Giuliano Abruzzo

Machine Learning Notes

2018/2019

CAP 1: Intro

Machine learning is a component of *AI*, in fact sometimes it is confused with it. It's a component that allows an agent to learn from experience, this intelligent agent need to learn to make decision, so learning is a fundamental part of the ML. The goal is to improve performance over time, based on experience. Data must be transformed into knowledge:

- **Data**: is formed by bit of information
- **Knowledge**: is a semantic representation of these data

So, ML can be seen as a tool for generating knowledge from data.

ML is useful when you don't know exactly how to model a problem, in fact sometimes an optimal solution for a model is not an optimal solution for a real system. So, when we don't know how to describe a dynamic system, then ML is useful and allows to solve the problem without requiring any model.

Model approach is based on modelling a problem and find a math formulation of this model, instead *ML will not require model*, it just uses data and information extracted from the system and then build a solution from these. This approach is many cases is more effective and improves the performance.

In this period, we have an exploit of ML, in fact the huge availability of *data* and the increasing of the *computational power*, made possible to use specific *algorithm* to find solution for a problem, these were not effective when they were theorized, now with this data and with this computational power they increase the performance. But it's not enough, we have to understand how to design a system.

A **learning problem** can be described as follow:

- **Learning**: Improving with experience at some task.
- Improve over task **T**, that is the *task* that we want to learn;
- With respect to performance measure **P**, that is the performance metrics;
- Based on experience **E**, these are the data user for experience.

If you don't specify these 3 elements the learning problems is not well defined, and the solution as well. In a learning problem we have to define:

- How to collect *experience* → ***target function***.
- What is the *structure* of the problem (what should be learned);
- What type of *data structure* we are going to use;
- What *algorithm* are we using to learn it;

Collecting the experience is the most difficult part, in many cases the data are already available, also the function sometimes is quite easy to define. The first thing is choose the function that you want to learn, then you choose a representation of the function and depending on the other choices you will choose an algorithm and get the solution.

There are 3 main categories for ML problems:

1. **Supervised Learning**
   - *Classification*
   - *Regression*
2. **Unsupervised Learning**
3. **Reinforcement Learning**

The general ML problem is defined as:

A **generic Machine Learning** (ML) **problem** is learning a function $f : X \rightarrow Y$, given a dataset *D* containing sampled information about *f*. (*D is a set of samples of this function)*. Learning a function *f* means computing an approximated function *f '* that returns values as close to *f* also on samples *x* not int the dataset *D*.

What is different in the ML problems categories is how data is represented.

Supervised Learning is a set of problems in which the *dataset* is formed by a set of pairs input-output. So, if I take an element from *X* I know the corresponding value in *Y*. So, *X* can be *continuous* or *discrete*, also *Y.* According to the different kind of set we have different names.

We talk about **Classification** when we have the task of learning a function that has a *finite* codomain. We talk about **Regression** when we have the task of learning a function whose domain output is *infinite*.

- Supervised Learning: $D \subset \{(x, y) \mid x \in X, y \in Y\}$;
  - X finite sets: Discrete;
  - $X \subset R^n$: Continuos;
  - Y finite sets: Classification;
  - $Y \subset R^k$: Regression;

Classification or *Pattern Recognition* return the class to which a specific instance belongs, so given the input, you want to classify it in a predefined class. **Regression** instead try to approximate real-valued functions.

In the **Unsupervised Learning** we have only the *input* values and we don't know nothing about the corresponding value in the *codomain*. From the input you can still extract knowledge, in fact you can make clusters and understand if they are significant or we can estimate density or other parameters. Very often this analysis is done in combination with the classification problem.

- Unsupervised Learning: $D \subset \{x | x \in X\}$

The **Reinforcement Learning** is applied to dynamic system, in which we want to learn a function where the input domain is a set of *all possible states* *S* with *associated actions* and *rewards* and the learned function is a *transition policy*. So, the data is a sequence of actions and states and then we have a value that say how good is that "episode" is. In the typical case you have something that changes over time and you want your agent to be able to learn from it, so the choice is the right to do according to the current situation.

- Reinforcement Learning: $D = \{(a_i^1 ... a_i^n, r_i) | i \in 1...|S|\}$

We also have a special case called **Concept Learning** in which we have: the input domain is finite, and the output is made by two elements, like Booleans, or 0 and 1, and so on.

We call c a *target function* where c: $\mathrm{X} \rightarrow \mathrm{Y}$ is the function that we want to learn, where X is an *instance space* (input set) all the possible inputs of the function, where $\mathrm{x} \in \mathrm{X}$ a *particular instance* in the set. The *dataset* $\mathrm{D} = \{(\mathrm{x_i}, \mathrm{c}(\mathrm{x_i}))\}$ is a set of pairs in which for each instance $\mathrm{x_i}$ we have the corresponding value of target function $\mathrm{c}(\mathrm{x_i})$ that can be computed only for instances of the dataset (x is in D). We call **hypothesis space** H the set of all possible functions that I can compute, where $\mathrm{h} \in \mathrm{H}$ an **hypothesis** and represent an approximation of the target function. We call $\mathrm{h}(\mathrm{x})$ an **estimation** of h over x (predicted value) and we want that $\mathrm{h}(\mathrm{x})$ is similar as possible to $\mathrm{c}(\mathrm{x})$ especially for values not in dataset.

So, the goal of a learning task can be a searching problem in H to find the best h (we find the best approximation). We say that a hypothesis h is **consistent** with the train set D if: $c(x) = h(x) \forall x \in D.$

The real goal is to find the **best hypothesis** that approximates the function c for elements that are **outside** the dataset.

Given a training set $\mathrm{D} = \{(\mathrm{x_i}, \mathrm{c}(\mathrm{x_i}))\}$ and a hypothesis $\mathrm{h} \in \mathrm{H}$, a performance measure is based on evaluating $\mathrm{c}(\mathrm{x_i}) = \mathrm{h}(\mathrm{x_i})$ for all $\mathrm{x_i} \in \mathrm{D}$. We call **inductive learning hypothesis** when if a hypothesis does well inside the dataset it will

do well also outside the dataset, of course the dataset must be representative for the problem that we are considering (large dataset).

We call **Version Space** the subset of H that is **consistent** with the dataset D. Each of these hypotheses is mapped to a subset of X that is also consistent with D, and this means that $\mathrm{h(x)}$ = the values of X.

$$VS_{H,D} := \{h \in H | h \text{ is consistent with } D\}$$

We call *list-then-eliminate algorithm* an algorithm that compute the version space by iterating on all the hypothesis and checking for any if they are consistent. This in practice is not possible (we can have infinite hypothesis).

If any subset of the instances can be represented in a **Version Space** and all the solution are computed (search is completed) then the system is not able to classify new instances. In fact, different hypotheses in a version space may return different values for a new instance $x' \notin D$.

Collecting data is difficult and it's easy to make some mistakes, these called **noisy data** can be into the dataset and so the output is different from the true value of the target function. In this case we can have *no consistent hypothesis* so we need **statistical methods** for remove the noise.

Cap 2: Classification Evaluation

For introduce the notion of **performance metrics** in ML we have to define a *probability distribution* of all the possible inputs. Let Z be the probability distribution over X, and S are a set of n instances of X sampled with Z. The performance evaluations are based on **accuracy** and **error rate**. We can have two definitions of error/accuracy:

- The **true error** of hypothesis h with respect to target function c and distribution Z is the probability that h will misclassify an instance according to Z:

$$error_Z(h) = Pr_{x \in Z}[c(x) \neq h(x)]$$

- The **sample error** of hypothesis h with respect to target function c and data sample S is the number of mistakes that h makes on S:

$$error_S(h) = \frac{|\{x \in S | c(x) \neq h(x)\}|}{|S|}$$

The **true error** represents the probability of making a mistake given an input taken from the *entire distribution*, and this is the error that the system will make at run time. The problem is that this error cannot be computed cause we don't know $\mathrm{c(x)}$.

The **sample error** is the number of mistakes that h makes on a data sample, it can be computed cause for the samples of S we know the values of the target function. The problem is this not enough cause we cannot be sure that this will be the "resulting error" of the entire distribution.

We have that $\mathrm{accuracy(h) = 1 - error(h)}$. So we have that the **true error** can be just estimated instead **sample error** can be computed. We need to remember that the goal of a learning system is to be accurate in: $h(x), \forall x \notin S$

We call **bias** the difference between the *expected value* of the *sample error* and the *true error*. We want to obtain a bias of 0, with this value in fact we can approximate the *true error* with the *expected value* of the *sample error*. For **unbiased estimate** we must have a h and a S chosen independently:

$$E[error_S(h)] = error_Z(h)$$

So in order to calculate the *true error* we need to calculate the **expected value** of the *sample error* by using an **unbiased estimator**. This is done as follow:

- Split the dataset D in: $D = T \cup S$ and train on T, and $T \cap S = \emptyset$, $|T| = 2/3|D|$
- Compute a hypothesis h using **training set** T and **evaluate** the *sample error* so we obtain an *unbiased estimator* for the *true error*.

At the end of this we can make a statement, with some probability the *true error* is in an interval defined by the *sample error*, where these 2 values are defined by a coefficient $\mathrm{Z_n}$.

$$error_S(h) \pm z_N \sqrt{\frac{error_S(h)(1 - error_S(h))}{n}}$$

When we are comparing two different hypothesis on a sample S, we have **no guarantee** that if an hypothesis is better than the other in S the same result will be also for the entire distribution.

We say that a hypothesis h overfits the training data if exists an h' such that:

$$error_S(h) < error_S(h') \wedge error_Z(h) > error_Z(h')$$

So, in the **overfitting** we have that the *sample error* of h is lower than the sample error of h' but the *true error* is greater, so we have a problem.

We can evaluate a hypothesis h by estimating the error on different samples S, by using the **K-Fold method**:

1 $D = S_1 \cup S_2 \cup ... \cup S_k$
2 **for** $i = 1$ **to** k
3 train h_i on $D \setminus S_i$
4 $\delta = \delta + error_{S_i}(h_i)$
5 $error_{k,D} = \frac{\delta}{k}$

In which we *partition* the dataset in **k set** (1), for all the *k set* (2) we train the hypothesis with the **training set** (3) and we **compute** the *sample error* (4). Once I did this k times, I just compute the **average** of all the errors (5) and I get the best approximation of the *true error*. If k is too small the average will be not good (in general k=30 is good enough).

Accuracy can be not a valid metric in some cases, in fact, let's consider a binary classification problem with a dataset D with 98% of *true* and 2% of *false*. A *dumb* hypothesis that returns always *true* has a *very high* accuracy. So we use other performance metrics:

	Predicted class	
True Class	Yes	No
Yes	TP: True Positive	FN: False Negative
No	FP: False Positive	TN: True Negative

Where:

- **True positive** represents the instances correctly classified as *positive*;
- **True negative** represents the instances correctly classified as *negative*;
- **False positive**: samples that were *negative* but classified as *positive*;
- **False negative**: samples that were *positive* but classified as *negative*;

From these values derives other *measures*:

- Accuracy: $(TP + TN)/(TP + FN + TN + FP)$
- Precision: $TP/(TP + FP)$
- Recall: $TP/(TP + FN)$
- False positives rate: $FP/(FP + TN)$
- False negatives rate: $FN/(FN + TP)$
- F-Measure: $2{*}\text{Preision}{*}\text{Recall}/(\text{Precision} + \text{Recall})$

A **confusion matrix** represents in each entry how many instances of class C_i is misclassified as an element of class C_j. So, the main diagonal contains accuracy for each class.

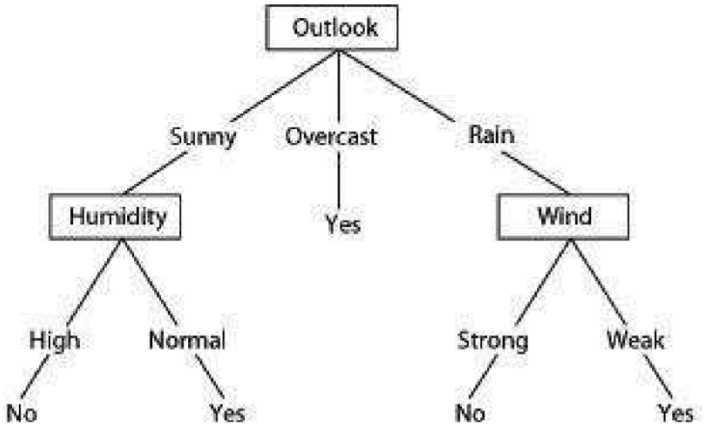
Cap 3: Decision Tree

The problem we are concentrating on is: given a *training set* D for a *target function* c, we need to compute a **consistent** hypothesis respecting D. The approach for resolve it can be: define an hypotheses space H, and implement an algorithm to search an $\mathrm{h} \in \mathrm{H}$ that is consistent with D.

We are going to considerate an hypotheses space made by the set of **decision tree**. This means that every hypothesis is a decision tree, that is a tree formed in this way:

- each **internal node** represents an attribute;
- each **branch** represents a value of an attribute;
- each **leaf** assigns a classification value;

A **decision tree** can be transformed in a **set of rules**, it represents the disjunction of conjunction of all the paths to the positive leaf nodes. So, if we have for example this decision tree:



We obtain these rules:

$$(Outlook = Sunny \wedge Humidity = Normal) \vee (Outlook = Overcast) \vee (Outlook = Rain \wedge Wind = Weak)$$

So, we have an **OR** for each *edge* from the *root*, and we have an **AND** for each *internal node* in the path.

Now if we want to **create a decision tree** from a dataset D, we have to choose a target attribute. We will use the ID3 algorithm, this algorithm takes in input:

- **Examples**: data instances;
- **Target attribute**: what we want to learn;
- **Attributes**: the set that form the domain of the function;

The tree is built in a recursive way. In the first step it creates the Root node for the tree. Then we can have 3 different steps (all final cases) and a 4th case that is the recursive case:

1. If all *Examples* are **positive** (+ , true , 1 , …) then return the node *Root* with a label **+** , this means that we will generate a tree that contains only *one leaf node* that is labelled with **+**.

2. If all *Examples* are **negative** (- , false , 0 , …) then return the node *Root* with a label **-** , this means that we will generate a tree that contains only *one leaf node* that is labelled with **-**.

3. If Attributes is **empty**, then return the node *Root* with a *label* = most common value of *Target attribute* in *Examples*. This will happen cause at some point the recursive will reduce the set of attributes. If it's empty, we are going to create a leaf, this is an important feature of decision tree cause it's using statistic, and this happened when we have a noisy dataset.

4. This is the **recursive case**: first we choose the "**best**" attribute in *Examples,* so given the best attribute we generate all branches associated. Now we take the subset of *Examples* that contains the branches of the best attribute. If this subset is empty like before we generate a leaf node with a label = most common value of *Target attribute* in *Examples.* Else we call the *recursive* algorithm by passing it, the new subset of *Examples* and the list of the *Attributes* without the *best attribute* used.

If the dataset is **consistent**, the choice based on *most common value* will never happen, in fact there will be never any conflict. This algorithm will produce *consistent data set* whatever is the *choice* of the attribute.

The algorithm can be written as follow:

```
1  algorithm ID3(examples, target_attribute, attributes)
2      root := new Node()
3      if label(e) is + ∀e ∈ examples
4          label(root) = +
5          return root
6      if label(e) is − ∀e ∈ examples
7          label(root) = −
8          return root
9      if attributes is ∅
10         label(root) = most common value of target_attribute in
               ↪ examples
11         return root
12     A := best_decision_attribute(examples)
13     label(root) = A
14     foreach v ∈ A
15         branch := add_branch(root, test(A = v))
16         E_v := {e ∈ examples|e_A = v}
17         if E_v is ∅
18             leaf := new Node()
19             label(leaf) = most common value of target_attribute in
                   ↪ examples
20             add_node(root, branch, leaf)
21         else
22             add_subtree(root, branch, ID3(E_v, target_attribute, attributes \ {A}))
```

We need to choose the **best attribute** because it will increase the accuracy of the tree. ID3 select the attribute with the highest **information gain**, this **measures** how well a given attribute separates the training *Examples* according to their target classification. We can measure it with the **entropy**.

Let's call $\mathrm{p+}$ the proportion of positive examples in S $(|positive\ examples|/|examples|)$ and $\mathrm{p\text{-}}$ the proportion of negative examples in S $(1 - \mathrm{p+})$. **Entropy** is defined as:

$$entropy(examples) = -p_+ * log_2(p_+) - p_- * log_2(p_-)$$

So, **Entropy** is max when the dataset contains exactly half positive and half negative, so when $\mathrm{p+} = \mathrm{p\text{-}} = 0{,}5$.

So, the **information gain** is defined as the expected reduction in *entropy* of the examples caused by the value of *attribute* A (reduction of the entropy if we use A to partition the examples/S).

$$gain(examples) = entropy(examples) - \sum_{v \in A} \frac{|E_v|}{|examples|} * entropy(E_v)$$

So, one way to define the *best attribute* for this step it to **compute** the *gain* of each attribute with this method and select the one with the *maximum gain.*

At the end the **ID3**:

- Return a hypothesis space that is **complete**;
- In output we have **only one** hypothesis consistent with the dataset;
- **No back-tracking** cause once a decision is taken there's no way to change it. This may limit the performance cause in some case you can find out that later this is not a good choice and you can't go back;
- It's **robust to noisy data** cause it use a *statically-based search* choices;
- Uses all training examples at each step, not good, in fact, adding a single sample could change the *best attribute* and therefore also the whole *tree*.

Now a question, is it good that the tree *grows* over time to "*accommodate*" all possible instances? The answer is no, cause can produce **overfitting**, in fact it could happen that even with a better accuracy computed on the dataset used for *training* we may have that the accuracy on *test* dataset decrease.

For avoid the **overfitting** we try to do two things: we stop *growing* the tree when data is not *splitting* in a significant way, or we build the *complete tree* and we start **cutting** some part of the tree for reduce his size (**post-prune**).

In order to decide which part of the tree I have to cut, I must estimate if this cut will improve the *accuracy* of the *classification*, and this must be done by using samples not used for training. So, I need an approach like **cross validation**, so before I start the process, I partition the dataset in *training* and *validation set* (the second one used for evaluating the accuracy of the tree). This is called **Reduced-Error Pruning**, in which we replace *subtrees* with the more common values in order to improve *accuracy* on the *validation set*.

If *Attributes* have *continuous* values, we can use intervals in order to have discrete attributes. Else we have *attributes* with Many values or with a cost we can use different *performance metrics* (not gain) but still based on the reduction of the *entropy*. If for some samples we have missing values of an attribute there are *statistical methods* to generate a solution, without losing the information.

There are other methods based on *decision trees*, like **Random Forest** that is a method that generates a set of *decision trees* using random criteria *(e.g. random picking attributes)* and integrates their values into a final value. Usually they perform better that one single decision tree. The result of the *Random Forest* is the most common *classification* of all the trees (majority vote) and they are less sensitive to overfitting.

Cap 4: Probability

Sample space

- Ω *sample space* (set of possibilities)
- $\omega \in \Omega$ is a *sample point/possible world/atomic event/outcome of a random process/...*

Probability space (or *probability model*)

- Function $P : \Omega \mapsto \Re$, such that
  - $0 \leq P(\omega) \leq 1$
  - $\sum_{\omega \in \Omega} P(\omega) = 1$

An *event* A is any subset of Ω
Probability of an event A is a function assigning to A a value in $[0, 1]$

$$P(A) = \sum_{\{\omega \in A\}} P(\omega)$$

A *random variable* (outcome of a random phenomenon) is a function from the sample space Ω to some range (e.g., the reals or Booleans) $X : \Omega \mapsto B$.

Prior probability is the probability of an event with knowing any other information (e.g. $\mathrm{P}(\mathrm{Odd} = \mathrm{true}) = 0.5$).

A **probability distribution** is the set of probability values for all the possible assignment of a random variable (the sum of all values must be 1), for continuous variables the probability distribution is a continuous function.

Conditional probability is the probability of an event given another event happened:

- Conditional probability: $P(a|b) = P(a \wedge b)/P(b)$ if $P(b) \neq 0$;
- Product rule: $P(a \wedge b) = P(a|b)/P(b) = P(b|a)/P(a)$;

- Sum rule: $P(a \vee b) = P(a) + P(b) - P(a \wedge b)$;
- Total probability: $P(a) = P(a|b) * P(b) + P(a|\neg b) * P(\neg b)$ and in general $P(X) = \sum_{y \in values(Y)} P(X|Y = y) * P(Y = y)$ when all values y are mutually exclusive.
- Idependence: X is independent from Y given Z if $P(X, Y|Z) = P(X|Z)$;
- Bayes rule: $P(a|b) = \frac{P(b|a)*P(a)}{P(b)}$;

Given a set of *random variables* we can generate a *graph* in which random variables are nodes while the edges represent *dependencies*. This graph is called **Bayesian Network**.

We can interpret a **classification** problem as a **probabilistic estimation**.

$$\text{Given } x' \notin D \text{ the best prediction is } h^*(x') = y^*$$

$$v^* = \arg\max_{y \in Y} P(y|x', D)$$

Where D is the *dataset*, and ***argmax*** returns the value of y for which the function is **maximum** (instead $\max\{\mathrm{f(x)}\}$ returns the max value of $\mathrm{f(x)}$). So, what we are going to compute is the probability distribution over Y (*codomain*):

$$P(Y|x', D)$$

We compute the *full distribution* and we want to apply a decision over a *classification* result. The powerful of this **probabilistic estimation** not only gives a prediction of the class that we want to assign to a new instance, but it also can give other information in terms of *probability distribution*. This not happened in decision tree, in that case it says yes or no, here we can have yes with a certain probability.

Also **learning** can be seen as a **probabilistic estimation**. In fact, given a dataset D, and a hypothesis space H, we can compute a *probability distribution* over H given D.

Cap 5: Bayes

From the *Bayes rules* we have that:

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

(1) • $P(h)$ = prior probability of hypothesis h
(2) • $P(D)$ = prior probability of training data D
(3) • $P(h|D)$ = probability of h given D
(4) • $P(D|h)$ = probability of D given h

(1) Represent the probability a priori of a hypothesis h before I get any dataset.

(2) Represent the **probability of extracting this dataset** from the entire distribution, independently from how the dataset is used in ML.

(3) Represent the **probability that h has been generated** from this dataset.

(4) Represent the **probability that given** one particular h I generate this dataset.

This is the main theorem that we will use. We have a typical situation in which we define a *hypothesis space*, but now we don't want to compute only the *best hypothesis*, we want a *probability distribution* over the hypotheses, so, we want to **assign to each of this hypothesis a probability value** that say how good is the hypothesis.

In general, we want the most probable hypothesis given D, this is called **maximum a posteriori hypothesis** (*MAP*) called $\mathrm{h}_{\mathrm{MAP}}$:

$$h_{MAP} = \arg\max_{h \in H} P(h|D) = \arg\max_{h \in H} \frac{P(D|h) * P(h)}{P(D)} = \arg\max_{h \in H} P(D|h) * P(h)$$

Is defined as the **argmax** [page 13 definition] over all possible *hypothesis* in the *hypothesis space*, so it's the value of h for which the probability is

maximized. We have eliminated the probability of D ($\mathrm{P}(\mathrm{D})$) because it doesn't depend on h.

If we assume that all the *hypotheses* have the same probability we can simplify and choose the **Maximum likelihood** (*ML*) hypothesis:

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

Note: $h_{ML} = \arg\max_{h \in H} P(D|h) = \arg\max_{h \in H} log(P(D|h))$ so, we can derivate to get the max.

For generate a **MAP hypothesis** we can iterate all over the *possible hypotheses*, and for each of this we can compute the *posteriori probability* [page 14 (*)] and we take the hypothesis $\mathrm{h}_{\mathrm{MAP}}$ with the highest *posteriori probability.* The problem is that we can have infinite hypothesis and we cannot enumerate all the possibilities. In some case if the space is limited and small it can be computed. So, in general a **learning process** can be the brute force of all $\mathrm{h} \in \mathrm{H}$ returning $\mathrm{h}_{\mathrm{MAP}}$ after computing all posteriori probability.

If $\mathrm{h}_{\mathrm{MAP}}$ is the most **probable hypothesis** given dataset D, given a new instance $\mathrm{x}' \notin \mathrm{D}$, $\mathrm{h}_{\mathrm{MAP}}(\mathrm{x}')$ may not be the **most probable classification**. We need to take a weighted average, and then return as value the class for which you have the highest weighted average.

Here we need to introduce the **Bayes Optimal Classifier**:

$$P(y|x', D) = \sum_{h \in H} P(y|x', h) * P(h|D)$$

For a target function $\mathrm{f}: \mathrm{X} \rightarrow \mathrm{Y}$, dataset D and a new instance $\mathrm{x}' \notin \mathrm{D}$. So, we have that:

$$y_{opt} = \arg\max_{y \in Y} \sum_{h \in H} P(y|x', h) * P(h|D)$$

This is a method in which the value of the *classification* of an instance x', given a dataset D, is given by the *argmax* of the sum of all possible hypothesis. There are proof that say no other ML method can give better result than the **Bayes optimal classifier**. But this is not a practical method when hypothesis space is large. So, this is **very powerful** and returns the *classification* with the *highest probability* but it's not practicable when H is large.

We can't use *Optimal Bayes* cause we cannot compute it. The idea now is: I can't solve the **real problem**, so under some *assumptions* the real problem can be simplified and for this **simplified problem** I can find the solution. Whenever I apply some *assumption* to simplify the problem, we have an **approximated solution**.

Naive Bayes Classifier uses **conditional independence** to approximate the solution. Two events A and B are *conditionally independent* if:

$$\Pr(A \cap B \mid C) = \Pr(A \mid C) \Pr(B \mid C).$$

Let's assume a target function $f: X \rightarrow Y$ where each instance x composed by attributes $(a_1, a_2, \ldots, a_n)$:

$$\arg\max_{y \in Y} P(y|x, D) = \arg\max_{y \in Y} P(y|a_1, a_2, ..., a_n, D)$$

Our goal is to compute the **argmax** given x and D, if x is represented as a set of attributes, we just put in the formula the values of the attributes that correspond to x. Now we introduce another instance $x' \notin D$ so we obtain for the ***Bayes rules***:

$$y_{MAP} = \arg\max_{y \in Y} P(y|a_1, a_2, ..., a_n, D) = \arg\max_{y \in Y} P(a_1, a_2, ..., a_n|y, D) * P(y|D)$$

Optimal Bayes solve this problem by applying this to space of all possible hypothesis. We want to avoid this, so we don't use *total probability* or *heuristic space of hypothesis*. In order to do this, we assume that all the *attributes* are **independent** each other given the dataset and classification value. This is a very strong assumption:

$$P(a_1, a_2, \ldots, a_n|\boldsymbol{y}, D) = \prod_i P(a_i|\boldsymbol{y}, D)$$

So, we obtain:

$$y_{NB} = \arg\max_{y \in Y} P(y|D) * \Pi_{i=1}^{n} P(a_i|y, D)$$

This is defined as the **argmax** over all the possible *classification* values of the probability of $y \mid D$ times the product of probabilities of all single values of each attribute of a new instance given y and D.

This formula can be easily *computed*, cause there is no iteration or sum over all the possible hypothesis. So, in this formula, all we need is just to estimate a very small number of probabilities, and D usually is very small. Also, the number of attributes of an instance are limited.

We can write a **Naive Bayes algorithm**, given a target function f with a *domain* X represented as *cartesian product* of n *attributes* and a *codomain* V that is a set of values, given a *dataset* D and a new instance x made by *attributes* (a_1,…,a_n):

Target function $f : X \mapsto V$, $X = A_1 \times \ldots \times A_n$, $V = \{v_1, ..., v_k\}$
data set D, new instance $x = \langle a_1, a_2 \ldots a_n \rangle$.

Naive_Bayes_Learn(A, V, D)
  for each target value $v_j \in V$
    $\hat{P}(v_j|D) \leftarrow$ estimate $P(v_j|D)$
    for each attribute A_k
      for each attribute value $a_i \in A_k$
        $\hat{P}(a_i|v_j, D) \leftarrow$ estimate $P(a_i|v_j, D)$

Classify_New_Instance(x)

$$v_{NB} = \underset{v_j \in V}{\operatorname{argmax}} \hat{P}(v_j|D) \prod_{a_i \in x} \hat{P}(a_i|v_j, D)$$

So **Naive Bayes Learn** iterates over all classification values and estimate P, for each attribute and each value of the attribute estimates probability of this value given $\mathrm{v_j}$ and given D. There are 3 nested loops. Once these terms are estimated, in the second part we can apply the estimation computed in previous step and we compute the **argmax**.

Usually a good way to make estimation is to consider number of times in which the event happens divided by the total number of times. $\mathrm{P(v_j \mid D)}$ can be estimate as numbers of elements that $\mathrm{v_j}$ as output divided by the total number of the elements in the dataset: $\hat{P}(v_j|D) = \frac{|\{< \ldots, v_j >\}|}{|D|}$

So P of a particular attribute given classification value and dataset is given by number of sample for which the attribute happens divided the number of samples for that category in the dataset: $\hat{P}(a_i|v_j, D) = \frac{|\{< \ldots, a_i, \ldots, v_j >\}|}{|\{< \ldots, v_j >\}|}$

Note that if there is a missing element (if none of the training instances with target value $\mathrm{v_j}$ have an attribute $\mathrm{a_i}$): $\hat{P}(a_i|v_j, D) = 0$ and thus $\hat{P}(v_j|D)\prod_i \hat{P}(a_i|v_j, D) = 0$

This is not good for this we can use a prior estimates (virtual samples) so there is always non-0 probability:

$$\hat{P}(a_i|v_j, D) = \frac{|\{< \ldots, a_i, \ldots, v_j >\}| + mp}{|\{< \ldots, v_j >\}| + m}$$

where

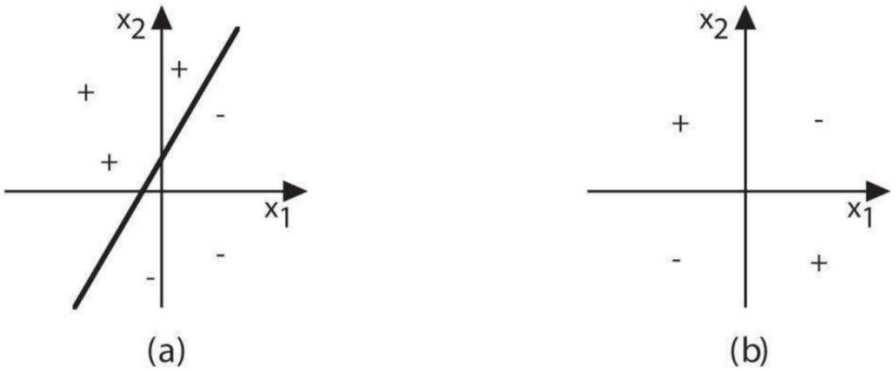
- p is a prior estimate for $P(a_i|v_j, D)$
- m is a weight given to prior (i.e. number of "virtual" examples)

Cap 6: Linear Classification

Now we are going to see another family of approaches based on casting the ML problem as *optimization problem*, whose solution is the solution of our *learning problem*.

We have a *classification problem* and instead of the previous case, we consider the input space as **continuous** in fact in the previous examples the input was formed by finite set of attributes. Output of our *target* is a *set of classes*, and this is a classification where input are real numbers and we assume that data have a special format so they can be separated by a **linear function**.

So, we have to learn a function f: $\mathrm{X} \rightarrow \mathrm{Y}$ with: $X \subset R^d$ and $Y = \{C_1, ..., C_K\}$ and with **linearly separable** data. We say that *instances* in a *dataset* (*binary*) are **linearly separable** if exist a **linear function** (hyperplane) such that the instance space is divided in <u>two regions</u>, in one you find only *positive* examples in the other only *negative* ones:



a. This is **linear separable**, we have the *dataset instances* denoted with a symbol (+ , -). This dataset is for a problem where input is *two-dimension space* and the output are *two classes* + and - . This dataset is **linearly separable**, in fact there exist at least one line that separates the space in 2 *regions*, one with only positive samples and the other with only negative ones. The solution is quite easy, in fact if I want to *classify* a new instance I simply see if it's in the region with + samples will be classified as a + else with a - . Once we choose the line that separates data this line can be used to predict classes of new instances.

b. It's not guaranteed that any dataset has this *capability*, in fact in this case given this dataset it's not possible to find any *line* that divides the space in 2 regions, one with only + and the other with only - . But now we are

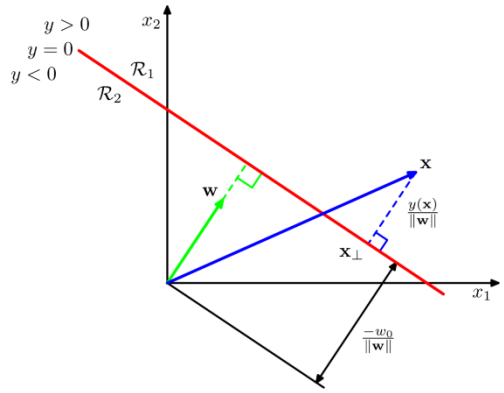
going to focus on learning a **classification function** under the assumption that the dataset is **linearly separable**.

We call Linear Discriminant function $y : X \rightarrow \{ C_1 , \dots , C_K \}$:

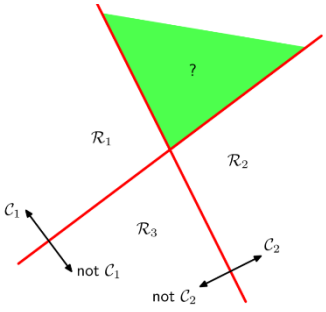
- 2 classes: $y(x) = w^T * x + w_0$;
- k classes: $y_k(x) = w_k^T * x + w_{k_0}$, $k = 1...K$;

In the first case we have **2 classes** like + and -, then the **discriminant function** can be given just by a *linear function*, where x is the input, w and w_0 are the *coefficient* of this function, and if you change this number you will have different functions in space.

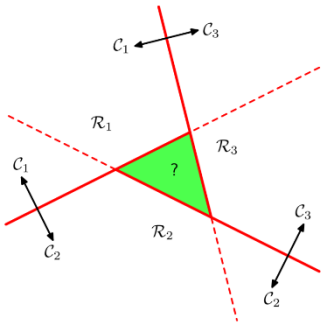
Instead in the second case, we have to classify more than 2 classes, so we need to define a **set of functions**. Model in this case is given by k functions exactly, where k is the *number of classes* of our classification problem. So, in two classes you have one function, instead in k classes you have k functions.



In a **binary classification** problem, the **geometrical interpretation** is: the red line is the function and represents the set of point x_1,x_2 for which y is 0, w is the vector perpendicular to the line, and the distance from any point x in the space and the line is: $y(x)/ ||w||$. The classification is made for $y(x) > 0$ or $y(x) < 0$.



When we have **k-classes** we can't use combination of *binary linear models*, in fact (figure) in this case we have 3 classes but there is a green region in which you cannot determinate the right classification (the "left one" says that is C_1, instead the "right one" says that that region is C_2). This is called *One-versus-the-rest classifier* in which K-1 binary classifiers: C_K vs not-C_K.

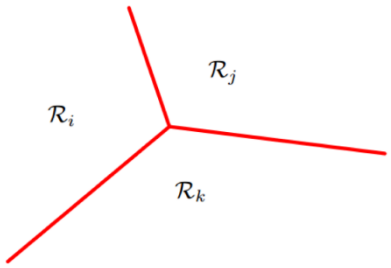


If you have a *quadratic number* of classifiers you take all possible pairs of classes and classify for example: C_1 and C_2, C_1 and C_3 and so on, there is still a problem in the middle. So, we cannot use these tricks. This example shows that we cannot transform **k-classes** classifier into a set of binary classifiers, so we need a different model. It's called *One-versurs-one classifier* in which $K(K\text{-}1)/2$ binary classifiers: C_K vs C_j.

So, for **multiple classes**, we need to define a **set of linear functions**, and in order to classify a new instance x, we can assign x to the class K for which the value $y_K(x)$ is > $y_j(x)$ for all the others:

$$y_k(\mathbf{x}) = \mathbf{w}_k^T \mathbf{x} + w_{k0} \quad \text{Assigning } \mathbf{x} \text{ to } C_k \text{ if } y_k(\mathbf{x}) > y_j(\mathbf{x}) \text{ for all } j \neq k$$

So, for new instances we compute all these functions and we classify the instance with the function K that returns the highest value.



The output of the model is a partition of the space like this in which we can easily classify new instances.

In order to make this problem simple we can use a more compact notation with vectors:

$$y(x) = \tilde{W}^T * \tilde{x}$$

$$\tilde{W} = \begin{pmatrix} w_{10} & w_{20} & \dots & w_{K0} \\ w_1 & w_2 & \dots & w_k \end{pmatrix} \quad \tilde{x} = \begin{pmatrix} 1 \\ x \end{pmatrix}$$

So, given a multi-class classification problem and a dataset D with linearly separable data, we need to determine **W** such that: $\mathbf{y}(\mathbf{x}) = \tilde{\mathbf{W}}^T \tilde{\mathbf{x}}$ is the K-class **discriminant**. We need to compute the coefficient of the matrix, and once we compute these values, we can classify new instances by applying this matrix multiplication and checking the result with the highest value.

There are 4 kinds of solution for this problem:

1. Least squares
2. Fisher's linear discriminant

3. Perceptron
4. Support Vector Machines

Let's consider a *dataset* $D = \{(x_n, t_n)_{n=1}^N\}$. , so the dataset is a set of pairs, where $\mathrm{x_n}$ is a point in our space and $\mathrm{t_n}$ is a vector of k components where k is the number of classes and this vector contains one value with 1 and all the other with 0, so it is 1 at position i when $\mathrm{x_n} \in \mathrm{C_i}$.

Least squares

We define:

$$\tilde{X} = \begin{pmatrix} \tilde{x}_1^T \\ \dots \\ \tilde{x}_N^T \end{pmatrix} \quad T = \begin{pmatrix} t_1^T \\ \dots \\ t_N^T \end{pmatrix}$$

Our dataset contains n *tuples of values* of our input and n encoding, so we can represent this information in a matrix in which the number of rows is the size of the dataset, and for each row we have one sample encoded as $\tilde{x}_1^T$ and so on. This matrix will have all 1 in the first column, in all the rows you have the values in the dataset, and this is called **sign matrix**. T is a matrix where in each row there is the encoding of the value of the *class* for each *sample*.

As our goal is to find the matrix W, we need to solve this as an optimization problem, so we need to define an **error function**. This function is defined in term of list square error, and the error is the difference between prediction of the sample in the data and the value that is in the dataset:

$$E(\tilde{W}) = \frac{1}{2} * Tr\Big\{(\tilde{X} * \tilde{W} - T)^T * (\tilde{X} * \tilde{W} - T)\Big\}$$

$$\tilde{W} = (\tilde{X}^T * \tilde{X})^{-1} * \tilde{X}^T * T$$

We want this error to be 0, and we want to find a value of W such that this error is minimum, this is a *simple quadratic problem* that can be solved with a *closed form solution*. The problem is that this solution is not robust to **outliners**, cause it find the line that is the *best average distance* from each points.

Fisher

In this case we consider two classes case, and we have to determine $\mathrm{y} = \mathrm{w^T x}$ and we want to classify $\mathbf{x} \in C_1$ if $y \geq -w_0$, $\mathbf{x} \in C_2$. Corresponding to the projection on a line determined by w. So we need to find the medium:

$$m_1 = \frac{1}{N_1}\sum_{n\in C_1} x_n \quad m_2 = \frac{1}{N_2}\sum_{n\in C_2} x_n.$$

We want to find a line that minimizes the overlap of two projections. So we need to maximize this function:

$$J(w) = \frac{w^T * S_B * w}{w^T * S_W * w}$$

with

$$S_B = (m_2 - m_1) * (m_2 - m_1)^T$$
$$S_W = \sum_{n\in C_1}(x_n - m_1) * (x_n - m_1)^T + \sum_{n\in C_2}(x_n - m_2) * (x_n - m_2)^T$$

$$w^* = -\frac{1}{S_w^{-1} * (m_2 - m_1)} \text{ maximizes } J.$$

S_B is “between class scatter”, S_W is “within class scatter”, by maxing this error function we find a solution based on maximum separation between projections of the mins of the 2 distributions rotated by a proper rotation matrix.

Perceptron

Is an **iterative method**, we have a linear combination of input dimensions with some weights, plus a constant value w_0. The output will be a function that depends on the sign of the linear combination:

$$o(x) = \begin{cases} 1 & \text{if } w^T * x > 0 \\ -1 & \text{otherwise} \end{cases}$$

So the output of this component is 1 if the linear combination of input with weight is greater than 0 or -1 otherwise. Again we need to minimize the square error (loss function). Considering that:

$$o = w_0 + w_1 * x_1 + ... + w_d * x_d$$

And this is equal to $w^T * x$, so the derivate of this error function is done with respect to each w_i, so it’s an iterative method with a gradient checking. In fact, the algorithm will just start to move to the negative gradient in order to minimize the error. The algorithm can update weights in order to adjust the classifier and this is done in an iterative way until we divide all the samples of the dataset.

$$E(w) = \frac{1}{2}\sum_{n=1}^{N}(t_n - o_n)^2 = \frac{1}{2}\sum_{n=1}^{N}(t_n - w^T * x)^2$$

With $o = w_0 + w_1 * x_1 + ... + w_d * x_d$ $(x_0 = 1)$.
Training rule:

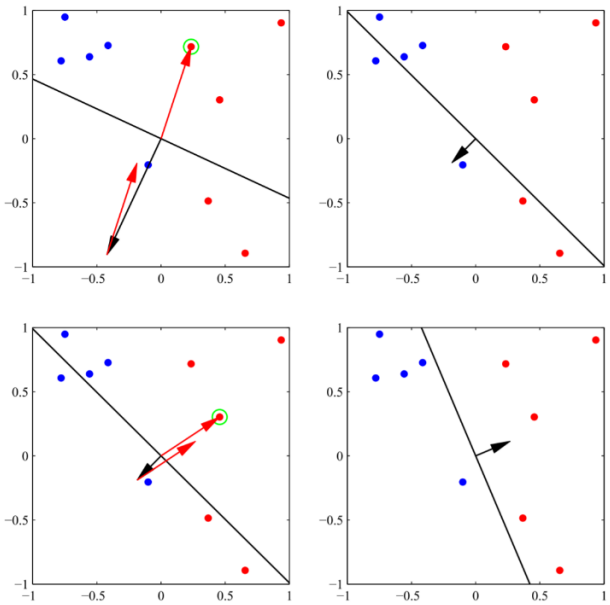
$$\frac{\partial E(w)}{\partial w_i} = \sum_{n=1}^{N}(t_n - w^T * x_n) * (-x_{i,n})$$

Updating w_i: $w_i \leftarrow w_i - \eta * \frac{\partial E(w)}{\partial w_i}$ whee η is small (like 0.05) and called learning rate.
Algorithm:

1. Initialize w' with small random values;
2. Repeat the udating w_i' procedure until termination condition (e.g. finite number of iteration or treshold on $E(w)$);
3. Output w';

----------@ Exam question ----------

The **perceptron** it's an iterative algorithm to reach the optimization problem and it works with the **linear combination** of the input dimensions with some weights plus a constant term w_0. The algorithm starts with a **random hyperplane** and he try to minimize the loss function (square error), this function is given by the difference between what I have in the dataset and the prediction that w will return on the input. For **minimize** the **loss function** I have to derivate the error respect to each w_i, so I can compute the **gradient** and I can see that the algorithm will move in the direction of negative gradient to minimize the error. So, the **weights** will be **updated** each time in an iterative way until some termination condition.

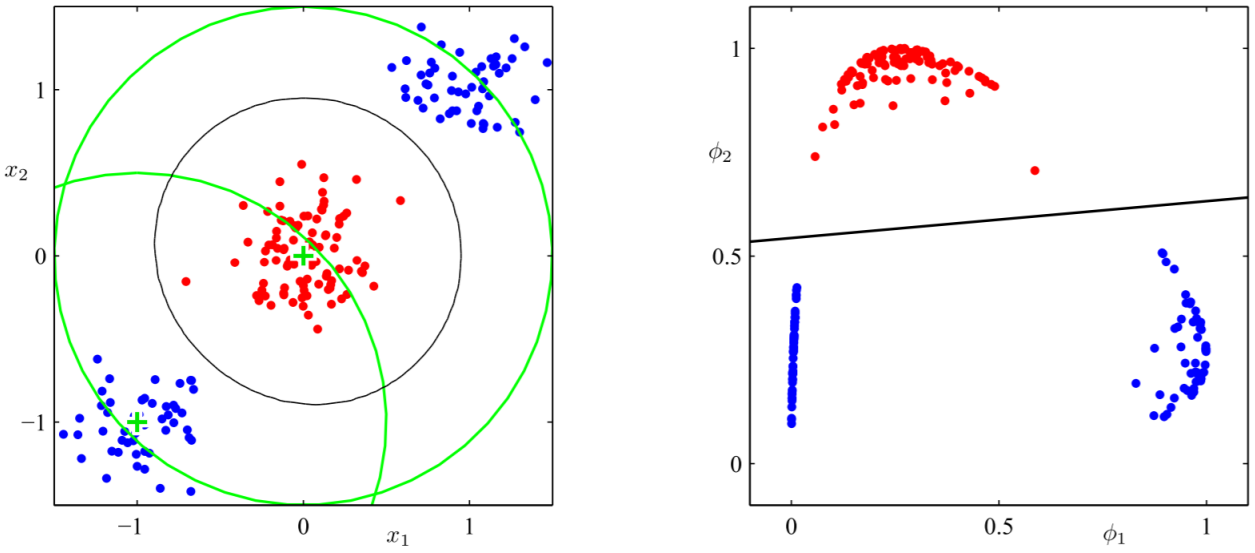


Support Vector Machine

SVM is a method in which we don't want to find a line that completely divides instances simply, but which does it in the best way. We want a line that guarantees the maximum distance between points of the two classes from it. So, we want the hyperplane that maximize the margin between two points. To compute this, we have to make a scaling operation, so we scale all these points by multiply all points by a constant term so this does not affect the solution. And we rescale in such a way that the closest point has a distance from the optimal hyperplane of 1. When we compute the optimal margin there will be 2 closest points, one for side, and these points will be at distance 1 from the optimal hyperplane.

----------@ Exam question ----------

If the dataset is **not perfectly separable** we can apply a **non-linear transformation** of the inputs, we apply a basis functions: $\phi(x)$ and with this function we obtain a dataset that is separable in the feature space ϕ



Cap 7: Probabilistic Classification

For *classification* we want to estimate the *posterior probability distribution* of a instance belonging to some class, in fact we know that our task for ML is to *predict* the class to which a new instance belongs, and we can formulate this problem as an **estimation**. There are two **probabilistic models** for classification:

- **Generative**: estimates $\mathrm{P}(\mathrm{C_i} \mid \mathrm{x})$ through $\mathrm{P}(\mathrm{x} \mid \mathrm{C_i})$ and Bayes
- **Discriminative**: estimates $\mathrm{P}(\mathrm{C_i} \mid \mathrm{x})$ from a model

These methods are similar, in fact both are based on the idea of maximizing the *likelihood*.

Generative

The idea is to compute the probability of a class given an instance by using *Bayes theorem*, and all the methods can be extended to *multiple classes*.

We find the **conditional probability**:

$$P(C_1|\mathbf{x}) = \frac{p(\mathbf{x}|C_1)P(C_1)}{p(\mathbf{x}|C_1)P(C_1) + p(\mathbf{x}|C_2)P(C_2)} = \frac{1}{1+\exp(-\alpha)} = \sigma(\alpha)$$

$$\text{with: } \alpha = \ln \frac{p(\mathbf{x}|C_1)P(C_1)}{p(\mathbf{x}|C_2)P(C_2)}$$

$$\text{and } \sigma(\alpha) = \frac{1}{1+\exp(-\alpha)} \text{ the } \textit{sigmoid function}.$$

Now we assume that $\mathrm{P}(\mathrm{x} \mid \mathrm{C_i})$ can be replaced by a **gaussian distribution**, so we get that $\mathrm{P}(\mathrm{C_i} \mid \mathrm{x})$ is given by the **sigmoid function** of this term with means and covariance of the two distributions. For 2 classes we need to find the **maximum likelihood solution**, so our goal is to find the parameters of this model, so the means of the two gaussian distributions, sigma and covariance. We just compute likelihood and then we solve the optimization problem for finding max likelihood.

Discriminative

We will estimate posterior probability directly without Bayes theorem. We consider a dataset in a transformed space, the likelihood function is the probability of receiving the output t given the input n, and the model is given by the sigmoid function of a linear combination of the input and the weights.

Cap 8: Linear Regression

Now we consider the problem of learning a function for which the **output** is **continuous**. Our *target function* now has the *output domain* given by the set of *real numbers*, or a subset of it.

We are going to learn a function $\mathrm{f} : \mathrm{X} \rightarrow \mathrm{Y}$ where:
$X \subseteq R^d$ and $Y = R$ from datatset $D = \{(x_n, t_n)_{n=1}^N\}$.

We have a dataset that is a *set of pairs* where $\mathrm{t_n}$ is a real value. We want to find a *function* such that when we give to it an input x the function returns a **real value**, *not a class already defined like before*. We need to define a *model*:

$\mathrm{y(x;w)}$ with parameters w to approximate the target function f, the linear model for linear function is:

$$\mathrm{y(x;\ w) = w_0 + w_1x_1 + \ldots + w_dx_d = w^Tx}$$

Where:

$$x = \begin{pmatrix} 1 \\ x_1 \\ \ldots \\ x_d \end{pmatrix} \quad w = \begin{pmatrix} w_0 \\ w_1 \\ \ldots \\ w_d \end{pmatrix}$$

We can use the linear basis function model to generate linear models of **non-linear functions**. We can generate a new model, as the linear combination of the coefficients w times the values that are obtained by transforming input space by any non-linear function phi:

$$y(x, w) = w^T * \phi(x)$$

Now we have a non-linear function in x but we still have a linear function in w. Since we are interested in computing w, this is still a linear model and we can apply all the methods we saw for linear models, in fact the **non-linear function** in x *does not affect the solution*.

For *minimize the error* in our model we will *maximise* the **likelihood**. Now let's define our regression problem, we consider the target value t, the values that we will obtain in the dataset are given by the value of true function plus an error. We define an *error model* in which we assume samples in dataset not perfect, and we assume that this error is gaussian (noise).

Least squares error minimization:

$$E_D(w) = \frac{1}{2}\sum_{n=1}^{N}\Big(t_n - w^T * \phi(x_n))\Big)^2 = \frac{1}{2} * (t - \Phi * w)^T * (t - \Phi * w)$$

Optimal condition: $\nabla E_D = 0$, $w_{ML} = (\Phi^T * \Phi)^{-1} * \Phi^T * t$.
Learning with stochastic gradient descent:
$\hat{w} \leftarrow \hat{w} - \eta \nabla E_n$.
So $\hat{w} \leftarrow \hat{w} - \eta * \Big(t_n + \hat{w}^T * \phi(x_n)\Big) * \phi(x_n)$.

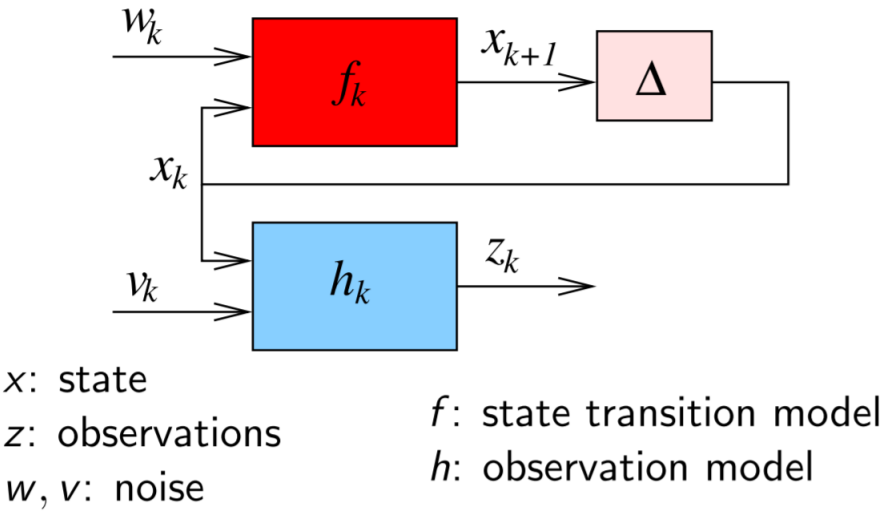
----------@ Exam question ----------

The **parameters** can be estimated in **batch mode** and in **sequential mode**. The difference is that in **batch mode** we take the **whole dataset** in order to minimize a **squared error function**, instead in the **sequential** one we update the parameters with an **SGD algorithm**, similar to the *perceptron* one, but with continuous values as t_n.

Cap 9: Markov Decision Processes and Reinforcement Learning

Reinforcement learning is the ability of an agent to learn a **behaviour**.

A **dynamic system** is a system whose **state** evolves overtime. **State** is a representation of the *information needed to model this system*.



The system evolves because there is a **dynamic transition function** denoted with f (red box), and this *transition function* is responsible to model how *states* evolve over times, how to compute *next state* given the current. And this evolution is affected by some **noises** (w,v), coming from *environment* (not modelled in the state). Sometimes the state can be observed (not always), we usually have an **observation model** that is a component able to extract *information* from the state and the outcome of the *observation* is denote with z.

When we want to define an **agent** we want that this agent is able to *drive the evolution of the state*, for example if we want an agent that is able to play chess we need an agent able to find a **sequence of states** that has some **advantages**, so we need an agent able to control the sequence of states to achieve some goal.

We have 2 possibilities for design such an AI:

- **Reasoning**
- **Learning**

Reasoning: we assume to know all information about the model (f, h) and the current state, we can compute the next state from my *current state*, and I can generate my *behaviour*, so we just predict without executing any action.

Learning: we don't have *any information* about the transition function and the observation model, so I don't know how the system evolves but I can make test. I can try, I execute action, move to next state and based on this knowledge I can learn how to do better the next time. Given *past experience* I determinate the *model*.

The **state** x contains all information needed in order to take decisions about the future, we can imagine that as a *snapshot* at a given time that you can take on the system you are analysing. The goal for the agent is to define a **function** that allows him to make decision about what to do in a given state. The agent has to compute the function: $\pi : \mathbf{X} \mapsto A$

This function is called **policy/behaviour function**, and is a function that maps states into actions, so for each state an agent has to know what is the action that has to be executed in that state. When the model is not known the agent must learn the function π.

The difference between **Supervised Learning** and **Reinforcement Learning** is in the dataset, in fact in supervised we have a dataset composed by set of pairs (input, output) of the function we want to learn, instead in reinforcement I have a dataset in which for each sequence of states I collect rewards. So, I have a numerical value that determinates how good is this action.

Supervised Learning
Learning a function $f : X \to Y$, given $D = \{\langle x_i, y_i \rangle\}$

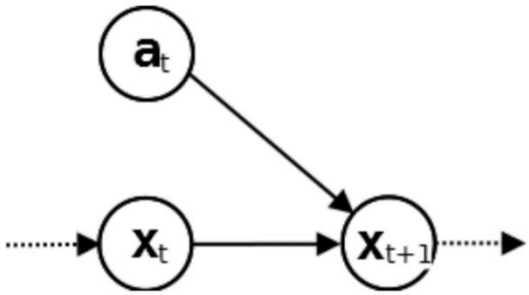
Reinforcement Learning
Learning a behavior function $\pi : \mathbf{X} \to A$, given $D = \{\langle \mathbf{x}_1, a_1, r_1, \ldots, \mathbf{x}_n, a_n, r_n \rangle^{(i)}\}$

The very interesting thing of **Reinforcement Learning** is that their algorithms are domain independent, so in order to learn how to play chess we don't need to encode in any way the rules of the game so they can learn any policy from any game.

For notation:

- X set of states;
- A set of actions;
- δ transition function;
- Z set of observations;

An important property that we are going to use is the **Markov property**, this allow a significant reduction of complexity. This property is: knowledge of current state is all we need to make predictions, we don't need to know what the history has been to reach current state, once we know it we can forget everything about the past and current state contains all the information needed to choose the best action. A **Markov process** is a process that has the *Markov property*. In some *dynamic system* the history of actions and states may be relevant, so this is an important simplification.



X_t denote any possible state that can be achieved at time t, we have another state X_{t+1} and a random variable at that is any possible action executed at time t. These variables are related each other and thanks to **Markov property** once I know X_t all the information about the past are useless, this means if I want to write Bayes network we need to put an arrow from X_t to X_{t+1} meaning that there is a probability distribution but we will now have an arrow from any previous points. Instead if you don't have any **Markov** assumption the probability distribution for reaching the next state depends on previous states. States are fully observable, and the agent can only know exactly in which state it is.

Fully observability means that the agent can understand fully the current state, it does not mean that he's able to predict next state.

We call ***Markov Decision Processes*** or **MDP** a tuple that contains the set of states that we denote with X (finite and discrete), the set of actions denoted with A (finite and discrete) and the transition function and the reward function.

A **deterministic MDP** is a MDP in which whatever the agent decides to execute an action it will end up always in the same resulting state, so the transition function is a function that given a pair $< \text{current-state} , \text{current-action} >$ return exactly one next state (always the same).

$$MDP = \langle \mathbf{X}, \mathbf{A}, \delta, r \rangle$$

- $\mathbf{X}$ is a finite set of states
- $\mathbf{A}$ is a finite set of actions
- $\delta : \mathbf{X} \times \mathbf{A} \to \mathbf{X}$ is a transition function
- $r : \mathbf{X} \times \mathbf{A} \to \Re$ is a reward function

Markov property: $\mathbf{x}_{t+1} = \delta(\mathbf{x}_t, a_t)$ and $r_t = r(\mathbf{x}_t, a_t)$
Sometimes, the reward function is defined as $r : \mathbf{X} \to \Re$

A **non-deterministic MDP** is an MDP in which whatever the agent executes an action from a state there is not anymore a guarantee what is the outcome. In fact the transition function codomain now is 2^X so is the set of all possible sets of x, this means in this case given a state and an action the transition function returns a set of possible states, but we will be able to observe next state only after the execution of the action. Also the reward function is extended in fact we have to consider the outcome of an action so is defined as: current state, action, next state and is given when I execute an action and this action brings to a next state.

Non-deterministic transitions $MDP = \langle \mathbf{X}, \mathbf{A}, \delta, r \rangle$

- $\mathbf{X}$ is a finite set of states
- $\mathbf{A}$ is a finite set of actions
- $\delta : \mathbf{X} \times \mathbf{A} \to 2^{\mathbf{X}}$ is a transition function
- $r : \mathbf{X} \times \mathbf{A} \times \mathbf{X} \to \Re$ is a reward function

There exist also ***Stochastic MDP*** in which we have a probabilistic transition, so given a state x and an action a, there is a probability distribution of reaching any next state, so it's an extension of non-deterministic state.

Stochastic transitions $MDP = \langle \mathbf{X}, \mathbf{A}, \delta, r \rangle$

- $\mathbf{X}$ is a finite set of states
- $\mathbf{A}$ is a finite set of actions
- $P(\mathbf{x}'|\mathbf{x}, \mathbf{a})$ is a probability distribution over transitions
- $r : \mathbf{X} \times \mathbf{A} \times \mathbf{X} \to \Re$ is a reward function

In an MDP we have that the states are fully observable, in fact even in presence of *non-deterministic* of *stochastic actions* the **resulting state** is not known before the execution of the action, but it is **fully observable** after the execution.

So, given an MDP, we want to find an **optimal policy**, this means to find the best action to be executed at any given time. Optimality is defined in terms of rewards, in fact I want to maximize it. Policy is a function $\pi : \mathbf{X} \mapsto A$ and for each state, $\pi(\mathrm{x}) \in \mathrm{A}$ is the optimal action to be executed in such state.

We can define the **optimization function** as the *sum of all rewards*. We give the same value to rewards obtained soon or in the *future*, but this is not good (you can make non-optimal choice) so we use a **discount factor**, a term that penalizes rewards that are in the future (an exponential factor):

$$V^{\pi}(x_1) = E\left[r_1 + \gamma * r_2 + \gamma^2 * r_3 + ...\right]$$

$$\text{where } r_t = r(x_t, a_t, x_{t+1}), \quad a_t = \pi(x_t) \text{ and } \gamma \in [0, 1] \text{ called discount factor}$$

$$\pi^* = \arg\max_{\pi} V^{\pi}(x) \forall x \in X.$$

So π^* it's the **optimal policy**, so the **optimization function** is the function for which this value is the max for all possible polices and for all possible states for which policies are applied. For infinite horizon problems, a stationary MDP always has an optimal stationary policy. Stationary policy means that the action that is chosen from the policy function is always the same.

Problem: MDP $\langle \mathbf{X}, \mathbf{A}, \delta, r \rangle$
Solution: Policy $\pi : \mathbf{X} \rightarrow \mathbf{A}$
If the MDP $\langle \mathbf{X}, \mathbf{A}, \delta, r \rangle$ is completely known $\rightarrow$ reasoning or planning
If the MDP $\langle \mathbf{X}, \mathbf{A}, \delta, r \rangle$ is not completely known $\rightarrow$ learning

Let's start with the simple MDP: **One-state MDP**. It's an MDP with only one state x_0 and we have a set of possible actions and whenever we execute an action, we come back to state x_0. So, the *transition function* just push it back to the unique state, *reward function* (r) instead assign a reward to any execution of the action. The *optimal policy* is just one particular action among all the possible actions.

$MDP = \langle \{\mathbf{x}_0\}, \mathbf{A}, \delta, r \rangle$

- $\mathbf{x}_0$ unique state
- $\mathbf{A}$ finite set of actions
- $\delta(\mathbf{x}_0, a_i) = \mathbf{x}_0, \ \forall a_i \in \mathbf{A}$ transition function
- $r(\mathbf{x}_0, a_i, \mathbf{x}_0) = r(a_i)$ reward function

Optimal policy: $\pi^*(\mathbf{x}_0) = a_i$

If r is deterministic and known $\pi^*(x_0) = \arg\max_{a_i \in A} r(a_i)$.
If r is deterministic and unknown then execute each a_i ($|A|$ executions), collect the rewards and return the best a_i.
If r is not deterministic and known $\pi^*(x_0) = \arg\max_{a_i \in A} E[r(a_i)]$.
If r is not deterministic and unknown collect many times T the rewards for actions and update a data structure at entry a_i with the average of all rewards of a_i.

When we have **more that one state**, the system will evolve and when you execute an action the system will change state. If transition function and reward function are not known the agent cannot predict the effects of its action, but it can execute them and then observe the outcome, so the **learning** now is:

1. Choose an action
2. Execute it
3. Observe the resulting state
4. Collect reward

In order to compute the policy there are 2 families of approaches: one based on estimating the value function called ***Value iteration*** and another based on estimating directly the policy called ***Policy iteration.***

The value iteration is based on estimate the value function $\mathrm{V}^{\pi^*}(\mathrm{x})$ and from it you can determinate the optimal policy:

$$\pi^*(x) = \arg\max_{a \in A} \left[r(x,a) + \gamma * V^*(\delta(x,a)) \right]$$

Where $\mathrm{r(x,a)}$ represent the reward obtained by executing the action, and the other is gamma times the best I can get in the future. But this is impossible since the two function are **unknown**.

This can be done in with an alternative method: we can use a **state action function** named **Q-function**, this is defined as the current reward plus gamma times the value of the policy from the next state:

$$Q^{\pi}(x,a) = r(x,a) + \gamma * V^{\pi}(\delta(x,a))$$

So is the expected value when executing an action a in the state x and then acting according to π. In case of **stochastic model** definition is extended as a weighted average, where the weights are given by the probability to reach the next state:

$$Q^{\pi}(x,a) = \sum_{x'} P(x'|x,a) * (r(x,a,x') + \gamma * V^{\pi}(x'))$$

If the agent learns Q the π* can be computed without knowing the two functions. The **learning rule** in this deterministic case is:

$$Q(x_t, a_t) = r(x_t, a_t) + \gamma * \max_{a' \in A} Q(x_{t+1}, a')$$

And the **training rule** is:

$$\hat{Q}(x,a) = \bar{r} + \gamma * \max_{a' \in A} \hat{Q}(x', a')$$

Where $\bar{r}$ is the immediate reward after the action and x' is the resulting state after that action. Q^ is the current learner approximation of Q. Now we are going to see an algorithm for learning called **Q-Learning Algorithm for Deterministic MDPs**: this algorithm has an entry table for each couple (x,a) and time instant:

1. Set all $\hat{Q}_{(0)}(x,a) = 0$;
2. Observe x;
3. For each time until the termination condition do
   (a) Choose an action a;
   (b) Execute it;
   (c) Observe the new state x';
   (d) Collect the immediate reward $\bar{r}$;
   (e) Set $\hat{Q}_{(t)}(x,a) = \bar{r} + \gamma * \max_{a' \in A} \hat{Q}_{(t-1)}(x', a')$;
   (f) $x = x'$;

This algorithm does not use the transition function and the reward function but it's still correct. And the **optimal policy** is:

$$\pi^*(x) = \arg\max_{a \in A} Q_{(\text{last})}(x,a)$$

The action can be chosen in two way: the agent can select an action that maximize the $\hat{Q}(x,a)$ and this is called **Exploitation**, or he can select an action with a low value of $\hat{Q}(x,a)$ and this is called **Exploration**.

Actions can be chosen in random way like in **ε-greedy strategy** where I select a random action with probability ε and the best action with probability 1 – ε, and usually *exploration* is done at the beginning (so we have an ε high) and then *exploitation*. In fact, in a learning process at the beginning there in nothing to *exploit* cause the first values depends on random execution, and ε can decrease over time.

Another strategy is the **soft-max strategy** which consist of assign higher probabilities to action with higher $\hat{Q}$ but all the actions have a probability more than 0.

$$P(a_i|x) = \frac{k^{\hat{Q}(x,a_i)}}{\sum_j k^{\hat{Q}(x,a_j)}}$$

A higher k selects an higher $\hat{Q}$ with more probability, so typically k increase with time (first exploration then exploitation).

Cap 10: Reinforcement Learning (non-deterministic)

When we are moving from deterministic to **non-deterministic** case the algorithm does not change, except for the update of the Q-table, in fact, the agent will choose an action with some strategy and then it will update the Q-function made with another formula.

Now let's consider this MDP:

$$MDP = \langle \mathbf{X}, \mathbf{A}, \delta, r \rangle$$

- $\mathbf{X}$ is a finite set of states
- $\mathbf{A}$ is a finite set of actions
- $P(\mathbf{x}'|\mathbf{x}, a')$ is a probability distribution over transitions
- $r(\mathbf{x}, a, \mathbf{x}')$ is a reward function

Rewards depends on current state and successor state, and whenever an action is executed, we are not sure about successor state cause we have a probability distribution. So, transition and rewards function are **non-deterministic.** The value function for non-deterministic is defined as expected value of the **cumulative discounted reward**:

$$V^{\pi}(\mathbf{x}) \equiv E[\sum_{i=0}^{\infty} \gamma^i r_{t+i}]$$

And the **optimal policy** will be:

$$\pi^* \equiv \underset{\pi}{\operatorname{argmax}}\, V^{\pi}(\mathbf{x}), (\forall \mathbf{x})$$

In the deterministic case Q is given by reward function plus gamma times the best I can do in the future, here Q is *defined as the expected value of this quantity* so we can write that Q for **non-deterministic** is the expected value of reward plus gamma time the best I can do in the future. So is given by the best I can do from any state with the probability of reaching that state:

$$Q(\mathbf{x}, a) = E[r(\mathbf{x}, a)] + \gamma \sum_{x'} P(\mathbf{x}'|\mathbf{x}, a) V^*(\mathbf{x}')$$
$$= E[r(\mathbf{x}, a)] + \gamma \sum_{x'} P(\mathbf{x}'|\mathbf{x}, a) \max_{a'} Q(\mathbf{x}', a')$$

The **optimal policy** is:

$$\pi^*(\mathbf{x}) = \underset{a \in A}{\operatorname{argmax}}\, Q^*(\mathbf{x}, a)$$

The **training rule** is:

$$\hat{Q}_n(x, a) = \hat{Q}_{n-1}(x, a) + \alpha * \left(r + \gamma * \max_{a'} \hat{Q}(x', a') - \hat{Q}_{n-1}(x, a)\right)$$
$$\text{With } \alpha = \alpha_{n-1}(x, a) = \frac{1}{1 + visits_{n-1}(x, a)}$$

When there are multiple states, you need to consider also the best I can do in the future. The difference from the deterministic one is that this term contains the immediate rewards plus gamma time the best I can do in the future. (*) this is the **estimation** of what I can get, so is the actual reward plus gamma times what I could get in the future, (**) instead is what I believed before executing an action on x. So, the difference is the error between my estimation and what I observe after the execution of the action.

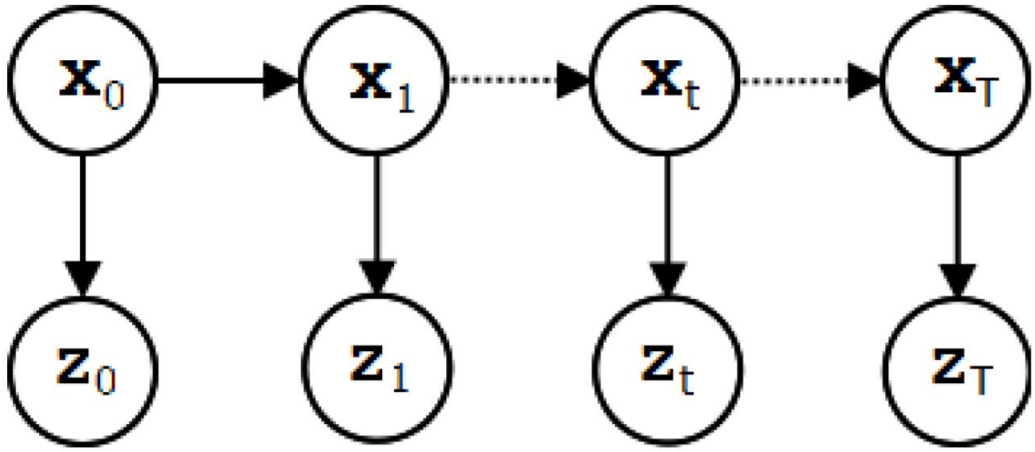
In non-deterministic $\hat{Q}_{n+1}(x, a) \geq \hat{Q}_n(x, a)$ is not valid. There are other algorithms for non-deterministic learning like **Temporal Difference,** where instead of only looking of what happens in next state I want to try to look in more than one state, so, instead of computing one step time difference I can compute n-step time difference, or **SARSA** a variant of Q-learning in which instead of trying to compute the best I can do with optimal policy I just compute the best I can do with current policy, so instead of considering max of all possible actions on Q-table I just choose an action without using the max operator.

We can even do Reinforcement learning with **Policy iteration**, in which we use directly π instead of $\mathrm{V}(\mathrm{x})$ or $\mathrm{Q}(\mathrm{x}, \mathrm{a})$, so with these methods we can overcome some difficulties in value iteration but they are more complicated to execute.

In fact one big problem is that size depends on number of states that generally is very high. These methods are based on hypothesis of generating policy in a parametric form, so $\pi_\theta(\mathrm{x})$ will be a **parametric representation** of the policy function and *p(θ)* will be the policy value. The RL algorithm in this case has simply to choose parameters in order to compute π_θ and then test it.

In the **Policy Gradient Algorithm** we compute the gradient of the policy in a local interval with respect to current point and then move in order to improve value of this policy. So, starting from a random policy we change parameters in order to move to a better different one.

Cap 11: Hidden Markov Models



The **Hidden Markov model** is a model useful to describe a system in which state are not observable. We have still a **Markov chain**, that it's just an assumption that current state contains all information needed to understand, but now we don't make any assumptions about the *fully observability* of the states, in fact agent cannot look into the states and cannot understand exactly the *current configuration*. For each state we have a dependency with the observation. In fact states x_t are discrete and **non-observable** and the observations z_t can be **discrete** or **continuous**, and controls u_t are not present so evolution is not controlled by our system. So the interest in this model is to understand the state in which we are.

We want to solve the **state estimation problem**, we can represent this HMM like this:

$\text{HMM} = \langle \mathbf{X}, \mathbf{Z}, \pi_0 \rangle$

- transition model: $P(\mathbf{x}_t|\mathbf{x}_{t-1})$
- observation model: $P(\mathbf{z}_t|\mathbf{x}_t)$
- initial distribution: π_0

State transition matrix $\mathbf{A} = \{A_{ij}\}$ $\quad A_{ij} \equiv P(\mathbf{x}_t = j|\mathbf{x}_{t-1} = i)$

Observation model (discrete or continuous): $b_k(\mathbf{z}_t) \equiv P(\mathbf{z}_t|\mathbf{x}_t = k)$

Initial probabilities: $\pi_0 = P(\mathbf{x}_0)$

We have **set of states** X, a **set of observations** Z, and an **initial probability distribution** for the initial state. Then we have a **transition model** that is a probability distribution that denote dynamic evolution of the system (like MDP

but without actions). The **observation model** is the probability of an observation given a state.

When we consider a finite set of states, we can consider the transition model like a matrix called **transition matrix of the HMM**, this has a size of $\mathrm{n}^*\mathrm{n}$, where n is the number of states and each component contains the probability distribution of moving from state i to state j.

Most of the solution of this problem are based on the solution of the **chain rule,** this says that the *joint probability* of a model is given by the product of all the terms computed considering probability of one random variable condition only to the direct parents:

$$P(\mathbf{x}_{0:T}, \mathbf{z}_{1:T}) = P(\mathbf{x}_0)P(\mathbf{z}_0|\mathbf{x}_0)P(\mathbf{x}_1|\mathbf{x}_0)P(\mathbf{z}_1|\mathbf{x}_1)P(\mathbf{x}_2|\mathbf{x}_1)P(\mathbf{z}_2|\mathbf{x}_2)\dots$$

We are interested in two kind of problems:

1. **Filtering:** is the estimation of current state given all the observations we have so far, this is made:
$$P(\mathbf{x}_T = k|\mathbf{z}_{1:T}) = \frac{\alpha_T^k}{\sum_j \alpha_T^j}$$
2. **Smoothing:** is the estimation of past states given other observations, this is defined:
$$P(\mathbf{x}_t = k|\mathbf{z}_{1:T}) = \frac{\alpha_t^k \beta_t^k}{\sum_j \alpha_t^j \beta_t^j}$$
Alpha and *beta* are forward and backward steps in time.

In the **HMM** we don't know *transition function* and *observation model* so in order to learn from this model we need to estimate these parameters and then we can apply algorithms. There are 2 different cases:

1.States can be observed at training time:

$$A_{ij} = \frac{|\{i \to j \ transitions\}|}{|\{i \to * \ transitions\}|}$$

$$b_k(v) = \frac{|\{observe \ v \wedge state \ k\}|}{|\{observe \ * \wedge state \ k\}|}$$ *model* can be estimated with statistical ; we look the transition from i to j divided

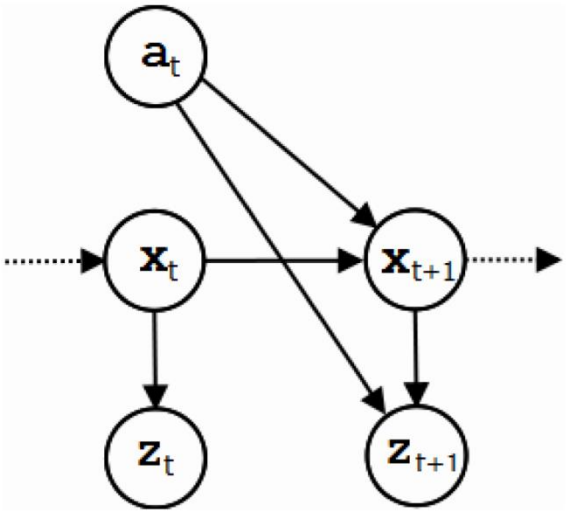
by number of times we look to other transition from I, b_k is the same but for the observation v.

2.States cannot be observed at training time:

This means that the system is not observable, but it's still possible to estimate parameters with methods based on **expectation maximization**. So, we can compute a **local maximum likelihood** with an EM method (Baum-Welch algorithm)

Each of these models make some simplification, **MDP** say that states are fully observable, **HMM** just avoid controlling system. If we want to put together all this so controlling the evolution of the system and estimate the state without making assumption on full observability, we can use a more realistic agent called **POMDP**.

This is a very simple model, that combines MDP and HMM:



Now we have **actions** that controls the evolution of the state, but states are not *fully observable* so we can observe them with the **observation**.

$$POMDP = \langle \mathbf{X}, \mathbf{A}, \mathbf{Z}, \delta, r, o \rangle$$

- $\mathbf{X}$ is a set of states
- $\mathbf{A}$ is a set of actions
- $\mathbf{Z}$ is a set of observations
- $P(\mathbf{x}_0)$ is a probability distribution of the initial state
- $\delta(\mathbf{x}, a, \mathbf{x}') = P(\mathbf{x}'|\mathbf{x}, a)$ is a probability distribution over transitions
- $r(\mathbf{x}, a)$ is a reward function
- $o(\mathbf{x}', a, \mathbf{z}') = P(\mathbf{z}'|\mathbf{x}', a)$ is a probability distribution over observations

We just put together the two models, so we have a *set of states*, a *set of actions* and a *set of observation*, an *initial probability* distribution. We have a *probability distribution over the transition* and a *reward function* like in MDP, and *probability distribution over the observations* so it's a combination of items of the two model (observation over the state and action). **POMDP** is the only model that can have actions used by the agent to get knowledge.

So, we want the agent to see the *behaviour*, a solution is still the *policy*, but we can't use the definition used for the MDP, in fact a policy in MDP is a *mapping* from states to actions, but here we don't know the state we can only estimate it. We introduce the **belief state** an estimation of current state, so we can make a process from which we can compute *belief state* and use it as an input to our decision-making approach. The *belief state* is the *probability distribution* over the *current state*, we call b a function that denotes probability of being in any state, so given the current state x (not known) I can compute a probability distribution over the states:

Belief $b(\mathbf{x})$ = probability distribution over the states.

POMDP can be described as an MDP in the belief states, but belief states are infinite.

- **B** is a set of belief states
- **A** is a set of actions
- $\tau(b, a, b')$ is a probability distribution over transitions
- $\rho(b, a, b')$ is a reward function

Policy: $\pi : \mathbf{B} \mapsto \mathbf{A}$

One way to solve **POMDP** is to model states in terms of *probability distributions*, by defining the state of belief states and build an *MDP* that has the set of belief states, sort this *MDP* and compute the **policy**. The problem is that set of belief states is *continuous*.

Given current **belief state** b, an action a, and an observation z' observed after the execution of a we can compute the next belief state: b'(x'):

$$
\begin{aligned}
b'(\mathbf{x}') &= \frac{P(\mathbf{z}'|\mathbf{x}', a) \sum_{\mathbf{x} \in \mathbf{X}} P(\mathbf{x}'|b, a, \mathbf{x}) P(\mathbf{x}|b, a)}{P(\mathbf{z}'|b, a)} \\
&= \frac{o(\mathbf{x}', a, \mathbf{z}') \sum_{\mathbf{x} \in \mathbf{X}} \delta(\mathbf{x}, a, \mathbf{x}') b(\mathbf{x})}{P(\mathbf{z}'|b, a)}
\end{aligned}
$$

We can define the **value function** and from this we can define the **optimal policy** in terms of optimal value function:

$$V(b) = \max_{a \in \mathbf{A}} [\sum_{\mathbf{x} \in \mathbf{X}} b(\mathbf{x}) r(\mathbf{x}, a) + \gamma \sum_{\mathbf{z} \in \mathbf{Z}} P(\mathbf{z}|b, a) V(b_{\mathbf{z}}^{a})]$$

We know that *belief states* are infinite, so we need to **discretize** them, and there exist several methods to do this. There are methods to approximate it with a linear function, or other methods that make a partition of the interval and for each interval they define a linear function. This is solving multiple linear regression problems.

In some situation in which the observation are discrete we can use a **policy tree**, so, instead of *mapping* states you can represent the *history* of observations in a *tree*. At each level we have an action and the set of all possible observation in the branches. So, we can go from root to a leaf, each time we choose the action to do. At the end the trace will be given by the path of actions and observations.

Cap 12: Kernels methods

So far, the object where represented as fixed-length feature-vector $\mathbf{x} \in \mathbb{R}^M$ or $\phi(\mathbf{x})$ but what if the object has variable length or infinite dimension? **Kernel methods** are useful to introduce *non-linear function* in linear models. We consider learning a function where X is a set of instances, now we are considering the situation in which each *instance* is represented as a **finite vector of values**, this set may be infinite, but each single instance is finite. In the case of each single instance infinite we can't write a *model*.

The idea is not to represent *instances*, but only a **function of the instances**, so we want to remove all places in which X appears and replace it with a **kernel function**, that is a function defined on *cartesian product* that measure the similarity of 2 *instances*. So, given 2 instances in the input spaces the **kernel function** returns a real value (usually greater than 0). Kernel function is any function that maps a pair of instances into real numbers.

Kernel function: a real-valued function $k(\mathbf{x}, \mathbf{x}') \in \mathbb{R}$, for $\mathbf{x}, \mathbf{x}' \in \mathcal{X}$, where $\mathcal{X}$ is some abstract space.

Typically k is:

- symmetric: $k(\mathbf{x}, \mathbf{x}') = k(\mathbf{x}', \mathbf{x})$
- non-negative: $k(\mathbf{x}, \mathbf{x}') \geq 0$.

For any linear model with a kernel k: $y(x, \hat{w}) = \sum_{i=1}^{N} \alpha_i * k(x, x')$.

The solution is: $\alpha = (K + \lambda * I_N)^{-1} * t$.

Where K is the **gram matrix**. This solution is general for any kernel. In fact if we consider a model in which we use kernel we can compute a solution in this way. So, this is a very powerful tool and we can generalise linear model and we have the kernelized version of the linear model. So, we obtain a linear model of a non-linear function.

$$K = \begin{pmatrix} k(x_1, x_1) & \dots & k(x_1, x_N) \\ \dots & \dots & \dots \\ k(x_N, x_1) & \dots & k(x_N, x_N) \end{pmatrix}$$

Gram Matrix

There is **kernel trick** that says: if input vector x appears in an algorithm only in the form of an **inner product** you can replace the inner product with some

kernel. This can be applied to any x even if it has infinite size, and with this trick you can extend a linear model.

If Iocchi ask “what kind of kernel you would use?”: no don’t say *fottiti*, the answer is **RBF**. This combination of letters gives you good points.

If you have a dataset **not linearly separable** you can’t use linear kernel, so I would use a not-linear kernel like RBF so that he makes a coordinate transformation in the input and so we obtain a linearly separable dataset on which we can apply the various algorithms.

Cap 13: Instance based Learning

The idea is to introduce a set of methods that allow to define **non-parametric models**. So, far we only consider *parametric models*, and the solution of our problem is given by finding a set of parameters for a model.

In a **parametric model**, a model has a fixed number of parameters. Remember that the size of the model does not depend on the size of the dataset. Instead in the non-parametric model the number of parameters grows with amount of data, so they are models for which we can not say in advance how many parameters of the model I have. So, if Iocchi ask how many parameters are in instance based learning you can say that is not possible to define the number of parameters cause it depends on the dataset.

Instance based Learning is an example of non-parametric model.

In the **K-nearest neighbours** we have a fixed dataset and I want to classify a new point (new instance) as follows:

1. Find K nearest neighbors of new instance **x**
2. Assign to **x** the most common label among the majority of neighbors

So, the *likelihood* of class c for a new instance x is:

$$p(c|\mathbf{x}, D, K) = \frac{1}{K} \sum_{i \in N_K(\mathbf{x}, D)} \mathbb{I}(y_i = c)$$

The problem is that I need **storage** for saving the whole dataset and it's not easy to compute for all the distances, so the nearest neighbour's problem can be **kernelized** as follow:

$$\|\mathbf{x} - \mathbf{x}_i\|^2 = \mathbf{x}^T\mathbf{x} + \mathbf{x}_i^T\mathbf{x}_i - 2\mathbf{x}^T\mathbf{x}_i$$

So instead of using a distance function we can use a kernel function to define what are the instances that should influence the prediction of this instance.

Let's assume now to have a **regression problem**:

$$f : X \mapsto \Re \text{ with data set } D = \{(x_i, y_i)_{i=1}^N\}$$

Given a new instance x we have that:

$$\hat{f}(\mathbf{x}) = \sum_{i \in N_K(\mathbf{x}, D)} y_i k(\mathbf{x}, \mathbf{x}_i)$$

We look in the dataset the closest instances with the new one we want to predict and then we do some computation to return a prediction based on local samples.

So, in **K-NN** you take the k samples in the dataset closer to the instance that you want to classify, in regression we consider all these values and we fit constant model.

Cap 14: Artificial Neural Networks

We will focus now on the idea of learning with the **Artificial Neural Networks (ANN)**, the neural networks are parametric models in which we define an error function and we find the minimum of this function. There are many terms that can create some confusion. ANN represent all possible ways in which we create a network of units that are connected, so it's a family of networks. After we will focus on **Feedforward Neural Network (FNN)**, that is a NN with a special constrain on the connection between nodes of the network.

All these families are parametric models used for *classification* and *regression*, in genal NN are used for approximating a function and for this they are called *function approximation*.

The problem is the same we seen so far, we have a **learning problem** in which you want to **estimate a function** and the output can be a finite set or real numbers. So, we will define a **parametric model** for learn parameters theta that approximate the function:

Goal:

Estimate some function $f : X \to Y$, with $Y = \{C_1, \ldots, C_k\}$ or $Y = \Re$

Data:

$D = \{(\mathbf{x}_n, t_n)_{n=1}^N\}$ such that $t_n \approx f(\mathbf{x}_n)$

Framework:

Define $y = \hat{f}(\mathbf{x}; \boldsymbol{\theta})$ and learn parameters $\boldsymbol{\theta}$ so that $\hat{f}$ approximates f.

The NN take inspiration from **neurons** of the brain, so there is a ***processing unit*** that receives some information from other units and then generate an **output** that will be sent to other units.

The idea of **FNN**, is that we have only links that go from 1st level called input level to the last level called output level. **Feedforward** means that the signal proceeds only in one direction (without loops). We also assume that we have a **layered architecture** in which we have input layer, 1st layer … , output layer. We want to constraint that each unit is connected only with next layer and we don't have any connection between non-adjacent layer.

So FNNs are **chain structures** and the length of the chain is the **depth** of the network and the final layer is called **output layer**. NN can do automatically something that we do manually in other tasks.

In the **XOR problem**, a classification problem, we have in input 2 Boolean variable (0,1) and in output a single Boolean variable that is 1 only there is exactly one element with 1, (00,01,10,11). This is not a *linearly separable dataset* so it can't be solved with *linear model*. This can be solved through NNs.

For the **depth** (how many hidden layers), there is the **Universal approximation theorem** that say that a NN with only one hidden layer, but wide enough (so with enough hidden units) can approximate any function with arbitrary precision. The **width** "enough" usually is exponential with the size of the input and for continuous is even bigger. Increasing of parameters does not always lead to a better NN. A **deep FFN** has at least 3 hidden layer.

We will distinguish 3 cases, and for each case we have a different **cost model** and a different **activation function** for the output layer:

1. Regression, **linear** output unit, mean squared error cost function
2. Binary classification, **sigmoid** output unit, binary cross-entropy
3. Multi-class classification: **softmax** output unit, categorical cross entropy

----------@ Exam question ----------

Linear regression squared error: $E(\theta) = \frac{1}{2}\sum_{n=1}^{N}(t_n - \theta^T * x_n)$

Sigmoid error: $E(\theta) = -ln\ P(t|x)$

Softmax error: $E(\theta)_i = ln \sum_j exp(\alpha_j) - \alpha_i$

For the *hidden units* we have to do many experiment and intuition to know which is the best, in fact predicting which **activation function** will work best is usually impossible. In general, in all the hidden units you can put **RELU**, that is not linear, and it is very easy to compute it and to optimize.

----------@ Exam question ----------

Hidden activation functions:

Relu activation function: $g(\alpha) = max(0, \alpha)$

Sigmoid activation function: $g(\alpha) = \sigma(\alpha)$

Tangent activation function**:** $g(\alpha) = tanh(\alpha)$

Output activation function:

Linear regression activation function: $y = \mathbf{W}^T\mathbf{h} + \mathbf{b}$

Sigmoid activation function: $y = \sigma(\mathbf{w}^T\mathbf{h} + b)$

Softmax activation function: $y_i = \mathrm{softmax}(\alpha)_i$

Units in **sigmoid** saturates only when it gives the correct answer. Units in **Softmax** saturates only when there are minimal errors. Units in **regression** do not saturate.

In NNs the information flows from input to output through the network. To train the network we need to compute the gradients with respect to the network parameters, and this is done by the **back-propagate algorithm** that in an algorithm that computes the gradient of a network.

In the 1st phase, the **forward phase,** it computes the *prediction* of the *current network* with respect to some input and compares the *output* of the *network* with the *output* of the *training set* and compute the *loss function*. In the 2nd phase, the backward phase, the cost computed is **back-propagated** from the output to the input to make an estimation.

Once we have the ***gradient***, we want to find some local minimum, this is the crucial part of the learning, and there exists several *learning algorithms*.

Regularization is an important feature to reduce to reduce the overfitting, that makes parameters smoother. Another way to reduce the overfitting is to *generate additional data* and include them in the dataset (**data augmentation**), for example we can add noise, or by applying data transformation (image rotation, blur ecc..). There are also other approaches: the **Early Stopping** in which we stop iterations early to avoid the over fitting, **Parameter sharing** in which we constrain some parameters to be the same, or **Dropout** in which we randomly remove network units so we use different subset of network units at each iteration.

----------@ Exam question ----------

If we have a **regression problem** with an ANN with a target function: $R^m \rightarrow R^n$

Let's say that we have x hidden units, so the number of hidden parameters is calculated as:

Connections from input to hidden units + hidden units + connections from hidden to output + output units, so we have:

$$m * x + x + x * n + n$$

Back-propagation is affected by **Local minima** and this can be resolved with the momentum. In fact, with this the momentum, you avoid "finding" a local minimum where the error function can no longer to be differentiable.

Back-propagation is not affected of **overfitting**, in fact is not a training algorithm, is only a method for compute the gradients. The model can be affected of overfitting and there are several methods for avoid it like: early stopping, in which you stop iterations early, parameter sharing, in which you constrain on having subset of model parameters to be equal, or dropout in which you randomly remove network units with some probability alpha.

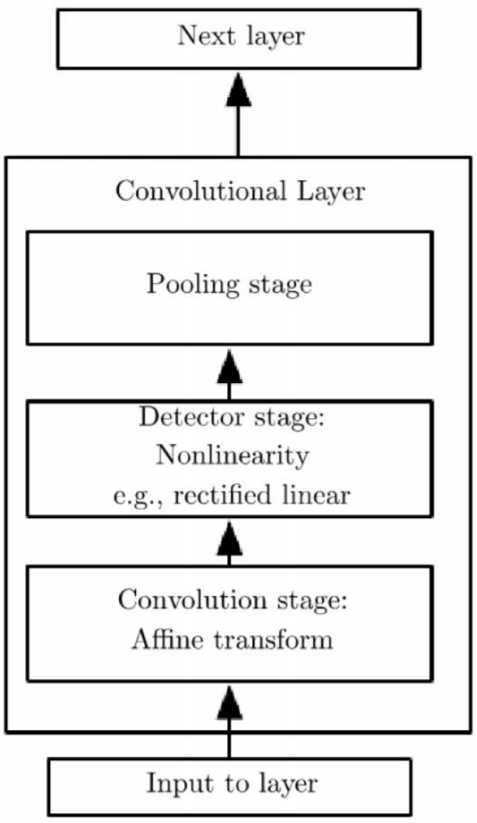
The **stochastic gradient descent** differs from the **back-propagation** cause it use mini-batch of the dataset, and modify the parameters based on the gradient and the learning rate.

Cap 15: CNN

We will see now how to apply **neural networks** to **input signals** like video, images, audio. We could have a pre-processing to digitalize the signal, in fact the signals are a numerical representation of physical quantities. **Convolution** or Cross-correlation is an operation used in these types of networks and consist of multiplying images with some *kernels* in order to do some *transformation*. We want to use this concept to define a **layer** that uses *convolution* as operation, so we want to learn the parameters of the *kernels* in order to transform the image in another image, and this operation will contribute to *classify correctly* out dataset.

A **CNN** is a *feedforward neural network* in which at least one layer is a **convolutional layer**. Usually we have 3 components, a **convolutional stage** in which we apply a *convolutional filter* (one or more filters), then an **activation function** (usually *non-linear function*) and then a **Pooling stage**:

In the **convolutional stage** w.r.t fully connected network for improving the performance and for reducing the *overfitting* are used *sparse connectivity* and *parameter sharing*. **Sparse connectivity** means that only a subset of units is connected to the input, in fact usually the *kernel* is much smaller that input, so the output depends only on few inputs. **Parameter sharing** instead is a method of a *CNN* in which the values of the *kernel* will be share through all the instances of the input, so some values are constrained to be the same of some other one.



Once we have done convolution, we have the **Detector Stage** in which we apply some *non-linear transformation* to the output of the *convolution stage* like *Relu* or *tanh*.

At least we have the **Pooling Stage**, in which we apply the *pooling operation*, this is usually used to reduce size of the layer or to introduce some invariants with respect to translation or also for scaling and rotation. In fact, the system must be *robust* to translation or small transformation. Is an operation similar to a *kernel*

operation but here the difference is that we will not train the values (in kernels we want to learn values), in *pooling step* we just want to perform a fixed operation. *Kernel operation* is a multiplication of its number with the values of the layer, and you have to find “good numbers” in the kernel. **Pooling** is an operation for which you don’t want to learn anything.

----------@ Exam question ----------

The **width** of the output of each layer of a **CNN** is calculated as:

$$w_{output} = \frac{w_{input} - w_{kernel} + 2 * padding}{stride} + 1$$

The **height** is:

$$h_{output} = \frac{h_{input} - h_{kernel} + 2 * padding}{stride} + 1$$

The **third dimension** is the number of the **feature maps** of the layer.

The number of parameters for a convolutional layer (in the pooling layer is 0 cause there are no parameters you can learn in pooling), is calculated as:

Parameter of the layer = (n*m*l + 1)*k

Where the Kernel “size” = n*m (es. 5x5, 3x3)

Number of feature maps in input in the layer = l

Number of feature maps in output of the layer = k

Cap 16: Unsupervised Learning

We have many situations in which the data that we want to analyse have some specific features that we want to find. **Unsupervised Learning (UL)** is learning without a teacher, so we have data, but we don't have labels on these. So, we just have input *dataset* available $\mathrm{D} = \{\mathrm{x_n}\}$ but we don't have any *target values*. Since we don't have labels, we cannot model this problem like we did for supervised learning so we cannot use *classification* or *regression*, in fact in these methods we have pairs of input and output. Sometimes, **UL** can be merged with *supervised learning* to provide results.

We have to cluster data depending on their characteristic, so we have to find multiple classes from data. **Unsupervised learning** *algorithm* determine *mixed probability distribution* from data, and we can calculate it:

Gaussian Mixture Model (GMM)

Mixed probability distribution P formed by k different Gaussian distributions

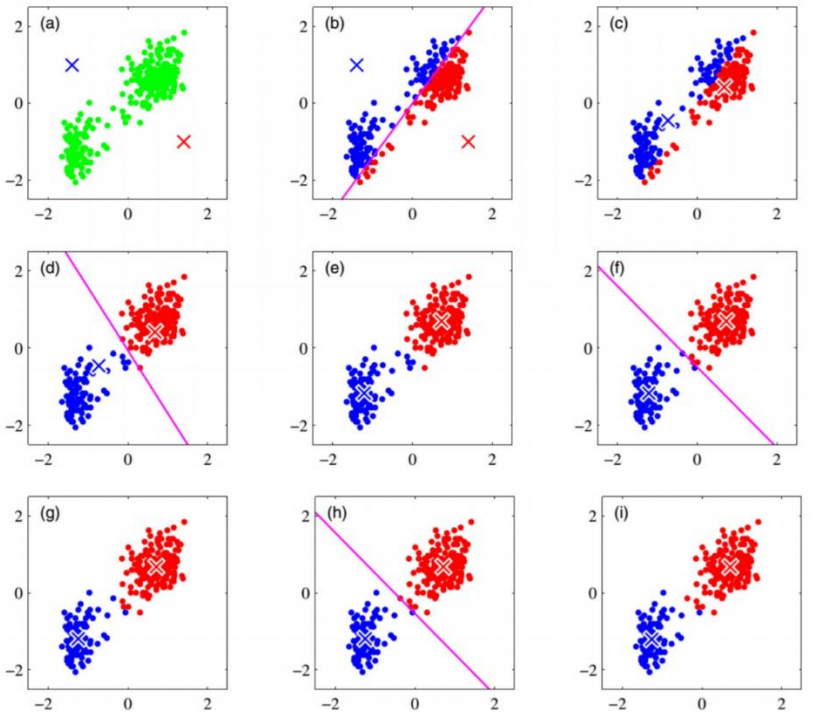
$$P(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

- $\boldsymbol{\mu}_k$, mean
- $\boldsymbol{\Sigma}_k$, covariance matrix

K-means has the goal of computing the means of the gaussian distribution, given a dataset. The input is a set of samples and a value k, so we have to tell to the system how many gaussian are present or how many clusters we want to form and the output of the algorithm will be the means of the distributions (without considering weights and covariance).

K-means it's an iterative process in which:

1. Begin with a decision of the *value of k* (number of cluster)
2. Put any **initial partition** that classify the data in k cluster, so you can take random training samples, or you can assign the first k elements of the dataset as training samples, and you assign the remaining (N-k) training samples to cluster with the nearest **centroid** (means of the distribution) and after each assign recompute the centroid.
3. Take each sample in sequence and compute its distance from the centroid of each cluster, if a sample is not in the closest centroid you switch it and you update the centroid of the two clusters involved
4. Repeat 3 until we obtain the convergence (the centroid has stabilized)



This method has several **drawbacks**, in fact you have to tell in advance how many **clusters** you want to create, and this is something that you are not sure you can predict in advance, it's not robust to **outliners**, and the result is a **circular cluster shape** cause it's based on distance.

We introduce a different **GMM** in which we have a dataset $D = \{(x_n)_{n=1}^N\}$ and each point $\mathrm{x_n}$ is associated to the corresponding variable $\mathrm{z_n}$ which is unknown. This variable $\mathrm{z_n}$ is called **latent variable**.

Let's define the posterior

$$\gamma(z_k) \equiv P(z_k = 1|\mathbf{x}) = \frac{P(z_k = 1)\, P(\mathbf{x}|z_k = 1)}{P(\mathbf{x})}$$

$$\gamma(z_k) = \frac{\pi_k \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}$$

Note:

π_k: prior probability of z_k

$\gamma(z_k)$: posterior probability after observation of $\mathbf{x}$.

So **Gaussian Mixture Model** is as model, and you have k-gaussian distributions and in order to get a probability distribution you have to sum all the distributions and you assign to each one a weight, so at the end you obtain a probability distribution.

Then we have the **Expectation Maximization (EM)** that is a generalization of the k-means algorithm, where I have to estimate all the *parameters* not only the means like before. EM is an *algorithm* that let you estimate the parameters of the *GMM model*, *means*, *covariance* and *weight* of each gaussian distribution included in the mixture model. In *k-means* we consider only mean and we use it to cluster the elements of a dataset, so **EM** is more general cause it considers also the *covariance*. It uses the **maximum likelihood** formulation. It can be done in an *iterative method* in which we have 2 steps, an **E step** in which we compute the *posterior probability* given the *parameters* and an **M step** in which we estimate the parameters of the model given the posterior probability, and we repeat these steps many time.

The **big difference** from the **k-means** is that in the k-means we assign each point to a cluster, here instead we assign a probability distribution to each point, so for example, maybe a point is 75% blue and 25% red.

EM is still a **greedy approach**, so we have no guarantee that this solution will converge to a global maximum likelihood, in fact this depends on initialization phase.

Cap 17: Dimension Reduction

Any **images dataset** has this issue, that the number of **dimensions** is given by the size of the images. If you take a dataset of digits of an image 64x57 (rows, columns) the input space has a **dimensionality** given by the product of these two values so is more or less 4000, that is our input x. So, in principle we have a set of all possible instances that have a size of 4000 dimensions. So if we consider all possible combinations that we will have in a dataset with 4000 dimensions we will have many images that are no meaningful. If we want to classify only images of a cat or a dog there are much less possible combination of color pixels in an image. So, the idea is to try to analyse the input to understand the real dimensions that make the variability of our dataset.

If you consider a dataset in which you only rotate one symbol, then you can plot only in 2 dimensions the variability. If you take a dataset and we operate a translation and a rotation of each digit of the dataset in terms of degree of freedom transformation we have to consider only 3 possible changes, translation in 2 direction and 1 rotation.

Dimensionality reduction is the process of reducing the number of random variables under consideration, by obtaining a set of principal variables. It can be divided into feature selection and feature extraction.

PCA or **Principal Component Analysis** is a technique for various tasks as: dimensionality reduction, data compression, data visualization and feature extraction.

It works on a condition that while the data in a higher dimensional space is mapped to data in a lower dimension space, the **variance** of the data in the lower dimensional space should be maximum.

The **covariance matrix** of the data is constructed and the eigenvectors on this matrix are computed. The **eigenvectors** that correspond to the largest eigenvalues (the principal components) can now be used to reconstruct a large fraction of the variance of the original data. Moreover, the first few eigenvectors can often be interpreted in terms of the large-scale physical behaviour of the system.

----------@ **Exam question** ----------

In order to **represent x** on the **M** components I need to project each point on u_1,u_2 , … till u_M using the formula $u_1^T x$, $u_2^T x$, and so on. In order to be coherent with what **PCA** says these vectors must be the vectors that **maximize** the **variance**, that means that I will first compute the **mean** of the x and after I compute the **covariance matrix** S on the whole dataset, and I will take the M **highest eigenvalues**:

$$u_{1...M}^T * S * u_{1...M}^T = \lambda_{1...M}$$

----------@ **Exam question** ----------

Autoencoders are **neural networks** with *hidden layers* smaller in size than other layers (bottleneck) and they are mainly used in **non-linear models**. The **bottleneck** will therefore be forced, due to its limited numbers of units, to re-transform/reconstruct the input values into output values smaller in size.

Cap 18: Multiple Learners

Last topic is how to combine results of **multiple learners** or models. The general idea is that: instead of trying to get the best from a *single model*, we try to train *different models* (maybe they are trained not in optimal way) and after that we combine them *the result* has a better *accuracy* and is more *efficient* than any individual one. In fact, there are several theoretical and practical evidence that says that.

There are 2 families of approaches, one is given by parallel training of different models, so we take these datasets and we train all the learner in parallel (**voting**), or another possibility is to train them sequentially (**boosting**).

In the **voting** we use the same dataset to train different models, then the *prediction* can be done by a simple *weighted average* of the *prediction* of each single model. We use the weights cause if you think that a model is better than another you can improve is influence on the weighted average. There are several types of voting system like:

- **Mixture of experts**: in which instead of fixed weight you use a function that depends from input instance, that decide the different distribution of the weights.

- **Stacking**: in which the function that decide the weights is not given apriori, the system adapt it.
- **Cascading**: in which you use the learns in an alternative way, in fact in the other all learners are queried, here instead we query one learner at time, and there is a selection on the result based on a threshold. At the end the result will not be a combination of the output, it will be the output of the one with higher confidence.
- In the **bagging** instead we generate different subsets of the dataset and we train each model with a different subset; the generation of the subset typically is made with random sampling with replacement.

In the **boosting** instead we train all the learner sequentially, so each time the dataset depends on training of the previous model. This approach is very effective cause when you generate dataset for next learner you give more importance to the samples that are not being correctly classified by the previous one.

Most common algorithm for boosting is **AdaBoost**, in which the algorithm will compute the *weights* for each model (samples not correctly classified with a bigger weight) and this process is repeated M times, where M is the number of *learner* that we want to train and once the M *learners* are trained we consider a ***weighted average*** of the output of all the learners. The problem of this algorithm is that the performance depends on *data* (fail if there are insufficient data) and is sensitive to *noise*. So, at the end combining multiple learners is a very good approach.

----------@ Exam question ----------

To minimize the error, **AdaBoost** instead of computing the *global error* uses a **sequential method** and starting from the last model (the last trained) he consider the remaining models as constant values, so the error can be derived by considering only the last model, and the algorithm does this backwards up to the first model so he obtain the minimum value for the error.
