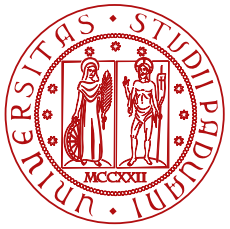


Università
degli Studi
di Padova

Dipartimento
di Ingegneria
dell'Informazione

Master degree in Computer Engineering

Machine Learning

University of Padova

Academic Year
2023/2024

Contents

1 Learning Model

1.1 A formal model (Statistical Learning)

Machine Learning is one of the branch of AI that's make previsions. We start from a huge quantity of data (big data) and we'll train our **learner**, so that it can give some previsions. The learner has access to:

- **Domain Set** X: it's the domain that contains all the possible instances that learner uses to make predictions.
  Usually $\vec{x}$ is a vector and each vector contains values of the features. In the example that we've seen in class which learner's aim is to predict if a post-graduation job will be fun or not based on the age of the graduate, the height of the graduate, ecc..., the domain contain all the graduates and for each vector $\vec{x} \in X$ we've a value of the features.
- **Label set** Y: it's our output, our response. Usually is a vector of two labels, like 1 and 0 or 1 and -1. In our example 1 corresponding to a funny job and -1 to the opposite.
- **Training data** S: It's usually a huge quantity of couple (x_i, y_i) like $S = ((x_1, y_1), (x_2, y_2), ..., (x_n, y_n))$ that we use to train our learner. It's a finite sequence of labeled domain points in XxY and represent the learner's input.
  In our example the training set corresponding to the twenty graduates with different value of features.
- **Learner's output** h: it's our function h that make the prediction of the next case that the learner will see.
  It's a function that for each $\vec{x}$ make a decision and choose a label of Y : $X \hookrightarrow Y$.
  It's called predictor or classifier and the prediction rule is provided by an algorithm A when a training set S is given to it or, in other world, when the learner is train.

- **Data-generation model** In a learner, there is a probability distribution D, like a Gaussian function, that generates vectors $\vec{x} \in X$, that will be labeled by a function f that's not known to the learner. For each x_i there will be $y_i = f(x_i)$.
  All of this point is contained in the training set S. In other words is how the training set is generated.

- **Measures of success**: it's a value that indicate the error of a classifier: probability that it doesn't predict the correct label on a new random data point $\vec{x}$

There is another measure of quality that we also use for creating the algorithm: the **loss**. With loss we indicate the probability that the prediction is incorrect respect the real label of the function f.
Given a domain subset $A \subseteq X$, $D(A)$ is the probability of observe a point $x \in A$. We refer to A as an event and we could express it with a function $\pi : X \hookrightarrow \{0, 1\}$ that is:

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

and we have $P_{x-D}[\pi(x)] = D(A)$.
Error of prediction rule: Given an hypothesis $h : X \hookrightarrow Y$, error is:

$$L_{D,f} = P_{x-D}[h(x) \neq f(x)] = D(\{x \ : \ h(x) \neq f(x)\})]$$

where L is the loss made on the true label function f and on the probability distribution D and is the probability that the label predicted by model h is different respect true label.
D says that given that x, the label made by h is uncorrect.
It's also called **generalization error, true error**. In Figure 1.1 the learning process (simplified).

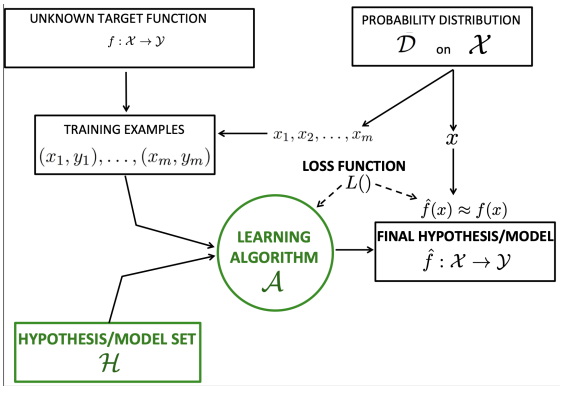


Figure 1.1: Diagram of a machine learning learner

We want to replicate the unknown target function f by training the model using training samples and hypothesis h that comes from class of hypothesis H.
The class of hypothesis H has the aim to restrict the research of the best model (for example linear models).
We assume that the features come from an unknown probability distribution D.
By using learning algorithm A we want to have a final model $\hat{f}$ that is similar to f. With the loss function L we can measure how much they're similar and it's also useful for the learning algorithm.
There are different types of learning:

- **supervised learning** if the y_i labels are known and the training set is composed by couple of points x, y
- **unsupervised learning** with the training set composed only by x points

There is also be two different types of output: Y **discrete or continuous**. In Figure 1.2 a scheme of these concepts.

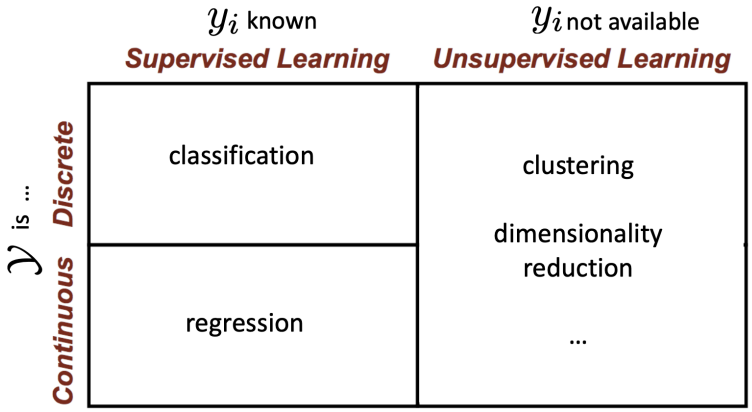


Figure 1.2: Difference between types of learning

1.1.1 Empirical Risk Minimization

The **learner output** h_S is a function that goes from instances to set of label i.e. $h_S : X \hookrightarrow Y$.
The goal for the learner / machine is to find a good hypothesis h_s that minimizes the generalization error $L_{D,f}(h_s)$
It's called h_S because it depends on the training set or rather on the data.
Remember that the distribution D and f are unknown and this is the difficult of

the problem. Since $L_{D,f}(h_s)$ is unknown it became a difficult problem.
Instead of trying to know the generalization error, we could concentrate the error on the training data, and choose h_S with minimum value of that.
The training error $L_s(h)$, for a given hypothesis h, is defined in this way:

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

that is the ratio between number of the instances of S and the cardinality of the set of point in S which are missclassified. The error on training set is also called **empirical error** or **empirical risk** or **training error**.
The **Empirical Risk Minimization** produces in output h that minimize $L_s(h)$.
However, choose h_s that minimize empirical error it is not enough. Consider the example shown in Figure 1.3 below.

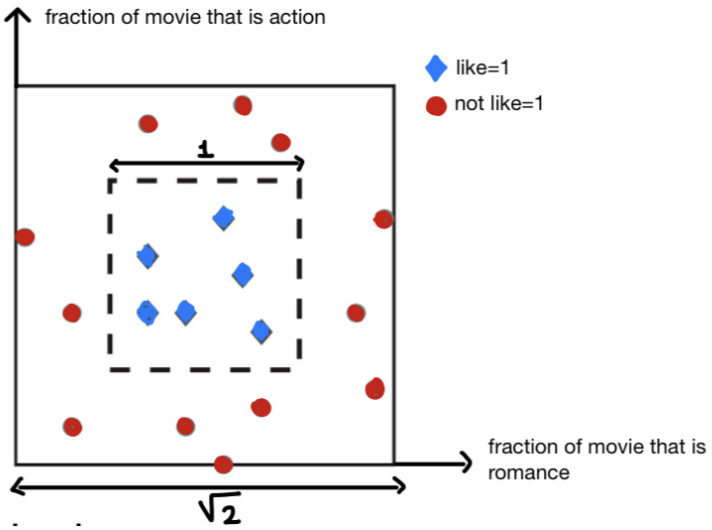


Figure 1.3: Example

Assume D and f such that:

- instance $\vec{x}$ is taken uniformly (with same probability) at random in the inner square so $D(\vec{x_i})$ is equal $\forall i \in \{1, ..., m\}$
- label is 1 inside the inner square (that is label function), 0 otherwise

The classifier given h_s says that if $\vec{x}$ is in the training set gives y, 0 otherwise. This means that it commits an error when the instance is not in the training set (inner square).
It has $L_s(h_S) = 0$ because it predicts correctly all the known data. But it's not a

good predictor because it makes an error whenever there is a point not in the training set but is in the inner square.
So the generalization error is the probability that the instance it's in the inner square so $L_{D,f}(h_s) = 1/2$.
Good results on training data but poor generalization error has a name that is **overfitting**.
We will search for conditions under which there is guarantee that ERM doesn't overfit.
To avoid this problem we apply ERM over a **restricted set** of hypothesis H that is called an hypothesis class. All $h \in H$ are functions that goes from $X \hookrightarrow Y$. We have to define which is the best class of hypothesis for each problem. We could define:

$$ERM_H \in argmin_{h \in H} L_S(h)$$

that is the model picked by H class of hypothesis which minimize the training set error.
Which class of hypothesis doesn not lead to overfit?
Assume H as a finite class and let be h_S the output of $ERM_H(S)$ or in other words the hypothesis chosen from the class H considering the training set S.
We do some important assumptions:

- **Realizability**: there exists $h^* \in H$ such that $L_{D,f}(h^*) = 0$ that implies also $L_S(h^*) = 0$ because if we don't make any error means there are not error also in the training set
- **i.i.d**: examples in the training set are independently and identically distributed according to D that is $S \approx D^m$

From this two assimption follows that h^* has $L_S(h^*) = 0$ The question now is: can we learn, using ERM, h^*?

1.1.2 PAC learning

PAC learning is the acronym of **Probably Approximately Correct** learning. Since the training data comes from D we can:

- only be **approximately** correct
- only be **probably** correct

because bad events can always happen. There are two parameters that indicate the goodness of the learner:

- **accuracy parameter** ϵ: we are satisfied with a good h_s if $L_{D,f}(h_s) \leq \epsilon$. The result is better if ϵ is smaller
- **confidence parameter** δ: want h_s to be a good hypothesis with probability $\geq 1 - \delta$

Theorem: Let H be a finite hypothesis class. Let $\delta, \epsilon \in (0, 1)$ and m a natural number such that

$$|S| = m \geq \frac{\ln \frac{|H|}{\delta}}{\epsilon}$$

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

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

For the **proof** see the notebook.
The theorem says a banal thing: lower is the error and higher is the cardinality of class H, higher is the number of training data that we need.
Definition of PAC learnability: A hypothesis class H is PAC learnable if there exist a function $m_H : (0, 1)^2 \hookrightarrow \mathbb{N}$ and a learning algorithm such that for every $\delta, \epsilon \in (0, 1)$, for every distribution D over X and for every labeling function $f : X \hookrightarrow \{0, 1\}$, if the realizability assumption holds with the respect to H, D, f, then when running the learning algorithm on $m \geq m_H(\delta, \epsilon)$ i.i.d examples generate by D and labeled by f (that means enough data), the algorithm returns a hypothesis h such that, with probability $\geq 1 - \delta$, value $L_{D,f} \leq \epsilon$.
In other words a class is PAC learnable if we could find in that class a good hypothesis.
It's important to define m_H that's the minimal integer number of data that satisfies the requirements and it's called **sample complexity of learning H**.
In other words say how many examples are required to guarantee a PAC solution. It also depends on properties of the hypothesis class H. From this definition born a **corollary** that is: every finite hypothesis class is PAC learnable with sample complexity $m_H(\delta, \epsilon) \leq int(\frac{\ln \frac{|H|}{\delta}}{\epsilon})$

1.1.3 Agnostic PAC Learning

We can relax the realizability assumption, that's too strong in many applications and sometimes there could be a problem: being the label full determined by the instance x, I could not have same instances x with different label in the

data training.
The **relaxation** of this is made changing $D : D$ is a probability distribution over $X\ x\ Y$ or rather a joint distribution over domain points and labels.
Now given $\vec{x_i}$, label y is obtained according to a conditional probability $P[y|\vec{x_i}]$, so depends also to the probability of encounter the instance $\vec{x_i}$ that is labeled with 1, and could exists another instance that is the same that is labeled with -1. This is **Agnostic PAC Learning** and it's quite different the notion of true error.
True error is now:

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

and the **empirical risk** is, as before:

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

L_S is like to say that probability for a pair of point (x_i, y_i), taken uniformly, the event $h(x_i) \neq y_i$ holds. So the expectation $E[L_S(h)] = L_D(h)$ from the moment that the training set is a subset of all the data of the domain.
We can think it also whit the example of the flipping coin: the generalization error says with which probability my hypothesis h make a wrong prediction. In the case of flipping coin suppose that is the probability to have head: to estimate this probability we can flip the coin many times and in expectation, the training error, is equal to the generalization error.
The learner's goal is to find the hypothesis that minimize the true error function and also here there is a best predictor that it's **Bayes Optimal Predictor** given a joint probability distribution D:

$$f_D(h) = 1 if P[y = 1|x] \geq 1/2\ else\ 0$$

so for any classifier g it holds that $L_D(f_D) \leq L_D(g)$.

Definition of Agnostic PAC learnability An hypothesis class H is Agnostic PAC learnable if there exist a function $m_H : (0, 1)^2 \hookrightarrow \mathbb{N}$ and a learning algorithm such that for every $\delta, \epsilon \in (0, 1)$, for every distribution D over XxY and for every labeling function $f : X \hookrightarrow \{0, 1\}$, if the realizability assumption holds with the respect to H, D, f, then when running the learning algorithm on $m \geq m_H(\delta, \epsilon)$ i.i.d examples generate by D and labeled by f (that means enough data), the

algorithm returns a hypothesis h such that, with probability $\geq 1 - \delta$, holds

$$L_{D,f} \leq min_{h' \in H} L_D(h') + \epsilon$$

. There are different types of learning problems based on the type of classification: binary classification, multiclass classification with > 2 labels or regression with the labels set is all the real numbers.
In **multiclass classification** the same concepts illustrated before apply. There are some differences for the regression model:

- Domain set $X \subset \mathbb{R}^p$ for some p
- Target set $Y = \mathbb{R}$
- Training data and learner's output as before
- Loss function

The loss used for classification can't be used for regression from the moment that if we have a "true" value $y = 13.75$ and a predicted $h(x) = 13.74$, the learner consider it as an error and the loss is 1.
Generalized Loss Function: Given any hypothesis set H and some domain $Z : XxY$, a loss function is any function $l : HxZ \hookrightarrow \mathbb{R}_+$ or in other word given an hypothesis and an instance the loss function gives a positive real number. We can define **risk function or generalization error** that is the expected loss of a hypothesis h with respect to D over Z i.e. over z objects pickled randomly according to probability distribution D.:

$$L_D(h) = E_{z \approx D}[l(h, z)] = E[L_S(h)]$$

The **empirical risk** that is the expected loss over a given sample and it's:

$$L_S(h) = 1/m * \sum_{i=1}^{m} l(h, z_i)$$

There are some common loss functions:
0-1 loss: $Z = XxY$

$$l_{0-1}(h, (x, y)) = \{0 \; if \; h(x) = y, \; 1 \; if \; h(x) \neq y\}$$

and it's common used in binary or multiclass classification.
Squared loss: $Z = XxY$

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

and it's common used in regression, where weights of big error is higher with this method.
However, in general, the loss function may depend on the application.
We can choose the loss function depending on the application. For example we have a classifier for a finger print that gives 1 is the finger is correct or -1 if the finger is incorrect.
There are two types of errors: false accepted and false reject. How do we choose the loss for those types of error? Depending on the application: if we have to protect a nuclear army the loss function of false accepted has a higher weight to show that is a worrying case.
We can recall the definition of agnostic PAC learnability for General Loss function:

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

$$L_{\mathcal{D}}(h) \leq \min_{h' \in \mathcal{H}} L_{\mathcal{D}}(h') + \varepsilon$$

where $L_{\mathcal{D}}(h) = \mathbb{E}_{z \sim \mathcal{D}}[\ell(h,z)]$

Figure 1.4: Agnostic PAC learning

2

UC and Bias TradeOff

2.1 Uniform Convergence

Recalling some concepts: given an hypothesis class H, the ERM learning paradigm works as follows: upon receiving a training sample S, the learner evaluates the risk of error of each h on the given samples and give as output an hypothesis h_s of H that minimizes the training error.
The hope is that this h_s that minimizes the empirical risk is a risk minimizer with respect to the true data probability distribution as well, i.e. data are picked correctly and h_s is picked correctly.
It suffices, to ensure that empirical risks of all members of H are good approximations of their true risk, that uniformly over all hypothesis, the empirical risk will be close to the true risk.
The meaning of **uniform convergence** is that: the training error or empirical risks of all members of H are good approximations or rather sufficient close to the true risk.
In Figure 2.1 there is an important definition:

A training set S is called ε-*representative* (w.r.t. domain Z, hypothesis class $\mathcal{H}$, loss function ℓ, and distribution $\mathcal{D}$) if

$$\forall h \in \mathcal{H}, |L_S(h) - L_{\mathcal{D}}(h)| \leq \varepsilon$$

Figure 2.1: ϵ-representative set

From this follows an important proposition with its proof in Figure 2.2:

Assume that training set S is $\frac{\varepsilon}{2}$-representative (w.r.t. domain Z, hypothesis class $\mathcal{H}$, loss function ℓ, and distribution $\mathcal{D}$). Then, any output of $\mathrm{ERM}_{\mathcal{H}}(S)$ (i.e., any $h_S \in \arg\min_{h\in\mathcal{H}} L_S(h)$) satisfies

$$L_{\mathcal{D}}(h_S) \leq \min_{h\in\mathcal{H}} L_{\mathcal{D}}(h) + \varepsilon$$

For every $h \in \mathcal{H}$:

$$\begin{aligned} L_{\mathcal{D}}(h_S) &\leq L_S(h_S) + \frac{\varepsilon}{2} \\ &\leq L_S(h) + \frac{\varepsilon}{2} \\ &\leq L_{\mathcal{D}}(h) + \frac{\varepsilon}{2} + \frac{\varepsilon}{2} \\ &= L_{\mathcal{D}}(h) + \varepsilon \end{aligned}$$

Figure 2.2: $\frac{\epsilon}{2}$-representative set

In other word, if a training set S is $\frac{\epsilon}{2}$-representative, then the hypothesis class H is agnostic PAC learnable.
But the question now is, when do we have uniform convergence?
It follows in Figure 2.3 the definition of uniform convergence:

A hypothesis class $\mathcal{H}$ has the *uniform convergence property* (w.r.t. a domain Z and a loss function ℓ) if there exists a function $m_{\mathcal{H}}^{UC} : (0,1)^2 \to \mathbb{N}$ such that for every $\varepsilon, \delta \in (0,1)$ and for every probability distribution $\mathcal{D}$ over Z, if S is a sample of $m \geq m_{\mathcal{H}}^{UC}(\varepsilon, \delta)$ i.i.d. examples drawn from $\mathcal{D}$, then with probability $\geq 1 - \delta$, S is ε-representative.

Figure 2.3: Uniform convergence

The preceding lemma implies that to ensure that the ERM rule is agnostic PAC learner, it suffices to show that with probability of at least $1 - \delta$ over the random choice of a training set, it will be an ϵ-representative training set
The function m_H^{UC} measures the minimal sample complexity to obtain the uniform convergence property, or rather, how many samples we need to ensure with a probability of at least $1 - \delta$ the training set S would be ϵ-representative. Uniform refers to having a fixed sample size that works for all members of H and over all possible probability distributions.

In Figure 2.4 a corollary that follows directly the ϵ-representative lemma and the definition of uniform convergence.

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

Figure 2.4: Proposition

But for which classes of hypothesis we have uniform convergence?
We can prove that the classes of finite sets hypothesis are agnostic PAC learnable under some restriction for the loss.
In Figure 2.5 the proposition.

Let $\mathcal{H}$ be a finite hypothesis class, let Z be a domain, and let $\ell : \mathcal{H} \times Z \to [0, 1]$ be a loss function. Then:

- $\mathcal{H}$ enjoys the uniform convergence property with sample complexity
$$m_{\mathcal{H}}^{UC}(\varepsilon, \delta) \leq \left\lceil \frac{\log(2|\mathcal{H}|/\delta)}{2\varepsilon^2} \right\rceil$$
- $\mathcal{H}$ is agnostically PAC learnable using the ERM algorithm with sample complexity
$$m_{\mathcal{H}}(\varepsilon, \delta) \leq m_{\mathcal{H}}^{UC}(\varepsilon/2, \delta) \leq \left\lceil \frac{2\log(2|\mathcal{H}|/\delta)}{\varepsilon^2} \right\rceil$$

Figure 2.5: Example

The idea for the proof is to:

- prove that uniform convergence holds for a finite hypothesis class
- use previous result on uniform convergence and PAC learnability

2.2 Bias Complexity

We have defined a generic learning task as: given a training set S and a loss function l, we have to find an hypothesis $\hat{h}$ such that the true error $L_D(\hat{h})$ is enough small. The hypothesis is usually find through an algorithm that produces as output this hypothesis $\hat{h} = A(S)$.
Our question is if there exist a learning algorithm A and a training set size m such that for every distribution D, if A receives as input m samples from D, there is an high chance of success.
The **No-Free Lunch Theorem** answers this question as shown in Figure 2.6:

> *Let A be any learning algorithm for the task of binary classification with respect to the 0-1 loss over a domain $\mathcal{X}$. Let m be any number smaller than $|\mathcal{X}|/2$, representing a training set size. Then, there exists a distribution $\mathcal{D}$ over $\mathcal{X} \times \{0,1\}$ such that:*
>
> - *there exists a function $f : \mathcal{X} \to \{0,1\}$ with $L_{\mathcal{D}}(f) = 0$*
> - *with probability of at least $1/7$ over the choice of $S \sim \mathcal{D}^m$ we have that $L_{\mathcal{D}}(A(S)) \geq 1/8$.*

Figure 2.6: No Free Lunch

Note that there are similar results for other learning tasks.
From this important proposition is born an important corollary: 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. The implication is that is very important use the priori knowledge that we have, in order to restrict the set of hypothesis.
However, there is a **trade off** because we want H large as well from the moment that larger is the class of hypothesis, higher is the probability that it may contain a function h with small error but, for no free lunch, H cannot be too large.
To answer this question we can decompose the error of an ERM predictor in two components:

$$L_D(h_S) = \epsilon_{app} + \epsilon_{est}$$

where

- $\epsilon_{app} = min_{h \in H} L_D(h)$ (approximation error)
- $\epsilon_{est} = L_D(h_S) - min_{h \in H} L_D(h)$ (estimation error)

The **approximation error** measures how much risk we have because we restrict our set H to a specific class or also how much **inductive bias** we have.

The inductive bias is the inability for a machine learning method to capture the true relationship between data.
It does not depend on the sample size and is determined by the hypothesis class chosen. Enlarging the hypothesis class can decrease the approximation error.
The **estimation error** is the difference between approximation error and the error achieved by ERM predictor.
It results because the training error is only an estimate of true risk.
The quality of this estimation depends on the training set size and on the complexity of the hypothesis class. This error increase with the enlarging of the set H.
The opposite of bias is **variance**, that is the variability in the model prediction, or rather how much the ML function has different prediction depending on the portion of the dataset. An high estimation error means an high variance.
Here is born the **complexity trade-off error**: we need to find a good tradeoff between this two quantity of error.
On one hand, choosing H to be a very rich class decreases the approximation error but at the same time might increase the estimation error, as a rich H might lead to overfitting.
On the other hand, choosing H to be a very small set reduces the estimation error but might increase the approximation error or, in other words, might lead to underfitting.
Underfitting happens when the model is too simple and with high bias and low variance and cannot capture the trend in the data, which results in a high total error.
Therefore, the model can't understand what is going on in the training set and ignores essential pieces of information that would help to make accurate predictions.
Overfitting is the extreme opposite of underfitting, which happens when the model is very complex and fits too closely to a limited set of data. In this case, the model memorizes the examples seen, and it can make highly accurate predictions in this training set.
However, when applied to a different data set, it will likely make an inferior prediction because it don't learn how to deal with different patterns; it just memorized examples. This will cause low bias and high variance.
In Figure 2.7 we can see a diagram in which is shown that a good tradeoff is the middle of this two vars.

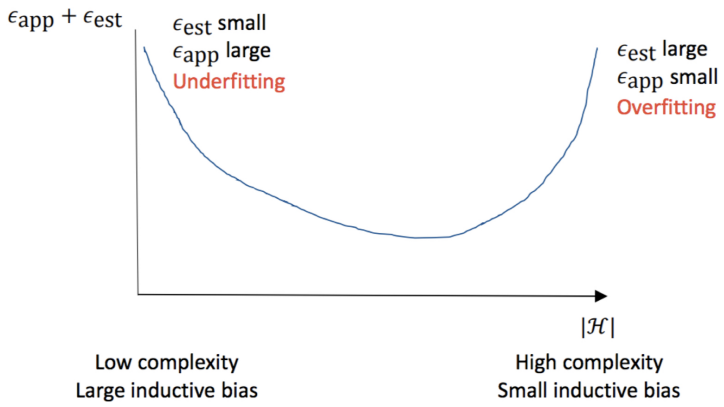


Figure 2.7: Diagram representing the tradeoff

How we can estimate the generalization error $L_D(h)$ for a function h, for example $h_s \in ERM_H$?
We can use a **test set**: new set of samples not used for picking h_S.

3

VC-dimension

The question is: which hypothesis classes H are PAC learnable? Up to now we know that if the cardinality of a class of hypothesis is less than ∞ means that H is PAC learnable.
But we don't know $|H| > \infty$ means that H is not PAC learnable.
We focus on binary classification with $0-1$ loss but similar results apply to other learning tasks.
We have to deal with the definition of restriction: let H be a class of function from X to $\{0,1\}$ and let $C = \{c_1, c_2, ..., c_m\} \subset X$ a vector of feature instances. The **restriction** H_C of H to C is

$$H_C = \{[h((c_1)...h(c_m)]\}$$

where H_C is a set of vector. We represent each function from C to $\{0,1\}$ as a vector in $\{0,1\}^{|C|}$.
It means that given a set C, H_C takes C as input and gives as output a vector of m binary components.
Given $C \subset X$, H shatters C if H_C contains all $2^{|C|}$ functions from C to $\{0,1\}$.
It means that in H_C there are all the possible combinations of 0 and 1 for vectors of size m.
The **VC-dimension** $VCdim(H)$ of an hypothesis class H is the maximal size of set C that can be shattered by H.
In other word, I look for a subset of instances that can generates all the possible 2^m combinations of $0, 1$ with the hypothesis that we have.

For example, with a VC-dimension of 2, we want that there is these combinations: 00,01,10,11.
Higher is the class of hypothesis and higher is the dataset, higher is the VC-dim.
If H can shatter sets of arbitrarily large size, we can say that $VCdim(H) = +\infty$ and, for counter, if $|H| < +\infty$, the $VCdim(H) = \log_2 |H|$.
The VC-dimension measures the complexity of H or rather how large a dataset that is perfectly classified using the functions in H can be.
We take as example to explain the following table, shown in Figure 3.1, that represents the dataset X with 9 samples and a class of hypothesis H of cardinality 8.

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
h_1	0	0	**1**	0	0	0	**1**	0	0
h_2	0	**1**	0	0	0	**1**	0	0	0
h_3	**1**	0	0	0	**1**	**1**	0	0	0
h_4	0	0	0	**1**	**1**	0	0	0	**1**
h_5	0	0	**1**	0	0	0	0	**1**	0
h_6	0	**1**	0	0	0	0	**1**	0	0
h_7	**1**	0	0	0	0	**1**	0	0	0
h_8	0	0	0	0	0	0	0	0	0

Figure 3.1: Example

What is the VC-dimension? We need to find the largest set C that can be shuttered by H.
Is the $VCdim(H) \geq 1$? Yes, if we select $C = \{x_3\}$, we have that $H_C = \{0, 1\}$ for $h_3(x_3)$ and for $h_5(x_3)$
Is the $VCdim(H) \geq 2$? Yes, if we select $C = \{x_5, x_6\}$, we have that $H_C = \{00, 01, 10, 11\}$ for $h_1(x_5, x_6) = 00$, $h_2(x_5, x_6) = 01$, $h_4(x_5, x_6) = 10$ and $h_3(x_5, x_6) = 11$.
Is the $VCdim(H) \geq 3$? No, because we need at least a column with ≥ 4 1's, but there is not.

To show that $VCdim(H) = d$ we need it show that is $\geq d$ and $\leq d$ that is translated to: there exists a set C of size d which is shattered by H and every set of size $d + 1$ is not shattered by H.

Follows the **Fundamental Theorems of Statistical Learning** in Figure 3.2:

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

- *$\mathcal{H}$ has the uniform convergence property with sample complexity*

$$C_1 \frac{d + \log(1/\delta)}{\varepsilon^2} \leq m_{\mathcal{H}}^{UC}(\varepsilon, \delta) \leq C_2 \frac{d + \log(1/\delta)}{\varepsilon^2}$$

- *$\mathcal{H}$ is agnostic PAC learnable with sample complexity*

$$C_1 \frac{d + \log(1/\delta)}{\varepsilon^2} \leq m_{\mathcal{H}}(\varepsilon, \delta) \leq C_2 \frac{d + \log(1/\delta)}{\varepsilon^2}$$

Figure 3.2: Theorem

or equivalently, in Figure 3.3:

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

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

where C is a universal constant.

Figure 3.3: Theorem

So, for concluding, if a class of hypothesis H has $VC(H) = +\infty$, then H is not PAC learnable.

4

Linear Models

Linear predictors/models is one of the most used families of hypothesis classes.
Consider $X = \mathbb{R}^d$ and we call **linear affine functions** the functions of this type;

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

where

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

where $\vec{w}$ is the vector of weight, b is the bias and each member of L_d is a function of this type. The cardinality of L_d is $+\infty$.
Notice that $\vec{x}$ are features and $\vec{w}$ and b is the parameters that we have to find such that the labeling is optimal.
So, after the ERM algorithm has finished is jobe, as output of this, in the case of linear models, we have a couple $(\vec{w}, b)$ that minimizes the error on S.
It's convenient use this notation:

$$L_d = \{\vec{x} \hookrightarrow \langle \vec{w}, \vec{x} \rangle + b \ : \ \vec{w} \in \mathbb{R}^d\}$$

where L_d is a set of functions where each function is parameterized by $\vec{w}$.
The different hypothesis classes H of linear predictors are given as composition of a function of L_d and a function $\phi : \mathbb{R} \hookrightarrow Y$ that takes a real value and gives a

prediction in this way:

$$h \in H : \phi \cdot L_d$$

It depends on the learning problem: for example for binary classification, $\phi(z) = sign(z)$.
There is also an equivalent notation that consider the bias in the vector. So, for given $\vec{x}$, b and $\vec{w}$ we can define:

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

  so then

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

4.1 Linear classification or halfspaces

Assume $X = \mathbb{R}^d$ and a label of this type $Y = \{-1, 1\}$ with a 0-1 loss.
The hypothesis class of **halfspaces** takes as instance $\vec{x}$ and returns the sign of the affine function described above, parametrized by $\vec{w}$ and b.
Geometrically speaking, considering an example $\mathbb{R}^2$, it subdivides the space in two parts like the example shown below in Figure 4.1. As we can see, the halfspace is defined by the affine function $\langle \vec{w}, \vec{x} \rangle = 0$

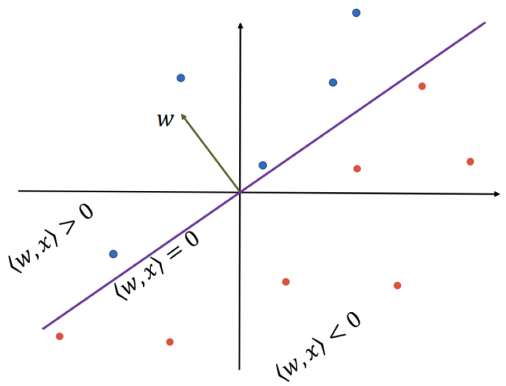


Figure 4.1: Halfspace

The hypothesis class is the one of **halfspaces** and it is:

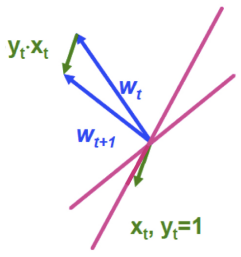
$$H = sign \cdot L_d = \{\vec{X} \hookrightarrow sign(h_{\vec{w},b}(\vec{x})) : h_{\vec{w},b} \in L_d\}$$

How do we find a good classifier, or rather, a small generalization error $L_D(h)$? We have to minimizes the training error (ERM).
The first algorithm that has achieved this goal is the **Perceptron Algorithm**.
Before the start we have to note one thing: if $y_i\langle\vec{w},\vec{x}_i\rangle > 0$ for all i means that all points are classified correctly by model w because if there is a mistake the value will be negative. So if there exists $\mathbf{w}$ such that $y_i\langle\vec{w},\vec{x}_i\rangle > 0$ the data are **linearly separable**.
The algorithm of Perceptron is shown below in Figure 4.2.

Input: training set $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)$
initialize $\mathbf{w}^{(1)} = (0, \ldots, 0)$;
for $t = 1, 2, \ldots$ **do**
  if $\exists i$ *s.t.* $y_i\langle\mathbf{w}^{(t)}, \mathbf{x}_i\rangle \leq 0$ **then** $\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} + y_i\mathbf{x}_i$;
  else return $\mathbf{w}^{(t)}$;

Figure 4.2: Perceptron Algorithm

If the algorithm see an error for the current model $\vec{w^{(t)}}$, it means that $\vec{x}_i$ is missclassified for some i so we have to update with the formula in the 4-th row.
One missclassified instance means that the hypothesis h of that halfspace is not correct to subdivide with the minimum error the data.
The first thing that we have to notice is that the update guides $\vec{w}$ to be "more correct" on the next $\vec{w}$, on the same point. If we develop the formula we obtain that, progressively, the value of the next prevision is higher, with the aim to make it positive, as we can see in Figure 4.3



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

$\Rightarrow$ update guides $\mathbf{w}$ to be "more correct" on $(\mathbf{x}_i, y_i)$.

Figure 4.3: Update of $\vec{w}$

If we concentrate on a graph, we can think that $\langle\vec{w},\vec{x}\rangle$ is an halfspace and the vector $\vec{w}$ is orthogonal to it. If there is a missclassified dat, we compute the new vector doing sum of the old vector and the product of $x_i y_i$ that create a new halfspace.

When we find an error we update our $\vec{w}$ until we find $\vec{w}$ such that no mistakes are made and all the data is correctly separate.
Termination depends on the reliability assumption or rather if the data is linearly separable.
If the data is linearly separable one can prove that the perceptron terminates.
In Figure 4.4 below there is a proposition of the number of iterations that the algorithm has to compute to find the best vector $\vec{w}$.

Assume that $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)$ is linearly separable, let:

- $B = \min\{||\mathbf{w}|| : y_i\langle \mathbf{w}, \mathbf{x}_i\rangle \geq 1 \quad \forall i, i = 1, \ldots, m, \}$, and
- $R = \max_i ||\mathbf{x}_i||$.

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

Figure 4.4: Proposition

It's important to say that perceptron is simple to implement, termination is guaranteed for separable data (even if it may require a number of iterations that is exponential in d).
However we can derive it in a different way.
Assume we want to solve a **binary classification** problem with linear models with a loss function $l(\vec{w}, (\vec{x}, y)) = max\{0, -y\langle \vec{w}, \vec{x}\rangle\}$. We want to find the hypothesis with the smallest training error, so $\vec{w}$ with smallest loss on training set S.
In Figure 4.5 is represented the graph of this loss function: if we are in the second quadrant, we have a wrong prediction.

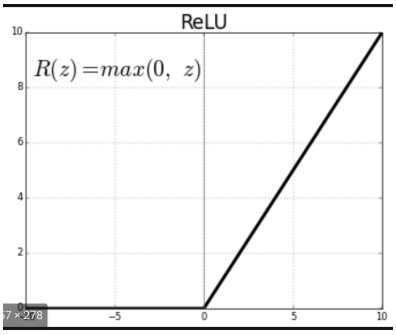


Figure 4.5: Loss function

There is an algorithm that is called **gradient descent** and it is a general approach

for minimizing a differentiable convex function $f(\vec{w})$ with f that is the cost function associated to a loss function.
Let $f : \mathbb{R}^d \hookrightarrow \mathbb{R}$ be a differentiable function, the gradient $\nabla f(\vec{w})$ of f at $\vec{w} = w_1, w_2, ..., w_d$ is:

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

with f that is the training error on S:

$$f(\vec{w}) = L_S(h_{\vec{w}}) \frac{1}{m} \sum_{i=1}^{m} l(\vec{w}, (\vec{x}_i, y))$$

The intuition is that the gradient points in the direction of the greatest rate of increase of f around w. So if we go in the opposite side respect to the gradient increasing, we go to a minimum. An idea of this implementation is shown in Figure 4.6

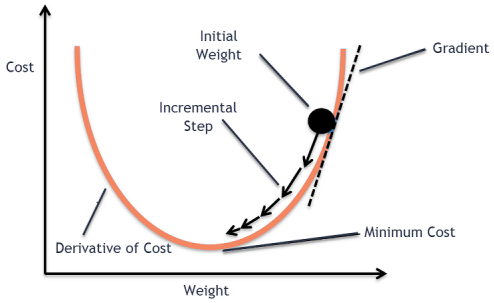


Figure 4.6: Gradient Descent

In the x axis there is w (assuming that it is in first dimension, so only with one value) and in the y axis there is the training error $L_S(h_{\vec{w}})$ that is our function f.
In Figure 4.7 shown below there is the algorithm for gradient descent.

GD algorithm:
$\mathbf{w}^{(0)} \leftarrow \mathbf{0}$;
for $t \leftarrow 0$ *to* $T - 1$ **do**
  $\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \eta \nabla f(\mathbf{w}^{(t)})$;
return $\bar{\mathbf{w}} = \frac{1}{T} \sum_{t=1}^{T} \mathbf{w}^{(t)}$;

Figure 4.7: GD algorithm

It's important to say that in our case the function f that has to be minimized is the loss function in order to minimize the training error.
The η parameter is called **learning rate** and give us the velocity which the gradient descents: an high learning rate means rapid descent but the probability to skip and go further the minimum but a small learning rate means a very slow descent and sure convergence.
As we can see the important step is at third row: the next weight vector is in the opposite direction of gradient, in order to find a minimum.
The output could also be w^T or the vector $\vec{w}$ with minimum cost.
One important problem of training is when stop the algorithm such that, at some point, the loss doesn't change too much.
In this case, we can change the point and restart from another point to verify if we was in a local minimum or not, or variate η to exit from a minimum local point and if return here it's a global minimum point.
Another type of approach is **stochastic gradient descent** that consists in taking random an instance, doing gradient descent and taking another instance.
Instead of using directly the gradient, calculating it and going in the opposite direction, we take a direction that in expectation is equal to the opposite of the gradient. In Figure 4.8 is shown the algorithm.

SGD algorithm:
$\mathbf{w}^{(0)} \leftarrow \mathbf{0}$;
for $t \leftarrow 0$ *to* $T-1$ **do**
  choose $\mathbf{v}_t$ at random from distribution such that $\mathbf{E}[\mathbf{v}_t|\mathbf{w}^{(t)}] \in \nabla f(\mathbf{w}^{(t)})$;
  /* $\mathbf{v}_t$ has *expected value* equal to the gradient of $f(\mathbf{w}^{(t)})$ */
  $\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \eta \mathbf{v}_t$;
return $\bar{\mathbf{w}} = \frac{1}{T}\sum_{t=1}^{T} \mathbf{w}^{(t)}$;

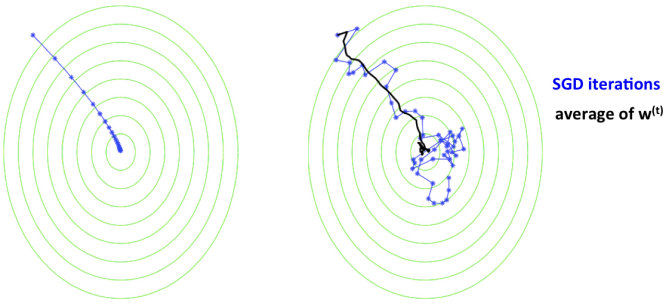


Figure 4.8: SGD algorithm

What is the meaning of row 3-4?
We need to pick a vector v_t that in expectation is equal to the gradient: how?

We pick a random point in the training set $z = (\vec{x}_i, y_i)$ and we define v_t to be the gradient of the function $l(\vec{w}, z)$ at iteration t.
So $v_t \in \nabla l(\vec{w}^t, (\vec{x}_i, y_i))$ is the vector to update $\vec{w}$, by linearity of the gradient we have:

$$\mathbb{E}[\vec{v}_t | \vec{w^{(t)}}] = \mathbb{E}_{Z \approx S}[\nabla l(\vec{w^{(t)}}, z)] = \nabla \mathbb{E}_{Z \approx S}[l(\vec{w^{(t)}}, z)] = \nabla L_S(\vec{w^{(t)}})$$

The useful thing of this method is that we have to use only one sample instead of all the training set to compute the gradient descent.
In the notebook is shown that, also for SGD, the function f that we have to minimize, is the error on training set $L_S(\vec{w})$. We take i uniformly at random from the training set and let $\vec{x}^1, y^1$ be the corresponding point in the training set and consider the vector $\nabla l(\vec{w}, (\vec{x}^1, y^1))$.
In the notebook is shown that, for the loss function given above, the gradient of the loss function is $-y_i \cdot \vec{x}_i$ so, the update of vector $\vec{w}$ become;

$$\vec{w}^{t+1} = \vec{w}^t + \eta y_i \vec{x}_i$$

It's quite common choose also a **mini-batch** of instances and do the same thing.

4.2 Linear Regression

Linear regression is an algorithm that provides a linear relationship between an independent variable and a dependent variable to predict the outcome of future events. It is a statistical method used in data science and machine learning for predictive analysis. The most important difference is that the output is $Y = \mathbb{R}$, or rather all the real numbers, so it's not discrete.
The hypothesis class is the same of linear classification:

$$H_{reg} = L_d = \{h_{\vec{w},b} \ : \ \vec{w} \in \mathbb{R}^d, \ b \in \mathbb{R}\}$$

where

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

where b is the bias, each member of L_d is a function of this type and the cardinality of L_d is infinity.

Note that in this case $h \in H_{reg} : \mathbb{R}^d \hookrightarrow \mathbb{R}$.
Commonly used loss function is **squared loss**:

$$l(h, (\vec{x}, y)) = (h(\vec{x}) - y)^2$$

and this implicate the empirical risk function (training error) that is **Mean Squared Error**

$$L_S(h) = \sum_{i=1}^{m} (h(\vec{x}_i) - y_i)^2$$

When we talk about simple linear regression we deal with only one independent variable so $d = 1$ and $h : \mathbb{R} \hookrightarrow \mathbb{R}$. and with $d = 1$ it can be represent in a bi-dimensional graph with the hypothesis $h_{w,b}(x) = b + x_1 w_1$
In Figure 4.9 is shown one example of simple linear regression.

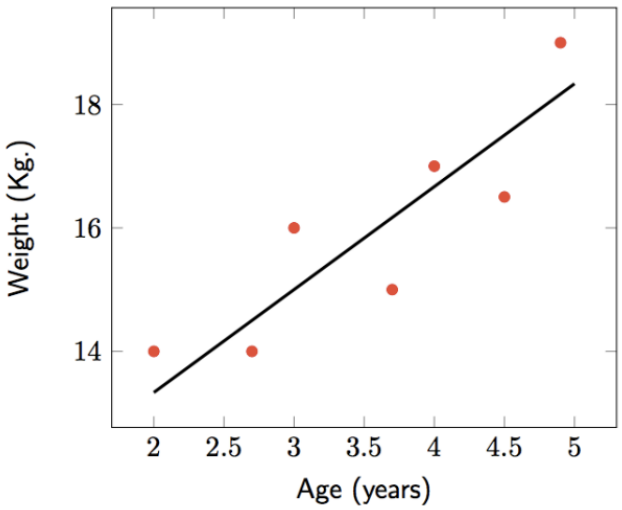


Figure 4.9: Example of regression

We want to minimize the training error and the algorithm to make it is **least squares algorithm**.
Best hypothesis will be:

$$argmin_{\vec{w}} L_s(h_{\vec{w}}) = argmin_{\vec{w}} \frac{1}{m} \sum_{i=1}^{m} (\langle \vec{w}, \vec{x}_i \rangle - y_i)^2$$

that is the vector of weight w that minimize the error or rather the vector of weight which line of regression has the minimum distance between taining set.
There is another formulation that is: w minimizing **Residual Sum of Squares**

that is:

$$argmin_{\vec{w}} \sum_{i=1}^{m} (\langle \vec{w}, \vec{x}_i \rangle - y_i)^2$$

We can see RSS in a matrix form with matrix X that is the matrix with for every column corresponds a feature and for every row corresponds an instance.
Matrix X:

$$X = \begin{bmatrix} ...\vec{x}_1... \\ \vdots \\ ...\vec{x}_n... \end{bmatrix}$$

Vector $\vec{y}$:

$$\vec{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$$

Using matrices we obtain, after some computational steps, RSS in this form:

$$(\vec{y} - X\vec{w})^T(\vec{y} - X\vec{w})$$

We want to find the vector $\vec{w}$ that minimizes RSS and to do that we compute **gradient** $\frac{\partial RSS(\vec{w})}{\partial w}$ and compare it to 0 in this way:

$$\frac{\partial RSS(\vec{w})}{\partial w} = -2X^T(\vec{y} - X\vec{w}) = 0$$

and this is equivalent to

$$X^T X\vec{w} = X^T\vec{y}$$

The most important result is if $X^T X$ is invertible: in this case we have a solution in a closed form to minimize the error of this type:

$$\vec{w} = (X^T X)^{-1} X^T \vec{y}$$

The most expensive operation is obviously the inversion of the matrix and there will be some problems is the matrix is not invertible.
In this case we consider:

$$A = X^T X$$

and we define A^+ that is the **generalized inverse** of A such that:

$$AA^+A = A$$

As result of this born an imporant **proposition**: if $A = X^TX$ is not invertible, then $\hat{w} = A^+X^T\vec{y}$ is a solution to $X^TX\vec{w} = X^T\vec{y}$.

4.2.1 Polynomial Models

Some learning tasks call for non linear predictors, such as polynomial predictors.
Suppose to have an as hyphotesis set a polynomial set of this type, of degree r:

$$\sum_{i=0}^{r} w_i x^i = w_0 + w_1 x + w_2 x^2 .. + w_r x^r$$

so we assume that $x \in \mathbb{R}$.
We can adapt our vectors in this way:

- $\vec{w} = (w_0, w_1, w_2, ..., w_r) \in \mathbb{R}^{d+1}$
- $\vec{x'} = (1, x, x^2, ..., x^r) \in \mathbb{R}^{r+1}$

In this way the hypothesis class of linear models for $\vec{x'}$ correspond to the hypothesis class of polynomials of degree r.
The same thing can be used also if is not a single instance but if it is a vector of d features.
Moreover, given a training set, it's so important **normalize** each feature $\vec{x_i}$ such that:

- the average of each feature across the training set is 0
- the standard deviation of each feature is 1

Data normalization is important for the stability of the computation and for the interpretability of linear models (weight is high means that a feature is important).
If we build a model normalizing the data, the same normalization funciton must bea pplied to the test data.

4.3 Logistic Regression

It's a regression task but the co-domain is not all the real numbers. Instead of all real numbers the prediction is in the range of $[0, 1]$ and often is used to classification problem.
For example, for a binary classification problem. the hypothesis h could be the probability that the label of $\vec{x}$ is 1. The hypothesis class is the following:

$$h \in H : \phi \cdot L_d$$

where L_d is the usual set of affine functions:

$$L_d = \{\vec{x} \hookrightarrow \langle \vec{w}, \vec{x} \rangle \ : \ \vec{w} \in \mathbb{R}^{d+1}\}$$

and $\phi_{sig} : \mathbb{R} \hookrightarrow [0, 1]$ is **sigmoid function** and the one used in this case is the **logistic regression**, shown in Figure 4.10 below.

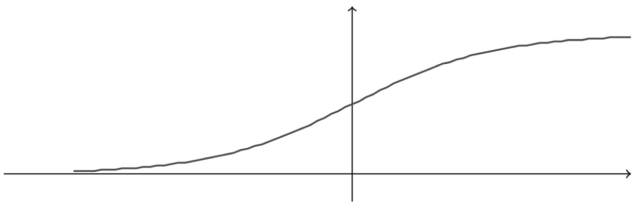


Figure 4.10: Sigmoid function

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

with z that represent the output of the linear model function.
Therefore

$$H_{sig} = \phi_{sig} \cdot L_d = \{\vec{x} \hookrightarrow \phi_{sig}(\langle \vec{w}, \vec{x} \rangle) \ : \ \vec{w} \in \mathbb{R}^{d+1}\}$$

so that, for $h_{\vec{w}}(\vec{x}) \in H_{sig}$ we have

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

If the output of this function is 1, there is an high confidence that the label of $\vec{x}$ is 1, if the output is 0, that the label of $\vec{x}$ is -1.
The main difference with classification with halfspaces is that if the inner product

of $\vec{w}, \vec{x}$ is 0, the output is $1/2$ that means uncertainty in predicted label.
As we can see before we want $h_{\vec{w}}(\vec{x})$ large if is y true is 1, also we want $1 - h_{\vec{w}}(\vec{x})$ large if y true is -1, or rather is $h_{\vec{w}}(\vec{x}) = 0$ means that there is an high confidence that the label of $\vec{x}$ is -1.
We have to note that:

$$\begin{aligned} 1 - h_{\mathbf{w}}(\mathbf{x}) &= 1 - \frac{1}{1 + e^{-\langle \mathbf{w}, \mathbf{x} \rangle}} \\ &= \frac{e^{-\langle \mathbf{w}, \mathbf{x} \rangle}}{1 + e^{-\langle \mathbf{w}, \mathbf{x} \rangle}} \\ &= \frac{1}{1 + e^{\langle \mathbf{w}, \mathbf{x} \rangle}} \end{aligned}$$

So, the reasonable loss function increase monotonically with:

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

that has sense from the moment that the increasing is inverted respect the desirable h_w for different cases.
If we invert the function we obtain

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

and so, the loss function for logistic regression is:

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

And for a given training set S the ERM problem for logistic regression is in Figure 4.11. Logistic loss function is a convex function so the ERM problem can

$$\arg \min_{\mathbf{w} \in \mathbb{R}^d} \frac{1}{m} \sum_{i=1}^{m} \log \left(1 + e^{-y_i \langle \mathbf{w}, \mathbf{x}_i \rangle}\right)$$

Figure 4.11: ERM problem for logistic regression

be solved efficiently.

We can show that ERM for logistic regression is equivalent to MLE (**Maximum**

Likelihood Estimation) that is a statistical approach for finding the parameters that maximize the joint probability of a given dataset assuming a specific parametric probability function.
The general approach for MLE is shown in Figure 4.12 below. Assuming $\vec{x_1}, .., \vec{x_m}$

1. given training set $S = ((\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m))$, assume each $(\mathbf{x}_i, y_i)$ is i.i.d. from some probability distribution of parameters θ
2. consider $\mathbb{P}[S|\theta]$ (likelihood of data given parameters)
3. *log likelihood*: $L(S;\theta) = \log(\mathbb{P}[S|\theta])$
4. *maximum likelihood estimator*: $\hat{\theta} = \arg\max_\theta L(S;\theta)$

Figure 4.12: General approach for MLE

fixed, with each instance labeled with an y_i. The probability that given $\vec{x_i}$, it has as label $y_i = 1$ is

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

where $\vec{w}$ is the parameters of the probability function.
Viceversa, The probability that given $\vec{x_i}$, it has as label $y_i = -1$ is

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

so, for each i, the probability that given $\vec{x_i}$, it has as label $y_i =$ is

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

From the moment that the likelihood on the given parameters is $\mathbb{P}[S|\Theta]$, for this training set is:

$$\prod_{i=1}^{m} \frac{1}{1 + e^{-y_i \langle \vec{w}, \vec{x_i} \rangle}}$$

or rather the probability that $\vec{x_i}$ has label y_i given parameters $\vec{w}$.
So after some computational steps and after have applied the log likelihood we obtain:

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

and the argmax of this function is **is equal to the ERM solution**

5 Model Selection

5.1 Validation

How we can choose the best algorithm or rather best parameters for the particular problem at hand? This task is often called **model selection**.
When we have a machine learning task there are different algorithms/classes and algorithms have different parameters.
To introduce model selection task we can consider a regression problem in one dimensional variable, so given an independent variable, try to predict the value of the dependent variable, in according to the unknown function $\hat{f}$.The training set is shown in Figure 5.1 below.

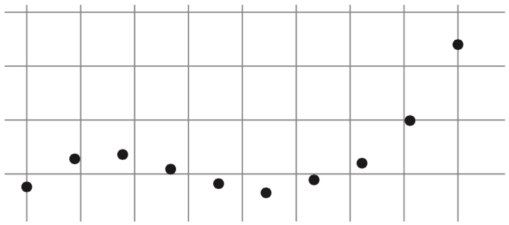


Figure 5.1: Example of regression for a training set

We know how to solve it by using the linear regression machinery that we've seen.
We choose the polynomial regression: suppose to have as hypothesis set a

polynomial set of this type, of degree d:

$$\sum_{i=0}^{d} w_i x^i = w_0 + w_1 x + w_2 x^2 .. + w_d x^d$$

so we assume that $x \in \mathbb{R}$.
We can adapt our vectors in this way:

- $\vec{w'} = (w_0, w_1, w_2, ..., w_d) \in \mathbb{R}^{d+1}$
- $\vec{x'} = (1, x, x^2, ..., x^d) \in \mathbb{R}^{d+1}$

How do we pick the degree d of the polynomial?
A small degree may not fit the data well (i.e it will have a large approximation error), whether an high degree may lead to overfitting (i.e it will have a large estimation error).
As we can see in Figure 5.2, when the degree expands, the error become lower.

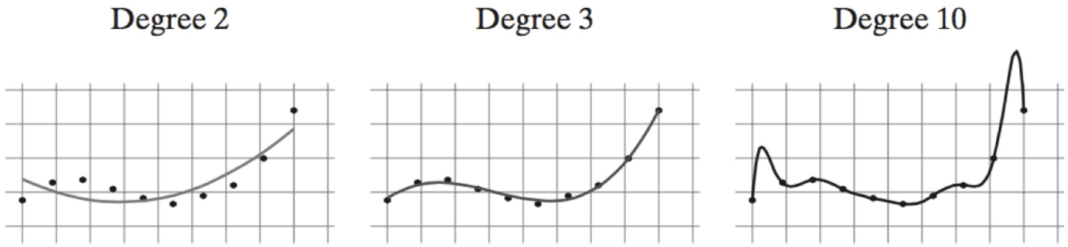


Figure 5.2: Different degrees

However, intuitively, the best choice is degree 3 such that for degree 10 we can occur in an high overfitting i.e. the next sample will be predicted wrong such that the model is well fitted only for the ten point of the training error.
The empirical risk is not enough: we have to consider **validation**.
Once you pick an hypothesis, we can use new data to estimate its true error.
Assume we've picked a predictor h, the best weight for the model after ERM rule on a classs H_d.
Let $V = (x_1, y_1), (x_2, y_2), ..., (x_{m_v}, y_{m_v})$ be a set of fresh samples from D and let

$$L_V(h) = \frac{1}{m_V} \sum_{i=1}^{m_V} l(h, (\vec{x_i}, y_i))$$

be the **validation error**.
Assuming that the loss function is in $(0, 1)$, after the application of Hoeffding

inequality we have the following proposition, shown in Figure 5.3

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

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

Figure 5.3: Proposition

The meaning of this proposition is that the validation error is close to the true error with a good probability under a certain value.
This is possible only because we use fresh sample, but, in practice we have only 1 dataset that we have to split into two parts: training set and validation set.
The validation can be used for **model selection**, i.e. to pick one hypothesis among hypothesis in several classes.
Assume to have $H = \bigcup_{i=1}^{r} H_i$ where, for example, H_i is the class of polynomials of degree i.
Given a training set S, let h_i be the hypothesis obtained by ERM rule from H_i using S.
We will have a set of hypothesis $\{h_1, h_2..., h_r\}$ and we have to choose the best.
We apply ERM rule using a set of fresh m_V sample called validation set and choose the one with the smallest error.

Assuming 0-1 loss function, we have the following bound, shown in Figure 5.4

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

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

Figure 5.4: Proposition

As we can see more hypothesis we have, higher is the bound.
We can use the **model-selection curve** that shows the training error and the validation error as function of the complexity of the model considered.
As we can see validation error is always higher than training error and at certain

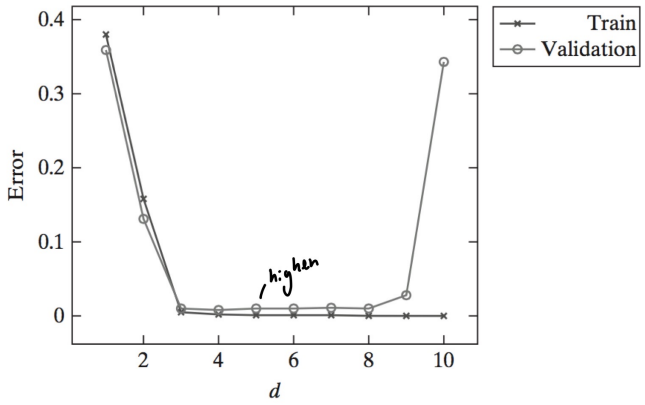


Figure 5.5: Model selection curve

point, training error decreases but validation error increases: here we deal with overfitting.
We have to remember that the empirical risk on the validation set is not an estimate of the true risk, in particular if r is large, because we have to choose among many models.
Now the question is: how can we estimate the true risk after model selection?
We can split the data into 3 parts such that:

- training set is used to find the best model h_i for the class H_i by using ERM
- validation set is used to pick one hypothesis h from $\{h_1, h_2, .., h_r\}$
- test set is used to estimate the true risk $L_D(h)$

5.2 Cross validation

However, can occurs that data are not enough and we cannot afford to use a fresh validation set. Fortunately we can appeal to the **cross validation**.
The k-fold cross validation consists into partition the training set into k folds of size m/k and for each fold we train the learner using the union of $k-1$ folds and then the error is estimated using remaining fold.
We do this in order to test the hypothesis with each fold.
At the end, the average of these errors is the estimation of the true error.
There is a special case that is when $k = m$ and it's called leave one out.
Often cross validation is used for model selection and, at the end, the final

hypothesis is obtained from training on the entire training set.
In Figure 5.6 there is a pseudo-code for the k-cross validation. In the second row

k-Fold Cross Validation for Model Selection

input:
training set $S = (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)$
set of parameter values Θ
learning algorithm A
integer k
partition S into $S_1, S_2, \ldots, S_k$
foreach $\theta \in \Theta$
for $i = 1 \ldots k$
$h_{i,\theta} = A(S \setminus S_i; \theta)$
$\text{error}(\theta) = \frac{1}{k} \sum_{i=1}^{k} L_{S_i}(h_{i,\theta})$
output
$\theta^\star = \text{argmin}_\theta \, [\text{error}(\theta)]$
$h_{\theta^\star} = A(S; \theta^\star)$

Figure 5.6: Pseudo-code for cross validation

we can see **set of parameters**: for example they may be the degree of polynomial, the regularization parameter λ.
However, despite everything looks good, could happen that results on test set are bad: we need to understand where the error comes from!
There are two cases: $L_S(h_s)$ is too large or $L_S(h_s)$ is too small.
Recalling the previous section in which we have talk about error decomposition we can do it also now: let $h^* \in argmin_{h \in H} L_D(h)$ such that the error $L_D(h^*)$ is the approximation error. We can use mathematical operation to change this equation in this form:

$$L_s(h_s) = (L_s(h_s) - L_s(h^*)) + (L_s(h^*) - L_D(h^*)) + L_D(h^*)$$

and from the moment that h_s is the hypothesis with the smallest training error, the first term is <0 and the second term is proximal to 0 so if $L_s(h_s)$ is large, also $L_D(h^*)$ is large and so the approximation error is large.
Instead, if $L_s(h_s)$ is small we don't know if $L_D(h^*)$ is small or big and we can use the **learning curves** to understand it.
Learning curves are plot of the training error and validation error when we run our algorithms on prefixes of the data increasing m. So the first time on the ten percent of the training set, the second time on the twenty percent and so on.

There are two possible cases, shown in Figure 5.7.

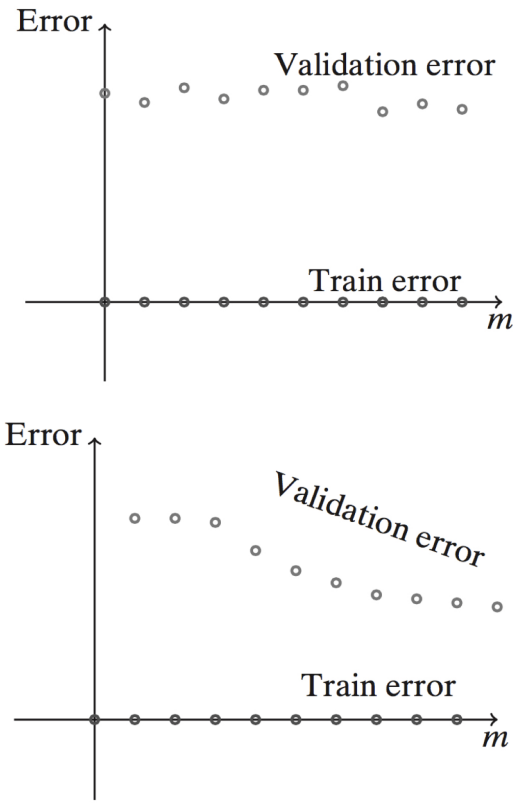


Figure 5.7: Two cases if $L_S(h_S)$ is small

In the first one there is no evidence that the approximation error of H is good, in the second case we may have a good approximation error but maybe we do not have enough data.
Summarizing, there are some potential steps to follow if learning fails:

- if we have parameters to tune, we can plot model-selection curve to make sure they are tuned appropriately
- if training error is excessively large consider to:
  - enlarge H
  - change H
  - change feature representation of the data
- if training error is small, use learning curves to understand whether problem is approximation error (or estimation error)
  - if approximation error seems small:

    * get more data
    * reduce complexity of H

- if approximation error seems large:
    * change H
    * change feature representation of the data

6

Regularization and Feature Selection

6.1 Regularization

We remember our task: we want to find an algorithm A such that the training made on m samples taken from our training set S give us a model, an hypothesis h that minimizes the training error and it's a good predictor (small true error).
We would like A to produce $A(S)$ such that $L_D(A(S))$ is small or at least close to the smallest generalization error $L_D(h^*)$ achievable by the best hypothesis h^* in term of generalization error.
All these considerations are done based on the loss function that, given an hypothesis h, compute the loss based on a point $z = (x, y)$
We've seen one **learning paradigms**, the ERM rule. For some learning problems it is possible to show that uniform convergence holds hence they are learnable using the ERM rule.
There is another learning paradigm that is **Regularized Loss Minimization** or RLM for short.
Assume h is defined by a vector $\vec{w} = (w_1, w_2, w_3, ..., w_d) \in \mathbb{R}^d$ for example for linear models.
The regularization function is $R : \mathbb{R}^d \hookrightarrow \mathbb{R}$ and the regularized loss minimization rule pick an hypothesis h:

$$argmin_{\vec{w}}(L_S(\vec{w}) + R(\vec{w}))$$

$R(w)$ is a measure of complexity of the hypothesis h defined by $\vec{w}$: the value is balanced between low empirical risk and too less complex hypothesis.

6.1.1 LASSO

The first regularization function that we use is:

$$R(\vec{w}) = \lambda||\vec{w}||_1$$

with $\lambda > 0$ and l_1 norm: $||\vec{w}||_1 = \sum_{i=1}^{d} |w_i|$
So retaking the **learning rule** we have:

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

The intuition is that $||\vec{w}||_1$ measures the complexity of hypothesis defined by w and λ regulates the tradeoff between empirical risk or overfitting and the complexity of the model we pick.
Why $||\vec{w}||_1$ measures the complexity of the model? Because if we have large value of w_i for such i, it implies that these features have high importance and the model risks to overfit because he bases its prediction paying more attention to these features.
Obviously, if a weight $w_i = 0$ means that this feature hasn't any importance in the model so we can have a simplier model.
Setting $\lambda = 0$ we don't care about the complexity of the system, we're interested only about a good result in term of training error.
If we set $\lambda = \infty$, the part that really matters is the complexity of the hyporhesis.
LASSO is the linear regression with squared loss + l_1 regularization.
This algorithm picks:

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

there is no closed form solution and being l_1 norm convex problem can be solved efficiently.

6.1.2 Ridge Regression

Another important function of regularization is **Tikhonov regularization function**:

$$R(\vec{w}) = \lambda ||\vec{w}||^2$$

with $\lambda > 0$ and l_2 norm: $||\vec{w}||^2 = \sum_{i=1}^{d} w_i^2$ so retaking the **learning rule**:

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

The intuition is that $||\vec{w}||_1$ measures the complexity of hypothesis defined by w and λ regulates the tradeoff between empirical risk or overfitting and the complexity of the model we pick So Ridge Regression is the union between linear regression with squared loss and Tikhonov regularization function.
We pick the classic solution for linear regression with squared loss

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

and we add the regularization function, finding

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

We have seen that the regression with squared loss can be expressed by using a closed form and we can do it also with ridge regression.
After some steps that we leave on the notebook, we reach a closed solution that is:

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

6.2 Feature Selection

In general, in machine learning one has to decide what to use as features of input for learning. How do we represent the instance space X?
When we have a real word object and we want to represent it by features, we use a **feature function**.
Every $\vec{x} \in X$ is a **feature vector** and X is how we encode the real word objects as an instance space and it's a prior knowledge of the problem.

Even we already have an instance space X, we might want to change it into a different representation and apply a hypothesis class on top of it.
We've already seen an example, with kernels and SVM, in which we map our set of instance X in another set.
We want to learn how to construct a good feature set that takes the name of **feature selection problem**.
We've a large pool of features and our goal is to select a small number of features that will be used.
Before now, we've always assumed that $X = \mathbb{R}^d$ so, each instance, is represented as a vector of d features. Our goal is to learn a predictor that only relies on $k <<< d$ features.
A predictor that uses only a small number of features is desirable because

- prevent overfitting: less predictors $\Rightarrow$ hypothesis of lower complexity
- predictions can be done faster

We could have tried all subsets of k out of d features and choose the subset which leads to the best performing predictor, but, an exhaustive search is usually computationally intractable.
Assume that we use the Empirical Risk Minimization (ERM) procedure. The problem of selecting k features that minimize the empirical risk can be written as:

$$min_{\vec{w}} L_S(\vec{w})$$

subject to $||\vec{w}||_0$ that is the number of value $w_i \neq 0$. Remember that if a weight $w_i = 0$, then the associated feature x_i is nor relevant.
We can solve it by:

- enumerating all subsets of the features composed by k elements. In other word we enumerate all the possible combinations of features composed by k elements
- build the best model (picked by ERM) on each subset
- keep the subset of minimum error

In Figure 6.1 the algorithm of this problem
The first row is the set with the d indices of the feature and p is a set of k indices. So at each iteration we have a k and for each subset of indices of dimension k we return the one with the smallest value on training error.
Solving this optimization problem is NP-hard.

Let:
- $\mathcal{I} = \{1, \ldots, d\}$;
- given $p = \{i_1, \ldots, i_k\} \subseteq \mathcal{I}$: $\mathcal{H}_p$ = hypotheses/models where only features $w_{i_1}, w_{i_2} \ldots, w_{i_k}$ are used

$P^{(k)} \leftarrow \{J \subseteq \mathcal{I} : |J| = k\}$;
foreach $p \in P^{(k)}$ **do**
  $h_p \leftarrow \arg\min_{h \in \mathcal{H}_p} L_S(h)$;
return $p_k \leftarrow \arg\min_{p \in P^{(k)}} L_S(h_p)$;

Complexity? Learn $\Theta\left(\binom{d}{k}\right) \in \Theta\left(d^k\right)$ models $\Rightarrow$ exponential algorithm!

Figure 6.1: Algorithm for subset selection

We can use an heuristic solution: **greedy algorithm**! The simplest instance of greedy selection is **forward greedy selection**: we start with an empty set of features and then we gradually add one feature to the set of selected feature. Supposing that our current set of selected feature is sol, we go over all $i \notin sol$ and we apply the learning algorithm on the set of features $sol \cup \{i\}$.
This process continue until we either select k features, where k is a predefined value.
In Figure 6.2 we can see the algorithm of this method.
There is also the **backward greedy selection** in which we start with a set of d

$sol \leftarrow \emptyset$;
while $|sol| < k$ **do**
  foreach $i \in \mathcal{I} \setminus sol$ **do**
    $p \leftarrow sol \cup \{i\}$;
    $h_p \leftarrow \arg\min_{h \in \mathcal{H}_p} L_S(h)$;
  $sol \leftarrow sol \cup \arg\min_{i \in \mathcal{I} \setminus sol} L_S(h_{sol \cup \{i\}})$;
return sol;

Complexity? Learns $\Theta(kd)$ models

Figure 6.2: Algorithm for forward selection

features and we remove one feature at each iteration until we reach a set of k features.
The complexity is $\Theta((d-k)d)$.

However, we have used only training set to select the best hypothesis and we may overfit. One solution can be use validation, so we test by using validation set.
The algprithms for subset selection and for forward selection using Validation data are in Figures 6.3 and 6.4.

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

Using training and validation:
for $\ell \leftarrow 0$ *to* k **do**
  $P^{(\ell)} \leftarrow \{J \subseteq \mathcal{I} : |J| = \ell\}$;
  foreach $p \in P^{(\ell)}$ **do**
    $h_p \leftarrow \arg\min_{h \in \mathcal{H}_p} L_S(h)$;
  $p_\ell \leftarrow \arg\min_{p \in P^{(\ell)}} L_V(h_p)$;
return $\arg\min_{p \in \{p_0, p_1, \ldots, p_k\}} L_V(h_p)$

Figure 6.3: Algorithm for subset selection with validation

Using training and validation:
$sol \leftarrow \emptyset$;
while $|sol| < k$ **do**
  foreach $i \in \mathcal{I} \setminus sol$ **do**
    $p \leftarrow sol \cup \{i\}$;
    $h_p \leftarrow \arg\min_{h \in \mathcal{H}_p} L_S(h)$;
  $sol \leftarrow sol \cup \arg\min_{i \in \mathcal{I} \setminus sol} L_V(h_{sol \cup \{i\}})$;
return sol;

Figure 6.4: Algorithm for forward selection with validation

7

SVM and Kernel

7.1 SVM

It's a very useful machine learning tool: it's a paradigm for learning linear predictors in high dimensional feature spaces that can result challenging in terms of sample complexity and computational complexity.
The SVM algorithm beats the sample complexity challenge by searching for a "large margin" separator. Roughly speaking an halfspace has a large margin if the sample data are more distant than the line that divides the data.
Consider a classification problem with two classes:

- instance set $X = \mathbb{R}^d$
- label set $Y = \{-1, 1\}$

with a training set S of m samples $z_i = (\vec{x}_i, y_i)$ and the hypothesis set H of halfspaces.
If data are linearly separable there is an halfspace that perfectly classifies the training set or rather there is a $(\vec{w}, b)$ such that:

$$y_i = sign(\langle \vec{w}, \vec{x}_i \rangle + b) \; \forall i \in \{1...m\}$$

and, in general, there are many halfspaces that classify correctly the data as we can see in Figure 7.1
The last one seems the best choice, since it can tolerate more noise. Informally we define, for a given halfspace, its **margin**, that is the distance to the closest

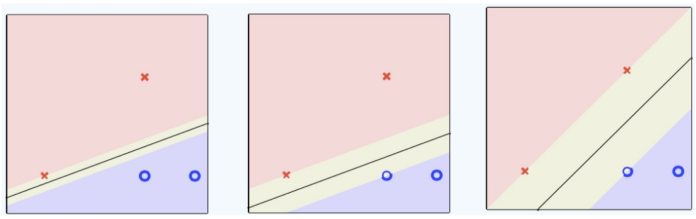


Figure 7.1: Different halfspaces with same result

sample in the training set S.
The intuition is that the best separating hyperplane is the one with the largest margin.
We have to formalize this thing: for first, we have to define the points of the hyperplane:

$$L = \{\vec{v} : \langle \vec{w}, \vec{v} \rangle + b = 0\}$$

and given an $\vec{x}$ the distance of $\vec{x}$ to L is:

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

and if $||\vec{w}|| = 1$ then $d(\vec{x}, L) = |\langle \vec{w}, \vec{x} \rangle + b|$.
So, in this case, the margin is:

$$min_{i=1,\dots,m} |\langle \vec{w}, \vec{x}_i \rangle + b|$$

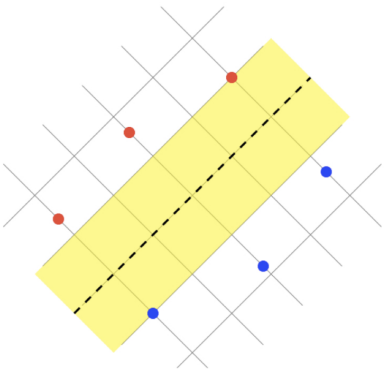


Figure 7.2: Support vectors

The closest examples are called **support vectors** and we can see in Figure 7.2 above that it's a vector because can be more points.

7.1.1 Hard-SVM

We now concentrate on **Hard-SVM** that is the learning rule in which we return an ERM hyperplane that separates the training set with the largest possible margin.

$$\arg \max_{(\mathbf{w},b):||\mathbf{w}||=1} \min_{i\in\{1,\dots,m\}} |\langle \mathbf{w}, \mathbf{x}_i\rangle + b|$$

$$\text{subject to } \forall i : y_i(\langle \mathbf{w}, \mathbf{x}_i\rangle + b) > 0$$

Figure 7.3: Hard-SVM constraint

In Figure 7.3 there is our computational problem: in other words we want to find we hyperplane with the biggest margin (argmax want to find the hyperplane $(\vec{w}, b)$ with the maximum margin (the margin is the min of the function inside verts)) subject to the fact that all the point are correctly classified (only these hyperplane).
So the equivalent formulation (due to the separability assumption) is the following, shown in Figure 7.4:

$$\arg \max_{(\mathbf{w},b):||\mathbf{w}||=1} \min_{i\in\{1,\dots,m\}} y_i(\langle \mathbf{w}, \mathbf{x}_i\rangle + b)$$

Figure 7.4: Equivalent formulation

and by solving it, we find a solution that is the same of the previous formulation. For Hard-SVM we can use the **Quadratic Programming Formulation** shown in Figure 7.5

- **input**: $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_m, y_m)$
- **solve**:

$$(\mathbf{w}_0, b_0) = \arg \min_{(\mathbf{w},b)} ||\mathbf{w}||^2$$

  $$\text{subject to } \forall i : y_i(\langle \mathbf{w}, \mathbf{x}_i\rangle + b) \geq 1$$

- **output**: $\hat{\mathbf{w}} = \frac{\mathbf{w}_0}{||\mathbf{w}_0||}, \hat{b} = \frac{b_0}{||\mathbf{w}_0||}$

Figure 7.5: Quadratic Programming Formulation

where $y_i(\langle \vec{w}, \vec{x}_i\rangle + b) \geq 1$ are linear inequalities.

By solving the second row we scan all the possible vectors $\vec{w}$ and we choose the one that classify correctly all the samples and has the lower norm.
The condition $y_i(\langle \vec{w}, \vec{x}_i \rangle + b) \geq 1$ essentially states that each data point must be on the correct side of the decision boundary and should be at least at a distance of 1 (or more) from the decision boundary, depending on the scale of the weight vector w.

If all the data are linearly separable, it means that it is possible to find a hyperplane that perfectly separates the positive and negative instances without any misclassifications. However, insisting on $y_i(\langle \vec{w}, \vec{x}_i \rangle + b) \geq 1$ rather than just $y_i(\langle \vec{w}, \vec{x}_i \rangle + b) \geq 0$ is a way to ensure that the margin is maximized.

By requiring a margin of at least 1, you are making the optimization problem more constrained, and this can help in finding a hyperplane that is not just any separating hyperplane but the one that maximizes the margin. This added constraint can lead to better generalization to unseen data because the decision boundary is not overly influenced by noise or small variations in the training data.
The output of this algorithm is a solution to the equivalent formulation seen before.
There is an equivalent formulation for homogeneous halfspaces (or rather the halfspaces where the bias term is put equal to zero: if we add one more feature to each instance of $\vec{x}_i$, thus increasing dimension to $d+1$, learning of nonhomogeneous halfspaces is equal to learning homogeneous halfspaces) the problem can be reduced as the following, shown in Figure 7.6

$$\mathbf{w}_0 = \min_{\mathbf{w}} ||\mathbf{w}||^2 \text{ subject to } \forall i : y_i \langle \mathbf{w}, \mathbf{x}_i \rangle \geq 1$$

Figure 7.6: Formulation for homogeneous halfspaces

$\vec{w}_0$ is the solution and the **support vectors** are the vectors at minimum distance from $\vec{w}_0$ and these are the only ones that matter for defining $\vec{w}_0$. Hence, if we think an example in $\mathbb{R}^2$ and suppose that x_1 is the point with the minimum distance, if we move another point farther respect to the hyperplane, the margin doesn't change.
From this intuition follows the proposition of Figure 7.7
With I that is the set of indices such that the distance from $\vec{w}_0$ to $\vec{x}_i$ is equal to 1 and these is the point at minimum distance from the hyperspace.

Let $\mathbf{w}_0$ be as above. Let $I = \{i : |\langle \mathbf{w}_0, \mathbf{x}_i \rangle| = 1\}$. Then there exist coefficients $\alpha_1, \ldots, \alpha_m$ such that

$$\mathbf{w}_0 = \sum_{i \in I} \alpha_i \mathbf{x}_i$$

Figure 7.7: Proposition

7.1.2 Soft-SVM

Hard-SVM works if data are linearly separable but if data is not linearly separable we use **Soft-SVM**.
We have seen that when we talk about Hard-SVM, i.e. SVM for linearly separable data, there is the hard constraints of $y_i(\langle \vec{w}, \vec{x}_i \rangle + b) \geq 1$ for all i.
The idea is to modify constraints of Hard-SVM to allow for some violations, but take account violations into the objective function in order to minimize it.
A natural relaxation is to allow the constraint to be violated for some of the examples in the training set and to do it, we introduce some non-negative slack variables $\psi_1, ..., \psi_m$ and replacing each constraint $y_i(\langle \vec{w}, \vec{x}_i \rangle + b) \geq 1$ with $y_i(\langle \vec{w}, \vec{x}_i \rangle + b) \geq 1 - \psi_i$ with the value of ψ_i that can be also an high value such that the right-hand side of the equation is negative, allowing an error. Each slack variable ψ_i measures how much each constraints is violated.
The objective of the Soft-SVM is to minimizes the norm $||w||$ corresponding to the margin and the average of ψ_i that corresponds to the violations of the constraints. The tradeoff among this two terms is controlled by a parameter $\lambda \in \mathbb{R}$. In Figure 7.8 is shown the algorithm for finding the best hyperplane that minimizes the violated constraint (i.e. minimizes the missclassified samples).

- **input**: $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)$, parameter $\lambda > 0$
- **solve**:

$$\min_{\mathbf{w}, b, \xi} \left(\lambda ||\mathbf{w}||^2 + \frac{1}{m} \sum_{i=1}^{m} \xi_i \right)$$

  subject to $\forall i : y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \geq 1 - \xi_i$ and $\xi_i \geq 0$
- **output**: $\mathbf{w}, b$

Figure 7.8: Algorithm

There is an equivalent formulation expressed as loss minimization problem

that uses the hinge loss:

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

where the graph of this function is shown in Figure 7.9 below.

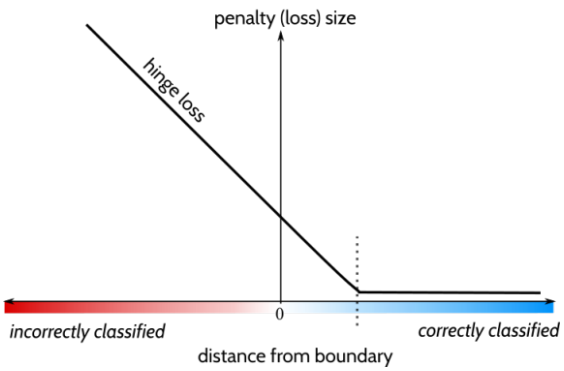


Figure 7.9: Hinge loss function

So given an hypothesis $(\vec{w}, b)$ and a training set S the averaged hing loss is denoted by $L_S^{hinge}(\vec{w}, b) = 1/m \sum_{i=1}^{m} l^{hinge}((\vec{w}, b)(\vec{x}_i, y_i))$ and the problem become, equivalent to the previous one, in the form shown in Figure 7.10

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

that is

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

Figure 7.10: Formulation

To solve this problem we can use standard solvers for optimization problem as the **Stochastic Gradient Descent**.
We want to solve this problem:

$$\vec{w} = min_w \, (\frac{\lambda}{2} ||\vec{w}||^2 + \sum_{i=1}^{m} max\{0, 1 - y(\langle \vec{w}, \vec{x}_i \rangle + b)\}$$

where the term $1/2$ is added in order to simplify some computations.
The SGD algorithm is shown in Figure 7.10

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

Figure 7.11: SGD for Soft-SVM

For concluding, in Figure 7.12 there is the biggest difference between Hard and Soft SVM: Hard SVM is only for linearly separable data, instead, Soft-SVM admits also some missclassified samples.

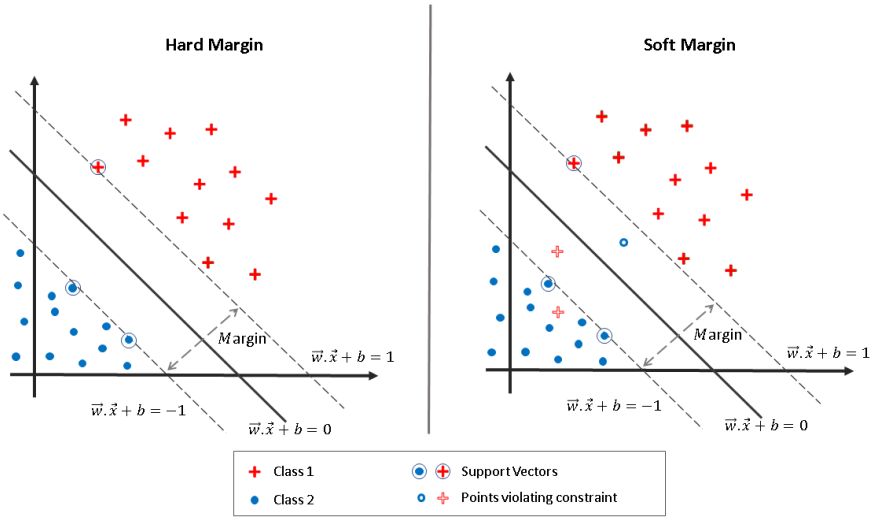


Figure 7.12: Difference between Hard and Soft SVM

7.1.3 SVM FOR LINEAR REGRESSION

SVMs can be also used for regression, in which the label is $Y = \mathbb{R}$

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

where

$$V_\epsilon(r) = 0 \; if \; |r| < \epsilon$$

$$V_\epsilon(r) = |r| - \epsilon \; otherwise$$

Here we have that the first term is the l_2 regularization function and the second term is the value of the loss function in which inside the brackets there is the difference between the observed value and the predicted value. If the difference is lower than ϵ, then the value of the loss function is 0.
One can prove that the solution has the form:

$$\vec{w} = sum_{i=1}^{m}(\alpha_i^* - \alpha_i)\vec{x}_i$$

where $\vec{x}_i$ is in the training set. The final model produced in output is

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

where $\alpha_i^*, \alpha_i \geq 0$ are the solution to a suitable QP.
The support vector are $\vec{x}_i$ such that $\alpha_i^* - \alpha_i \neq 0$

7.2 Kernels

Retaking the Hard-SVM problem, we can expand it in a different way in order to being useful for **kernels**. So, we want to solve:

$$\vec{w}_0 = min_w(\frac{1}{2}||\vec{w}|| \; subject \; to 1 \; \forall i: \; y_i\langle\vec{w}, \vec{x}_i\rangle \geq 1 \;)$$

this is equivalent to find $\vec{\alpha}$ that minimizes this **dual problem**

$$max_{\alpha\in\mathbb{R}^m:\alpha\geq 0}(\sum_{i=1}^{m}\alpha_i - 1/2\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j\langle\vec{x}_i, \vec{x}_j\rangle)$$

The solution is the vector $\vec{\alpha}$ which defines the support vectors $\{\vec{x}_i \; : \; \alpha_i \neq 0\}$.
$\vec{w}_0$ can be derived from $\vec{\alpha}$. I
t's useful because allow us to use kernels for SVM.
The dual problem requires only to compute the inner product and does not need to consider $\vec{x}_i$ itself.
SVM is a powerful algorithm, but still limited to linear models...and linear models cannot always be used (directly), as shown in Figure 7.12

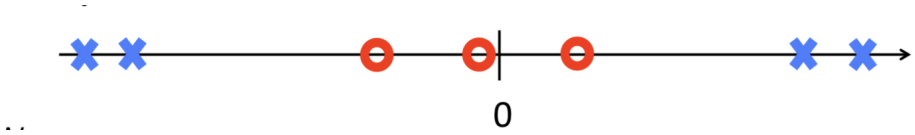


Figure 7.13: Linear model inefficient

An intuitive idea is to:

- apply a non linear transformation $\varphi()$ to each point in the training set S: $S' = ((\varphi(\vec{x}_1), y_1), ..., (\varphi(\vec{x_m}), y_m))$
- learn a predictor $\bar{h}$ in the trasformed space using S'
- make prediction for a new instances $\vec{x}$ as $\bar{h}(\varphi(\vec{x}))$

In Figure 7.14, this idea is applied and the set S of the previous example is transformed:

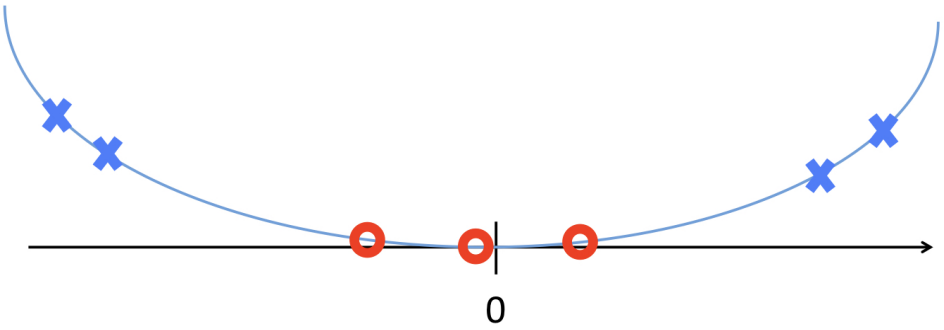


Figure 7.14: Transformation with kernels

Another image can help in order to visualize the function of a kernel: in Figure 7.15 the transformation is done in $\mathbb{R}^2$

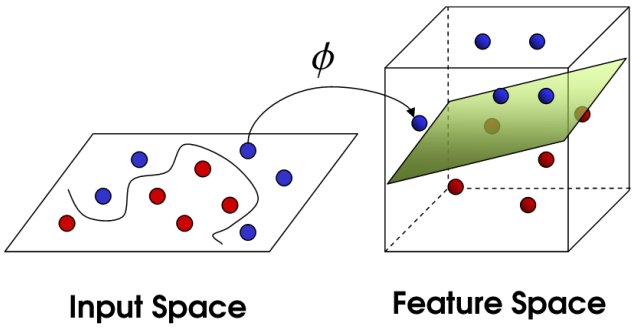


Figure 7.15: Transformation with kernels

What if we want to apply a non linear transformation before using SVM?
Given a training set S, we obtain S' by applying the transformation function φ. After that we learn a model h_{SVM} with SVM using S' and, given $\vec{x} \in X$, the predictor is $h_{SVM}(\varphi(\vec{x}))$
Let $\varphi()$ be the nonlinear transformation and we focus on the dual formulation: for hard-SVM we only need to be able to compute $\langle \varphi(\vec{x}_i, \varphi(\vec{x}_j)) \rangle$ for some i, j.
We can define a **kernel function** that is a function of the type:

$$K_\varphi(\vec{x}_i, \vec{x}_j) = \langle \varphi(\vec{x}_i, \varphi(\vec{x}_j)) \rangle$$

We can think K_φ as a function that specify similarity between two instances after the transformation. The similarity is given by the inner product.
Not always computing K_φ requires the computation of $\varphi\vec{x}$: sometimes we can compute it without computing $\varphi(\vec{x})$. In the notebook is shown an example of it.

When $K_\varphi(\vec{x}_i, \vec{x}_j)$ is efficiently computable, we don't need to explicitly compute $\varphi(\vec{x})$: this is the **kernel trick**.
The following kernels are the most commonly used:

- linear kernel: $\varphi\vec{x}$
- sigmoid: $K(\vec{x}_i, \vec{x}_j) = \tanh(\langle\vec{x}_i, \vec{x}_j\rangle + \zeta)$
- degree-Q polynomial kernel
- Gaussian-radial basis function (RBF) kernel

We focus on the **Degree-Q** polynomial kernel: for given constant $\psi > 0, \gamma > 0$ and for $Q \in \mathbb{N}$, the degree-Q polynomial kernel is

$$K(\vec{x}_i, \vec{x}_j) = (\psi + \gamma\langle\vec{x}_i, \vec{x}_j\rangle)^Q$$

Instead, the **Gaussian RBF** kernel is

$$K(\vec{x}_i, \vec{x}_j) = e^{-\gamma||\vec{x}_i - \vec{x}_j||^2}$$

The choice of the kernel is done based on the **Mercer's condition**, enunciated in Figure 7.16

$K(\mathbf{x}, \mathbf{x}')$ is a valid kernel function if and only if the kernel matrix

$$K = \begin{bmatrix} K(\mathbf{x}_1, \mathbf{x}_1) & K(\mathbf{x}_1, \mathbf{x}_2) \dots & K(\mathbf{x}_1, \mathbf{x}_m) \\ K(\mathbf{x}_2, \mathbf{x}_1) & K(\mathbf{x}_2, \mathbf{x}_2) \dots & K(\mathbf{x}_2, \mathbf{x}_m) \\ \vdots & \vdots & \vdots \\ K(\mathbf{x}_m, \mathbf{x}_1) & K(\mathbf{x}_m, \mathbf{x}_2) \dots & K(\mathbf{x}_m, \mathbf{x}_m) \end{bmatrix}$$

is always symmetric positive semi-definite for any given $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_m$.

Figure 7.16: Mercer's condition

8

Neural Networks

Simplified models of the brain, with a large number of basic computing units that are **neurons**, connected in a complex network.
A neuron is done by a function $\vec{x} \hookrightarrow \sigma(\langle \vec{v}, \vec{x} \rangle)$ where $\vec{v}$ is the vector of the weight and $\vec{x} \in \mathbb{R}^d$. σ is the activation function, defined on $\mathbb{R} \hookrightarrow \mathbb{R}$.
In Figure 8.1 an example of how it works.

Example: $\mathbb{R}^5$

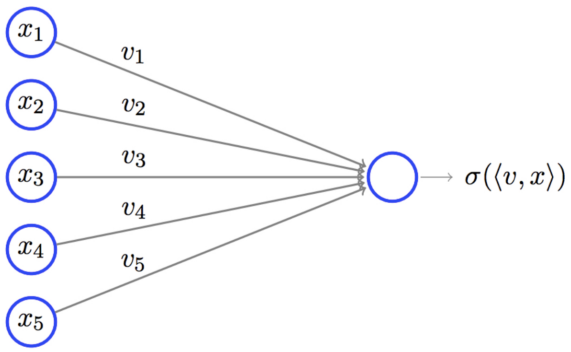


Figure 8.1: Activation function

We will consider σ to be among:

- sign function: $\sigma(a) = sign(a)$
- threshold function: $\sigma(a) = 1[a > 0]$
- sigmoid function: $\sigma(a) = \frac{1}{1+e^{-a}}$

Connecting many neurons together implicate **Neural Networks**, defined by a directed acyclic graph $G = (V, E)$ organized in layers, as we can see in Figure 8.2.

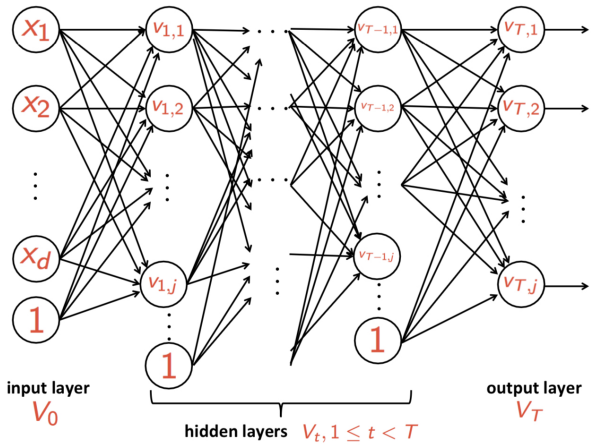


Figure 8.2: Neural Network

The input layer V_0 is our instance on which we have to make a prediction. The value of each node is computed, with the weight of each arcs, to obtain the argument of the activation function.
In the hidden layer, from V_1 to V_{T-1}, we compute nodes until we reach the output layer V_T, with the values of the prediction. The number of nodes of the output layer depends on the activation function: if we have a classification problem, the output layer will have only one node.
As we can see there is also the possible bias term (integer constant).

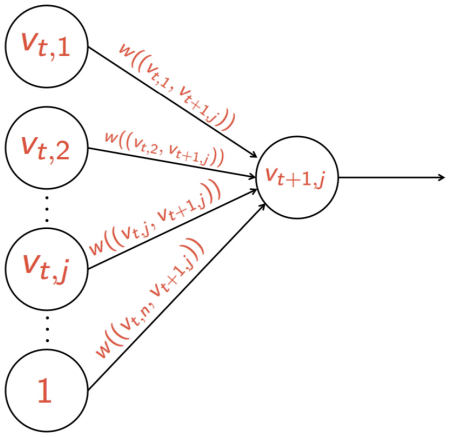


Figure 8.3: Point of view of one node

Focusing on the point of view of One Node, considering node $v_{t+1,j}$ i.e. node of layer $t+1$ at level j we can say that:

- $a_{t+1,j}(\vec{x})$ is the input of the node wen $\vec{x}$ is given as input of the NN
- $o_{t+1,j}(\vec{x})$ is the input of the node wen $\vec{x}$ is given as input of the NN

The input of one node is the inner product of all the edges that are connected with the node and the output of these nodes, so

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

and the output of this node is

$$o_{t+1,j}(\vec{x}) = \sigma(a_{t+1,j}(\vec{x}))$$

It's important deal with the concept of depth and width: depth T is the number of layers minus one and the width is the number of the nodes of the layer with maximum nodes.
In Figure 8.4 there is an example

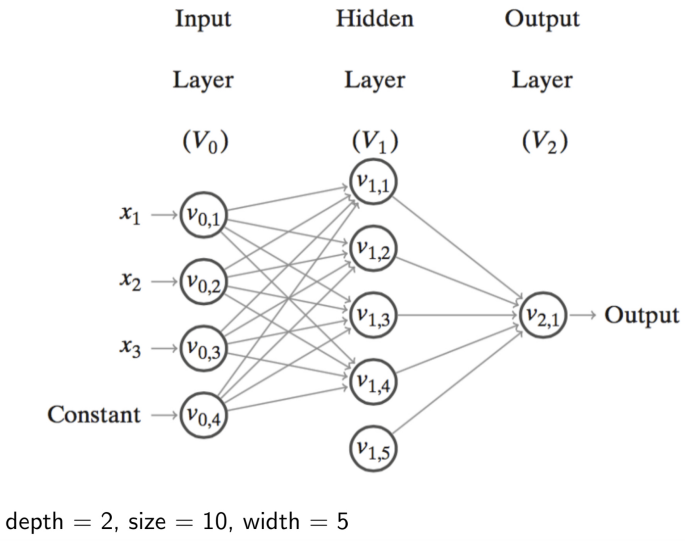


Figure 8.4: Example

What is the hypothesis set defined by a neural network?
The architecture of a NN is composed by: (V, E, σ) Once we specify the architecture and w, the function that gives a weight at each edge, we obtain this

function:

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

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

$$H_{V,E,\sigma} = \{h_{V,E,\sigma,w} \ : \ w \ is \ a \ map \ E \hookrightarrow \mathbb{R}\}$$

or, in other words, with V, E and σ fixed, all the NNs that can be find only changing w, are in the hypothesis class.
What type of functions can be implemented using a neural network?
We can take as example, the table of truth of a XOR function: it's a non linear function as we can see in Figure 8.5

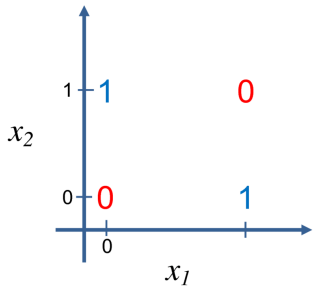


Figure 8.5: Xor diagram

We can represent this function by using a neural network, that is represented in Figure 8.6

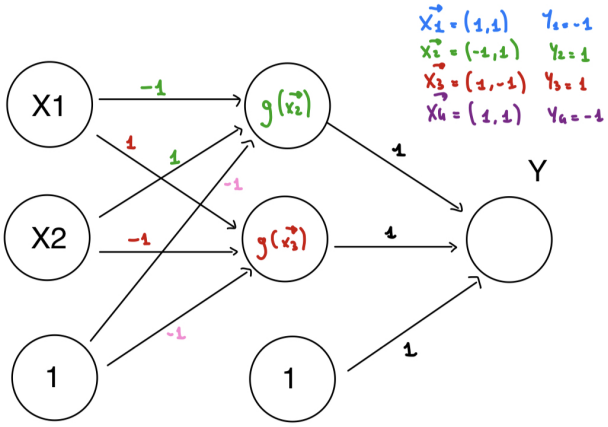


Figure 8.6: Xor NN

NNs are richer models than linear models: we can represent all linear models by using a neural network with the identity as activaction function.
Moreover, with NNs we can represent also non linear functions, for example boolean functions.
We take an arbitrary function $f : \{-1,1\}^d \hookrightarrow \{-1,1\}$ with the goal to build a NN that "computes" f: if the input is $\vec{x}$ then the predictor is $f(\vec{x})$.
We remember that we have a fixed an architecture $H_{V,E,sign}$.
How we can fix the weights in order to implement f?
In $\{-1,1\}^d$ there are 2^d+1 possible vectors. We consider only instances $\vec{x_1}, ..., \vec{x_k}$ such that the prediction is 1: for each such $\vec{x_i}$ there is a neuron in the only hidden layer that "corresponds" to $\vec{x_i}$.
To do this, for each vector $\vec{x_k}$ of input with $f(\vec{x_k}) = 1$, we choose as vector of weight $\vec{x_k}$, for the arcs that goes to the "correspondent" node $\vec{x_k}$ in the hidden layer (supposing $\vec{x_k} = [-1,1,1]$, we define a node in the hidden layer that is connected with x_1 with an arc of weight -1, with x_2 with an arc of weight 2 and with x_3 with an arc of weight 1) and we compute the only hidden layer in this way:

$$g_i(\vec{x'}) = sign(\langle \vec{x}, \vec{x'} \rangle - d + 1)$$

where $\vec{x}$ is the vector of weight, $\vec{x'}$ is the input to the neuron and also to the NN. If $\vec{x'} = \vec{x}$, then the output of $g_i(\vec{x'})$ is 1. It means that in the hidden layer, for a given $\vec{x'}$ as input, there is only one neuron that implement the function $g_i(\vec{x'}) = 1$.
The output node is implemented by using putting by 1 all the weight and then

$$f(\vec{x'}) = \sum_{i=1}^{k} g_i(\vec{x'}) + k - 1$$

In this way we will have in the only output layer implemented the function f.
So, resuming, if we have an exercise and we have to put the weight, we take all the vector $\vec{x'_k}$ for which the output of the function is 1, we create one node in the hidden layer for all these possibilities in which the weight that connect x_i of input to the node, is the value of x'_i.In this way, if in input we have $\vec{x_k}$, then that node will be the onliest with a positive value.
After that, we connect each node of the hidden layer with the output layer with an arc of weight 1 and the bias term with an arc of weight $-k+1$ and then we have our graph that implements the function f.
So the overall graph is: For example, retaking the XOR function, we have that

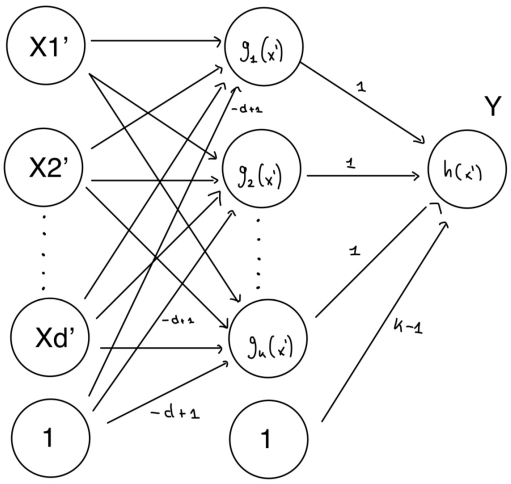


Figure 8.7: General NN for a function

only two vectors have as label 1. They correspond to $[-1, 1]$ and $[1, -1]$. We create two nodes in the hidden layer and we connect x_1 and x_2, the only two features, with the respected values for each node.
After that, we can see that if we put as input $[1, 1]$ the output will be -1, instead if we put as input $[1, -1]$, the output will be 1.
From this important intuition, are born two important proposition:
Proposition: For every d, there exists a graph (V, E) of depth 2 such that $H_{V,E,sign}$ contains all function from $\{-1, 1\}^d \hookrightarrow \{-1, 1\}$.
Unfortunately, the graph is very big

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

Figure 8.8: Proposition

with similar result for sigmoid as activation function.
There are similar results also for regression tasks:

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

Figure 8.9: Proposition

So NNs are **universal approximators**. But again: How much data is needed to

> 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 $\mathcal{H}_{V,E,\sigma}$, with σ = sigmoid, can approximate, with precision ε, every 1-Lipschitz function $f : [-1,1]^d \to [-1,1]$. Then $s(d)$ is exponential in d.

Figure 8.10: Proposition

learn with NNs?
The VC-dimension of $H_{V,E,sign}$ is $O(|E| \log |E|)$ and, with a different σ, the VC-dimension of $H_{V,E,\sigma}$ is :

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

that means tha large NNs requires a lot of data.
Assume that we have a lot of data, can we find the best hypothesis?
What is the runtime for learning by using NNs?
Informally, applying the ERM rule with respect to $H_{V,E,sign}$ is **computationally difficult**, even for small NN. In Figure 7.xx there is an important proposition about this concept:

> 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 $\mathcal{H}_{V,E,\text{sign}}$.

Figure 8.11: Proposition

All others variations for the previous proposition (not using ERM, change the activation function) result **computationally infeasible**!
We can use **heuristic** for training NNs: SGD and its improved version are used with good results in practice.

We can represent our NN by using a matrix notation: considering layer t, $0 < t < T$:

- let $d^{(t)} + 1$ the number of nodes which have:
  - constant node 1
  - values of nodes for variables: $v_{t,1}, ..., v_{t,d^{(t)}}$
- arc from $v_{t-1,i}$ to $v_{t,j}$ has weight $w_{ij}^{(t)}$

So we can define:

$$\vec{v^{(t)}} = (1, v_{t,1}, ..., v_{t,d^{(t)}})^T$$

$$\vec{w_j^{(t)}} = (w_{0j}^{(t)}, w_{1j}^{(t)}, ..., w_{d^{(t-1)}j}^{(t)})^T$$

and then

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

So, the value of the nodes at layer t is:

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

and let

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

and vector $\vec{a^{(t)}}$:

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

and so

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

then, for final, we can write the vector $v(t)$ as:

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

We can finally represent the vector of weight $\vec{w^{(t)}}$ by using a matrix that contains all the weights of edges that go from layer $t-1$ to t:

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

Then

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

So, now that we have the matrix notation, in Figure 8.12 is shown the algorithm, the **forward propagation algorithm**, that given in input a vector $\vec{x}$, for which we want a prediction, and an NN, with a well defined matrix of weight $w^{(t)}\ \forall\ t$, that is our hypothesis h, the algorithm gives us as output the prediction.

Input: $\mathbf{x} = (x_1, \dots, x_d)^T$; NN with 1 output node
Output: prediction y of NN;

$\mathbf{v}^{(0)} \leftarrow (1, x_1, \dots, x_d)^T$;
for $t \leftarrow 1$ **to** T **do**
  $\mathbf{a}^{(t)} \leftarrow \left(\mathbf{w}^{(t)}\right)^T \mathbf{v}^{(t-1)}$;
  $\mathbf{v}^{(t)} \leftarrow \left(1, \sigma\left(\mathbf{a}^{(t)}\right)^T\right)^T$;
$y \leftarrow \sigma\left(\mathbf{a}^{(T)}\right)$;
return y;

Figure 8.12: Algorithm

Unfortunately, we haven't an oracle that gives us the best NN, so the best matrix of weight $w^{(t)}$ for the arcs. How do we compute the weights $w_{ij}^{(t)}$?
We can use ERM that, given a training set $S = \{(\vec{x_1}, y_1), \dots, (\vec{x_m}, y_m)$, pick $w_{ij}^{(t)}$, or rather our h, minimizing the training error

$$L_S(h) = 1/m * \sum_{i=1}^{m} l(h, z_i)$$

It's not easy and in the notebook are shown all the details to reach the **back-propagation algorithm**, shown in Figure 8.13, based on SGD, to train a NN with

lower error.

Input: training data $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)$, NN (no weights $w_{ij}^{(t)}$)
Output: NN with weights $w_{ij}^{(t)}$
initialize $w_{ij}^{(t)}$ for all i, j, t;
for $s \leftarrow 0, 1, 2, \ldots$ **do** `/* until convergence */`
  pick $(\mathbf{x}_k, y_k)$ at random from training data;
  `/* forward propagation */`
  compute $v_{t,j}$ for all j, t from $(\mathbf{x}_k, y_k)$;
  `/* backward propagation */`
  compute $\delta_j^{(t)}$ for all j, t from $(\mathbf{x}_k, y_k)$;
  $w_{ij}^{(t)} \leftarrow w_{ij}^{(t)} - \eta v_{t-1,i} \delta_j^{(t)}$ for all i, j, t; `/* update weights */`
  if *converged* **then return** $w_{ij}^{(t)}$ for all i, j, t;

Figure 8.13: Proposition

In backpropagation algorithm there are some useful things that we need to know:

- preprocessing: all inputs are normalized and centered
- initialization of $w_{ij}^{(t)}$ is done by giving random values around 0
- we stop when there is a small training error or a small marginal improvement in error or we reach the upper bound on number of iterations

8.0.1 Regularized NN

Instead of training a NN by minimizing $L_S(h)$, we can find h that minimizes

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

where λ is the regularization parameter. How do we find h? We can use SGD or improved algorithms.
This is called squared weight decay regularizer but other regularizations are possible.

9

Clustering

It is the first example of **unsupervised learning** that we see. In unsupervised learning, the training dataset is $(\vec{x_1}, \vec{x_2}, \vec{x_3}, ..., \vec{x_m})$, so there are not target values. We're interested to organize this data in some meaningful way and we will use the most common unsupervised learning approaches that is **clustering**.
We're going to focus on the most commonly used techniques:

- k-means
- linkage-based clustering

More formally, **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 group as we can see in Figure 9.1

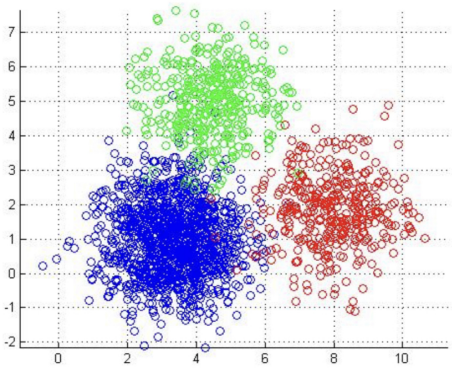


Figure 9.1: Clustering

Let's formulate the clustering problem more formally:

- **Input**: set of elements X and distance function $d : X \; x \; X \hookrightarrow \mathbb{R}_+$ that is a function that
  - is symmetric $d(\vec{x}, \vec{x'}) = d(\vec{x'}, \vec{x})$ for all $\vec{x}, \vec{x'} \in X$
  - $d(\vec{x}, \vec{x}) = 0$ for all $\vec{x} \in X$
  - d satisfies the triangle inequality $d(\vec{x}, \vec{x'}) \leq d(\vec{x}, \vec{z}) + d(\vec{z}, \vec{x'}) =$

- **Output**: a partition of X into clusters that is $C = (C_1, C_2, ..., C_k)$ with
  - $\bigcup_{i=1}^{k} C_i = X$
  - for all $i \neq j : C_i \cap C_j = \emptyset$

Sometimes the input also includes the number k of clusters to produce in output.
Moreover, sometimes, instead of having a distance function we have a **similarity** function $s : X \; x \; X \hookrightarrow \mathbb{R}_+$ that is a function that:

- is symmetric $s(\vec{x}, \vec{x'}) = s(\vec{x'}, \vec{x})$ for all $\vec{x}, \vec{x'} \in X$
- $s(\vec{x}, \vec{x}) = 1$ for all $\vec{x} \in X$

The choice among these two possibilities depends on the type of data and from the fact that different distances may be used for the same dataset.
There are two classes of algorithm for Clustering that are: **Cost minimization algorithms** and **Linkage-based algorithm**

9.1 Cost Minimization Clustering

A common approach in clustering is:

- define a cost function over possible partitions of the objects
- find the partition (=clustering) of minimal cost

There are also some assumptions:

- data points $\vec{x} \in X$ come from a larger space X' that is $X \subseteq X'$
- distance function $d(\vec{x}, \vec{x'})$

For simplicity we assume that $X' \in \mathbb{R}^d$ and $d(\vec{x}, \vec{x'}) = ||\vec{x} - \vec{x'}||$
The k-Means clustering has the following goal: taking as input some data points $\vec{x_1}, ..., \vec{x_m}$ and k that is the number of clusters, we want to find a partition $C = (C_1, ..., C_k)$ for $\vec{x_1}, ..., \vec{x_m}$ and **centers** $\mu_1, ..., \mu_k$ with $\mu_i \in X'$ center for C_i that minimizes the k-**means objective** (cost):

$$\sum_{i=1}^{k} \sum_{\vec{x} \in C_i} d(\vec{x}, \mu_i)^2$$

So each cluster C_i has a center μ_i and we want to find the partitions and the centers that minimizes the distance to the center from the points of each cluster. Each cluster has its own center, and the data points within a cluster are considered closer to the center of that cluster than to the centers of other clusters.
If the clusters are fixed, we have to find the centers that minimize the cost of the function.

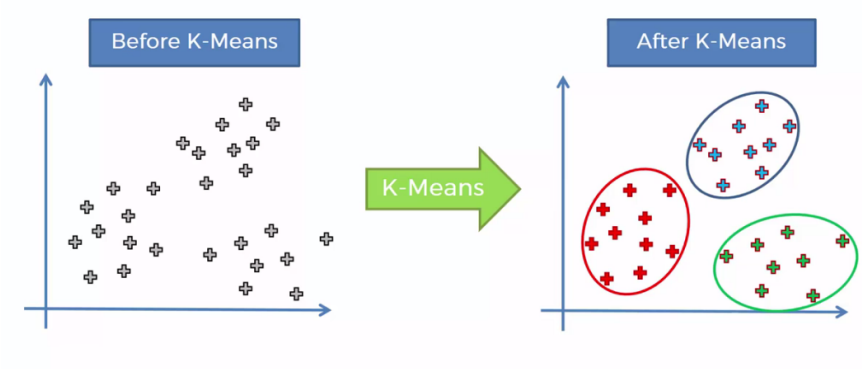


Figure 9.2: K-means method

There are also different types of objective function:

$$min_{\mu_1,...,\mu_k \in X} \sum_{i=1}^{k} \sum_{\vec{x} \in C_i} d(\vec{x}, \mu_i)^2$$

so the centers are point of the input set. It's called k-**medoids** objective.

$$min_{\mu_1,...,\mu_k \in X} \sum_{i=1}^{k} \sum_{\vec{x} \in C_i} d(\vec{x}, \mu_i)$$

so the centers are point of the input set. It's called k-**median** objective.

Back to k-means clustering: is it more difficult finding the clusters or finding the centers?

Proposition: Given a cluster C_i, the center μ_i that minimizes the function $\sum_{\vec{x} \in C_i} d(\vec{x}, \mu_i)^2$ is

$$\mu_i = \frac{1}{C_i} \sum_{\vec{x} \in C_i} \vec{x}$$

A naive brutal force algorithm to solve this problem is to try all possible partitions of the m points into k clusters, evaluate each partition and find the best one.

The complexity depends on the number of partitions of m points into k cluster: as trivial upper bound we have k^m, the exact bound is the Stirling number of the second kind.

The fact is that finding the optimal solution for k-means clustering is computationally difficult (NP-hard).

A good practical heuristic to solve k-means is **Lloyd's algorithm**, shown in Figure 9.3

Input: data points $\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m\}$; $k \in \mathbb{N}^+$
Output: clustering $C = (C_1, C_2, \ldots, C_k)$ of $\mathcal{X}$; centers $\mu_1, \mu_2, \ldots, \mu_k$ with μ_i center for C_i, $1 \leq i \leq k$;

randomly choose $\mu_1^{(0)}, \ldots, \mu_k^{(0)}$;
for $t \leftarrow 0, 1, 2, \ldots$ **do** `/* until convergence */`
  for $i = 1, \ldots, k$: $C_i \leftarrow \{\mathbf{x} \in \mathcal{X} : i = \arg\min_j d(\mathbf{x}, \mu_j^{(t)})\}$;
  for $i = 1, \ldots, k$: $\mu_i^{(t+1)} \leftarrow \frac{1}{|C_i|} \sum_{\mathbf{x} \in C_i} \mathbf{x}$;
  if *convergence reached* **then**
    return $C = (C_1, \ldots, C_k)$ and $\mu_1^{(t+1)}, \mu_2^{(t+1)}, \ldots, \mu_k^{(t+1)}$

Figure 9.3: Lloyd's algorithm

In order to ensure the **convergence of Lloyd's algorithm**, the most commonly used criteria are:

- the k-means objective for the cluster at iteration t is not lower than the k-means objective for the cluster at iteration $t-1$
- $\sum_{i=1}^{k} d(\mu_i^{(t+1)}, \mu_i^{(t)}) \leq \epsilon$
- $max_{1 \leq i \leq k} d(\mu_i^{(t+1)}, \mu_i^{(t)}) \leq \epsilon$

If the first convergence criteria above is used, then Lloyd's algorithm always terminates.
The complexity of Lloyd's algorithm depends on:

- assignment of points $\vec{x} \in X$ to clusters C_i: time $O(kmd)$ with d time to compute the distance
- computations of centers μ_i: time $O(md)$

It convergence after t iteration: $O(tkmd)$. How many iterations are required? The rough studies has an upper bound of $\approx k^m$, a more sophisticated one $O(m^{kd})$. A lower bound is $2^{\Omega(\sqrt{m})}$. So, in practice, much less than m iterations are required. The convergence and the quality of the clustering depends on the initialization of the centers: the $k - means + +$ is a simple but effective center initialization strategy.
The algorithm for this strategy is shown in Figure 9.4

$\mu_1 \leftarrow$ random point from $\mathcal{X}$ chosen uniformly at random;
$F \leftarrow \{\mu_1\}$;
for $i \leftarrow 2$ **to** k **do**
  $\mu_i \leftarrow$ random point from $\mathcal{X} \setminus F$, choosing point $\mathbf{x}$ with probability $\frac{(d(\mathbf{x},\mathbf{F}))^2}{\sum_{\mathbf{x}' \in \mathcal{X} \setminus F} (d(\mathbf{x}',\mathbf{F}))^2}$;
  $F \leftarrow F \cup \{\mu_i\}$;
return F;

Figure 9.4: Algorithm for $k + +$ means

9.1.1 Linkage Based Clustering

General class of algorithms that follow the general scheme:

- 1) start from trivial clustering: each data point is a single point cluster
- 2) **"until" termination condition**: repeatedly merge the "closest" clusters of the previous clustering

We need to specify two "parameters": how to define distance between clusters and termination condition.
There are different types of distance between two clusters A and B defined as $D(A, B)$, resulting into different linkage methods:

- **single linkage**: $D(A, B) = min\{d(\vec{x}, \vec{x'}) : \vec{x} \in A, \vec{x'} \in B)\}$
- **average linkage**: $D(A, B) = min\{d(\vec{x}, \vec{x'}) : \vec{x} \in A, \vec{x'} \in B)\}$
- **max linkage**: $D(A, B) = min\{d(\vec{x}, \vec{x'}) : \vec{x} \in A, \vec{x'} \in B)\}$

Instead, talking about termination condition, we have:

- data points are partitioned into k clusters
- minimum distance between pairs of clusters is $> r$, where r is a parameter provided by input
- all points are in a cluster, such that the output is a dendogram

A **dendogram** is a tree with input points $\vec{x} \in X$ as leaves, that shows the arrangement/relation between clusters.

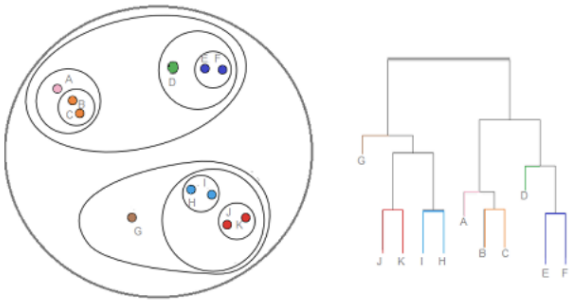


Figure 9.5: Dendogram

What is the correct choice for the number of clusters k? A common approach may be:

- 1) run clustering algorithm for various values of k, obtaining a clustering $C^{(k)} = \{C_1^{(k)}, C_2^{(k)}, ..., C_k^{(k)}\}$ for each value of k considered
- 2) use a score S to evaluate each clustering $C^{(k)}$, getting scores $S(C^{(k)})$ for each value of k
- 3) pick the value of k and clustering of maximum score $C = argmax_{C^{(k)}} S(C^{(k)})$

A very common score based on distance alone is **silhouette**.
Given a clustering $C = (C_1, C_2, ..., C_k)$ of X and a point $\vec{x} \in X$, let $C(\vec{x})$ be the cluster to which $\vec{x}$ is assigned to. Assume $|C_i| \geq 2 \; \forall 1 \leq i \leq k$. Define:

$$A(\vec{x}) = \frac{\sum_{\vec{x'} \neq \vec{x}, \vec{x'} \in C(\vec{x})} d(\vec{x}, \vec{x'})}{|C(\vec{x})| - 1}$$

Given a cluster $C_i \neq C(\vec{x})$, let

$$d(\vec{x}, C_i) = \frac{\sum_{\vec{x'} \in C_i} d(\vec{x}, \vec{x'})}{|C_i|}$$

and $B(\vec{x}) = min_{C_i \neq C(\vec{x})} d(\vec{x}, C_i)$.
Then the silhouette $s(\vec{x})$ of $\vec{x}$ is:

$$s(\vec{x}) = \frac{B(\vec{x}) - A(\vec{x})}{max\{B(\vec{x}), A(\vec{x})\}}$$

So the intuition is that $s(\vec{x})$ measures if $\vec{x}$ is closer to points in its "nearest cluster" than to the cluster it is assigned to.
The question is, what is the range for $s(\vec{x})$?
The silhouette of clustering $C = (C_1, C_2, ..., C_k)$ is

$$S(C) = \frac{\sum_{\vec{x} \in X} s(\vec{x})}{|X|}$$

The higher $S(C)$, the better clustering quality.