APPUNTI DI MACHINE LEARNING

a cura di:
MARCO ZANETTI

Indice

1 Machine learning

Given a collection of examples (called training data), we want to be able to make predictions about novel, but incomplete, examples. In machine learning, we believe there is a pattern in the data and we can learn it, but we do cannot pin down the pattern; It most often focuses on prediction. It differs from *data mining*, where we do have a model in mind to characterize the data, and from *statistics*, which focus on statistical properties of the data without computational considerations. We can pin down the definition of learning to:

- The existence of a pattern in the data;
- We have no knowledge of this pattern $\Rightarrow$ we cannot pin it down mathematically or with very simple rules;
- We can try to learn the pattern using the data at our disposition.

1.1 Formal model (statystical learning)

Given a specific *learner*, which has access to:

1 **Domain set $\boldsymbol{X}$**: X represent the set of all the possible object to make predictions about. X is called the *instance space*. We say that a domain point $x \in X$ is called instance and it's usually represented by a vector of features;

2 **Label set $\boldsymbol{Y}$**: Y represent the set of all possible layers. Often we have binary layers like $\{-1, 1\}$ or $\{0, 1\}$

3 **Training data $\boldsymbol{S}$**: (also called *training set*) S represent the data used to train the learner. $S = ((x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m))$ is the set of pairs given by an instance x_i and a label y_i for that instance. This set is also the learner input.

4 **Learner's output $\boldsymbol{h}$**:(also called predictor, hypothesis or classifier) The prediction rule h is a function $h : X \to Y$ that takes an instance x and produce a label y. Sometimes we'll write h as $\hat{f}$ because we can look at the problem as looking for an estimate function. We are also going to represent the learner's output as $A(S)$ where S is the training data where $A()$ is the algorithm used to produce the prediction rules when the input is S.

5 **Data-generation model**: we assume that data are generated by some probability distribution, and then this vectors are labelled according to a specific function. We are going to call D the probability distribution over a set of instances X. This is **not known** to the learner. Then there's a labelling function $f : X \to Y$ that takes an instance and produce the **correct** label that the learner **don't know**. Therefore I know that $x_i : y_i = f(x_i)\ \forall i = 1, \ldots, m$. The data generation model tells me that for each point in S, I have obtained that point first sampling the features, then applying the label, and then trough the function f to obtain the correct label.

6 **Measure of success**: *error of classifier* is the probability that the learner does not predict the correct label on a random data point generated by the distribution D.

1.2 Loss function

Given a subset $A \subset X$ we are going to use $D(A)$ to express the probability of observing a certain point $x \in A$. In many cases, We are going to refer to A as an *event* and express it trough the function $\pi : X \to \{0, 1\}$, that is:

$$A = \{x \in X : \pi(x) = 1\}$$

In this case we can talk about the probability that a point taken by the distribution D is equal to

$$P_{x \sim D}\left[\pi(x)\right] = D(A)$$

With this equivalence we can define the **error of a prediction rule h** as the probability that if i take a point x according to the distribution D, the label predicted by the learner, *h(x)*, is different by the label that the "correct function" f assign to x. Mathematically we write:

$$L_{D,f}(h) = P_{x \sim D}\left[h(x) \neq f(x)\right] = D(\{x : h(x) \neq f(x)\}) \tag{1}$$

Where L stands for loss. This error can also be called generalization error, loss, true error, risk. Often f is obvious, so we can omit it.

1.3 Types of learning

We can classify the types of learning looking at what the learner receives in input. If the learner has access to both the vector x and the label y, we say that we are in a **supervised learning** problem. If the learner has access only to the vector but not to the label, we say that we're in a **unsupervised learning** problem. Furthermore, there can be different types of output, since Y can be discrete or continuous.

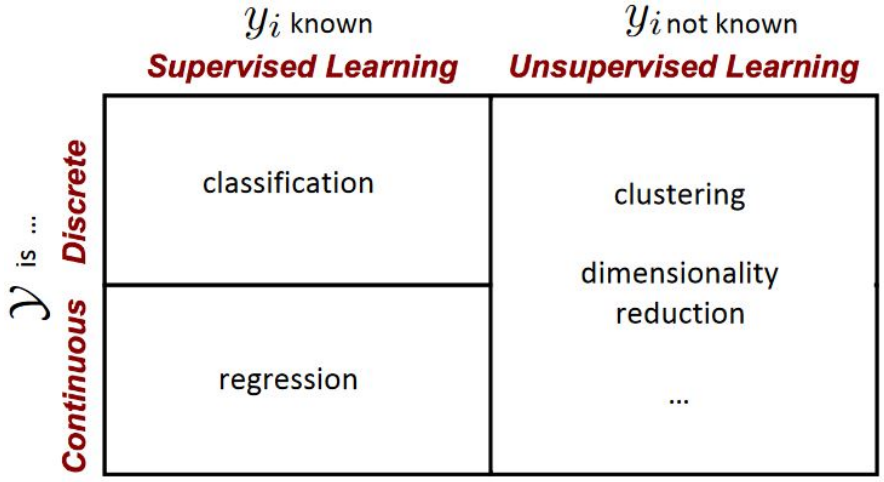


Figura 1: Different types of learning in ML

2 Probability Review

Definition 1

Probability Space A **probability space** has 3 component:

1 A sample space Z, which is the set of all possible outcomes of the random process modelled by the probability space;

2 A family F of sets representing the allowable events, where each event A is a subset of $Z : A \subseteq Z$;

3 A probability distribution $D : F \to [0, 1]$ that satisfies the following conditions:

- $D[Z] = 1$;
- Let $E_1, E_2, \ldots$ be any finite or countably infinite sequence of pairwise mutually disjoint events ($E_i \bigcap E_j = \emptyset$ for all $i \neq j$)

$$D\left[\bigcup_{i\geq 1} E_i\right] = \sum_{i\geq 1} D[E_i]$$

Distribution and probability We use $z \sim D$ to say that $z \in Z$ is sampled according to D. Given a function $f : Z \to \{true, false\}$ we can define the probability of *f(z)* as

$$P_{z\sim D}[f(z)] = D(\{z \in Z : f(z) = true\}) \tag{2}$$

In many cases, we express an event $A \subseteq Z$ using a function $\pi : Z \to \{true, false\}$, that is:

$$A = \{z \in Z : \pi(z) = true\} \tag{3}$$

where $\pi(z) = true$ if $z \in A$ and $\pi(z) = 0$ otherwise. In this case we have $P_{z\sim D}[f(z)] = D(A)$. For an event A we may use $P[A]$ instead of $P[\pi(z)]$ for the corresponding π. Also, we can use $\pi : Z \to \{0, 1\}$ instead of $\pi : Z \to \{true, false\}$

Definition 2

Independents events Two event E and F are independent ($E \perp F$) if and only if

$$P[E \cap F] = P[E] \cdot P[F]$$

More generally, events $E_1, E_2, \ldots, E_k$ are mutually independent if and only if for **any** subset $I \subseteq [1, k]$,

$$P\left[\bigcap_{i\in I} E_i\right] = \prod_{i\in I} P[E_i] \tag{4}$$

Definition 3

Random Variable A **(scalar) random variable** *X(z)* on a sample space Z is a real-valued function on Z; that is, $X : z \in Z \to \mathbb{R}$. We can divide it in 2 subcategories:

1 **Discrete random variable**: codomain is finite or countable.

- $p_X(x) = P[X = x]$ is the probability mass function (PMF);
- $F_X(x) = P[X \geq x] = \sum_{k \geq x} p_x(k)$ is the cumulative distribution function CDF.

2 **Continuous random variable**: codomain is continuous.

- $F_X(x) = P[X \geq x]$ is the cumulative distribution function or Distribution;
- $p_X(x) = \frac{dF_x(x)}{dx}$ is the probability distribution function;
- $F_X(x) = \int_{-\infty}^{x} p_x(a)da$

Definition 4

Vector value R.V The vector $\overrightarrow{X}$ represent a random vector in the sense that $X_1, X_2, \ldots,$ are random variables.

In particular, we can consider as example: $X = \begin{bmatrix} X_1 \\ X_2 \end{bmatrix}$
If X_1, X_2 are discrete random variable, we can write

$$p_X(x) = p_{X_1,X_2}(x_1, x_2) = P_X[X_1 = x_1, X_2 = x_2]$$

While if X_1, X_2 are continuous random variable, we write

$$F_X(x) = F_{X_1,X_2}(x_1, x_2) = P_X[X_1 \geq x_1, X_2 \geq x_2]$$

The arguments in P_X are called **joint events**.

Definition 5

Independence Two discrete random variables X and Y are independent $(X \cap Y)$ if and only if

$$P((X = x) \cap (Y = y)) = P(X = x) \cdot (Y = y)) \quad \forall x, y \tag{5}$$

Similarly, discrete random variables $X_1, X_2, \ldots, X_k$ are mutually independent if and only if for **any** subset $I \subseteq [1, k]$ and any values $x_i, i \in I$,

$$P_X(x) = \prod_{i \in I} P(X_i = x_i) = \prod_{i \in I} p_{X_i}(x_i) \tag{6}$$

The continuous random variables X_1 and X_2 are independent if and only if for all x_1, x_2,

$$F_{X_1,X_2}(x_1, x_2) = F_{X_1}(x_1)F_{X_2}(x_2) \tag{7}$$

In general, for $X_1, X_2, \ldots, X_n$:

$$F_X(x) = \prod_{i=1}^{n} F_{X_i}(x_i) \tag{8}$$

Definition 6

Expected Value and Moments The **expectation** of a random variable X is:

- If X is discrete, $E[X] = \sum_x xp_x(x)$
- If X is continuous, $E[X] = \int_{-\infty}^{+\infty} xp_x(x)dx$

Theorem 1. *Let $g(x)$ be a function of a random variable X. Then:*

- *If X is discrete, $E[g(X)] = \sum_x g(x)p_x(x)$*
- *If X is continuous, $E[g(X)] = \int_{-\infty}^{+\infty} g(x)p_x(x)dx$*

For a random variable $\mathbf{X}$, we define

- **Mean**: $m_X = E[X]$;
- **Variance**: $\sigma_X^2 = E[(X - m_X)^2] = E[X^2] - m_X^2 = Var[x]$;
- k-th moment: $E[X^k]$.

In particular, for a vector valued R.V $X \in \mathbb{R}^n$ we define **expectation** as $E[X] = \begin{bmatrix} m_{X_1} \\ \cdots \\ \cdots \\ m_{X_n} \end{bmatrix}$

Instead of variance, we have a **covariance matrix**:

$$\Sigma = E[(X - m_X)(X - m_X)^T] = \begin{bmatrix} \sigma_{X_1}^2 & \sigma_{X_1,X_2} & \cdots & \sigma_{X_1,X_n} \\ \sigma_{X_2,X_1} & \sigma_{X_2}^2 & \vdots & \sigma_{X_2,X_n} \\ \vdots & \vdots & \vdots & \vdots \\ \sigma_{X_n,X_1} & \sigma_{X_n,X_2} & \cdots & \sigma_{X_n}^2 \end{bmatrix}$$

where $\sigma_{X_i,X_j} = Cov(X_i, X_j) = E[(X_i - m_{X_i})(X_j - m_{X_j})] =$ *covariance of* X_i, X_j

Theorem 2. *If X_1 and X_2 are independent, then the covariance $\sigma_{X_1,X_2} = 0$*

Theorem 3. Properties of Mean, Variance, ... *Some useful properties of Mean, Variance and expectation are:*

$E[X_1 + X_2] = E[X_1] + E[X_2]$;

- $Var[aX + b] = a^2 Var[X]$;
- $Var[X_1 + X_2] = Var[X_1] + Var[X_2] + 2\sigma_{X_1,X_2}$

Corollary 1

If $\sigma_{X_1,X_2} = 0$ then $Var[X_1 + X_2] = Var[X_1] + Var[X_2]$.

Definition 7

Conditional Probability Given two events A, B, we define the **conditional probability** as

$$P[A|B] = \frac{P[A \cap B]}{P[B]} \tag{9}$$

This is well defined only if $P[B] > 0$.

Example. *Consider an event A. $X_1, \ldots, X_n$ are dependent and identically distributed random variables that are indicator functions:*

$$X_i(z) = \begin{cases} 1 & \text{if } z \in A \\ 0 & \text{if } z \notin A \end{cases}$$

The sum of those R.V is:

$$S_n = \sum_{i=1}^{n} X_i$$

so we can define the relative frequency of the event A as the function:

$$f_n(A) = \frac{S_n}{n}$$

This represent how many times i have "seen" the events A in the n times.

Example. *Let's consider coin flips as events, and A as "the result of the throw being head". Then each X_i is a Bernoulli R.V of parameter p: $X_i \sim B(p)$*

$$p = P[X_i = 1] = P[A \in Z]$$

Then $S_n = \sum_{i=1}^{n} X_i$ is a binomial R.V of parameters n, p: $S_n \sim Bin(n, p)$:

$$P[S_n = k] = \binom{n}{k} p^k (1-p)^{n-k}$$

Then $E[S_n] = np$ and $Var[S_n] = np(1-p)$

Theorem 4. Chebyshev's inequality *Let X be a r.v with $E[X] = \mu$ and $Var[X] = \sigma^2$. Then:*

$$P[|X - \mu|] > \varepsilon] \geq \frac{\sigma^2}{\varepsilon^2} \tag{10}$$

Therefore we can say that

$$P[|f_n(A) - p|] > \varepsilon \leq \frac{1(1-p)}{n\varepsilon^2} \tag{11}$$

This gives us how close we can be with some probability given the number of trials we have done. When n tends to infinity, we have that the relative frequency converge to p:

$$\lim_{n \to +\infty} f_n(A) = p \tag{12}$$

2.1 Intermission

We can now enunciate the **Law of the large numbers**:

Theorem 5. Law of large numbers *Let X_i, $i = 1, \dots, n$ be i.i.d with $E[X_i] = \mu$ and $Var[X_i] = \sigma^2 < +\infty$. Then*

$$\lim_{n \to +\infty} P\left[|\frac{1}{n} \sum_i X_i - \mu| > \varepsilon \right] = 0$$

That means I converge to the true value with my estimate.

Example.

- ***Remark 1:*** $\lim_{n \to +\infty} f_n(A) = P[A]$
- ***Remark 2:***

$$\frac{P[A \cap B]}{P[B]} = \lim_{n \to +\infty} \frac{f_n(A \cap pB)}{f_n(B)}$$

In particular

$$\frac{f_n(A \cap B)}{f_n(B)} = \frac{S_n(A \cap B}{S_n(B)}$$

This represent the fraction of times $A \cap B$ happens among those in which B happens.

Thus, we can define as **Conditional probability** how frequently we expect to see A when B happens:

Theorem 6. Conditional probability *Given two events A, B, we can say that*

$$P[A|B] = \frac{P[A \cap B]}{P[B]}$$

This is well defined only if $P[B] > 0$.

Using this enunciate we can write down the **Bayes rule**:

Theorem 7. Bayes rule

$$P[A|B] = \frac{P[B|A]P[A]}{P[B]}$$

This is useful because it gives me a way to compute the probability of an event A given that B has happened. I can estimate A given B using the single event probability and the probability of B given A.
We can enunciate the Law of total probability as:

Theorem 8. Law of total probability *Let $C_1, C_2, \dots, C_n$ be a partition of Ω, therefore:*
$\bigcup_{i=1}^{n} C_i = \Omega$

- $C_i \cap C_j = \emptyset$

Then for all $A \subset \Omega$:

$$P[A] = \sum_{i=1}^{n} P[A|C_i]P[C_i]$$

3 Learning Model

We have already explained what is a *Learner* [chapter 1.1] and the *Loss function* [chapter 1.2]. Let's now define the

3.1 Empirical Risk Minimization

We define the learner's output as $h_s : X \to Y$. Our goal is trying to find the h_s that minimize the generalization error $L_{D,f}(h)$. Such generalization error is unknown. The idea is to use the error on the training data as a substitute on the generalization error. What i see on the training data will be similar to the error i will make on the general case. The **error on the training data** (also called empirical error or empirical risk), is defined as:

$$L_s(h) = \frac{|\{i : h(x_i) \neq y_i, 1 \leq i \leq m\}|}{m} \tag{13}$$

This idea is the so called **Empirical risk minimization** (ERM), which produce in output the function h that minimize the training error $L_s(h)$. If we are fitting the data way too accurately on the training set, we might have issues on the test data. This phenomena is called **overfitting**.

Hypothesis Class In order to have ERM to provide good results, we have to decide a "type of function", and then, provide we have enough data, ERM is a good way to pick a class in that model. ERM is good only when we are restricting the class of functions we can use. We are going to apply ERM over a restricted set of hypothesis H also called *hypothesis class* from which we will pick functions. Each $h \in H$ is a function $h : X \to Y$. Our Empirical Risk Minimizer Learner is going to produce:

$$ERM_H \in argmin_{h \in H} L_s(h) \tag{14}$$

How can we define a good hypothesis class H? Let's start by finding hypothesis classes. That means that the cardinality of H is finite $|H| < +\infty$. Let h_s be the output of $ERM_h(S)$, i.e

$$h_s \in argmin_{h \in H} L_s(h)$$

. We now need some assumptions in order to "learn":

- (**Realizability:**) There exist a function $h^* \in H$ such that $L_{D,f}(h^*) = 0$. This assumption is going to be removed later;
- (**i.i.d:**) The examples on the training set are independent and identically distributed according to D, that is $S \sim D^m$.

Whatever the training set S, the training error of h^* is going to be zero. Then it's generalization error is also going to be zero. What we care about is to understand if we can find this h^* on the data. This is impossible, because even if it can look good on our data, it can't look as good on term of generalization error. However, if we have enough data, it will not be too far from h^*. This is captured from the so called framework **PAC learning**.

3.2 PAC Learning (simplified)

Since the data gets generated from probabilities, it's easy to build a data distribution such that a function h will look as good as h^* on the data, but will not look as good in term of generalization error. Then the best we can do is to build a function that is approximately correct. Since the data get generated from a probabilities, sometimes the function i will get will not look as good. In the framework **Probably Approximately Correct**(PAC) we are going to capture those two parameters:

- *accuracy parameter* ε: we are satisfied with a good h_s such that $L_{D,f}(h_s) < \varepsilon$;
- *confidence parameter* δ: we want h_s to be a good hypothesis with probability $\geq 1 - \delta$.

Under this framework the result we are going to prove is the following:

Theorem 9. *Let H be a finite hypothesis class. Let $\delta \in (0,1)$, $\varepsilon \in (0,1)$, and $m \in N$ such that*

$$m \geq \frac{log(|H|/\delta)}{\varepsilon}$$

Then for any f and D or which the realizability assumption holds, with probability $\geq 1 - \delta$ we have that for every ERM hypothesis h_s it holds that

$$L_{D,f}(h_s) \leq \varepsilon$$

What this theorem is telling us is that if i have a finite hypothesis class and the assumptions holds, then for any way we generate the data and for any function we use to assign the labels, if the number of samples is enough big, then i can use the ERM procedure to find a good hypothesis, which is at most ε. Let's now see the proof of the theorem we just enunciated.

Dimostrazione. Let $S|x = (x_1, \ldots, x_n)$ be the instances of the training set. We want an upper bound to the probability

$$D^m\left(\{S|x : L_{D,f}(h_s) > \varepsilon\}\right)$$

Let H_b be the bad hypothesis:

$$H_b = \{h \in H : L_{D,f}(h) > \varepsilon\}$$

and M_s as the misleading samples:

$$M_s = \{S|x : \exists h \in H_b, L_s(h) = 0\}$$

Because we have the realizability assumption $L_s(h_s) = 0 \Rightarrow L_{D,f}(h_s) > \varepsilon$ only if for some hypothesis $h \in H_b$ we have $L_s(h) = 0$. That is, our training data must be in the set of misleading samples:

$$\{S|x : L_{D,f}(h_s) > \varepsilon\} \subseteq M_s$$

Note that M can be rewritten as

$$M = \bigcup_{h \in H} \{S|x : L_s(h) = 0\}$$

Hence

$$D^m\left(\{S|x : L_{D,f}(h_s) > \varepsilon\}\right) \leq D^m(M)$$

$$\Rightarrow D^m(\bigcup_{h\in H_b}\{S|x : L_s(h) = 0\}) =_{(union\ bound)} \sum_{h\in H_b} D^m(\{S|x : L_s(h) = 0\}) \quad (*)$$

Now fix $h \in H_b$. If We look at

$$L_s(h) = 0 \Leftrightarrow \forall i, h(x_i) = f(\overrightarrow{x}_i)$$

Therefore,

$$D^m(\{S|x : L_s(h) = 0\} = D^m(\{S|x : \forall i, h(\overrightarrow{x}_i) = f(\overrightarrow{x}_i) =$$

$$[\text{i.i.d assumption}] \prod_{i=1}^{m} D^m(\{\overrightarrow{x}_i : h(\overrightarrow{x}_i) = f(\overrightarrow{x}_i)\}) \quad (**)$$

Let's consider i between $1 \leq i \leq m$:

$$P(\{\overrightarrow{x}_i : h(\overrightarrow{x}_i) = f(\overrightarrow{x}_i)\}) = 1 - L_{D,f}(h) \quad (\text{from the definition of } L_{D,f}(h))$$

$$1 - L_{D,f} \leq 1 - \varepsilon(\text{because } h \in H_b) \leq e^{-\varepsilon} \quad (e^{-x} \leq 1 - x \text{ by tayolor expansion})$$

Combining with $(**)$ we obtain

$$D^m(\{S|x : L_s(h) = 0\}) \leq \prod e^{-\varepsilon} = e^{-\varepsilon m}$$

Combining with $(*)$

$$D^m(\{S_i : L_{D,f}\}) > \varepsilon\}) \leq |H_b|e^{-\varepsilon m} \leq |H|e^{-\varepsilon m}$$

By the choice of m the theorem follows. □

Let's see a simple exercise.

Exercise 3.1. *Let's assume we have the following training set S*

i	x_1	x_2	y_i
1	-3	4	1
2	2	-3	-1
3	-3	-4	-1
4	1	1.5	1

Tabella 1: Training set of exercise 3.1

Let's assume we decide that the hypothesis set is $H = \{h_1, h_2, h_3h_4\}$, *with*

- h_1 = *sign*$(-x_1,-x_2)$
- h_2 = *sign*$(x_1,-x_2)$
- h_3 = *sign*$(-x_1,x_2)$
- h_4 = *sign*(x_1,x_2)

in addition, our machine learning algorithm uses the Empirical Risk Minimization (ERM) rule, and the $0-1$ *loss* $(h(x) \neq f(x))$. *Then*

1 What model h_s *is produced in output by your Machine Learning algorithm?*

i	$L_s(h_i)$
1	$\frac{(1+1+1+1)}{4} = 1$
2	$\frac{0}{4} = 0$
3	$\frac{(1+1+1+1)}{4} = 1$
4	$\frac{0}{4} = 0$

2 Let's assume that the realizability assumption holds. What can you say about the generalization error $L_{D,f}(h_s)$?

Let's start discussing with point [1]. Let's compute the training error $L_s(h_i)$ for each hypothesis: So, the ERM rule will produce either h_2 or h_4.
Let's focus now on point [2] of the exercise. From the proof of the theorem [9], we know that

$$Pr(L_{D,f}(h_s) > \varepsilon) \leq |H|e^{\varepsilon m}$$

If We fix y to be $\varepsilon = \frac{1}{m}\log_n(\frac{|H|}{\delta})$ that we assume < 1. For a given $\delta \in (0,1)$, we have that

$$Pr(L_{D,f}(h)) \leq \frac{1}{m}\log_n(\frac{|H|}{\delta}) \geq 1 - \delta$$

For example, if i choose $\delta = 0,1$ we have that with probability 0.9

$$L_{d,f}(h_s) = \frac{1}{4}h_1(\frac{4}{0.1}) \simeq 0.75$$

In practice, to do better i could only have increased the number of samples m, due to me not having control over the reset of the values.

Now let's enunciate the mathematical definition of PAC learning:

Theorem 10. PAC Learning *An hypothesis class H is PAC learnable if there exist a function $m_h : (0,1)^2 \to N$ and a learning algorithm such that for every $\sigma, \varepsilon \in (0,1)$, for every distribution D over X, and for every labelling function $f : X \to \{0,1\}$, if the realizability function hold with respect to H, D, f, then when running the learning algorithm on $m \geq m_H(\varepsilon, \sigma)$ independent and identically distributed examples generated by D and labelled by f, the algorithm returns a hypothesis h such that, with probability $\geq 1 - \sigma$ (over the choices of examples):*

$$L_D, f \leq \varepsilon$$

.

In particular, the function $m_h : (0,1)^2 \to N$ represent the sample complexity of learning H. The term m_H is the minimal integer that satisfies the requirements.

Corollary 2

Every finite hypothesis class is PAC learnable with sample complexity

$$m_H(\varepsilon, \delta) \leq \lceil \frac{\log(\frac{|H|}{\delta})}{\varepsilon} \rceil$$

How can we generalize those results? In practice, the realizability assumption that $\exists\ h^* \in H$ such that

$$L_{D,f}(h^*) = 0$$

may not hold. For this realizability assumption the label y must be fully determined by x. This is too strong in many applications. We can relax this hypothesis assuming that our probability distribution D generate not only the data X but also Y. That means for the same value of x i can have different values of y with different probabilities. So now D is a *joint distribution* over the domain points and the labels. We can imagine D divided in two components:

- The first component generate the features. Strictly speaking, it's the marginal distribution of D over the domain points;
- The second component consider the conditional distribution $D((x, y)|x)$. So now since x is fixed, this probability distribution is a probability over y.

This gives me different labels every time with different probabilities. Given x, label y is obtained according to a conditional probability $P[y|x]$. Now we don't have to consider that or our data exist a function that generate the true label. We now have to redefine what it is the generalization error and the empirical error.

- If D is the probability distribution that generate both the feature and the label, the **generalization error** is the defined as:

  $$L_D(h) = P_{(x,y)\sim D}[h(x) \neq y]$$

  As before D is not known to the learner; the learner only knows the training data S;

- Same as before:

  $$L_s(h) = \frac{|\{i : h(x_i) \neq y_i, 1 \leq i \leq m\}|}{m}$$

3.3 Optimal detector

The goal of the learner is to find a $h : X \longleftarrow Y$ that minimize L_D. Let's define the best predictor for a general case. Given a probability distribution D over $X = \{0, 1\}$, the best predictor is the **Bayes Optimal Predictor**

$$f_D(x) = \begin{cases} 1 \text{ if } P[y = 1|x] \geq \frac{1}{2} \\ 0 \text{ otherwise} \end{cases} \tag{15}$$

Proposition 1

For any classifier $g : X \longrightarrow \{0, 1\}$, it holds $L_D(f_D) \leq L_D(g)$.

We cannot use Bayes Optimal Predictor due to the fact that the learner can't know $P[y = 1|x]$. Let's now consider only predictors form a hypothesis class H. We are going to be ok with not finding the best predictor, but not being too far off. Thus, we can define:

Definition 8

An hypothesis class H is **agnostic PAC learnable** if there exist a function $m_h : (0,1)^2 \to N$ and a learning algorithm such that for every $\delta, \varepsilon \in (0,1)$, for every distribution D over $X \times Y$, when running the learning algorithm on $m \geq m_H(\varepsilon, \delta)$ i.i.d examples generated by D the algorithm returns a hypothesis h such that, with probability $\geq 1-\delta$ (over the choices of the m training examples):

$$L_D(h) \leq min_{h' \in H} L_D(h') + \varepsilon$$

.

3.4 Beyond Binary Classification

All the cases we have analized so far were in the binary classification learning model. We can defire more model based on different properties:

- **Multiclass classification**: Similar to before, but there are more than 2 labels (not only (0,1));
- **Regression** : here the label set is $Y = \mathbb{R}$.

Let's examine in particular the regression model.

Regression When using a regression learning model, the domain set X is usually $\mathbb{R}^p$ for some p. The target set Y is equal to $\mathbb{R}$. The training data are the same as before: $S = ((x_1, y_1), \ldots, (x_n, y_n))$. This goes also from the learner's output: $h : X \longleftrightarrow Y$. What change now is the loss function. Let's define a more generalized loss function:

Definition 9

Given any hypotheses set H and some domain Z, a loss function is any function

$$l : H \times Z \longrightarrow \mathbb{R}_+$$

In particular, the domain Z represent the pairs (x, y). We can now define the **generalization error** as the expected loss of a hypothesis $h \in H$ with respect to D over Z:

$$L_D(h) = E_{Z \sim D}\left[l(h, z)\right] \tag{16}$$

Before we were looking at the probability that the label predicted was different from the distribution, while now the loss function tells me how much did i lose by using my function h with respect to the distribution D. The **empirical risk** is then the expected loss over a given sample $S = (z_1, \ldots, z_m) \in Z^m$:

$$L_S(h) = \frac{1}{m} \sum_{i=1}^{m} l(h, z_i) \tag{17}$$

Let's now examine some common Loss functions.

0-1 loss Given $Z = X \times Y$, the 0-1 loss function is written as

$$l_{0-1}(h,(x,y)) = \begin{cases} 0 & \text{if } h(x) = y \\ 1 & \text{if } h(x) \neq y \end{cases} \tag{18}$$

This is commonly used in binary or multiclass classification.

Squared loss Given the correct value we take the square of the difference. Considering $Z = X \times Y$:

$$l_{sq}(h,(x,y)) = (h(x) - y)^2 \tag{19}$$

This is what we use in regression. In general the loss function picked depends on the application.

4 Uniform Convergence

We are now going to express the most general definition of PAC Learning, which is called **Agnostic PAC Learning**. This is the same as the previous one but now we use general loss functions.

Definition 10

Agnostic PAC Learning (general) An hypothesis class H is **agnostic PAC learnable** with respect to a set Z and a loss function $l : H \times Z \longrightarrow \mathbb{R}_+$ if there exist a function $m_h : (0,1)^2 \to N$ and a learning algorithm such that for every $\delta, \varepsilon \in (0,1)$, for every distribution D over Z, when running the learning algorithm on $m \geq m_H(\varepsilon, \delta)$ i.i.d examples generated by D the algorithm returns a hypothesis h such that, with probability $\geq 1 - \delta$ (over the choices of the m training examples):

$$L_D(h) \leq min_{h' \in H} L_D(h^{'}) + \varepsilon$$

where

$$L_D(h) = E_{z \sim D}\left[l(h,z)\right]$$

Let's now focus on the concept of **uniform convergence** is. This tells us that the training error of all the $h \in H$ are good approximation of the true risk. What i see in my data is very close to what i will see in data i haven't see yet. Then the best h given what i have will not be too bad for what i still have to see. In particular, we are going to define:

Definition 11

$\epsilon-$*representativeness* A training set S is called $\varepsilon-$representative if $\forall h \in H$,

$$|L_S(h) - L_D(h)| \leq \varepsilon$$

This is important for the next theorem.

Proposition 2

Assume that the training set S is $\frac{\varepsilon}{2}-$representative. Then, any output of $ERM_H(S)$ (i.e

any $h_S \in argmin_{h \in H} L_S(h))$ satisfies

$$L_D(h_S) \leq min_{h \in H} L_D(h) + \varepsilon$$

Dimostrazione. For every $h \in H$

$$L_D(h_S) \leq L_S(h_S) + \frac{\varepsilon}{2} \leq L_S(h) + \frac{\varepsilon}{2} \leq L_D(h) + \frac{\varepsilon}{2} + \frac{\varepsilon}{2} = L_D(h) + \varepsilon \tag{20}$$

□

This tells us that h_S is not too bad with respect to what we can do in our hypothesis class H. Therefore if my training set is $\frac{\varepsilon}{2}-$representative, then i can use ERM and get an hypothesis that is "good".
What i proved till now is that if i have a good training set, what i pick with ERM is good in sense of PAC learning. When is it that our samples is $\frac{\varepsilon}{2}-$representative?

Definition 12

Uniform convergence property A hypothesis class H has the **uniform convergence property** if there exists a function $m_H^{UC} : (0,1)^2 \longrightarrow N$ such that for every accuracy ε and confidence δ, $\varepsilon, \delta \in (0,1)$, and for every probability distribution D over Z, if S is a sample of $m \geq m_H^{UC}(\varepsilon, \delta)$ i.i.d examples drawn from D, then with probability $\geq 1 - \delta$, S is $\varepsilon-$representative.

How well the ERM does whenever we have a class which as the uniform convergence property?

Proposition 3

If a class H has the uniform convergence property with a function m_H^{UC} then the class is agnostic PAC learnable with the sample complexity $m_H(\varepsilon, \delta) \leq m_H^{UC}(\frac{\varepsilon}{2}, \delta)$. Furthermore, in that case the ERM_H paradigm is a successful agnostic PAC learner for H.

We are now going to prove that finite sets of hypotheses are agnostic PAC learnable under some restriction for the loss.

Proposition 4

Let H be a finite hypothesis class , let Z be a domain, and let $l : H \times Z \longrightarrow [0,1]$ be a loss function. Then:

- H enjoy the uniform convergence property with sample complexity

$$m_H^{UC}(\varepsilon, \delta) \leq \left\lceil \frac{log(\frac{2|H|}{\delta})}{2\epsilon^2} \right\rceil$$

- H is agnostically PAC learnable using the ERM algorithm with sample complexity

$$m_H(\varepsilon, \delta) \leq m_H^{UC}(\frac{\varepsilon}{2}, \delta) \leq \left\lceil \frac{2log(\frac{2|H|}{\delta})}{\epsilon^2} \right\rceil$$

Let's prove this proposition.

Dimostrazione. Let's fix $\varepsilon, \delta \in (0,1)$. We need a sample size M such hat for any data distribution D with probability $\geq 1-\delta$ (on the choice of $S = (z_1, \ldots, z_m)$ sampled i.i.d from D) we want to prove that for all $h \in H$,

$$|L_S(h) - L_D(h)| \leq \varepsilon$$

is true. That is

$$D^m\left(\{S : \forall h \in H, |L_S(h) - L_D(h)| \leq \varepsilon)\right\} \geq 1-\delta$$

Or equivalently

$$D^m\left(\{S : \forall h \in H, |L_S(h) - L_D(h)| > \varepsilon)\right\} < \delta \quad (*)$$

What we have is

$$\{S : \exists h \in H, |L_S(h) - L_D(h)| > \varepsilon\} = \bigcup_{h \in H} \{S : h, |L_S(h) - L_D(h)| > \varepsilon\}$$

If we look at $(*)$, by union bound is going to be

$$\leq \sum_{h \in H} D^m\left(\{S : |L_S(h) - L_D(h)| > \varepsilon)\right\}$$

How do we get a bound on every addend? We recall that the generalization error is $L_D(h) = E_{z\sim D}\left[l(h,z)\right]$, while the training error is $L_S(h) = \frac{1}{m}\sum_{i=1}^{,} l(h, z_i)$. Note that since each z_i is taken i.i.d for D, we have that

$$\Rightarrow E\left[l(h, z_i)\right] = L_D(h)$$

Therefore,

$$E\left[L_S(h)\right] = \frac{1}{m}\sum_{i=1}^{,} l(h, z_i) = \frac{1}{m}(mL_D(h)) = L_D(h)$$

$$\Rightarrow L_S(h) - L_D(h) = L_S(h) - E\left[L_S(h)\right]$$

We can define θ_i to be a random variable that takes the value $l(h, z_i)$. In particular, h is fixed since we are considering a specific $h \in H$, while z_i are sampled i.i.d from D. Therefore $\theta_1, \ldots, \theta_m$ are i.i.d. Note that $L_S(h) = \frac{1}{m}\sum_{i=1}^{m} \theta_i$. In addition $E\left[\theta_i\right] = \mu = L_D(h)$. Let's define for a moment what's called the **Hoeffding's inequality**:

Definition 13

Let $\theta_1, \ldots, \theta_m$ be a sequence of i.i.d random variables, and assume that $\forall i = 1, \ldots, m$, we have that $E\left[\theta_i\right] = \mu$, and $P\left[a \leq \theta_i \leq b\right] = 1$ (finite support). Then for any $\varepsilon > 0$ we can say that

$$P\left[|\frac{1}{m}\sum_{i=1}^{m} \theta_i - \mu| > \varepsilon\right] \leq 2e^{\frac{-2m\varepsilon^2}{(b-a)^2}} \quad (21)$$

Let's assume that $\ell : H \times Z \to [0,1] \Rightarrow \theta_i \in [0,1]$
By Hoeffding's inequality,

$$P^m\left(\{S : |L_S(h) - L_D(h)| > \varepsilon\}\right) = P\left[|\frac{1}{m}\sum_{i=1}^{m} \theta_i - \mu| > \varepsilon\right] \leq 2e^{\frac{-2m\varepsilon^2}{(1-0)^2}} = 2e^{-2m\varepsilon^2}$$

Combining with

$$(**) \quad D^m\left(\{S : h \in H, |L_S(h) - L_D(h)| > \varepsilon\}\right) =$$

$$\sum_{i=1}^{m} D^m\left(\{S : |L_S(h) - L_D(h)| > \varepsilon\}\right) \leq 2|H|e^{-2m\varepsilon^2} \leq (?)\delta$$

If we chose $m \geq \frac{log(\frac{2|H|}{\delta})}{2\varepsilon^2}$, then

$$\Rightarrow (**) \leq 2|H|e^{-2\varepsilon^2 \frac{log(\frac{2|H|}{\delta})}{2\varepsilon^2}} = 2|H|\frac{\delta}{2|H|} = \delta$$

□

5 Basics of statistics

Let X_i be a Gaussian (Normal) random variable

$$X_i \sim N(\mu, \sigma^2_{x_i}), X_i \in \mathbb{R}$$

The data generated from our Gaussian random variable have probabilities

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

We are interested in what's the most likely value for μ and which interval contains μ with high probability. Thus we call back the definition of *Confidence set*:

Definition 14

Confidence Set: We say that $I(x)$ is a **confidence interval** (for a parameter θ) of level $1-\alpha$, if

$$P\left[I(x) \in \theta\right] > \alpha$$

We would like to find a interval confidence that

- Is small (in term of area of $I(x)$);
- Contains θ with a high probability (small value of $\alpha > 0$).

In this way we are able to locate θ with *high precision* and *high confidence*. In particular, given some data for which we assume some model, and a parameter that we want to estimate, how do we get the *best* estimate of the parameter? To define the "best" we used (in lab 0)

5.1 Maximum Likelihood Estimation (MLE)

Maximum Likelihood Estimation is a statistical approach for finding the parameters that maximize the joint probability of a given data set assuming a specific parametric probability function. The idea is that MLE essentially assumes a *generative model* for the data. The general approach we have used is the following:

1 Given a training set $S((x_1, y_1), \ldots, (x_m, y_m))$, assume each (x_i, y_i) is i.i.d from some probability distribution of parameters θ;

2 Consider the likelihood of given parameters $P\left[S|\theta\right]$;

3 We calculate the log likelihood as $L(S,\theta) = \log(P[S|\theta])$;

4 We find the maximum likelihood estimator $\hat{\theta} = argmax_\theta L(S;\theta)$.

Let's now find the MLE for θ. We consider the log likelihood function (for continuous random variable):

$$L(x,\theta) = logp_x(x;\theta) = -\frac{m}{2}\log(2\pi\sigma^2) - \frac{1}{2}\sum_{i=1}^{m}\frac{(x_i-\theta)^2}{\sigma^2}$$

Then to find the maximum likelihood estimator $\hat{\theta_m}$ of θ we compute the derivative $\frac{d}{d\theta}L(x,\theta)$ and compare it to 0. Then the maximum likelihood estimator of θ equals to the simple mean

$$\hat{\theta_m} = argmax_\theta L(x,\theta) = \frac{1}{m}\sum_{i=1}^{m} X_i$$

We are now searching for a good confidence interval I for θ. We know that the samples $X_1,\ldots,X_m$ are i.i.d random variables $\sim N(\theta,\sigma^2)$. Therefore the sample mean $\hat{\theta_m}$ is also normal distributed:

$$\hat{\theta_m} = \frac{1}{m}\sum_{i=1}^{m} X_i \sim N\left(\theta, \frac{\sigma^2}{m}\right)$$

Based on this fact, it is easy to show that for any given confidence $1-\alpha$, the smallest confidence set for θ is symmetric and centered around the sample mean. Once I get the point estimates $\hat{\theta_m}$, the confidence interval for θ is given by:

$$I(x) = \left[\hat{\theta_m} - z_{\frac{\alpha}{2}}\frac{\sigma}{\sqrt{m}}, \hat{\theta_m} + z_{\frac{\alpha}{2}}\frac{\sigma}{\sqrt{m}}\right]$$

defined by the standard deviation (that we assume to know in this case) σ, the mean of the samples θ and $z_{\frac{\alpha}{2}}$ is the $1-\frac{\alpha}{2}$ percentile of a zero mean variance normal distribution $N(0,1)$, i.e such that for the random variable $Y \sim N(0,1)$ has probability:

$$P\left[Y \leq z_{\frac{\alpha}{2}}\right] = 1 - \frac{\alpha}{2} \Leftrightarrow P\left[Y \geq z_{\frac{\alpha}{2}}\right] = \frac{\alpha}{2}$$

Graphically, we obtain an estimate around which we define a symmetric interval. With high probability, the mean will be inside this interval. The lower α is, the larger the confidence interval will be. The number of points it's also an important parameter. The more data i have, the better i can locate the mean inside the interval.

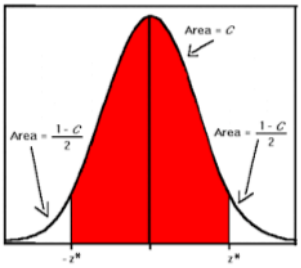


Figura 2: The smallest confidence set (the symmetric region around the origin) of level (or Confidence) $C = 1 - \alpha$

6 Linear Models

For **linear models**, we are going to consider that our input is $X = \mathbb{R}^d$. The function on our H are based on linear functions:

$$L_D = \{h_{w,b} : w \in \mathbb{R}^d, b \in \mathbb{R}\}$$

Where w are the weights and b is a real value called *bias*. In particular, each of this function is:

$$h_{w,b}(x) = \langle w, x \rangle + b = \left(\sum_{i=1}^{d} w_i x_i \right) + b$$

In particular, each member of L_D is a function $x \longrightarrow \langle w, x \rangle + b$. The linear models are going to be based on this functions. The hypothesis class is going to be obtain as $H : \phi \circ L_D$, where $\phi : \mathbb{R} \longrightarrow Y$. This function depends on the problem you are solving. Also, $h \in H$ is $h : \mathbb{R}^d \longrightarrow Y$. Let's now look at the equivalent notation. Given $x \in X$, $w \in \mathbb{R}^d$, $b \in \mathbb{R}$, we define:

- $w' = (b, w_1, w_2, \ldots, w_d) \in \mathbb{R}^{d+1}$
- $x' = (1, x_1, x_2, \ldots, x_d) \in \mathbb{R}^{d+1}$

Then:

$$h_{w,b}(x) = \langle w, x \rangle + b = \langle w', x' \rangle$$

That means we can consider bias term as part of w and assume $x = (1, x_1, \ldots, x_d)$ when needed, with $h_w(x) = \langle w, x \rangle$

6.1 Linear Classification

We talk about classification problem when the instance domain is $\mathbb{R}^d$ and the value we want to predict is discrete. For the moment we'll assume this is a binary value such that $Y = \{-1, 1\}$. (**0-1 loss**) We need to specify the Loss function and the algorithm we use to pick a good hypothesis. In particular, since we are using linear models, the hypothesis class H is given by sets called *halfspaces*, which are composition of a linear function with the *sign* function.

$$H_{S_d} = sign \circ L_d = \{x \longrightarrow sign(h_{w,b}(x)) : h_{w,b} \in L_d\} \tag{22}$$

One of this function takes a vector x and produce the sign of $h_{w,b}$, which is a set of a finite functions L_d. A graphic representation on $X = \mathbb{R}^2$ can be seen on picture [2]. When the bias

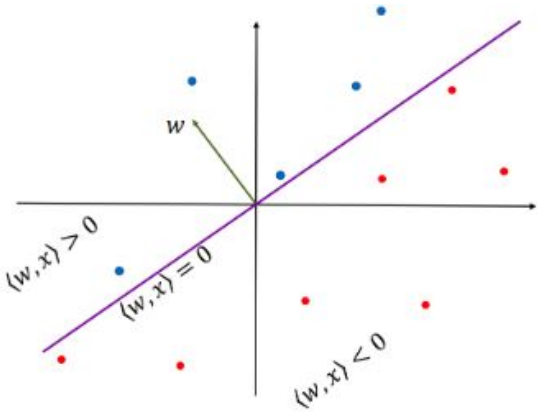


Figura 3: graphic representation on $X = \mathbb{R}^2$ of linear classification

is zero, the function will pass trough the origin. If i change w i change the slope of the line. The terms with positive and negative value of the inner product $\langle w, x\rangle$ will be divided into two different halfspaces. Note that the set of points that lie on the line satisfies the effect that the inner product is equal to zero. That means that if I represent the vector of weights w, it must be a vector orthogonal to the line. How can we pick a good hypothesis function in the set H? We can use ERM to pick the hypothesis that is going to be the model we learn. However this is not a well-specified algorithm. The idea is to start somewhere and improve the algorithm until the point it can't improve any more.

6.2 Perceptron Rule

This algorithm is based on the fact that if the product $y_i\langle w, x_i\rangle > 0 \ \forall i = 1, \ldots, m$, then all points are classified correctly by the model w. This is the realizability assumptions for the training set. A basic approach is exposed as pseudo-code below:

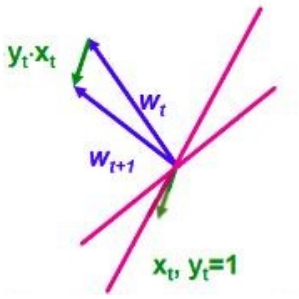
```
Input: training set (x_1, y_1), ..., (x_m, y_m)
Initialize: w^(1) = (0, ..., 0)
for t = 1, 2, ... do
        if ∃i such that y_i⟨w^(t), x_i⟩ ≤ 0 then
                w^(t+1) ⟵ w^(t) + y_i x_i ;
        else return w^(t) ;
```

We first set w to a certain value. Then the algorithm reiterate. In each iteration we check if there exist an index i such that has a value $w^{(i)}$ that makes a wrong decision. In that case, we update that to better fit the data.

Let's consider a more detailed interpretation of the update:

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

This update guides w to be more correct on (x_i, y_i). However there's no guaranteed that x_i will have a correct value. How does the perceptron perform with linearly separable data? Then exists an halfspace that separate the points with label -1 from the one with label +1. Then



Proposition 5

Assume that $(x_1, y_1), \ldots, (x_m, y_m)$ is linearly separable, then let:

- $B = min\{\|w\| : y_i\langle w, x_i\rangle \geq 1 \ \forall i, i = 1, \ldots, m\}$
- $R = max_i\{\|x_i\|\}$

Then the perceptron algorithm stops after at most $(RB)^2$ iterations (and when it stops it holds that $\forall i, i \in \{1, \ldots, m\}$: $y_i\langle w^{(t)}, x_i\rangle > 0$)

For separable data, the convergence is guaranteed and could even be multiple solutions. When the data are non linearly separable there is no halfspaces that makes no error. Then the algorithm won't stop. I should keep the best one stored. This is called *pocket-algorithm*.

6.3 Linear Models notebook and Residual Sum of Squares

This section refers to the linear models notebook seen in class and explain the theoretical assumptions behind the code. We are considering a problem of linear regression. Therefore $X = \mathbb{R}^d$, $Y = \mathbb{R}$ and the hypothesis class is

$$H_{reg} = L_d = \{x \longrightarrow \langle w, b\rangle + b : w \in \mathbb{R}^d, b \in \mathbb{R}^d\}$$

The loss function we are going to use is the *squared loss*

$$\ell(h(x, y)) = (h(x) - y)^2$$

While the training error is the **Mean-Squared Error** (MSE):

$$L_S(h) = \frac{1}{m}\sum_{i=1}^{m}(h(x_i) - y_i)^2$$

Training and Test set Given a dataset, we split the data that we are provided with in 2 parts. Given m total data, keep m_t data as training data, and $m_{test} := m - m_t$ for test data. For instance one can take $m_t = \frac{3}{4}m$ of the data as training, and $m_{test} = \frac{m}{4}$ as testing. Let us define

- S_t the training data set;
- S_{test} the testing data set.

The reason for this splitting is as follows:

- **Training Data**: The training data are used to compute the empirical loss

$$L_S(h) = \frac{1}{m_t}\sum_{z_i \in S_t}\ell(h, z_i)$$

  which is used to get h_S in a given model class $\mathcal{H}$. i.e.

$$h_S = \arg\min_{h\in\mathcal{H}} L_S(h)$$

- **Testing Data**: Last, the test data set can be used to estimate the performance of the chosen hypothesis h_S using:

$$L_{\mathcal{D}}(h_S) \simeq \frac{1}{m_{test}} \sum_{z_i \in S_{test}} \ell(h_S, z_i)$$

Data Normalization It's common practice in Machine Learning to scale the data independently so that it is centered (zero mean) and has standard deviation equal to 1. This helps in terms of numerical conditioning of the (inverse) problems of learning the model (the coefficients of the linear regression in this case), as well as to give the same scale to all the coefficients.

Model training The model is trained minimizing the empirical error

$$L_S(h) := \frac{1}{N_t} \sum_{z_i \in S_t} \ell(h, z_i)$$

When the loss function is the quadratic loss

$$\ell(h, z) := (y - h(x))^2$$

we define the Residual Sum of Squares (RSS) as

$$RSS(h) := \sum_{z_i \in S_t} \ell(h, z_i) = \sum_{z_i \in S_t} (y_i - h(x_i))^2$$

so that the training error becomes

$$L_S(h) = \frac{RSS(h)}{m_t}$$

We recall that, for linear models we have $h(x) =< w, x >$ and the Empirical error $L_S(h)$ can be written in terms of the vector of parameters w in the form

$$L_S(w) = \frac{1}{m_t} \|Y - Xw\|^2$$

where Y and X are the matrices whose $i-$th row are, respectively, the output data y_i and the input vectors $x_i^\top$.

The least squares solution is given by the expression

$$\hat{w} = \arg\min_w L_S(w) = (X^\top X)^{-1} X^\top Y$$

When the matrix X is not invertible (or even when it is invertible), the solution can be computed using the Moore-Penrose pseudonverse $(X^\top X)^\dagger$ of $(X^\top X)$

$$\hat{w} = (X^\top X)^\dagger X^\top Y$$

The Moore-Penrose pseudoinverse $A^\dagger$ of a matrix $A \in \mathbb{R}^{m \times n}$ can be expressed in terms of the Singular Value Decomposition (SVD) as follows: let $A \in \mathbb{R}^{m \times n}$ be of rank $r \leq \min(n, m)$ and let

$$A = USV^\top$$

be the singular value decomposition of A where

$$S = \text{diag}\{s_1, s_2, .., s_r\}$$

Then

$$A^\dagger = VS^{-1}U^\top$$

In practice some of the singular values may be very small (e.g. $< 1e-10$). Therefore it makes sense to first approximate the matrix A truncating the SVD and then using the pseudoinverse formula.

More specifically, let us postulate that, given a threshold T_h (e.g $= 1e-12$), we have $\sigma_i < T_h$, for $i = \hat{r}+1, .., r$. Then we can approximate (by SVD truncation) A using:

$$A = USV^\top = U\,\mathrm{diag}\{s_1, s_2, .., s_r\}\,V^\top \simeq \hat{A}_r = U\,\mathrm{diag}\{s_1, s_2, .., s_{\hat{r}}, 0, .., 0\}\,V^\top$$

So that

$$A^\dagger \simeq \hat{A}_r^\dagger := V\,\mathrm{diag}\{1/s_1, 1/s_2, .., 1/s_{\hat{r}}, 0, .., 0\}\,U^\top$$

7 Bias-Complexity Trade-off

Let's recap the goal of the *learning* in machine learning. Given a training set $S = ((x_1, y_1), \ldots, (x_m, y_m))$. Let's assume we have picked a certain loss function $l(h(x, y))$ which tells how much I lose when i predict a value for x using h instead of using the correct value y. We want a certain function $\hat{h}$ such that the generalization error $L_D(\hat{h})$ is small. I need to specify the algorithm A, that given S will produce $\hat{h} = A(S)$. In particular, the algorithm A comprises:

- The hypothesis set H;
- The procedure to pick $\hat{h} = A(S)$ from H.

Is there an algorithm that provide the best $\hat{h}$ for every data distribution? No, due to the

Theorem 11. No Free Lunch Theorem *Let* A *be any learning algorithm for the task of binary classification with respect to the 0-1 loss over a domain* X. *Let* m *be any number smaller than* $\frac{|H|}{2}$*, representing a training set size. Then, there exists a distribution* D *over* $X \times \{0, 1\}$ *such that:*

There exist a function $f : X \longrightarrow \{0, 1\}$ *with* $L_D(f) = 0$*;*

- *With probability at least* $\frac{1}{7}$ *over the choice* $S \sim D^m$ *we have that* $L_D(A(S)) \geq \frac{1}{8}$.

This means that for whatever algorithm we can think of, it's not going to be the answer for every ML problem, since what really matter is the way data are being generated. One consequence of this algorithm is the following:

Corollary 3

Let X be an infinite domain set and let H be the set of all functions from X to $\{0, 1\}$. Then, H is not PAC learnable.

This tells us that we need to pick our hypothesis h somehow. We need some prior knowledge about the data distribution to pick the good hypothesis set. In particular, we want an H that is large enough to contain a function h with a small generalization error, but we don't want it too large due to the no lunch function. Let's call h_S the hypothesis we pick using the ERM on the hypothesis class H. Then the generalization error for this hypothesis class can be rewritten as the sum of two components:

$$L_D(h_S) = \varepsilon_{app} + \varepsilon_{est} \tag{23}$$

where

- $\varepsilon_{app} = min_{h \in H} L_D(h)$ is the **approximation error**.
- $\varepsilon_{est} = L_D(h_S) - min_{h \in H} L_D(h)$ is the **estimation error**.

Thus the generalization error is composed by the sum of the minimum error i could achieve plus what's left. In particular:

Approximation error This derives from our choice of H. Once i chose H, there's no way I can avoid this error. To find an hypothesis h that minimizes this ε_{app} we need to search a large H.

Estimation error This error depends on the training set. In particular it derives from our inability of choosing the best hypothesis h_S in H. This error could be avoided if I had chosen the best hypothesis. However, since this is not possible, to decrease ε_{est} we need a low number of hypotheses in H so that the training error is a good estimate of the generalization error for all of them, thus we need a small H. If i look at the plot of the generalization error:

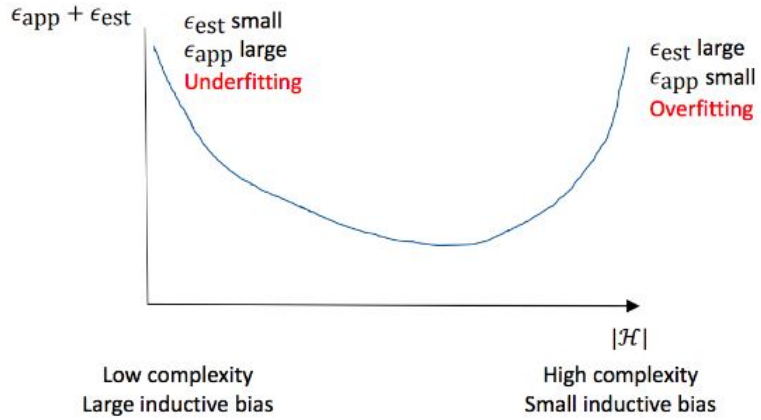


Figura 4: graphic representation of the generalization error over the set H

To estimate a good generalization error for a function h, we can use a **test set**. This is a new set of samples that we didn't used to train the data. The test must not be looked at until we have picked our final hypothesis. In practice, usually we have 1 set of samples and we split it into the training set and the test set.

7.1 Significance Testing for Linear Regression Coefficients

In order to study linear regression coefficients, we have to make some assumptions on the way our data are being generated. We assume that the data is being generated from a linear model with some noise. Thus, the noise ε is distributed according to a normal distribution with $\varepsilon \sim N(0, \sigma^2)$. We also assume that the variance σ^2 is known. The value of Y can be considered as the expectation

$$Y = E[Y|X_1, \ldots, X_d] + \varepsilon = b + \sum_{i=1}^{d} w_i X_i + \varepsilon$$

Once I have this data, I can derive the coefficients for the best linear model I can find using the data. For doing this, we can use least square to estimate the parameters $\hat{w_i}$ of w_i for all $i = 1, \ldots, d$. In this case, we can derive the bias explicitly as

$$\hat{b} = \frac{1}{m}\sum_{j=1}^{m} y_j - \sum_{i=1}^{d} \hat{w_i} \left(\frac{1}{m} \sum_{j=1}^{m} x_{ji} \right)$$

Where x_{ij} is the value of the $i-$th feature in the $j-$th sample.
In particular,the estimates we get from the linear regression are normally distributed:

$$\hat{w_i} = N(w_i, v_i\sigma^2)$$

where v_i is the $i-$th diagonal element of $(X^T X)^{-1}$, which is the matrix that combines the value of w_i to make the predictions. From my estimates of the w_i we can build confidence intervals. To do this we can consider the value $z_i = \frac{\hat{w_i}}{\sigma}$ and compute it as before.
However, the variance might be *unknown*. In those cases, we need to estimate it from the data. We can derive an estimation $\hat{\sigma^2}$ of σ^2 as

$$\hat{\sigma^2} = \frac{1}{m-d-1}\sum_{i=1}^{m}(y_i - \hat{y_i})^2 \tag{24}$$

where the predicted value is expressed as

$$\hat{y_i} = \hat{b} + \sum_{i=1}^{m} \hat{w_i} x_i$$

The corresponding normalized variable is

$$z_i = \frac{\hat{w_i}}{\hat{\sigma}\sqrt{v_i}}$$

where v_i as expressed before is the $i-$th diagonal element of the matrix $(X^T X)^{-1}$
Let's now briefly recap on what a **student distribution t** is.

Definition 15

The **Student t distribution** (sometimes just called the t distribution) is designed for use with small data sets for which the variance is unknown. It look almost identical to the normal distribution curve, only a bit "shorter and fatter". The larger the sample size, the more the t distribution looks like the normal distribution.

Now we can enunciate the following

Proposition 6

z_i follows a student t distribution with $m - d - 1$ degrees of freedom.

In particular, more degrees of freedom involve the t distribution to look like more as a normal distribution. One example is shown in figure [6]. We can derive then the confidence interval $1 - \alpha$ for w_i when we don't know the variance as:

$$\left(\hat{w_i} - t_{\alpha/2)}^{(m-d-1)}\sqrt{v_i}\hat{\sigma}, \hat{w_i} + t_{\alpha/2)}^{(m-d-1)}\sqrt{v_i}\hat{\sigma}\right) \tag{25}$$

where $t_{\alpha/2)}^{(m-d-1)}$ is the $1 - \frac{\alpha}{2}$-percentile of the Student's t distribution with $m - d - 1$ degrees of freedom.

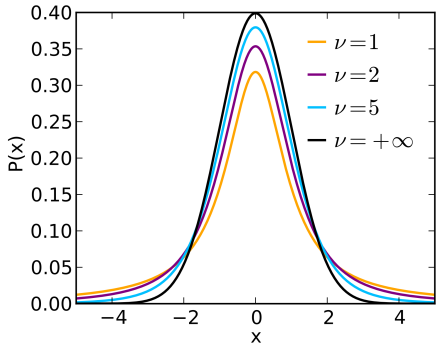


Figura 5: Student T distribution

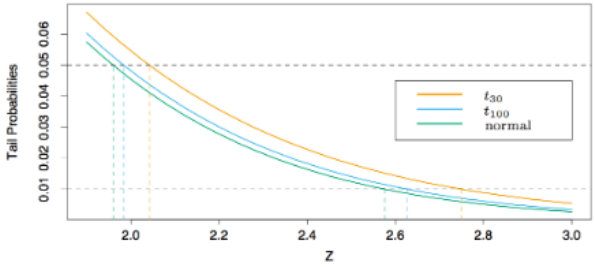


Figura 6: Tail probabilities for a student T distribution

8 Logistic regression

When we talk about **logistic regression** we want to learn a function $h : \mathbb{R}^d \to [0, 1]$. In general, this is useful because this value can be a probability. For example in binary classification, we can obtain the probability that a target is 1 by using $(Y = \{-1, 1\}) - h(x)$. Logistic regression can also be applied for multiclass classification, but for the sake of this course we'll restrain to binary classification.
The model (or hypothesis class) is defined as the composition $H : \phi_{sig} \circ L_d$, where $\phi_{sig} : \mathbb{R} \longrightarrow [0, 1]$ is the **sigmoid function**. For logistic regression, the sigmoid function ϕ_{sig} used is:

$$\phi_{sig}(z) = \frac{1}{1 - e^{-z}}$$

Therefore the hypothesis set is

$$H_{sig} = \phi_{sig} \circ L_d = \{x \longrightarrow \phi_{sig}(\langle w, x \rangle) : w \in \mathbb{R}^d\}$$

and a given hypothesis $h_w(x) \in H_{sig}$ is

$$h_w(x) = \frac{1}{1 + e^{-\langle w,x \rangle}}$$

This is still a linear model since the way we discriminate the features is still based on linear values. The main difference with a binary classification using halfspaces is obtained when $\langle w, x \rangle \approx 0$:

- With halfspaces our prediction is deterministically either 1 or -1;

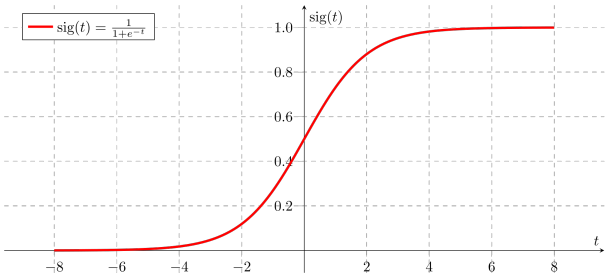


Figura 7: Sigmoid function

- With $\langle w, x\rangle \approx \frac{1}{2}$ we have uncertainty in predicted label.

Now we can define the loss function for the logistic regression. The loss function needs to capture how bad it is to predict $h_w(x) \in [0, 1]$ given that the true label is ± 1. In this case we would like to have

- Our function $h_w(x)$ to be "large" if the true label $y = 1$, so we have a high confidence in our prediction;
- Instead, if the label we observe is $y = -1$, we want to have a "large" $1 - h_w(x)$.

In particular, we can derive $1 - h_w(x)$ to obtain

$$1 - h_w(x) = 1 - \frac{1}{1 + e^{-\langle w,x\rangle}} = \frac{e^{-\langle w,x\rangle}}{1 + e^{-\langle w,x\rangle}} = \frac{1}{1 + e^{\langle w,x\rangle}}$$

We can observe that given the definition of $h_w(x)$, if $y = +1$, the probability of our error will be equal to $1 - h_w(x)$. Then a reasonable loss function for logistic regression increase monotonically with

$$\frac{1}{1 + e^{y\langle w,x\rangle}}$$

therefore it increase monotonically with $1 + e^{-\langle w,x\rangle}$.
Then the loss function for logistic regression in the end turns out to be just

$$\ell(h_w, (x, y)) = \log(1 + e^{-\langle w,x\rangle}) \tag{26}$$

Therefore, given a training set $S = ((x_1, y_1), \ldots, (x_m, y_m))$ to pick a good hypothesis we need to find the ERM. Such minimizer for logistic regression is

$$argmin_{w\in\mathbb{R}^d} \frac{1}{m} \sum_{i=1}^{m} \log(1 + e^{-y_i\langle w,x\rangle})$$

Note that the logistic loss function is a convex function, so it can be find a minimum. Therefore the ERM problem can be solved efficiently. The definition of the loss function may look a bit arbitrary. Actually ERM formulation is the same as the one arising from *Maximum Likelihood Estimation* for w under some generative model (see chapter 5.1 for the definition of MLE).

Now assume that $x_1, \ldots, x_m$ are fixed, the probability that x_i has label $y_i = 1$ is given by

$$h_w(x_i) = \frac{1}{1 + e^{-\langle w,x_i\rangle}}$$

with probability that x_i has label $y_i = -1$ is then

$$1 - h_w(x_i) = \frac{1}{1 + e^{\langle w, x_i \rangle}}$$

Then the likelihood for the training set S is given by

$$\prod_{i=1}^{m} \left(\frac{1}{1 + e^{-y_i \langle w, x_i \rangle}} \right) \tag{27}$$

Then for the log likelihood we have:

$$-\sum_{i=1}^{m} \log(1 + e^{-y_i \langle w, x_i \rangle})$$

Note that the maximum likelihood estimator for w is

$$argmax_{w \in \mathbb{R}^d} \left(-\sum_{i=1}^{m} \log(1 + e^{-y_i \langle w, x_i \rangle}) \right) = argmin_{w \in \mathbb{R}^d} \left(\sum_{i=1}^{m} \log(1 + e^{-y_i \langle w, x_i \rangle}) \right) \tag{28}$$

9 VC-Dimension

VC-Dimension capture the complexity of a model. It is very useful for design algorithms for big dataset since it says how much samples we need to have a good approximation of the dataset. Up till now we have seen that every finite $|H| < +\infty$ is PAC learnable. We want to extend this consideration also for $H : |H| = +\infty$. In particular, for the sake of this course, we'll focus on:

- Binary classification;
- 0-1 loss.

This restrictions are not necessary since similar results apply to other learning tasks and losses. Let's now define a **restriction** of an hypothesis class H to a set C of instances (or samples).

Definition 16

Restriction of H to C: Let H be a class of function from X to $\{0, 1\}$ and let $C = \{c_1, \ldots, c_m\} \subset X$ a subset of instances of X. The restriction H_C of H to C is defined as

$$H_C = \{[h(c_1), \ldots, h(c_m)] : h \in H\}$$

where we represent each function from C to $\{0, 1\}$ as a vector in $\{0, 1\}^{|C|}$.

Basically, H_C is the set of functions from C to $\{0, 1\}$ that i can derive from H. If we have a set C with cardinality m, we'll have a vector of dimension 2^m (in the binary case). This restriction set tells me how related the functions in my hypothesis set are. It's telling me, for a given set of points, in how many ways I can partition or classify them according to the $h \in H$. We're gonna define now the definition of **shattering**:

Definition 17

Shattering: Given $C \subset X$, we say that the hypothesis class H *shatters* C if H_C contains all $2^{|C|}$ functions from C to $\{0, 1\}$.

Shatter means that C can be partitioned in all possible ways. Now we can finally define

Definition 18

VC-Dimension The VC-dimension $VCdim(H)$ of an hypothesis class H is the maximal size of the set $C \subset X$ that can be shattered by H.

To know the VC-Dimension of H I need to find the largest set of C such that when i look at the restriction H_C it contains all the possible binary vectors. Some useful properties are:

- If H can shatter sets of arbitrarily large size then we can say that $VCdim(H) = +\infty$;
- If $|H| < +\infty$ then $VCdim(H) \leq \log_2(|H|)$.

The intuition behind VC-Dimension is that it measures the complexity of H. We can imagine it tells me how large a dataset that is perfectly classified using the functions in H can be.

9.1 Application

To prove that the $VCdim(H) = d$ with $d \in N$ we need to show that:

- $VCdim(H) \leq d$;
- $VCdim(H) \geq d$.

from the conjunction of those 2 points can only follows that $VCdim(H) = d$. This translate into:

1 There exists a set C of size d which is shattered by H;

2 *every set* of size $d + 1$ is not shattered by H.

Therefore, to prove that $VCdim(H) = d$ we need first find a d such that $VCdim(H) \leq d$. Then, if a set of size $d + 1$ don't shatters H, no other set $d + i$ is able to shatter it.
Let's now see some examples:

Threshold Functions We consider an hypothesis set $H = \{h_a : a \in \mathbb{R}\}$ where $h_a : \mathbb{R} \longrightarrow \{0, 1\}$ is

$$h_a(x) = \mathbb{1}[x < a] = \begin{cases} 1 & \text{if } x < a \\ 0 & \text{if } x \geq a \end{cases}$$

This can be visualized as if we got a straight line which describe our function in $\mathbb{R}$ where we can "move" the value a. Given two fixed values c_1, c_2, we can't have the combination $[0, 1]$ where $c_1 < a$ and $c_2 \leq a$. Therefore $VCdim(H) = 1$.

Intervals We consider an hypothesis set $H = \{h_{a,b} : a, b \in \mathbb{R}, a < b\}$ where $h_{a,b}(x) : \mathbb{R} \longrightarrow \{0, 1\}$ is

$$h_{a,b}(x) = \mathbb{1}[x \in (a, b)] = \begin{cases} 1 & \text{if } a < x < b \\ 0 & \text{otherwise} \end{cases}$$

This can be visualized as a straight line in $\mathbb{R}$ where a, b are the boundaries of an interval. In particular, we know that we can obtain all binary combinations for two fixed instances by moving a and b. However, if we consider three instances c_1, c_2, c_3, it's impossible to arrange a, b such that we obtain $[1, 0, 1]$. This is the combination where the more external samples c_1, c_3 are inside the interval but c_2 is outside. This is impossible, therefore $VCdim(H) = 2$.

Axis Aligned Rectangles We consider an hypothesis set $H = \{h_{a_1,a_2,b_1,b_2} : a_1, a_2, b_1, b_2 \in \mathbb{R}, a_1 \leq a_2, b_1 \leq b_2\}$.The function is described as follows

$$h_{a_1,a_2,b_1,b_2}(x) = \begin{cases} 1 & \text{if } a_1 \leq x_1 < a_2, b_1 \leq x_2 < b_2 \\ 0 & \text{otherwise} \end{cases}$$

This can be visualized on a Cartesian plane where if the point x_i is inside the rectangle delimited by the 4 vertices a_1, a_2, b_1, b_2, the function $h(x_i)$ is 1. We can easily prove that the $VCdim(H)$ is at least 4. However, for five points $(x_1, x_2, x_3, x_4, x_5)$ we can't consider the vector where one single point in the center between the 4 points is 0 while the other 4 points are inside the rectangle: $[1, 1, 0, 1, 1]$. Therefore, $VCdim(H) = 4$.

Convex sets We consider an hypothesis set H such that for $h \in H$,$h : \mathbb{R}^2 \longrightarrow \{0, 1\}$ with

$$h(x) = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{otherwise} \end{cases}$$

where S is a convex subset of $\mathbb{R}^2$.
In this case it can be proved that $VCdim(H) = +\infty$. We can imagine that given an unlimited set of points X in $\mathbb{R}^2$, we can always just consider straight line which goes from one point to another without interpolating the others. Therefore every binary vector is possible.

9.2 PAC Learnability of H

We can now enunciate the *Fundamental Theorems of Statistical Learning* which shows why finite VC-Dimension implies learnability and why infinite VC-Dimension implies non-learnability.

Theorem 12. *Let H be a hypothesis class of functions from a domain X to $\{0, 1\}$ and consider the 0-1 loss function. Assume that $VCdim(H) = d < +\infty$. Then there are absolute constants C_1, C_2 such that*

- *H has the uniform convergence property with sample complexity*

$$C_1 \frac{d + \log(1/\delta)}{\varepsilon^2} \leq m_H^{UC}(\varepsilon, \delta) \leq C_2 \frac{d + \log(1/\delta)}{\varepsilon^2} \tag{29}$$

- *H is agnostic PAC learnable with sample complexity*

$$C_1 \frac{d + \log(1/\delta)}{\varepsilon^2} \leq m_H(\varepsilon, \delta) \leq C_2 \frac{d + \log(1/\delta)}{\varepsilon^2} \tag{30}$$

This theorem explain that if the VC-Dimension is finite, with enough data i will find a good hypothesis in H. In particular, with enough data i can pick a good hypothesis looking at the training error. Equivalently:

Theorem 13. *Let H be an hypothesis class with VC-Dimension $VCdim(H) < +\infty$. Then, with probability $\geq 1 - \delta$ (over $S \sim D^m$) we have:*

$$\forall h \in H, L_D(h) \leq L_S(h) + C\sqrt{\frac{VCdim(H) + \log(1/\delta)}{2m}} \tag{31}$$

Dimostrazione. We need to prove 2 points to prove the theorem:

1

$$VCdim(H) \geq d$$

I need to find d points that are shattered by my hypothesis class so that my hypothesis applied to this d points give all the possible labellings. We consider the vector of the normal basis $\{\overrightarrow{e_i}, 1 \leq i \leq d\}$ where

$$\overrightarrow{e_i} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

We are going to show that this set is shattered by the halfspaces. For every labelling $(y_1, y_2, \ldots, y_d)$ let's set $\overrightarrow{w}$ as

$$\overrightarrow{w} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_d \end{bmatrix}$$

For every i, it holds that $\langle \overrightarrow{w}, \overrightarrow{e_i} \rangle = y_i$. Therefore

$$sign(\langle \overrightarrow{w}, \overrightarrow{e_i} \rangle) = y_i$$

2

$$VCdim(H) \leq d + 1$$

Let's consider a set of $d + 1$ points $\overrightarrow{x_1}, \overrightarrow{x_2}, \ldots, \overrightarrow{x_{d+1}}$. If i choose $d + 1$ points $\in \mathbb{R}^d$, those cannot be linearly independent. Therefore, $\exists a_1, \ldots, a_{d+1} \in \mathbb{R}$ such that not all the a_i are zero

$$\Rightarrow sum_{i=1}^{d+1} a_i \overrightarrow{x_i} = 0 \quad (*)$$

Let's define two sets of indexes

$$I = \{i : a_i > 0\}$$

$$J = \{j : a_j < o\}$$

Both of them can be the empty set $\emptyset$. Note that $I = \emptyset = J$ is impossible. We are going to look at three cases:

1 Assume that $I \neq \emptyset \neq J$. Then from $(*)$ follows that

$$\sum_{i \in I} a_i \overrightarrow{x_i} = \sum_{j \in J} -a_j \overrightarrow{x_j} = \sum_{j \in J} |a_j| \overrightarrow{x_j}$$

Let's assume that $\overrightarrow{x_1}, \ldots, \overrightarrow{x_{d+1}}$ is shattered by H. Therefore must exist $\overrightarrow{w}$ such that

$$\langle \overrightarrow{w}, \overrightarrow{x_i} \rangle > 0 \ \forall i \in I$$

$$\langle \overrightarrow{w}, \overrightarrow{x_j} \rangle < 0 \ \forall j \in J$$

Let's consider

$$0 < \sum_{i \in I} a_i \langle \overrightarrow{w}, \overrightarrow{x_i} \rangle \ \text{ (strictly positive)} = \langle \sum_{i \in I} a_i \overrightarrow{x_i}, \overrightarrow{w} \rangle =$$

$$\langle \sum_{j \in J} |a_j| \overrightarrow{x_j}, \overrightarrow{w} \rangle = \sum_{j \in J} |a_j| \langle \overrightarrow{x_j}, \overrightarrow{w} \rangle < 0 \ \text{ (strictly negative)}$$

But this is a contradiction! There cannot exist such $\overrightarrow{w}$.

2 We assume $J = \emptyset$ and $I \neq \emptyset$. The same steps of point [1] lead to

$$0 < \ldots \leq 0 \Rightarrow \text{ contradiction!}$$

3 We assume $I = \emptyset$ and $J \neq \emptyset$. The same steps of point [1] lead to

$$0 \leq \ldots < 0 \Rightarrow \text{ contradiction!}$$

□

If I have an hypothesis to bound a VC-Dimension, for every training error i see for my hypothesis, I know the maximum of the generalization error that such hypothesis could get. To find the hypothesis $h \in H$ that minimizes the training error (the upper bound to L_D) i need to use the ERM rule. This means that ERM is a good hypothesis for class of infinite cardinality but with bounded VC-Dimension. In the case of infinite VC-Dimension, we use the theorem that follows:

Theorem 14. *Let H be a class with $VCdim(H) = +\infty$. Then H is not PAC learnable.*

We can say that VC-dimension characterizes PAC learnable hypothesis classes.
Let's enunciate now an important theorem which tells us that the complexity of the model depends on the dimensionality of the spaces.

Theorem 15. *Let H be the class of (homogeneus) halfspaces in $\mathbb{R}^d$. Then $VCdim(H) = d$.*

Exercise 9.1. *Consider a linear regression problem, where $X = \mathbb{R}^d$ and $Y = \mathbb{R}$, with mean squared loss. The hypothesis set is the set of constant functions, that is $H = \{h_a : a \in \mathbb{R}\}$, where $h_a(x) = a$. Let $S = ((x_1, y_1) \ldots, (x_m, y_m))$ denote the training set.*

1 *Derive the hypothesis $h \in H$ that minimizes the training error.* If we compute the training error, for hypothesis $h_a(x)$, then for the definition of the training error:

$$L_s(h_a) = \frac{1}{m}\sum_{i=1}^{m}(h_a(x_i) - \overline{y_i})^2 = \frac{1}{m}\sum_{i=1}^{m}(a - \overline{y_i})^2$$

If we look at L_s as a function of a, we have something like a parabola with only positive values for the function. To find the best hypothesis, I need to look at how the training error changes in respect to the parameter that defines my function. By looking at this function, in term of the parameter a is a polynomial of degree 2. To compute the best hypothesis, i consider the derivative of L_s with respect to a and set it to zero:

$$\frac{dL_s(h_a)}{da} = \frac{1}{m}\sum_{i=1}^{m}2(a - y_i) = 0 \Leftrightarrow \sum_{i=1}^{m}(a - y_i) = 0 \Leftrightarrow ma - \sum_{i=1}^{m}y_i = 0 \Longleftrightarrow a = \frac{\sum_{i=0}^{m} y_i}{m}$$

2 *Use the result above to explain why, for a given hypothesis $\hat{h}$ from the set of all linear models, the coefficient of determination*

$$R^2 = 1 - \frac{\sum_{i=1}^{m}(\hat{h}(x_i) - y_i)^2}{\sum_{i=1}^{m}(y_i - \overline{y_i})^2}$$

where $\overline{y}$ is the average of the y_i, $i = 1, \ldots, m$ is a measure of how well $\hat{h}$ performs (on the training set). If I look at the definition of R^2, i can see that

$$\frac{\sum_{i=1}^{m}(\hat{h}(x_i) - y_i)^2}{\sum_{i=1}^{m}(y_i - \overline{y_i})^2}$$

is the error of $\hat{h}$ relative to the error of the best naive predictor. That does not depends on the role of the feature x

Exercise 9.2. *Your friend has developed a new machine learning algorithm for binary classification with 0-1 loss and tells you that it achieves a generalization error of only 0.05. However, when you look at the learning problem he is working on, you find out that $Pr_D[y = 1] = 0.95$.*

1 *Assume that $Pr_D[y = \ell] = p_\ell$. Derive the generalization error of the (dumb) hypothesis/model that always predict ℓ.* We are considering the hypothesis $h_\ell(x) = \ell \ \forall x$. The generalization error is by definition

$$L_D(h) = E_{(x,y)\sim D}[\ell(h(x, y))] = 0 \cdot Pr_{(x,y)\sim D}[h(x) = y] + 1 \cdot Pr_{(x,y)\sim D}[h(x) \neq y] = Pr_{(x,y)\sim D}[h(x) \neq y] = 1 - p_\ell \quad (32)$$

If I take a dumb hypothesis that always predict a label independently by the data, using the (0-1) loss, the generalization error will always be $1 - p_\ell$

2 *Use the result above to decide if your friend's algorithm has learned something or not.* The dumb model achieves a generalization error of $1 - Pr_D[y = 1] = 0.05$. This is the same as your friend 's algorithm. Then, we can affirm that you friend algorithm has not learned the relation between x and y, but only that the $Pr_D[y = 1] = 0.95$. This is trivial to learn with enough data. Therefore, your friend algorithm has not learned anything.

Exercise 9.3. *Consider the classification problem with* $X = \mathbb{R}^2$, $Y = [0, 1]$. *Consider the hypothesis class* $H = \{h_{(c,a)}, c \in \mathbb{R}^2, a \in \mathbb{R}\}$ *with*

$$h_{(c,a)}(x) = \begin{cases} 1 & \text{if } \|x - c\| \leq a \\ 0 & \text{otherwise} \end{cases}$$

find the VCdimension of H. We note that our x are circles in the $\mathbb{R}^2$ spaces. We can find that it's impossible to shatter 4 points in any possible configuration (think point in a line with 0101 combination). Also, we note that instead every combination is possible with $VCdim(H) \leq 3$. Therefore $VCdim(H) = 3$.

10 Model selection and validation

Model selection is a problem that recurs frequently. When we have to solve a machine learning task there are different algorithm and classes we can use. Algorithm and classes have however parameters to set which can be critical to the performance of the problem. We are exploring two different approaches:

- **Validation**;
- **Structural risk minimization**.

10.1 Validation

The idea behind the validation is using new data to estimate the true error after picking an hypothesis using the training set. Let's assume we have picked a naive predictor h (for example by applying ERM on a H_d). Let $V = (x_1, y_1), \ldots, (x_m, y_m)$ be a set of m_v fresh samples from D and let

$$L_V(h) = \frac{1}{m_v} \sum_{i=1}^{m_v} \ell(h(x_i, y_i))$$

Assume that the loss function is in $[0, 1]$. Then by Hoeffding inequality we have the following

Proposition 7

For every $\delta \in (0, 1)$, with probability $\geq 1 - \delta$ (over the choice of the validation set V) we have

$$|L_V(h) - L_D(h)| \leq \sqrt{\frac{\log(2/\delta)}{2m_v}}$$

This gives me a bound on the generalization error of the hypothesis h that I've picked using the ERM from some class us once I use new data to estimate the generalization error. We've previously seen the VC-Dimension bound . Let's assume the VC-dimension of H_d is $VCdim(H_d)$.

Then, by the fundamental theorem of learning we can say that the generalization error is bounded by

$$L_D(h) \leq L_S(h) + \sqrt{C\frac{VCdim(H_d) + \log(1/\delta)}{2m}}$$

where C is a constant (usually 0.5). From previous proposition,

$$L_D(h) \leq L_V(h) + \sqrt{\frac{log(2/\delta)}{2m_v}}$$

This is telling us that by using a validation strategy I can get a better bound. If we pick $m_v \in \Theta(m)$, the second bound is more accurate. In practice, we have only one dataset that we split into:

- Training set;
- Hold out or validation set.

We can use the validation set for model selection. Assume we have $H = \bigcup_{i=1}^{r} H_i$. Given a training set S, let h_i be the hypothesis obtained by ERM rule from H_i using S, we want to pick a final hypothesis from $\{h_1, h_2, \ldots, h_r\}$. We can pick a validation set $V = (x_1, y_1), \ldots, (x_{m_v}, y_{m_v})$ with m_v fresh samples taken from D. Then we can choose our final hypothesis (or class or value of the parameter) from $\{h_1, h_2, \ldots, h_r\}$ by ERM over validation set. Assuming that the loss function is in $[0, 1]$. Then we have the following

Proposition 8

With probability $\geq 1 - \delta$ over the choice of V we have

$$\forall h \in \{h_1, h_2, \ldots, h_r\} : |L_D(h) - L_V(h)| \leq \sqrt{\frac{log(2r/\delta)}{2m_v}}$$

In general, when you have to chose a value for the parameter, you can use model-election curve.

Model Selection curve A model-election curve shows how the training error and validation error change as a function of the complexity of the model considered. The parameter is usually a way to set the complexity of the model. By plotting the training error and the validation error,

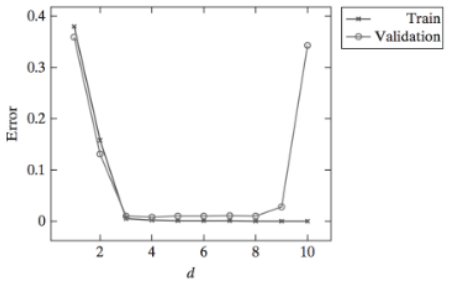


Figura 8: Model Selection curve example

I can observe that the training error, as expected, decreases with the increase of the complexity

of the model. In other words, a complex model gives me more "power" to find a good hypothesis for the data I have. Although, the validation error decreases up to some points but then it goes back up. In that case, we ended up in a situation where we over fitted the data. I need to balance the complexity of the class with the error that I observe. If we are considering one or more parameters with values in $\mathbb{R}$,

1 Start with a rough grid of values;

2 Plot the corresponding model-selection curve;

3 Based on the curve, zoom in to the correct regime;

4 Restart from [1] with a finer grid.

Note that the empirical risk on the validation set is not an estimate of the true risk, in particular if r is large. To estimate the true risk after doing the validation procedure, we need another fresh dataset.

Train-Validation-Test Split We assume the hypothesis as before: $H = \bigcup_{i=1}^{r} H_i$. The idea is to split the data in three parts instead of only two:

1 *Training set*: used to learn the best model h_i from each H_i;

2 *Validation set*: used to pick one hypothesis h from $\{h_1, h_2, \ldots, h_r\}$;

3 *Test set*: used to estimate the true risk $L_D(h)$.

The estimate from the test set has the guarantees provided by the proposition on estimate of $L_D(h)$ for 1 class. Note that the test set **is not involved** in the choice of h. Also, If after using the test set to estimate $L_D(h)$ we decide to choose another hypothesis (because we have seen the estimate of $L_D(h)$ from the test set), we **cannot use** the test set again to estimate $L_D(h)$.

10.2 K-Fold cross validation

If you don't have a lot of data, splitting the data in 3 parts isn't optimal. In such cases, we use the **k-fold cross validation** scheme. The k-fold works as follows:

1 Partition the set into k folds of size $\frac{m}{k}$;

2 for each fold:

- Train on the union of the other folds;
- estimate the error (for the learned hypothesis) from the fold.

3 Then the estimate of the error is the average of the estimated errors above.

Often cross validation is used for model selection: at the end, the final hypothesis is obtained from training on the entire training set. This method could also be done in a very skew way. It's possible to use all data but one samples for training, and the single sample remaining for learning. This technique is called **leave-one-out** cross validation. Here what I do is learning the model for one but all samples, and then repeat leaving out one samples at at the time. A simple implementation of the k-fold-cross is done as follows:

```
Input:
        training set (x_1, y_1), ..., (x_m, y_m)
        set of parameters Θ
        learning algorithm A
        integer k
Partition: S into S_1, S_2, ..., S_k
foreach θ in Θ:
        for i = 1, ..., k:
                h_{i,θ} = A(S_i, θ)
                error(θ) = (1/k) Σ_{i=1}^{k} L_{S_i}(h_{i,θ})
Output:
        θ* = argmin[error(θ)]
        h_{θ*} = A(S; θ*)
```

10.3 What should we do if learning fails?

This is a common problem in machine learning. For fails, we mean that after finding a good hypothesis on the training set, the results on the test set are bad. If this is the case, the first thing to do is to understand where the problem comes from. I can split the problem in two cases:

- $L_S(h)$ is large;
- $L_S(h)$ is small.

Based on those two cases, there are choice that i should make.

10.4 Large training error

When the $L_S(h)$ is large, it means that in H there is no good hypothesis in term of generalization error. Let's call $h^* \in argmin_{h \in H} L_D(h)$ the best hypothesis. We can write the training error of the hypothesis class i chose with the ERM (h_S) in this way:

$$L_S(h_S) = (L_S(h_S) - L_S(h^*)) + (L_S(h^*) - L_D(h^*)) + L_D(h^*)$$

where, in particular,

- $L_S(h_S) - L_S(h^*) < 0$;. Thus, the training error on the hypothesis i pick in my training set will surely be worse than using the best hypothesis.
- $L_S(h^*) \approx L_D(h^*)$. Thus, in general the training error of the best hypothesis is similar to the generalization error of the best hypothesis.

Therefore, if $L_S(h_S)$ is large, since is going to be equal to a negative component, an almost zero component and $L_D(h^*)$, then the generalization error of the best hypothesis $L_D(h^*)$ is going to be large. This is the same as saying that the approximation error is large. That means, if the training error is large we have no hope of finding a good hypothesis. We perhaps need to change our hypothesis class.

10.5 Small training error

Firstly, we need to understand if the approximation error $L_D(h^*)$ is large or not. To understand if this is the case we can use *learning curves*. A learning curve is the plot of training error and validation error when we run our algorithms on prefixes of the data of increasing size m. Here we can see how the training error and the generalization error change with respect to the amount of data I have. If i use the entire dataset, the training error is going to be small. Let's assume that the training error is zero. When I have fewer than m data points, the error will be the same, since it cannot decrease.

Case 1 In this case the validation error stay around the same. By looking at the fact that the error is not decreasing, then I don't have any evidence that the approximation error in this class is good. It may well be that even the best hypothesis in my H will have a high generalization error. In this case I should pick a more complex hypothesis class, so the approximation error

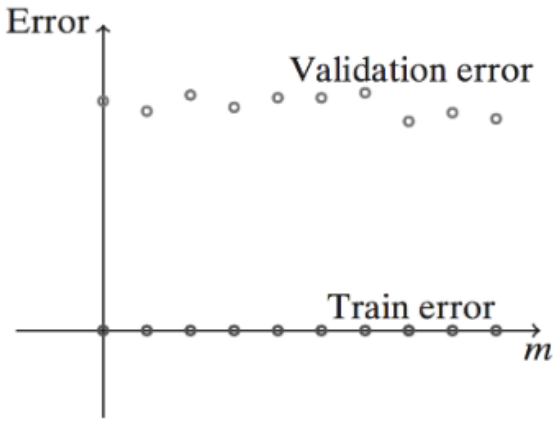


Figura 9: learning curve case 1

will decrease.

Case 2 In this case, the validation error seems to be decreasing. That means with the more data I get, the gap between the error on my data and the generalization error is decreasing. This means the error that i observe might be due that i cannot find a good hypothesis in H, however such hypothesis could exist. Therefore, the problem here is the lack of samples. The solution here is to increase the number of samples.

10.6 Summarizing

Some potential steps to follow if learning fail are the followings:

1 If you have parameters to tune, plot model-selection curve to make sure they are tuned appropriately;

2 If training error is excessively large (depends on the applciation) consider:

- enlarge H;

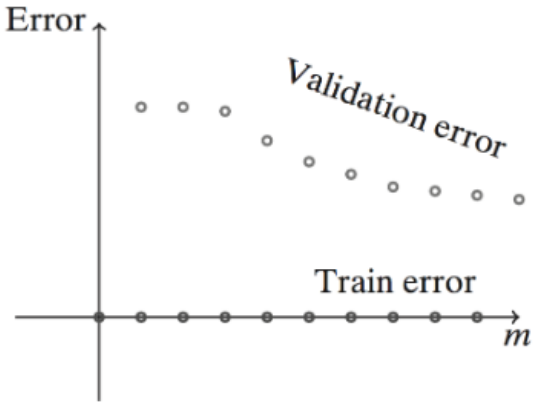


Figura 10: learning curve case 2

- change H;
- change feature representation of the data.

3 If the training error is small, use learning curves to understand whether the problem is the approximation error (or estimation error).

4 If the approximation error seems small:

- Get more data;
- Reduce the complexity of H.

5 If the approximation error seems large:

- change H;
- change feature representation of the data.

10.7 Structural Risk Minimization

Another way to make model selection is using **Structural Risk Minimization**. Up to now we have only looked at ERM, which pick the hypothesis that minimizes the training error. the SRM tries to balance directly between the bias and the complexity (approximation error and estimation error). Let's focus on binary classification on simplicity. Let $H = \bigcup_{n \in N} H_n$, where N is a finite set and each H_n has VC-dimension $VCdim(H)_n$. SRM follows a *bound minimization approach*. Let's express the bound: with probability $\geq 1 - \delta$, for every $n \in N$ and $h \in H_n$,

$$L_D(h) \leq L_S(h) + \sqrt{C \frac{log(1/\delta) + log|N| + VCdim(H_n)}{2m}}$$

where C is a constant. Now, the SRM rule depends on the class H_n i picked. Here we have that, picking a n and $h \in H_n$ minimizing

$$L_S(h) + \sqrt{C \frac{log(1/\delta) + log|N| + VCdim(H_n)}{2m}}$$

We must note that the upper bound might be pessimistic. Also, this is a specific form of SRM, more general schemes are possible. One similar approach is the so called

Minimum Description Length (MDL) For $h \in H$, let $|h|$ be the length of the representation of h. This depends on how I represent the hypothesis. The MDL says to pick the hypothesis that minimizes the training error plus a term that express the complexity of the hypothesis:

$$L_S(h) + \sqrt{C\frac{log(1/\delta) + |h|}{2m}}$$

This can be easily derived from the (general) SRM paradigm. Another related approach is the

Akaike Information Criterion (AIC) Let $d(h)$ be the number of parameters for hypothesis h. Then the AIC rule says to pick h minimizing

$$L_S(h) + 2d(h)$$

11 Regularization and Feature Selection

A learning paradigm is called

11.1 Regularized Loss Minimization

Assume for simplicity that h is defined by a vector $w = (w_1, \ldots, w_d)^T \in \mathbb{R}^d$. (i.e linear models). We have a regularizer function $f : \mathbb{R}^d \longrightarrow \mathbb{R}$. What the regularization function does is giving a weights to a picked hypothesis. The RLM paradigm says to pick h obtained as

$$argmin_w(L_S(h) + R(w))$$

The idea is that $R(w)$ is a measure of complexity of the hypothesis h defined by w. Basically, regularization balances between low empirical risk and less complex hypotheses. A common regularization function is the

Tikhonov regularization Given a regularization function $R(w) = \lambda\|w\|^2$, with $\lambda > 0 \in \mathbb{R}$, and ℓ_2 norm: $\|w\|^2 = \sum_{i=1}^d w_i^2$ The learning rule is to pick

$$A(S) = argmin_w(L_S(h) + \lambda\|w\|^2) \tag{33}$$

The intuition here is that $\|w\|^2$ measures the complexity of the hypothesis defined by w, while λ regulates the trade-off between the empirical risk $L_S(w)$ or overfitting and the complexity $\|w\|^2$ of the model we pick.

11.2 Ridge regression

We talk about **Ridge regression** when we do a task of linear regression with squared loss using Tikhonov regularization. In linear regression with squared loss, given a training set $S = ((x_1, y_1), \ldots, (x_m, y_m))$ with $x_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$, we want to find w which minimizes the empirical risk

$$w = argmin_w \frac{1}{m}\sum_{i=1}^{m}(\langle w, x_i\rangle - y_i)^2$$

equivalently, find w which minimizes the residual sum of squares $RSS(w)$:

$$w = arming_w RSS(w) = argmin_w \sum_{i=1}^{m}(\langle w, x_i\rangle - y_i)^2 \tag{34}$$

In Ridge regression, what you pick is the w that minimizes

$$w = argmin_w \left(\lambda \|w\|^2 + \sum_{i=1}^{m} (\langle w, x_i \rangle - y_i)^2 \right) \tag{35}$$

RSS matrix form Let

$$X = \begin{bmatrix} \dots & x_1 & \dots \\ \dots & x_2 & \dots \\ \dots & \vdots & \dots \\ \dots & x_m & \dots \end{bmatrix}$$

be the design matrix, and

$$y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$$

the labels vector. Then we can derive the matrix form of RSS as

$$\sum_{i=1}^{m} (\langle w, x_i \rangle - y_i)^2 = (y - Xw)^T (y - Xw)$$

Ridge regression matrix form While in linear regression we are searching for the w that minimizes the RSS

$$argmin_w (y - Xw)^T (y - Xw)$$

in Ridge regression instead we want to pick w that minimizes

$$argmin_w (\lambda \|w\|^2 + (y - Xw)^T (y - Xw))$$

where λ is the regularization factor. Therefore we minimizes the sum of the part which correspond to regularization plus the training error. Generally, we are picking a w that gives us a slightly higher training error but it's less complex in term of ℓ^2 norm. In order to find w that minimizes

$$f(w) = \lambda \|w\|^2 + (y - Xw)^T (y - Xw)$$

I need to compute the gradient of the objective function and compare it to 0:

$$\frac{\partial f(w)}{\partial w} = 2\lambda w - 2X^T (y - Xw) = 0$$

This is equivalent to

$$(\lambda I + X^T X) w = X^T y$$

where I is the identity matrix. Note that $X^T X$ is positive semidefinite, and λI is positive definite, therefore $(\lambda I + X^T X)$ is positive definite matrix, thus is also invertible. Then we can derive the Ridge regression solution as

$$w = (\lambda I + X^T X)^{-1} X^T y$$

Some Theoretical Guarantees about Tikhonov Regularization: It's possible to show that Tikhonov regularization makes the learner stable with respect to small perturbations of the training set, which in turn leads to small bounds on generalization error. Informally, an algorithm A is stable if a small change of the training data S will lead to a small change of its output hypothesis. This stability notions are related to the

Fitting-Stability Tradeoff By looking at the expectation of the generalization error over a data distribution considering a certain training set S and an algorithm A we can derive

$$\mathbb{E}_{\mathbb{S}}[L_D(A(S))] = \mathbb{E}_{\mathbb{S}}[L_S(A(S))] + \mathbb{E}_{\mathbb{S}}[L_D(A(S)) - L_S(A(S))]$$

In particular, $\mathbb{E}_{\mathbb{S}}[L_S(A(S))]$ describe how well the algorithm performs in terms of finding an hypothesis that fits the training set. Meanwhile, $\mathbb{E}_{\mathbb{S}}[L_D(A(S)) - L_S(A(S))]$ express the difference between the generalization and the training error. When this difference is high we are in the case of overfitting. In fact, by using the theorem of stability, you can find the bound to this term which relates to the stability of the algorithm A. In Tikhonov regularization, λ controls the trade-off between the two terms. By using regularization, I can control how much my hypothesis will change by changing the input a little bit.
Using the fitting-stability trade-off decomposition and the result for convex, Lipschitz losses we can prove that knowing the properties one can pick λ to guarantee that

$$\mathbb{E}_{\mathbb{S}}[L_D(A(S))] \leq min_{w \in H} L_D(w) + \sqrt{\frac{c}{m}}$$

where $c > 0$ depends on the parameters of the loss function. This is telling us that there is a way to pick λ to guarantee that what i get in expectation by learning a linear model with ridge regression on the data I have it's not too far from the best possible hypothesis, where best is meant in term of generalization error. In practice, we can pick λ by using validation.

11.3 ℓ_1 regularization

The regularization function using ℓ^1 regularization is

$$R(w) = \lambda \|w\|_1$$

As before, $\lambda > 0 \in \mathbb{R}$. In particular, we can rewrite the ℓ_1 norm as

$$\|w\|_1 = \sum_{i=1}^{d} |w_i|$$

Therefore as the learning rule we are going to pick w such that

$$A(S) = argmin_w (L_S(w) + \lambda \|w\|_1)$$

Once again, the intuition is that we can use the ℓ_1 norm to measure the complexity of the hypothesis defined by w. The larger the ℓ_1 norm, the more complex our vector w is. Again, λ regulates the tradeoff between the empirical risk or overfitting and the complexity of the model we pick.

11.4 LASSO regression

We talk about **LASSO regression** (Least Absolute Shrinkage and Selection Operator regression) when we do a task of linear regression with squared loss using ℓ_1 regularization. In LASSO regression we want to pick w such that

$$w = argmin_w \lambda \|w\|_1 + \sum_{i=1}^{m} (\langle w, x_i \rangle - y_i)^2$$

Here we cannot compute the gradient as we did in Ridge regression. However, since ℓ_1 norm is a convex function and squared loss is convex we can solve this problem efficiently.
Let's now look at an example of how our solution looks like when our domain $X = \mathbb{R}$. If you write down the LASSO for this problem, it turns out that w is a real value. Therefore the problem we are solving is to find the w that minimizes

$$argmin_{w \in \mathbb{R}} \left(\frac{1}{2m} \sum_{i=1}^{m} (x_i w - y_i)^2 + \lambda |w| \right)$$

This is equivalent to

$$argmin_{w \in \mathbb{R}} \left(\frac{1}{2} \left(\frac{1}{m} \sum_{i=1}^{m} x_i^2 \right) w^2 - \left(\frac{1}{m} \sum_{i=1}^{m} x_i y_i \right) w + \lambda |w| \right)$$

We can assume for simplicity that $\frac{1}{m} \sum_{i=1}^{m} x_i^2 = 1$, and let $\sum_{i=1}^{m} x_i y_i = \langle x, y \rangle$. Then the optimal solution is

$$w = sign(\langle x, y \rangle) \left[\frac{\langle x, y \rangle}{m} - \lambda \right]_+$$

where $[a]_+ = max\{a, 0\}$. For very small λ, w is going to be equal to the parameter. When i increase λ at some point the parameter will become zero (when $\lambda = \frac{\langle x, y \rangle}{m}$). At that point it λ always remain zero. By setting λ the weight in my model is either going to be zero or a value different from zero. In ℓ_2 regularization all coefficients are different from zero. By comparing the way LASSO and Ridge perform on the same dataset by varying lambda, what happens are very different behaviours. This can be seen in picture [11]. For lasso we can notice that as we vary $\frac{1}{\lambda}$ some of the features will be different from zero while other remain zero. LASSO is actually used to **select** some features.

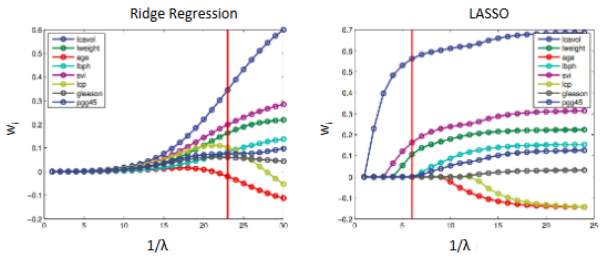


Figura 11: LASSO and Ridge regression comparison

11.5 Feature selection

Very often, in machine learning scenarios, we need to decide what to use as **features** for learning. In general, given the data that somebody provide us, they might be a *better* way to represent such data. This really depends on the choice of the hypothesis class. The best thing we can do to determine the better representation is to try different hypothesis class and pick the feature that perform the best. The scenario for the feature selection problem is the following: we have a large pool of features, and the goal is to select a small subset of features that we're going to use in the final predictor. Let's assume for simplicity that the instance domain is $X = \mathbb{R}^d$. Then our goal is to learn a model that uses only k features, where $k << d$. Some motivation for this choice are:

- To prevent overfitting. By having less predictors, our model will be simpler;
- The predictions can be done faster;
- It's useful in many applications, since we want to be able to look at the model used and understand which features are used.

Let's assume we are using the Empirical Risk Minimization (ERM) procedure. The problem of selecting k features that minimize the empirical risk can be written as

$$min_w L_S(w) \text{ subject to } \|w\|_0 \leq k$$

where $\|w\|_0 = |\{i : w_i \neq 0\}|$. This mean that I want to find a vector w (the hypothesis) that minimizes the training error such that it's $0 - norm$ (the number of component different from zero) is less or equal than k. To solve this problem, by reasoning on the choice of the number of feature, the smallest training error will come by taking exactly k features. In terms of training error the most complex model will always have an advantage since it has more parameter to use to fit the data. Let $I = \{1, \ldots, d\}$ be the indexes of the features. Given $p = \{i_1, \ldots, i_k\} \subseteq I$, we define the corresponding hypothesis class H_p where only the $w_1, \ldots, w_k$ features are going to be used as. The simple pseudo-code to solve the problem is:

$P^{(k)} \longleftarrow \{J \subseteq I : |J| = k\}:$
foreach $p \in P^{(k)}$ do
    $h_p \longleftarrow argmin_{h \in H_p} L_S(h)$;
return $h^{(k)} \longleftarrow argmin_{p \in P^{(k)}} L_S(h_p)$;

The complexity of this algorithm is exponential. In particular

Proposition 9

The optimization problem of feature selection NP-hard.

This mean that there are no algorithms that can do better in asymptotic term. In practice, we can use heuristic solutions. We follow a greedy algorithm by using a **forward selection**. We start from the empty solution, then add one feature at the time, until the solution has cardinality k. The simple pseudo-code to solve the problem is:

```
sol ⟵ ∅
while |sol| < k do
        foreach i ∈ I\sol do
                p ⟵ sol ∪ {i};
                h_p ⟵ argmin_{h∈H_p} L_S(h);
        sol ⟵ sol ∪ argmin_{i∈I sol} L_S((h_{sol∪{i}});
return sol;
```

This has linear complexity ($\Theta(kd)$). We can also do a **backward selection**. Instead of starting from zero, we start from the set of all features and we start removing one feature at the time until the feature set has cardinality k. The pseudo code is similar to what we've saw for forward selection. This has linear complexity ($\Theta(kd)$). In particular, up to now we have used only training set to select the best hypothesis. This might lead to overfitting. The solution to this is to use validation (or cross-validation). The idea is to split data into training data and validation data, learn models on the training set, then evaluate on validation data. The pseudo-code of the Subset Selection with Validation Data is:

```
S =training data(from data split)
V =validation data (from data split)

for k ⟵ 0 to d do
        P^(k) ⟵ {J ⊆ I : |J| = k}:
        foreach p ∈ P^(k) do
                h_p ⟵ argmin_{h∈H_p} L_S(h);
        h^(k) ⟵ argmin_{p∈P^(k)} L_S(h_p);
return argmin_{h∈{h^(0),...,h^(d)}} L_V(h);
```

while the Forward Selection with Validation Data is

```
sol ⟵ ∅
while |sol| < k do
        foreach i ∈ I\sol do
                p ⟵ sol ∪ {i};
                h_p ⟵ argmin_{h∈H_p} L_S(h);
        sol ⟵ sol ∪ argmin_{i∈I sol} L_V((h_{sol∪{i}});
return sol;
```

12 Support Vector Machines

Let's consider a classification problem where $X = \mathbb{R}^d$ and the label set is $Y = \{-1, 1\}$. The training data are the pairs $S = ((x_1, y_1), \dots, (x_m, y_m))$ while the hypothesis set H used are halfspaces. Let's start by making a simple assumption: the data are linearly separable. Therefore exist a halfspace that perfectly classify the training set. With this assumptions, we can choose between infinite models. In general, for a given model, we can define it's **margin** as the minimum distance to an example in the training set S. We want to pick the halfspace that maximize the margin.
We can say that given a training set $S = ((x_1, y_1), \dots, (x_m, y_m))$, this is *linearly separable* if there exists a halfspace (w, b) such that $y_i = sign(\langle w, x_i \rangle + b))$ for all $i = 1, \dots, m$. This is equivalent to write

$$\forall i = 1, \dots, m : y_i(\langle w, x_i \rangle + b) > 0$$

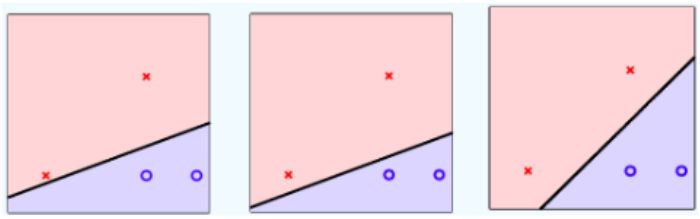


Figura 12: Different halfspaces classify perfectly the same data

To compute the distance, given the hyperplane defined by $L = \{v : \langle w, v\rangle + b > 0\}$ and given x, the distance of x to L is defined as

$$d(x, L) = min\{\|x - v\| : v \in L\}$$

In particular, we can claim that if $\|w\| = 1$ then $d(x, L) = |\langle w, x\rangle + b|$. The **margin** of a

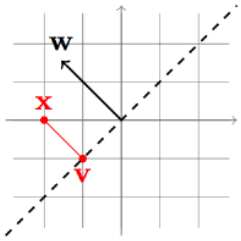


Figura 13: geometric distance

separating hyperplane is the distance of the closest example in training set to it. If $\|w\| = 1$ the margin is

$$min_{i\in\{1,\ldots,m\}}|\langle w, x_i\rangle + b|$$

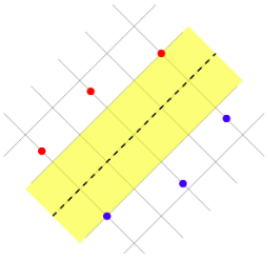


Figura 14: margin of a hyperplane

The closest examples to the halfspace are called **support vectors**. For **support vector machines** the problem we have to solve is the following: we want to look for the separating hyperplane that has the largest margin. When you look at linearly separable data, we are in the case of **Hard-SVM**. The computational problem can be rewrite as follows:

$$argmax_{(w,b):\|w\|=1, i\in\{1,\ldots,m\}} min\ |\langle w, x_i\rangle + b|$$

subject to $\forall i : y_i(\langle w, x_i\rangle + b) > 0$. Since we've made the separability assumption, we can rewrite this formula as

$$argmax_{(w,b):\|w\|=1, i\in\{1,\dots,m\}} min\ y_i(\langle w, x_i\rangle + b)$$

the reason is that y_i is actually equal to the sign of the prediction. By taking the absolute value and multiply by y_i we obtain the same thing. Here we don't need to put explicitly the constraints of linearly separable data. To solve this we are going to resort to quadratic programming formulations. Therefore we can rewrite the problem as follows.

- Input: $S = ((x_1, y_1), \dots, (x_m, y_m))$;
- Solve: $(w_0, b_0) = argmin_{(w,b)}\|w\|^2$ subject to $\forall i : y_i(\langle w, x_i\rangle + b) \geq 1$;
- Output: $\hat{w} = \frac{w_0}{\|w_0\|}$, $\hat{b} = \frac{b_0}{\|w_0\|}$.

The idea is that we are still forcing the margin, but rescaling the w to produce in output the optimal solution and then scale w back. We can do this because only the sign matters, and the scale doesn't change the sign. Since the objective is convex quadratic function, and the constraints are linear inequalities this is easy to solve in practice.

Proposition 10

The output of algorithm above is a solution to the Equivalent Formulation

$$argmax_{(w,b):\|w\|=1, i\in\{1,\dots,m\}} min\ y_i(\langle w, x_i\rangle + b)$$

An equivalent formulation of what we need to solve considering the first component of $x \in X$ is 1 is

$$w_0 = argmin_w\|w\|^2 \text{ subject to } \forall i : y_i\langle w, x_i\rangle \geq 1$$

b is now included in w. This formulation is useful to define what the support vectors are. The **support vectors** are the vectors with minimum distance from w_0. Then the follow proposition holds

Proposition 11

Let w_0 be as above. Let $I = \{i : |\langle w_0, x_i\rangle| = 1\}$. Then there exist coefficients $\alpha_1, \dots, \alpha_m$ such that

$$w_0 = \sum_{i\in I} \alpha_i x_i$$

To find the defining model for SVM all i need to know are the support vectors $\{x_i : i \in I\}$ and the corresponding α_i.
If data are however not linearly separable, we use the so-called **Soft-SVM**. The idea is to modify constraints of Hard-SVM to allow for some violation, but take into the account violations into objective function. The constraints for Hard-SVM were

$$y_i(|\langle w, x_i\rangle| + b) \geq 1$$

In the case of Soft-SVM, we introduce *slack variables* $\xi_1, \dots, \xi_m \geq 0$. We can represent those as a vector ξ. For each $i = 1, \dots, m$, the constraint is

$$y_i(|\langle w, x_i\rangle| + b) \geq 1 - \xi_i$$

In particular, ξ_i represent how much the constraint $y_i(|\langle w, x_i\rangle| + b) \geq 1$ is violated. Soft-SVM then tries to find a w that minimizes a combination of the norm of w and the average of ξ_i. The tradeoff among two terms is controlled by a parameter $\lambda \geq 0 \in \mathbb{R}$. Therefore we can rewrite the problem as follows:

- Input: $S = ((x_1, y_1), \ldots, (x_m, y_m))$, parameter $\lambda > 0$;
- Solve:
$$min_{w,b,\xi} \left(\lambda \|w\|^2 + \frac{1}{m} \sum_{i=1}^{m} \xi_i \right)$$
subject to $\forall i : y_i(\langle w, x_i\rangle + b) \geq 1 - \xi_i$ and $\xi_i \geq 0$;
- Output: w, b.

The idea is that I want to find w, b, ξ that minimizes the sum subject to the modified constraint for Hard-SVM. An equivalent formulation can be given by defining the *hinge loss*:

$$\ell^{hinge}((w, b), (x, y)) = max\{0, 1 - y(\langle w, x\rangle + b)\}$$

The value $1 - y(\langle w, x\rangle + b)$ represent how much the constraint is violated. If the constrain is satisfied, then this term is zero. The Soft-SVM problem is therefore equivalent to finding w, b that maximizes a different objective function where I have an empirical risk in term of hinge loss. Given (w, b) and a training set S, the empirical risk $L_S^{hinge}((w, b))$ is

$$L_S^{hinge}((w, b)) = \frac{1}{m} \sum_{i=1}^{m} \ell^{hinge}((w, b), (x_i, y_i))$$

Now we can then look at the problem of solving Soft-SVM in an equivalent formulation using hinge loss:

$$min_{w,b}(\lambda \|w\|^2 + L_S^{hinge}(w, b))$$

that is

$$min_{w,b} \left(\lambda \|w\|^2 + \frac{1}{m} \sum_{i=1}^{m} \ell^{hinge}((w, b), (x_i, y_i)) \right)$$

Where in particular $\lambda\|w\|^2$ is the ℓ^2 regularization and $L_S^{hinge}((w, b))$ is the empirical risk for hinge loss. To find w and b that solves the Soft-SVM problem it's used a standard solvers for optimization problems technique called **stochastic gradient descent**.

12.1 Gradient Descent

Gradient descent is a general approach to minimizing a differentiable convex function $f(w)$. Let $f : \mathbb{R}^d \longrightarrow \mathbb{R}$ be a differentiable function. Then we can define the *gradient* of a function as

Definition 19

The gradient $\nabla f(w)$ of f at $w = (w_1, \ldots, w_d)$ is

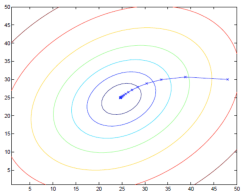
$$\nabla f(w) = \left(\frac{\partial f(w)}{w_1}, \ldots, \frac{\partial f(w)}{w_d} \right)$$

The idea is that the gradient points in the direction of the greatest rate of increase of the function f around w. By taking the opposite of the gradient we can move towards the minimum values of the function. With this in mind, we can write down the pseudo-code of the gradient descent. Let $\eta \in \mathbb{R}, \eta > 0$ be a parameter. Then we define the **gradient descent algorithm** as

$w^{(0)} \longleftarrow 0$;
for $t \longleftarrow 0$ to $T-1$ do
    $w^{(t+1)} \longleftarrow w^{(t)} - \eta \nabla f(w^{(t)})$;
return $\overline{w} = \frac{1}{T}\sum_{i=1}^{T} w^{(t)}$;

The algorithm update the value that it's looking at by moving in the direction of the greatest decrease that is given by the opposite of the gradient multiplied by η. This parameter decide if the step is going to be "small" or "big". Usually η change over time. In particular,

- The output vector could also be $w^{(t)}$ or $argmin_{w^{(t)} \in \{1,\dots,T\}} f(w^{(t)})$. This is because if we passed trough the minima in previous iteration, and the algorithm terminate in a $\overline{w} > w^{(t)}$, we still want to return the value that minimize the most f;
- Returning $\overline{w}$ is useful for nondifferentiable functions and for stochastic gradient descent;
- The parameter η is called *learning rate*. The learning rate decide how much to move in the direction of the gradient. Sometimes, a time dependent $\eta^{(t)}$ can be used.



The *guarantees* of the gradient descent are the following: let f be a convex function and ρ-Lipschitz continuous; Let $w^* \in argmin_{w:\|w\| \leq B} f(w)$. Then for every $\varepsilon > 0$, to find $\overline{w}$ such that $f(\overline{w} - f(w^*) \leq \varepsilon$ it is sufficient to run the gradient descent algorithm for $T \geq \frac{B^2\rho^2}{\varepsilon^2}$. This proves that gradient descent actually converge to a very good solution.

12.2 Stochastic Gradient Descent

Stochastic Gradient Descent instead of using exactly the gradient take a (random) vector with expected value equal to the gradient direction. The algorithm of SGD is

$w^{(0)} \longleftarrow 0$;
for $t \longleftarrow 0$ to $T-1$ do
    choose v_t at random from distribution such that $\mathbb{E}[v_t | w^{(t)}] \in \nabla f(w^{(t)})$;
    $w^{(t+1)} \longleftarrow w^{(t)} - \eta v_t$;
return $\overline{w} = \frac{1}{T}\sum_{i=1}^{T} w^{(t)}$;

In SGD we are taking directions that are not exactly the ones given by the gradient, but that on average the direction will be similar. As you get closer to the end, it won't end up exactly in the local minimum but there's going to be some noise. In SGD the *guarantees* are: let f be a convex function and ρ-Lipschitz continuous; Let $w^* \in argmin_{w:\|w\| \leq B} f(w)$. Then for every $\varepsilon > 0$, to find $\overline{w}$ such that $\mathbb{E}[f(\overline{w})] - f(w^*) \leq \varepsilon$ it is sufficient to run the stochastic gradient descent algorithm for $T \geq \frac{B^2\rho^2}{\varepsilon^2}$. This proves that gradient descent actually converge to a very good solution.

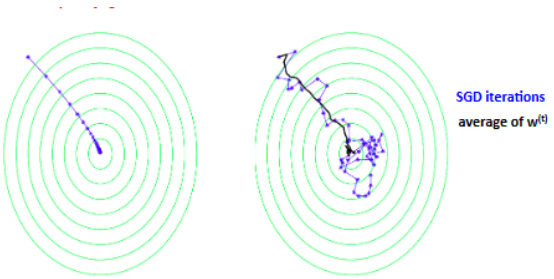


Figura 15: GD and SGD

Using GD vs SGD We can actually use gradient descent to evaluate w that minimizes $L_S(w)$. By considering as a function $f(w) = L_S(w)$, the gradient $\nabla f(w)$ depends on the m pairs $(x_i, y_i) \in S$. This may require a lot of time if the number of points is really high. However, in Stochastic gradient descent we need to pick v_t such that $\mathbb{E}[v_t|w^{(t)}] \in \nabla f(w^{(t)}$. One simple way to do this is by picking a random sample $(x_i, y_i) \in S$ and compute the gradient of the loss for that particular point. Therefore $v_t \in \nabla \ell(w^{(t)}, (x_i, y_i))$. This satisfies the requirements and requires much less computation than Gradient Descent, since we just need to compute the gradient fora single point. Analogously we can use SGD for regularized losses, etc.

Soft-SVM using SGD Let's now consider the following problem of Soft-SVM

$$min_w \left(\frac{\lambda}{2}\|w\|^2 + \frac{1}{m}\sum_{i=1}^{m} max\{0, 1 - y\langle w, x_i\rangle\} \right)$$

note that we can add a $\frac{1}{2}$ in the regularization term to simplify some computations. The **Stochastic Gradient Descent algorithm** is the following:

$\theta^{(1)} \longleftarrow 0$;
for $t \longleftarrow 1$ to T do
  $\eta^{(t)} \longleftarrow \frac{1}{\lambda t}$; $w^{(t)} \longleftarrow \eta^{(t)}\theta^{(t)}$;
  choose i uniformly at random from $\{1, \ldots, m\}$;
  if $y_i\langle w^{(t)}, x_i\rangle < 1$ then
    $\theta^{(t+1)} \longleftarrow \theta^{(t)} + y_i x_i$;
  else $\theta^{(t+1)} \longleftarrow \theta^{(t)}$;
return $\overline{w} = \frac{1}{T}\sum_{i=1}^{T} w^{(t)}$;

12.3 Duality

We now present (Hard-)SVM in a different way which is very useful for *kernels*. We want to solve

$$w_0 = min_w \frac{1}{2}\|w\|^2 \text{ subject to } \forall i : y_i\langle w, x_i\rangle \geq 1$$

One can prove that w that minimizes the function above is equivalent to find α that solves the **dual problem**:

$$max_{\alpha \in \mathbb{R}^m : \alpha \geq 0} min_w \left(\frac{1}{2}\|w\|^2 + \sum_{i=1}^{m} \alpha_i (1 - y_i\langle w, x_i\rangle) \right)$$

In particular,solving this new problem will be much easier than the previous one. This is due to the fact that once α is fixed, the optimization with respect to w is unconstrained. Also, the objective is *differentiable*. Therefore, at the optimum value of w for a fixed α the gradient is equal to 0:

$$w - \sum_{i=1}^{m} \alpha_i y_i x_i = 0 \Rightarrow w = \sum_{i=1}^{m} \alpha_i y_i x_i$$

By replacing this in the dual problem we obtain

$$max_{\alpha \in \mathbb{R}^m : \alpha \geq 0} \left(\frac{1}{2} \| \sum_{i=1}^{m} \alpha_i y_i x_i \|^2 + \sum_{i=1}^{m} \alpha_i \left(1 - y_i \langle \sum_{j=1}^{m} \alpha_j y_j x_j, x_i \rangle \right) \right)$$

by rearranging the problem we get

$$max_{\alpha \in \mathbb{R}^m : \alpha \geq 0} \left(\sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j \langle x_j, x_i \rangle \right)$$

The solution is the vector α which defines the support vectors $\{x_i : \alpha_i \neq 0\}$. Also, dual problem requires only to compute inner products $\langle x_j, x_i \rangle$. Thus I can still solve the problem without knowing x_i by itself. However, we are still bounded to linear models, and linear models can't

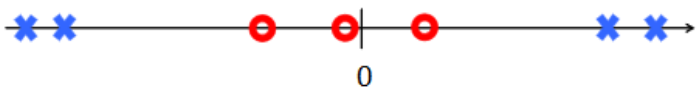


Figura 16: Linear model

always be (directly) used. We can however apply a nonlinear transformation $\psi()$ to each point in the training set S first. We obtain the transformed set $S' = ((\psi(x_1), y_1), \ldots, (\psi(x_m), y_m))$. Then i could learn a linear predictor $\hat{h}$ in the transform space using S'. We can then make a prediction for a new istance x as $\hat{h}(\psi(x))$.

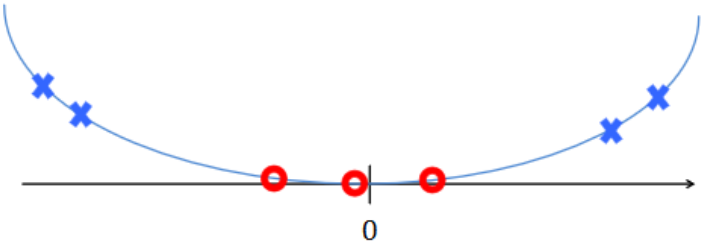


Figura 17: Nonlinear model

12.4 Kernel Trick for SVM

Let's suppose we want to apply some nonlinear transformation before using SVM. Let $\psi U)$ be a nonlinear transformation. If we look at the dual formulation, we know that to solve Hard-SVM (and it's the same for Soft-SVM) we need to be able to compute to find the best hypothesis the dot product $\langle \psi(x), \psi(x') \rangle$ for some x and x'. We can define

Definition 20

A **kernel function** is a function of the type:

$$K_\psi(x, x') = \langle \psi(x), \psi(x') \rangle$$

where $\psi()$ is a transformation of x.

One way to think about the kernel function K_ψ is to think of them as function that specifies how similar two instances are where $\psi()$ is a mapping between the original space X and a space where these similarities are realized as inner products. A particular property of the kernel (the Kernel Trick) is that we can sometimes compute $K_\psi(x, x')$ without knowing how to compute $\psi(x)$. Let's consider the following example:

Example. *Consider $x \in \mathbb{R}^d$. We consider the transformation*

$$\phi(x) = (1, x_1, x_2, \ldots, x_d, x_1x_1, x_1x_2, \ldots, x_dx_d)^T$$

The dimension of $\psi(x)$ is $1 + d + d^2$. If I look at the dot product, i obtain

$$\langle \psi(x), \psi(x') \rangle = 1 + \sum_{i=1}^{d} x_i x_i' + \sum_{i=1}^{d} \sum_{j=1}^{d} x_i x_j x_i' x_j'$$

Note that

$$\sum_{i=1}^{d} \sum_{j=1}^{d} x_i x_j x_i' x_j' = \left(\sum_{i=1}^{d} x_i x_i' \right) \left(\sum_{j=1}^{d} x_j x_j' \right) (\langle x, x' \rangle)^2$$

therefore

$$K_\psi(x, x') = \langle \psi(x), \psi(x') \rangle = 1 + \langle x, x' \rangle + (\langle x, x' \rangle)^2$$

We can observe that computing $\phi(x)$ requires $\Theta(d^2)$ time, while computing the kernel $K_\phi(x, x')$ from the last formula requires $\Theta(d)$ time.

This example shows that when $K_\phi(x, x')$ is efficiently computable, we don't need to explicitly compute $\phi(x)$. This goes by the name of **Kernel Trick**. That is, I can have extremely complicated transformations but for which there is a way to compute the corresponding kernel efficiently. In that way, I can use the kernel efficiently in SVM. The most commonly used kernels are:

- **Linear kernel**: $\phi(x) = x$;
- **Sigmoid**: $K_\phi(x, x') = tanh(\gamma \langle x, x' \rangle + \zeta)$ for $\gamma, \zeta > 0$;
- **Degree-Q polynomial kernel**;
- **Gaussian-radial basis function (RBF) kernel**.

We are going to look into those two more in details.

Degree-Q polynomial kernel

Definition 21

For given constants $\gamma > 0, \zeta > 0$ and for $Q \in N$, the **degree-Q polynomial kernel** is

$$K(x, x') = (\zeta + \gamma\langle x, x'\rangle)^Q$$

For example, we can consider the degree-2 polynomial kernel as

$$\psi(x) = [\zeta, \sqrt{2\zeta\gamma}x_1, \sqrt{2\zeta\gamma}x_d, \ldots, \sqrt{2\zeta\gamma}x_d, \gamma x_1 x_1, \gamma x_1 x_2, \ldots \gamma x_d x_d]^T$$

Gaussian-RBF Kernel

Definition 22

For a given constant $\gamma > 0$ the **Gaussian-RBF kernel** is

$$K_\phi(x, x') = e^{-\gamma\|x - x'\|^2}$$

Here the idea is that if x and x' are equal, the exponent is equal to 0 so the kernel shows they are very similar. If the samples are very different, they are zero.

Choice of kernel For the polynomial kernel we usually consider a $Q \leq 10$. With Gaussian-RBF kernels, we consider a γ between 0 and 1 because it works well in practice. γ gets picked using validation between 0and 1. However, there are many other choice possible, both for the parameters and the kernels in general. The choice of the kernel used depends validation over the problem. Sometimes you might know some particular features on the data that might relate to some specific kernel. It's also possible to come up with a totally made up kernel. The **Mercer's condition** states that

Definition 23

$K(x, x\prime)$ is a valid kernel function if and only if the kernel matrix

$$K = \begin{bmatrix} K(x_1, x_1) & K(x_1, x_2) & \ldots & K(x_1, x_m) \\ K(x_2, x_1) & \ldots & \ldots & K(x_2, x_m) \\ \vdots & \vdots & \vdots & \vdots \\ K(x_m, x_1) & K(x_m, x_2) & \ldots & K(x_m, x_m) \end{bmatrix}$$

is always symmetric positive semi-definite for any given $x_1, x_2, \ldots, x_m$

12.5 Support Vector Machines for Regression

It's also possible to use SVM for regression problems. In particular, the function that we look at to pick our model w can be minimized as

$$\frac{\lambda}{2}\|w\|^2 + \sum_{i=1}^{m} V_\varepsilon(y_i - \langle x_i, w\rangle - b)$$

where

$$V_\varepsilon(r) = \begin{cases} 0 & \text{if } |r| < \varepsilon \\ |r| - \varepsilon & \text{otherwise} \end{cases}$$

This is similar to the function used for classification. It has the regularization parameter plus a term that corresponds essentially to a loss function. One can prove that the solution has the form

$$w = \sum_{i=1}^{m} (\alpha^* - \alpha_i) x_i$$

and that the final model produced in output is

$$h(x) = \sum_{i=1}^{m} (\alpha^* - \alpha_i) \langle x_i, x \rangle + b$$

where $\alpha^*, \alpha \geq 0$ and are the solution to a suitable quadratic program. Therefore we can define the *support vectors* for SVM for regression as

Definition 24

support vectors: x_i such that $\alpha^* - \alpha_i \neq 0$.

One can define kernels, similarly to SVM for classification.

13 Neural Networks

Neural networks are a type of Machine learning model with associated algorithms to learn a good hypothesis in the class. Neural Networks are a simplified models of the brain. A NN usually has a large number of basic computing units called *neurons*. Those neurons are connected in a complex network.

Neuron A **neuron** is a function $x \longrightarrow \sigma(\langle v, x \rangle)$, with $x \in \mathbb{R}^d$. The function $\sigma : \mathbb{R} \longrightarrow \mathbb{R}$ is called **activation function**. An example with $\mathbb{R}^5$ can be seen in picture The activation functions

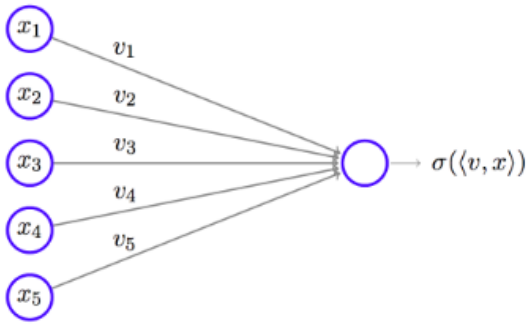


Figura 18: example of neuron

we'll consider are

- **sign function:** $\sigma(a) = sign(a)$. Fire only if a certain number of nearby neurons is firing;
- **threshold function**: $\sigma(a) = \mathbb{1}[a > 0]$ which shoot only when a is grater than zero;
- **sigmoid function**: $\sigma(a) = \frac{1}{1+e^{-a}}$.

Another activation function used a lot is the ReLU which will be covered later on.

Neural Networks A neural network can be obtained by combining many neurons together. We focus on **feed-forward networks**, which are defined by an acyclic graph $G = (V, E)$ organized in *layers*. Each edge e has a weight $w(e)$ specified by $w : E \longrightarrow \mathbb{R}$. The first layer (on

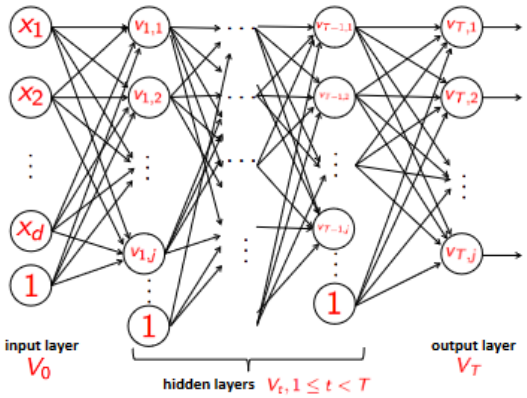
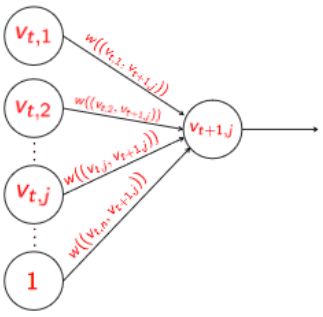


Figura 19: example of feed-forward neural network

the left) has no incoming edge and represent the *input*. Each of the node of this layer is going to contain a value of the input vector $x = (x_1, \ldots, x_d)$ plus a constant node which represent the bias. The bias node exists in every layer (except the output layer) and has no connections going into it. The layers between the input and the last layer are called *hidden layers*. Those layers corresponds values not observed by our machine learning problem. The last layer is representing the *output*. In particular, each node in the hidden layer is going to be represented using 2 vertices, the first is the index of the hidden layer, while the second is the index of the vertex in that layer: $V_{t,j}$. The input layer is represented as V_0 while the output layer is V_T.
Let's consider a node $v_{t+1,j}$, $0 \leq t < T$. Let's call

- $a_{t+1,j}(x)$ its input when x is fed to the Neural Network;
- $o_{t+1,j}(x)$ its output when x is fed to the Neural Network.



Then the input of the next neuron is

$$a_{t+1,j}(x) = \sum_{r:(v_{t,r},v_{t+1,j})\in E} w((v_{t,r}v_{t+1,j})o_{t,r}(x)$$

The output propagated by the neuron is $o_{t+1,j}(x) = \sigma(a_{t+1,j}(x))$. Thus it's just the sum over the vertices of the previous layers of the value for that vertex times the weight of the edge. The number of outgoing edges of the neuron depends on the architecture. For every edge it propagates the same value.

NN Formalism Let's introduce some *formalism* that we'll refer to from now on:

- Neural Network: described by a *directed acyclic graph* $G = (V, E)$ and a weight function : $E \longrightarrow \mathbb{R}$;
- The set of vertices is given by the vertices in all the layers: $V = \cup_{t=0}^{T} V_t, V_i \cap V_j = \emptyset \; \forall i \neq j$;
- The edges $e \in E$ can from V_t to $V_t + 1$ for some t;
- The vertices in the **input layer** form the set V_0;
- The vertices in the **output layer** form the set V_T;
- The vertices in the middle form the **hidden layers** $V_t, 0 < t < T$;
- T is the **depth** of the network;
- The number of vertices $|V|$ is the **size** of the network;
- The maximum number of vertices in a layer $max_t|V_t|$ is the **width** of the network.

In particular, for binary classification and regression (for one variable) the output layer has only one node. In general, different layer could have **different** activation functions.

Hypothesis set of a Neural Network In general, when you use Neural Network, what's specified is the *architecture* of the network. The architecture of a Neural Network is given as the triple (V, E, σ). Once we specify the architecture and w, we obtain the corresponding hypothesis for the neural network:

$$h_{V,E,\sigma,w} : \mathbb{R}^{|V_0|-1} \longrightarrow \mathbb{R}^{|V_T|}$$

The **hypothesis class** of a neural network is defined by fixing its architecture:

$$H_{V,E,\sigma} = \{h_{V,E,\sigma,w} : w \text{ is mapping from } E \text{ to } \mathbb{R}\}$$

Once I have decided the hypothesis we need to find the weights.
We want to examine what kind of functions can be implemented in Neural Networks.

13.1 Binary function in Neural Networks

In general, we can say that

Proposition 12

For every d, there exists a graph (V, E) of depth 2 such that $H_{V,E,sign}$ contains all function from $\{-1, 1\}^d$ to $\{-1, 1\}$.

Therefore, Neural Network can implement every boolean function. However the resulting graph might be very large. It can be proved that

Proposition 13

For every d, let $s(d)$ be the minimal integer such that there exists a graph (V, E) with $|V| = s(d)$ such that $H_{V,E,sign}$ contains all function from $\{-1,1\}^d$ to $\{-1,1\}$. Then, $s(d)$ is an exponential function of d.

When we are doing regression instead of classification, we can state that

Proposition 14

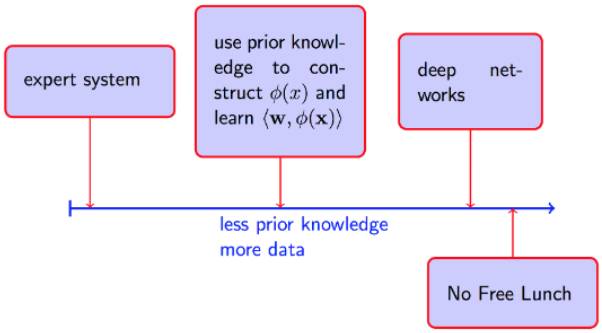
For every fixed $\varepsilon > 0$ and every Lipschitz function $f : [-1,1]^d \longrightarrow [-1,1]$ it is possible to construct a neural network such that for every input $x \in [-1,1]^d$ the output of the neural network is in $[f(x) - \varepsilon, f(x) + \varepsilon]$.

Therefore is possible to use Neural Networks to approximate function that produce a real value if this function are not too "complex".This first result proved by Cybenko (1989) for the sigmoid activation function using only 1 hidden layer. Due to this property, NNs are also called *universal approximates*. To obtain this result we need an exponential neural network. We can state that

Proposition 15

Fix some $\varepsilon \in (0,1)$. For every d, let $s(d)$ be the minimal integer such that there exists a graph (V, E) with $|V| = s(d)$ such that $H_{V,E,\sigma}$, with σ =sigmoid, can approximate, with precision ε,every $1-$Lipschitz function $f : [-1,1]^d \longrightarrow [-1,1]$. Then $s(d)$ is exponential in d.

In neural network we move away from models where you have to put previous knowledge about the problem in trying to capture a good model in your class. NNs can extract automatically informations from the data provided you have enough data and the architecture is not too bad.



Solving boolean functions in Neural networks Consider an arbitrary function

$$f : \{-1,1\} \longrightarrow \{-1,1\}$$

and consider $\overrightarrow{x}$ such that $f(\overrightarrow{x}) = 1$. Then consider a Neural Network with an input layer, one hidden layer and one output layer. In the hidden layer, each neuron represent one of the $\overrightarrow{x}$'s

above and implements $g_i(\overrightarrow{x}) = sign(\langle \overrightarrow{x}, \overrightarrow{w_i} \rangle - d + 1)$. In particular, $\overrightarrow{w_i} = \overrightarrow{x_i}$, and in the last layer $f(\overrightarrow{x}) = sign(\sum_{i=1}^{k} g_i(\overrightarrow{x}) + k - 1)$. Therefore, we have in the layer V_0, V_1 and V_2,

$$d \begin{cases} x_1 \\ x_2 \\ \vdots \\ x_d \\ 1(bias) \end{cases} \qquad k-1 \begin{cases} g_i(\overrightarrow{x}) \\ \vdots \\ \vdots \\ 1(bias) \end{cases} \qquad 1 \Big\{ outputnode$$

where $0 \leq k \leq 2^d$. $\{\overrightarrow{x_1}, \ldots, \overrightarrow{x_k}\}$ = set of vectors for which $f() = 1$. The, $\forall \overrightarrow{x_i} \in \{-1, 1\}^d$ and $\forall 1 \leq i \leq k$:

a) if $\overrightarrow{x} \neq \overrightarrow{x_i}$ then

$$\langle \overrightarrow{x}, \overrightarrow{x_i} \rangle = \langle \overrightarrow{x}, \overrightarrow{w_i} \rangle \leq d - 1 - 1 = d - 2 \Rightarrow g_i(\overrightarrow{x}) = -1$$

where $d-1$ comes from the upper bound to the number of agreements between $\overrightarrow{x_i}, \overrightarrow{x_i}$, and the -1 comes from the lower bound to the number of disagreements between $\overrightarrow{x_i}, \overrightarrow{x_i}$.

b) if $\overrightarrow{x} = \overrightarrow{x_i}$, then

$$\langle \overrightarrow{x}, \overrightarrow{x_i} \rangle = \langle \overrightarrow{x}, \overrightarrow{w_i} \rangle = d \Rightarrow g_i(\overrightarrow{x}) = 1$$

Therefore, from $[a]$ and $[b]$ we have that $g_i(\overrightarrow{x}) = 1 \Leftrightarrow \overrightarrow{x} = \overrightarrow{x_i}$ $(g_i(\overrightarrow{x}) represent \overrightarrow{x_i})$. Basically, if the input is a vector that is not a vector for which the function is 1, then all the hidden layers produce in output a value equals to -1. I need to set the weights in my function $F()$ such that i can distinguish the two cases by setting the weights. We can look at $f(\overrightarrow{x})$ as

$$f(\overrightarrow{x}) = \vee_{i=1,..k} g_i(\overrightarrow{x}) = sign(\sum_{i=1}^{k} g_i(\overrightarrow{x}) + k - 1)$$

where $\vee$ represent the *or*. Then, the two cases are

a) $\overrightarrow{x} \neq \overrightarrow{x_i}$ $\forall i \Rightarrow g_i(\overrightarrow{x}) = -1$ $\forall 1 \leq i \leq k \Rightarrow \sum_{i=1}^{k} g_i(\overrightarrow{x}) = -k \Rightarrow f(\overrightarrow{x}) = -1$;

b) $\overrightarrow{x} = \overrightarrow{x_i}$ for an arbitrary i, with $1 \leq i \leq k$. Then

$$\sum_{j=1}^{k} g_j(\overrightarrow{x}) = \sum_{j=1, j \neq i}^{k} g_j(\overrightarrow{x}) + g_i(\overrightarrow{x}) = -k + 2 \Rightarrow f(\overrightarrow{x}) = sign(-k + 2 + k - 1) = 1$$

What we are doing is using the hidden layer to represent each different input, then the final layer collects the information and implements the *or*. This shows that whatever binary function you want to represent, you can do it with neural network.

13.2 Sample Complexity of NNs

How much data do we need to learn with Neural Networks? It can be stated that

Proposition 16

The VC dimension of $H_{V,E,\sigma=sign}$ is on the order of $O(|E| \log |E|)$.

In practice, Neural Networks still work well with a smaller amount of data. The VC dimension gives an upperbound which is somewhat reflected in practice in the requirements of large datasets. If we consider the sigmoid activation function, we have

Proposition 17

Let σ be the sigmoid function. The VC dimension of $H_{V,E,\sigma}$ is:

- $\Omega(|E|^2)$;
- $O(|V|^2|E|^2)$.

Assuming we have a large set of data, what's the runtime of Neural Network? Applying the ERM rule with respect to $H_{V,E,sign}$ is computationally difficult (NP-Hard), even for small NN. A famous result state that

Proposition 18

Let $k \geq 3$. For every d, let (V, E) be a layered graph with d input nodes,$k+1$ nodes at the (only) hidden layer, where one of them is the constant neuron, and a single output node. Then, it is NP-hard to implement the ERM rule with respect to$H_{V,E,sign}$.

This is true even considering different hypothesis class, different activation functions or smarter embeddings in larger network.

13.3 Matrix Notation

Let's now explicit some matrix notation regarding Neural Network which will helps us later on when explaining the backpropagation. Let's consider a layer t, where $0 \leq t \leq T$:

- Let $d^{(t)}+1$ the number of nodes. We have the constant node 1 and the values of nodes for (hidden) variables $v_{t,1}, \ldots, v_{t,d^{(t)}}$;
- Every Edge from $v_{t-1,i}$ to $v_{t,j}$ has weight $w_{ij}^{(t)}$.

Let

$$v^{(t)} = \left(1, v_{t,1}, \ldots, v_{t,d^{(t)}}\right)^T$$

be the vector which contains the value of the layer t and

$$w_j^{(t)} = \left(w_{0j}^{(t)}, w_{qj}^{(t)}, \ldots, w_{d^{(t-1)}j}^{(t)}\right)^T$$

the vector used to describe the weights. Then to compute the value we have

$$v_{t,j} = \sigma(\langle w_j^{(t)}, v^{(t-1)} \rangle)$$

Therefore we can rewrite $v^{(t)}$ as

$$v^{(t)} = \begin{bmatrix} 1 \\ v_{t,1} \\ \vdots \\ v_{t,d^{(t)}} \end{bmatrix} = \begin{bmatrix} 1 \\ \sigma(\langle w_1^{(t)}, v^{(t-1)} \rangle) \\ \vdots \\ \sigma(\langle w_{d^{(t)}}^{(t)}, v^{(t-1)} \rangle) \end{bmatrix}$$

We call those arguments

$$a_{t,j} = \langle w_j^{(t)}, v^{(t-1)} \rangle$$

Then we can write

$$a^{(t)} = \begin{bmatrix} a_{t,1} \\ \vdots \\ a_{t,d^{(t)}} \end{bmatrix} \qquad \sigma(a^{(t)}) = \begin{bmatrix} \sigma(a_{t,1}) \\ \vdots \\ \sigma(a_{t,d^{(t)}}) \end{bmatrix}$$

Then we can write down the relation between the vector $v^{(t)}$ and the vector of the input arguments as

$$v^{(t)} = \begin{bmatrix} 1 \\ \sigma(a^{(t)]}) \end{bmatrix}$$

We are going to describe the weights of edges from layer $t-1$ to layer t as

$$w^{(t)} = \begin{bmatrix} w_{01}^{(t)} & w_{02}^{(t)} & \dots & w_{0d^{(t)}}^{(t)} \\ w_{11}^{(t)} & w_{12}^{(t)} & \dots & w_{1d^{(t)}}^{(t)} \\ \vdots & \vdots & \dots & \vdots \\ w_{d^{(t-1)}1}^{(t)} & w_{d^{(t-1)}2}^{(t)} & \dots & w_{d^{(t-1)}d^{(t)}}^{(t)} \end{bmatrix}$$

Then we can write that the vector that we use to compute the output for the layer t is

$$a^{(t)} = (w^{(t)})^T v^{(t-1)}$$

Let's now write down the algorithm which takes in input a vector x and produce in output the corresponding predictions.

```
Input: x = (x_1, ..., x_d)^T; NN with 1 output node;
Output: prediction y of NN;
v^(0) ⟵ (1, x_1, ..., x_d)^T;
for t ⟵ 1 to T do:
        a^(t) ⟵ (w^(t))^T v^(t-1));
        v^(t) ⟵ (1, σ(a^(t))^T)^T;
y ⟵ σ(a^(t));
return y
```

To compute the weights $w_{ij} * (t)$ we follow the Empirical Risk Minimization. Given the training data, we're looking for the $w_{ij} * (t), \forall i, j, t$ which minimize the training error.To compute this we use **Stochastic Gradient Descent**.

13.4 SGD for backpropagation

Here we use Stochastic Gradient Descent seeing $L_S(h)$ as a function of $w^{(t)}$, $\forall 1 \leq t \leq T$. The SGD update rule is

$$w^{(t)} \longleftarrow w^{(t)} - \eta \nabla L_S(w^{(t)})$$

where $\nabla L_S(w^{(t)})$ is the gradient of L_S and η is the learning rate. To compute it we need $\forall t, 1 \leq t \leq T$:

$$\frac{\partial L_S}{\partial w^{(t)}} = \frac{\partial}{\partial w^{(t)}} \left(\frac{1}{m} \sum_{i=1}^{m} \ell(h, (x_i, y_i)) \right) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial L(h, (x_i, y_i))}{\partial w^{(t)}}$$

Therefore we need to be able to compute $\frac{\partial L}{\partial w^{(t)}}$. We can now define the **Sensitivity vector for layer** t as

Definition 25

Sensitivity vector for layer t $\delta^{(t)} = \frac{\partial L}{\partial a^{(t)}} = \begin{bmatrix} \frac{\partial L}{\partial a^{(t)}_{t,1}} \\ \vdots \\ \frac{\partial L}{\partial a^{(t)}_{t,d^{(t)}}} \end{bmatrix} = \begin{bmatrix} \delta^{(t)}_1 \\ \vdots \\ \delta^{(t)}_{d^{(t)}} \end{bmatrix}$

This vector quantifies how the training error changes with $a^{(t)}$ (the inputs to the t layer - before the nonlinear transformation). Let's consider a weight $w^{(t)}_{ij}$: a change in $w^{(t)}_{ij}$ change only $a_{t,j}$ therefore by chain rule we have

$$\frac{\partial L}{\partial w^{(t)}_{ij}} = \frac{\partial L}{\partial a_{t,j}} \frac{\partial a_{t,j}}{\partial w^{(t)}_{ij}} = \delta^{(t)}_j \frac{\partial}{\partial w^{(t)}_{ij}} \left(\sum_{k=0}^{d^{(t-1)}} w^{(t)}_{kj} v_{t-1,k} \right) = \delta^{(t)}_j v_{t-1,i}$$

Therefore to compute the gradient we only need $\delta^{(t)} = \frac{\partial L}{\partial a^{(t)}} \forall t$. Since the loss L depends from $a_{t,j}$ only through $v_{t,j}$, then from the chain rule :

$$\delta^{(t)}_j = \frac{\partial L}{\partial a_{t,j}} = \frac{\partial L}{\partial v_{t,j}} \frac{\partial v_{t,j}}{\partial a_{t,j}} = \frac{\partial L}{\partial v_{t,j}} \sigma\prime(a_{t,j})$$

where the last equality derives from the definition of $v_{t,j}$. To compute $\frac{\partial L}{\partial v_{t,j}}$ we need to understand how the loss L changes with respect to $v_{t,j}$:

- A change in $v^{(t)}$ affects only $a^{(t+1)}$(and then L);
- Changes in $v_{t,j}$ can affect every $a_{t+1,k}$.

therefore we need to sum up the contributions of the chain rule. Then, we can write

$$\frac{\partial L}{\partial v_{t,j}} = \sum_{k=1}^{d^{(t+1)}} \frac{\partial a_{t+1,k}}{\partial v_{t,j}} \frac{\partial L}{\partial a_{t+1,k}} = \sum_{k=1}^{d^{t+1)}} w^{(t+1)}_{jk} \sigma^{(t+1)}_k$$

It turns out that computing the sensitivity vector for the layer t and the node j you need to compute

$$\delta^{(t)}_j = \sigma\prime(a_{t,j}) \sum_{k=1}^{d^{(t+1)}} w^{(t+1)}_{jk} \delta^{(t+1)}_k$$

In particular, $\sigma\prime(a_{t,j})$ depends on the function σ chosen. Moreover, to compute the $\delta^{(t)}_j$, according to the formula I need to have the sensitivity vector of the layer $(t+1)$, $\delta^{(t+1)}_k$,$1 \leq k \leq d^{(t+1)}$. Given such values, we use the **backpropagation algorithm**. Starting from the last layer, you need to compute the sensitivity of the loss function and propagate in the previous layers trough chain rule. Therefore, to start to backpropagate we only need $\delta^{(L)} = \frac{\partial L}{\partial a^{(L)}}$. This depends on the loss function used.

Backpropagation algorithm The pseudo code for the algorithm to compute sensitivities $\delta^{(t)}$,$\forall t$, for a given data point(x_i, y_i) is

```
Input: data point (x_i, y_i); NN (with weights w_ij^(t), for 1 ≤ t ≤ T);
Output: δ^(t) for t = 1,...,T;
compute a^(t) and v^(t) for t = 1,...,T;
δ^(T) ⟵ ∂L/∂a^(T);
for t = T − 1 downto 1 do:
        δ_j^(t) ⟵ σ'(a_{t,j}) Σ_{k=1}^{d^(ℓ+1)} w_jk^(t+1) δ_k^(t+1) for all t = 1,...,T;
return δ^(1),...,δ^(T)
```

The full backpropagation algorithm where we also compute the weights is the following

```
Input: data point (x_1, y_1),...,(x_m, y_m);
                NN (weights initialized randomly);
Output: NN with weights w_ij^(t);
initialize w_ij^(t) for all i,j,t;
for s ⟵ 0,1,2,... do /* until convergence */
        pick (x_k, y_k) at random for training data;
        /* forward propagation */
        compute v_{t,j} for all j,t from (x_k, y_k);
        /* backward propagation */
        compute δ_j^(t) for all j,t from (x_k, y_k);
        /* update weights */
        w_ij^(t) ⟵ w_ij^(t) − η v_{t−1,i} δ_j^(t) for all i,j,t;
        if converged then return w_ij^(t) for all i,j,t;
```

Before using the algorithm, we need to do some preprocessing on the data by normalizing and centring all the inputs. The initialization of the weights $w_{ij}^{(t)}$ is done trough the use of a gaussian distribution $N(0, \sigma^2)$ with small variance. Setting all the weights to zero would lead to the same final weight on every neuron. Since the training error has many local minima, we run stochastic gradient descent for different (random)initial weights.

Regularized NN Instead of training a NN by minimizing the training error $L_S(h)$, we find h that minimizes:

$$L_S(h) + \frac{\lambda}{2} \sum_{i,j,t} (w_{ij}^{(t)})^2$$

where λ is the regularization parameter. This is called *squared weight decay regularizer*. To find the hypothesis h we use SGD or improved algorithms. Different kind of regularization are available.

14 Deep Learning

Deep learning is a class of specific neural network. Deep learning tries to build and learn a hierarchical representations of the data using a neural network. The main inspiration is based on biology. The main difference between Machine Learning and deep learning is that, whereas in traditional machine learning we have a model can't learn features directly from the data, deep learning extract features autonomously by learning a NN with more than one layer.

This is why deep learning is often referred to as *end-to-end learning*. One example of feature

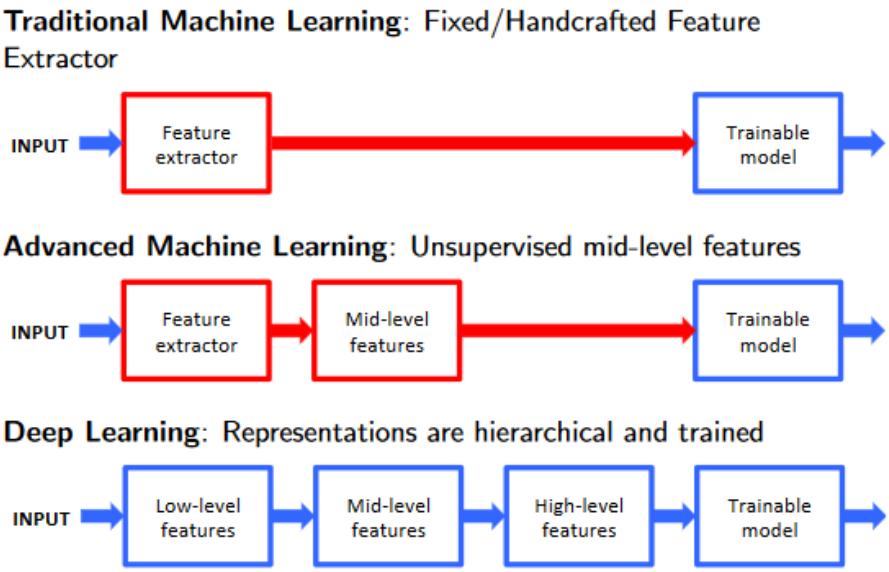


Figura 20: Hierarchical Representations and ML

recognition on an image can be seen in picture [21]. Deep learning has been designed to solve

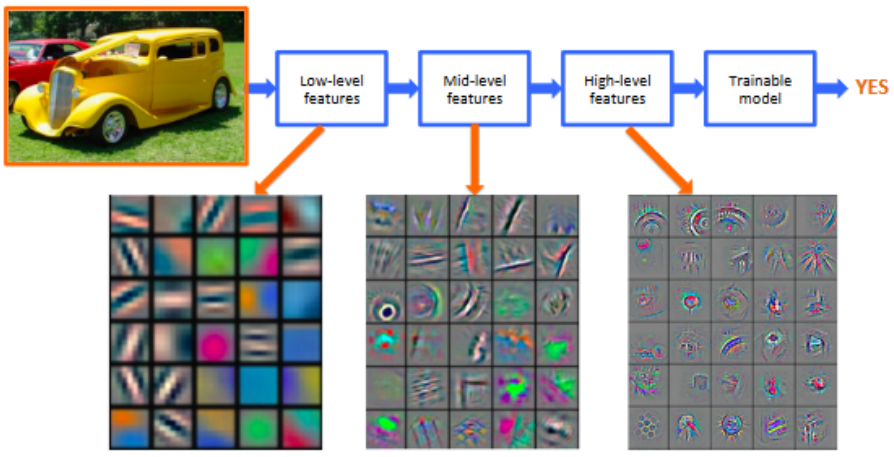


Figura 21: Feature extraction from an image

some of the issues of the traditional neural networks. One of the issue is that there is a huge number of edges, which lead to a huge number of weights (one weight for each edge in a fully-connected NN). This might lead to overfitting. The second issue is that the structure of the domain is not taken into account. We'll now look at two specific architectures for deep learning:

- **Convolutional Neural Networks (CNN):** Those are specialized deep learning architecture designed for computer vision and images. Those works well when there is an input with a lot of structure;

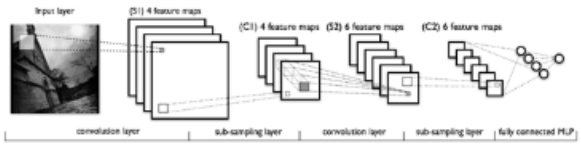


Figura 22: convolutional neural network

- **Recurrent Neural Networks (RNN):** Used mostly for time series analysis, speech recognition and machine translation. Those are not feed-forwards but also have edges that "go back", which keep a status that get propagated.

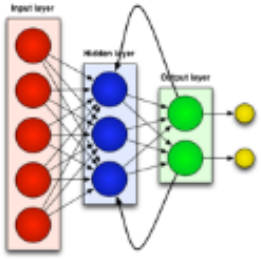


Figura 23: Recurrent neural network

14.1 Convolutional Neural Network

Convolutional Neural Networks are NN that employ the convolution operation in place of general matrix multiplication in at least one of their layers. In particular, CNNs are a specialized kind of neural network for processing data that has a known grid-like topology (for example an image). Let's assume we have a bi-dimensional input I and a two-dimensional function K. Then the (discrete) convolution S of I and K is:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n)$$

Discrete convolution can be viewed as (a sort of) multiplication by a matrix, but the matrix has several entries constrained to be equal to other entries. In particular, K is called *kernel*. Note that the kernel is learned from the training data. Learning kernels is useful because their application can lead to different transformation of data. In CNNs, the matrix $K(m,n)$ is zero but for small values of m and n. This means we can consider the convolution as in figure [24]. The application of the convolution operator to the input and the kernel is obtained by doing the entry-wise product of a sub-matrix of the input times the kernel and doing the sum of the corresponding results. he use of convolution leads to useful properties:

- sparse interaction in the network: instead of having all possible edges, you only have some;

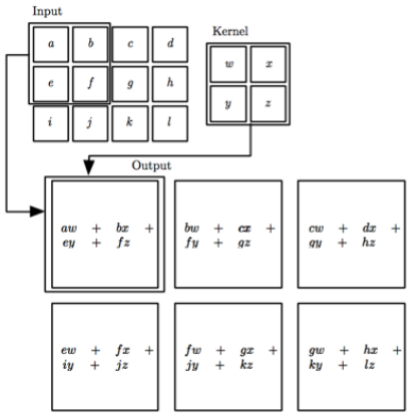
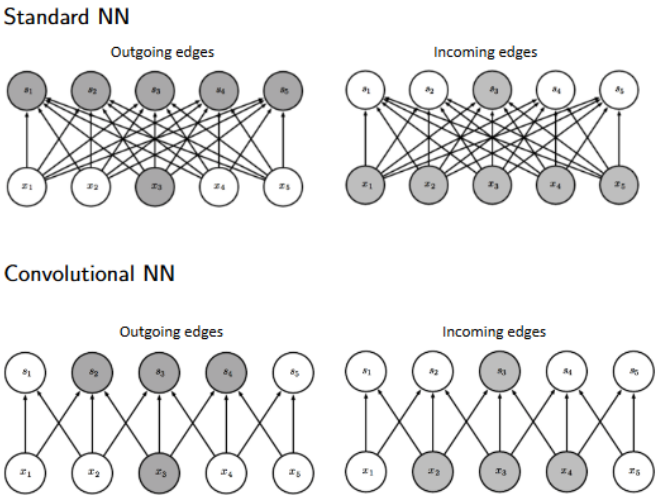


Figura 24: Convolution in CNNs

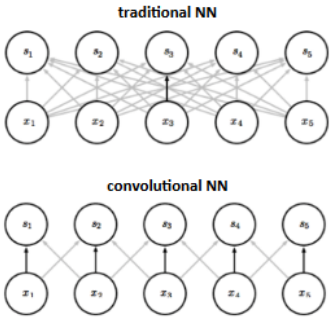
- there is parameter sharing between different edges on the network;
- it allows for an equivariant representation.

Sparse interactions and parameter sharing reduce the number of parameters. This can be thought as a form of regularization. The equivariant representations is instead useful for images (same image/object moved in space).

Sparse interaction Having sparse interaction means that many edges do not exist in the network, which therefore have zero weights. Therefore, in convolutional neural network, a node has much fewer outgoing edges and incoming edges overall.



Parameter sharing With parameter sharing we're using the same parameter for more than one function in a model. Therefore: rather than learning a separate set of parameters for every input location, we learn only one set of parameters (the matrix K). By definition of convolution,



the output is always going to be smaller than the input if the kernel is not 1×1. Sometimes you want to apply the kernel the same number of times to each input or not to reduce dimension of next layer). Therefore there is a need to add extra (virtual) inputs. This is done trough the use of *zero-padding*.

Feature map Now we have to build a feature map for the input. We can learn multiple kernels at one time. Instead of just applying the standard transformation shown before, we can apply a whole different transformation to learn a different map to represent the input. This corresponds in a neural network to say that what we're going to do applies to get one set of output vertices, but this can be repeated for a different set of output vertices. The edges between the two sets are disjoint. For example, i can think of having learned two different features map (orange and red in picture [22]). All the orange units compute the same function but with a different input windows, and orange and red units compute different functions.

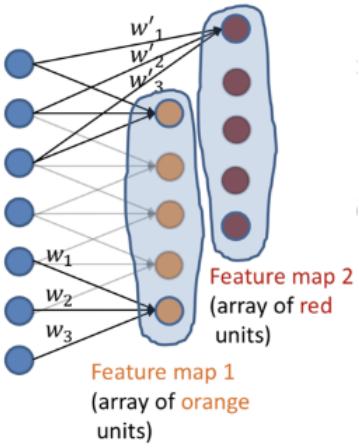


Figura 25: Multiple Feature Maps

Convolutional Layer In a CNN there are **convolutional layers**(which implement convolution) and other types of layers (like sub-sampling layers or fully-connected layers). In particular, the convolutional layer models consists of several filters/kernels. After a convolutional layer, the activation function commonly used is the **ReLU** (Rectified Linear Unit). Then we have a pooling

layer, often combined with subsampling. At the end, there is a fully connected NN (MLP - Multi Layer Perceptron), which learns from the features extracted at the previous hidden layers.

ReLU The ReLU (Rectified Linear Unit) is defined as

$$\sigma(z) = max\{0, z\}$$

This is useful due to some properties of the ReLU:

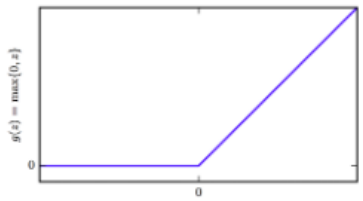


Figura 26: ReLU function

- It helps convergence in SGD;
- It can be proved that it doesn't hurt expressiveness;
- It prevents the *vanishing gradient function*. This is due to the fact that with small/large z the gradient is essentially equal to zero. This means that the weights are not going to change or will took a lot of time;
- Lead to sparse activation;
- It is more efficiently computable than other activation functions.

In general, the layer with ReLU activation is sometimes called *detector layer*.

Pooling In **pooling** you replace the output of the network at a given layer with a summary statistic of the nearby output. Therefore, instead of only looking at the output of a certain node, I also look at their neighbours. Some common pooling strategies are:

- **max pooling:** maximum of a rectangular neighbourhood;
- **average pooling:** average of a rectangular neighbourhood;
- weighted average based on the distance from the network location;
- ...

In particular, max pooling introduces invariance to local translation: many neurons have the same output value even if input values are "shifted" a bit. Pooling has invariance to local translation. This is useful if we care more about whether some feature *is present* than exactly *where it is* or *how it appears*. Often pooling is combined with *subsampling*: it skip the application of pooling for a given number of steps. This is called *stride*. Pooling and subsampling have some nice properties:

- almost scale invariant representation;

- makes the input representations (feature dimension)smaller and more manageable;
- reduces the number of parameters), thus it controls overfitting;
- reduces computation.

14.2 CNN Details

To learn the parameters for a CNN, we use Stochastic Gradient Descent with backpropagation. However, we know that ReLU is not differentiable in zero. Therefore, we use the right/left/average derivative. We also have to make sure to do input normalization so that our input is always between $[0, 1]$. The initialization of the weights is done by setting the bias weight to zero and the neuron's weight as a random value in $[-\frac{1}{d}, \frac{1}{d}]$. We can also use mini-batches: at each iteration of SGD we calculate the average loss on k random examples for $k > 1$. The advantage of doing this is that it reduces the variance of the update direction, leading to a faster convergence, and don't loose a lot in running time. The update rule is not the standard SGD but it's something slightly different (i.e *ADAM*). Techniques to help avoiding overfitting are also used. The loss function we use is a particular function we're going to see later.

ADAM *Adaptive Moment Estimation Iterative method* (ADAM) is a variant of SGD. Let's assume $g^{(t)} \in \nabla f(x)$. Then update at each iteration t:

1) $m^{(t)} \longleftarrow \beta_1 m^{(t-1)} + (1 - \beta_1) g^{(t)}$;
2) $v^{(t)} \longleftarrow \beta_2 m^{(t-1)} + (1 - \beta_2)(g^{(t)})^2$;
3) $\hat{m}^{(t)} \longleftarrow \frac{m^{(t)}}{(1-\beta_1^t)}$;
4) $\hat{v}^{(t)} \longleftarrow \frac{v^{(t)}}{(1-\beta_2^t)}$;
5) $w^{(t+1)} \longleftarrow w^{(t)} - \eta \frac{\hat{m}^{(t)}}{\hat{v}^{(t)}+\delta}$;

14.3 Dropout

Dropout is a technique that provides a computationally inexpensive but powerful method of regularizing a broad family of models. Dropout is *bagging* method, because it train multiple models and evaluate multiple models on each test example. This is however impractical when each model is a large neural network, because training and evaluating such networks is costly in terms of runtime and memory. Dropout provides an inexpensive approximation to training and evaluating a bagged ensemble of many neural network. It trains the ensemble consisting of all subnetworks that can be formed by removing nonoutput units (setting their weights to zero) from an underlying base network o train with dropout we use a minibatch-based SGD. Each time an input example is loaded into a minibatch:

1 Randomly sample a different binary mask to apply to all the input and hidden units in the network;

2 Mask for each unit is sampled independently from all the others;

3 Probability of sampling a mask value of one (causing a unit to be included) is a hyperparameter fixed before training begins;

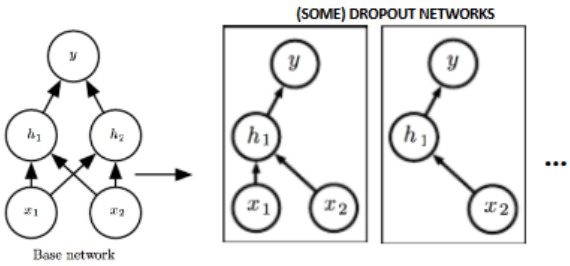


Figura 27: Example of dropout

4 Run forward propagation, back-propagation, and learning update.

Typically input units are included with probability 0.8 while hidden units are included with probability 0.5.

14.4 Early stopping

Early stopping consists in using validation error to decide when to stop training. We stop when monitored loss has not improved after n (n is called *patience*) subsequent epochs.

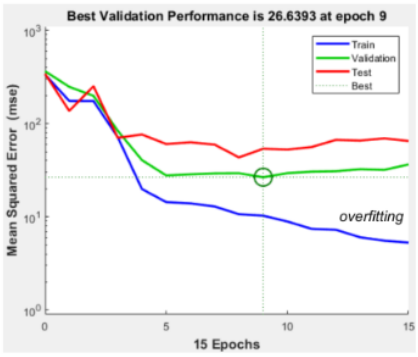


Figura 28: Example of early stopping

14.5 Data Augmentation

Another important technique is called Data Augmentation. The idea is to overcome the large amount of data needed for deep learning trough generating additional samples trough adding a little bit of variance to the data and "virtually" increasing number of training sample. This is done by artificially adding some noise, or apply random transformations o crop part of the data such as rotation, resize and Custom transformation (i.e contrast, saturation, ...). This works really well in practice in images and signals domain.

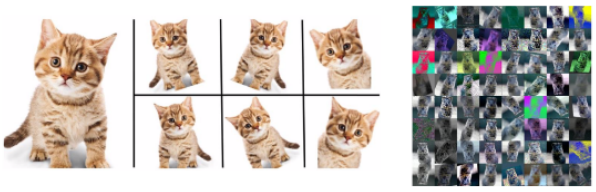


Figura 29: Example of data augmentation

14.6 Loss Function

In classification problem it's usually used *cross entropy*. To use cross entropy loss you need a model where the prediction is not just 0 or 1 but it's a value between 0 and 1. To do this we can use sigmoid activation function for output. Cross Entropy loss is defined as

$$\ell(h,(x,y)) = -y \log h(x) - (1-y)\log(1-h(x))$$

Cross entropy is a convex function, therefore SGD works better. Also, the best hypothesis h with respects to the cross entropy loss is

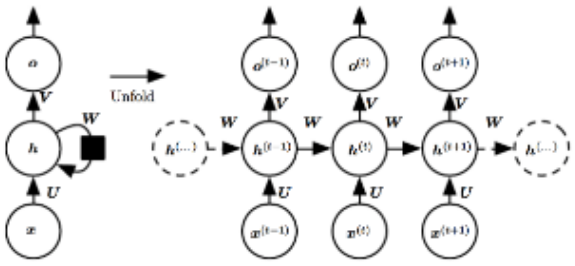
$$h(x) = Pr[y = 1|x]$$

which is related to Bayes optimal predictor. By using Cross Entropy loss we are trying to find an hypothesis as close as possible to the Bayes optimal predictor. For multi-class classification instead we have k classes $(0,\ldots,k)$. The output is a vector y with $y_i = 1$ if the correct class is i, or zero otherwise. In particular, $h(x) \in (0,1)^d$ with $h_i(x)$ equals to the probability that the label of x is i, with $1 \leq i \leq k$. The CE loss function for multiclass classification is then defined as

$$\ell(h,(x,y)) = -\sum_{i=1}^{k} y_i \log(h_i(x))$$

14.7 Recurrent Neural Networks

The main idea behind Recurrent Neural Networks (RNNs) is that whenever you have sequential data (like text), you can think of the input data as $x^{(t)}$ where $x_i^{(t)}$ is the input at the instant t. Then to capture the sequentiality we are going to make sure that the hidden state h at the time t depends both on the input but also on the hidden state at the previous time $h^{(t-1)}$. The new



hidden state is therefore:

$$h^{(t)} = f(h^{(t-1)}, x^t)$$

The recurrent neural network above is universal (can compute any function computable by a Turing machine). RNNs can be used in text/speech recognition, machine translation, ...

15 Unsupervised learning

While in *supervised learning* we use as data both the input and their labels, in **unsupervised learning** you only have the set of features $(x_1, \ldots, x_m)$ but no target values. Here we aren't looking for *predictions*, but interesting *structures* on the data or, equivalently, to organize it in some meaningful way. We are going to see focus on unsupervised learning techniques such as *clustering* and *dimensionality reduction*. There are also other general techniques that won't be covered but that you're suggest to read by yourself.

15.1 Clustering

Definition 26

Clustering is the task of grouping a set of objects such that similar objects end up in the same group and dissimilar objects are separated into different groups.

The one above is just a general definition. Different definitions have been proposed that may lead to different types of clustering. We will see only few of them. In particular, clustering has some inherent difficulties:

- **Similarity is not transitive**: saying we want to put similar object in the same group and dissimilar object in different groups may contradict each other as goals;
- In general **we do not have a ground truth** to evaluate our clustering, since we are doing unsupervised learning. In practice, however, a given set of objects can be clustered in various different meaningful ways.

A model for clustering Let's formulate the clustering problem more formally:

- **input:** We assume we have a set of elements X and a *distance* function $d : X \times X \longrightarrow \mathbb{R}_+$. The function *distance* is :
  - symmetric: $d(x, x') = d(x', x)\ \forall x, x' \in X$;
  - $d(x, x) = 0\ \forall x \in X$;
  - (often) d satisfies the triangle inequality: $d(x, x') \leq d(x, z) + d(z, x')$.
- **output:** we want to obtain a partition of the set X into clusters, that is $C = (C_1, \ldots, C_k)$ where
  - the union of the C_i is $\cup_{i=1}^{k} C_i = X$;
  - no point is in two clusters: $\forall i \neq j,\ C_i \cap C_j = \emptyset$.

In short, given a set of points, partition them into k groups called clusters. Note that sometimes (often) the input also includes the number k of clusters to produce in output. In most formulation k is part of the input. Also, sometimes the output is not really a partition but it's an object from which we can obtain the partition. In particular, a common output is the so-called **dendrogram**. A dendrogram is a tree diagram that shows how you can arrange points in two clusters for different

numbers of cluster. Sometimes, instead of having a distance function you have a *similarity* function. That is a function $s : X \times X \longrightarrow \mathbb{R}_+$ that is once again symmetric ($s(x, x') = s(x', x)\ \forall x, x' \in X$) but now has $s(x, x) = 1\ \forall x \in X$ instead of being equal to zero. The choice between using distance or similarity depends on the type of data. Given the definition of clustering, is clear that one of the things to pick is the distance function.
We can consider two classes of Algorithms for Clustering: *Cost minimization algorithms* and *Linkage-based algorithms.*

15.2 Cost Minimization Clustering

This is the most common method of clustering and it's based on the idea of finding the partition (=clustering) of minimal cost given a defined cost function over possible partitions of the objects. The assumptions we are going to make are that the data points $x \in X$ comes from a larger space X', that is $X \subseteq X'$. This is useful because in some algorithms you refer to points that are not in the input but are part of this X'. Again we assume we have a distance function $d(x, x')\ \forall x, x' \in X$. For simplicity, we'll also assume by the time being that $X' = \mathbb{R}^d$ and the distance function is given by the euclidean distance $d(x, x') = \|x - x'\|$.
Now we are going to see how we can define different clustering problems based on the cost function that we define. The first model for clustering we are going to look at is

K-means clustering **K-means** is one of the main way to do clustering. Here we have in input our data points $(x_1, \ldots, x_m)$ and the number of groups we want to obtain $k \in N^+$. The goal is to find the partition $C = (C_1, \ldots, C_k)$ of $x_1, \ldots, x_m$ and the *centroids* $\mu_1, \mu_2, \ldots, \mu_k$ (those points are outside the input) with $\mu_i \in X'$ centroid for C_i, $\leq i \leq k$ that minimizes the **k-means objective** (cost)

$$\sum_{i=1}^{k} \sum_{x \in C_i} d(x, \mu_i)^2$$

Therefore, given the points in input and a value k, I want to partition them in k groups and assign centroids to each of the group such that when I look at the distances of all the points to their centroid in their cluster, take the square of the distances and sum them up I obtain the minimum value. Other possible solutions are obtained by having the same partitions but choosing different centroids. One different cost-based clustering formulation where the objective function changes is the *k-medoids objective*:

$$min_{\mu_1, \ldots, \mu_k \in X} \sum_{i=1}^{k} \sum_{x \in C_i} d(x, \mu_i)^2$$

where now I'm forced to pick centroids in the input. Another formulation is the so-called *k-median objective*:

$$min_{\mu_1, \ldots, \mu_k \in X} \sum_{i=1}^{k} \sum_{x \in C_i} d(x, \mu_i)$$

where we just take the distance as it is instead of squaring it.
To solve the k-mean clustering problem, we use the following proposition

Proposition 19

Given a cluster C_i, the center μ_i that minimizes $\sum_{x \in C_i} d(x, \mu_i)^2$ is

$$\mu_i = \frac{1}{|C_i|} \sum_{x \in C_i} x$$

By knowing the partitions, I can therefore find the optimal centroids. Based on this proposition, we can think about a naive way to solve k-means by trying all the possible partitions of the m points into k clusters, evaluate each partition, and find the best one. The efficiency depends on the number of partitions of m points into k clusters: the number of ways in which we can partition a set of m objects into k subsets is

$$S(m,k) = \frac{1}{k!} \sum_{j=0}^{k} (-1)^{k-j} \binom{k}{j} j^m$$

which is a Stirling number of the second kind. Thus the brute-force algorithm complexity is exponential. However, there are no polynomial time algorithms. Finding the optimal solution for k-means clustering is computationally difficult (NP-hard). This is true for most optimization problems of cost minimization clusterings (including k-medoids and k-median). An heuristic for k-means is the

Lloyd's Algorithm Lloyd's Algorithm is the way we solve k-means in practice. The pseudo-code is the following:

```
Input: data point X = (x_1, ..., x_m); k ∈ N^d;
Output: clustering C = (C_1, ..., C_k) of X; centers μ_1, ..., μ_k
                with μ_i center for C_i, 1 ≤ i ≤ k;
randomly choose μ_1^(0), ..., μ_k^(0);
for t ⟵ 0,1,2,... do: /* until convergence */
        for i = 1, ..., k: C_i ⟵ {x ∈ X : i = argming_j d(x, μ_j^(t)){;
        for i = 1, ..., k: μ_i^(t+1) ⟵ 1/|C_i| Σ_{x∈C_i} x;
        if \textit{convergence reached} then
                return C = (C_1, ..., C_k) and μ_1^(t+1), ..., μ_k^(t+1);
```

The algorithm choose a random centroid, assign each point to the closest centroid and update the centroids. we'll then iterate until we reach convergence. The most commonly criteria used for convergence are

$$\sum_{i=d}^{k} d(\mu_i^{(t+1)}, \mu_i^{(t)}) \leq \varepsilon$$

which is the sum of the changes over the centroid must be at most ε. A most common variation doesn't consider the sum but the maximum:

$$max_{1 \leq i \leq k} d(\mu_i^{(t+1)}, \mu_i^{(t)}) \leq \varepsilon$$

Once I find stable centroid the algorithm stop. Let's reason now about complexity. The assignment of points $x \in X$ to clusters C_i takes time $O(kmd)$, while the computation of centres μ_i takes time $O(md)$. Therefore, the total complexity is on the order of $O(tkmd)$ assuming convergence after t iterations.

15.3 Linkage-Based Clustering

This algorithm build a clustering starting from the "simplest" one. In particular, the simplest one is when each point is it's own cluster. Until *termination condition*, we repeatedly merge the "closest" clusters of the previous clustering. With this algorithm, we need to specify two parameters: the way we define *distance* between cluster, and the *termination condition*.

distance Different distances $D(A, B)$ between two clusters A and B can be used, resulting into different linkage methods:

- **single linkage:** $D(A, B) = min\{d(x, x\prime) : x \in A, x\prime \in B\}$. This is the minimum distance between a pairs of points in different clusters;
- **average linkage:** $D(A, B) = \frac{1}{|A||B|} \sum_{x \in A, x\prime \in B} d(x, x\prime)$. This is the average of the distances between the pairs of point in two different clusters;
- **max linkage:** $D(A, B) = max\{d(x, x\prime) : x \in A, x\prime \in B\}$. This is the maximum distance between a pairs of points in different clusters.

Termination Common termination conditions are:

- The data points are partitioned into k clusters. I keep merging clusters until i'm left with k cluster;
- I can specify in input a parameter r and I stop when the minimum distance between pairs of clusters is $> r$;
- All points are in a cluster, we produce in output the dendogram.

Definition 27

A **dendrogram** is a particular tree, with input points $x \in X$ as leaves, that shows the arrangement/relation between clusters.

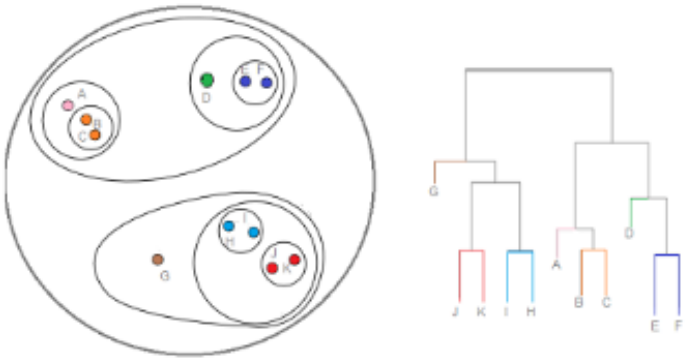


Figura 30: example of dendogram

15.4 Choice of number k of clusters

Choosing the number k of clusters is not an easy task. A common approach is to run the algorithm for different values of k based on previous knowledge (or random),obtaining a clustering $C = (C_1, \dots, C_k)$ for each value of k considered. Then i can use a *score* S to evaluate each clustering $C^{(k)}$, getting scores for each $S(C^{(k)})$ for each value of k. Finally I can pick the value of k (and clustering) with maximum score $C = argmax_{C^{(k)}}\{S(C^{(k)})\}$.
There are different types of score that depends on the feature of the clustering. One which is very common is the *silhouette*.

Silhouette Given a clustering $C = (C_1, \dots, C_k)$ of X and a point $x \in X$, let $C(x)$ be the cluster to which x is assigned to. Assume $|C_i| \geq 2\ \forall 1 \leq i \leq k$ Define:

$$A(x) = \frac{\sum_{x\prime \neq x, x\prime \in C(x)} d(x, x\prime)}{|C(x)| - 1}$$

Given a cluster $C_i \neq C(x)$, let

$$d(x, C_i) = \frac{\sum_{x\prime \in C(x)} d(x, x\prime)}{|C_i|}$$

and then we can define the minimum distance of x to another cluster as:

$$B(x) = min_{C_i \neq C(x)} d(x, C_i)$$

Then the **silhouette** $s(x)$ of x is defined as

$$s(x) = \frac{B(x) - A(x)}{max\{A(x), B(x)\}}$$

The main idea is that $s(x)$ measures if x is closer to points in its "nearest cluster" than to the cluster it is assigned to. The value of the silhouette will change between $\sim (-1, 1)$. When $A(x)$ is small (x is in a good cluster), $s(x)$ is going to be roughly 1. If I place x in a bad cluster (the average distance of x to the point in its cluster is much larger than the distance with the nearby cluster), $A(x)$ is going to be large while $B(x)$ is going to be small. In this situation, the value of $s(x)$ is going to be around -1. Therefore the score of a clustering is

$$S(C) = \frac{\sum_{x \in X} s(x)}{|X|}$$

The higher $S(C)$, the better the clustering quality.

16 Principal Component Analysis

Principal Component Analysis (PCA) is a method to "change the way you look at the data". Before talking about PCA is useful to introduce some notions of linear algebra we'll use afterwards:

16.1 Change of basis

Let $x_1, x_2, \ldots, x_m \in \mathbb{R}^d$ be our data points points. Then we can represent the data matrix as

$$D = \begin{bmatrix} x_1^T \\ x_2^T \\ \vdots \\ x_m^T \end{bmatrix}$$

Each point x_i lives in $\mathbb{R}^d$, which is the space spanned by the standard basis $e_1, e_2, \ldots, e_d$ where e_i is the vector of all zeros and a 1 in the i-th component. Usually we represents points in $\mathbb{R}^d$ with their representation with respect to the standard basis. We can look at a different representation for vectors. Let's consider d orthonormal (thus orthogonal) vectors $u_1, u_2, \ldots, u_d$, that is:

- $u_i^T u_j = 0 \ \forall i \neq j$;
- $\|u_i\| = 1 = u_i^T u_i, \ \forall i$.

Let's consider the matrix

$$U = \begin{bmatrix} u_1 & u_2 & \ldots & u_d \end{bmatrix}$$

Then for any point $x \in \mathbb{R}^d$, we can actually represent it with respect to this basis U. The vector x can be rewritten as $x = Ua = \sum_{i=1}^{d} a_i u_i$, where a is the vector of coordinates of x in basis $u_1, u_w, \ldots, u_d$:

$$a = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_d \end{bmatrix}$$

So this a vector is just a different way to represent x, where the coordinates are with respect to the new basis U. Since U is orthogonal, that means that the inverse of U is equal to the transpose:

$$U^{-1} = U^T \Rightarrow U^T x = U^T U a = a$$

16.2 Dimensionality Reduction

The goal of dimensionality reduction is to find a good representation of the data you have with a reduction of dimensionality of the data you are considering. In particular, if d is the dimension of the data, we want to find a good r-dimensional representation of D, with $r << d$. Let's see some more concepts of linear algebra:

Projection matrix Given the basis $u_1, \ldots, u_d$ for$\mathbb{R}^d$, let's consider the first r basis vectors among this $u_1, \ldots, u_r$ (that is a basis for a subspace of dimension r of $\mathbb{R}^d$) and project x on those. Projecting x in a subspace obtained by consider only r basis vectors corresponds to discarding the contribution of the remaining $d - r$ basis vectors. Therefore the projection in the new space x' will be defined as

$$x' = \sum_{i=1}^{r} a_i u_i = \begin{bmatrix} u_1 & u_2 & \ldots & u_r \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_r \end{bmatrix} = U_r a_r$$

It's easy to see that $a_r = U_r x$. Then the vector x' becomes

$$x' = U_r U_r^T x = P_r x$$

where $P_r = U_r U_r^T$ is the **projection matrix** for the subspace we are considering. Note that x' is a vector with d components which just lives in a subspace of dimension r. This projection a matrix has some interesting properties:

- P_r is symmetric;
- $P_r^2 = P_r P_r = P_r$.

Let's now define the

Error vector the error vector is the error that we makes if instead of considering a vector x we consider its projection x':

$$\varepsilon = \sum_{i=r+1}^{d} a_i u_i = x - x'$$

In particular, ε and x' are orthogonal.

Dimostrazione. $(x')^T \varepsilon = \sum_{i=1}^{r} \sum_{j=r+1}^{d} a_i a_j u_i^t u_j = 0$ □

Dimensionality Reduction Applications Dimensionality Reduction might seems like it's not useful since by using it we're losing information. However, This might be useful when the number of samples at your disposal is limited: here dimensionality reduction can reduce overfitting by reduce the number of features. While feature selection *discards* the features, dimensionality reduction still reduces the number of features, but now the feature used are not the original features any more, but are instead obtained as combination of the original ones, so hopefully we can retain most of the information on the data. Another case of use is to get rid of noise. Capturing the most important aspects of the data may help in reducing the noise on the data. One of the most important application is however *visualization*. Trough a reduction of the dimensionality of data, we can visualize the data in a more meaningful way. PCA is just one specific way to do dimensionality reduction.

16.3 PCA

The goal of Principal Component Analysis is to find an r dimensionality basis that best capture the variance in the data. So what I want to do is to project points which live in high dimensional space in a lower dimension space such that the variance in the data is conserved. I want to preserve the variance because it's (more or less) where the signal lies. To get the r-dimensional basis, we're going to look at the principal component of the data. In PCA, we can represent the first principal component as the direction with largest projected variance. Then the second element of the basis is going to be the orthogonal direction (with respect to the first principal component) with again the largest projected variance. Then the procedure reiterate for the remaining basis.

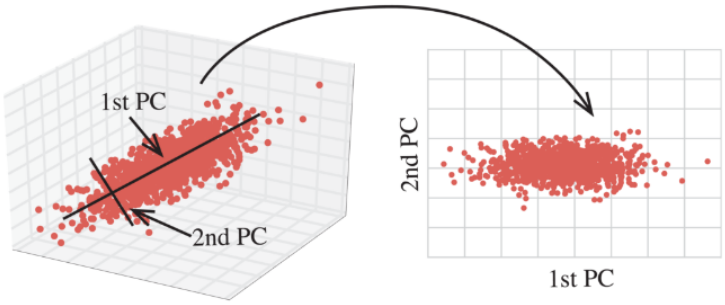


Figura 31: principal component analysis example

Computing the principal components Let's see how we can get the first r principal components. Firstly, we need to make some assumptions. We can assume that

- The data is given by matrix D, where the i-th row is the i-th input vector $x_i \in \mathbb{R}^d$;
- The data has been centred, thus the mean of the data is $\mu = \sum_{i=1}^{m} x_i = 0$, where 0 is a vector of dimension d.

Then to compute the first r principal components we proceed as follows:

1 We compute the (sample) covariance matrix of (centered) data D:

$$\Sigma = D^T D$$

2 We compute the eigenvalues $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_d \geq 0$ of Σ (all the eigenvalues of Σ are ≥ 0 since Σ is positive semidefinite);

3 Let $u_1, u_2, \ldots, u_d$ be the eigenvectors associated to $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_d$;

4 Then the first r principal components are the eigenvectors $u_q, u_2, \ldots, u_r$.

Therefore in PCA, the first r principal components are obtained as the eigenvectors associated to the r largest eigenvalues of the sample covariance matrix of the data once the data has been centred. Let's now look at how PCA compute the projected variance:

PCA projected variance Given the r components of the basis $u_1, u_2, \ldots, u_r$, which are in PCA the r eigenvectors associated to the r largest eigenvalues, and the projection matrix defined as $P_r = U_r U_r^T$. Let's call A the projected dataset, which is the matrix obtained by taking each point in the dataset projected in the subspace of dimension r: $a_i = U_r^T x_i$ for $i = 1, \ldots m$. Let's denote as $var(A)$ the variance of the projected dataset A. PCA maximizes $var(A)$, so any other choices of the basis that differs from what PCA choose will results in a variance of A which is smaller compared to the one we found trough PCA. To compute the variance of A, we are going to derive the variance for A given our choice of the basis and we want to be able to project the points that is the origin of $\mathbb{R}^d$ according to our choice of the basis. In particular, the projection

of the origin is the origin itself (the vector 0 with r components). Thus, we write:

$$var(A) = \frac{1}{m}\sum_{i=1}^{m}\|a_i - 0\|^2 = \frac{1}{m}\sum_{i=1}^{m}\left(U_r^T x_i\right)^T\left(U_r^T x_i\right) = \frac{1}{m}\sum_{i=1}^{m} x_i^T U_r U_r^T x_i =$$
$$\frac{1}{m}\sum_{i=1}^{m} x_i^T P_r x_i = \frac{1}{m}\sum_{i=1}^{m}\left(x_i^T\left(\sum_{j=1}^{r} u_j u_j^T\right)x_i\right) = \sum_{j=1}^{r}\left(\frac{1}{m}\sum_{i=1}^{m}(u_j^T x_i)(x_i^T u_j)\right) = \tag{36}$$
$$\sum_{j=1}^{r} u_j^T \Sigma u_j = \sum_{j=1}^{r}\lambda_j u_j^T u_j = \sum_{i=1}^{r}\lambda_j$$

Note that in the process we also showed that

$$var(A) = \frac{1}{m}\sum_{i=1}^{m} x_i^T P_r x_i$$

Let's now look at what is the

Mean Squared Error for PCA For this particular problem, we can define the MSE as the mean of the squared error for each point. This error for a point x_i when we use its projection x_i' is the difference between the two. We can derive as follows

$$MSE = \frac{1}{m}\sum_{i=1}^{m}\|x_i - x_i'\|^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - x_i')^T(x_i - x_i') = \frac{1}{m}\sum_{i=1}^{m}\left(\|x_i\|^2 - 2x_i^T x_i' + (x_i')^T x_i'\right) =$$
$$var(D) + \frac{1}{m}\sum_{i=1}^{m}(-2x_i^T P_r x_i + (P_r x_i)^T(P_r x_i)) = var(D) + \frac{1}{m}\sum_{i=1}^{m}(-2x_i^T P_r x_i + x_i^T P_r^2 x_i) =$$
$$var(D) + \frac{1}{m}\sum_{i=1}^{m}(-2x_i^T P_r x_i + x_i^T P_r x_i) = var(D) - \frac{1}{m}\sum_{i=1}^{m}(x_i^T P_r x_i) =$$
$$var(D) - var(A) = var(D) - \sum_{j=1}^{r}\lambda_j \tag{37}$$

Since the variance of A is the maximum variance, we are minimizing the MSE for the projection.

Choice of the dimension r One parameter that I need to look at in practice is the reduced dimension r. The choice of the parameter depends on the application, however there are some common rules that are used when doing PCA to make sure you're not discarding too much information. Note that $var(D)$ is invariant to base change and by running PCA with $r = d$. It turns out that

$$var(D) = \sum_{i=1}^{d}\lambda_i$$

Therefore I know how much variance is in my data, and I know how much variance is kept using r principal components. So one way to choose r, is to look at the fraction of variance of the data which is captured by PCA with r principal components. This quantity is equal to:

$$f(r) = \frac{\sum_{i=1}^{r}\lambda_i}{var(D)} = \frac{\sum_{i=1}^{r}\lambda_i}{\sum_{i=1}^{d}\lambda_i}$$

One way to pick r is to plot the fraction of variance over the number of principal components. A typical behaviour for this function is that it increase fast at the beginning an then smooth later on. One common rule is to pick the smallest r such that $f(r) \geq \alpha$ where α is a parameter of choice (usually $\alpha = 0.9$) which represent the fraction of the variance which is maintained by the data.