ML Notes

Riccardo Cappi

January 2024

0.1 Disclaimer

These are just my notes that I used to prepare for the exam. So, probably, there will be both spelling and conceptual errors. Feel free to contact me at riccardo.cappi@studenti.unipd.it if you find any errors. This is the github repo where you can find the latex files of the notes: `https://github.com/riccardocappi/Computer-Science-notes`

Contents

Chapter 1

Lec 03 - Machine learning Keywords

1.1 Keywords

- **Input/Instance space $\mathbf{x} \in X$:** Representation of model's input (e.g. you can choose a Vector as a representation for your input). It contains all the possible inputs for a model. Suppose the model takes in a vector, $input = [x1, x2], x1, x2 \in [1, 10]$, then we have 10^2 possible inputs.
- **Output space $y \in Y$:** In supervised learning we want to perform a prediction based on the input. This prediction can be in the form of:
  - Binary Classification $y \equiv \{-1, +1\}$
  - Multi-Class Classification $y \equiv \{1, ..., m\}$
  - Regression $y \equiv \mathbb{R}$
- **Oracle/Nature:** It determines how examples are generated. We can have two cases of Oracle
  - Target function $f : X \to Y$ It's deterministic and given an object of the input space returns an object of the output space. This function is ideal and **unknown**.
  - Probability distribution $P(\mathbf{x}), P(y \mid \mathbf{x})$ The *selection* of y occurs from a probability distribution. This distribution is still unknown
- **Training set:** Set of pairs $\{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)\}$ where each pair is composed by an instance of the input space and it's corresponding label.

x	y
000	0
001	1
010	1
.	.
.	.
.	.

  Data are typically:

  - Independent: Given two pairs A, B $P(A \mid B) = P(A)$. The choice of one pair is independent from the choice of other pairs.
  - Identically distributed: All pairs are generated by the same probability distribution (the Oracle) $P(\mathbf{x}, y) = P(\mathbf{x})P(y \mid \mathbf{x})$. *Concept drift* is when data aren't identically distributed

- **Hypothesis space:** A **predefined** set of hypothesis/functions $H \equiv \{h \mid h : X \to Y\}$
- **Empirical error/risk:** Discrepancy between the target function f and my approximation of that function $g \in H$ (chosen from the hypothesis space) **on training data**. For example, in a binary classification problem we can compute empirical error as follows:

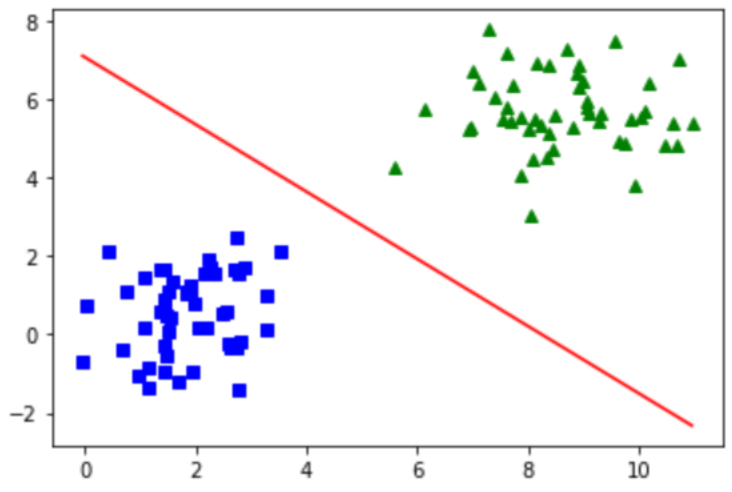
$$\frac{1}{n}\sum_{i=1}^{n}[\![y_i \neq g(\mathbf{x}_i)]\!]$$

where $[\![\cdot]\!]$ is a function that is 1 if $\cdot$ is true and 0 otherwise.

- **Ideal error:** The **expected** error on given hypothesis g and pairs $(\mathbf{x}, y)$. This can only be estimated. One way to estimate this quantity is testing the model over new examples that are not in the training set (test set)
- **Inductive bias:** Since the hypothesis space can't contain all possible functions, we must make assumptions about the type of the unknown target function. The inductive bias consists of:
  - The hypothesis space: how H is defined
  - The learning algorithm: how H is explored

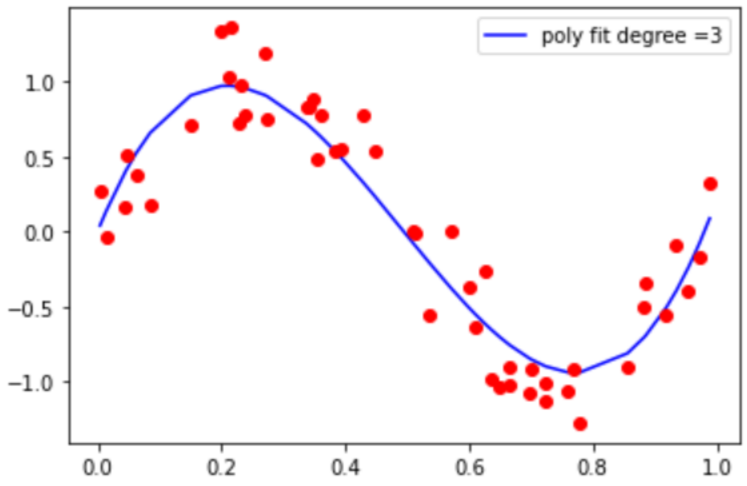
  Examples of Inductive bias:

  - **Hyperplanes in** $\mathbb{R}^2$: We chose as input space points in the plane $X = \{y \mid y \in \mathbb{R}^2\}$, and as hypothesis space the dichotomies induced by hyperplanes in $\mathbb{R}^2$, that is, $H = \{f_{w,b}(y) = sign(\mathbf{w} \cdot y + b), \mathbf{w} \in \mathbb{R}^2, b \in \mathbb{R}\}$.

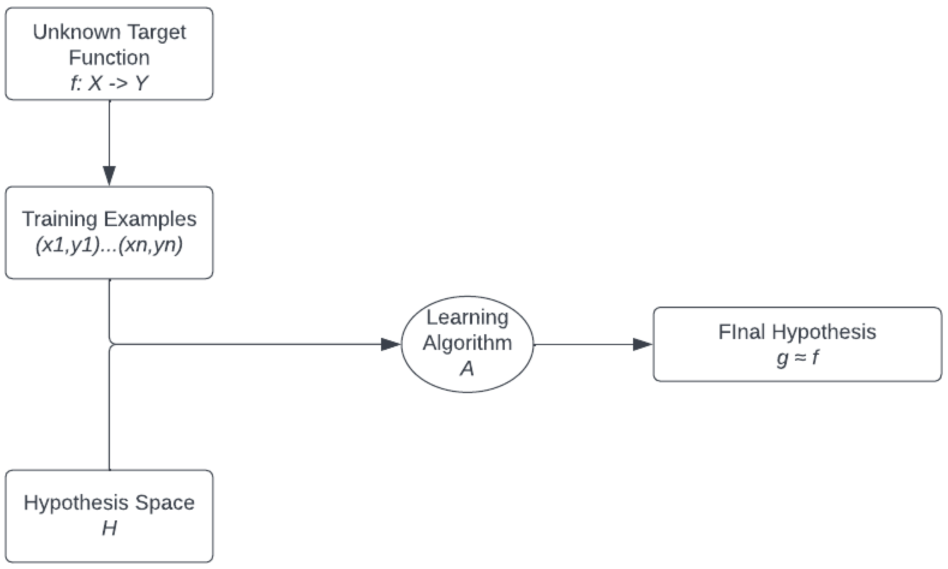


    In this case the assumption is that examples are linearly separable

  - **Polynomial functions**: Given a training set $S = \{(x_1, y_1), ..., (x_n, y_n)\}, x \in \mathbb{R}, y \in \mathbb{R}$, the hypothesis space is the one containing functions of type: $h_w(x) = w_0 + w_1x + w_2x^2 + ... + w_px^p, p \in \mathbb{N}$. The assumption is on the degree p of the polynomial function.



1.2 How Supervised Learning works



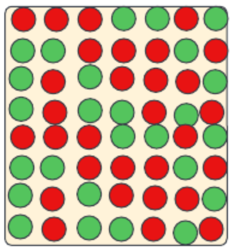
The *Training examples* are generated according to the *Target function* (unknown). Once we have this set of pairs, we choose the *hypothesis space*. The *learning algorithm* (e.g a Neural Network) searches in the hypothesis space for a function g that approximates the target function f. Then we can check if selected function g generalizes well testing it over new unseen examples (minimizing ideal error).

Chapter 2

Lec 04 - PAC, Generalization and SRM

2.1 Hoeffding's Inequality

Let's start with an example. Consider a bin full of red and green marbles.



We denote:

- $P(red) = \pi$ the probability to draw a red marble (unknown)
- $P(green) = 1 - \pi$ the probability to draw a green marble (unknown)

Then we draw N marbles (the *sample*) from the bin, **independently**[1], and we set σ as the fraction of red marbles in the sample. So π represent the proportion of red marbles in the **bin**, while sigma represent the proportion of red marbles in the **sample**. So the question is, does σ say anything about π?

Consider the following formula:

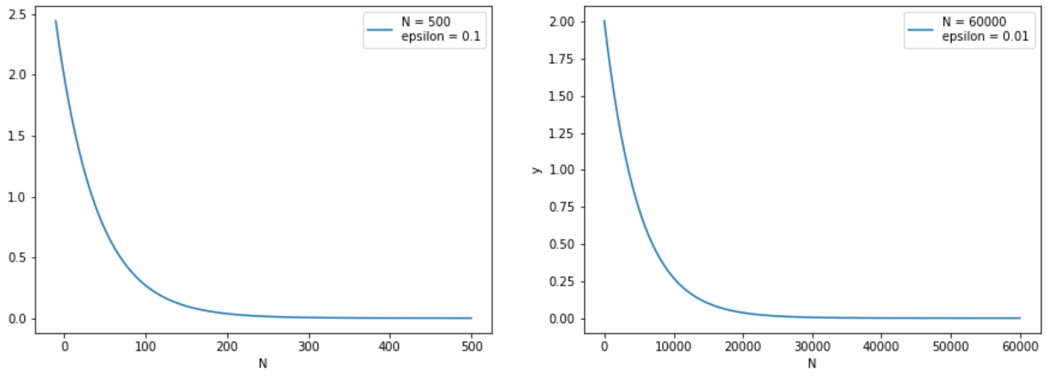
$$P(|\sigma - \pi| > \epsilon) \leq 2e^{-2\epsilon^2 N}$$

It's called **Hoeffding's Inequality** and it means that in a large sample (large N), the value of σ is likely close to π (within ϵ).

- $|\sigma - \pi| > \epsilon$ is called *bad event* because when the absolute value of the difference between σ an π is greater than ϵ, it means that σ an π are not so similar. This event depends on the *tolerance* value ϵ that we choose.

[1]the draw of one marble doesn't influence the draw of the next one

- $2e^{-2\epsilon^2 N}$ is a negative exponential curve.

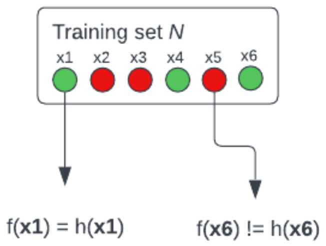


As you can see from the graphs above, the curve decreases as N increases. So with larger samples, the *bad event* will occur with a lower probability. Furthermore, if we set a stricter tolerance value (e.g $\epsilon = 0.01$) the curve decreases more slowly.

For example, if we set $\epsilon = 0.1$, $P(|\sigma - \pi| > \epsilon)$ is the probability of having a discrepancy between σ and π greater than 10%; this probability, for $N = 200$, is ≤ 0.03663127777746833. So $\sigma = \pi$ is **P.A.C** (Probably Approximately Correct).

2.2 Connection to Learning

- The target function $f : X \to Y$ is unknown
- The bin is the input space X which contains all possible inputs for a model.
- The sample N is the training set.
- The training set is composed by a set of pairs $\{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)\}$ where $y_i = f(\mathbf{x}_i)$. Once we have fixed an hypothesis $h : X \to Y$ (chosen from the hypothesis space), we can "color" each example in N as follows:
  - green if $h(\mathbf{x}) = f(\mathbf{x})$
  - red if $h(\mathbf{x}) \neq f(\mathbf{x})$



Now we can say that σ is the **empirical error**, while π is the **ideal error** (for this h) and following the previous example we can observe that σ tends to be close to π. If we have a small error σ it's more and more probable that h is a good approximation of the target function **in general** as N increases. So this process **verifies** how good the approximation h of f is, but it has nothing to do with learning process, because π and σ depend on which h we choose. How do we choose h?

2.2.1 Learning

First of all, let's change the notation:

- empirical error on training data $\sigma \rightarrow E_i(h)$
- ideal error $\pi \rightarrow E_o(h)$
- $P(|E_i(h) - E_o(h)| > \epsilon) \leq 2e^{-2\epsilon^2 N}$. The bound depends on h

The hypothesis space $H \equiv \{h_1, h_2, ..., h_m\}$ it's composed by many different hypotheses $h_1, .., h_m$ and each of them has an empirical error $E_i(h_i)$ and an ideal error $E_o(h_i)$. Learning algorithms search in the hypothesis space (following some criterion) and select an hypothesis h_i that minimizes $E_o(h_i)$. However, the Hoeffding's Inequality can't be used for learning because in Supervised Learning we have to choose among several hypotheses. If we consider a worst case analysis, where all the bad events over the hypothesis space are **disjointed**, we have to bound the probability of interest in the following way:

$$P[|E_i(h_i) - E_o(h_i)| > \epsilon] \leq \sum_{m=1}^{M} P[|E_i(h_m) - E_o(h_m)| > \epsilon] \leq 2Me^{-2\epsilon^2 N}$$

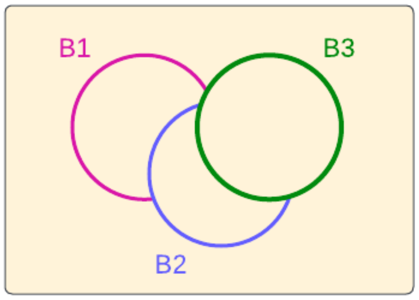
where M is the number of hypotheses (can also be infinite).

The formula above says that since h_i can be any of all $h_1, .., h_m$ hypotheses in the hypothesis space, the bad event $|E_i(h_i) - E_o(h_i)| > \epsilon$ can happen for h_1 **or** for h_2 **or** for h_3 **or** ... for h_m. So, since we are assuming that all bad events are disjointed ($P(A \cup B) = P(A) + P(B)$), $P[|E_i(h_i) - E_o(h_i)| > \epsilon]$ must be bounded by $\sum_{m=1}^{M} P[|E_i(h_m) - E_o(h_m)| > \epsilon]$. Note that for very big M the resulting bound is $>> 1$, so it's useless.

Just to recap a little bit:

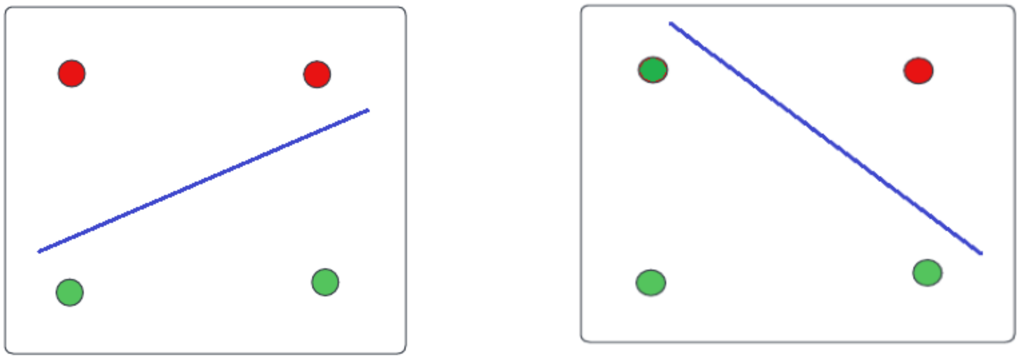
- **Testing:** Hoeffding's Inequality can help us to verify if a **fixed** hypothesis h is a good approximation of the target function f over a sample N.
- **Training:** Hoeffding's Inequality is not so useful for learning process since we have to choose among several hypotheses.

Fortunately, bad events are not always disjointed but they are overlapped.



So M can be replaced by the **growth function** $m_H(N) \leq 2^N$ which is related to the complexity of the hypothesis space. When the hypothesis space has a low complexity bad events overlap a lot ($m_H(N) << 2^N$), when it's very complex they tend to be disjointed. So if $m_H(N)$ is **polynomial** with respect to N, the upper bound $2Me^{-2\epsilon^2 N}$ tends to 0 as N increases. How can i compute the complexity of the hypothesis space ?

2.2.2 VC-Dimension



Let's consider a plane with 4 points. If we choose the hyperplanes in $\mathbb{R}^2$ as hypothesis space H we can divide the points with a line and classify them with two colors (red and green). These partitions are called dichotomies. Note that this particular H **can't** implement all possible dichotomies.

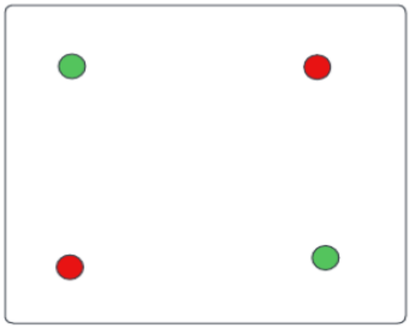


Figure 2.1: This classification can't be implemented by any hyperplane in H

- **Shattering:** Given $S \subset X, S$ is shattered by the hypothesis space H iff
$$\forall S' \subseteq S, \exists h \in H, s.t. \forall x \in S, h(x) = 1 \iff x \in S'$$
That means that H is able to implement all possible dichotomies of S.

- **VC-dimension:** The VC-dimension of an hypothesis space H defined over an instance space X is the size of the largest finite subset of X shattered by H:
$$VC(H) = \max_{S \subseteq X} |S|$$

such that S is shattered by H.

If we consider $H_1 = \{f_{w,b}(y) | f_{w,b}(y) = sign(w \cdot y + b), w \in \mathbb{R}^2, b \in \mathbb{R}\}$ as hypothesis space (hyperplanes in $\mathbb{R}^2$), as we saw before $VC(H_1) = 3$ because it doesn't exist **any** configuration of 4 points that can be shattered by H_1. In general, VC-dimension of hyperplanes in $\mathbb{R}^n = n + 1$: in 2 dimensions $VC(H) = 3$, in 3 dimensions $VC(H) = 4$ and so on.

2.2.3 VC bound and SRM

Consider a binary classification learning problem with:

- Training set $S = \{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)\}$.
- Hypothesis space $H = \{h_\theta(\mathbf{x})\}$.
- Learning algorithm L, returning the hypothesis $g = h_\theta^*$ minimizing the empirical error on S, that is $g = argmin_{h \in H} error_S(h)$.

It is possible to derive [2] an upper bound of the ideal error which is valid with probability $(1 - \delta)$, δ being arbitrarily small, of the form:

$$error(g) \leq error_S(g) + F(\frac{VC(H)}{n}, \sigma)$$

where $error(g)$ is the ideal error. The term $F(\frac{VC(H)}{n}, \sigma)$ is called **VC-confidence** and it depends on:

- The training size n (inversely).
- The VC-dimension of $HVC(H)$ (proportionally).
- The confidence σ (inversely).

As the VC-dimension grows, you usually observe that the empirical error decreases and that the VC confidence increases. So you can use the inductive principle **Structural Risk Minimization (SRM)** in order to minimize the right side of the confidence bound to get a tradeoff between the empirical error and the VC confidence.

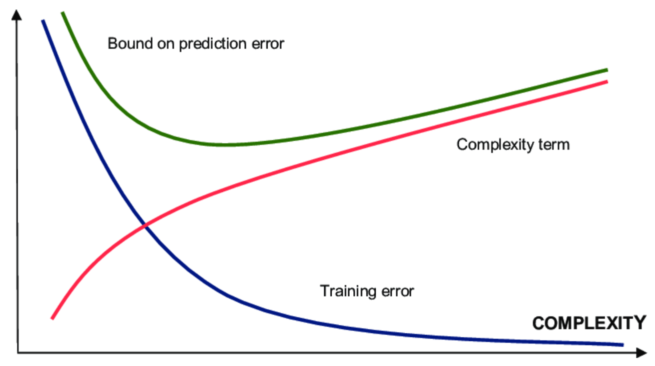


Figure 2.2: The green curve *Bound on prediction error* is the sum between empirical error curve and VC confidence curve.

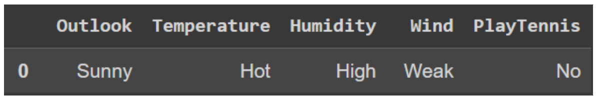
[2]We will not see the derivation here.

Chapter 3

Lec 05 - Decision Trees I

3.1 Decision Trees

Decision trees (DTs) allow to learn **discrete** functions that are representable by a tree. Given a training set instance like the following one:



	Outlook	Temperature	Humidity	Wind	PlayTennis
0	Sunny	Hot	High	Weak	No

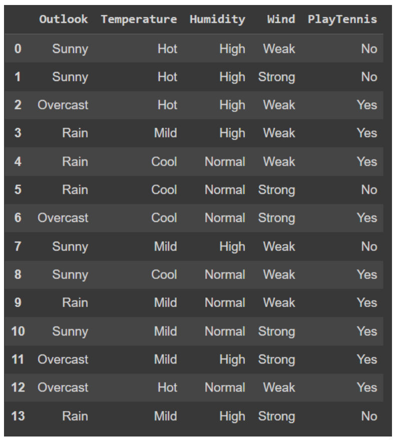
Attributes are $[Outlook, Temperature, Humidity, Wind]$, while $PlayTennis$ is the feature to predict (we want to predict if a day is suitable to play tennis). The values that $PlayTennis$ can take are called **classes**. In a decision Tree:

- An **inner node** corresponds to an attribute of an instance.
- A **branch** descending from a node corresponds to one of the possible values the attribute can assume.
- A **Leaf node** assigns a classification.

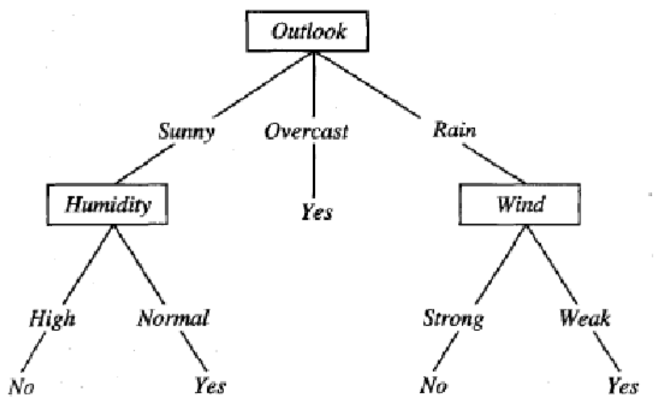
The classification of an instance works in the following way:

- Start from the root
- Select the attribute attached to the current node
- Select a subtree following the branch corresponding to the value of that attribute in the instance.
- If we reach a leaf node we classify the instance with the associated label, else return to step 2.

Let's consider the following Decision Tree related to the *tennis dataset* mentioned before.



	Outlook	Temperature	Humidity	Wind	PlayTennis
0	Sunny	Hot	High	Weak	No
1	Sunny	Hot	High	Strong	No
2	Overcast	Hot	High	Weak	Yes
3	Rain	Mild	High	Weak	Yes
4	Rain	Cool	Normal	Weak	Yes
5	Rain	Cool	Normal	Strong	No
6	Overcast	Cool	Normal	Strong	Yes
7	Sunny	Mild	High	Weak	No
8	Sunny	Cool	Normal	Weak	Yes
9	Rain	Mild	Normal	Weak	Yes
10	Sunny	Mild	Normal	Strong	Yes
11	Overcast	Mild	High	Strong	Yes
12	Overcast	Hot	Normal	Weak	Yes
13	Rain	Mild	High	Strong	No



The instance [O=Sunny, T=Hot, H=High, W=Strong] is classified with label *NO* (not suitable to play tennis). In fact, following the path $O = Sunny \rightarrow H = High$ we end up in a leaf node with label 'NO'. Note that a decision tree can be represented as a boolean function:

- Each path from the root to a leaf node codifies a conjunction of constraints on the attribute values of the instance.
- Different paths that lead to a same classification codify disjunctions of conjunctions.

So the classification can be seen as a series of **DNF** (disjunctive normal form) formulas, one DNF for each class. For example, DNF corresponding to *YES* is (O=Sunny **and** H=Normal) **or** (O=Overcast) **or** (O=Rain **and** W=Week). This last feature, absent in most of the machine learning techniques, makes decision trees comprehensible and interpretable by humans. So they are particularly interesting for medical, biological and financial applications where interpreting a model is very important.

3.1.1 DTs learning algorithm

A recursive implementation of the **ID3** algorithm (the most popular learning algorithm for DTs) is the following:
ID3(S**,** A**)** Given a sample S and a set of attributes A

- If examples in S are all of the same class c, return a leaf node with label c.
- if A is empty, return a leaf node with label equals to the majority class[1] in S.
- Select $a \in A$, the **optimal** attribute in A and remove it from A ($A = A - a$).
- For each distinct value v_j that a can take in S_a, partition S according to v_j ($S_{a=v_j}$) and return the tree T having sub-trees the trees obtained by recursively calling **ID3**($S_{a=v_j}, A$).

Note that in the 4th point of the algorithm, when you partition S according to v_j, it is possible that in S there are instances having only some of the possible values that the best attribute a can take, so you have to consider only these values in that specific S_a partition.

3.1.2 Select optimal attribute

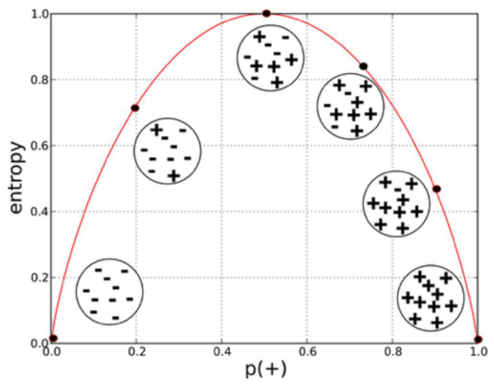
Different learning algorithms for DTs mainly differ in how they select the optimal attributes. ID3 uses the concepts of **Entropy** and **Information Gain**.

[1]the class associated with the largest number of examples in S

- **Entropy**: Let C be the number of classes and S_c the subset of S of instances of class c, the entropy is computed as follows:

$$E(S) = -\sum_{c=1}^{C} p_c log_2(p_c)$$

where $p_c = \frac{|S_c|}{|S|}$. In the case of binary classification (only two classes), it has the following shape:



Entropy is lowest at the extremes, when S either contains no positive instances or only positive instances (sample is pure without disorder) and it's highest where positive and negative instances are evenly split.

- **Information Gain:** The optimal attribute will be the one that maximizes the Information Gain $G(S, a)$

$$G(S, a) = E(S) - \sum_{v \in V(a)} \frac{|S_{a=v}|}{|S|} E(S_{a=v})$$

where

  - S is the sample (could be the entire training set or a partition of it depending on which level of recursion we are in: see 4^{th} point of the ID3 algorithm.
  - a is the selected attribute.
  - $E(S)$ is the total entropy of S.
  - $V(a)$ is the set of distinct values that a can take in S_a.
  - $S_{a=v}$ is the partition of S in which instances have the attribute a equals to v.

We can generalize the notion of Information Gain to other impurity measures:

- **Cross-Entropy:** $I_H = -\sum_{c=1}^{C} p_c log_2(p_c)$.
- **Gini Index:** $I_G = 1 - \sum_{c=1}^{C}(p_c^2)$.
- **Misclassification:** $I_E = 1 - \max_c(p_c)$.

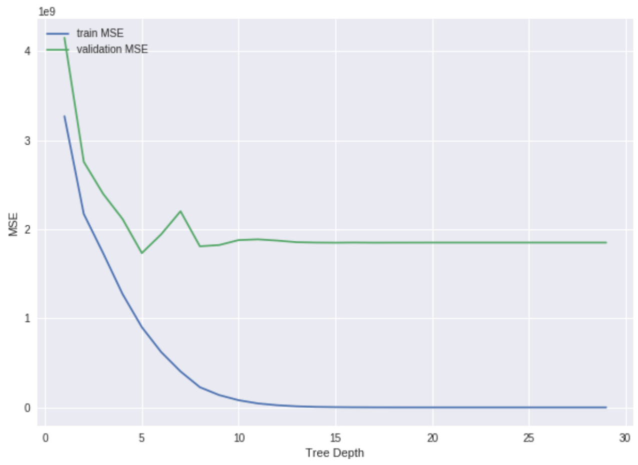
Let I be any of the impurity criteria mentioned above, the Information Gain definition becomes:

$$G(S, a) = I(S) - \sum_{v \in V(a)} \frac{|S_{a=v}|}{|S|} I(S_{a=v})$$

3.1.3 DTs problems

The information gain tends to favor attributes that can assume many possible values.
Example: Consider a sample S with an attribute consisting in dates. This attribute is likely going to be the one with maximum gain because each subset $S_{a=v}$, that is composed by **only one instance**, will be pure with zero impurity. In fact, the entropy of a subset with only one class is 0.

Another problem is the overfitting: The model is very accurate on training data but not on test data.



We can observe that the problem increases as the number of nodes increases. A partial solution can be to set a minimal number of examples to accept on leaf nodes or limit the maximal depth of the tree (at a certain point i don't split S anymore and i consider that node as a leaf with label the majority class in S).

3.1.4 When to use DTs

Decision trees are useful when dealing with problems having the following characteristics:

- A fixed set of attributes and, for each attribute, a fixed set of values
- Discrete values for the attributes (decision trees can also be extended to deal with continuous values).
- Target function with discrete output values (classification problems).
- The target function can be approximated by disjunctions of boolean functions (Inductive bias).
- Training examples can contain noise (two or more instances with same attributes values but different classes) and missing values.

Chapter 4

Lec 06 - Decision Trees II

4.1 Inductive Bias

Like other algorithms, ID3 can be seen as a search in a hypothesis space for a hypothesis that best fits the data. The inductive bias of ID3 consists of:

- The hypothesis space: is the set of possible decision trees. The hypothesis is searched over a **subset** of DNF. This restriction is due to the fact that the root of the tree will always be the first term of the conjunction, so we can't consider the set of all the possible DNF.
- How the hypothesis space is explored: The search is of type **hill climbing**. ID3 starts from the empty tree and continues with more and more elaborate trees. Then, it stops when it finds a tree that is consistent with the examples. Hence, applying the information gain criterion implies that:
  - Shorter trees are preferred to larger tree. This is because, at each level, it finds sets of examples with minimum entropy.
  - Attributes with high information gain are closer to the root.

4.2 Real-valued attributes

Let's consider an extension of the *standard* ID3 algorithm that supports also real-valued attributes.

Given a real-valued attribute A, a new **boolean** pseudo-attribute is dinamically created as:

$$A_c = \begin{cases} true & if\ A < c \\ false & otherwise \end{cases}$$

where c is a threshold.

Then, the content of this attribute is used to compute the information gain as the *standard* ID3 does. How can we select the "*correct*" value for c? One possibility is to select the value c that provides the maximal information gain. Given the set of possible values that an attribute A can take $\{v_1, ..., v_n\}$, the optimal value of c can be proved to be always localized between two values with different labels. So, we can find the best value of c looking at the information gain obtained by splitting A according to $c = \frac{v_i+v_{i+1}}{2}$, where v_i and v_{i+1} are values that examples of the data-set with different labels have for attribute A.

Note that in this case, when we find the best attribute, differently from the case of discrete-valued ID3, that attribute it's not removed and can be used multiple times on the same path (in such case, the optimal threshold will change). We remove it only when it's not informative anymore.

4.3 Attributes with missing values

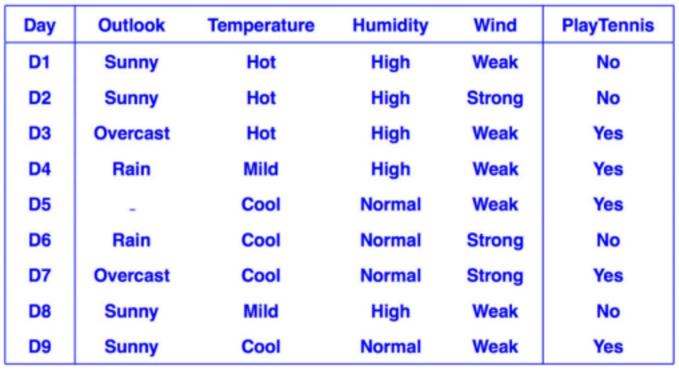
In practical applications it can happen that, for some instances, some attributes don't have a value. (missing value).

Given a set of examples $\hat{Tr}$, when a value of an attribute A is missing for an example (x, y), we can:

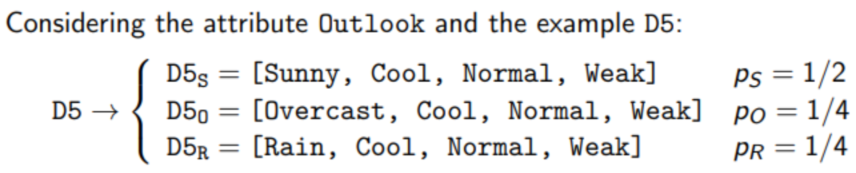
1. Assign to that missing value the most frequent value of the attribute A in $\hat{Tr}$.
2. Assign the most frequent value of A in $\hat{Tr}$, but just considering those examples with label y (of the same class).
3. Consider the set of **distinct** values that attribute A can have $V(A)$. Then, for each value $v_i \in V(A)$, we consider their probability of occurrence $P(v_i|\hat{Tr})$, estimated on $\hat{Tr}$. We **substitute** the example (x, y) with $|V(A)|$ *fractionated examples*, one for each possible value v_i, and we weight them according to $P(v_i|\hat{Tr})$.

4.3.1 Handling missing values with solution 3

Let's see an example of the behavior of the third method for handling missing values mentioned before.



Day	Outlook	Temperature	Humidity	Wind	PlayTennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	-	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes



Considering the attribute Outlook and the example D5:

$$\text{D5} \rightarrow \begin{cases} \text{D5}_\text{S} = \texttt{[Sunny, Cool, Normal, Weak]} & p_S = 1/2 \\ \text{D5}_\text{O} = \texttt{[Overcast, Cool, Normal, Weak]} & p_O = 1/4 \\ \text{D5}_\text{R} = \texttt{[Rain, Cool, Normal, Weak]} & p_R = 1/4 \end{cases}$$

As you can see, the example **D5** is substituted with three other examples (one for each distinct value of the attribute *Outlook*) with their respective weights. If multiple attributes have missing values, the examples are fractionated again, and the associated weight will be the product of the weights obtained by considering the single attributes in turn.

In this case, we need to modify the definition of Information Gain such that the fractionated weights are considered. Basically, we compute the Information Gain using the same formula, but substituting the concept of cardinality as follows:

$$|Q| = \sum_{q \in Q} w_q$$

The cardinality of a set Q is the sum of the weights of the examples in the set. Note that unfractionated examples will have weight 1.

Consider now the **classification** of a new instance that has missing values. In this case, each fractionated example is classified (using the usual classification), and, for each label, we cumulate the weights of examples classified with that label. The final classification is performed by looking at the label with the highest cumulated value.

4.4 Overfitting

A possible solution to overfitting is **Reduced-Error Pruning.**

Split the training set Tr in $Tr^{'}$ and Va (validation set), $Tr = Tr^{'} \cup Va$. Given a decision tree already learned, repeat until the performance gets worse:

1. For each inner node, evaluate the accuracy in Va obtained by pruning the tree on that node. We assign the label y to this new leaf where y is the majority label in the pruned sub-tree.
2. If the accuracy of the new tree is higher than the *original* one, return the tree.

Another possibility is **Rule-Post Pruning:**

The basic idea is to turn the decision tree into a set of rules, and then do the rule pruning. A rule R_i is generated for each path (r, f_i) from the root r to the i-th leaf f_i. R_i will be of the form:

$$IF(Att_{i_1} = v_{i_1}) \cap (Att_{i_2} = v_{i_2}) \cap ...(Att_{i_k} = v_{i_k})\, THEN\, label_{f_i}$$

An independent pruning is carried out for each rule R_i. Basically, after removing some constraints from the preconditions of the rule, we look at the accuracy of the new rule on the validation set. If we can't improve the accuracy anymore, we return the new pruned rule. Then, the pruned R_i are sorted in descending order of performance; eventually, add a default rule that returns the most frequent class.

The classification is performed following the order established for the rules:

- The first rule whose preconditions are satisfied by the instance is used to classify it.
- If no rules have preconditions satisfied, the default rule is used to classify the instance (most frequent class).

Note that the transformation Tree $\rightarrow$ Rules **changes** the hypothesis space. We can obtain, by pruning, rules that are not expressible by a decision tree. For example, when we classify a new instance using a decision tree, we first look at the attribute attached to the root of the tree and we follow the path to a leaf according to the attributes values of the instance. But if we prune the corresponding rule at the root, the constraint of starting from the root is not present anymore.

Usually, Post-Rule pruning is able to improve the performance with respect to corresponding decision tree and performs better than Reduce-Error Pruning.

Chapter 5

Lec 07 - Neural Networks I

5.1 Introduction

There are a lot of different neural networks models which answer different computational/learning needs:

- **Supervised learning** (classification, regression, time series prediction, ...)
- **Unsupervised learning** (clustering, representation learning, ...)

All these models differ for: Network topology, function computed by single neurons, training algorithm and how the training proceeds.

Characteristics of the problem:

- Input: discrete and/or real-valued vectors.
- Output: discrete (classification) or real-valued (regression) vectors.
- Data (input and/or output) can be noisy (e.g. the same instance has two different labels) and the form of the target function is unknown. Note that having no assumption on the target function is not always a pro, but can be a problem.
- Having long learning time is acceptable and a quick evaluation of the learned function is required. These models can be used, for example, for real time predictions.
- The final solution does not need to be understood by a human expert (”*black box problem*”).

Examples of application of Neural Networks are speech recognition, image classification and time series prediction.

5.2 Perceptron

Consider the space of hyperplanes in $\mathbb{R}^n$, where n is the dimension of the input.

$$H = \{f_{(\mathbf{w},b)}(\mathbf{x}) = sign(\mathbf{w} \cdot \mathbf{x} + b) : \mathbf{w},\mathbf{x} \in \mathbb{R}^n, b \in \mathbb{R}\}$$

where $\mathbf{w}$ is a vector of weights and b is the **bias** term.

We can redefine H as:

$$H = \{f_{\mathbf{w}'}(\mathbf{x}') = sign(\mathbf{w}' \cdot \mathbf{x}') : \mathbf{w}', \mathbf{x}' \in \mathbb{R}^{n+1}\}$$

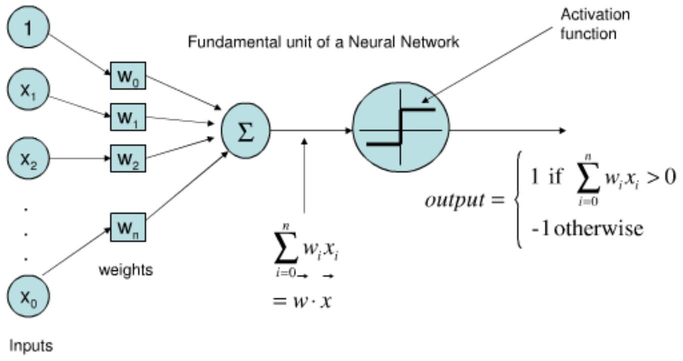
after the following change of variables:

$$\mathbf{w}^{'} = [b, \mathbf{w}], \quad \mathbf{x}^{'} = [1, \mathbf{x}]$$

it follows that:

$$\mathbf{w}^{'} \cdot \mathbf{x}^{'} = b + \sum_{i=1}^{n} \mathbf{w}_i \mathbf{x}_i = \mathbf{w} \cdot \mathbf{x} + b$$

Basically, we add a dimension to **w** and **x** just to simplify the notation of H.


Fundamental unit of a Neural Network
Activation function
1
X1
X2
Xo
W0
W1
W2
Wn
weights
Inputs
Σ
$\sum_{i=0}^{n} w_i x_i = \vec{w} \cdot \vec{x}$
$output = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$

The model described in the image above is called **Perceptron**. It first computes the dot-product between the weights **w** and the input **x**. The result of this computation is usually called *net*

$$net = \sum_{i=0}^{n} w_i x_i$$

The final output is obtained by applying the **step function** to the *net*.

$$o = \sigma(net) = sign(net)$$

We will refer to this neuron (and associated learning algorithm) as Perceptron.

Since the hypothesis space of the Perceptron is defined as the hyperplanes in $\mathbb{R}^n$, it converges only if the examples in $\mathbb{R}^n$ are **linearly separable**. Otherwise, it will never *find* a hyperplane that separates them.

5.2.1 Perceptron: learning algorithm

Assume to have training examples in $\mathbb{R}^n$ that are linearly separable:

Input: Training set $S = \{(\mathbf{x}, t), \mathbf{x} \in \mathbb{R}^{n+1}, t \in \{-1, +1\}, \eta \geq 0\}$

1. Initialize the value of the weights **w** randomly;
2. Repeat (N epochs)
   (a) Select (randomly) one of the training examples $(\mathbf{x}, t)$
   (b) if $o = sign(\mathbf{w} \cdot \mathbf{x}) \neq t$ then
$$\mathbf{w} \leftarrow \mathbf{w} + \eta(t - o)\mathbf{x}$$

A small value of the learning rate η can make the learning process slow but more stable, that is, it prevents sharp changes in the weights vector. If the training set is linearly separable, it can be shown that the Perceptron training algorithm terminates in a finite number of steps.

Let R be the radius of the **smallest** hyper-sphere centered in the origin enclosing all the instances (how the instances are *spread out*). Let γ be the maximal value such that $t_i net_i = t_i(\mathbf{w} \cdot \mathbf{x}_i) \geq \gamma > 0$ (how much the instances are *separated*). Then, it can be shown that the number of steps of the Perceptron algorithm is bounded from above by the quantity R^2/γ^2. Basically, bigger *distance* between instances means a smaller number of steps to converge and vice versa.

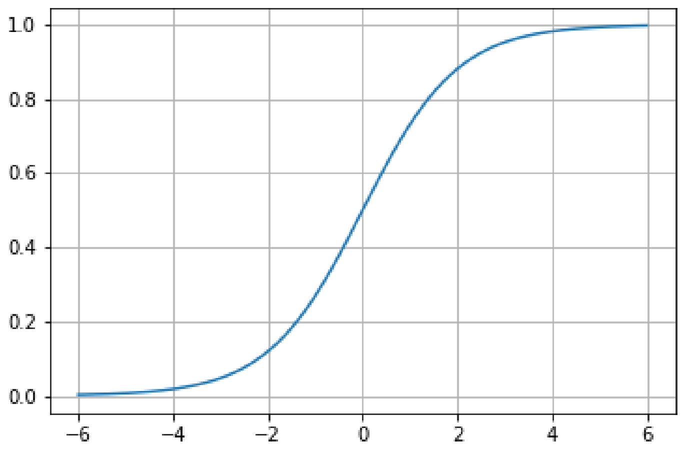
Chapter 6

Lec 08 - Neural Networks II

6.1 Activation functions

Let's consider the Perceptron artificial neuron. The output of the neuron is obtained by computing the *net* and passing it to the step function (hard threshold). Due to the discontinuity of this function, it could be difficult to combine multiple neurons, so, other **activation functions** can be defined to solve this problem. A first alternative to the step function is the **sigmoid activation function**, which is defined as follows:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



This function has the advantage of being **derivable**. So, the output of this new artificial neuron would be:

$$o = \sigma(net) = \frac{1}{1 + e^{-net}}$$

6.2 Delta rule

The **Delta rule** is a weight update rule different from the Perceptron rule that allows to obtain a best-fit solution approximating the target function. In particular, it exploits **gradient descent** to explore the hypothesis space and select the hypothesis that best approximates the target function. Its goal is to minimize an error function, appropriately defined.

6.2.1 Gradient descent

Derivative:
The derivative tells us the slope of a function at any point. Given a point:

- Positive derivative means that the function increases at that point
- Negative derivative means that the function decreases at that point.
- Null derivative means that there is a stationary point (minimum, maximum or saddle point).

In vector calculus, the gradient of a scalar-valued differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ is a vector-valued function $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$ whose value at point p is the vector whose components are the partial derivatives of f at p.

The gradient vector can be interpreted as the direction and rate of fastest increase of the function. The gradient is the zero vector at a point if and only if the point is a stationary point.

Let $f(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}$ be a vector-valued function, find the vector $\theta \in \mathbb{R}^n$ that minimizes the function f:

$$\theta = argmin_{\mathbf{x}} f(\mathbf{x})$$

This minimization problem can be solved using gradient descent. Starting from a random configuration of θ, each parameter is updated in the following way:

$$\theta_{k+1} = \theta_k - \eta \nabla f(\theta_k)$$

where:

- $\nabla f(\theta_k)$ is the partial derivative of the function in θ_k.
- The parameter $\eta > 0$ is known as the *learning rate.*

The derivative term $\frac{\partial}{\partial \theta_k} f(\theta_k)$ can be:

- ≥ 0 it means that the function is increasing, so we are decreasing θ_k in the *right direction.*
- ≤ 0 it means that the function is decreasing, so we are increasing θ_k in the *right direction*

If η is too small, gradient descent can be slow. Anyway, if it is too large, it can overshoot the minimum (fail to converge).

The gradient descent algorithm can be defined as follows:

```
1. $k \leftarrow 0, \theta_0 \in \mathbb{R}^n$
2. while $\nabla f(\theta_k) \neq 0$
    (a) Compute the descent direction $\mathbf{p}_k := -\nabla f(\theta_k)$
    (b) Compute the learning rate (fixed) $\eta_k$
    (c) Update $\theta_{k+1} \leftarrow \theta_k + \eta_k \mathbf{p}_k$
    (d) $k \leftarrow k + 1$
```

The gradient $\nabla f(\theta_k)$ represents the rate of the fastest increase. So, its opposite represents the direction where the function decreases the most. Note that it is not guaranteed that it converges to a global minimum.

6.2.2 Gradient computation (linear activation)

Consider a Perceptron without the step function [1] (linear activation).

$$o(\mathbf{x}) = \sum_{i=0}^{n} w_i x_i = \mathbf{w} \cdot \mathbf{x}$$

and define a measure of the committed error (e.g. mean squared error) given a specific weight vector $\mathbf{w}$:

$$E[\mathbf{w}] = \frac{1}{2N} \sum_{(\mathbf{x}^{(s)}, t^{(s)}) \in S} (t^{(s)} - o(\mathbf{x}^{(s)}))^2$$

where N is the cardinality of the training set S. We want to minimize $E[\mathbf{w}]$ with respect to the parameters $\mathbf{w}$ using gradient descent. Let's start with the derivation of the loss function $E[\mathbf{w}]$:

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \left[\frac{1}{2N} \sum_{s=1}^{N} (t^{(s)} - o^{(s)})^2 \right]$$

$$= \frac{1}{2N} \sum_{s=1}^{N} \frac{\partial}{\partial w_i} \left[(t^{(s)} - o^{(s)})^2 \right]$$

$$= \frac{1}{2N} \sum_{s=1}^{N} 2(t^{(s)} - o^{(s)}) \frac{\partial}{\partial w_i} \left[t^{(s)} - o^{(s)} \right]$$

Note that $t^{(s)}$, which is the target value, is a constant that does not depend on w_i. So, it can be removed from the derivation.

$$= \frac{1}{N} \sum_{s=1}^{N} (t^{(s)} - o^{(s)}) \left(-\frac{\partial}{\partial w_i} \left[\mathbf{w} \cdot \mathbf{x}^{(s)} \right] \right)$$

$\mathbf{w} \cdot \mathbf{x}^{(s)} = w_1 x_1^{(s)} + w_2 x_2^{(s)} + ... + w_i x_i^{(s)} + ...$ The partial derivative $-\frac{\partial}{\partial w_i} \left[\mathbf{w} \cdot \mathbf{x}^{(s)} \right]$ with respect to w_i is equal to $x_i^{(s)}$ because all the other terms are constant. Therefore, the derivation becomes:

$$= -\frac{1}{N} \sum_{s=1}^{N} (t^{(s)} - o^{(s)}) x_i^{(s)}$$

The learning algorithm with gradient descent is defined as follows:

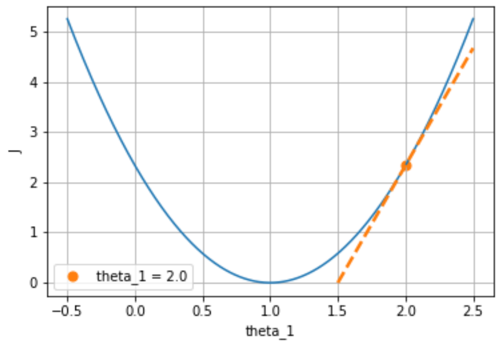
Gradient-Descent(S, η):
η is the learning rate (that encompasses the constant term $1/N$)

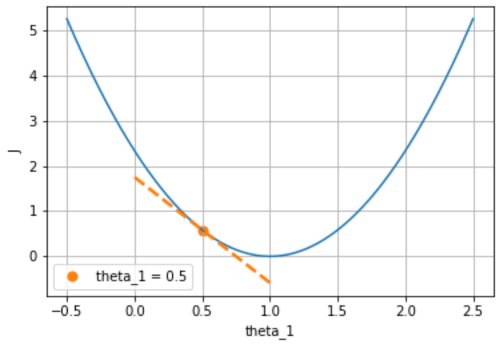
```
1. Initialize the weights $w_i$'s with **small** random values
2. Repeat until the termination condition is met:
    (a) $\Delta w_i \leftarrow 0$
    (b) For each $(\mathbf{x}, t) \in S$:
        i. Compute the output of the neuron $o(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x}$
        ii. For each $i \in \{1, ..., n\}$
                $\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$
```

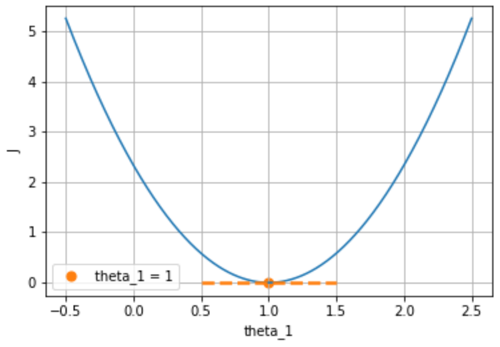
[1]The step function is not derivable, so we wouldn't be able to compute the gradient

(c) For each $i \in \{1, ..., n\}$:
$$w_i \leftarrow w_i + \Delta w_i$$

Note that we update the weights using the $+$ because the derivative of the loss function has a $-$ in front of it (so we need to change the sign).







The graphs above show that this error function has a **convex** shape (the same is also valid in more than 2 dimensions). This implies that there is a unique global minimum.

6.2.3 Gradient computation (sigmoidal activation)

Consider now a Perceptron with sigmoidal activation function:

$$o(\mathbf{x}) = \sigma\left(\sum_{i=0}^{n} w_i x_i\right) = \sigma(\mathbf{w} \cdot \mathbf{x}) = \sigma(y)$$

where $y = \mathbf{w} \cdot \mathbf{x}$ and $\sigma(y) = \frac{1}{1+e^{-y}}$. Following the same approach as before, we want to find the weight vector that minimizes the mean squared error in the training set using a gradient descent based algorithm.

The partial derivative of the error function with respect to w_i is defined as follows:

$$\frac{\partial E}{\partial w_i} = -\frac{1}{N}\sum_{s=1}^{N}(t^{(s)} - o^{(s)})\sigma(y^{(s)})(1 - \sigma(y^{(s)}))x_i^{(s)}$$

Hence, the point 2.b.ii of the algorithm presented above becomes:

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)\sigma(y)(1 - \sigma(y))x_i$$

Although the sigmoid function is a non-linear function, the Perceptron artificial neuron with sigmoidal activation is still a linear model, because it still produces an hyperplane that separates data.

Example:
In the case of binary classification, given the output of the Perceptron $o^{(s)}$, we can choose to predict the class y of a given instance in the following way:

- $y = 1$ if $o^{(s)} \geq 0.5$
- $y = 0$ if $o^{(s)} < 0.5$

For multi-class classification we can use other methods such as the soft-max function.

Chapter 7

Lec 09 - Neural Networks III

7.1 Other activation functions

Multiple layers of cascaded units makes a Neural Network able to implement complex non linear functions. The non linearity of the model is given by the activation functions. In fact, without them (linear activation) the result of the model, even if it's very complex, would still be linear. If we used step activation functions we wouldn't be able to resort to gradient descent to optimize the weights. This is because step function is not derivable. We have to choose one or more activation functions that are derivable.

- **Sigmoid:**
$$\sigma(y) = \frac{1}{1 + e^{-y}}$$
- **ReLu:**
$$r(y) = max(0, y)$$

7.2 Multi-layer Neural Networks: Keywords

Let's define the most important keywords for Multi-layer Neural Networks:

- d **input units**: d input data size $\mathbf{x} \equiv (x_1, ..., x_d)$, $d + 1$ when including the bias in the weight vector $\mathbf{x}^{'} \equiv (x_0, x_1, ..., x_d)$
- n_H **hidden units** (with output $\mathbf{y} \equiv (y_1, ..., y_{n_H})$)
- c **output units**: $\mathbf{z} \equiv (z_1, ..., z_c)$. The number of desired output is $\mathbf{t} \equiv (t_1, ..., t_c)$
- w_{ji} weight from the i-th input unit to the j-th hidden unit (w_j is the weight vector of the j-th hidden unit)
- w_{kj} weight from the j-th hidden unit to the k-th output unit (w_k is the weight vector of the k-th output unit)

7.3 Learning algorithm

Let's consider, for example, a Neural Network with just one hidden layer and **sigmoid activation functions**. The basic idea of the learning algorithm consists in two phases:

- **Forward phase:** for each example in the training set, present it to the network and compute the output.
- **Backward phase:** Back-propagate the committed error and update the weights of the hidden units accordingly.

This algorithm is called **Backpropagation**

As we said before, this algorithm is based on gradient descent, so we need to minimize an error function $E[\mathbf{w}]$ that can be defined as follows:

$$E[\mathbf{w}] = \frac{1}{2cN} \sum_{s=1}^{N} \sum_{k=1}^{c} (t_k^{(s)} - z_k^{(s)})^2$$

In order to implement Backpropagation, we need to first compute the gradient for the weights of both output and hidden units.

- **Gradient of the weights of an output unit:**

$$\frac{\partial E}{\partial w_{\hat{k}\hat{j}}} = -\frac{1}{cN} \sum_{s=1}^{N} (t_{\hat{k}}^{(s)} - z_{\hat{k}}^{(s)}) z_{\hat{k}} (1 - z_{\hat{k}}) y_{\hat{j}}^{(s)}$$

  where $y_{\hat{j}}^{(s)}$ is the output of the $\hat{j}$-th hidden unit.

- **Gradient of the weights of a hidden unit:**

$$\frac{\partial E}{\partial w_{\hat{j}\hat{i}}} = -\frac{1}{cN} \sum_{s=1}^{N} y_{\hat{j}}^{(s)} (1 - y_{\hat{j}}^{(s)}) x_{\hat{i}}^{(s)} \sum_{k=1}^{c} (t_{\hat{k}}^{(s)} - z_{\hat{k}}^{(s)}) z_{\hat{k}} (1 - z_{\hat{k}}) w_{k\hat{j}}$$

Note that, for simplicity, we are considering a Neural Network with just one hidden layer, but the derivation can be extended for more than one hidden layer.

7.3.1 Back-propagation algorithm

Back-propagation is an algorithm that aims to update the weights of a Neural Network in order to minimize the error between the predicted output and the true output. The algorithm works by propagating the error back through the network, starting with the output layer and moving backwards through the hidden layers, adjusting the weights at each layer to reduce the error. It can be implemented in different ways:

- **Batch gradient descent:** It **cumulates** gradients over all the training examples and then updates the weights.
- **Stochastic gradient descent:** For each example in S, it computes the gradients and update the weights.
- **Mini-batch gradient descent:** It updates the weights considering a subset of examples $Q \subseteq S$.

This process is typically repeated many times until the error is sufficiently small or the maximum number of iterations is reached. Let's define the algorithm according to the derivations seen before:

Back-propagation (stochastic)

1. Initialize all weights with small random values (e.g. between -0.5 and +0.5)

```
2. Until the termination condition is satisfied:
    (a) For each $(\mathbf{x}, \mathbf{t}) \in S$:
        i. Present $\mathbf{x}$ to the net and compute the vectors $\mathbf{y}$ and $\mathbf{z}$
        ii. For each output unit $k$:
            $\delta_k = z_k(1 - z_k)(t_k - z_k)$
            $\Delta w_{kj} = \delta_k y_j$
        iii. For each hidden unit $j$:
            $\delta_j = y_j(1 - y_j) \sum_{k=1}^{c} w_{kj}\delta_k$
            $\Delta w_{ji} = \delta_j x_i$
        iv. Update all weights:
            $w_{sq} \leftarrow w_{sq} + \eta \Delta w_{sq}$
```

Note that this algorithm works only for a network with one hidden layer, but can be easily extended. In fact, the algorithm computes the error term δ for each unit of each layer, and then it multiplies this error term for the input of the unit.

7.4 Computational power of neural networks

Theorem (Pinkus, 1996) simplified:
Given a feed-forward NN with just one hiddel layer, any continuous function $f : \mathbb{R}^n \to \mathbb{R}$ and an arbitrarily small $\epsilon > 0$, then, for a large class of activation functions, there always exists an integer M such that the function $g : \mathbb{R}^n \to \mathbb{R}$ computed by the net using at least M hidden units approximates the function f with tolerance ϵ, that is:

$$max_{x\in\Omega}|f(\mathbf{x}) - g(\mathbf{x})| < \epsilon$$

Note that the theorem attests the existence of a NN with M hidden units that approximates the target function with the desired tolerance, but it says nothing about how M can be computed.

7.5 Fine tuning neural networks

The choice of the net typology determines the hypothesis space used. With three-layers architecture (input, hidden, output), the number of hidden units detrmines the complexity of the hypothesis space. As we said for the Perceptron, the choice of the learning rate η can be crucial for the convergence.

7.5.1 Avoiding local minima

The local minima problem in neural networks refers to the scenario where a neural network gets stuck in a sub-optimal solution during the training process. Instead of reaching the global minimum, which is the best possible solution, the network settles for a local minimum, that may not be the best fit for the data. This can happen because the loss function used can have multiple local minima, and the optimization algorithm may only find one of them. In order to avoid this problem, we can use different techniques:

- Adding a term to the weight update rule (called **momentum**). Basically, a contribution deriving from the previous step is added, imposing a kind of inertia on the system.
- Using **stochastic training** (randomizing on the examples) instead of batch training can facilitate to avoid local minima.
- **Training multiple NNs** on the same data with different initialization and combining the outputs.

7.5.2 Over-fitting

With neural networks, usually, the error on the validation set increases as the number of weight updates increases. This is because, at the beginning, with small absolute values of the weights (similar each other), the decision surfaces are *smoother*. When the absolute values of the weights increase , the complexity of the decision surfaces increases and with it the possibility to suffer over-fitting.

A possible solution can be to use an additional term in the error function to minimize based on the norm of the weights (regularization) $E + ||w||^2$.

7.6 Hidden layer representation

An important feature of multi-layer NNs is that they can find useful representations of input data (alternative to the input representation). Specifically, the output of the hidden units is an effective new input representation for the next layers. This feature can be used to define a particular type of NNs called Auto-encoders.

Chapter 8

Lec 10 - Generalized Linear Models and SVM I

8.1 Linear Models

Linear models are one of the most important types of models in machine learning. A linear model is of the form:

$$f_{\mathbf{w},b}(\mathbf{x}) = \sum_{i=1}^{m} w_i x_i + b = \mathbf{w} \cdot \mathbf{x} + b$$

Which can also be written as

$$f_{\mathbf{w},b}(\mathbf{x}) = \sum_{i=0}^{m} w_i x_i = \mathbf{w} \cdot \mathbf{x}$$

where $x_0 = 1$.

- For **classification**, the sign of the result is returned, that is:

$$h(\mathbf{x}) = sign(f_{\mathbf{w}}) \in \{-1, +1\}$$

- For **regression**, the *original* function can be taken, that is:

$$h(\mathbf{x}) = f_{\mathbf{w}} \in \mathbb{R}$$

8.1.1 Linear Regression

Given a training set $S = \{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)\}$, in linear regression we look for a hypothesis $h_{\mathbf{w}}$ which minimizes the mean squared error on the training set, that is:

$$argmin_{\mathbf{w}} \frac{1}{n} \sum_{i=0}^{n} (h_{\mathbf{w}}(\mathbf{x}_i) - y_i)^2$$

We can formalize it in matrices terms:

$$E(\mathbf{w}) = \frac{1}{n} \sum_{i=0}^{n} (h_{\mathbf{w}}(\mathbf{x}_i) - y_i)^2$$

$$= \frac{1}{n} \sum_{i=0}^{n} (\mathbf{w} \cdot \mathbf{x}_i - y_i)^2$$

$$= \frac{1}{n}||\mathbf{X}\mathbf{w} - \mathbf{y}||$$

where:

- $\mathbf{X}$ is a matrix of size $n \times d$
- $\mathbf{w}$ is a vector of size $d \times 1$
- $\mathbf{y}$ is a vector of size $n \times 1$

d is the dimension of the feature space (including the bias term). Basically, we want that $\mathbf{Xw} \approx \mathbf{y}$.

Since $||\vec{z}||^2 = \sum_i z_i^2$, the optimization problem can be defined as follows:

$$min_{\mathbf{w}} E(\mathbf{w}) \equiv \frac{1}{n}||\mathbf{X}\mathbf{w} - \mathbf{y}||^2$$

We want to find the vector $\mathbf{w}$ that minimizes E. Basically, we want to find the points $\mathbf{w}$ where the gradient vanishes (equals to 0), that is:

$$\nabla E(\mathbf{w}) = \frac{2}{n}\mathbf{X}^T(\mathbf{X}\mathbf{w} - \mathbf{y}) = 0$$

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

By multiplying the quantity $(\mathbf{X}^T\mathbf{X})^{-1}$ both on the left and on the right of the equation, it becomes:

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

Since $(\mathbf{X}^T\mathbf{X})^{-1}(\mathbf{X}^T\mathbf{X})$ is the identity matrix, the final equation is:

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

This is a **closed form solution**. It means that with just one computation we are able to find the solution, it's not am iterative process. However, computing the inverse of a matrix can be very computational inefficient.

8.1.2 Non-linear mapping

The goal of non-linear mapping is to apply a transformation to non-linearly separable points in order to map them in a new feature space that is linearly separable. Then, a linear model (hyperplane) in the transformed space will correspond to a non-linear decision surface in the original space (Generalized Linear Models).

8.2 Linear separability

Consider the hypothesis space of hyperplanes. Given a set of linearly separable points, we have different hyperplanes that separates them. Intuitively, the widest possible margin (or **optimal**) hyperplane is the best one, because is the one that generalizes better, but how can we compute to which $\mathbf{w}$, b the best hyperplane corresponds.

8.2.1 Margin of a hyperplane

The margin of a hyperplane is the distance between the hyperplane and the closest data points of each class.

Let's consider the **binary classification** problem. Given the hyperplane $\mathbf{w} \cdot \mathbf{x} + b = 0$ (**optimal hyperplane**), the *"distance"* of a point $\mathbf{x}$ from the hyperplane can be expressed by the algebraic measure $g(\mathbf{x}) = \mathbf{w}^T \cdot \mathbf{x} + b$. We can write $\mathbf{x}$ as follows:

$$\mathbf{x} = \mathbf{x}_p + r\frac{\mathbf{w}}{||\mathbf{w}||}$$

where:

- $||\mathbf{w}||$ is the norm of $\mathbf{w}$
- $\mathbf{x}_p$ is the normal projection of $\mathbf{x}$ onto the hyperplane.
- r is the desired algebraic distance ($r > 0$ if $\mathbf{x}$ is on the positive side of the hyperplane, otherwise $r < 0$).

Note that $g(\mathbf{x}_p) = 0$ because $\mathbf{x}_p$ is on the hyperplane. Furthermore, since we are considering the optimal hyperplane, the absolute distance between this hyperplane and any nearest positive example is the same as the absolute distance between any negative example. So, we can *normalize* the value of g such that $g(\mathbf{x}) = 1$ if $\mathbf{x}$ is in the positive margin hyperplane and $g(\mathbf{x}) = -1$ if $\mathbf{x}$ is in the negative margin hyperplane.

Take $\mathbf{x}_k$ in the positive margin hyperplane, it follows that

$$g(\mathbf{x}_k) = \mathbf{w}^T\mathbf{x}_k + b$$

Since $\mathbf{x}_k = \mathbf{x}_p + r\frac{\mathbf{w}}{||\mathbf{w}||}$:

$$= \mathbf{w}^T\left(\mathbf{x}_p + r\frac{\mathbf{w}}{||\mathbf{w}||}\right) + b$$

$$= \mathbf{w}^T\mathbf{x}_p + b + r\frac{\mathbf{w}^T\mathbf{w}}{||\mathbf{w}||}$$

Since $\mathbf{w}^T\mathbf{w} = ||\mathbf{w}||^2$:

$$= \mathbf{w}^T\mathbf{x}_p + b + r||\mathbf{w}||$$

The term $\mathbf{w}^T\mathbf{x}_p + b = 0$ because, as we said before, $g(\mathbf{x}_p) = 0$. This implies that:

$$g(\mathbf{x}_k) = r||\mathbf{w}|| = +1$$

This is true only if

$$r = \frac{1}{||\mathbf{w}||}$$

Then, the margin ρ is defined as follows:

$$\rho = 2r = \frac{2}{||\mathbf{w}||}$$

8.3 Support Vector Machines (SVM)

Support Vector Machines (SVMs) are a set of supervised learning methods used for classification and regression tasks. The idea behind SVMs is to find the best hyperplane that separates the training data. This hyperplane is chosen in such a way that it maximizes the margin.

8.3.1 Separable case: Quadratic optimization

If we have n linearly separable examples $\{(\mathbf{x}_i, y_i)\}$ (binary classification), it is possible to find the optimal hyperplane $h(\mathbf{x}) = sign(\mathbf{w} \cdot \mathbf{x} + b)$ by solving the following **constrained** quadratic optimization problem:

$$min_{\mathbf{w},b}\frac{1}{2}||\mathbf{w}||^2$$

$$\textit{subject to}: \ \forall i \in \{1, ..., n\} : y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1$$

Since the margin is defined as $\rho = \frac{2}{||\mathbf{w}||}$ and we want to maximize it, we need to minimize $||\mathbf{w}||$. This minimization problem is subject to the constraint that each point in the training set must be on the *correct* side of the hyperplane, which means that for each point, the following inequality must hold: $\forall i \in \{1, ..., n\}$: $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1$. This constraint is given by the fact that:

- $\mathbf{w} \cdot \mathbf{x}_i + b \geq 1$ if $y_i = +1$
- $\mathbf{w} \cdot \mathbf{x}_i + b \leq 1$ if $y_i = -1$

This is a **convex** constrained quadratic problem, and for this reason it guarantees a unique solution.

With a large number of features it can be computationally inefficient to solve this problem, so we can use its **dual formulation**

8.3.2 Dual formulation

In the dual problem, Lagrange multipliers $\alpha_i \geq 0$ are associated with every constraint in the primal problem (one for each example).

$$max_\alpha \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} y_i y_j \alpha_i \alpha_j (\mathbf{x}_i \cdot \mathbf{x}_j)$$

$$subject\ to: \ \forall i \in \{1, ..., n\}: \ \alpha_i \geq 0 \ and \ \sum_{i=1}^{n} y_i \alpha_i = 0$$

At the solution, most of the α_i's are zeros. Those examples associated with non zero multipliers are called **support vectors**.

8.3.3 SVM solution

The primal solution turns out to be:

$$\mathbf{w} = \sum_{i=1}^{n} y_i \alpha_i \mathbf{x}_i$$

$$b = y_k - \mathbf{w} \cdot \mathbf{x}_k \ \forall \mathbf{x}_k \ s.t \ \alpha_k > 0$$

Hence:

$$h(\mathbf{x}) = sign(\mathbf{w} \cdot \mathbf{x} + b) = sign\left(\sum_{i=1 \in support\ vector}^{n} y_i \alpha_i (\mathbf{x}_i \cdot \mathbf{x}) + b \right)$$

The decision function only depends on dot products between the point and othe points in the training set (the support vectors).

Chapter 9

Lec 11 - Generalized Linear Models and SVM II

9.1 SVM for non-separable case

Let's extend the SVM model for non-separable data. With this approach we try to find a balance between maximizing the margin of the hyperplane and minimizing the number of misclassifications.

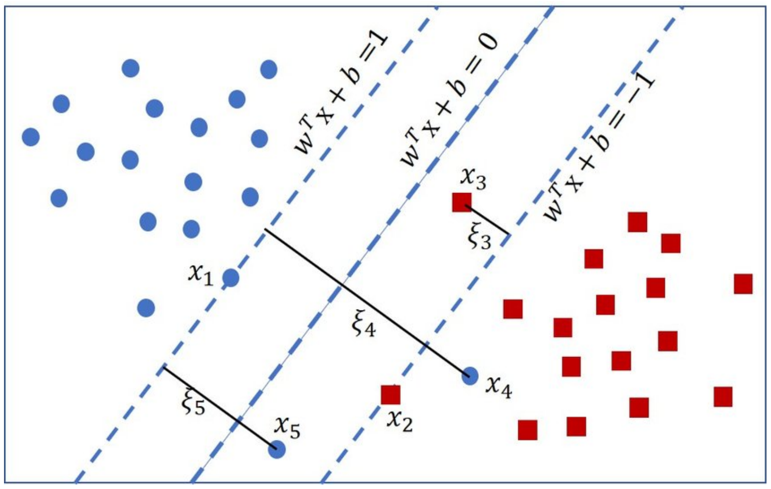
In the *standard* SVM model, the solution is subject to the constraint that each point in the training set must be on the *correct* side of the hyperplane:

$$\forall i \in \{1, ..., n\} : y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1$$

If the examples are not linearly separable we have to allow that some constraints are violated. This can be done by introducing *slack* variables $\xi_i \geq 0$, $i = 1, ..., n$, one for each constraint:

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 - \xi_i$$

Basically, we replace the hard constraint with a softer one. These slack variables measure the distance between the point and the margin hyperplane where that point was supposed to be.



We need to modify the cost function so to penalize slack variables which are not 0. Then, the **minimization** problem becomes the following:

$$\frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n}\xi_i$$

where C (regularization hyper-parameter) is a positive constant controlling the trade-off between the complexity of the hypothesis space and the number of margin errors.

The dual formulation is very similar to the one defined for separable data:

$$max_\alpha \sum_{i=1}^{n}\alpha_i - \frac{1}{2}\sum_{i,j=1}^{n} y_i y_j \alpha_i \alpha_j (\mathbf{x}_i \cdot \mathbf{x}_j)$$

$$subject\ to:\ \forall i \in \{2, ..., n\} : 0 \le \alpha_i \le C,\ and \quad \sum_{i=1}^{n} y_i \alpha_i = 0$$

The main difference between the two is due to the fact that the dual variables are upper bounded by C. The value of b is obtained similarly to the separable case.

Note that a smaller C value results in a larger margin but allows for more misclassifications, while a larger C value results in a smaller margin but fewer misclassifications. This parameter can be seen as a way to control over-fitting, because increasing C means to minimize the training error.

Nevertheless, this formulation is not always satisfactory because of the limited separation capability of a hyperplane (we are not able to perform non-liner separation of training data).

9.2 Non-separable case: Another approach

When the examples are not linearly separable the idea is to use a non-linear transformation in order to project all the points into a higher dimensional space.

$$\mathbf{x} \rightarrow \phi(\mathbf{x})$$

This approach is based on the following two steps:

1. The input vectors (**input space**) are projected into a larger space (**feature space**). This is justified by Cover's theorem on separability, which states that non-linearly separable patterns may be transformed into a new feature space where the patterns are linearly separable with high probability, if the transformation is non-linear and if the dimensionality of the feature space is high enough.
2. The optimal hyperplane in the feature space is computed.

The hyperplane in the transformed space will correspond to a non-linear decision surface in the original space.

We can assume that any of the new feature space coordinate is generated by a non-linear function $\phi_i(\cdot)$. Given M functions $\phi_j(\mathbf{x})$ with $j = 1, ..., M$, a generic vector $\mathbf{x}$ is mapped into the following M-dimensional vector:

$$\phi(\mathbf{x}) = [\phi_1(\mathbf{x}), ..., \phi_M(\mathbf{x})]$$

A hyperplane in the feature space is defined as:

$$\sum_{j=1}^{M} w_j \phi_j(\mathbf{x}) + b = 0$$

$$= \sum_{j=0}^{M} w_j \phi_j(\mathbf{x}) = \mathbf{w} \cdot \phi(\mathbf{x}) = 0$$

where $\phi_0(\mathbf{x}) = 1$ and $w_0 = b$

By changing the definition of the dual formulation, it follows that:

$$\mathbf{w} = \sum_{k=1}^{n} y_k \alpha_k \phi(\mathbf{x}_k)$$

The weight vector **w** can be expressed as a linear combination of the n training examples. So, the equation defining the hyperplane becomes:

$$\sum_{k=1}^{n} y_k \alpha_k \phi(\mathbf{x}_k) \cdot \phi(\mathbf{x}) + b = 0$$

where $\phi(\mathbf{x}_k) \cdot \phi(\mathbf{x})$ represents the dot product **in feature space** between vectors induced by the k-th training instance and the input $\mathbf{x}$.

9.3 Kernel functions

A Kernel function K is a non-linear function whose result is equal to the dot-product in the feature space between two given vectors:

$$K(\mathbf{x}_k, \mathbf{x}) = \phi(\mathbf{x}_k) \cdot \phi(\mathbf{x}) = \sum_{j=0}^{M} \phi_j(\mathbf{x}_k)\phi_j(\mathbf{x}) = K(\mathbf{x}, \mathbf{x}_k)$$

Definition:
A kernel function is a function $K(\cdot, \cdot)$ such that for all $\mathbf{x}, \mathbf{z} \in X$, satisfies $K(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x}) \cdot \phi(\mathbf{z})$ where $\phi(\mathbf{x})$ is a mapping from X to a higher dimensional feature space H.

In general, a kernel function represents a dot-product between vectors generated by some non-linear transformation.

Function with this property do actually exists, if some conditions are satisfied (Mercer's conditions). Note that it's not guaranteed that for all non-linear transformations ϕ it exists such a function K.

Examples of popular kernel functions:

- linear kernel, $K(\mathbf{x}, \mathbf{z}) = x \cdot z$
- polynomial kernel of degree p, $K(\mathbf{x}, \mathbf{z}) = (\mathbf{x} \cdot \mathbf{z} + u)^p$, $u > 0$
- Exponential kernel, $K(\mathbf{x}, \mathbf{z}) = exp(\mathbf{x} \cdot \mathbf{z})$
- Radial-basis function (RBF) kernel, $K(\mathbf{x}, \mathbf{z}) = exp(-\gamma||\mathbf{x} - \mathbf{z}||^2),\ \gamma > 0$

if we get such a (symmetric) function, we could compute the decision function on feature space **without** explicitly representing the vectors into the feature space (we do not need to know anything about the non-linear transformation ϕ).

9.4 SVM problem formulation with Kernel

The introduction of a kernel, actually, does not modify the problem formulation:

$$max_\alpha \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i,j}^{n} y_i y_j \alpha_i \alpha_j K(\mathbf{x}_i, \mathbf{x}_j)$$

$$subject\ to:\ \forall i \in \{1, ..., n\} : 0 \leq \alpha_i \leq C\ and\ \sum_{i=1}^{n} y_i \alpha_i = 0$$

Basically, all the occurrences of the dot-product between transformed examples are replaced with the kernel function. The needed kernel values are computed over all pairs of vectors $K(\mathbf{x}_i, \mathbf{x}_j)$, with $i, j = 1, ..., n$ and arranged into a matrix $\mathbf{K} \in \mathbb{R}^{n\times n}$ (where n is the number of examples) known as **kernel matrix** or **gram matrix**.

Definition:
The Gram (or kernel) matrix associated with the kernel function $K(\cdot,\cdot)$, evaluated on a finite subset of examples $X = \{\mathbf{x}_1, ..., \mathbf{x}_n \in X\}$, is the matrix $\mathbf{K} \in \mathbb{R}^{n\times n}$ such that:

$$\mathbf{K}_{i,j} = K(\mathbf{x}_i, \mathbf{x}_j)$$

For example, if we use a polynomial kernel with degree $p = 3$ we obtain that $\mathbf{K}_{i,j} = (\mathbf{x}_i \cdot \mathbf{x}_j + 1)^3$. Therefore, a new instance $\mathbf{x}$ is classified by the following discriminant function:

$$h(\mathbf{x}) = sign(\sum_{\mathbf{x}_k \in SV} y_k \alpha_k^* K(\mathbf{x}_k, \mathbf{x})) = sign(\sum_{\mathbf{x}_k \in SV} y_k \alpha_k^* (\mathbf{x}_k \cdot \mathbf{x} + 1)^3)$$

With this approach, we can use a non-linear transformation $\phi(\cdot)$ **implicitly**, in fact, what we need is not the explicit representation of vectors in feature space, but their dot-product into the feature space. This can be directly computed in the input space using the kernel function.

Note that the formulation with kernel works only with respect to the dual formulation of the SVM problem. This is because in the primal formulation the decision function does not depend only on dot-products between examples, but we have to minimize a vector of weights $\mathbf{w}$.

9.5 Polynomial kernel

Given two vectors $\mathbf{x}$ and $\mathbf{z}$ and the following mapping $\phi()$

$$\mathbf{x} = (x_1, x_2);\ \phi(\mathbf{x}) = (x_1^2, x_2^2, \sqrt{2}x_1x_2)$$

$$\mathbf{z} = (z_1, z_2);\ \phi(\mathbf{z}) = (z_1^2, z_2^2, \sqrt{2}z_1z_2)$$

A dot product between $\phi(\mathbf{x})$ and $\phi(\mathbf{z})$ corresponds to evaluate the function $K_2(\mathbf{x}, \mathbf{z}) = \langle x, z\rangle^2$, which is the polynomial kernel of degree 2:

$$\begin{aligned}\langle\phi(\mathbf{x}), \phi(\mathbf{z})\rangle &= \langle(x_1^2, x_2^2, \sqrt{2}x_1x_2), (z_1^2, z_2^2, \sqrt{2z_1z_2})\rangle \\ &= x_1^2z_1^2 + x_2^2z_2^2 + 2x_1z_1x_2z_2 \\ &= \langle x, z\rangle^2 \\ &= (x_1z_1 + x_2z_2)^2 \\ &= K_2(\mathbf{x}, \mathbf{z})\end{aligned}$$

$K_2()$ is faster to evaluate than the dot-product between transformed vectors $\langle\phi(\mathbf{x}), \phi(\mathbf{z})\rangle$

This demonstration shows that we can see the polynomial kernel as a dot-product in this specific feature space between two vectors.

9.6 Representation with kernels

A kernel function $K : X \times X \rightarrow \mathbb{R}$ can be seen as a (symmetric) function that gives, for a given pair of examples, a real value which represents the similarity between them. For example, let's consider the Radial-basis kernel:

$$K(\mathbf{x}_i, \mathbf{x}_j) = exp(-\gamma||\mathbf{x}_i - \mathbf{x}_j||^2),\ \gamma > 0$$

Note that when $\mathbf{x}_i = \mathbf{x}_j$ $K(\mathbf{x}_i, \mathbf{x}_j)$ is closed to 1, and when $\mathbf{x}_i \neq \mathbf{x}_j$ it tends to 0. So, given an object $S = \{x_1, x_2, ..., x_n\}$, it can be represented by a symmetric matrix $\mathbf{K} = [K(x_i, x_j)]_{i,j} \in \mathbb{R}^{n \times n}$. Basically, we build the kernel matrix $\mathbf{K}$ by computing the kernel function on each possible pair of examples.

The **advantages** of using kernel matrices representation are different:

- It allows you to build a pipeline in which you first compute the kernel matrix, with respect to your data, and then you change the learning algorithm (that takes as input this matrix) according to your needs. Basically, it improves the **modularity** of the system.
- The dimensionality of data depends on the number of objects and not on their vector dimensionality. This is because the kernel matrix is computed considering all the possible pairs of examples.
- Comparison between objects can result computationally simpler than using the explicit object representation.
- The implicit mapping of data into a higher-dimensional feature space can capture complex patterns and non-linear relationships in data.

Disadvantages of kernels are:

- The memory space required to store the kernel matrix. In fact, the dimension of the matrix is squared with respect to the dimension of the training set.
- The time required to evaluate the scoring function is not suitable for real-time applications. This is because the scoring function is computed by evaluating many kernels.

Recently, there have been effort to *linearize* the kernel. This can be done by approximating the kernel using a fine subset of non-linear features:

$$\langle\overline{\phi}(\mathbf{x}), \overline{\phi}(\mathbf{z})\rangle\ \approx \langle\phi(\mathbf{x}), \phi(\mathbf{z})\rangle = K(\mathbf{x}, \mathbf{z})$$

9.6.1 The Kernel Trick

Any algorithm for vectorial data that can be expressed in terms of dot-products, can be implicitly executed in the feature space associated to a kernel by replacing dot-products with kernel evaluations. Two algorithms that can be **kernelized** are the Perceptron algorithm and k-NN.

Moreover, by defining an appropriate kernel, we can apply algorithm for vectorial data (SVM, Perceptron, etc..) to non-vectorial data (strings, trees, sequances, etc..).

For example, in k-NN we need to compute the distances between examples, but if we want to compute the distances in the feature space we can use the kernel trick: Given two objects $x, z \in X$, the squared distance between the two objects in the feature space is computed by:

$$d(x,z)^2 = ||\phi(x) - \phi(z)||^2$$

Computing the squared distance between two points corresponds to compute the squared norm of $\phi(x) - \phi(z)$.

$$= \phi(x) \cdot \phi(x) + \phi(z) \cdot \phi(z) - 2\phi(x) \cdot \phi(z)$$

$$= K(x, x) + K(z, z) - 2K(x, z)$$

That is:

$$d(x, z) = \sqrt{K(x, x) + K(z, z) - 2K(x, z)}$$

Note that the values of $\phi(x), \phi(z)$ are not explicitly used.

9.7 Kernel properties and extensions

Let K_1, K_2 be kernels defined on $X \times X$, $a \in \mathbb{R}^+$, $\phi : X \to \mathbb{R}^N$ with K_3 a kernel over $\mathbb{R}^N \times \mathbb{R}^N$:

- $K_1(\mathbf{x}, \mathbf{z}) + K_2(\mathbf{x}, \mathbf{z})$ is a kernel.
- $aK_1(\mathbf{x}, \mathbf{z})$ is a kernel
- $K_1(\mathbf{x}, \mathbf{z}) \cdot K_2(\mathbf{x}, \mathbf{z})$ is a kernel
- $K_1(\phi(\mathbf{x}), \phi(\mathbf{z}))$ is a kernel

A kernel can be easily normalized such to have the norm of the transformed vector equals to one ($||\phi(\mathbf{x}) = 1||$):

$$\tilde{K} = \frac{K(\mathbf{x}, \mathbf{z})}{\sqrt{K(\mathbf{x}, \mathbf{x})K(\mathbf{z}, \mathbf{z})}}$$

Kernel definition can be extended to other types of inputs:

- Kernel for strings
- Kernel for trees
- Kernel for graphs

Chapter 10

Lec 12 - Preprocessing

10.1 Supervised learning pipeline

The supervised learning pipeline can be summarized as follows:

1. Analysis of the problem (Classification, Regression, ...)
2. Collection, analysis and cleaning data
3. Pre-processing and managing missing values
4. Study of correlation between variables
5. Feature selection/weighting/learning
6. Choice of the predictor and model selection
7. Test

Some of the most common objects that we can find in machine learning are:

- **Vectors** (Set of values)
- **Strings**
- **Sets and Bags**: for example, the set of terms in a document or their frequency
- **Tensors**: for example, images and video
- **Trees and graphs**

The easiest way to represent these objects is to map them in a vectorial representation.

Features values can be divided in two classes: **Categorical** and **Quantitative**:

- Categorical or symbolic features
  - Nominals: They are used for naming or labeling variables, without any quantitative value. There is usually no intrinsic ordering to nominal data.
  - Ordinals: Is a type of categorical data with an order.
- Quantitative and numeric features:
  - Intervals (Enumerables)
  - Ration (Real values)

Let's see how to encode categorical and quantitative variables into a vector.

10.2 Encoding categorical or symbolic variables

One of the most common method to encode a categorical variable into a vector is the **OneHot Encoding**. This technique allows you to represents categorical variables in a vector with as many components as the number of possible values for the variable.

For example, let's say that we want to encode a variable corresponding to the brand of a car. The possible values that an instance can take are: *(Fiat, Toyota, Ford).* The OneHot encoding of an instance having *Toyota* as brand value could be the following: $[0, 1, 0]$. Basically, it's a vector where we set to one the component corresponding to the value of the instance.

10.3 Encoding of continuous variables

In this case, it is more difficult to find a good mapping. Therefore, the features are typically transformed to obtain values *comparable* with other features.

Given a matrix of instances (training set), for each feature j, let $\hat{x}_j = \frac{1}{n}\sum_{i=1}^{n} x_{ij}$ be the average value of the current feature and let $\sigma_j = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_{ij} - \hat{x}_j)^2}$ be the standard deviation of j. We can apply the following transformations:

- Centering: $c(x_{ij}) = x_{ij} - \hat{x}_j$. For each value x_{ij} we compute $c(x_{ij})$ which is equal to the difference between the value and the average of the corresponding feature.
- Standardization: $s(x_{ij}) = \frac{c(x_{ij})}{\sigma_j}$. We want the *columns* (features) of the matrix to have mean equals to 0 and variance equals to 1.
- Scaling to a range: $h(x_{ij}) = \frac{x_{ij} - x_{minj}}{x_{maxj} - x_{minj}}$. Each value is scaled in a range between 0 and 1.
- Normalization: $g(\mathbf{x}) = \frac{\mathbf{x}}{||\mathbf{x}||}$. In this case, we don't work over features but we want to normalize the instances (rows).

The python module *sklearn* provides several methods to perform this kind of pre-processing.

10.3.1 K-Nearest Neighbors

K-Nearest Neighbors is a simple classification algorithm where a test example is classified with the majority class of its k-neighbors in the training set.

When the instances are **normalized** we can define the squared distance between them in terms of dot-product:

$$||\mathbf{x} - \mathbf{z}||^2$$

$$= ||\mathbf{x}||^2 + ||\mathbf{z}||^2 - 2\langle \mathbf{x}, \mathbf{z} \rangle$$

Since the norm of the instances is equal to 1, it follows that:

$$= 2 - 2\langle \mathbf{x}, \mathbf{z} \rangle$$

So, with normalized data, the distance between two instances is inversely proportional to their dot-product.

10.4 Feature selection and feature extraction

Feature **selection** is the reduction of the dimensionality of the features obtained by removing irrelevant or redundant features. The interpretability of the model is maintained.

Feature **extraction** is the reduction of the dimensionality of the features obtained by combining the original features (e.g. PCA). Generally, the interpretability of the model is lost.

10.4.1 Feature selection

The top reasons to use feature selection are:

- It enables the machine learning algorithm to train faster
- It reduces the complexity of a model and makes it easier to interpret.
- It improves the accuracy of a model if the right subset is chosen.
- It reduces over-fitting.

Feature selection methods can be divided in 3 classes:

- **Filter methods:** They perform feature selection before applying the predictor. They use an efficient scoring function (e.g. Mutual information, Chi squared, Information gain) that determines the usefulness of a given set of features.
- **Wrapped methods:** The predictor is evaluated on a hold-out sample using subset of different features. Then, the subset with highest score is chosen.
- **Embedded methods:** The selection of features occurs in conjunction with the training of the model, for example, by modifying the objective function to be optimized.

10.4.2 Feature extraction (PCA)

Principal Component Analysis (PCA) converts a set of instances with possibly related features into corresponding values on another set of linearly unrelated features (principal components).

Neural Networks can also be seen as a particular way to perform feature extraction on their hidden layers. In fact, the output values of one of the hidden layers can be considered as a new representation of the original data.

Chapter 11

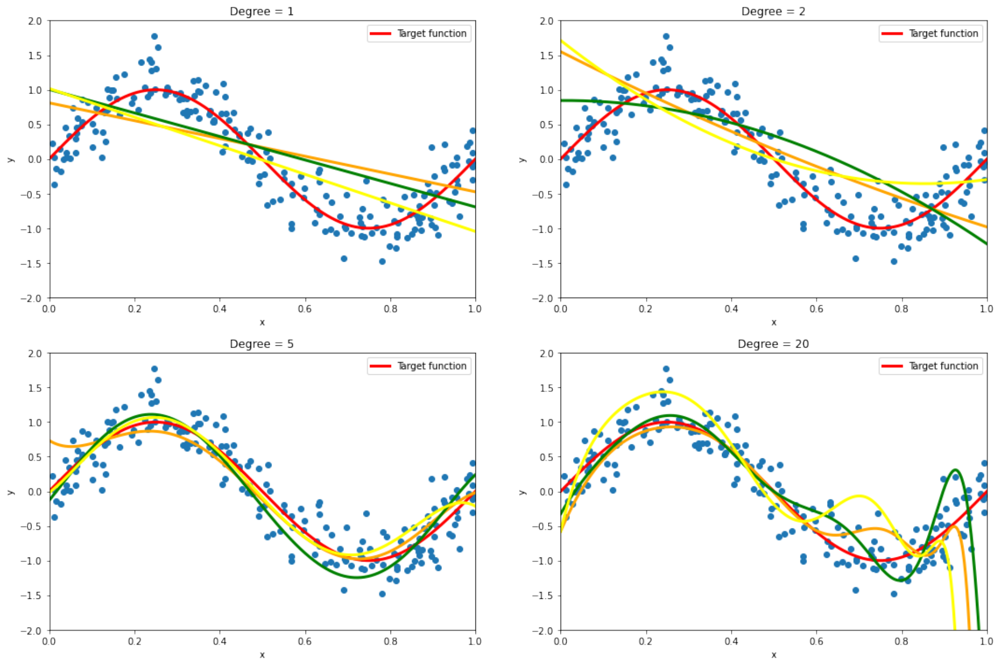
Lec 13 - Model Selection

11.1 Underfitting/Overfitting and learning parameters

Suppose to have some data that we want to fit a curve to (regression problem). Let fit a polynomial of the form:

$$y = w_0 + w_1 x + w_2 x^2 + ... + w_p x^p$$

How can we choose the *best* degree p?



We can see that, by fitting different polynomial predictors (for different values of p), the complexity of the curve increases as the polynomial degree increases. For what concerning the training data, the **training** error decreases as p increases, but when p is too high the resulting curves cannot generalize well (**overfitting**). On the other hand, too low values of the polynomial degree can make the predictor to miss relevant relations between features (**underfitting**).

In order to understand more formally the concepts of overfitting and underfitting we can resort to the notions of **Bias** and **Variance**.

- The Bias measures the *distortion* of an estimate
- The Variance measures the *dispersion* of an estimate

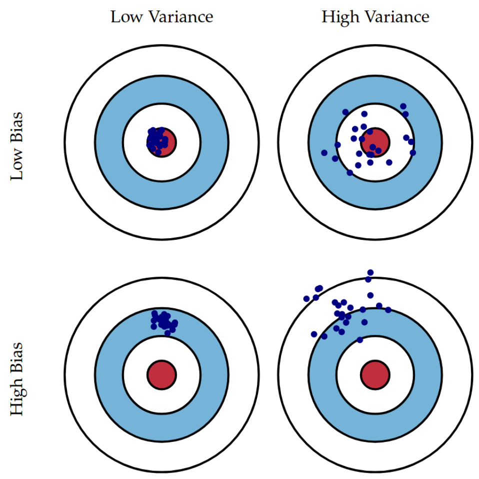


Fig. 1 Graphical illustration of bias and variance.

11.2 Bias-Variance Decomposition for Regression

We can decompose the squared error of a given model in 3 terms:

- Bias error term
- Variance error term
- irreducible error term

Let $y = f(\mathbf{x}) + \epsilon$ be the target function, where ϵ is the error term *selected* from a Gaussian distribution with zero mean and variance σ^2. We want to find a function $\hat{f}(\mathbf{x})$ that approximates the target function $f(\mathbf{x})$ as well as possible.

Given any pair $(\mathbf{x}, y)$, the following holds:

$$E[(y - \hat{f}(\mathbf{x}))^2] = (Bias[\hat{f}(\mathbf{x})])^2 + Var[\hat{f}(\mathbf{x})] + \sigma^2$$

The formula says that the expected value of the squared error between the real output and the predicted output is equal to the sum of 3 terms:

- The bias of the predictor $\hat{f}$ squared: $Bias[\hat{f}(\mathbf{x})] = E[\hat{f}(\mathbf{x})] - f(\mathbf{x})$. It is the difference between the **expected** value of the predictor on $\mathbf{x}$ and the true output $f(\mathbf{x})$.
- The variance of the predictor $\hat{f}$: $Var[\hat{f}(\mathbf{x})] = E[\hat{f}(\mathbf{x}) - E[\hat{f}(\mathbf{x})]]^2$.
- The irreducible error (variance of the noise ϵ)

To obtain $E[\hat{f}(\mathbf{x})]$ (the expected value of $\hat{f}$), for each $\mathbf{x}$, we can train n predictors $\hat{f}$ (over different small training examples) and compute the average of the predictions made by each model.

Note that the bias term will be higher if the expected value of the predictor is very different from the real output given by the target function. Furthermore, the variance error will be higher if the outputs of the different models used to compute $E[\hat{f}(\mathbf{x})]$ are different from each other.

The learning goal is to find the best trade-off between bias and variance:

- The bias error is produced by weak assumptions in the learning algorithm. For example, for very low p the model is too simple and cannot capture the full complexity of the data (underfitting).
- The variance error is produced by an over-sensitivity to small fluctuations in the training set. High variance can cause an algorithm to model the random noise in the training data, rather than the intended outputs (overfitting).

This estimation can be done only if we fix the target function, but in real applications this is not the case, because it is unknown. So, Bias-variance decomposition is a mathematical method for understanding and analyzing the errors made by a machine learning model, but it cannot be used to set the right value of the hyper-parameters.

11.3 Model selection and Hold-out

Let's see some common ways to find the optimal hyper-parameters values using only the training data without resorting to any statistical formula.

The first method is the **Hold-out procedure**: The idea is to obtain a validation set (or hold-out set) Va by splitting the training set Tr. Then, the fixed model is trained using examples in $Tr - Va$, trying different values of the hyper-parameters, and tested against the validation set. This procedure allows you to get an estimate of the error of the model on new unseen data.

In an operational setting, after the parameter optimization, the model is typically re-trained on the entire training set in order to boost effectiveness.

11.3.1 K-fold Cross Validation

An alternative approach for model selection (and evaluation) is the K-fold cross-validation:

K-fold CV procedure:

1. The training set is partitioned in k disjoint sets $Va_1, ..., Va_k$ and K different predictors $h_1, h_2, ..., h_k$ are trained by iteratively applying the Hold-out approach on the k-pairs $(Tr_i = Tr - Va_i, Va_i)$.
2. Final error is obtained by individually computing the errors of $h_1, ..., h_k$ on the corresponding validation set and averaging the results.

The above procedure is repeated for different values of the hyper-parameters and the predictor with the smallest final error is selected.

The special case where $k = |Tr|$ (the validation sets are made of only one example) is called **leave-one-out** cross-validation.

- For higher values of k, we have larger training sets, hence less bias, but smaller validation sets, hence more variance.
- For lower values of k, we have smaller training sets, hence more bias, but larger validation sets, hence less variance.

11.4 Evaluation of unbalanced data - Beyond accuracy

A common measure in machine learning to evaluate the performances of a model for what concerning the classification is the accuracy. Basically, it is the proportion of correct decisions. However, it is not appropriate when we have unbalanced data (e.g. the number of *positive* examples is much lower than the number of *negative* examples, or viceversa). This is because if we have a training set made of few positive examples (+) and a lot of negative examples (-) and we evaluate a model that always predicts (-), its accuracy will be very high even if it is a trivial predictor. So, we need to find other evaluation measures that take into account the unbalancing between data.

Before introducing these measures, let's define the contingency table:

	Relevant	Not relevant
Retrieved	True Positive (TP)	False Positive (FP)
Not Retrieved	False Negative (FN)	True Negative (TN)

The accuracy α is defined as follows:

$$\alpha = \frac{TP + TN}{TP + TN + FP + FN}$$

11.4.1 Precision and Recall

If relevance is assumed to be **binary-valued**, effectiveness is typically measured as a combination of:

- **Precision**:
$$\pi = \frac{TP}{TP + FP}$$
is the *degree of soundness* of the system. It measures how many of the Retrieved classifications are actually Relevant.

- **Recall:**
$$\rho = \frac{TP}{TP + FN}$$
is the *degree of completeness* of the system. It measures how many of the Relevant classifications are actually Retrieved.

In other words, Precision measures the proportion of positive predictions that are actually correct, while Recall measures the proportion of actual positive instances that were correctly predicted.

The choice between precision and recall often depends on the specific use case and the desired outcome of the model:

- If the cost of false positive predictions is high, it may be desirable to prioritize precision. For example, in a medical diagnosis system, a false positive result could lead to unnecessary treatments, so a high Precision is desired.

- If the cost of false negatives is high, it may be desirable to prioritize recall. For example, in a fraud detection system, a false negative result could result in a missed fraud, so a high Recall is desired.

11.4.2 F-Measure

In some cases, it may be necessary to get a trade-off between Precision and Recall. This can be done using the **F-measure**, which is a weighted harmonic mean of the Precision (π) and Recall (ρ):

$$F_\beta = \frac{(1+\beta^2)\pi\rho}{\beta^2\pi + \rho}$$

If $\beta < 1$ it emphasizes Precision, while $\beta > 1$ emphasizes Recall. When $\beta = 1$, we obtain the so called F1 score:

$$F_1 = 2\frac{\pi\rho}{\pi + \rho}$$

11.5 Multi-class classification

Multi-class classification consists of a classification task with more than two classes. It makes the assumption that each instance is assigned to only one label; for example, a fruit can either be an apple or an orange, but not both at the same time.

However, some models are defined only for binary problems (e.g. SVM), so how can the multi-class problem be reduced to a set of binary problems ?

- The first strategy is the so called **one-vs-rest** strategy. It consists of fitting one classifier per class. For each binary classifier, its corresponding class is labelled as positive (+) and all the other classes as negative (-).

  The classification of new unseen examples is performed by classifying the example with all the classifier and selecting the one that matches with the highest response.

  One advantage of this approach is its interpretability. Since each class is represented by only one classifier, it is possible to gain knowledge about the class by inspecting its corresponding classifier. This is the most common strategy and is a fair default choice.

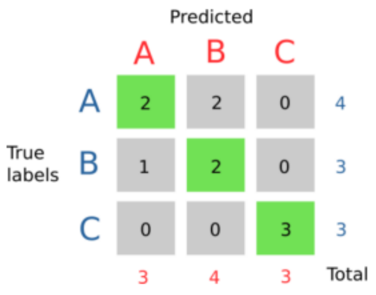
- Another strategy is the **one-vs-one** strategy. It consists in fitting one classifier for each pair of classes. At prediction time, the class that received most votes is selected.

  Since it requires to fit $n\_classes * (n\_classes - 1)/2$ classifiers, this method is usually slower than one-vs-rest. However, this technique may be advantageous for kernel algorithms which don't scale well with $n\_samples$. This is because each individual learning problem only involves a small subset of examples, while with one-vs-rest method the complete dataset is used $n\_classes$ times.

11.6 Evaluation of Multi-class classification

A very intuitive way to show the results of a multi-class predictor is using the **confusion matrix**.

```
Prediction :  A B A B C C C A B
True labels:  A A A A C C C B B
```



		Predicted			
		A	B	C	
True labels	A	🟩 2	2	0	4
	B	1	🟩 2	0	3
	C	0	0	🟩 3	3
		3	4	3	Total

It is a table with true labels over the rows and predicted labels over columns. Each cell over the diagonal contains the number of times the true label was correctly predicted.

Then, we can compute the Precision and Recall measures for the multi-class problem according to this matrix:

- Precision can be calculated separately for each class. For each column, we take the number on the diagonal and we divide it by the sum of all the numbers in the column.
  - $prec\_A: \ 2/3 = 0.67$
  - $prec\_B: \ 2/4 = 0.50$
  - $prec\_C: \ 3/3 = 1$
- Recall can be computed separately for each class. It is the value on the diagonal divided by the sum of the values on the row.
  - $recall\_A: \ 2/4 = 0.50$
  - $recall\_B: \ 2/3 = 0.67$
  - $recall\_C: \ 3/3 = 1$

To extract a single number from the Precision, Recall or F1-score of the model, it is common to adopt two different kinds of average:

- The first is to compute the score separately for each class and taking the average value (**macro averaging**).
- The second is to compute the measure from the grand total of the numerator and denominator (**micro averaging**). For our Recall example, the micro average is the following:

$$\frac{2+2+3}{4+3+3} = 0.636$$

Chapter 12

Lec 14 - Representation Learning

12.1 Representation Learning

Representation Learning aims at learning representation of input data, typically by transforming it, such to make the input *easier* for a classification or prediction task. The performance of any machine learning model is critically dependent on the representation used. The set of priors that should drive the choice of a representation is the following:

- Smoothness
- Multiple explanatory factors
- A hierarchical organization of explanatory factors
- Sparsity
- ...

12.2 Principal Component Analysis (PCA)

Principal component analysis, or PCA, is a dimensionality-reduction method that is often used to reduce the dimensionality of large data sets, by transforming a large set of variables into a smaller one that still contains most of the information in the large set.

PCA tries to answer the following question: *is there any other basis, which is a linear combination of the original one, that best re-express out data ?* In mathematical terms, PCA tries to re-express the dataset $\mathbf{X} \in \mathbb{R}^{m \times n}$ (**examples on columns**) by a simple linear transformation $\mathbf{P} \in \mathbb{R}^{m \times m}$:

$$\mathbf{PX} = \hat{\mathbf{X}}$$

Basically, we want to find the best transformation $\mathbf{P}$ following some criteria.

The equation above is actually a change of basis that can be interpreted in different ways:

- $\mathbf{P}$ is a matrix that transforms $\mathbf{X}$ into $\hat{\mathbf{X}}$.
- Geometrically, $\mathbf{P}$ transforms $\mathbf{X}$ into $\hat{\mathbf{X}}$ via a rotation and a stretching.
- The rows of $\mathbf{P}$ are a new set of basis vectors.

Two phenomena can potentially *contaminate* our data: **noise** and **redundancy**. The linear transformation we aim to find is the one that minimizes both the noise and the redundancy of the signal.

Both noise and redundancy can be estimated using measures related to the variance of the data:

- The noise of a signal can be measured by the *signal-to-noise* ratio (SNR):

$$SNR = \frac{\sigma_{signal}}{\sigma_{noise}}$$

- Redundancy between features can be measured through their **covariance**, that is, how much are the features dependent by each other.

Given a dataset $\mathbf{X} \in \mathbb{R}^{m \times n}$ (centered with mean 0) the **covariance matrix $\mathbf{S_X}$** $\in \mathbb{R}^{m \times m}$ is computed as follows:

$$\mathbf{S_X} = \frac{1}{n}\mathbf{X}\mathbf{X}^T$$

This matrix has as entries the covariances associated with all possible pairs of the initial features. Since the covariance of a variable with itself is its variance ($Cov(\mathbf{X}_i, \mathbf{X}_i) = Var(\mathbf{X}_i)$), in the main diagonal we actually have the variances of each initial feature. Furthermore, the covariance is commutative ($Cov(\mathbf{X}_i, \mathbf{X}_j) = Cov(\mathbf{X}_j, \mathbf{X}_i)$), so the entries of the covariance matrix are symmetric with respect to the main diagonal.

In order to reduce the features covariance, we want to have all the entries $\mathbf{S_X}(i, j)$ equal to 0. So, the idea is to transform our data in a new representation $\hat{\mathbf{X}}$ such that if we compute the covariance matrix of $\hat{\mathbf{X}}$, we obtain a **diagonal matrix**.

In mathematical terms, the goal of PCA is to find some orthonormal matrix $\mathbf{P}$ where $\hat{\mathbf{X}} = \mathbf{PX}$ such that

$$\mathbf{S}_{\hat{\mathbf{X}}} = \frac{1}{n}\hat{\mathbf{X}}\hat{\mathbf{X}}^T$$

is a diagonal matrix.
The rows of $\mathbf{P}$ are called **principal components**.

The PCA algorithm steps are:
Given a dataset $\mathbf{X} \in \mathbb{R}^{m \times n}$ (examples on the columns).

1. Standardize the data. Center $\mathbf{X}$ by subtracting off the mean of each measurement type.
2. Compute the covariance matrix.
3. Compute the **eigenvalues** ($\mathbf{D} \in \mathbb{R}^{m \times m}$) and **eigenvectors** ($\mathbf{E} \in \mathbb{R}^{m \times m}$) of the covariance matrix and set $\mathbf{P} = \mathbf{E}^T$.
4. Select the $k << m$ most important component from $\mathbf{P}$.

Note that in step 4 is performed the dimensionality reduction. In fact, often the most interesting dynamics occur only in the first k dimensions. Throwing out the less important axes can help to reveal hidden and simplified dynamics in high dimensional data.

Formally, given $k << m$, the dimensionality of $\mathbf{X}$ is reduced by:

$$\mathbf{P}_k\mathbf{X} = \mathbf{X}_k \in \mathbb{R}^{k \times n}$$

where $\mathbf{P}_k$ is a sub-matrix of $\mathbf{P}$ containing the k most important component.

Interestingly, $\mathbf{P}_k$ is also the minimizer of the squared **reconstruction error**:

$$min_{\mathbf{P}\in\mathbb{R}^{k\times n}}||\mathbf{X}-\mathbf{P}^T\mathbf{P}\mathbf{X}||_F^2 \quad s.t.\ \mathbf{P}\mathbf{P}^T=\mathbf{I}_{k\times k}$$

It represents how much is the loss in the transformation. This is because we obtain $\hat{\mathbf{X}}$ by applying $\mathbf{P}_k$ to $\mathbf{X}$. So, applying $\mathbf{P}_k^T$ to $\hat{\mathbf{X}}$ is like *going back* to $\mathbf{X}$. However, if $k << m$ i don't obtain $\mathbf{X}$ but something similar.

12.2.1 Kernel PCA

Kernel PCA extends the conventional PCA to deal with non-linear correlations using the **kernel trick**. A high level algorithm can be the following:

1. Instead of using data points $\mathbf{X}$ directly, we first map them to some feature space (implicitly).

2. Center the data in the feature space.

3. Implicitly extract principal components in the feature space, that is, apply PCA in the feature space. Projections of these principal components can be computed in terms of kernels only.

4. The result will be non-linear transformations in the original input space.

12.2.2 PCA - Applications

Some of the possible scenarios in which PCA can be useful:

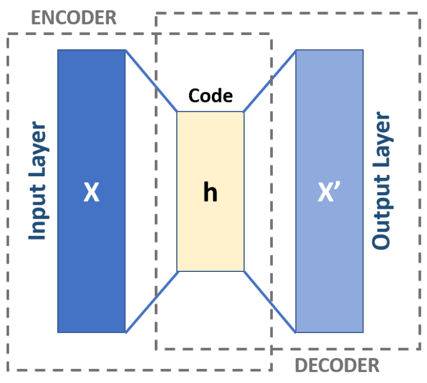
- Apply *lossy compression.*
- Visualize the multi-dimensional data in 2D or 3D.
- Reduce the number of dimensions or discard noisy features.
- Perform a change of representation in order to make analysis of data at hand.

12.3 Autoencoders

Autoencoders (AEs) are an **unsupervised learning** technique based on feed forward neural networks. The aim of autoencoders is to learn a representation (often called *encoding* or *code*) for a set of data, typically for the purpose of dimensionality reduction. The network is composed by the following elements:

- **Encoder:** Creates a new representation (the code) of the input
- **Decoder:** Reconstructs the input starting from the code
- **Bottleneck layer:** Hidden layer with a smaller dimensionality than the input layer (it makes the task harder).

The goal of an autoencoder network is to *copy* the input on its output. Basically, when we present an input example to the network, we want to obtain the same values in the output layer.



The easiest form of autencoder has:

- A single hidden layer
- A set of parameters $\mathbf{W}$, $\mathbf{W}^{'}$ defined by two weight matrices and two bias vectors:

$$\mathbf{h} = f_{\mathbf{W}}(\mathbf{x}) = s_1(\mathbf{W}\mathbf{x} + \mathbf{b})$$

$$\hat{\mathbf{x}} = g_{\mathbf{W}'}(\mathbf{h}) = s_2(\mathbf{W}^{'}\mathbf{h} + \mathbf{b})$$

  where s_1 and s_2 denote the activation functions (for the hidden layer and the output layer respectively), which are usually non-linear.

The goal is to find the parameters $\mathbf{W}$ and $\mathbf{W}^{'}$ such that the output of the network approximates the input:

$$g_{\mathbf{W}'}(f_{\mathbf{W}}(\mathbf{x})) \approx \mathbf{x}$$

The output of the hidden layer is a **new lower-dimensional representation** of the input.

The standard loss function is a mean squared error loss (called **reconstruction loss**):

$$L(\mathbf{x}, \tilde{\mathbf{x}}) = ||\mathbf{x} - \tilde{\mathbf{x}}||_2^2$$

where $\tilde{\mathbf{x}} = g_{\mathbf{W}'}(f_{\mathbf{W}}(\mathbf{x}))$. The final objective function that autoencoders aim to optimize is:

$$J(\mathbf{W}, \mathbf{W}^{'}; \mathbf{X}) = \sum_{\mathbf{x} \in \mathbf{X}} L(\mathbf{x}, g_{\mathbf{W}'}(f_{\mathbf{W}}(\mathbf{x})))$$

The learning procedure is performed using standard techniques for training neural networks, such as Back-propagation and Stochastic Gradient Descent.

A simple extension of autoencoders are Deep (non-linear) autoencoders:

- **Deep AEs** add more hidden layers between the input and the code (as well as between the code and the output)
- **Non-linear AEs** set non-linear activation functions to the neurons.

12.3.1 Regularized autoencoders

Encodings produced by basic AEs do not generally present any special property. Additionally, like many other machine learning techniques, AEs are susceptible to overfitting. One possible technique to overcome these problems is to regularize the autoencoders.

Regularized autoencoders use a loss function that encourages the model to have other properties besides the ability to copy its input to its output:

- **Sparsity** of the representation (most values for a given sample are zero or closed to zero)
- **Robustness** to noise or to missing inputs
- **Smallness** of the derivative of the representation

The general form of a regularized AE objective function is:

$$J(\Theta, \Theta^{'}; \mathbf{X}) = \sum_{\mathbf{x} \in \mathbf{X}} L(\mathbf{x}, g_{\Theta'}(f_{\Theta}(\mathbf{x}))) + \Omega(\Theta, \Theta^{'}; \mathbf{X})$$

Basically, we want to find a trade-off between optimizing the reconstruction of the input and regularize the parameters.

12.3.2 Denoising autoencoders

The goal of Denoising autoencoders (DAE) is to clean the corrupted input. They have the same architecture of *standard* autoencoders, but they are trained in a different way. The training process of a DAE works as follows:

- The input $\mathbf{x}$ is corrupted into $\mathbf{x}^{'}$ through a stochastic mapping: $\mathbf{x}^{'} \sim q(\mathbf{x}^{'}|\mathbf{x})$.
- $\mathbf{x}^{'}$ is then mapped to a hidden representation $\mathbf{h} = f_{\Theta}(\mathbf{x}^{'})$.
- Finally, the model reconstructs the input.

12.4 Convolutional Neural Networks (CNN)

Convolutional Neural Networks are a specialized kind of NN for processing data that has a **grid-like topology** (e.g. images). The name CNN indicates that the network employs a mathematical operation called **convolution**.

CNN learns different levels of abstraction of the input, e.g. for images:

- in the first few hidden layers, the CNN usually detects general pattern, like edges.
- the deeper we go into the CNN, these learned abstractions become more specific, like textures, patterns and parts of objects.

12.4.1 Convolution

Let's say that we have an image as input. Convolution is a mathematical operation that works by moving a sliding window (kernel) through the image, pixel by pixel. The window contains coefficients which characterize the transformation. At each position, the result of the convolution is calculated by combining the values of the image subtended to the window with the coefficients of the window itself. In order to compute the new value of the central pixel, the coefficients are:

1. multiplied by the values of the original image subtended to the window

2. added

Borders:
Given an $n \times n$ kernel, its external row/column coincides with the border of the image when the center of this mask is at distance $\frac{n-1}{2}$ from the edge. If you move further out, part of the window *leaves* the image. This situation can be managed in three different ways:

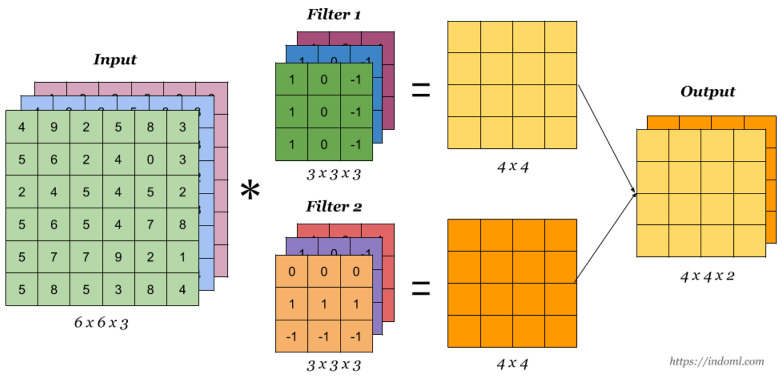
- limit the movement of the mask, keeping it at a minimum distance of $\frac{n-1}{2}$ from the edges.

- duplicate the external rows/columns of the image

- enlarge the image with rows/columns of zeros

Solution 1 gives reliable results, but produces a different size image from the original. Solutions 2 and 3, on the other hand, give results that are not exactly authentic near the edges, but are often convenient because they allow you to obtain an output image with the same size as the input one.

For what concerning convolution of multi-channel images, we perform the convolution for each channel and we sum up the results.



Note that if we apply k filters, the output will be a tensor with $(n \times m \times k)$ dimension.

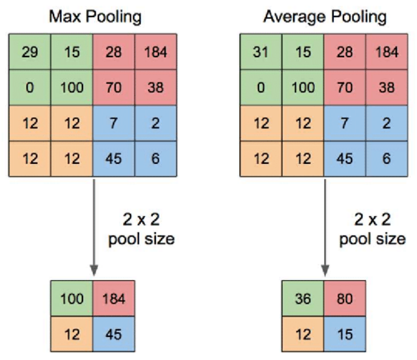
12.4.2 Pooling

Another commonly used technique in CNN is **Pooling**.

Pooling layer is used to reduce the size of the representations and to speed up calculations, as well as to make some features it detects a bit more robust.

- **Max pooling:** It gets the maximum value of the pixels for each section of the image.

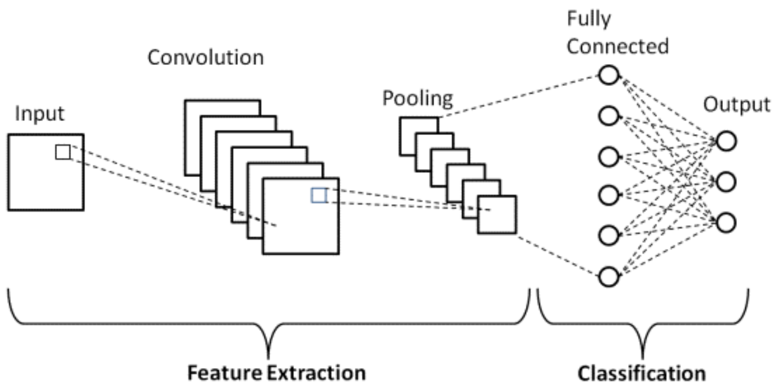
- **Average pooling:** It gets the average value of the pixels for each section of the image.



12.4.3 Convolution layer

The Convolution layer applies a set of filters to the input data (performing convolution) in order to extract features or representations from the data. The result is usually passed through an activation function (ReLu) in order to achieve non linearity.

Basically, a CNN is a sequence of convolution layers and sub-sampling layers with a fully-connected layer at the end.



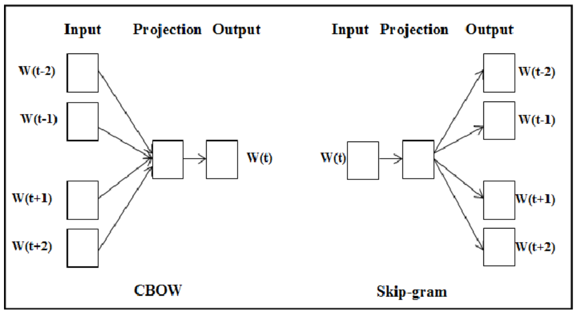
12.5 Word Embedding - Word2Vec

(Neural) **Word embedding** belongs to the text pre-processing phase. The goal is to transform a text (set of words) into a vector of numbers such that similar words produce similar vectors. The word2vec technique is based on a feed-forward, fully connected architecture. It is similar to an autoencoder, but rather than performing input reconstruction, word2vec trains words according to other words that are neighbors in the input corpus (called **context**).

Word2vec aims at computing a **vector** representation of words able to capture semantic and syntactic word similarity.

By using this concept of context, word2vec can learn word embedding in two different ways:

- **CBOW** (Continuous Bag of Words) in which the neural network uses the context to predict a target word.
- **Skip-gram** in which it uses the target word to predict a target context.



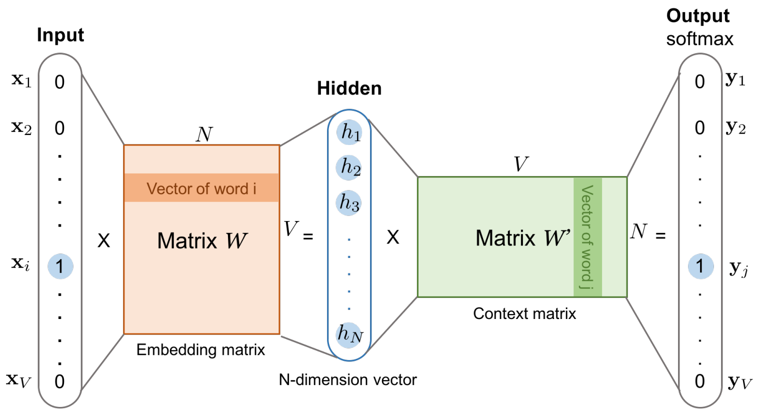
12.5.1 Word2Vec Skip-gram model

The input of the model is the one-hot encoding of the word according to a vocabulary. The output is a distribution over the words of being in the context of the target word. The formal method is defined as follows:

$$\mathbf{h} = \mathbf{W}^T\mathbf{x}$$
$$\mathbf{u} = \mathbf{W}^{'T}\mathbf{h} = \mathbf{W}^{'T}\mathbf{W}^T\mathbf{x}$$
$$\mathbf{y} = \sigma(\mathbf{u})$$

where $\sigma(\cdot)$ is the **softmax** function.

The goal of skip-gram model is to maximize the **average log probability**.



12.5.2 Word2Vec - Semantic and syntactic relations

Word2vec captures different degrees of similarity between words. For example, patterns such as *Man is to Woman as Brother is to Sister* can be generated through algebraic operations on the vectors representations of the words.

$$Brother - Man + Woman \approx Sister$$

12.6 Knowledge graph

A knowledge graph (KG) is a multi-relational graph composed of entities (nodes) and relations (edges). Edges are represented as triples of the form *(head entity, relation, tail entity)* also called **facts**, indicating that two entities are connected by a specific relation.

Formally, given a KG of facts S, a relation is denoted by a triple $(h, r, t) \in S$ where:

- h is the **head** of the relation
- r is the kind of relation
- t is the **tail** of the relation

We assume that $h, t \in \epsilon$ where ϵ is the set of all possible entities and $r \in R$ where R is the set of all possible relations.

12.6.1 Trans-E

The basic idea behind Trans-E is that a relation corresponds to a linear translation in the embedding space. The method assumes that entities can be mapped into a k-dimensional embedding space ($h, t \to \mathbf{h}, \mathbf{t} \in \mathbb{R}^k$) in which when the relation (h, r, t) holds, then $\mathbf{h} + \mathbf{r} \approx \mathbf{t}$. Otherwise, $\mathbf{t}$ should be far away from $\mathbf{h} + \mathbf{r}$.

Chapter 13

Lec 15 - Bayesian Learning I

13.1 Bayesian Methods

Bayesian methods provide computational techniques of learning, but they are also useful for the interpretation/analysis of non-probabilistic algorithms.

These methods refer to set of statistical techniques for building models and making predictions based on the Bayesian framework. The observed training examples increase or decrease the probability that a hypothesis is correct.

The fundamental formula for Bayesian methods is the following:

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

where:

- $P(h)$: a priori probability of the hypothesis h
- $P(D)$: a priori probability of training data. It is the probability to observe exactly this training set when we don't know anything about the hypothesis.
- $P(h|D)$: probability of h given D. It is the probability that h is the hypothesis that generates data D.
- $P(D|h)$: probability if D given h. Given a hypothesis h, it is the probability of data D to be generated by h.

In general, we want to select the most probable hypothesis given the learning data, known as **maximum a posterior hypothesis** h_{MAP} :

$$\begin{aligned} h_{MAP} &= argmax_{h \in H} P(h|D) \\ &= argmax_{h \in H} \frac{P(D|h)P(h)}{P(D)} \end{aligned}$$

Since $P(D)$ does not depend on h, we can consider it as a constant and remove it from the equation.

$$= argmax_{h \in H} P(D|h)P(h)$$

If we assume uniform probabilities on the hypotheses, that is $P(h_i) = P(h_j)$, we can choose the so called **maximum likelihood hypothesis** h_{ML} :

$$h_{ML} = argmax_{h \in H} P(D|h)$$

How can we learn maximum a posterior hypothesis h_{MAP}? A very simple algorithm can be the *brute force* method, in which, for each hypothesis $h \in H$, it computes h_{MAP} and returns the hypothesis with the highest probability. Obviously, this approach cannot be implemented since the hypothesis space contains too many hypothesis (can also be infinite).

13.1.1 Learning of a real-valued function

Consider any real-valued target function f and learning examples $\langle \mathbf{x}_i, d_i \rangle$ where d_i has some noise:

- $d_i = f(\mathbf{x}_i) + e_i$
- e_i is a random variable (noise) extracted independently, for each $\mathbf{x}_i$, according to a Gaussian distribution with mean 0.

It can be shown that the hypothesis h_{ML} (maximum likelihood) is the one that minimizes the mean squared error:

$$h_{ML} = argmin_{h \in H} \sum_{i=0}^{m} (d_i - h(\mathbf{x}_i))^2$$

Basically, the solution of the minimization of the MSE can be motivated by finding the maximum likelihood hypothesis.

13.1.2 Learning a hypothesis that predicts a probability

For what concerning the classification task, the usual scenario is to predict a class according to the input data. How can we learn a hypothesis that predicts a probability ?

Consider the scenario of a probabilistic function $f : X \rightarrow \{0, 1\}$. X might represent medical patients in terms of their symptoms and $f(x)$ might be 1 if the patient survives and 0 if not. We want to learn a predictor (e.g. a neural network) $f^{'} \rightarrow [0, 1]$ which predicts the probability that $f(x) = 1$ given x.

Given the input data $D = \{(\mathbf{x}_1, d_1), ..., (\mathbf{x}_n, d_n)\}$, it can be shown that the criterion we should optimize to find a maximum likelihood hypothesis for $f^{'}$ is the cross-entropy loss function:

$$argmax_{h \in H} \sum_{i=1}^{m} d_i ln(h(\mathbf{x}_i)) + (1 - d_i) ln(1 - h(\mathbf{x}_i))$$

13.2 Most likely classification for new instances

Given a new instance $\mathbf{x}$, which is the most likely **classification**? The classification given by the most likely hypothesis h_{MAP} is not necessarily the most likely classification. For example, given the following three possible hypothesis:

$$P(h_1|D) = 0.4 \quad P(h_2|D) = 0.3 \quad P(h_3|D) = 0.3$$

We want to classify a new instance $\mathbf{x}$:

$$h_1(\mathbf{x}) = + \quad h_2(\mathbf{x}) = - \quad h_1(\mathbf{x}) = -$$

The most likely hypothesis h_1 classifies $\mathbf{x}$ with the label (+), but the most likely classification is (-). This is because the optimal (Bayes) classification of a certain instance is the class $v_j \in V$ which maximizes the following probability:

$$argmax_{v_j \in V} = \sum_{h_i \in H} P(v_j|h_i) P(h_i|D)$$

where V is the set of possible labels.

Computing this optimal classification can be very expensive if there are many hypothesis. A method to overcome this problem can be the Gibbs algorithm:

1. Choose a hypothesis at random, with probability $P(h|D)$
2. Use it to classify the new instance

The surprising fact is that if we assume that the target concepts are randomly extracted from H according to an a priori probability on H, it follows that:

$$E[\epsilon_{Gibbs}] \leq 2E[\epsilon_{Bayes}]$$

By repeating the algorithm multiple times, The error of Gibbs sampling tends to be less than the expected value of bayes error.

Chapter 14

Lec 16 - Bayesian Learning II

14.1 Naive Bayes Classifier

One of the simplest and most popular techniques based on Bayesian learning is the **Naive Bayes classifier**. It is an effective method when we have large data-sets and when the attributes describing the instances are conditionally independent given the classification.

Given a target function $f : X \rightarrow V$, with instances x described by a set of attributes $\langle a_1, a_2, ..., a_n \rangle$, the most probable classification of a new instance $f(x)$ is:

$$v_{MAP} = argmax_{v_j \in V} P(v_j | a_1, a_2, ..., a_n)$$

We want to maximize the probability to have class v_j given the values of the instance attributes. By resorting to the Bayes formula, it follows that:

$$= argmax_{v_j \in V} \frac{P(a_1, a_2, ..., a_n | v_j) P(v_j)}{P(a_1, a_2, ..., a_n)}$$

Note that $P(a_1, a_2, ..., a_n)$ does not depend on v_j, so we can consider it as a constant and remove it from the formula:

$$= argmax_{v_j \in V} P(a_1, a_2, ..., a_n | v_j) P(v_j)$$

- Compute $P(v_j)$ is easy. We can compute the ratio between the occurrences of instances of class v_j and the number of instances in the training set.
- Estimate $P(a_1, a_2, ..., a_n | v_j)$ is **impossible**. This is because, since we should compute a probability for each combination of $a_1, a_2, ..., a_n$, which is computationally unfeasible. Furthermore, we need to see every instance in the instance space many times in order to obtain reliable estimates. Hence, we would have a very, very large set of training data.

So, in order to overcome this problem, the Naive Bayes method makes an assumption:

$$P(a_1, a_2, ..., a_n | v_j) = \prod_i P(a_i | v_j)$$

which means that the attributes describing an instance are independent each other.

Finally, the most probable class according to the Naive Bayes classifier is given by the following formula:

$$v_{NB} = argmax_{v_j \in V} P(v_j) \prod_i P(a_i | v_j)$$

One interesting difference between the naive Bayes learning method and other learning methods is that there is no explicit search through the space of possible hypotheses (in this case, the space of possible hypotheses is the space of possible values that can be assigned to the various $P(v_j)$ and $P(a_i|v_j)$ terms). Instead, the hypothesis is formed without searching, simply by counting the frequency of various data combinations within the training examples.

14.1.1 Naive Bayes algorithm

Given a training set Tr:

1. For each target value v_j
   (a) $\hat{P}(v_j) \leftarrow$ estimate $P(v_j)$ on Tr
   (b) $\forall a_i$: $\hat{P}(a_i|v_j) \leftarrow$ estimate $P(a_i|v_j)$ on Tr (using the ratio method as done for $P(v_j)$)
2. return $\hat{P}(v_j)$, $\hat{P}(a_i|v_j)$ $\forall i, j$

The **classification** of a new instance x works as follows:

1. $v_{NB} = argmax_{v_j \in V} \hat{P}(v_j) \prod_i \hat{P}(a_i(x)|v_j)$
2. return v_{NB}

where $a_i(x)$ is the value of the i-th attribute of the instance x.

14.1.2 Naive Bayes: additional considerations

The assumption of conditional independence is often violated, that is:

$$P(a_1, a_2, ..., a_n|v_j) \neq \prod_i P(a_i|v_j)$$

Despite this, the Naive Bayes algorithm still works. This is because it is not necessary to correctly estimate the posterior probability $\hat{P}(v_j|x)$. It is sufficient that the order of those probability is *correct*.

Note that if no training examples with target value v_j has the attribute value a_i equal to k, it follows that:

$$\hat{P}(a_i = k|v_j) = 0, \; and \; \hat{P}(v_j) \prod_i \hat{P}(a_i|v_j) = 0$$

This can compromise the classification of the Naive Bayes method. A typical solution is the Bayesian *m-estimate* for $\hat{P}(a_i|v_j)$:

$$\hat{P}(a_i = k|v_j) \leftarrow \frac{n_c + mp_k}{n + m}$$

where:

- n is the number of training examples where $v = v_j$
- n_c is the number of training examples where $v = v_j$ and $a = a_i$
- p is the prior estimate of $\hat{P}(a_i|v_j)$.
- m is a constant which determines how heavily to weight p

Applications for Naive Bayes classifier are:

- Diagnosis

- Classification of textual documents

In the context of text classification, it can be used to:

- learn which documents are of interest
- learn to classify web pages by topic
- spam / no spam
- ...

14.2 Learning to classify a text

Given a document (*doc*), we want to classify it as interesting or not interesting for a particular user: $doc \rightarrow \{+, -\}$. We can represent a document by a vector of words where each word position in the document is an attribute:

$$doc = \{a_1 = w_1, a_2 = w_2, ..., a_n = w_n\}$$

We need to estimate the following probabilities:

$$P(+),\ P(-),\ P(doc|+),\ P(doc|-)$$

In this case, in addition to the assumption of conditional independence made before, we have to make another one:

$$P(a_i = w_k|v_j) = P(a_m = w_k|v_j),\ \forall i, j, k, m$$

where $P(a_i = w_k|v_j)$ is the probability that the word in position i is w_k, given v_j. Basically, we assume that the probability for each word to appear in any position is the same.

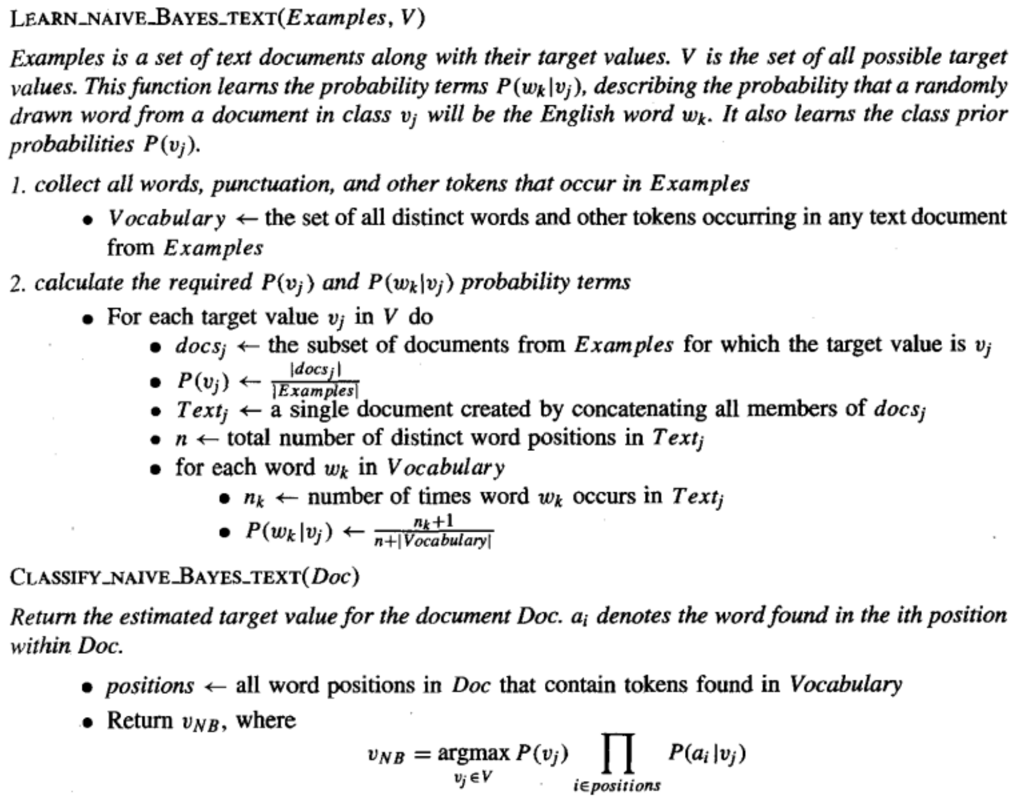
Therefore, we need to estimate *only* the $P(v_j)\ \forall j$ and $P(w_k|v_j)\ \forall k, j$.

We can use a m-estimate with uniform priors and m equal to the size of the vocabulary:

$$\hat{P}(w_k|v_j) = \frac{n_k + 1}{n + |Vocabulary|}$$

where

- n is the total number of word positions in all training documents having class v_j
- n_k is the number of times the word w_k is in these positions
- $|Vocabulary|$ is the total number of distinct words found in the training set.



```
LEARN_NAIVE_BAYES_TEXT($Examples, V$)

Examples is a set of text documents along with their target values. V is the set of all possible target
values. This function learns the probability terms $P(w_k|v_j)$, describing the probability that a randomly
drawn word from a document in class $v_j$ will be the English word $w_k$. It also learns the class prior
probabilities $P(v_j)$.

1. collect all words, punctuation, and other tokens that occur in Examples
    • $Vocabulary \leftarrow$ the set of all distinct words and other tokens occurring in any text document
      from $Examples$
2. calculate the required $P(v_j)$ and $P(w_k|v_j)$ probability terms
    • For each target value $v_j$ in $V$ do
        • $docs_j \leftarrow$ the subset of documents from $Examples$ for which the target value is $v_j$
        • $P(v_j) \leftarrow \frac{|docs_j|}{|Examples|}$
        • $Text_j \leftarrow$ a single document created by concatenating all members of $docs_j$
        • $n \leftarrow$ total number of distinct word positions in $Text_j$
        • for each word $w_k$ in $Vocabulary$
            • $n_k \leftarrow$ number of times word $w_k$ occurs in $Text_j$
            • $P(w_k|v_j) \leftarrow \frac{n_k+1}{n+|Vocabulary|}$

CLASSIFY_NAIVE_BAYES_TEXT($Doc$)

Return the estimated target value for the document Doc. $a_i$ denotes the word found in the ith position
within Doc.

    • $positions \leftarrow$ all word positions in $Doc$ that contain tokens found in $Vocabulary$
    • Return $v_{NB}$, where
          $v_{NB} = \operatorname{argmax}_{v_j \in V} P(v_j) \prod_{i \in positions} P(a_i|v_j)$
```

Chapter 15

Lec 17 - Ensemble Learning

15.1 Ensemble Learning - general idea

The idea of ensemble learning is to get predictions from multiple models and aggregate them. In the classification case, an **ensemble** of classifiers (base/weak learners) is a set of classifiers whose individual decisions are combined in some way (e.g. majority voting) to classify new examples.

We need to motivate why combining classifiers should perform better than individual classification. We will try to do it in two ways:

- Intuitively: following the Dietterich's "*3 reasons why*"
  - Statistical;
  - Computational;
  - Representational;
- Theoretically: bias-variance trade-off.

15.1.1 Statistical reason

Without sufficient data, many hypotheses can have the same level of accuracy on the training data. By "*averaging*" the classifications of several good classifiers the risk of choosing the wrong classifier is reduced. The idea is that if you average several classifications is more likely to find an hypothesis that better approximates the target function.

15.1.2 Computational reason

Even with enough training examples, learning algorithms may stuck in local optima. An ensemble constructed by running the local search from many different *starting points* (e.g. different initial weights values of Neural Networks), may provide a better approximation of the target function.

15.1.3 Representational reason

In some applications of machine learning the target function can't be represented by any of the hypotheses in H. By forming weighted sums of hypotheses drawn from H, it may be possible to expand the space of representable functions.

15.2 Bias-Variance decomposition

Given $y = f(x) + \epsilon$ where f is the target function and ϵ is the noise, and given an hypothesis g, the squared error can be decomposed as:

$$E[(y - g(x))^2] = bias^2 + variance + noise^2$$

- $noise^2$ is the **irreducible** error;
- $bias^2 = (E[g(x)] - f(x))^2$ where $E[g(x)]$ is the expected value of the hypothesis g. The bias term shows how much the average of the chosen model differs from the real target function.
- $variance = E[(g(x) - E[g(x)])^2]$ it shows how much different models differ from each other.

Generally, averaging multiple hypotheses reduces variance, but can also reduce the bias. This is because if you average multiple hypotheses, the model is more robust to the change of training set.

15.3 Types of ensemble

There are two types of ensemble:

- **Parallel:** These methods take advantage of the **independence** between the base learners.
- **Sequential:** These methods takes advantage of the **dependence** between the base learners since the overall performance can be boosted in an incremental way. This approach is usually called **boosting**.

15.3.1 Parallel ensemble

Suppose that each base (binary) classifier h_i has an **independent generalization error**, i.e., $P(h_i(\mathbf{x}) \neq f(\mathbf{x})) = \epsilon$. It means that the error of one of the classifiers on a particular instance in the training set doesn't say anything about the error of another classifier on the same instance.

We can combine T of these classifiers according to:

$$H(\mathbf{x}) = sign\left(\sum_{i=1}^{T} h_i(\mathbf{x})\right)$$

That is, taking the **majority voting**. Note that h_i, since it is a binary classifiers, can only compute either -1 or $+1$. Basically, $H(\mathbf{x})$ is > 0 if the number of *positive* predictions $(+1)$ is greater than the number of *negative* predictions (-1). Since we are using the *sign* function, H makes an error when $> 50\%$ of its base classifiers make errors. Therefore, by **Hoeffding inequality** the generalization error of the ensemble is:

$$P(H(\mathbf{x}) \neq f(\mathbf{x})) = \sum_{k=0}^{T/2} \binom{T}{k} (1-\epsilon)^k \epsilon^{T-k} \leq e^{-\frac{T}{2}(2\epsilon-1)^2}$$

It shows that the error reduces exponentially as the ensemble size T increases.

unfortunately, this is not always true. The problem is in the assumption that each classifier has an independent generalization error. In fact, similar hypothesis tend to make errors on the same examples. The goal of parallel ensemble is to build hypotheses which are as independent as possible.

One way to achieve independence of models is by using **bootstrapping**, that is, build new training sets by sampling with replacement M overlapping groups of instances of the same size. The idea is to train a different model for each sample and combine the results.

Another method is **feature randomization:** each model sees a random subset of features.

Bagging

The first approach based on this intuition is **bagging**:

1. Create k bootstrap samples:
2. Train a distinct classifier on each sample;
3. Classify new instances by majority voting / average.

For example, considering a linearly separable training set, we can train a perceptron on each bootstrap sample. At the end, we'll obtain a number of perceptrons equals to the number of samples and we can take the majority voting of their predictions.

Ideally, bagging eliminates variance while keeping the bias almost unchanged. However, in practice, weak learners are not independent, so bagging tends to reduce variance and increase bias. In fact, bagging fails when models are very similar (not independent enough). This happens if the learning algorithm is stable, that is, it doesn't change much after changing a few instances. If we have high-bias models and we want to combine them, the result tends to be bad. These methods are more suitable when we want to minimize the variance error. For example, bagging is effective when used with Decision Trees, but not effective with SVM. This is because SVM models are more robust to the change of training set; variance of SVM models is lower than the variance of Decision Trees models.

Note that combining Decision Trees does not always result in another Decision Tree (extended representation of hypothesis space).

Random Forests

A method based on bagging and Decision Trees is **Random Forests:**

1. Use k bootstrap samples to train k different Decision Trees (DTs);
2. At each node, pick a subset of features at random (it allows us to have more different DTs);
3. Aggregate the predictions of each tree to make the classification decision.

15.3.2 Sequential ensemble (boosting)

The intuition behind **boosting** is the following:

- Use the training set to train a simple predictor;
- Re-weight the training examples giving more weight to examples that were wrongly classified;
- Repeat n times;
- Combine the simple hypotheses into a single accurate predictor.

Boosting reduces **bias** (zero error on training set) by making each classifier focus on previous mistakes. It assumes that weak learners are slightly better than random guessing ($accuracy = 0.5 + \epsilon$). Basically, we have to choose weak learners that have an accuracy > 0.5. We can get classifiers that satisfy this property by using sequential training with examples re-weighting:

$$H(\mathbf{x}) = sign\left(\sum_t \alpha_t h_t(\mathbf{x})\right)$$

We combine the predictions of weak learners by weighting them, at each iteration t, by a factor α.

The most representative model based on this technique is **AdaBoost**.

AdaBoost (Adaptive Boosting)

The general idea of AdaBoost is the following: At each iteration t the training set is re-weighted (D_t), giving larger weights to examples that were wrongly classified and train a new weak classifier.

Weak learners need to maximize **weighted accuracy**, that is, minimize weighted error ϵ_t:

$$\epsilon_t = P_{i \sim D_t} = \sum_i D_t(i) [\![h_t(\mathbf{x}_i) \neq y_i]\!]$$

where $[\![\cdot]\!]$ is a function that computes 1 if $\cdot$ is true and 0 otherwise. Once ϵ_t is computed, the weight of each classifiers is computed according to its weighted error:

$$\alpha_t = \frac{1}{2} log \frac{1 - \epsilon_t}{\epsilon_t}$$

- when ϵ_t is closed to 0.5, α_t tends to 0.
- when ϵ_t is closed to 0, α_t tends to ∞

The training set is weighted using an exponential rule: *harder* examples are weighted exponentially more than *easy* ones.

It can be shown that repeating the whole procedure multiple times is a different way to minimize the following loss function:

$$E = \sum_{i=1}^{n} e^{-y_i H(\mathbf{x}_i)}$$

where $H(\mathbf{x}) = \sum_t \alpha_t h_t(\mathbf{x})$.

Many iterations of AdaBoost generate more and more complex hypothesis. Despite that, usually, AdaBoost does not over-fit. In fact, test error continues to drop even after training error reaches 0.

When/why does boosting works?

- Bagging may fail if the considered weak learners are wrong in the same regions. Boosting solves the problem by concentrating the efforts on those regions.
- Boosting works well when we have high-bias models. By combining them, we get more expressive classifiers. Hence, boosting is a **bias-reduction technique**.
- Since boosting focuses the efforts on hard examples, it is very sensitive to noise.
- The classification of new examples is done by evaluating all the hypotheses and combining them. So, it may be inefficient.

15.4 Stacking

Both bagging and boosting assume that we have a single base learning algorithm (set of homogeneous weak learners). **Stacking** is a technique for combining an arbitrary set of learning models using a **meta-model**.

The idea is to use a set of non-homogeneous models to get a different data representation. Then, we combine all the predictions by training a meta-model that takes as input the output of the predictors.

Any supervised model can be used as a meta-model, but the common choices are simple models. This is because we don't want to increase too much the complexity of the entire system:

- Averaging (regression)
- Majority voting (classification)
- Linear regression (regression)
- Logistic regression (classification)

Stacking works best when the base model have complementary strengths and weakness, that is, different inductive biases.

Chapter 16

Lec 18 - Clustering

16.1 Clustering

Clustering is the process of grouping a set of objects into groups of similar objects. It is the most common form of **unsupervised learning**, where we don't have a classification for the examples.

Given:

- A set of instances $X = \{x_1, ..., x_n\}, x_i \in X$
- A measure of similarity
- A desired number of clusters K(optional)

Compute an assignment function $\gamma : X \to \{1, ..., K\}$ such that no clusters are empty. γ is a function that maps each instance in X to one [1] of the clusters. Note that we use the term *instances* rather than *examples* because an example is usually a pair $(instance, label)$. Since in clustering we don't have labels, X is a set of instances.

The clustering problem is an ill-posed problem. This is because the notion of group is vague and arbitrary (e.g. fruits can be clustered by shape or color and in both cases clustering would make perfect sense). Therefore, a crucial step to get good results from clustering is data representation. We need to represent data in such a way that similar objects have similar representations. When we choose a representation we are implicitly choosing a measure of similarity and vice versa (inductive bias). Another important choice to make in clustering is how many clusters we want. This parameter can be fixed in advance or can be completely data driven. In general, we want to avoid *trivial* clusters (too big or too small).

Often, the goal of a clustering algorithm is to optimize an **objective function**. In theory, we could generate all possible clusterings, evaluate them and choose the one that optimizes the objective function. This method is not practical at all because there are too many possible clusterings ($\frac{K^n}{K!}$ where n is the number of instances). So, clustering is a search (optimization) problem.

Most of clustering algorithms (e.g. k-means) start with an initial assignment and then refine it. The problem of having local minima in the objective function implies that different starting points can lead to very different (and not optimal) final partitions.

[1] there are also techniques that give a probability distribution for each instance over different clusters

16.2 Clustering evaluation

How can we evaluate a clustering? There are two approaches:

- **Internal criteria** which depends on the notion of similarity and/or on the chosen representation. We evaluate the intra-class similarity (the similarity between instances of the same cluster should be high) and the inter-class similarity (it should be low).
- **External criteria** that, given an **external** *ground truth*, measures its proximity to the clustering we want to evaluate.

16.2.1 Internal methods for evaluation

A clustering is a good clustering when it produces clusters in which:

- the intra-class similarity (between examples in the same cluster) is high
- the inter-class similarity (between examples in different clusters) is low

Basically, we want to maximize the intra-class similarity and minimize the inter-class similarity. Note that this quality measure strongly depends on the chosen representation and on the similarity measure used. For example, an evaluation function V could be defined as follows:

$$V(X, \gamma) = \sum_{k=1}^{K} \sum_{i:\gamma(x_i)=k} ||x_i - c_k||^2$$

where γ is the assignment and c_k is the **centroid** of the k-th cluster (i.e. the mean of the instances assigned to the k-th cluster).

16.2.2 External methods for evaluation

The quality of a clustering is measured as the ability to recognize the hidden patterns and/or latent classes in the data. We have an external classification (*ground truth*) for the data and we want to measure how much the clustering produced resembles this ground truth. Since evaluation methods used for classification are not directly usable, we need other techniques:

- **Purity:** it is the ratio between the number of elements of the dominant **class** in a cluster (according to the ground truth classification) and the cardinality of the cluster. It's a good clustering when most of the clusters have all their instances of the same class in the ground truth. The total purity of the clustering will be the average of the purity of different clusters.
- **RandIndex:** it is similar to the notion of accuracy in classification. For each pair of examples, we evaluate the following statistics:
  - A: number of pairs of the same class assigned to the same cluster (true positives)
  - B: number of pairs of different class assigned to the same cluster (false positives)
  - C: number of pairs of the same class assigned to different clusters (false negatives)
  - D: number of pairs of different class assigned to different clusters (true negatives)

  The RandIndex definition is given by:

$$RI = \frac{A + D}{A + B + C + D}$$

We can also consider measures corresponding to Precision and Recall, that is:

$$P = \frac{A}{A+B} \qquad R = \frac{A}{A+C}$$

There exists an extension of this method called **Adjusted RandIndex** (ARI):

$$ARI = (RI - Expected\_RI)/(max(RI) - Expected\_RI)$$

where $Expected\_RI$ is a random clustering. As you can see, this measure is always < 1 and it is 1 only if RI is the $max(RI)$. It can be < 0 if the clustering produced is *worst* than a random clustering.

- **Other methods** such as entropy (or mutual information) among classes of the ground truth and the clusters produced.

16.3 Clustering algorithms

Let's start to introduce two different methods for clustering:

- **Partitioning** (or flat) algorithms: they usually start with a random partition and iteratively refine it. On each step the objective function of the algorithm improves.
- **Hierarchical algorithms:** there are two different approaches:
  - Bottom-up (or agglomerative)
  - Top-down (or divisive)

16.3.1 Partitioning algorithms

Partitioning algorithms build a partition of n instances in K clusters (we need to define the number of clusters in advance). Given a set of instances and a number K, they find a K clusters partition that optimizes a certain criterion. The overall optimal can be found by exhaustively enumerating all possible partitions and selecting the one that maximizes the criterion. This method is obviously inefficient and can't be implemented, so, usually, these kind of algorithms work on heuristics (k-means, k-medoids). There are also probabilistic or model-based methods: EM-type approaches (e.g. density measure $p(\mathbf{x}) = \sum_{i=1}^{K} p(\mathbf{x}|c_i)p(c_i)$)

K-Means Algorithm

In k-means the goal is to minimize, for each cluster, the average of the distances between the instances[2] and the *center* of the cluster (centroid). A centroid is computed as follows:

$$\mu(c) = \frac{1}{|c|}\sum_{x \in c} x$$

Instances are assigned to clusters according to the *nearest* centroid.

K-means algorithm:

1. Generate K points (seeds) in the space. These points represent the initial centroids (e.g. K randomly chosen instances);
2. Assign each instance to the cluster whose centroid is the closest according to the considered similarity/distance measure;

[2]we assume that the instances are represented as real-valued vectors

3. After assigning all the instances, recompute each cluster center as the mean of all the instances assigned to it;

4. Repeat steps 2 and 3 until the centroids stabilize.

It can be shown that, at each execution of steps 2 and 3, the value of the objective function is reduced:

$$V(X, \gamma) = \sum_{k=1}^{K} \sum_{i:\gamma(x_i)=k} ||x_i - c_k||^2$$

16.3.2 Hierarchical algorithms

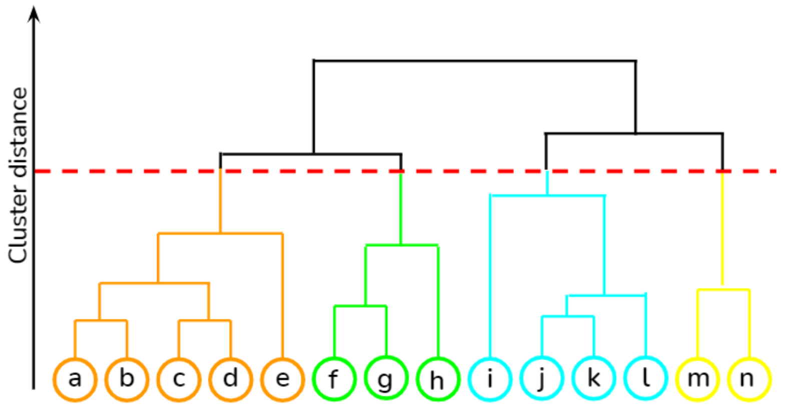
When the optimal number of clusters is not given, we can use hierarchical algorithms. The idea is to build a tree-based taxonomy from a set of instances that represents the cluster structure (dendrogram).

A first approach is to consider the recursive application of a partitioning clustering algorithm (divisive or top-down methods). These algorithms start from one single cluster that contains all the instances and partition it in multiple clusters.

A second (more popular) approach is through aggregation of clusters in a bottom-up mode. Initially, we have one cluster for each instance, and then we merge different clusters until we reach just one cluster.

Agglomerative Hierarchical Clustering (HAC)

Let's start with singleton clusters, one for each instance. Then, we gradually merge the closest pairs of clusters until we obtain a single cluster. The *history* of merges forms a binary tree or hierarchy (dendrogram).



The y axis of the dendrogram represents the similarity of the combination, that is, with which similarity the two clusters have been merged. The clustering is obtained by *"cutting"* the dendrogram to the desired level. Each connected component under a certain threshold is a cluster.

How can we measure the distance between two clusters? There are different methods:

- **Single-link:** Similarity between the most similar instances in the clusters. It tends to aggregate all the closest instances and produce *"elongated"* clusters (chaining effect).

- **Complete-link:** Similarity between the most distant instances in the clusters. It is sensitive to outliers.

- **Centroid:** Similarity between the centroids of the clusters. Note that the merge operation is assumed to be monotone, that is, if $s_1, ..., s_k$ are successive similarities obtained by the merge, then $s_1 > s_2 > ... > s_k$. With the centroid measure this is not always the case.

- **Average-link:** Mean similarity between pairs of instances of clusters. It computes the similarities between all the possible pairs of instances between the two clusters and averages these values.

Name	**Description**	**Complexity**	**Pros/Cons**
Single-link	Max sim. of any two points	$O(N^2)$	Chaining effect
Complete-link	Min sim. of any two points	$O(N^2 logN)$	Sensitive to outliers
Centroid	Similarity of centroids	$O(N^2 logN)$	Non monotonic
Group-average	Avg. sim. of any two points	$O(N^2 logN)$	OK

Chapter 17

Lec 19 - Recommender Systems I

17.1 Recommender Systems

The goal of recommender systems is to suggest to users only the items that can be really relevant. Let's consider the following two definitions of recommender system:

- **Wikipedia:** A recommender system is a subclass of information filtering system that seeks to predict the preference a user would give to an item.
- **Handbook:** Recommender systems (RSs) are software tools and techniques that provide suggestions for items that are most likely of interest to a particular user.

There are many different types of Reccomender systems:

- **Non-personalized**: they suggest the same item to all users.
  - Most popular (it suggests the most popular item)
  - Highest rated
- **Personalized:** a different recommendation may be given to different users.
  - Content-based: the recommendation is produced using only the information related to users and items;
  - Collaborative: the recommendation is produced by using all the *historical* user-item interactions;
  - Context-aware: it exploits the additional **context** information to improve the recommendation. The context represents any situation that can influence the preferences of a user.
  - Hybrid: it combines all the approaches.

17.2 Types of feedback/interaction/rating

A component that characterizes a recommender system is the type of interactions that are observed.

- **Explicit feedback**: it is a **direct** indication (usually numeric) of the relevance of an item for a user. It can be both qualitative and quantitative (e.g. number of likes/dislikes for a particular item). It is a reliable information but hard to collect since it requires an effort by the user.
- **Implicit feedback:** it is inferred by the system without user interaction, so it is easier to collect, but noisy. Examples of this feedback type are:

  - The number of views (e.g. on a website)
  - The number of click
  - The time elapsed on a particular item

17.2.1 Rating matrix

All the information gathered by implicit or explicit feedback need to be stored. We can use a **rating matrix** that has as many rows as the number of users and as many columns as the number of items. Each *cell* of the matrix represents the rating given by the user u to the item i (r_{ui}). For example, r_{ui} can either be:

- Explicit: $r_{ui} \in \{0, 1, ..., 5\}$
- Implicit: $r_{ui} \in \{0, 1\}$. It indicates if the user u has already interacted with the item i, 0 means no information.

In the real world, the majority of possible interactions are not present. In fact, the rating matrix density is typically $< 0.01\%$. For example:

- Netflix rating matrix density $\approx 0.002\%$
- MovieLens rating matrix density $\approx 0.005\%$

Moreover, both users activity and items popularity usually follow a **long tail distribution**. It means that there are few users that have given many ratings, while the majority of users have given few ratings, the same is valid also for items popularity.

17.3 Recommendation tasks

There are two main recommendation tasks:

- **Rating prediction:** when the feedback is explicit, the recommender system aims at predicting the missing ratings in the rating matrix.
- **Top-N item recommendation:** when the feedback is implicit, the recommender system aims at predicting the N items the user will like the most. Given a user, a recommender system associates a relevance score to the items. This scoring will be used to rank items and make the recommendation. Specifically, the N highest scored items are recommended to the users.

17.4 Quality indicators

How can we measure the quality of a recommender system? For the *rating prediction* case, we can check if the predicted ratings match the real ones given by the user (similar to the supervised learning task), but for the *top-N* approach is not obvious. In this case, there are different quality indicators:

- **Relevance:** recommend items that users like
- **Coverage:** ability to recommend most of the items in a catalogue
- **Novelty:** recommend items unknown to the user
- **Diversity:** diversify the recommended items (suggest group of items that are different each other)
- **Serendipity:** ability of surprising the user, that is, the ability to recommend items that users would have never been able to discover by themselves

There are two main paradigms to **evaluate recommender systems:**

- **Offline:** It is based on benchmark data-sets (training set + test set) and does not require the user interaction. It cannot be used to evaluate the experience as a whole.
- **Online:** users are directly involved in the evaluation that can be both qualitative and quantitative. The overall user-experience plays a role. However, users are not always consistent.

17.4.1 Online evaluation

This evaluation can be performed in two different ways:

- Direct user feedback: users directly provide a feedback about the recommendation through questionnaires or self-reports.
- A/B testing: the recommender is tested against a baseline (usually a previous version of the system) on two sets of users; a control set that uses the baseline and a variation set that uses the new system. The improvements are evaluated in terms of standard metric or through users feedback.

17.4.2 Offline evaluation

Usually, offline evaluation methods work by dividing the rating matrix in two parts: training set and test set. How can we perform this split? There are two alternatives:

- Divide users in training users and testing users. Then, the testing users partition is split into training ratings and test ratings. It avoids the cold-start problem, i.e., no user are unknown at testing time.
- The rating matrix is split into training ratings and testing ratings (can have cold-start). When available, it is good practice to split training-set ratings based on the timestamp.

Offline evaluation for rating prediction

We have at least 3 different ways to measure how good the predicted ratings are.

Given the predicted ratings $\hat{r}_{ui}$, for (u, i) in the test-set (Te), the usual evaluation metrics are:

- **MAE (Mean Absolute Error)** $= \frac{1}{|Te|} \sum_{(u,i)\in Te} |r_{ui} - \hat{r}_{ui}|$
- **MSE (Mean Squared Error)** $= \frac{1}{|Te|} \sum_{(u,i)\in Te} (r_{ui} - \hat{r}_{ui})^2$. The problem of this measure is that the results are not comparable with the real ratings.
- **RMSE (Root Mean Squared Error)** $= \sqrt{\frac{1}{|Te|} \sum_{(u,i)\in Te} (r_{ui} - \hat{r}_{ui})^2}$

Offline evaluation for top-N approach

In this case we can use the typical evaluation metrics, for example [1]:

- **Recall**
$$\frac{\#relevant\,recommended\,items}{\#relevant\,items} = \frac{TP}{FN + TP}$$
- **Precision**
$$\frac{\#relevant\,recommended\,items}{\#recommended\,items} = \frac{TP}{FP + TP}$$

[1]legend: TP: True Positives, FN: False Negatives, FP: False Positives

Note that all these metrics can be limited to the first k retrieved items to give more emphasis to the top of the recommended list.

Other evaluation methods are:

- **AUC (Area Under Curve)**

  $$\frac{1}{N^+N^-}\sum_i \sum_j [\![s(i) > s(j)]\!]$$

  where N^+ are the relevant items and $N^- = N - N^+$. $s(i)$ is the score given by the recommender to the item i of the recommended list. It computes the number of miss ordered pairs of items in the ranking list. Note that making a mistake at the top of the list is the same of making a mistake at the end. For this reason, this measure is not the best choice.

- **AP (Average Precision)**

  $$\frac{\sum_k Precision@k \cdot rel(k)}{\#relevant\, items}$$

  It computes the average of the precision at a given level k. $rel(k) \in \{0, 1\}$ indicates whether the k-th item is relevant or not. As for precision and recall it can be truncated at level k.

- **DCG (Discounted Cumulative Gain)**

  $$\sum_{i=1}^{p} \frac{rel_i}{log_2(i + 1)}$$

  where rel_i is the graded relevance of the item i, usually $\in \{0, 1\}$. This quantity decreases as the position i of the recommended list increases. We have a high quality recommendation when we have all the relevant items proposed at the top of the list.

- **MRR (Mean Reciprocal Rank)**

  $$\frac{1}{|Q|}\sum_{i\in Q} \frac{1}{rank_i}$$

  where Q is the set of relevant items. It averages the reciprocal of the positions that the relevant items have in the ranked list produced by the recommender. Similar to DCG, it is useful at top ranks.

As we said before, there are other quality indicators beyond relevance. For example, **Diversity** and **Novelty**:

- **Diversity:**

  $$\frac{\sum_{i\in R} \sum_{j\in R, j\neq i} 1 - sim(i, j)}{m(m - 1)}$$

  where R is the retrieved set of m items and sim is a similarity measure. This quantity is high when the similarity between all the possible pairs in the recommendation is low.

- **Novelty:**

  $$\frac{\sum_{i\in TP} log_2\left(\frac{1}{popularity(i)}\right)}{|TP|}$$

  Approximately, the inverse of the popularity of the retrieved items.

17.5 Non-personalized RS

Non-personalized recommender systems are usually used as baseline to compare other systems.

- **Most popular:** it recommends the item with the highest number of ratings k. If the user u has already interacted with item k (already given a rating), it will be recommended the most popular item after k.
- **Highest rated:** it recommends the item with the highest average rating. Usually, a normalization factor is added to the average in order to give a bias towards popular items.

17.6 Other RS approaches

Let's have a look at other standard RS approaches:

- **Content Based (CB):** recommend the most similar items to the ones the user liked in the past. For example, same genre (movies), same artist (song), etc...
- **Collaborative Filtering (CF):** recommend to a user the items liked by similar users, or vice versa, items that are similar to the ones liked in the past. In this case, what is changing with respect to the Content Based approach is how the similarity between items or users is measured:
  - item-item similarity: two items are similar if they share many users.
  - user-user similarity: two user are similar if they share many ratings.

  Note that in these approaches only the interactions are used to compute similarities. No specific users and items characteristics are used.
- **Hybrid approaches:** Content Based approaches tend to perform better than Collaborative Filtering when there is no much *history* about interactions. On the other hand, when we have a lot of information about user-item interactions, than Collaborative Filtering approaches perform better. Hybrid approaches tend to take advantage of the strength of the different methods while mitigate their weakness.
- **Context-aware (CARS):** In these cases the assumption is that the quality of a recommendation depends on the user (and item) state. The recommender uses contextual information to tune the recommendation, for example, the mood, the weather, the time, the presence of kids, etc...

Chapter 18

Lec 20 - Recommender Systems II

18.1 Notation

Let's start with the most important keywords for recommender systems:

- U is the set of users, $|U| = n$;
- I is the set of items, $|I| = m$;
- $R \equiv \{(u, i) | u \in U \ rated \ i \in I\}$ is the set of ratings/interactions;
- $\mathbf{R} \in \mathbb{R}^{n \times m}$ is the rating matrix with n users and m items. r_{ui} is the rating given by u to item i. $\mathbf{r}_u$ is the row vector representing u and $\mathbf{r}_i$ is the column vector representing i;
- $I_u \equiv \{i | (u, i) \in R\}$ is the set of items rated by u;
- $U_i \equiv \{u | (u, i) \in R\}$ is the set of users who rated i;
- $I_{uv} \equiv I_u \cap I_v$ is the set of items rated by both users u and v;
- $U_{ij} \equiv U_i \cap U_j$ is the set of users who rated both i and j.

18.2 Collaborative Filtering approaches (CF)

We can classify CF approaches in two different classes:

- **Similarity-wise:** the recommendation is performed on the basis of similarity between either items or users (e.g. two users are similar if they share many ratings).
- **Algorithm-wise:**
  - Memory-based: approaches that use the user rating data to compute the similarity between users or items.
  - Model-based: in this approach, models are developed using different algorithms to predict users ratings of unrated items.

18.3 Computing similarity

How can we compute the similarity between users or items? The idea is to compute similarity between users/items starting from the user rating matrix $\mathbf{R}$. Ideally, two users (items) are similar if they share many ratings.

18.3.1 Implicit feedback: cosine similarity

Let's start with the similarity between users or items with respect to implicit feedback:

- **User-based:**

$$s_{uv} = \frac{|I_u \cap I_v|}{\sqrt{|I_u||I_v|}} = \frac{\mathbf{r}_u \mathbf{r}_v^T}{||\mathbf{r}_u||||\mathbf{r}_v||}$$

- **Item-based:**

$$s_{ij} = \frac{|U_i \cap U_j|}{\sqrt{|U_i||U_j|}} = \frac{\mathbf{r}_i^T \mathbf{r}_j}{||\mathbf{r}_i||||\mathbf{r}_j||}$$

Note that , since the values of the rating matrix are binary values (implicit feedback), computing the dot product between $\mathbf{r}_u \mathbf{r}_v^T$ corresponds to compute the number of shared items (the same is valid also for the item-based case). Also, the squared norm represents the size of the given set.

18.3.2 Explicit feedback: Pearson correlation

The following similarity measure is used when in the rating matrix there are non-binary values (explicit feedback):

- **User-based:**

$$s_{uv} = \frac{\sum_{i \in I_{uv}} (r_{ui} - \mu_u)(r_{vi} - \mu_v)}{\sqrt{\sum_{i \in I_{uv}} (r_{ui} - \mu_u)^2 \cdot (r_{vi} - \mu_v)^2}} \in [-1, 1]$$

  where μ_u and μ_v are the user-bias (average of ratings that user u gave to different items):

$$\mu_u = \frac{1}{|I_u|} \sum_{i \in I_u} r_{ui}, \quad \mu_v = \frac{1}{|I_v|} \sum_{i \in I_v} r_{vi}$$

  Basically, μ_u and μ_v are used to normalize the quantity with respect to different users behaviors (some users can give very high ratings to items, while other users may give low ratings).

- **Item-based:**

$$s_{ij} = \frac{\sum_{u \in U_{ij}} (r_{ui} - \mu_i)(r_{uj} - \mu_j)}{\sqrt{\sum_{u \in U_{ij}} (r_{ui} - \mu_i)^2 \cdot (r_{uj} - \mu_j)^2}} \in [-1, 1]$$

  where μ_i and μ_j are the item-bias (average of ratings that users gave to item i):

$$\mu_i = \frac{1}{|U_i|} \sum_{u \in U_i} r_{ui}, \quad \mu_j = \frac{1}{|U_j|} \sum_{u \in U_j} r_{uj}$$

  The differences in the ratings scales of users are usually more pronounced than the differences in ratings given to items. Therefore, it may be more appropriate to compare ratings that are centered on their user mean, instead of their item mean:

$$s_{ij} = \frac{\sum_{u \in U_{ij}} (r_{ui} - \mu_u)(r_{uj} - \mu_u)}{\sqrt{\sum_{u \in U_{ij}} (r_{ui} - \mu_u)^2 \cdot (r_{uj} - \mu_u)^2}} \in [-1, 1]$$

  where μ_u is the user-bias seen before.

18.3.3 Shrinkage:

The problem with the similarity measures seen before is that they could be maximized despite the number of ratings. In order to solve this problem, we can use **shrinkage**. It aims at re-weighting the similarity penalizing the one based on few ratings.

- **User-based**

$$s'_{uv} = \left(\frac{|I_{uv}|}{|I_{uv}| + \beta}\right) s_{uv}$$

- **Item-based**

$$s'_{ij} = \left(\frac{|U_{ij}|}{|U_{ij}| + \beta}\right) s_{ij}$$

where $\beta > 0$ (usually set around 100) is a hyper-parameter. Note that when $|I_{uv}|$ ($|U_{ij}|$) is small and β is high, the coefficient will be very small (penalize s_{uv}). On the other hand, when $|I_{uv}|$ ($|U_{ij}|$) is very high, the coefficient is closed to 1.

18.4 K-Nearest Neighbours

k-nearest neighbours is a memory-based method that computes the recommendation as a weighted combination of the ratings given by the most similar users (items).

- **User-based:**

$$N_i^k(u) \equiv \{v \in U_i| \; |\{v' \in U_i | s_{uv'} > s_{uv}\}| < k\}$$

- **Item-based:**

$$N_u^k(i) \equiv \{j \in I_u| \; |\{j' \in I_u | s_{ij'} > s_{ij}\}| < k\}$$

18.5 Matrix factorization

Matrix factorization (MF) is a model-based technique that is not based on a predefined similarity measure. It maps both users and items to a **joint latent factor space** of dimension k. User-item interactions are modelled as dot-products in that space. The goal is to learn users and items vector representations $\mathbf{p}_u$ and $\mathbf{q}_i$.

- $\mathbf{q}_i$ measures how much those k factors are present in the item i.
- $\mathbf{p}_u$ measures the importance that user u gives to different factors.

The dot-product $\mathbf{p}_u\mathbf{q}_i^T$ captures the interaction between user u and item i.

Each element of the summation $\mathbf{p}_u\mathbf{q}_i^T = \sum_f p_{uf} \cdot q_{if}$ represents the relevance of factor f in the prediction.

- High values of p_{uf} means that the factor f is relevant for user u;
- High values of q_{if} means that the factor f is present in item i;

If the result of the dot-product is high, it means that a positively relevant factor for u is present in i.

$$\mathbf{R} \approx \mathbf{P}\mathbf{Q}^T$$

where $\mathbf{R} \in \mathbb{R}^{n \times m},\ \mathbf{P} \in \mathbb{R}^{n \times k},\ \mathbf{Q} \in \mathbb{R}^{m \times k}$. Basically, $\mathbf{P}\mathbf{Q}^T$ should approximates the rating matrix $\mathbf{R}$.

Learning $\mathbf{P}$ and $\mathbf{Q}$ can be done by optimizing the following loss function:

$$L(\mathbf{P}, \mathbf{Q}) = min_{\mathbf{P},\mathbf{Q}} \sum_{(u,i) \in Tr} \left(r_{ui} - \mathbf{p}_u \mathbf{q}_i^T\right)^2 + \lambda(||\mathbf{p}_u||^2 + ||\mathbf{q}_i||^2)$$

This loss function depends on two terms:

- $\epsilon_{ui} = \left(r_{ui} - \mathbf{p}_u \mathbf{q}_i^T\right)^2$: is the prediction error of the model on the rating r_{ui};
- $\lambda(||\mathbf{p}_u||^2 + ||\mathbf{q}_i||^2)$: it depends on the norm of $\mathbf{p}_u$ and $\mathbf{q}_i$. $\lambda \geq 0$ is a regularization hyper-parameter.

This learning process can be done using, for example, **Stochastic Gradient Descent** (SGD). For each training rating r_{ui} it makes a prediction using the current model and computes the prediction error e_{ui}. Then, it computes the partial derivatives with respect to $\mathbf{p}_u$ and $\mathbf{q}_i$ and updates them accordingly.

18.5.1 Alternate Least Square (ALS)

An alternative way to do the same is Alternate Least Square method.

In this case, we fix $\mathbf{p}_u$ and $\mathbf{q}_i$ alternatively while optimizing for the other. In this way, the new optimization problem becomes a **quadratic problem** and its solution has a closed form.