion

Regularized Discriminant Analysis

Pros & cons

The Naive-Bayes classifier

Extensions

Null empirical probabilities

The kNN classifier

7. Classification theory and linear classification models (III).

Discriminative classifiers

Maximum Likelihood (ML) framework (I)

- Generalized Linear Methods
 - Logistic regression
 - Interpretation of the model
 - Parameter obtaining
 - Newton-Raphson
 - Deviance and AIC
 - Interpreting the coefficients
 - Poisson Regression
 - Maximum Likelihood framework (II)
- 8. Artificial Neural Networks (I): The Multilayer Perceptron (MLP)
 - How to train a single layer ANN: the Delta Rule
 - Towards non-linear models
 - Error functions for classification
 - Training MLPs: An introduction to the backpropagation algorithm for regression
 - Relation between backpropagation and order methods
 - Tricks of the trade: sensible choices when implementing and training MLPs
 - Number of hidden neurons
- 9. Artificial Neural Networks (II): RBF Networks
 - Introduction
 - Regularization
 - RBF networks
- 10. Ensemble methods: Random Forests
 - Methods for constructing ensembles
 - Bootstrap Resampling
 - Random Forests

1. Introduction to Machine Learning

Machine learning is a field that lies at the intersection of statistics, probability, computer science, and optimization. The main goal is to explore **automatic methods for inferring models from data** (for example: finding structure, making predictions).

Examples of learning tasks:

- **SUPERVISED LEARNING:** uses labeled data.
 - **Classification:** predicting a class or category to each example; note multi-label, probabilistic generalizations.
 - **Regression:** predicting a real value for each example; note multi-variable generalization.
- **UNSUPERVISED LEARNING:** does not use or have data labels.
 - **Clustering:** discovering homogeneous groups (clusters) in data.
 - **Dimensionality reduction:** finding lower-dimensional data representations.
 - **Density estimation:** estimating the probabilistic mechanism that generates data.
 - **Novelty detection:** finding anomalous/novel/outlying data.
- **SEMI-SUPERVISED LEARNING:** uses partly labeled data.
 - **Ranking:** ordering examples according to some criterion.
 - **Reinforcement:** delayed rewarding.
- **TRANSFER LEARNING:** learning in a new task through the transfer of knowledge from a related task that has already been learned.

Useful probability and statistics facts

- **Central Limit Theorem:**

If X_1, \dots, X_n are *independent identically distributed random variables*, with $\mathbb{E}[X_i] = \mu$ and $\text{Var}(X_i) = \sigma^2$, then the sample mean

$$\frac{X_1 + \dots + X_n}{n} \sim \mathcal{N}\left(\mu, \frac{\sigma^2}{n}\right)$$

approaches a normal distribution as $n \rightarrow \infty$.

- **Product rule:**

If X_1, \dots, X_n have a joint probability distribution $p(X_1, \dots, X_n)$, then we can factorize the distribution as the product

$$p(X_1, \dots, X_n) = p(X_1) \prod_{i=2}^n p(X_i | X_1, \dots, X_{i-1}).$$

- **Bayes Theorem:**

$$P(B_i | A) = \frac{P(A|B_i)P(B_i)}{\sum_j P(A|B_j)P(B_j)} = \frac{P(A|B_i)P(B_i)}{P(A)}.$$

- **Bayes formula for densities:**

In a data analysis context, θ is a parameter vector and the following equality holds:

$$\pi_{\text{POST}}(\theta | \text{data}) = \frac{\pi_{\text{LIK}}(\text{data} | \theta) \cdot \pi_{\text{PRIOR}}(\theta)}{\int_{\Theta} \pi_{\text{LIK}}(\text{data} | \theta) \cdot \pi_{\text{PRIOR}}(\theta) d\theta}.$$

This can also be expressed loosely as

$$P(\theta | D) = \frac{P(D | \theta)P(\theta)}{P(D)} = \frac{P(D | \theta)P(\theta)}{\int_{\Theta} P(D | \theta)P(\theta) d\theta},$$

where D is the data. This expression gives rise to the notions of **likelihood**, **prior**, **posterior**, and **unconditional (expected likelihood)** distributions:

- $P(\theta)$: prior probability, confidence in θ before observing D .
- $P(D|\theta)$: likelihood, probability of observing D if parameters are θ .
- $P(D)$: expected likelihood of observing data D , also unconditional.
- $P(\theta|D)$: posterior probability, confidence in θ after observing D .

- **Conjugacy:**

Definition: Suppose a prior distribution $\pi_{\text{PRIOR}}(\theta)$ belongs to a class of parametrized distributions Π . Then the distribution is said to be **conjugate** with respect to a likelihood $\pi_{\text{LIK}}(\cdot | \theta)$ if the posterior distribution $\pi_{\text{POST}}(\theta | \cdot) \in \Pi$.

Remember that $\pi_{\text{POST}}(\theta | \cdot) \propto \pi_{\text{LIK}}(\cdot | \theta) \pi_{\text{PRIOR}}(\theta)$. For example, Gaussian is conjugate to Gaussian, and Beta is conjugate to Binomial.

Using the posterior:

$\hat{\theta}_{\text{MAP}} := \text{argmax}_{\theta \in \Theta} \{P(\theta | D)\}$: the value of θ that maximizes the posterior.

$\hat{\theta}_{\text{ML}} := \text{argmax}_{\theta \in \Theta} \{P(D | \theta)\}$: the value of θ that maximizes the likelihood.

$\hat{\theta}_{\text{EV}} := \mathbb{E}[P(\theta | D)] = \int_{\Theta} P(\theta | D) \cdot P(\theta) d\theta$: the expected value of theta.

Inductive bias

Example: complete the following series: 2, 4, 6, 8, ...

Answer 1: 132 (model 1: $f(n) = n^4 - 10n^3 + 35n^2 - 48n + 24$)

Answer 2: 10 (model 2: $f(n) = 2n$)

How can we rule out the more complex model?

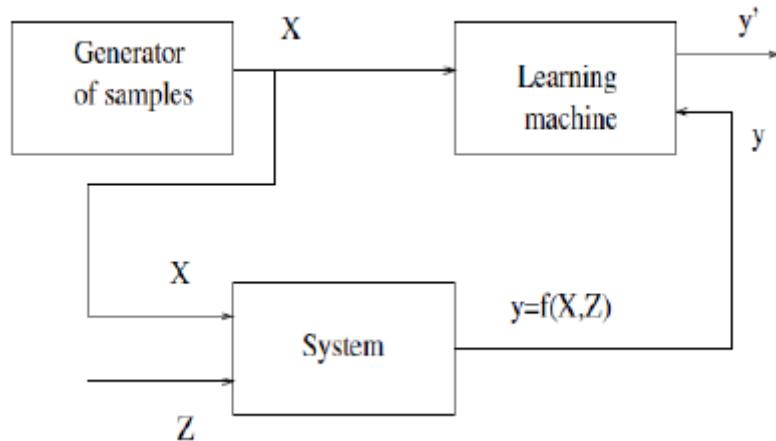
1. Supply more training data: 2, 4, 6, 8, 10, 12, 14, ...
2. Regularize: add penalty to higher-order terms.
3. Reduce the hypothesis space; for example, restrict to quadratic models.

Class of functions: $\mathcal{F} := \{f_\theta : \mathcal{X} \rightarrow \mathcal{Y} \mid \theta \in \Theta\}$.

So, the conclusions are this: based *only* on training data D , there is no means of choosing which function f is better (generalization is not *guaranteed*). Thus, we must add control to the **fitting ability** of our methods (complexity control).

$$\text{true error}(f) \leq \text{training error}(f) + \text{complexity of } f.$$

Formulation of ML



X are the measured variables, Z are the unmeasured ones, y is the true function and y' , which would be \hat{y} , is the modeled function.

Prediction vs. inference

Prediction: produce a good estimate for the predicted variable.

Inference:

1. Which predictors actually affect the target variable?
2. How strong are these dependencies?
3. Are these relationships positive or negative?

Common Tasks

- **Regression:** The goal is to predict some quantitative outcome subject to probabilistic uncertainty.
- **Classification:** The goal is to obtain a model based on available **training data** (*known* examples) with high classification accuracy on *unseen unknown* examples (**test data**), i.e. achieving good **generalization**.
- **Clustering:** The goal is to find homogeneous groups of data and set them apart accordingly. Looks like a very different task from regression or classification, but it's both of them with some added difficulty: it has an inherent large subjectivity.

Why are these tasks stochastic?

We have a (complete) input data object (x, z) and an output data object y . The true relationship is $f_c : \mathcal{X} \times \mathcal{Z} \rightarrow \mathcal{Y}$, that is $f_c(x, z) = y$. When we measure data about f_c , we measure only the x portion of the input variables. Therefore, the relation between x and y becomes stochastic.

Setting up the tasks

There are (at least) two ways of setting up these tasks formally:

Optimization view

$$\min_{\theta \in \Theta} E(\theta) := \frac{1}{n} \sum_{i=1}^n l(y_i, f_\theta(x_i)) + \Omega(f_\theta).$$

true error of $f_\theta \leq$ training error of f_θ + complexity of f_θ (empirical risk + regularizer)

$l(y_i, f_\theta(x_i))$ is called the *loss/error function*.

Statistics view

Use Bayes' formula to compute $P(\theta|\text{data})$ and choose one according to this (posterior) distribution.

Many times these two views can yield the same results (which is good!). An example would be LSQ \equiv MaxLik+Gaussian.

The most general description of the data generation mechanism is in terms of the pdf $p(x, y)$ in the joint input-output space: this is the key to generalization.

$$p(x, y) = p(y|x) \cdot p(x), \text{ where } p(x) = \int p(y, x) dy.$$

Some techniques use $p(x)$, others do not. The important pdf is $p(y|x)$. *Discriminative* methods use only $p(y|x)$, while *generative* methods use the joint pdf $p(x, y)$.

So, what is a Machine Learning algorithm/technique?

A ML algorithm gets a dataset D and returns a model of D (a representation of D that either gives structure to D or that allows to make predictions on unseen observations), together with an estimation of the model quality. The algorithm itself typically determines the model space \mathcal{F} and the loss function l .

And why are linear models so nice?

We will begin our analyses with linear models and techniques. A model is linear when, up to an invertible mapping, it is a **linear function of its parameters**: $f_\theta(x)$ is linear when it depends linearly on θ , but we do not say anything about x . For example, $f_\theta(x) = \sum_{i=0}^m \theta_i \sin(\exp(-x_i^2))$ is linear. A linear model:

- **Is analytically tractable:** we have closed-form solutions or fast convergent iterative methods for the solution.
- **Has a unique solution:** there are no local optima.
- **Is highly interpretable.**
- **Is amenable to inference:** we can ask (and answer) questions about the importance and weight on the target of the different variables.
- Has **user-defined fitting ability**, via the basis functions.
- Is capable of being **regularized**: complicated models can be penalized.

General form of a linear model

A linear model has a general expression as

$$f(x; \theta) = g \left(\theta_0 + \sum_{i=1}^h \theta_i \varphi_i(x) \right).$$

The functions φ_i are called **basis functions** (they constitute a *feature map*) and are non-linear wrt x . g is a strictly monotonic function: in Neural Networks, this is called an **activation function**.

On data pre-processing

Each problem requires a different approach in what concerns data cleaning and preparation. This pre-processing procedure is very important because it can have a deep impact on performance; it can easily take us a significant part of the time. So, the important things to take into account on data pre-processing are:

- Treatment of missing, anomalous, and incoherent or incorrect values.

- Coding of non-continuous or non-ordered variables.
- Possible elimination of irrelevant or redundant variables (*feature selection*).
- Creation of new variables that can be useful (*feature extraction*).
- Normalization of the variables (standardization).
- Transformations of the variables (for example, corrections of serious skewness and/or kurtosis)

Non-standard data (images, audio, text...) may need completely *ad hoc* treatments.

2. Linear Data Visualization

Dimensionality reduction

There are two main goals associated to these techniques:

- **Signal representation:** the goal is to represent the data accurately in a lower-dimensional space.
- **Signal classification:** the goal is to enhance the class-discriminatory information in the lower-dimensional space.

Unfortunately, there is no systematic way to generate non-linear transforms, so we will focus on **linear** methods for **feature extraction**:

- **PCA:** Principal Components Analysis.
- **FDA/LDA:** Fisher's Discriminant Analysis.
- **ICA:** Independent Components Analysis.

Principal Components Analysis

This feature extraction method is explained in the file [.../AD/AD.pdf](#).

Fisher's Discriminant Analysis

FDA is a technique for **dimensionality reduction, supervised classification, feature extraction and data visualization**.

Idea: projection of the data onto a lower dimensional linear space, such that the separability of projected data is maximized.

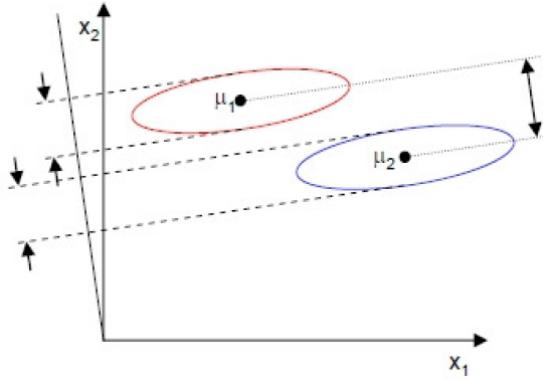
Fisher's idea is to regard **dot product** as the projection y of some $x \in \mathbb{R}^p$ from classes ω_1 or ω_2 , via a projection vector w : $y = w^T x \in \mathbb{R}$. In order to find a good projection vector, we need to define a measure of separation between the projections:

$$m_k = \frac{1}{n_k} \sum_{i \in \omega_k} x_i, \quad k \in \{1, 2\},$$

where $n_1 + n_2 = n$ is the number of examples on every class. We then choose to maximize the *squared* distance between the projected means,

$$(\mu_2 - \mu_1)^2 = (w^T m_2 - w^T m_1)^2 = (w^T(m_2 - m_1))^2.$$

However, the distance between the projected means is not a very good measure since it does not take into account the dispersion (**scatter**) within the classes. The problem is that the covariance matrices for each class are far from being diagonal. We actually want to look for the projection where examples from the same class are projected very close to one another and the projected means are as far apart as possible:



The solution (proposed by R. Fisher) is to maximize a function that represents the difference between the means, normalized by a measure of the within-class scatter:

1. $\forall k$ a class we define the scatter as

$$s_k^2 = \sum_{i \in \omega_k} (w^T x_i - \mu_k)^2, \quad k \in \{1, 2\}.$$

2. The total scatter is $s_1^2 + s_2^2$.

3. Fisher's idea was to maximize the following function:

$$J(w) = \frac{(\mu_2 - \mu_1)^2}{s_1^2 + s_2^2}.$$

It can be shown that $J(w)$ can be rewritten as:

$$J(w) = \frac{(\mu_2 - \mu_1)^2}{s_1^2 + s_2^2} = \frac{w^T S_B w}{w^T S_W w},$$

where

- $S_B = (m_2 - m_1)(m_2 - m_1)^T$ is the **between-class scatter matrix** (rank 1).
- $S_W = \sum_{i \in \omega_1} (x_i - m_1)(x_i - m_1)^T + \sum_{i \in \omega_2} (x_i - m_2)(x_i - m_2)^T$ is the **within-class scatter matrix**.

To find the maximum of J we derive and equal to zero,

$$\frac{\partial J}{\partial w} = 0,$$

and upon solving we arrive at the following **generalized eigenvalue problem**:

$$(S_W^{-1} S_B) w = J(w)w;$$

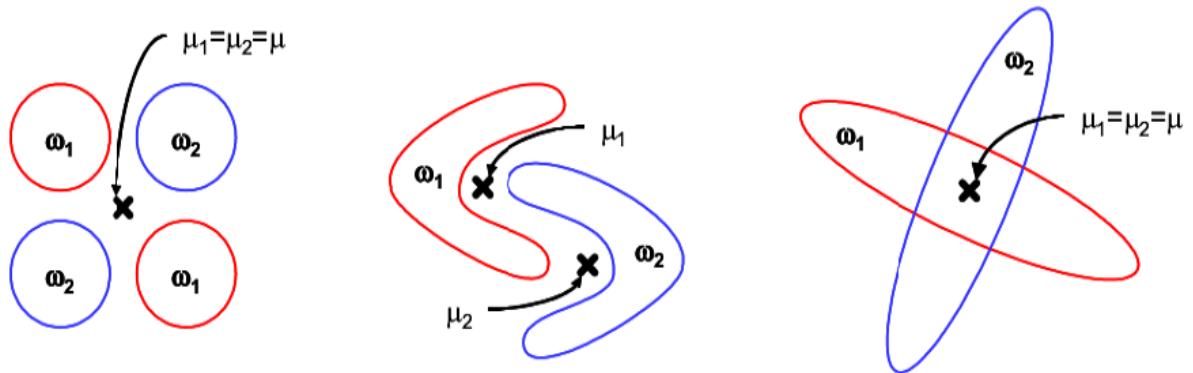
solving it yields $\hat{w} = S_W^{-1}(m_1 - m_2)$, known as **Fisher's Linear Discriminant** (1936), although it is not a discriminant but a specific choice for projection down to one dimension.

FDA generalizes very gracefully for K class problems: the only restriction is that the maximum number of projection directions is $K - 1$. FDA can also be derived as the Maximum Likelihood result for the case of Gaussian class-conditional densities with equal covariance matrices; in this case, it is known as LDA.

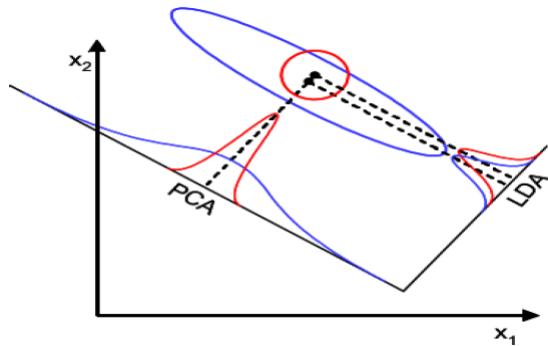
WARNING! FDA is able to extract a maximum of $K - 1$ projection directions, maybe insufficient for complex data. PCA is able to extract d projection directions, but it is not clear how many are necessary.

Counterparts

When will FDA presumably fail? If the classes are far from Gaussian, the FDA projections will not be able to preserve any complex structure; for an example, this image:



FDA will also fail when the discriminatory information is not in the mean but rather in the *variance* of the data (e.g., if $J(w) = 0$); for example,



3. Theory for regression and linear models (I).

The regression framework

Given data $D = \{(x_n, t_n)\}_{n=1,\dots,N}$, where $x_n \in \mathbb{R}^d, t_n \in \mathbb{R}$,

- **Statistics:** estimation of a continuous random variable T conditioned on a random vector X .
- **Mathematics:** estimation of a real function f based on a finite number of *noisy* examples $(x_n, f(x_n))$.

The departing statistical setting is $t_n = f(x_n) + \varepsilon_n$; a model is any approximation of f . We assume ε_n are iid random variables such that $\mathbb{E}[\varepsilon_n] = 0$ and $\text{Var}(\varepsilon_n) = \sigma^2 < \infty$, and that ε_n and x_n are independent variables.

The **risk** of a model y is

$$R(y) := \int_{\mathbb{R}} \int_{\mathbb{R}^d} L(t, y(\mathbf{x})) p(t, \mathbf{x}) d\mathbf{x} dt,$$

where L is a suitable **loss** function that satisfies:

- $L(t, y(\mathbf{x})) \geq 0$
- $L(t, y(\mathbf{x})) = 0 \iff t = y(\mathbf{x})$ (not necessarily in the other direction)
- $L(t, y(\mathbf{x}))$ does not increase when $|t - y(\mathbf{x})|$ decreases.

L is closely related to the distribution of the noise model ε_n .

Example: if we assume for example that $\varepsilon_n \sim \mathcal{N}(0, \sigma^2)$, using a maximum likelihood argument it can be shown that the *right* loss function is the **square error**:

$$L_{\text{SE}}(t, y(\mathbf{x})) := (t - y(\mathbf{x}))^2.$$

The **risk** is therefore

$$R(y) = \int_{\mathbb{R}} \int_{\mathbb{R}^d} (t - y(\mathbf{x}))^2 p(t|\mathbf{x}) p(\mathbf{x}) d\mathbf{x} dt$$

If we enjoy complete freedom to choose y , the solution is:

$$y^*(\mathbf{x}) = \int_{\mathbb{R}} t p(t|\mathbf{x}) dt = f(\mathbf{x}),$$

known as the **regression function**. Since $\mathbb{E}[\varepsilon_n] = 0$, we can alternatively express the regression setting by stating that t is a continuous random variable such that $f(\mathbf{x}) = \mathbb{E}[t | X = \mathbf{x}]$.

Claim: $y^*(\mathbf{x}) = f(\mathbf{x})$.

Proof:

$$\begin{aligned} y^*(\mathbf{x}) &= \mathbb{E}[t | X = \mathbf{x}] = \mathbb{E}[f(\mathbf{x}) + \varepsilon | X = \mathbf{x}] = \\ &= \mathbb{E}[f(\mathbf{x}) | X = \mathbf{x}] + \mathbb{E}[\varepsilon | X = \mathbf{x}] = f(\mathbf{x}) + 0 = \\ &= f(\mathbf{x}). \quad \square \end{aligned}$$

In a practical setting, we don't know $p(t|\mathbf{x})$. Instead, we have a finite i.i.d. **data sample** of N labeled observations $D = \{(\mathbf{x}_n, t_n)\}_{n=1,\dots,N}$, where $\mathbf{x}_n \in \mathbb{R}^d, t_n \in \mathbb{R}$. Then, **intuition** tells us to solve for y

$$\min_y \int_{\mathbb{R}^d} (f(\mathbf{x}) - y(\mathbf{x}))^2 p(\mathbf{x}) d\mathbf{x}.$$

This is equivalent to minimizing the risk function; we'll see this in the next part. For now, we must impose restrictions on the possible solutions y , this is, we must restrict the search space to a specific **class of functions** \mathcal{Y} .

We can compute an approximation to the true risk, called the **empirical risk**, by averaging the loss function on the available data D :

$$R_{\text{emp}}(y) := \frac{1}{N} \sum_{n=1}^N (t_n - y(\mathbf{x}_n))^2.$$

This quantity is also known as the **apparent error**. The **Empirical Risk Minimization (ERM)** principle states that a learning algorithm should choose a hypothesis (model) \hat{y} which minimizes the empirical risk among a predefined class of functions \mathcal{Y} :

$$\hat{y} := \operatorname{argmin}_{y \in \mathcal{Y}} R_{\text{emp}}(y).$$

The quantity $R_{\text{emp}}(\hat{y})$ is known as the **training error**. In theoretical ML, we are very much interested in:

- How this error fluctuates as a function of the data D .
- How far this error is from the true error, this is, to bound $|R_{\text{emp}}(\hat{y}) - R(y)|$; at the very least, to bound $|\mathbb{E}[R_{\text{emp}}(\hat{y})] - R(y)|$.
- How far this error is from the best possible error, this is, to bound $|R_{\text{emp}}(\hat{y}) - R(y^*)|$; at the very least, to bound $|\mathbb{E}[R_{\text{emp}}(\hat{y})] - R(y^*)|$.

Bias-Variance analysis

Recall the assumption that $\varepsilon_n \sim \mathcal{N}(0, \sigma^2)$. In this case, using the square error, the risk can be decomposed as:

$$\begin{aligned} R(y) &= \int_{\mathbb{R}} \int_{\mathbb{R}^d} (t - y(\mathbf{x}))^2 p(t, \mathbf{x}) d\mathbf{x} dt = \int_{\mathbb{R}} \int_{\mathbb{R}^d} (t - f(\mathbf{x}))^2 p(t, \mathbf{x}) d\mathbf{x} dt \\ &+ \int_{\mathbb{R}^d} (f(\mathbf{x}) - y(\mathbf{x}))^2 p(\mathbf{x}) d\mathbf{x} = \sigma^2 + \int_{\mathbb{R}^d} (f(\mathbf{x}) - y(\mathbf{x}))^2 p(\mathbf{x}) d\mathbf{x} = [\sigma^2 + \text{MSE}(y)], \end{aligned}$$

where f is the **regression function**. Therefore, we arrive at $R(y) = \sigma^2 + \text{MSE}(y)$. We can now forget about σ^2 and the risk and instead aim at minimizing the $\text{MSE}(y)$:

$$\text{MSE}(y) = \int_{\mathbb{R}^d} (f(\mathbf{x}) - y(\mathbf{x}))^2 p(\mathbf{x}) d\mathbf{x}.$$

A **learning algorithm** for **regression** is a procedure that, given data D and the search space \mathcal{Y} , outputs a model $y_D \in \mathcal{Y}$ that aims at minimizing $\text{MSE}(y)$.

Consider now one particular \mathbf{x}_0 ; different D will produce different y_D and therefore different predictions $y_D(\mathbf{x}_0)$. Let us concentrate on the quantity $(f(\mathbf{x}_0) - y(\mathbf{x}_0))^2$: we wish to eliminate the dependence on D . Therefore, we investigate its expected value, $\mathbb{E}[(f(\mathbf{x}_0) - y(\mathbf{x}_0))^2]$, taking over all possible D of size N . If we develop a little more their formulas,

$$\mathbb{E}[(f(\mathbf{x}_0) - y(\mathbf{x}_0))^2] = (f(\mathbf{x}_0) - \mathbb{E}[y_D(\mathbf{x}_0)])^2 + \mathbb{E}[(y_D(\mathbf{x}_0) - \mathbb{E}[y_D(\mathbf{x}_0)])^2].$$

We can interpret these summands as $f(\mathbf{x}_0) - \mathbb{E}[y_D(\mathbf{x}_0)] = \text{Bias}(y_D(\mathbf{x}_0))$, and $\mathbb{E}[(y_D(\mathbf{x}_0) - \mathbb{E}[y_D(\mathbf{x}_0)])^2] = \text{Var}(y_D(\mathbf{x}_0))$. Then, the formula is more clearly stated as

$$\text{MSE}(y_D(\mathbf{x}_0)) = \text{Bias}^2(y_D(\mathbf{x}_0)) + \text{Var}(y_D(\mathbf{x}_0)),$$

and the **risk** can be expressed as a sum of three summands:

$$R(y_D(\mathbf{x}_0)) = \sigma^2 + \text{Bias}^2(y_D(\mathbf{x}_0)) + \text{Var}(y_D(\mathbf{x}_0)).$$

The derivation above depends on a particular point \mathbf{x}_0 , so let us put it all back in place within their integrals:

$$\text{Bias}^2(y_D) = \int_{\mathbb{R}^d} \text{Bias}^2(y_D(\mathbf{x}))p(\mathbf{x}) d\mathbf{x},$$

$$\text{Var}(y_D) = \int_{\mathbb{R}^d} \text{Var}(y_D(\mathbf{x}))p(\mathbf{x}) d\mathbf{x},$$

$$R(y_D) = \sigma^2 + \text{Bias}^2(y_D) + \text{Var}(y_D).$$

In general, an **underfit** model will have a big bias, while an **overfit** model will have a high variance. The *ability to fit* has a name: it's called the **complexity** of the function class. Both models that are more or less complex than needed will tend to have large prediction errors. In the former, this will be dominated by the variance term, while in the latter, it will be dominated by the (square) bias term.

4. Regression theory and linear regression models (II)

Our departing statistical model still is

$$t_n = f(\mathbf{x}_n) + \varepsilon_n, \quad \mathbf{x}_n \in \mathbb{R}^d, \quad t \in \mathbb{R}$$

where ε_n is a continuous rv such that $\mathbb{E}[\varepsilon_n] = 0$ and $\text{Var}(\varepsilon_n) = \sigma^2$. Let's assume again that we further model $\varepsilon_n \sim \mathcal{N}(0, \sigma^2)$, and:

$$f(\mathbf{x}) \approx y(\mathbf{x}; \boldsymbol{\beta}) = \sum_{i=0}^d \beta_i x_i = \boldsymbol{\beta}^T \mathbf{x}$$

with $\mathbf{x} = (1, x_1, \dots, x_d)^T$ and $\boldsymbol{\beta} = (\beta_0, \beta_1, \dots, \beta_d)^T$. Suppose we have an iid sample of N labeled observations $D = \{(\mathbf{x}_n, t_n)\}_{n=1, \dots, N}$, where $\mathbf{x}_n \in \mathbb{R}^d, t_n \in \mathbb{R}$. Therefore, our statistical model is $t_n \sim \mathcal{N}(y(\mathbf{x}_n; \boldsymbol{\beta}), \sigma^2)$ or:

$$p(t_n | \mathbf{x}_n; \theta) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2} (t_n - \boldsymbol{\beta}^T \mathbf{x}_n)^2\right),$$

with unknown parameters $\theta := \{\beta_0, \beta_1, \dots, \beta_p, \sigma^2\}$. Put $\mathbf{t} = (t_1, \dots, t_N)^T$ and $X_{N \times (d+1)}$ the matrix of the \mathbf{x}_n . Define the **likelihood** as $\mathcal{L}(\theta) := P(\mathbf{t}|X; \theta)$. Let us maximize the log-likelihood:

$$\begin{aligned} l(\theta) &:= \log \mathcal{L}(\theta) = \log \prod_{n=1}^N p(t_n | \mathbf{x}_n; \theta) = \sum_{n=1}^N \log p(t_n | \mathbf{x}_n; \theta) = \\ &= -\frac{N}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} \sum_{n=1}^N (t_n - \boldsymbol{\beta}^T \mathbf{x}_n)^2 = \\ &= -\frac{N}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} (\mathbf{t} - X\boldsymbol{\beta})^T (\mathbf{t} - X\boldsymbol{\beta}) = \\ &= -\frac{N}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} \|\mathbf{t} - X\boldsymbol{\beta}\|^2. \end{aligned}$$

If we derive this wrt $\boldsymbol{\beta}$ and σ^2 , and set equal to zero, we get:

$$\begin{aligned} \frac{\partial l}{\partial \boldsymbol{\beta}} &= -\frac{1}{2\sigma^2} (-2X^T \mathbf{t} + 2X^T X \boldsymbol{\beta}) = 0 \\ \frac{\partial l}{\partial \sigma^2} &= -\frac{N}{2\sigma^2} + \frac{1}{2\sigma^4} (\mathbf{t} - X\boldsymbol{\beta})^T (\mathbf{t} - X\boldsymbol{\beta}) = 0. \end{aligned}$$

Therefore, we can calculate the estimates for both parameters:

$$\begin{aligned} \hat{\boldsymbol{\beta}} &= (X^T X)^{-1} X^T \mathbf{t}, \\ \hat{\sigma}^2 &= \frac{1}{N} (\mathbf{t} - X\hat{\boldsymbol{\beta}})^T (\mathbf{t} - X\hat{\boldsymbol{\beta}}) = \frac{1}{N} \|\mathbf{t} - X\hat{\boldsymbol{\beta}}\|^2. \end{aligned}$$

Note that $\hat{\sigma}^2 = R_{\text{emp}}(y_D)$, which is a **biased estimator** for σ^2 . An unbiased estimator is

$$\bar{\sigma}^2 = \frac{N}{N-d} \hat{\sigma}^2.$$

It's also known that $\hat{\boldsymbol{\beta}}$ is an unbiased estimator of $\boldsymbol{\beta}$ and that $\text{Var}(\hat{\boldsymbol{\beta}}) = (X^T X)^{-1} \sigma^2$. All of this implies that $\hat{\boldsymbol{\beta}} \sim \mathcal{N}(\boldsymbol{\beta}, (X^T X)^{-1} \sigma^2)$.

The matrix $X^+ = (X^T X)^{-1} X^T$ is known as the **Moore-Penrose pseudo-inverse** of X . It is the generalization of the notion of an inverse matrix to non-square matrices. It has the property that $X^+ X = I$, although in general $XX^+ \neq I$. However, both are symmetric.

Theorem. Let $X_{N \times M}$, with $N > M$. If the column vectors of X are linearly independent, i.e., if $\text{rank}(X) = M$, then:

1. The matrix $X^T X$ is symmetric and positive definite. In particular, it is non-singular.
2. The least squares problem

$$\min_{\beta \in \mathbb{R}^M} \|\mathbf{t} - X\beta\|^2,$$

has a unique solution.

3. This solution can be found solving the so-called Gauss' normal equations,

$$(X^T X) \beta = X^T \mathbf{t}$$

for β .

Quality of the fit

- In statistics, $-2l = -2 \log \mathcal{L}$ is called the **deviance**.
- In ML, this quality measure is the **square error**:

$$N \log(2\pi\sigma^2) + \frac{1}{\sigma^2} \|\mathbf{t} - X\hat{\beta}\|^2$$

- A much better quantity to report is the NRMSE,

$$\text{NRMSE}(\hat{\beta}) = \sqrt{\frac{\|\mathbf{t} - X\hat{\beta}\|^2}{(N-1)\text{Var}(\mathbf{t})}}.$$

In statistics, $R^2 = 1 - \text{NRMSE}^2$ is the proportion of the target variability *explained* by the model.

Leaping forward: basis functions

Recall that a model is **linear** if up to an invertible function its parameters play a linear role in the model. For example,

$$y(x; \beta) = \sum_{j=0}^d \beta_j x^j, \quad x \in \mathbb{R}$$

is a polynomial on x , but also a linear model on β .

A simple but powerful idea is the introduction of **basis functions**:

$$y(\mathbf{x}; \mathbf{w}) = \sum_{j=0}^M w_j \varphi_j(\mathbf{x}) = \mathbf{w}^T \varphi(\mathbf{x}),$$

where $\varphi_0(\mathbf{x}) = 1$, $\varphi(\mathbf{x}) = (1, \varphi_1(\mathbf{x}), \dots, \varphi_M(\mathbf{x}))^T$, $\mathbf{w} = (w_0, w_1, \dots, w_M)^T$. This is still a **linear model**. Define $\mathbf{t} = (t_1, \dots, t_N)^T$ as the vector of targets, and $\varphi_{N \times (M+1)}$ as the matrix of the $\varphi_{ij} = \varphi_j(\mathbf{x}_i)$, $i = 1, \dots, N$, $j = 1, \dots, M$:

$$\boldsymbol{\varphi} = \begin{pmatrix} 1 & \varphi_1(\mathbf{x}_1) & \varphi_2(\mathbf{x}_1) & \cdots & \varphi_M(\mathbf{x}_1) \\ 1 & \varphi_1(\mathbf{x}_2) & \varphi_2(\mathbf{x}_2) & \cdots & \varphi_M(\mathbf{x}_2) \\ \cdots & \cdots & \cdots & \ddots & \cdots \\ 1 & \varphi_1(\mathbf{x}_N) & \varphi_2(\mathbf{x}_N) & \cdots & \varphi_M(\mathbf{x}_N) \end{pmatrix}.$$

So, let us maximize the new log-likelihood: the Gauss' normal equations are

$$(\boldsymbol{\varphi}^T \boldsymbol{\varphi}) \mathbf{w} = \boldsymbol{\varphi}^T \mathbf{t}$$

and their solution is

$$\hat{\mathbf{w}} = (\boldsymbol{\varphi}^T \boldsymbol{\varphi})^{-1} \boldsymbol{\varphi}^T \mathbf{t} = \boldsymbol{\varphi}^+ \mathbf{t},$$

$$\hat{\sigma}^2 = \frac{1}{N} (\mathbf{t} - \boldsymbol{\varphi} \hat{\mathbf{w}})^T (\mathbf{t} - \boldsymbol{\varphi} \hat{\mathbf{w}}) = \frac{1}{N} \|\mathbf{t} - \boldsymbol{\varphi} \hat{\mathbf{w}}\|^2.$$

Singular Value Decomposition

The direct computation of the pseudo-inverse of $\boldsymbol{\varphi}$ has two major drawbacks:

- When M is large, $\boldsymbol{\varphi}^T \boldsymbol{\varphi}$ is a large $(M + 1) \times (M + 1)$ matrix; then, the computation of the required inverse $(\boldsymbol{\varphi}^T \boldsymbol{\varphi})^{-1}$ can be costly.
- If $\boldsymbol{\varphi}^T \boldsymbol{\varphi}$ is singular, or close to, then the required inverse $(\boldsymbol{\varphi}^T \boldsymbol{\varphi})^{-1}$ can be impossible, or numerically delicate.

Theorem. Every matrix $X_{N \times M}$ can be expressed as $X = U \Delta V^T$, with $U \in \mathcal{M}_N(\mathbb{R}), V \in \mathcal{M}_M(\mathbb{R})$, $\Delta \in \mathcal{M}_{N \times M}(\mathbb{R})$ diagonal. The columns of U are the eigenvectors of XX^T , and the columns of V are the eigenvectors of $X^T X$.

Let $\text{rank}(X) = r \leq \min(N, M)$. Then exactly r elements λ_k in the diagonal of Δ are strictly positive; the remaining elements are null. These $\lambda_k > 0$ are called the **singular values** and correspond to the square roots of the positive eigenvalues of XX^T (same as $X^T X$).

Sometimes an *economy size* decomposition is delivered: If X is $N \times M$ with $N > M$, then only the first M columns of U are given and Δ is $M \times M$.

SVD for least squares

Given the least squares problem

$$\min_{\mathbf{w} \in \mathbb{R}^M} \|\mathbf{t} - X\mathbf{w}\|^2,$$

the solution can be obtained with the SVD as:

- Compute the economy size SVD of $X = U \Delta V^T$.
- Solve for \mathbf{w} as $\hat{\mathbf{w}} = V \text{diag}(\lambda_k^{-1}) U^T \mathbf{t}$, where only the $\lambda_k > 0$ are considered.

Regularized least squares

The maximum likelihood framework can yield unstable parameter estimates, specially when

- the explanatory variables are highly correlated;
- there is an insufficient number of observations (N) relative to the number of predictors (basis functions $M + 1$ or dimensions $d + 1$).

In the context of regression with Gaussian noise (square error), it is quite common to penalize the parameter vector. Define the **penalized empirical error** as:

$$R_{\text{emp}}(y(\cdot; \mathbf{w})) := \|\mathbf{t} - \boldsymbol{\varphi} \mathbf{w}\|^2 + \lambda \|\mathbf{w}\|^2, \quad \lambda > 0.$$

If we set its derivative wrt w equal to zero

$$-2\varphi^T t + 2\varphi^T \varphi w + 2\lambda w = 0,$$

we solve for w and we get

$$\hat{w} = (\varphi^T \varphi + \lambda I)^{-1} \varphi^T t.$$

This is known as **Tikhonov** or L^2 **regularization** in ML. Perhaps it's best known as **ridge regression** in statistics, where it's usually explained as a "penalized log-likelihood". This can also be derived from Bayesian statistics arguments. Tikhonov regularization has some advantages:

- Pushing the length of the parameter vector $\|w\|$ to 0 allows the fit to be under explicit control with the regularization parameter λ .
- The matrix $\varphi^T \varphi$ is positive semi-definite; therefore $\varphi^T \varphi + \lambda I$ is guaranteed to be positive definite (hence non-singular), for all $\lambda > 0$.

Bayesian derivation:

$$\left. \begin{array}{l} D \text{ data} \\ \theta \text{ parameter vector} \end{array} \right\} \underset{\theta \in \Theta}{\operatorname{argmax}} P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)} \sim \underset{\theta \in \Theta}{\operatorname{argmax}} P(D|\theta)P(\theta) \sim \sim \underset{\theta \in \Theta}{\operatorname{argmax}} \ln(P(D|\theta)P(\theta)) \sim \underset{\theta \in \Theta}{\operatorname{argmax}} \{\ln P(D|\theta) + \ln P(\theta)\} \sim \sim \underset{\theta \in \Theta}{\operatorname{argmin}} \{-\ln P(D|\theta) - \ln P(\theta)\}$$

We change names of $\theta = w$, for it is fancier. If we assume that the $w \sim \mathcal{N}(0, \sigma^2 \text{Id})$, then $-\ln P(w) = \frac{\|w\|^2}{2\sigma_w^2}$ and then, observe that $-\ln \mathcal{L}$ is the mean square error,

$$\begin{aligned} \dots \underset{w \in W}{\operatorname{argmin}} \{-\ln \mathcal{L} - \ln P(w)\} &= \underset{w \in W}{\operatorname{argmin}} \left\{ \|t - \varphi w\|^2 + \frac{\|w\|^2}{2\sigma_w^2} + C(d, \sigma_w^2) \right\} \sim \\ &\sim \underset{w \in W}{\operatorname{argmin}} \{\|t - \varphi w\|^2 + \lambda \|w\|^2\}, \quad \lambda > 0. \end{aligned}$$

This is all nice, but how do we control the fit explicitly?

- Regularization allows the specification of models that are more complex than needed because it limits the effective complexity.
- Instead of trial-and-error on complexity, we can set a large complexitr and adjust the λ .

And how do we set the value of λ ? Using a technique called **Leaving-one-out cross validation (LOOCV)**, because

- In this case, λ is a very forgiving parameter; we usually perform a log search.
- There is a closed efficient formula for the LOOCV for **linear models**.

To get to the best model we can, we follow this steps:

1. Choose a (large) set of values Λ .
2. For every $\lambda \in \Lambda$,
 1. Solve for $\hat{w} = (\varphi^T \varphi + \lambda I)^{-1} \varphi^T t$.
 2. Compute the **hat matrix** $H := \varphi \varphi^+ \equiv \varphi (\varphi^T \varphi + \lambda I)^{-1} \varphi^T$.
 3. Compute the LOOCV of $y(\cdot) = \hat{w}^T \varphi(\cdot)$ in D as

$$\text{LOOCV}(y) = \frac{1}{N} \sum_{n=1}^N \left(\frac{t_n - \hat{w}^T \varphi(x_n)}{1 - h_{nn}} \right)^2.$$

3. Choose the model with the lowest LOOCV.

A very popular method is **Generalized Cross-Validation (GCV)**:

$$\text{GCV}(y) = \frac{1}{N} \frac{\sum_{n=1}^N (t_n - \hat{\mathbf{w}}^T \boldsymbol{\varphi}(x_n))^2}{\left(1 - \frac{\text{tr}(H)}{N}\right)^2},$$

which is a more stable computation for the LOOCV. Note that λ is needed to compute both $\hat{\mathbf{w}}$ and H .

LASSO Regression

The LASSO (**Least Absolute Shrinkage and Selection Operator**) regression is L^1 -regularized linear regression. The choice for the regularizer is $\|\mathbf{w}\|_1$ and we get:

$$R_{\text{emp}}(y(\cdot; \mathbf{w})) = \|\mathbf{t} - \boldsymbol{\varphi}\mathbf{w}\|^2 + \tau\|\mathbf{w}\|_1, \quad \tau > 0.$$

This turns out to be equivalent to

$$R_{\text{emp}}(y(\cdot; \mathbf{w})) = \|\mathbf{t} - \boldsymbol{\varphi}\mathbf{w}\|^2, \quad \text{subject to } \|\mathbf{w}\|_1 \leq \tau.$$

In ridge regression, as the penalty λ is increased, all coefficients are reduced while still remaining non-zero. In the LASSO regression, increasing the τ penalty causes more and more of the coefficients to be driven to zero. As the dimension d increases, the multidimensional L^1 -spheres have an increasing number of corners, and so it is highly likely that some coefficients will be set equal to zero. Hence, the LASSO regression model performs **shrinkage** and therefore, **feature selection**.

The LASSO loss function is no longer quadratic, but it is still convex. The **minimization problem** tied to LASSO regression is a special **quadratic programming (QP)** problem, for which the **Least Angle Regression (LARS)** procedure is used. It exploits the special structure of the problem, and provides an efficient way to compute the solutions for all possible values of $\tau > 0$ (the **regularization path**).

Conclusions

We have introduced **linear models** as linear combinations of non-linear **basis functions (BF)**:

ADVANTAGES:

- We can represent non-linear functions of the data using linear fitting techniques; we have the freedom to choose the form of the BFs.
- The fit can be under tight explicit control by regularization.
- The computations can be very efficient, no need to refit for LOOCV.
- Interpretability of the model is rather high.

LIMITATIONS: the most important weak point is the BFs.

- Many interesting BFs scale very poorly with dimension (polynomials, Fourier series, splines, ...)
- Our BFs are not flexible; they are data-independent.
- As a consequence, their number may be very high, which in turn leads to instability (because of low significance of the coefficients).

The solution is to **develop basis functions with parameters** such that:

- This BFs scale well with dimension (inner products, distances, ...)
- They are **data-dependent**, because of the parameters.
- As a consequence, their number might be much lower, and the coefficients will be significant.
- Unfortunately, the new parameters will play a **non-linear role** in the model: their optimization is plagued with local optima.

5. Classification theory and linear classification models (I). Bayesian decision theory.

Introduction: Bayes' formula

Discrete Random Variables. Let A be a discrete r.v. with probability mass function (pmf) P_A . We use the shorthand notation $P(a)$ to mean $P_A(A = a)$. Similarly, we write $P(b|a)$ to mean $P_{B|A}(B = b|A = a)$, etc, where

$$P(b|a) = \frac{P(b,a)}{P(a)}, \quad P(a) > 0.$$

Let $\mathcal{A} = \{a_1, \dots, a_n\}$, $\mathcal{B} = \{b_1, \dots, b_m\}$ be the possible values that A and B can take, respectively. Then, $\forall a \in \mathcal{A}$,

$$P(a) = \sum_{i=1}^m P(a, b_i) = \sum_{i=1}^m P(a|b_i)P(b_i).$$

Since $P(a,b) = P(b,a)$, it follows that, for any $a \in \mathcal{A}, b \in \mathcal{B}$

$$P(b|a) = \frac{P(a|b)P(b)}{\sum_{b' \in \mathcal{B}} P(a|b')P(b')}, \quad \text{with } \sum_{b' \in \mathcal{B}} P(b'|a) = 1.$$

Continuous Random Variables. Let X, Y two continuous r.v. with pdfs p_X, p_Y and joint density p_{XY} . We use the shorthand notation $p(x)$ to mean $p_X(X = x)$, etc.

$$p(x) = \int_{\mathbb{R}} p(x,y) dy; \quad p(y) = \int_{\mathbb{R}} p(x,y) dx.$$

Therefore,

$$p(y|x) = \frac{p(x|y)p(y)}{\int_{\mathbb{R}} p(x|y)p(y) dy}, \quad \text{with } \int_{\mathbb{R}} p(y|x) dy = 1.$$

Observation. Mixed random variables.

Suppose X is a continuous r.v. and Y is a discrete r.v. with values in $\{y_1, \dots, y_m\}$. In this case, $p(\cdot|y_i)$ is a continuous r.v. and $P(\cdot|x)$ is a discrete r.v. Moreover,

$$P(y_j|x) = \frac{p(x|y_j)p(y_j)}{\sum_{i=1}^m p(x|y_i)p(y_i)}, \quad \text{with } \sum_{j=1}^m P(y_j|x) = 1.$$

Decision rules

We are interested in determining the class or category of objects of nature according to Ω , a discrete r.v. with values $\{\omega_1, \omega_2\}$ that represent the two possible classes. The prior probabilities are $P(\omega_1), P(\omega_2)$. How should we classify objects?

Decision Rule 1. We don't measure any variable. We have no information other than "a new object comes".

If $P(\omega_1) > P(\omega_2)$ then class of object is ω_1 else class is ω_2 .

This rule classifies all objects into the same class; therefore, it will eventually classify an object into the wrong class. Thus, the **probability of error** of this rule is

$$P_e(\text{rule1}) = \min\{P(\omega_1), P(\omega_2)\}.$$

This rule is useful only if $P(\omega_1) << P(\omega_2)$ or if $P(\omega_2) << P(\omega_1)$. This is the optimum rule when no information is measured.

Discrete feature measuring. Suppose now that X is a discrete r.v. taking values in $\mathcal{X} = \{x_1, \dots, x_d\}$ that measures a **feature** of objects. Now, $P(\omega_i|x) = \frac{P(x|\omega_i)P(\omega_i)}{P(x)}$ is the **posterior** probability that an object with measured feature x belongs to class ω_i , $i \in \{1, 2\}$. Moreover, $P(x) = P(x|\omega_1)P(\omega_1) + P(x|\omega_2)P(\omega_2)$.

Upon observing x , the Bayes formula converts **prior** class probabilities $P(\omega_i)$ into **posterior** probabilities $P(\omega_i|x)$. How should we classify objects now?

Decision Rule 2. We now measure a feature x of the object coming forth.

If $P(\omega_1|x) > P(\omega_2|x)$ then class of object is ω_1 else class is ω_2 .

The probability of error for this rule is

$$P_e(\text{rule2}) = \sum_{i=1}^d \min\{P(\omega_1|x_i), P(\omega_2|x_i)\}P(x_i).$$

This rule is known as the **Bayes rule** or **classifier**.

Lemma. For all $a, b, c, d \in \mathbb{R}$, $\min(a, b) + \min(c, d) \leq \min(a+c, b+d)$.

Proposition. $P_e(\text{rule2}) \leq P_e(\text{rule1})$.

Proof:

$$\begin{aligned} & \sum_{i=1}^d \min\{P(\omega_1|x_i), P(\omega_2|x_i)\}P(x_i) = \\ &= \sum_{x \in \mathcal{X}} \min\{P(\omega_1|x)P(x), P(\omega_2|x)P(x)\} = \\ &= \sum_{x \in \mathcal{X}} \min\{P(x|\omega_1)P(\omega_1), P(x|\omega_2)P(\omega_2)\} \leq \\ &\leq \min \left\{ \sum_{x \in \mathcal{X}} P(x|\omega_1)P(\omega_1), \sum_{x \in \mathcal{X}} P(x|\omega_2)P(\omega_2) \right\} = \\ &= \min \left\{ P(\omega_1) \sum_{x \in \mathcal{X}} P(x|\omega_1), P(\omega_2) \sum_{x \in \mathcal{X}} P(x|\omega_2) \right\} = \\ &= \min \{P(\omega_1), P(\omega_2)\} \square \end{aligned}$$

Equality holds only if $P(x|\omega_1) = P(x|\omega_2)$, $\forall x \in \mathcal{X}$.

Continuous feature measuring. The next step is to consider a r.v. X with pdf $p(x)$ that measures a *continuous* feature of an object. Let \mathcal{P} be the support of p , i.e. $\mathcal{P} = \{x \in \mathbb{R} | p(x) > 0\}$. In this setting, $p(x|\omega_i)$, $i \in \{1, 2\}$ are the conditional densities of x for every class.

Proposition. $P_e(\text{rule2}) \leq P_e(\text{rule1})$.

Proof:

$$\begin{aligned} \int_{\mathcal{P}} \min\{P(\omega_1|x), P(\omega_2|x)\} p(x) dx &= \\ \int_{\mathcal{P}} \min\{P(\omega_1|x)p(x), P(\omega_2|x)p(x)\} dx &= \\ \int_{\mathcal{P}} \min\{p(x|\omega_1)P(\omega_1), p(x|\omega_2)P(\omega_2)\} dx &\leq \\ \min \left\{ \int_{\mathcal{P}} p(x|\omega_1)P(\omega_1) dx, \int_{\mathcal{P}} p(x|\omega_2)P(\omega_2) dx \right\} &= \\ \min \left\{ P(\omega_1) \int_{\mathcal{P}} p(x|\omega_1) dx, P(\omega_2) \int_{\mathcal{P}} p(x|\omega_2) dx \right\} &= \\ \min \{P(\omega_1), P(\omega_2)\} \square \end{aligned}$$

Equality holds only if $p(\cdot|\omega_1) = p(\cdot|\omega_2)$.

The Bayes classifier

The Bayes classifier can be extended in two ways:

1. Consider a vector $X = (X_1, \dots, X_d)^T$ of continuous r.v. with pdf $p(\mathbf{x}) = p(x_1, \dots, x_d)$ that measures d continuous features.
2. Consider a finite set of classes Ω , a discrete r.v. with values $\omega_1, \dots, \omega_K$, that represent the possible classes ($K \geq 2$).

Therefore, we have new probabilities $p(\mathbf{x}|\omega_i)$, $P(\omega_i|\mathbf{x})$, $1 \leq i \leq K$. The new Bayes rule says:

Decision rule.

The class $\hat{\omega}(\mathbf{x})$ of object \mathbf{x} is ω_k when $k = \operatorname{argmax}_{i=1,\dots,K} P(\omega_i|\mathbf{x})$.

The sets $\mathcal{R}_k = \{\mathbf{x} | \hat{\omega}(\mathbf{x}) = k\}$ are called **regions**, and depend on the specific classifier. It is worth noting they form a partition of the total space, which is in general thought of as \mathbb{R} or, in the vector setting, \mathbb{R}^d .

We now want to see that the Bayes classifier is **optimal** in terms of probability of error. To do this, let us assume a classifier with regions $\mathcal{R}_1, \mathcal{R}_2$. Then,

$$\begin{aligned} P_e &= P(\mathbf{x} \in \mathcal{R}_2, \omega_1) + P(\mathbf{x} \in \mathcal{R}_1, \omega_2) = \\ &= P(\mathbf{x} \in \mathcal{R}_2|\omega_1)P(\omega_1) + P(\mathbf{x} \in \mathcal{R}_1|\omega_2)P(\omega_2) = \\ &= \int_{\mathcal{R}_2} p(\mathbf{x}|\omega_1)P(\omega_1) d\mathbf{x} + \int_{\mathcal{R}_1} p(\mathbf{x}|\omega_2)P(\omega_2) d\mathbf{x} \geq \\ &= \int_{\mathcal{P}} \min \{p(\mathbf{x}|\omega_1)P(\omega_1), p(\mathbf{x}|\omega_2)P(\omega_2)\} d\mathbf{x} = \\ &= P_e(\text{Bayes}) \square \end{aligned}$$

So, if any other classifier has a smaller error, the Bayes classifier is optimal.

The Bayes classifier can also have a **rejection class** (illustrated here for two classes); if we fix $\varepsilon \in (0, 1)$,

```

if  $P(\omega_1|\mathbf{x}) - P(\omega_2|\mathbf{x}) > \varepsilon$  then class of object  $\mathbf{x}$  is  $\omega_1$ 
else if  $P(\omega_1|\mathbf{x}) - P(\omega_2|\mathbf{x}) < \varepsilon$  then class of object  $\mathbf{x}$  is  $\omega_2$ 
else do not classify.
```

For every feature vector \mathbf{x} we take one of three possible **actions**.

Consider a finite set of actions $A = \{a_1, \dots, a_m\}$. For each $a_i \in A$, denote by $l(a_i|\omega_j)$ the **loss** for choosing a_i when \mathbf{x} is known to be in ω_j . This is a simplified setting in which this loss does not depend on \mathbf{x} .

Example: Let $m = K + 1$ and let a_i stand for "classify \mathbf{x} into class ω_i " for $1 \leq i \leq K$; let a_{K+1} stand for "do not classify \mathbf{x} ". A possible set of losses is:

$$l(a_i|\omega_j) = \begin{cases} 1, & \text{for } 1 \leq i, j \leq K, i \neq j; \\ 0, & \text{for } 1 \leq i \leq K, i = j; \\ \frac{1}{2}, & \text{for } i = K + 1, 1 \leq j \leq K. \end{cases}$$

This example suggests that a decision not to classify is less costly than a misclassification.

The notion of risk

Definition. *Conditional risk.* For a given feature vector \mathbf{x} , define the conditional risk of an action as:

$$r(a_i|\mathbf{x}) := \sum_{j=1}^K l(a_i|\omega_j)P(\omega_j|\mathbf{x}).$$

Definition. *Decision rule, Total risk.* A decision rule is any function $a : \mathcal{P} \rightarrow A$ from the support of the probability density function \mathcal{P} to the action set A that assigns an action $a(\mathbf{x})$ to every \mathbf{x} such that $p(\mathbf{x}) > 0$. The total risk of a decision rule is

$$R(a) := \mathbb{E}_{\mathbf{x}}[r(a(\mathbf{x})|\mathbf{x})] = \int_{\mathcal{P}} r(a(\mathbf{x})|\mathbf{x})p(\mathbf{x}) d\mathbf{x}.$$

We are interested in the decision rule that minimizes the total risk. Consider the rule

$$\hat{a}(\mathbf{x}) = \operatorname{argmin}_{1 \leq j \leq m} r(a_j|\mathbf{x}).$$

You may recognize it as the Bayes rule. Given that this rule minimizes the argument of the integral for every possible \mathbf{x} , it follows that the Bayes rule has the lowest possible risk. The value $R(\hat{a})$ is called the **Bayes risk** and is the minimum risk possible in global terms.

0/1 losses

In many applications the 0/1 loss is used, usually in absence of more precise information:

$$l_{ij} = \begin{cases} 0, & \text{if } i = j; \\ 1, & \text{if } i \neq j. \end{cases}$$

Consider K classes, and actions $a_i : \text{classify } \mathbf{x} \text{ into } \omega_i$. Then, the conditional risk for each action is

$$r(a_i|\mathbf{x}) = \sum_{j=1}^K l_{ij}P(\omega_j|\mathbf{x}) = \sum_{j=1, i \neq j}^K P(\omega_j|\mathbf{x}) = 1 - P(\omega_i|\mathbf{x}).$$

Discriminant functions

Functions of the form $g_k : \mathcal{P} \rightarrow \mathbb{R}$ are a useful tool to build an abstract classifier. An object \mathbf{x} is assigned to class ω_i when $g_i(\mathbf{x})$ is the highest among the values $g_1(\mathbf{x}), \dots, g_K(\mathbf{x})$. For example,

- $g_k(\mathbf{x}) = P(\omega_k | \mathbf{x})$.
- $g_k(\mathbf{x}) = P(\omega_k) p(\mathbf{x} | \omega_k)$.
- $g_k(\mathbf{x}) = -r(a_k | \mathbf{x})$.

If g_k is a discriminant function, then so is $h \circ g_k$, for any strictly monotonic function h . For two classes, we can use a single discriminant function, called a **dichotomizer**:

1. Define $g(\mathbf{x}) := g_1(\mathbf{x}) - g_2(\mathbf{x})$.
2. Assign \mathbf{x} to class ω_1 if $g(\mathbf{x}) > 0$ and to class ω_2 if $g(\mathbf{x}) < 0$.

The Gaussian Distribution

A normally distributed d -variate random vector $X = (X_1, \dots, X_d)^T$ has a pdf like

$$p(\mathbf{x}) = \frac{1}{(2\pi)^{\frac{d}{2}} \|\Sigma\|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^T \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu})\right),$$

where $\boldsymbol{\mu}$ is the **mean vector** and $\Sigma_{d \times d} = (\sigma_{ij}^2)$ is the real symmetric and positive definite **covariance matrix**.

- $\mathbb{E}[X] = \boldsymbol{\mu}$ and $\mathbb{E}[(X - \boldsymbol{\mu})(X - \boldsymbol{\mu})^T] = \Sigma$.
- $\text{Cov}[X_i, X_j] = \sigma_{ij}^2$ and $\text{Var}[X_i] = \sigma_{ii}^2$.

As $X \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$, then X_i, X_j are statistically independent $\iff \text{Cov}[X_i, X_j] = 0$. In general, only \implies holds.

The quantity $d(\mathbf{x}, \boldsymbol{\mu}) = \sqrt{(\mathbf{x} - \boldsymbol{\mu})^T \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu})}$ is called the **Mahalanobis distance**. What is behind the choice of a multivariate Gaussian distribution for a class?

- We want to have a **prototype object**, which is modeled by the mean vector.
- Also, the **noise modeling** is easy because in a multivariate Gaussian, this is modeled by the covariance matrix.
- Even though, it is very important to take into account that the number of parameters of the multivariate Gaussian is $\frac{d(d+1)}{2} + d$, for dimension d .

The surfaces of equal probability, $d(\mathbf{x}, \boldsymbol{\mu}) = \text{const}$ are **hyperellipsoids**. The principal directions or components (PC) of the hyperellipsoids are given by the eigenvectors u_i of Σ , which satisfy $\Sigma u_i = \lambda_i u_i$, for $\lambda_i > 0$. The lengths of the hyperellipsoids along these axes are proportional to $\sqrt{\lambda_i}$, the **singular values** associated with u_i . Note that as Σ is positive definite, all $\lambda_i > 0$.

Properties

- **Simplified forms:** if the X_i are statistically independent, then $p(\mathbf{x}) = \prod_{i=1}^d p(x_i)$, Σ is diagonal and the PCs are axis-aligned.
- **Analytical properties**, for example, any moment $\mathbb{E}[X^p]$ can be expressed as a function of $\boldsymbol{\mu}$ and Σ .
- **Central limit theorem**, the mean of d i.i.d. random variables tends to a normal distribution, as $d \rightarrow \infty$.
- **Linear transformation invariance**, the distribution of a linear transformation of the coordinate system remains normal.

6. Classification theory and linear classification models (II).

Generative Bayesian classifiers

Discriminant functions for the Gaussian density

We showed that the Bayes rule minimizing the probability of error could be formulated in terms of a family of **discriminant functions**:

assign the feature vector \mathbf{x} to class ω_k whenever $g_k(\mathbf{x})$ is the largest, $1 \leq k \leq K$.

When $X_{|\Omega=\omega_k} \sim \mathcal{N}(\boldsymbol{\mu}_k, \Sigma_k)$, this family can be reduced to very simple expressions. Using Bayes rule and the natural logarithm, the discriminant function for class ω_k becomes

$$g_k(\mathbf{x}) = \ln[P(\omega_k) p(\mathbf{x}|\omega_k)] = \ln P(\omega_k) - \ln[(2\pi)^{\frac{d}{2}} \|\Sigma_k\|^{\frac{1}{2}}] - \frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_k)^T \Sigma_k^{-1} (\mathbf{x} - \boldsymbol{\mu}_k).$$

Erasing constant terms,

$$g_k(\mathbf{x}) = \ln P(\omega_k) - \frac{1}{2}(\ln \|\Sigma_k\| - (\mathbf{x} - \boldsymbol{\mu}_k)^T \Sigma_k^{-1} (\mathbf{x} - \boldsymbol{\mu}_k)).$$

This expression is called a **quadratic discriminant function**, and the **decision boundaries** $g_i(\mathbf{x}) = g_j(\mathbf{x})$ are general hyper-quadratics in d -dimensional space. The resulting classifier out of the class of discriminant functions together with the decision rule is called **Quadratic Discriminant Analysis (QDA)**.

- If we assume that **all class-conditional distributions $p(\mathbf{x}|\omega_k)$ have the same covariance matrix Σ** , after removing all terms that do not depend on k or \mathbf{x} we get:

$$g_k(\mathbf{x}) = \ln P(\omega_k) + \boldsymbol{\mu}_k^T \Sigma^{-1} \mathbf{x} - \frac{1}{2} \boldsymbol{\mu}_k^T \Sigma^{-1} \boldsymbol{\mu}_k.$$

Reorganizing terms we obtain $g_k(\mathbf{x}) = \mathbf{w}_k^T \mathbf{x} + w_{k0}$, where

$$\begin{aligned} \mathbf{w}_k &= \Sigma^{-1} \boldsymbol{\mu}_k, \\ w_{k0} &= -\frac{1}{2} \boldsymbol{\mu}_k^T \Sigma^{-1} \boldsymbol{\mu}_k + \ln P(\omega_k). \end{aligned}$$

This is because Σ^{-1} is a symmetric matrix for being the inverse of a symmetric matrix. These are **linear discriminant functions** (linear in \mathbf{x}) and the **decision boundaries** are hyperplanes if d -dimensional space.

- If we further assume that **all X_i, X_j pairs are statistically independent**, that is, Σ is a diagonal matrix, we get:

$$g_k(\mathbf{x}) = \ln P(\omega_k) - \frac{1}{2} \sum_{i=1}^d \frac{(\mu_{ki} - x_i)^2}{\sigma_i^2}.$$

- If we further assume that **all X_i have the same variance σ^2** (for example, standardizing all variables), that is $\Sigma = \sigma^2 I_d$, we get:

$$g_k(\mathbf{x}) = \ln P(\omega_k) - \frac{1}{2\sigma^2} \|\boldsymbol{\mu}_k - \mathbf{x}\|^2.$$

- If we further assume that **all classes have the same prior distribution**, $P(\omega_k) = \frac{1}{K}$, we get:

$$g_k(\mathbf{x}) = -\|\boldsymbol{\mu}_k - \mathbf{x}\|^2.$$

In all cases, we have a **minimum-distance** classifier in \mathbb{R}^d :

- In the general case (some covariance matrices are different), the classifier uses a different Mahalanobis distance (a fully-weighted Euclidean distance) from \mathbf{x} to each class. This is called **Quadratic Discriminant Analysis (QDA)**.
- In case all covariance matrices are equal, the classifier uses the same Mahalanobis distance from \mathbf{x} to all classes. This is called **Linear Discriminant Analysis (LDA)**.
- In case all covariance matrices are diagonal, the classifier uses a simply-weighted Euclidean distance from \mathbf{x} to all classes.
- In case all covarianve matrices are a multiple of the identity matrix, the classifier uses an unweighted Euclidean distance from \mathbf{x} to all classes.

A numerical example

Derive a linear discriminant function for the two-class classification problem defined by the following Gaussian class-conditional densities:

$$\boldsymbol{\mu}_1 = (0, 0, 0)^T, \boldsymbol{\mu}_2 = (1, 1, 1)^T, \Sigma_1 = \Sigma_2 = \text{diag} \left(\frac{1}{4}, \frac{1}{4}, \frac{1}{4} \right), P(\omega_2) = 2P(\omega_1).$$

Solution: since all the X_i, X_j are statistically independent ($i \neq j$) and have the same variance $\sigma^2 = \frac{1}{4}$, that is $\Sigma = \frac{1}{4}I$, we get:

$$g_1(\mathbf{x}) = \ln P(\omega_1) - \frac{1}{2} \frac{\|\boldsymbol{\mu}_1 - \mathbf{x}\|^2}{\sigma^2} = \ln \frac{1}{3} - \frac{1}{2} \frac{\|(0, 0, 0) - \mathbf{x}\|^2}{\frac{1}{4}},$$

$$g_2(\mathbf{x}) = \ln P(\omega_2) - \frac{1}{2} \frac{\|\boldsymbol{\mu}_2 - \mathbf{x}\|^2}{\sigma^2} = \ln \frac{2}{3} - \frac{1}{2} \frac{\|(1, 1, 1) - \mathbf{x}\|^2}{\frac{1}{4}}.$$

Then we can build an optimal **dichotomizer**:

$$g(\mathbf{x}) = g_1(\mathbf{x}) - g_2(\mathbf{x}), \quad \mathbf{x} = (x_1, x_2, x_3)^T$$

and the decision rule is

assign \mathbf{x} to ω_1 when $g(\mathbf{x}) > 0$, and to ω_2 when $g(\mathbf{x}) < 0$.

Substituting, $g(\mathbf{x}) = -\ln 2 - 2\|\mathbf{x}\|^2 + 2\|(1, 1, 1) - \mathbf{x}\|^2$, which results in

$$(x_1 + x_2 + x_3) > ? \frac{3}{2} - \frac{1}{4}\ln 2 \approx 1.32.$$

The **prediction** for the test example $\mathbf{x}^* = (0.1, 0.7, 0.8)^T$ is $\mathbf{x}^* \in \omega_2$, given that $0.1 + 0.7 + 0.8 > 1.32$.

Computations in practice

In practical situations, only an i.i.d. data sample S is available. Let $S_k \subset S$ be the subset of observations known to belong to class ω_k . Then S_1, \dots, S_K is a partition of S . We can use **unbiased estimates** for the vector means and for the class priors:

$$\hat{\boldsymbol{\mu}}_k = \frac{1}{|S_k|} \sum_{\mathbf{x} \in S_k} \mathbf{x}, \quad \hat{P}(\omega_k) = \frac{|S_k|}{|S|}.$$

If we know (or assume) that covariance matrices are **different** (we wish to use QDA):

$$\hat{\Sigma}_k = \frac{1}{|S_k| - 1} \sum_{\mathbf{x} \in S_k} (\mathbf{x} - \hat{\boldsymbol{\mu}}_k)(\mathbf{x} - \hat{\boldsymbol{\mu}}_k)^T.$$

Otherwise, if we know (or assume) that covariance matrices are **equal** (we wish to use LDA):

$$\hat{\Sigma}_{\text{pooled}} = \frac{1}{|S_k| - K} \sum_{k=1}^K (|S_k| - 1) \hat{\Sigma}_k.$$

Discussion

Bayesian classifiers are optimal when the class-conditional densities and priors are known; the methods are well-principled, fast and reliable. For Gaussian classes, we get a quadratic classifier – QDA (if all covariance matrices are equal, a linear classifier, LDA); using a specific distance function corresponds to certain statistical assumptions:

- If the class-conditional densities are far from the assumptions, the model will be poor.
- Even if the class-conditional densities are Gaussian, the parameters should be reliably estimated (particularly for QDA).
- Once we use sample statistics instead of population parameters, we lose optimality.

The question whether these assumptions hold can rarely be answered in practice; in most cases we are limited to posing and answering the question "*does this classifier give satisfactory predictions or not?*".

Regularized Discriminant Analysis

If $d > |S_k|$ for some k , QDA cannot be applied, because the class covariance matrix $\hat{\Sigma}_k$ is singular. If $d > N$, neither QDA nor LDA can be used, because both $\hat{\Sigma}_k$ and $\hat{\Sigma}_{\text{pooled}}$ are singular. These problems can be overcome by applying **regularization**:

$$\hat{\Sigma}_k(\lambda, \gamma) := (1 - \gamma) [(1 - \lambda) \hat{\Sigma}_k + \lambda \hat{\Sigma}_{\text{pooled}}] + \frac{\gamma}{d} \text{tr}[\hat{\Sigma}_k(\lambda)] I_d,$$

where $\hat{\Sigma}_k(\lambda) = (1 - \lambda) \hat{\Sigma}_k + \lambda \hat{\Sigma}_{\text{pooled}}$. LDA is $(\lambda, \gamma) = (1, 0)$ and QDA is $(\lambda, \gamma) = (0, 0)$.

Pros & cons

Pros:

- Adaptable to all class-conditional distributions (not only Gaussian), even with mixed variables.
- Very resistant to overfitting the data sample.
- Accepts class priors and losses for misclassifications.

Cons:

- Assumption of Gaussianity may be far from true.
- Needs sufficient examples per class if we wish to use QDA.
- Requires matrix inversions (costly or numerically delicate).

The Naive-Bayes classifier

We showed that the 0/1 loss Bayes rule minimizing the probability of error could be formulated in terms of discriminant functions $g_k(\mathbf{x}) = P(\omega_k)P(\mathbf{x}|\omega_k)$, for $k = 1, \dots, K$. We can expand the conditional probability to be

$$\begin{aligned} P(\omega_k)P(\mathbf{x}|\omega_k) &= P(\omega_k)P(X_1 = x_1 \wedge X_2 = x_2 \wedge \dots \wedge X_d = x_d | \omega_k) = \\ &= P(\omega_k)P(X_1 = x_1 | \omega_k) \prod_{j=2}^d P(X_j = x_j | \omega_k, X_1 = x_1 \wedge \dots \wedge X_{j-1} = x_{j-1}), \end{aligned}$$

and if we assume X_1, \dots, X_d are pairwise independent *given the class*, this is equal to

$$\dots = P(\omega_k)P(X_1 = x_1|\omega_k) \prod_{j=2}^d P(X_j = x_j|\omega_k) = P(\omega_k) \prod_{j=1}^d P(X_j = x_j|\omega_k) \stackrel{\text{def}}{=} [\text{NB}_k(\mathbf{x})].$$

Extensions

Multiplying numbers smaller than 1, we can easily get numeric errors in a computer, so we want to ease the computations. To do this, we can numerically extend (and ease) the classifier by taking logarithms as such

$$\text{NB}_k(\mathbf{x}) = \ln P(\omega_k) + \sum_{j=1}^d \ln P(X_j = x_j|\omega_k)$$

How do we deal with continuous variables? We have two options:

1. Assume a particular pdf for the variable and estimate its parameters from the data.
2. Discretize the variable and treat it as discrete.

Null empirical probabilities

In test examples \mathbf{x}^* it may happen that some variable X_j has a value x_j^* **not present in the sample** used to create the classifier, hence $\hat{P}(X_j = x_j^*|\omega_k) = 0$ and we are in trouble. One way to avoid this kind of situations is to use the **Laplace correction**:

$$\hat{P}_L(X_j = x_j^*|\omega_k) = \frac{\text{card}(\{\mathbf{x} \in S_k : X_j = x_j\}) + p}{\text{card}(S_k) + p}, \quad p \in \mathbb{N}.$$

Here p is the "weight" assigned to the prior probability and V_k is the number of different values of variable k .

The kNN classifier

Definition. Metric. Let X be a set. A metric in X is a two-parameter function $d : X \times X \rightarrow \mathbb{R}^+ \cup \{0\}$ satisfying the following properties $\forall x, y, z \in X$:

1. $d(x, y) = 0 \iff x = y$
2. $d(x, y) = d(y, x)$
3. $d(x, y) \leq d(x, z) + d(z, y)$

A pair (X, d) is a metric space.

Decision rule. *The 1NN technique.* The 1NN technique classifies any $x \in X$ in the same class of the "nearest neighbour" of x in S , that is

The class of x is the class of $\underset{x' \in S \setminus \{x\}}{\operatorname{argmin}} d(x, x')$.

Decision rule. *The kNN technique.* The kNN technique considers the $k \geq 1$ "nearest neighbours" of x in S and votes for the most represented class:

The class of x is the majority class among its k closest elements in S .

In this rule, ties may happen. These are broken randomly. There are no ties for two-class problems and odd k .

Theorem (Cover & Hart, 1965). *Asymptotic analysis of the 1NN technique.* Call $\varepsilon_{1\text{NN}}$ the probability of error of 1NN and ε_B be the Bayes error. Then, as $N \rightarrow \infty$,

$$\varepsilon_B \leq \varepsilon_{1\text{NN}} \leq \varepsilon_B \left(2 - \varepsilon_B \frac{K}{K-1} \right) \leq 2\varepsilon_B.$$

In particular, for two-class problems,

$$\varepsilon_B \leq \varepsilon_{1\text{NN}} \leq 2\varepsilon_B(1 - \varepsilon_B).$$

7. Classification theory and linear classification models (III).

Discriminative classifiers

Maximum Likelihood (ML) framework (I)

Generalized Linear Methods

Generalized linear models are a very general and classical technique for fitting **linear models**. They are the genuine representatives of **discriminative methods**. The main difference between discriminative and generative methods is that using generative methods, we can generate new data from the parameter estimation we have previously done, while in discriminative methods we neither can't or are interested in doing so.

These methods work for many **target types**:

- Binary (two-class) and nominal (multi-class).
- Proportions and counts.
- Ordinal (ordered classes).
- Continuous target variables.

They admit very **general predictors**. Categorical variables are binarized.

Definition. *Generalized linear model.* A GLM is a linear predictor of a convenient function h of the conditional expected value of the target variable. Mathematically,

$$h(\mathbb{E}[T_n | X_n]) = \beta^T X_n + \beta_0.$$

This function h is typically smooth (of class C^k for some $k \geq 2$) and globally invertible. It's called the **link function**. We optimize the β and β_0 parameters directly, without any assumptions of the distribution of the x . The target variable rows T_n are taken as independent and drawn from a distribution of the **exponential family**: Poisson, Gaussian, Bernoulli, Gamma, ...

The modeler chooses a suitable distribution for T_n given X_n :

1. If we are predicting a real continuous value, we will usually use the Gaussian distribution; hence, h is the identity function. This is called **linear regression**.
2. If we are in a two-class problem, we will use the Bernoulli distribution, with link function $h = \text{logit}$. This method is called **logistic regression**.
3. If we want to predict a counter, we will usually use the Poisson distribution, with link function $h = \ln$. This is called **Poisson regression**.

This generality comes at a cost: in general we need an iterative procedure for the optimization of the β and β_0 parameters. A popular procedure is to set it up as a Maximum Likelihood problem and use a preferred numerical optimization method (e.g. Newton-Raphson).

Logistic regression

We are in the case of binary classification ($K = 2$). We **model** the posterior probability for class ω_1 as:

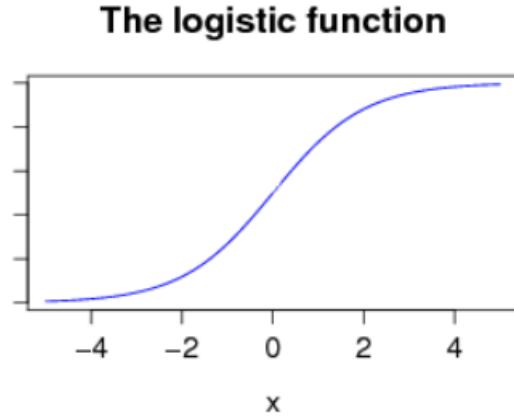
$$P(\omega_1 | \mathbf{x}) = g(\beta^T \mathbf{x} + \beta_0),$$

where $g(z) = \frac{\exp(z)}{1+\exp(z)} = \frac{1}{1+\exp(-z)}$ is the **logistic function**. Obviously,
 $P(\omega_2 | \mathbf{x}) = 1 - P(\omega_1 | \mathbf{x}) = 1 - g(\beta^T \mathbf{x} + \beta_0)$.

The logistic function is a \mathcal{C}^∞ function $g : \mathbb{R} \rightarrow (0, 1)$. This function is a bijection, with inverse

$$g^{-1}(z) = \ln\left(\frac{z}{1-z}\right) \stackrel{\text{def}}{=} \text{logit}(z)$$

for $z \in (0, 1)$. This is called the **logit function**, and looks like this:



Each $T_n \sim B(p_n)$, where $p_n = g(\beta^T X_n + \beta_0)$,

$$\implies P(T_n | X_n, \beta) = \begin{cases} p_n, & T_n = 1 (X_n \in \omega_1) \\ 1 - p_n, & T_n = 0 (X_n \in \omega_2) \end{cases} = p_n^{T_n} (1 - p_n)^{(1-T_n)}.$$

Notice that $g(\beta^T X_n + \beta_0) = \mathbb{E}[T_n] = p_n$. Hence, we are identifying p_n with $P(\omega_1 | X_n)$.

Interpretation of the model

The main thing to remember about logistic regression is that "The log of the odds is a linear function of the predictors". Since $P(\omega_1 | X) = g(\beta^T X + \beta_0)$, we have

$$\ln\left(\frac{P(\omega_1 | X)}{P(\omega_2 | X)}\right) = \ln\left(\frac{P(\omega_1 | X)}{1 - P(\omega_1 | X)}\right) = \text{logit}(P(\omega_1 | X)) = \beta^T X + \beta_0.$$

Parameter obtaining

Suppose we have an i.i.d. sample of N labeled observations $S = \{(\mathbf{x}_n, t_n)\}_{n=1,\dots,N}$, where $\mathbf{x}_n \in \mathbb{R}^n$, $t_n \in \{0, 1\}$.

The first thing we notice is that we have $d + 1$ parameters to fit. In the "equivalent" case of generative methods (LDA), we had $\frac{d(d+1)}{2} + 2d$ parameters. So, this cases scales better (linearly) than LDA did (quadratically). We re-write $P(\omega_1 | X) = g(\beta^T X)$, with

$$X = (1, X_1, \dots, X_d)^T, \quad \beta = (\beta_0, \beta_1, \dots, \beta_d)^T.$$

Now we see the obtaining of the parameters from a Maximum Likelihood perspective:

$$\begin{aligned} l(\beta) &= \ln \mathcal{L}(\beta) = \ln P(\{t_n\}_{n=1}^N | \{\mathbf{x}_n\}_{n=1}^N, \beta) = \\ &\ln \prod_{n=1}^N P(t_n | \mathbf{x}_n, \beta) = \sum_{n=1}^N \ln P(t_n | \mathbf{x}_n, \beta) = \\ &\sum_{n=1}^N \ln \left(g(\beta^T \mathbf{x}_n)^{t_n} (1 - g(\beta^T \mathbf{x}_n))^{(1-t_n)} \right) = \\ &\sum_{n=1}^N \ln \left((p_n)^{t_n} (1 - p_n)^{(1-t_n)} \right), \quad p_n = g(\beta^T \mathbf{x}_n). \end{aligned}$$

Now,

$$(p_n)^{t_n} (1 - p_n)^{(1-t_n)} = \left(\frac{p_n}{1 - p_n} \right)^{t_n} (1 - p_n) = \\ (\exp(\beta^T \mathbf{x}_n))^{t_n} (1 + \exp(\beta^T \mathbf{x}_n))^{-1}.$$

Therefore,

$$l(\beta) = \sum_{n=1}^N [t_n \beta^T \mathbf{x}_n - \ln(1 + \exp(\beta^T \mathbf{x}_n))]$$

When we set the derivative of this expression equal to zero, we are not able to isolate the parameter vector β . Hence, we need an iterative method, such as Newton-Raphson.

Newton-Raphson

As the Maximum Likelihood of the logistic regression does not have a closed-form solution, we iterate over k to obtain a good enough approximation to the parameter vector β :

$$\begin{aligned} \beta^{k+1} &= \beta^k - \left(\frac{\partial^2 l}{\partial \beta \partial \beta^T} \right)^{-T} \left(\frac{\partial l}{\partial \beta} \right) = \\ &\beta^k + (X^T W X)^{-1} X^T (\mathbf{t} - \mathbf{p}) = \\ &(X^T W X)^{-1} X^T W z, \\ &z = X \beta^T + W^{-1} (\mathbf{t} - \mathbf{p}) \end{aligned}$$

All of this because

$$\frac{\partial l}{\partial \beta} = X^T (\mathbf{t} - \mathbf{p}), \quad \frac{\partial^2 l}{\partial \beta \partial \beta^T} = -X^T W X.$$

Here, X is the data matrix of the $\{\mathbf{x}_n\}$, W is a diagonal matrix such that $w_{ii} = p_i(1 - p_i)$, for $i = 1, \dots, N$, and $\mathbf{t} = (t_1, \dots, t_N)^T$, $\mathbf{p} = (p_1, \dots, p_N)^T$. When all of this is put into place, we are left with the following algorithm, called **Iterated Reweighted Least Squares (IRLS)**:

1. Initialize $\beta_0 \leftarrow \ln \left(\frac{P(\omega_1)}{1-P(\omega_1)} \right)$ and $\beta_i \leftarrow 0$, for $i = 1, \dots, d$. (This is the *null model*)
2. Iterate the following until convergence:
 1. Update $\mathbf{p} \leftarrow (p_1, \dots, p_N)$, where $p_n = g(\beta^T \mathbf{x}_n)$.
 2. Update $W^{-1} \leftarrow \text{diag} \left(\frac{1}{p_n(1-p_n)} \right)$, for $n = 1, \dots, N$.
 3. Update $z \leftarrow X \beta^T + W^{-1} (\mathbf{t} - \mathbf{p})$.
 4. Update $\beta \leftarrow (X^T W X)^{-1} X^T W z$.
 3. Return $\hat{\beta}$.

Deviance and AIC

In the context of GLM,

$$-2l(\hat{\beta}) = -2 \ln \mathcal{L}(\hat{\beta}) \sim \chi^2_{v=N-d-1}$$

is called the **deviance** (in ML, this is the **error**). The **null deviance** is the deviance of the null model, and the **residual deviance** is the deviance of the proposed model. We can also define, for linear models, the **Akaike Information Criterion**, which is the deviance with a complexity penalization, $-2l(\hat{\beta}) + 2d$. Actually, this penalization $2d \approx 2\|\hat{\beta}\|_0$ is a rudimentary form of **regularization**.

Interpreting the coefficients

$$\text{LOGODDS}(\boldsymbol{x}_0) = \ln \left(\frac{P(\omega_1 | \boldsymbol{x}_0)}{P(\omega_2 | \boldsymbol{x}_0)} \right) = \boldsymbol{\beta}^T \boldsymbol{x}_0 + \beta_0,$$

$$\text{ODDS}(\boldsymbol{x}_0) = \frac{P(\omega_1 | \boldsymbol{x}_0)}{P(\omega_2 | \boldsymbol{x}_0)} = \exp(\boldsymbol{\beta}^T \boldsymbol{x}_0 + \beta_0).$$

Define $\mathbf{1}_i = (0, \dots, 1^{i)}, \dots, 0)^T$, so $\boldsymbol{x}_0 + \mathbf{1}_i = (x_{01}, \dots, x_{0i} + 1, \dots, x_{0N})^T$. Then, we can interpret the coefficient in position i using the following relation between odds:

$$\frac{\text{ODDS}(\boldsymbol{x}_0 + \mathbf{1}_i)}{\text{ODDS}(\boldsymbol{x}_0)} = \exp(\boldsymbol{\beta}^T (\boldsymbol{x}_0 + \mathbf{1}_i - \boldsymbol{x}_0)) = \exp(\beta_i).$$

Poisson Regression

In many statistical studies, one tries to relate a count to some scientific variables. When there is no upper natural bound to this count, the logistic regression is not appropriate, and hence, the **Poisson regression** appears as a good alternative. This model relies on the Poisson distribution, a discrete distribution $X \sim \text{Pois}(\lambda)$ with probability mass function

$$P(X = k) = \exp(-\lambda) \frac{\lambda^k}{k!}, \quad \lambda \in \mathbb{R}^+, k \in \mathbb{N}.$$

We consider independent Poisson random variables T_1, \dots, T_N with $T_n \sim \text{Pois}(\lambda_n)$. We know that $\mathbb{E}[T_n] = \lambda_n$. We have an i.i.d. sample of N observations $\boldsymbol{x}_n \in \mathbb{R}^d$ and a corresponding sample t_1, \dots, t_N , where each t_n is drawn from T_n . Our idea here is to model λ_n as $\exp(\boldsymbol{\beta}^T \boldsymbol{x}_n + \beta_0)$, with link function \ln . Then, the Poisson regression model is $T_n \sim \text{Pois}(\exp(\boldsymbol{\beta}^T \boldsymbol{x}_n + \beta_0))$, or $\ln \lambda_n = \boldsymbol{\beta}^T \boldsymbol{x}_n + \beta_0$.

If we want to estimate the parameter vector using maximum likelihood, we proceed like the previous case for Logistic regression, and we arrive at

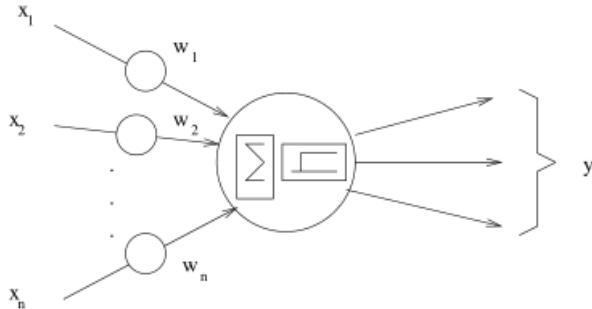
$$l(\boldsymbol{\beta}) = \sum_{n=1}^N [-\exp(\boldsymbol{\beta}^T \boldsymbol{x}_n) + t_n \boldsymbol{\beta}^T \boldsymbol{x}_n - \ln(t_n!)]$$

Again, this expression has no closed form solution for $\boldsymbol{\beta}$ once we derive and set it equal to zero; however, we can still use Newton-Raphson, because $-l$ is a convex function.

Maximum Likelihood framework (II)

8. Artificial Neural Networks (I): The Multilayer Perceptron (MLP)

An (artificial) **neuron** is an abstract computing unit that gets an **input vector**, combines this vectors with a vector of local parameters (called **weights**) and sometimes with other local information, and then **outputs** a scalar quantity. It's usually represented like this:



The output can be delivered as part of the input of another neuron or to the neuron itself (self connection).

Reminder: a directed graph (DG) is a structure composed by a set of nodes and a set of labeled directed segments that connect the nodes.

An **artificial neural network (ANN)** is a parallel and distributed information-processing structure that takes the form of a DG, where the nodes are neurons and the labels correspond to the weights.

A **layer** is a collection of neurons:

- Sharing a common input vector (usually computing the same function).
- Not connected with one another.

The **output layer** in an ANN is the last in the direction of the arrows. All other layers are called **hidden**. A **hidden neuron** is a neuron in a hidden layer. An ANN is **recurrent** if its graph contains cycles; otherwise, it is a **feed-forward** network. A recurrent network represents a dynamical system; a feed-forward network represents a single function.

The simplest choice of an ANN is a linear combination of the inputs:

$$y(\mathbf{x}) = \sum_{i=1}^d w_i x_i + w_0$$

This represents just a **single neuron**. Recall that we previously defined (in Chapter 1) a linear model as a model that, up to an invertible map, is linear on its parameters. Then, this previously defined ANN is a linear model. It can be extended to multiple outputs:

$$y_k(\mathbf{x}) = \sum_{i=1}^d w_{ki} x_i + w_{k0}, \quad k = 1, \dots, m$$

Now, this represents a **layer** of neurons. Finally, if we add a non-linearity to the output:

$$y_k(\mathbf{x}) = g \left(\sum_{i=1}^d w_{ki} x_i + w_{k0} \right), \quad k = 1, \dots, m$$

This still represents a single neuron layer and a linear model, all performing the linear combination and the computation of g . Now, we compact the notation.

1. Define $\mathbf{x} := (1, x_1, \dots, x_d)^T$ and $\mathbf{w}_k := (w_{k0}, w_{k1}, \dots, w_{kd})^T$. We now have:

$$y_k(\mathbf{x}) = g\left(\sum_{i=0}^d w_{ki} x_i\right) = g(\mathbf{w}^T \mathbf{x}), \quad 1 \leq k \leq m$$

2. Let the weight matrix $\mathbf{W}_{(d+1) \times m}$ gather all the weight vectors by columns. We introduce the notation $g[\cdot]$ to mean that g is applied component-wise. The network we now have computes, then,

$$y(\mathbf{x}) = g[\mathbf{W}^T \mathbf{x}]$$

The **activation function** g is often a sigmoidal one: this is defined as a function that satisfies:

- Being differentiable.
- Having a non-negative (or non-positive) bell-shaped first derivative.
- Having horizontal asymptotes in $\pm\infty$.

The most commonly used sigmoidal functions are:

- The **logistic** family:

$$g_\beta^{\text{log}}(z) = \frac{1}{1 + \exp(-\beta z)} \in (0, 1), \quad \beta > 0$$

- The **hyperbolic tangent** family:

$$g_\beta^{\text{tanh}}(z) = \frac{\exp(\beta z) - \exp(-\beta z)}{\exp(\beta z) + \exp(-\beta z)} \in (-1, 1), \quad \beta > 0$$

How to train a single layer ANN: the Delta Rule

We wish to fit $y(\mathbf{x}) = g(\mathbf{w}^T \mathbf{x})$ to a set of learning examples $\{(\mathbf{x}_1, t_1), \dots, (\mathbf{x}_N, t_N)\}$, where $\mathbf{x}_n \in \mathbb{R}^d$, $t_n \in \mathbb{R}$. In order to do this, we define the (empirical) **mean-square error** of the network as

$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^N (t_n - y(\mathbf{x}_n))^2 = \frac{1}{2} \sum_{n=1}^N \left[t_n - g\left(\sum_{i=1}^d w_i x_{n,i} + w_0\right) \right]^2$$

Let $f : \mathbb{R}^r \rightarrow \mathbb{R}$ be a differentiable function; we wish to minimize it by making changes in its variables. Then, the increment in each variable is proportional to the corresponding derivative: $x_i(t+1) := x_i(t) + \Delta x_i(t)$, with

$$\Delta x_i(t) = -\alpha \frac{\partial f}{\partial x_i} \Big|_{\mathbf{x}=\mathbf{x}(t)}, \quad \alpha > 0, i = 1, \dots, r$$

In our case, the function to be minimized is the empirical error and the variables are the weights \mathbf{w} of the network:

$$\begin{aligned} \Delta w_i(t) &= -\alpha \frac{\partial E(\mathbf{w})}{\partial w_j} \Big|_{\mathbf{w}=\mathbf{w}(t)}, \quad \alpha > 0, j = 0, \dots, d \\ \frac{\partial E(\mathbf{w})}{\partial w_j} &= -\sum_{n=1}^N (t_n - y(\mathbf{x}_n)) \cdot g'(\mathbf{w}^T \mathbf{x}_n) \cdot x_{n,j} \end{aligned}$$

This $t_n - y(\mathbf{x}_n)$ is called the **delta**, and $\mathbf{w}^T \mathbf{x}_n$ is called the **net input**. Then, combining the two equations above,

$$\Delta w_j(t) = \alpha \sum_{n=1}^N (t_n - y(\mathbf{x}_n)) \cdot g'(\mathbf{w}(t)^T \mathbf{x}_n) \cdot x_{n,j}$$

When g is the identity, we get the **α -LMS Learning Rule**:

$$\Delta w_j(t) = \alpha \sum_{n=1}^N (t_n - y(\mathbf{x}_n)) \cdot x_{n,j} = \alpha \sum_{n=1}^N (t_n - \mathbf{w}(t)^T \mathbf{x}_n) \cdot x_{n,j}$$

This technique represents a **linear regressor** where the regression coefficients are estimated iteratively. This is a form of learning, but it is **not incremental**: we need all the examples from the beginning. This is also usually called **batch learning**.

We are so sure the iterations get to a minimum because the mean-square error function is convex in \mathbf{w} : it defines a convex hyper-paraboloidal surface with a single **global minimum** \mathbf{w}^* . The constant α controls the stability and speed of convergence. If chosen sufficiently small, the gradient descent procedure asymptotically converges towards \mathbf{w}^* , regardless of the initial vector $\mathbf{w}(0)$. A sufficient condition on α is $0 < \alpha < \frac{2}{\lambda_{\max}}$, where λ_{\max} is the largest eigenvalue of the input self-correlation matrix $\mathbb{E}[\mathbf{x}\mathbf{x}^T] \approx \mathbf{X}\mathbf{X}^T$. In practice, one may use $\alpha < \frac{2}{\sum_{n=1}^N \|\mathbf{x}_n\|^2}$, since $\lambda_{\max} < \text{tr}(\mathbb{E}[\mathbf{x}\mathbf{x}^T]) \approx \text{tr}(\mathbf{X}\mathbf{X}^T) = \sum_{n=1}^N \|\mathbf{x}_n\|^2$.

In the **on-line version** of the delta rule, we also begin with $\mathbf{w}(0)$ arbitrary and apply:

$$\Delta w_j(t) = \alpha_t (t_{n(t)} - y(\mathbf{x}_{n(t)})) x_{n(t),j}$$

At each step t , the example $n(t)$ is drawn at random from $\{1, \dots, N\}$. It can be shown that if $\sum_{t \geq 0} \alpha_t = \infty$ and $\sum_{t \geq 0} \alpha_t^2 < \infty$, then $\mathbf{w}(t)$ converges to the **global minimum** \mathbf{w}^* in the mean square sense:

$$\lim_{t \rightarrow \infty} \|\mathbf{w}(t) - \mathbf{w}^*\|^2 = 0$$

One such procedure is to take $\alpha_t = \frac{\alpha}{t+1}$, with initial $\alpha > 0$.

Towards non-linear models

How could we obtain a model that is non-linear in the parameters (a **non-linear** model)? We depart from the basic linear model:

$$y_k(\mathbf{x}) = g \left(\sum_{i=1}^d w_{ki} x_i + w_{k0} \right), \quad k = 1, \dots, m$$

where g is a sigmoidal function. If we apply any non-linear function to the weights, we are just transforming them into another set of weights, and the model is still linear of the parameters. Then, suppose we apply non-linear functions to the input data:

$$y_k(\mathbf{x}) = g \left(\sum_{i=0}^h w_{ki} \phi_i(\mathbf{x}) \right), \quad k = 1, \dots, m$$

This is a generalization of the previous case, which we can recover by setting $h = d$ and $\phi_i(\mathbf{x}) = x_i$, with $\phi_0(\mathbf{x}) = 1$.

Approach 1. Make $\Phi = (\phi_0, \dots, \phi_h)$ a set of predefined functions. For example, $d = 1$ and polynomial fitting. Consider the problem of fitting the function

$$p(x) = w_0 + w_1 x + \dots + w_h x^h = \sum_{i=0}^h w_i x^i$$

to $x_1, \dots, x_N \in \mathbb{R}$, which is a special case of linear regression, where the set of **regressors** is $1, x, x^2, \dots, x^h$. Therefore $\phi_i(\mathbf{x}) = x^i$. The weights here can be estimated by standard techniques (ordinary least squares).

What if we have a multivariate input $\mathbf{x} = (x_1, \dots, x_d)^T$? The corresponding polynomial is:

$$p(x) = w_0 + \sum_{i_1=1}^d w_{i_1} x_{i_1} + \sum_{i_1=1}^d \sum_{i_2=i_1+1}^d w_{i_1 i_2} x_{i_1} x_{i_2} + \dots$$

The number of possible regressors grows like $\binom{h}{d+h}$. So many regressors (while holding N fixed) causes increasing troubles for estimating parameters: it is mandatory to have more observations than regressors (h must be less than N). Also, the statistical significance of the weights decreases with h and increases with N .

Approach 2. Why not try to engineer **adaptive regressors**? By adapting the regressors to the problem, it is reasonable to expect that we shall need a much smaller number of them for a correct fit.

The basic neural network idea is to **duplicate over and over the model**:

$$y_k(\mathbf{x}) = g\left(\sum_{i=0}^h w_{ki} \phi_i(\mathbf{x})\right), k = 1, \dots, m,$$

where

$$\phi_i(\mathbf{x}) = g\left(\sum_{j=0}^d v_{ij} x_j\right),$$

with $\phi_0(\mathbf{x}) = 1, x_0 = 1$. With one duplication we have created **another layer of neurons**. It is clear that we can play this game as many times as we want, further extending this to any number of **hidden layers** we want. So, to summarize this:

- We have a new set of regressors $\Phi(\mathbf{x}) = (\phi_0(\mathbf{x}), \dots, \phi_h(\mathbf{x}))^T$, which are adaptive via the v_i parameters (called the non-linear parameters).
- Once the new regressors are fully specified (i.e., the v_i parameters are estimated), the remaining task is linear (via the w_k parameters).
- What kind of network gives rise to this function if we keep duplicating? The **Multilayer Perceptron** or **MLP**.
- Under other choices for the regressors, other networks are obtained. For example, the standard **RBF (Radial Basis Functions) network** with parameters μ_i :

$$\phi_i(\mathbf{x}) = \exp\left(-\frac{\|\mathbf{x} - \mu_i\|^2}{2\sigma_i^2}\right).$$

Error functions for classification

In **classification** we model the posteriors $P(\omega_k | \mathbf{x})$. In two-class problems, we model by creating an ANN with one output neuron ($m = 1$) to represent $y(\mathbf{x}) = P(\omega_1 | \mathbf{x})$ and thus $1 - y(\mathbf{x}) = P(\omega_2 | \mathbf{x})$.

Suppose we have a set of learning examples $S = \{(\mathbf{x}_n, t_n)\}_{n=1, \dots, N}$, where $\mathbf{x}_n \in \mathbb{R}^d, t_n \in \{0, 1\}$ (assume S is i.i.d.). We take the convention that $t_n = 1$ means $\mathbf{x}_n \in \omega_1$ and $t_n = 0$ means $\mathbf{x}_n \in \omega_2$, to **model**:

$$P(t | \mathbf{x}) = \begin{cases} y(\mathbf{x}) & \text{if } \mathbf{x} \in \omega_1 \\ 1 - y(\mathbf{x}) & \text{if } \mathbf{x} \in \omega_2 \end{cases}$$

which is conveniently expressed as $P(t|\mathbf{x}) = y(\mathbf{x})^t (1 - y(\mathbf{x}))^{1-t}$. This is a **Bernoulli distribution**. Assuming an i.i.d. sample, the **likelihood function** is:

$$\mathcal{L} = \prod_{n=1}^N y(\mathbf{x}_n)^{t_n} (1 - y(\mathbf{x}_n))^{1-t_n}$$

So, which error should we use? Let us define and minimize (again) the negative log-likelihood as the **error**:

$$E := -\ln \mathcal{L} = -\sum_{n=1}^N [t_n \ln y(\mathbf{x}_n) + (1 - t_n) \ln (1 - y(\mathbf{x}_n))],$$

popularly known as **cross-entropy**. This is the **error function we should use in binary classification**.

The case for more than two classes, $K > 2$, is obtained analogously, though with a bit more work. The error function for the multiclass classification problem turns out to be:

$$E := -\sum_{n=1}^N \sum_{k=1}^K t_{n,k} \ln y_k(\mathbf{x}_n),$$

known as the **Generalized Cross-Entropy**.

Training MLPs: An introduction to the backpropagation algorithm for regression

(adaptation to binary/multiclass classification)

A **regression MLP** of c hidden layers is a function $F : \mathbb{R}^d \rightarrow \mathbb{R}^m$ made up of pieces F_1, \dots, F_m of the form:

$$F_k(\mathbf{x}) = g\left(\sum_{j=0}^{H_c} w_{kj}^{(c+1)} \phi_j^{(c)}(\mathbf{x})\right), \quad k = 1, \dots, m$$

where, for every $l = 1, \dots, c$, $W^{(l)} = [w_{ji}^{(l)}]$ is the matrix of weights connecting layers $l-1$ and l ($w_{ji}^{(l)}$ is the weight of the i -th input of the j -th neuron in layer l), H_l is the number of neurons in layer l (its **size**) and $\phi_j^{(l)}$ is the j -th basis function in layer l ,

$$\phi_j^{(l)}(\mathbf{x}) = g\left(\sum_{i=0}^{H_{l-1}} w_{ji}^{(l)} \phi_i^{(l-1)}(\mathbf{x})\right), \quad l = 1, \dots, c$$

with $\phi_i^{(0)}(\mathbf{x}) = x_i$, $\phi_0^{(l)}(\mathbf{x}) = 1$, $H_0 = d$. In particular, $x_0 = 1$.

The goal in **regression** is to minimize the empirical error of the network on the training data sample $S = \{(\mathbf{x}_n, \mathbf{t}_n)\}_{n=1, \dots, N}$, where $\mathbf{x}_n \in \mathbb{R}^d$, $\mathbf{t}_n \in \mathbb{R}^m$. This empirical error is

$$E_{\text{emp}}(\boldsymbol{\omega}) := \frac{1}{2} \sum_{n=1}^N \sum_{k=1}^m (t_{nk} - F_k(\mathbf{x}_n))^2,$$

where $\boldsymbol{\omega}$ is the vector of all network weights. Some mathematical concerns about this function:

- E is a continuous function if g is so.
- E is differentiable if g is so.
- g is usually of class C^∞ .

If we want to apply, for example, **gradient descent**, we need to compute the partial derivative of the error w.r.t. every weight. This, as you may already know, is called the **gradient vector**,

$$\nabla E_{\text{emp}}(\boldsymbol{\omega}) = \left(\frac{\partial E_{\text{emp}}(\boldsymbol{\omega})}{\partial w_{ji}^{(l)}} \right)_{l,i,j}$$

There exists a reasonably efficient algorithm for computing this gradient vector: the **backpropagation algorithm**. But, before we get to this, let us first define some redundant notation that will ease computations:

- First of all, we consider a MLP where, for notational simplicity, we define:

$$z_j^{(l)} := g(a_j^{(l)}) := g\left(\sum_i w_{ji}^{(l)} z_i^{(l-1)}\right), \quad z_j^{(0)} = x_j$$

- Also, note that E_{emp} is the sum of the (independent) errors for every I/O example $(\mathbf{x}_n, \mathbf{t}_n)$:

$$E_{\text{emp}}(\boldsymbol{\omega}) = \sum_{n=1}^N \frac{1}{2} \sum_{k=1}^m (t_{nk} - F_k(\mathbf{x}_n))^2 := \sum_{n=1}^N E_{\text{emp}}^{(n)}(\boldsymbol{\omega})$$

Therefore,

$$\frac{\partial E_{\text{emp}}(\boldsymbol{\omega})}{\partial w_{ji}^{(l)}} = \sum_{n=1}^N \frac{\partial E_{\text{emp}}^{(n)}(\boldsymbol{\omega})}{\partial w_{ji}^{(l)}}$$

As we are using the gradient descent method, the updating formula for the weights is:

$$w_{ji}^{(l)}(t+1) := w_{ji}^{(l)}(t) - \alpha \frac{\partial E_{\text{emp}}(\boldsymbol{\omega})}{\partial w_{ji}^{(l)}} \Big|_{\boldsymbol{\omega}=\boldsymbol{\omega}(t)}$$

Please note that there are better methods than gradient descent, albeit this is a fair starting point.

Now, suppose we present \mathbf{x}_n to the network and compute all the neuron's outputs $z_j^{(l)}$; this is known as **forward propagation**. Now,

$$\Delta^n w_{ji}^{(l)} := \frac{\partial E_{\text{emp}}^{(n)}(\boldsymbol{\omega})}{\partial w_{ji}^{(l)}} = \frac{\partial E_{\text{emp}}^{(n)}(\boldsymbol{\omega})}{\partial a_j^{(l)}} \cdot \frac{\partial a_j^{(l)}}{\partial w_{ji}^{(l)}} = \delta_j^{(l)} \cdot z_i^{(l-1)}$$

where we have defined $\delta_j^{(l)} := \frac{\partial E_{\text{emp}}^{(n)}(\boldsymbol{\omega})}{\partial a_j^{(l)}}$. Note that, for $l = c + 1$, this δ is exactly the δ from the

Delta Rule for single-layer ANN training. The algorithm to update the weights is the following:

BACKPROPAGATION ALGORITHM (BPA):

- set initial values for the weights $w_{ji}^{(l)}$.
- **repeat:**
 - **for** n **in** $[N] := \{1, \dots, N\}$
 - **Forward step:** present \mathbf{x}_n to the network and compute the outputs $z_i^{(l)}$ of all units.
 - **Backward step:** compute the deltas $\delta_j^{(l)}$ of all units, from $l = c + 1$ down to $l = 1$:
 - **if** $l = c + 1$ **then** $\delta_j^{(l)} := g'(a_j^{(c+1)}) \cdot (z_j^{(c+1)} - t_{nj})$

- **if** $l < c + 1$ **then** $\delta_j^{(l)} := g'(a_j^{(l)}) \sum_q \delta_q^{(l+1)} w_{qj}^{(l+1)}$
- Set $\Delta^n w_{ji}^{(l)} := \delta_j^{(l)} \cdot z_i^{(l-1)}$.
- **end**
- Update the weights as $w_{ji}^{(l)}(t+1) := w_{ji}^{(l)}(t) + \alpha \sum_{n=1}^N \Delta^n w_{ji}^{(l)}$.
- **until** convergence **or** max. epochs

We now set the activation functions to be the examples we have considered: if $g = g_\beta^{\log}(z)$, then

$$g' = \left(g_\beta^{\log}(z)\right)' = \beta g_\beta^{\log}(z) \left[1 - g_\beta^{\log}(z)\right],$$

and we obtain

$$\begin{aligned} g'\left(a_j^{(c+1)}\right) &= \beta g\left(a_j^{(c+1)}\right) \left(1 - g\left(a_j^{(c+1)}\right)\right) = \beta z_j^{(c+1)}(1 - z_j^{(c+1)}), \\ g'\left(a_j^{(l)}\right) &= \beta g\left(a_j^{(l)}\right) \left(1 - g\left(a_j^{(l)}\right)\right) = \beta z_j^{(l)} \left(1 - z_j^{(l)}\right). \end{aligned}$$

Analogously for $g_\beta^{\tanh}(z)$, where we have

$$\left(g_\beta^{\tanh}(z)\right)' = \beta^2 \left(1 - \left(g_\beta^{\tanh}(z)\right)^2\right).$$

Relation between backpropagation and order methods

Let $\omega(t)$ represent a vector with **all** network weights at time t . When training an MLP, the minimization of E_{emp} using the BPA involves a sequence of weight iterates $\{\omega(t)\}_{t=0}^\infty$, where t indicates iterations through S , the so-called **epochs**.

A basic algorithm finds the next weights using the relation

$$\omega(t+1) := \omega(t) - \alpha(t) \nabla E_{\text{emp}}(\omega(t)).$$

The role of the BPA is to compute the elements of $\nabla E_{\text{emp}}(\omega(t))$ at each iteration, recursively and rather efficiently. But how do we choose this parameter α ? Let us present a theorem that shines little light onto this:

General convergence theorem. Let λ_k be the eigenvalues of the Hessian matrix of the empirical error, $\nabla^2 E_{\text{emp}}(\omega(t))$ for a given $\omega(t)$, a matrix we assume to be positive definite. Then, if $|1 - \alpha \lambda_k| < 1$ for all k , $\omega(t)$ tends towards a local minimum of $E_{\text{emp}}(\omega(t))$ as $t \rightarrow \infty$.

Observations on this theorem:

- Recall that $\nabla^2 E_{\text{emp}}(\omega)$ is the Hessian matrix, with elements $\frac{\partial^2 E_{\text{emp}}(\omega)}{\partial \omega_i \partial \omega_j}$.
- It is straightforward to see that $\alpha < \frac{2}{\lambda_{\max}}$ is a sufficiently small α (*but it is not a necessary condition*).
- Too large values of α show fast convergence but a tendency to oscillate.
- Too small values of α show slow convergence.

A generic (iterative) **minimization algorithm** would be:

- Choose an initial point x_0 ; set $t = 0$.
- Select a search direction p_t .
- Select a step size α_t , and set $x_{t+1} := x_t - \alpha_t p_t$.
- Return to 2, unless a convergence criterion has been met.

Suppose that $f : \mathbb{R}^r \rightarrow \mathbb{R}$ we wish to minimize (we assume it is differentiable and analytic). A **first order** Taylor expansion around the current point x_t is:

$$f(\mathbf{x}) \approx f(\mathbf{x}_t) + \nabla f(\mathbf{x}_t)^T (\mathbf{x} - \mathbf{x}_t) = \hat{f}(\mathbf{x})$$

A simple minimization algorithm is to set $\mathbf{x}_{t+1} := \mathbf{x}_t - \alpha_t \nabla f(\mathbf{x}_t)$ for $\alpha_t > 0$, since

$$\begin{aligned}\hat{f}(\mathbf{x}_{t+1}) &= \\ f(\mathbf{x}_t) + \nabla f(\mathbf{x}_t)^T (\mathbf{x}_{t+1} - \mathbf{x}_t) &= \\ f(\mathbf{x}_t) + \nabla f(\mathbf{x}_t)^T (\mathbf{x}_t - \alpha_t \nabla f(\mathbf{x}_t) - \mathbf{x}_t) &= \\ f(\mathbf{x}_t) - \alpha_t \nabla f(\mathbf{x}_t)^T \nabla f(\mathbf{x}_t) &= \\ f(\mathbf{x}_t) - \alpha_t \|\nabla f(\mathbf{x}_t)\|^2 &< \\ f(\mathbf{x}_t)\end{aligned}$$

provided that $\nabla f(\mathbf{x}_t) \neq 0$ and $\alpha_t > 0$ is small enough. Therefore we derive a **learning rule**:

$$\boldsymbol{\omega}(t+1) := \boldsymbol{\omega}(t) - \alpha_t \nabla E_{\text{emp}}(\boldsymbol{\omega}(t)).$$

We have recovered the **gradient descent** method. But, this Taylor expansion gives no clue on how to choose $\alpha_t > 0$. In ANNs, there are two common strategies:

- Use a **constant** small $\alpha > 0$ or a **variable** $\alpha_t = \frac{\alpha_0}{t+1}$, $\alpha_0 > 0$.
- Perform a costly **line search** to find $\alpha_t = \operatorname{argmin}_\alpha f(\mathbf{x}_t - \alpha \nabla f(\mathbf{x}_t))$.

A perhaps better strategy than all of this is to use a second-order Taylor expansion around the current point \mathbf{x}_t :

$$f(\mathbf{x}) \approx f(\mathbf{x}_t) + \nabla f(\mathbf{x}_t)^T (\mathbf{x} - \mathbf{x}_t) + \frac{1}{2} (\mathbf{x} - \mathbf{x}_t)^T \nabla^2 f(\mathbf{x}_t) (\mathbf{x} - \mathbf{x}_t) = \hat{f}(\mathbf{x}),$$

where $\nabla^2 f(\mathbf{x}_t)$ is the Hessian matrix of f evaluated at \mathbf{x}_t . Provided $\nabla^2 f(\mathbf{x}_t)$ is positive definite, this time the minimum of $\hat{f}(\mathbf{x})$ occurs at the \mathbf{x} satisfying the zero-derivative condition:

$$\nabla f(\mathbf{x}_t) + \nabla^2 f(\mathbf{x}_t) (\mathbf{x} - \mathbf{x}_t) = 0$$

This leads to the minimization step $\mathbf{x}_{t+1} := \mathbf{x}_t - (\nabla^2 f(\mathbf{x}_t))^{-1} \nabla f(\mathbf{x}_t)$. This **second-order** method is known as a **Newton method**. Note that there is no need for a learning rate α_t here, sinze the inverse of the Hessian determines both the step size and the search direction. However, this method has some practical drawbacks:

- It requires the Hessian to be positive definite, otherwise there is no unique minimum.
- It requires the exact computation of the Hessian at every iteration.
- It requires a matrix inversion at every iteration.
- The knowledge of the local curvature provided by the Hessian is only useful very close to \mathbf{x}_t .

To avoid this drawbacks, we will replace the Hessian matrix with an approximation: let's see **quasi-Newton** methods.

In quasi-Newton methods, the inverse of the Hessian matrix is directly approximated, leading to

$$\mathbf{x}_{t+1} := \mathbf{x}_t - \alpha_t M_t \nabla f(\mathbf{x}_t), \text{ where } M_t \approx (\nabla^2 f(\mathbf{x}_t))^{-1}.$$

The most common variant is the **BFGS method**, suggested independently by Broyden, Fletcher, Goldfarb and Shanno. In BFGS, the positive definite estimate of the inverse Hessian does not require matrix inversion and uses **only gradient information**, supplied by the BPA:

- A positive definite matrix M_0 is chosen (usually, the identity matrix I).
- The search direction is set to $M_t \nabla f(\mathbf{x}_t)$.
- A **line search** is performed along this direction to find α_t .
- $\mathbf{x}_{t+1} := \mathbf{x}_t - \alpha_t M_t \nabla f(\mathbf{x}_t)$.
- M_{t+1} is set, according to some conditions:

We impose a quasi-Newton condition: M_{t+1} has to satisfy the *secant equation*,

$$M_{t+1}(\mathbf{x}_{t+1} - \mathbf{x}_t) = \nabla f(\mathbf{x}_{t+1}) - \nabla f(\mathbf{x}_t)$$

If we define $s_t := \mathbf{x}_{t+1} - \mathbf{x}_t$ and $y_t := \nabla f(\mathbf{x}_{t+1}) - \nabla f(\mathbf{x}_t)$, the secant equation is $B_{t+1}s_t = y_t$. Then, the approximation of the Hessian is updated by summing two matrices, U_t and V_t , of rank one. As we want to maintain the positive definiteness of M_t , we can choose

$U_t = \beta uu^T$, $V_t = \gamma vv^T$. If we impose the secant equation and choose $u = y_t$, $v = M_t s_t$, we can obtain

$$\beta = \frac{1}{y_t^T s_t}, \quad \gamma = \frac{1}{s_t^T M_t s_t}.$$

Finally, substituting in the formula for M_{t+1} , we can set it to be

$$M_{t+1} = M_t + \beta uu^T + \gamma vv^T = M_t + \frac{1}{y_t^T s_t} y_t y_t^T + \frac{1}{s_t^T M_t s_t} M_t s_t s_t^T M_t^T.$$

Tricks of the trade: sensible choices when implementing and training MLPs

First of all, let us present the R function `nnet`:

```
nnet(x, y, weights, size, wts, mask,
      linout = FALSE, entropy = FALSE, softmax = FALSE,
      censored = FALSE, skip = FALSE, rang = 0.7, decay = 0,
      maxit = 100, Hess = FALSE, trace = TRUE, MaxNWts = 1000,
      abstol = 1.0e-4, reltol = 1.0e-8, ...)
```

This function fits **single-hidden-layer neural networks**, using the following parameters:

- `x` as the inputs
- `y` as the targets
- `weights` as an optional weights vector for the inputs, in case we want to give more importance to some of them
- `size` as the size of the hidden layer (i.e. the number of neurons)
- `wts` as an optional initialization for the network weights
- `mask` is an optional bitmask to interpret which variables to take into account in the inputs
- `linout`, `entropy` and `softmax` are three boolean parameters that can be set to true (only one at once) to train the network for the following purposes:

	<code>softmax</code>	<code>entropy</code>	<code>linout</code>
Classification ($K = 2$)	FALSE	TRUE	FALSE
Classification ($K > 2$)	TRUE	FALSE	FALSE
Regression	FALSE	FALSE	TRUE

- `skip` lets us say if there are any connections from the input directly to the output layer
- `decay` is a regularization parameter
- The other parameters can be easily understood

The output layer of a feed-forward neural network is schematized like this:

	number of neurons	activation function	$y_k, k = 1, \dots, m$
Classification ($K = 2$)	1	logistic	$P(\omega_1 \mathbf{x})$
Classification ($K > 2$)	K	softmax	$P(\omega_k \mathbf{x})$
Regression	m	identity	f_k

Given that f_k is the k -th component of the function to learn.

On-line vs. Batch learning:

On-line training is much faster, but the notion of convergence is unexistent. Even though, it is much faster than batch, particularly in large redundant datasets, and often results in better solutions. It can also track changes in the data.

Batch training, on the other hand, is much better understood as the notion of convergence does exist. It can be accelerated using second-order information, and it is not sensitive to the ordering of the training examples, which on-line learning is.

As of today, we call **mini-batches** (different) subsets of training observations used in consecutive epochs.

Activation function: It is known experimentally that an MLP trains much faster (as in, reaches an optimum in less iterations) if the activation function is anti-symmetric, $g(-z) = -g(z)$. This is the case for the tanh.

Target values: these should not be the asymptotic values of the activation function, but they should leave a margin. For example, in a two class problem, it is better to encode class 0 as 0.05 and class 1 as 0.95 than to encode them as 0 and 1.

Input values: these should be preprocessed so that:

- their mean value is zero and their standard deviation is 1 (**standardized**)
- they are uncorrelated (perhaps using **PCA**)

Initial weights: these should be small and zero-centered, to avoid initially driving the neurons into saturation.

Number of hidden neurons

While the number of inputs and outputs are dictated by the problem, the number of hidden neurons is not directly related to the application domain. This largely remains an unsolved problem.

- A small number may not be sufficient and provoke **underfitting** of the model.
- A large number may be too much and provoke **overfitting** of the model.

There exist many methods to set this number: constructive (for example, **cascade correlation**), pruning or destructive (for example, **optimal brain damage**), golden searches, ...

But, arguably a good and recommendable method, as it is more principled and arguably faster, is to use a large number (to **overfit** the model) and **regularize** the network. In `nnet`, we set the decay parameter and we can then perform cross-validation on it. There's more on ANN regularization and neuron dropout in the `articles` directory of this repository.

So, there are two simple and common ways to find the best single-hidden-layer network architecture:

- Explore different number of hidden networks, without regularization.
- Fix a large number of hidden units and explore different regularization values.

What should not be done is both at once. It is usually a waste of computing resources.

9. Artificial Neural Networks (II): RBF Networks

Introduction

RBFs have their roots at exact function interpolation in mathematics. Its formulation as a single-hidden-layer neural network came later. In a RBF model, the output of a neuron is determined by the **distance** between the input and the neuron's parameter vector, called its **center** or **prototype**. This latter fact has two important consequences:

1. It allows to give a precise interpretation to the network output: it will be a **linear combination of similarities**. This similarities will be calculated from the distances using the activation function.
2. It allows to design training algorithms other than BPA, which will be **de-coupled**, meaning there will be different ways for establishing the parameters of the hidden layer (non-linear part) and the output layer (linear part).

But what is **exact function interpolation**? The setting is the following: we have a set of data points \mathbf{x}_n who all live in \mathbb{R}^d , and a set of real numbers t_n . Given this, we want a function $h : \mathbb{R}^d \rightarrow \mathbb{R}$ such that

$$h(\mathbf{x}_n) = t_n, \quad \mathbf{x}_n \in \mathbb{R}^d, t_n \in \mathbb{R}, n = 1, \dots, N$$

In our case, we will express h as a combination of **basis functions** like the following:

$$\phi_n(\mathbf{x}) := \phi(\|\mathbf{x} - \mathbf{x}_n\|)$$

Each data point \mathbf{x}_n acts as a prototype. This combination will be linear with respect to the basis functions:

$$h(\mathbf{x}) = \sum_{n=1}^N w_n \phi_n(\mathbf{x}) = \sum_{n=1}^N w_n \phi(\|\mathbf{x} - \mathbf{x}_n\|),$$

which we will force to be exact for all the data points: $h(\mathbf{x}_n) = t_n$. The function $\|\cdot\|$ is any norm in \mathbb{R}^d , and most often an **Euclidean norm** (a norm heir of a scalar product). Because of the norm, the ϕ_n are functions that exhibit **radial contours** of constant value **centered** at the data points \mathbf{x}_n . In matrix notation,

$$\begin{pmatrix} \phi_1(\mathbf{x}_1) & \phi_2(\mathbf{x}_1) & \cdots & \phi_N(\mathbf{x}_1) \\ \phi_1(\mathbf{x}_2) & \phi_2(\mathbf{x}_2) & \cdots & \phi_N(\mathbf{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(\mathbf{x}_N) & \phi_2(\mathbf{x}_N) & \cdots & \phi_N(\mathbf{x}_N) \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{pmatrix} = \begin{pmatrix} t_1 \\ t_2 \\ \vdots \\ t_N \end{pmatrix}$$
$$\Phi \mathbf{w} = \mathbf{t}$$

Note that Φ is a symmetric matrix, as $\|\mathbf{x}_i - \mathbf{x}_j\| = \|\mathbf{x}_j - \mathbf{x}_i\|$. Assuming that this matrix is non-singular, \mathbf{w} can be found as $\mathbf{w} = \Phi^{-1}\mathbf{t}$. This can be done, for example, using $\Phi = LU$ decomposition. There exist some sufficient conditions on ϕ as to make sure that Φ is invertible, known as **Micchelli's theorem**. These sufficient conditions include:

- $\phi(z) = \exp\left(-\frac{z^2}{\sigma^2}\right)$ (Gaussian RBF)
- $\phi(z) = (z^2 + \sigma^2)^\alpha, \alpha \in \mathbb{R}^* < 1$
- $\phi(z) = z^3$
- $\phi(z) = z^2 \ln z$

If the interpolation problem has image in \mathbb{R}^m (i.e., $t_n \in \mathbb{R}^m$), the generalization is straightforward:

$$h_k(\mathbf{x}) = \sum_{n=1}^N w_{kn} \phi_n(\mathbf{x}) = \sum_{n=1}^N w_{kn} \phi(\|\mathbf{x} - \mathbf{x}_n\|), \quad 1 \leq k \leq m$$

that we will force to be exact for all the data points: $h_k(\mathbf{x}_n) = t_{nk}$. This problem leads to

$$\Phi W = T$$

solved again by simple matrix inversion as $W = \Phi^{-1}T$.

Regularization

Very often, in Machine Learning, the exact function interpolation setting is **not attractive** at all:

- High number of interpolation points N means **complex** and **unstable** solutions.
- The outputs t_n depend stochastically on the inputs \mathbf{x}_n . So, if we have **exact** solutions, we will be victims of **overfitting**.
- The interpolation matrix Φ can be singular or ill-conditioned.
- The inversion of Φ grows as $O(N^3)$. For symmetric positive definite matrices, Cholesky decomposition takes some $\frac{N^3}{3}$ steps.

We are in need of a tighter **control of complexity** of our solution. Enter **regularization**.

From previous lectures we know that regularization penalizes the size of the weight matrix,

$$E_{\text{emp}}(W) = \frac{1}{2} \sum_{n=1}^N \sum_{k=1}^m (t_{nk} - h_k(\mathbf{x}_n))^2 + \frac{\lambda}{2} \sum_{k=1}^m \|w_k\|^2,$$

which results in $W = (\Phi + \lambda I_N)^{-1}T$; the value of $\lambda > 0$ is proportional to the amount of noise in the data. Another way of obtaining much simpler solutions is to use a **subset** of the data points to center the basis functions; more generally, they can be centered at a **very carefully selected** set of points in \mathbb{R}^d . With these two modifications (regularization and lowering the number of centers), we can get into **RBF networks**.

RBF networks

The so-called **RBF network** is of the form:

$$h_k(\mathbf{x}) = \sum_{i=0}^H w_{ki} \phi_i(\mathbf{x}) = \sum_{i=0}^H w_{ki} \phi(\|\mathbf{x} - \mathbf{c}_i\|), \quad 1 \leq k \leq m,$$

which is a **two-layer neural network**:

- The first (hidden) layer of $H \ll N$ neurons computes the basis functions $\phi_i(\mathbf{x})$ centered at the vectors \mathbf{c}_i .
- A constant basis function $\phi_0(\mathbf{x}) = 1$ compensates for the difference between the mean values of the output and the targets.

A very popular choice for the ϕ_i is a simple Gaussian:

$$\phi_i(\mathbf{x}) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{c}_i\|^2}{\sigma_i^2}\right).$$

The new matrix $\Phi_{N \times (H+1)}$ is sometimes known as the **design matrix**; now the weight matrix is

$$W = (\Phi^T \Phi)^{-1} \Phi^T T.$$

If the original $\Phi_{N \times N}$ matrix was non-singular, then the matrix $\Phi_{N \times (H+1)}^T \Phi_{N \times (H+1)}$ is also non-singular (this is a very important result). Note that this solution is not regularized: if we do regularize it, we get that

$$W = (\Phi^T \Phi + \lambda I_{H+1})^{-1} \Phi^T T.$$

RBF network training is typically performed in a decoupled way:

1. The first stage finds $H, \{\mathbf{c}_i\}, \{\sigma_i^2\}$ using a **clustering** algorithm such as k-means.
2. The second stage finds W by any of the usual (linear) methods:
 - o Solution using the Moore-Penrose pseudo-inverse (via the SVD), for **regression**.
 - o Solution using IRLS (logistic regression), for **binary classification**.

10. Ensemble methods: Random Forests

Ensemble methods are learning methods that learn by training a number of **better-than-chance individual learners** and then combine their predictions. This methods have a number of advantages:

- They provide **more accurate models** by combining the output of multiple local "experts".
- A complex problem can be decomposed into multiple sub-problems that are easier to understand and solve (**divide-and-conquer**).
- We often train many models and feel we can get more by **combining a number of them**. In this way, computational efforts are not wasted.

We should use ensemble learning methods when we can design better-than-chance individual learners that are **independent from each other**. How do we combine the multiple individual outputs of each learner to get a unique answer?

- For the **classification** task, we use **plurality voting**: the class that has the most votes is the answer.
- For the **regression** task, we **average** all the answers (usually with the arithmetic mean).

Regression: call \mathcal{E} the ensemble, \mathcal{L}_i the i -th individual learner, $1 \leq i \leq L$; fix an input vector \mathbf{x} and let $\varepsilon_i := \mathbb{E}[|f(\mathbf{x}) - \mathcal{L}_i(\mathbf{x})|]$, where f is the function we want to learn:

1. The **expected MSE for a randomly chosen individual learner** is:

$$\frac{1}{L} \sum_{i=1}^L \varepsilon_i^2$$

2. The **expected MSE for the ensemble** is:

$$\left(\frac{1}{L} \sum_{i=1}^L \varepsilon_i \right)^2$$

Using Cauchy-Schwarz inequality, we can prove that

$$\left(\frac{1}{L} \sum_{i=1}^L \varepsilon_i \right)^2 \leq \frac{1}{L} \sum_{i=1}^L \varepsilon_i^2$$

In particular, if the individual learners are independent, the reduction in expected error would be $\frac{1}{L}$. In practice this is not the case, because the individual learners are normally trained **out of the same data**. Thus, they will tend to have correlated errors, and then the improvement will be smaller.

Classification: to ease the problem, let us assume the following:

- The number of classifiers L is odd and we have $K \geq 2$ class labels.
- Each classifier gives the correct class with probability p for any \mathbf{x} .
- The classifier outputs are independent.

Then, the majority vote will give an accurate class label if at least $\lfloor \frac{L}{2} \rfloor + 1$ classifiers give correct answers. The majority ($50\% + 1$) of votes is necessary and sufficient for a correct decision in the case of $K = 2$, and is a sufficient condition (but not a necessary one) for $K > 2$. Thus, the real accuracy of an ensemble using plurality when $K > 2$ could be greater than the majority vote accuracy.

Consider $K = 2$. The **accuracy** of the ensemble is:

$$a_{\mathcal{E}} := \sum_{i=\left[\frac{L}{2}\right]+1}^L \binom{L}{i} p^i (1-p)^{L-i}$$

The **Condorcet Jury Theorem** states that:

1. If $p > \frac{1}{2}$, then $a_{\mathcal{E}}$ is monotonically increasing and $a_{\mathcal{E}} \rightarrow 1$ as $L \rightarrow \infty$.
2. If $p < \frac{1}{2}$, then $a_{\mathcal{E}}$ is monotonically decreasing and $a_{\mathcal{E}} \rightarrow 0$ as $L \rightarrow \infty$.
3. If $p = \frac{1}{2}$, then $a_{\mathcal{E}} = \frac{1}{2}$ for any L .

CART algorithm

The **CART** (Classification And Regression Tree) algorithm generates a classification tree using the **Gini score Gain**: how often a randomly chosen example would be incorrectly labeled if it were labeled according to the empirical probability distribution of the classes. Mathematically, the Gini score is:

$$\text{Gini}(S) := \sum_{k=1}^K p_k(S) (1 - p_k(S)) = 1 - \sum_{k=1}^K p_k^2(S)$$

being $p_k(S)$ the fraction of examples in S labeled with class k . Note that if a variable completely characterizes the class (perfectly separates S into the labeled classes), then $\text{Gini}(S) = 1 - 1^2 = 0$, which is a "pure" scenario. If the probability distribution is uniform, then the Gini score achieves its maximum value, $1 - \frac{1}{K}$.

The **Gini Gain of a discrete variable** S is an impurity-based criterion that measures the divergence between probability distributions of the class labels. It is mathematically calculated as

$$\text{GiniGain}(S) := \text{Gini}(S) - \sum_{v=1}^V \frac{|S_v|}{|S|} \text{Gini}(S_v),$$

assuming S_v are the elements of S with value v . If S is a continuous variable, we sort the data rows and bin-split: we calculate the Gini score for the binary question "do the elements have a value greater than μ_i ? ", with $\mu_i = \frac{\text{value in row } i + \text{value in row } i+1}{2}$, for $i = 1, \dots, |S| - 1$. In this way, we get a binary tree.

CART can also build regression trees, albeit it does not use the Gini score for that matter, but tries to maximize the **RSS gain** (Residual Sum of Squares) at each step. In this sense, both classification and regression trees built by CART follow a greedy algorithm.

Methods for constructing ensembles

We want to make the individual learners in the ensemble be the least correlated possible. There have been a lot of ways proposed to do this, such as:

- **Subsampling the training examples**
- Manipulating the features
- Manipulating the targets: for example, make each individual learner be an expert of classifying observations in a particular class.
- Changing the learning parameters
- ...

Or a combination of the above. We will delve into subsampling of the training examples. The most common way to do this is by **Bootstrap Resampling**.

Bootstrap Resampling

Assuming we have a data set $D = \{(\mathbf{x}_1, t_1), \dots, (\mathbf{x}_N, t_N)\}$, we draw **bootstrap resamples** D_1^*, \dots, D_B^* of size N by sampling D with replacement and then fit a model to each of the D_b^* . A statistic $\hat{\theta} = \theta(D)$ can be estimated in the usual ways. For example,

$$\bar{\theta}^* = \frac{1}{B} \sum_{b=1}^B \theta_b^*, \quad \text{Var}(\theta^*) = \frac{1}{B-1} \sum_{b=1}^B (\theta_b^* - \bar{\theta}^*)^2$$

are the **mean** and **variance** of the bootstrap distribution of $\hat{\theta}$, and $\theta_b^* := \theta(D_b^*)$. An estimation for the **bias** of $\hat{\theta}$ is $\bar{\theta}^* - \hat{\theta}$.

If we want to **subsample the training set**, we might use a method created by Leo Breiman (1996) called **Bagging (Bootstrap aggregating)**: this creates an ensemble by training individual classifiers on bootstrap resamples of the training set. The procedure is the following:

1. Generate a bootstrap sample from a sample of size N (N independent draws with replacement).
2. Train a predictor y_b^* on every bootstrap resample D_b^* .
3. Repeat the process until you have B resamples and predictors.

What is the expected size of a bootstrap resample? Well, in a data set with N examples, each has a probability of not being chosen over all the N choosings of

$$\left(1 - \frac{1}{N}\right)^N,$$

so the probability of being chosen at least once is

$$p(N) := 1 - \left(1 - \frac{1}{N}\right)^N.$$

If $N \rightarrow \infty$,

$$\lim_{N \rightarrow \infty} p(N) = 1 - \exp(-1) \approx 0.632.$$

The perturbation in the training set due to bootstrap resampling causes different methods to be built, particularly if the classifier is unstable. A modelling method is **unstable** if a small change in the training data (e.g., order of presentation, addition or deletion of data) can lead to a radically different model. This is typical of **overfit** models: they have a **low bias** and a **high variance**. Examples of these include **decision trees** and **neural networks**. Using the bagging procedure, we can dramatically **reduce the variance** of unstable methods of the same type, leading to improved prediction capabilities, at the price of keeping or increasing the bias (although not much).

Now, we can see that for bagging we do not need new data to estimate the prediction error of a model. We can do so by calculating what's called the **Out-of-bag error (OOB)**. To calculate this, we use data that is not in the bootstrap resamples: define

- $\mathbb{I}(z)$ as the indicator function of boolean condition z .
- $D^{-n} \subseteq \{1, \dots, B\}$ as the set of indices of the resamples that do not contain observation \mathbf{x}_n ,

$$D^{-n} = \{i \in \{1, \dots, B\} : \mathbf{x}_n \notin D_i^*\}.$$

- y_b^* as the model fitted to D_b^* .

Then, the OOB error (**validation error**) is calculated using

$$\text{Err}^* := \frac{1}{N} \sum_{n=1}^N \frac{1}{|D^{-n}|} \sum_{b \in D^{-n}} \mathbb{I} [t_n \neq y_b^*(\mathbf{x}_n)] .$$

Similarly, the **resubstitution** (aka training) error is estimated as:

$$\bar{e}^* := \frac{1}{N} \sum_{n=1}^N \frac{1}{|D^n|} \sum_{b \in D^n} \mathbb{I} [t_n \neq y_b^*(\mathbf{x}_n)] ,$$

where $D^n = \{1, \dots, B\} \setminus D^{-n}$. The **0.632-bootstrap estimate** is defined by:

$$\text{Err}^{(0.632)} := 0.368\bar{e}^* + 0.632\text{Err}^* .$$

Intuitively, this pulls the OOB bootstrap estimate down toward the training error, thereby reducing its likely upward bias.

Random Forests

Random Forests (Breiman, 2001) are an ensemble method similar to bagging, where the individual learners are **decision trees**, with additional randomization:

- We first generate bootstrap resamples and build a tree for each one. But, the trees are built in a more random fashion than normal decision trees:
 - Their **diversity** is increased (they are *decorrelated*) by the following procedure: every time a split is considered in a tree (an *internal node*), a subset of $m \ll p$ predictors is randomly chosen from the whole predictor pool. Then, the split is only allowed to use one predictor of those m (the best one in terms of Gini gain). The default values are \sqrt{p} for classification and $\frac{p}{3}$ for regression; these are due to L. Breiman also. If we used $m = p$, we would be directly using **bagging**.

So, we have two basic hyperparameters: B , the number of trees, and m , the number of local features chose at each split of every tree; both can be optimized via the out-of-bag error. Construction of the random forest is fast, since just a few features are explored per tree.

Random Forests usually outperform both the individual predictors and direct bagging. They can be also used to **estimate variable relevance**:

- For each tree, the OOB prediction error is computed.
- Then again after permuting each predictor variable.
- The difference between the two is averaged over all trees, and normalized by its standard deviation:
 - $\text{raw}_b(j) := \text{OOB}(b, j) - \text{OOB}(b)$, $1 \leq b \leq B$.
 - $\text{imp}_j := \frac{1}{B} \sum_{b=1}^B \text{raw}_b(j)$

If we have a probabilistic classifier, instead of bagging we could average the probabilities for each class in an ensemble, and then assign the class with the highest probability.