Machine Learning Notes

by George Paul Pislariu

Instructor Prof. Patrizi

1 Introduction

Machine learning is programming computers to improve a performance criterion using example data or past experience. Machine learning (or data mining) is the task of producing knowledge from data.

Machine learning techniques are useful in some cases when you don't know how to solve a problem: you haven't the algorithm you need. In other words 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. Some examples when ML is useful are:

- Human expertise does not exist (navigating on Mars);
- Humans are unable to explain their expertise (speech recognition);
- Solution changes in time (routing on a computer network);

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.

The aim of ML is to write programs that improve with experience. In order to do that you have first to formalize the problem. We must have:

- a task T;
- a performance measure P (how good you are at that task);
- an experience E;

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:

1. how to collect *experience* $\rightarrow$ **target function**;
2. what is the *structure* of the problem (what should be learned);
3. what type of *data structure* we are going to use;
4. 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.

1.1 Types of training experience

We can have different types of training experience: human expert suggests optimal move for each configuration of the board, human expert evaluates each configuration of the board or the computer plays against a human and automatically detects win/draw/loss configurations.

The main problem we have about training experience is: is training experience representative of performance goal? Or at least is it informative at all?

1.2 Target function

Considering the checkers example, a possible solution can be to assign an high rank value to the won final board state and a low rank value to the loss final board state.

We use "b" as a board state and V(b) as a target function. In our example if b is a not a final state in the game, then V(b) = V(b'), where b' is the best final board state that can be achieved starting from b and playing optimally until the end of the game.

This way gives correct values but it is not operational. That means it will take a lot of resources: time and memory.
This is the case when ML can be useful: we don't have an algorithm so the best we can do is to learn from experience.
We can represent the target function as a learned function that is a linear combination of some features. Our goal is to estimate the unknown parameters that allow me to maximize this function. Considering the above example let's formalize:

1. T: play checkers
2. E: $\langle$ b, V(b) $\rangle$
3. P: number of wins or losses

My problem is how, based on the experience, maximize the approximate function $\tilde{V}(b)$ in order to be as close as possible to the real function V(b).

$$\tilde{V}(b) \sim V(b)$$

1.3 Design choices

In a learning system determine the algorithm is the final step because first I need to know the problem. The first thing to establish is if I need ML or not.

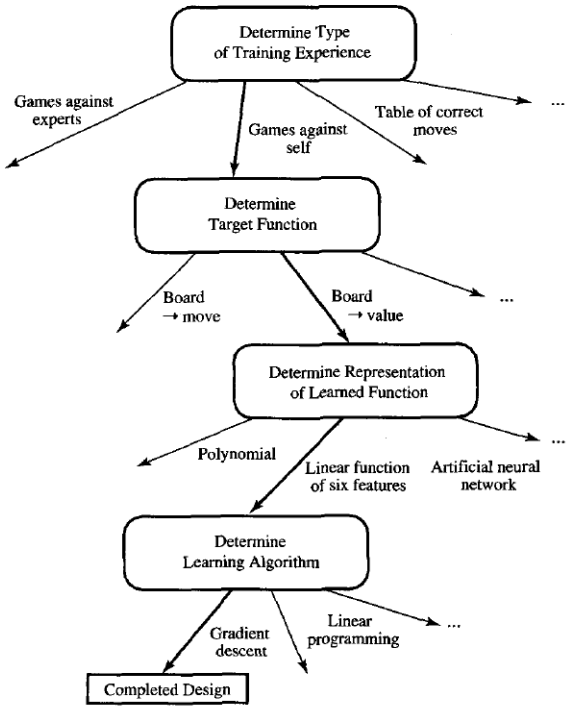


Figure 1: Design choices

1.4 Machine learning problems

A generic Machine Learning (ML) problem is learning a function $f : X \to Y$, 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 in the dataset D. What is different in the ML problems categories is how data is represented. There are three different classes of problems that deal with ML:

- **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:
  - **Classification**: return the class to which a specific instance belong. We talk about Classification when we have the task of learning a function that has a *finite codomain*.
  - **Regression**: approximate real-valued functions. We talk about Regression when we have the task of learning a function whose domain output is *infinite*.
- **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. In other words no output available: we group similar instances from D and try to find similarities.
- **Reinforcement learning**: learning a policy, a sequence of outputs. 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.
- **Concept Learning**: is a special case 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.

A general machine learning problem Learning a function f : X ⇒ Y , given a training set D containing sampled information about f.
Learning a function f means computing an approximated function $\tilde{f}$ that returns values as close as possible to f , specially for samples x not present in the training set D.

Learning a function $f : X \to Y$, given ...

- $D = \{\langle x_i, y_i \rangle\}$ **(Supervised Learning)**
- $D = \{x_i\}$ **(Unsupervised Learning)**

Learning a behavior function $\pi : S \to A$, given ...

- $D = \{< a_i^{(1)}, \ldots, a_i^{(n)}, r_i >\}$ **(Reinforcement Learning)**

$$X \equiv \begin{cases} A_1 \times \ldots \times A_n,\ A_i \text{ finite sets } \textbf{(Discrete)} \\ \Re^n \textbf{ (Continuous)} \end{cases}$$

$$Y \equiv \begin{cases} \Re^k \textbf{ (Regression)} \\ \{C_1, \ldots, C_k\},\ \textbf{(Classification)} \end{cases}$$

Special case:
$X \equiv A_1 \times \ldots A_n$ (A_i finite) and $Y \equiv \{0, 1\}$ **(Concept Learning)**

D is the available data and in supervised learning is a set of pairs ⟨x, f(x)⟩. Otherwise, in unsupervised learning D doesn't have f(x), it is just ⟨x⟩. The f(x) value in supervised learning is given by the supervisor that tells me which is the output.

1.5 Notations

1. c: target function c : X ⇒ {...}
2. X: instance space
3. D: training set
4. H: hypothesis space
5. h ∈ H: one hypothesis

1.6 Learning task

Given a training set D we want to find the best approximation h* ∈ H of the target function c ⇒ {...}
We call c a target function where $c : X \to Y$ is the function that we want to learn, where X is an instance space (input set) all the possible inputs of the function, where $x \in X$ a particular instance in the set. The dataset $D = \{(x_i, c(x_i))\}$ is a set of pairs in which for each instance xi we have the corresponding value of target function $c(x_i)$ 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 $h \in H$ an hypothesis and represent an approximation of the target function. We call h(x) an estimation of h over x (predicted value) and we want that h(x) is similar as possible to c(x) especially for values not in dataset.

The first step is to define the hypothesis space H and then to define a performance metric to determine the best approximation. h* has to work well on D but especially in the whole state space. We need a performance metric to evaluate an hypothesis. We can see the learning task as the problem of searching in the hypothesis space. The deal in this case is H is too large to check it properly: sometimes it can be infinite.

So, the goal of a learning task can be a searching problem in H to find the best h (we find the best approximation). The **real goal** is to find the best hypothesis that approximates the function c for elements that are **outside the dataset**.

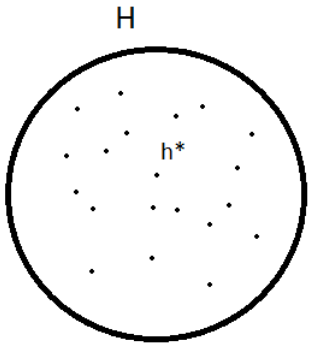


Figure 2: Learning as a search problem

1.7 Evaluating instances on hypothesis

A hypothesis h is consistent with a set of training examples D of target concept c if and only if h(x) = c(x) for each training examples (x, c(x)) in D.

Consistent(h, D) $\equiv$ ($\forall$ x $\in$ D) h(x) = c(x)

The real goal of a machine learning system is to find the best hypothesis h that predicts correct values of h(x') for instances x' $\notin$ D with respect to the unknown values c(x').

1.8 Performance measure

Given a training set D = $\{\langle x_i, c(x_i)\rangle\}$ and the hypothesis h $\in$ H, a performance measure is based on evaluating $c(x_i = h(x_i)$ for all $x_i \in D$

Inductive learning hypothesis: any hypothesis that approximates the target function well over a sufficiently large set of training examples will also approximate the target function well over other unobserved examples.

1.9 Representation of hypothesis space

In concept learning (i.e., binary classification), every hypothesis is associated to a set of instances (i.e., all the instances that are classified as positive by such hypothesis).

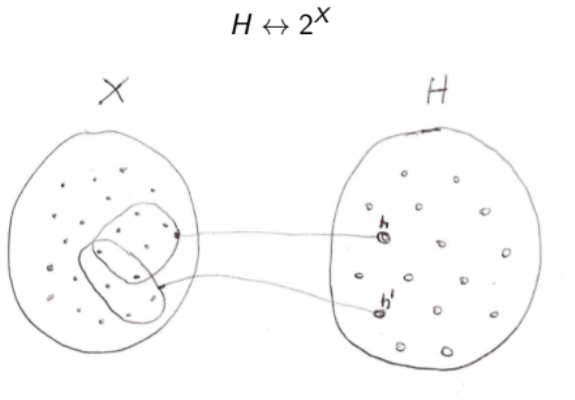


Figure 3: Representation of hypothesis space

Version spaces An hypothesis is **consistent** with training set D if $h(x) \in c(x)$ for each training example $\langle x, x(x)\rangle$.
The version space, $VS_{H,D}$, with respect to hypothesis space H and training examples D, is the subset of hypotheses from H consistent with all training examples in D.

$$VS_{H,D} \equiv \{h \in H | Consistent(h, D)\}$$

1.10 List-then-eliminate algorithm

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 $x' \notin D$.

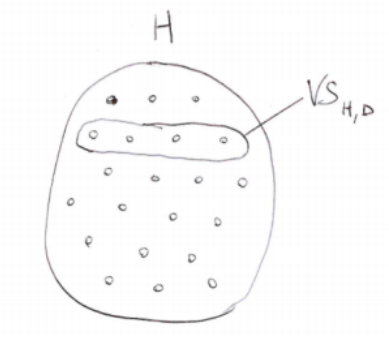


Figure 4: Version Spaces

- VersionSpace ← a list containing every hypothesis in H
- For each training example, $\langle x, c(x)\rangle$) remove from VersionSpace any hypothesis h for which $h(x) \neq c(x)$
- Output the list of hypotheses in VersionSpace

The output of a learning system is:

- $VS_{H',D}$ (H' can represent all the subsets of X);
- $VS_{H,D}$ (H can represent all the subsets of X);
- $h* \in VS_{H,D}$ (h* is one hypothesis chosen within the VS).

1.11 Machine Learning representation issue

If hypothesis space is too powerful (any subset of the instances can be represented in it), and the search is complete (all the solutions are computed), then the system is not able to classify new instances (no generalization power).

- $VS_{H',D} \rightarrow$ cannot predict instances not in D (for all $x' \notin D$, it will answer unknown)
- $VS_{H,D} \rightarrow$ limited prediction instances not in D (for some $x' \notin D$ it will answer unknown)
- $h* \in VS_{H,D} \rightarrow$ maximum prediction power on values not in D (for all $x' \notin D$ it will always return a predicted value)

1.12 Machine learning noisy data issue

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.
Data set may contain noisy data (a typical case in real applications) D = $\{\langle x_i, y_i\rangle\}$ with $y_i \neq c(x_i)$ for some i.

There may be no consistent hypotheses, i.e., $VS_{H,D} = 0$ In case of noisy data set, statistical methods must be used to implement robust algorithms.

2 Classification evaluation

Performance evaluation in classification is based on **accuracy** or **error rate**. What we want to know is given a hypothesis h, solution of a learning algorithm, what is the best estimate of the accuracy of h over future instances drawn from the same distribution?

2.1 Definitions of error

For introduce the notion of performance metrics in ML we have to define a probability distribution of all the possible inputs. Let D be the probability distribution over X, and S are a set of n instances of X sampled with D. 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 f and distribution D is the probability that h will misclassify an instance according to D. To compute the error I feed x on h so h(x) = *something*

$$error_D(h) \equiv Pr_{x \in D}[f(x) \neq h(x)]$$

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

$$error_S(h) \equiv \frac{1}{n} \Sigma_{x \in S} \delta(f(x) \neq h(x))$$

where $\delta(f(x) \neq h(x))$ is 1 if $f(x) \neq h(x)$, and 0 otherwise.

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 f(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:

$$\text{accuracy(h)} = 1 - \text{error(h)}$$

So we have that the true error can be just estimated instead sample error can be computed only on a small data sample. We need to remember that the goal of a learning system is to be accurate in h(x) $\forall x \notin S$. If $accuracy_S(h)$ is very high but $accuracy_D(h)$ is poor, then our system would not be very useful. 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.
We call **bias** the difference between the expected value of the *sample error* and the *true error*.

$$bias \equiv E[error_S(h)] - error_D(h)$$

If S is the training set used to compute h, $error_S(h)$ is optimistically biased. For unbiased estimate, h and S must be chosen independently:

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

Even with unbiased S, $error_S(h)$ may still vary from $error_D(h)$. The smaller the set S, the greater the expected variance.

2.2 Confidence intervals

So in order to calculate the true error we need to calculate the expected value of the sample error by using an unbiased estimator. We compute $error_S(h)$ as follow: definitions of error/accuracy:

- partition the data set D: $D = T \cup S, T \cap S = 0, |T| = \frac{2}{3}|D|$
- compute a hypothesis h using training set T
- Evaluate $error_S(h) = \frac{1}{n}\Sigma_{x \in S}\delta(f(x) \neq h(x))$ 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 Z_n. With approximately N% probability, $error_D(h)$ lies in interval:

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

where

$N\%$:	50%	68%	80%	90%	95%	98%	99%
z_N:	0.67	1.00	1.28	1.64	1.96	2.33	2.58

Figure 5: Confidence Intervals

- $error_S(h)$ is a random variable (i.e., result of an experiment)
- $error_S(h)$ is an unbiased estimator for $error_D(h)$

In general

- Having more samples for training and less for testing improves performance of the model: potentially better model, but $error_S(h)$ does not approximate well $error_D(h)$
- Having more samples for evaluation and less for training reduces variance of estimation: $error_S(h)$ approximates well $error_D(h)$, but this value may be not satisfactory.

Trade off for medium sized datasets: 2/3 for training, 1/3 for testing.

2.3 Comparing two hypotheses

Given two hypotheses h_1, h_2, the true comparison is

$$d \equiv error_D(h_1) - error_D(h_2)$$

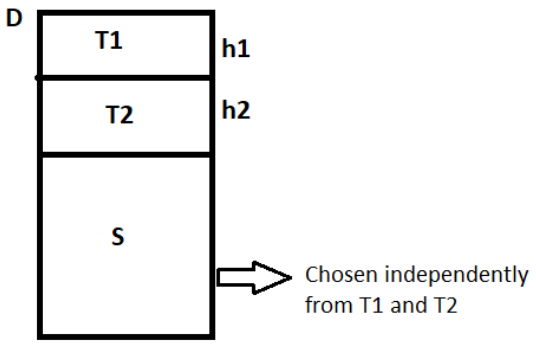
and its estimator is

$$\hat{d} \equiv error_{S_1}(h_1) - error_{S_2}(h_2)$$

$\hat{d}$ is an unbiased estimator for d, iff h_1, h_2, S_1, S_2 are independent each other. If $\hat{d}$ is positive i pick h_1, if negative i pick h_2.

$$E[\hat{d}] = d$$

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.



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

$$error_S(h) < error_S(h') < error_D(h) < error_D(h')$$

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. Basically we made too much effort to build the training set, the hypotheses are too much tailored on data. Overfitting must be avoided.

2.4 Evaluation of a learning algorithm

How can we evaluate the performance of a learning algorithm?

- h is the solution of learning algorithm L when using a training set T, $h = L(T)$
- $error_S(h)$ is the result of only one experiment and the confidence interval can be large

We can perform many experiments and compute $error_{S_i}(h)$ for different independent sample S_i. Doing that we will use the **K-fold method**.

K-fold cross validation

- partition data set D into k disjoint sets $S_1, ..., S_k$ ($|S_i| > 30$)
- for i = 1, ... , k do

    1. use S_i as test set, and the remaining data as training set T_i
    2. $T_i \leftarrow \{D - S_i\}$
    3. $h_i \leftarrow L(T_i)$
    4. $\delta_i \leftarrow error_{S_i}(h_i)$

- return $error_{L,D} \equiv \frac{1}{k} \sum_{i=1}^{k} \delta_i$

In which we partition the dataset in k set, for all the k set we train the hypothesis with the training set and we compute the sample error. Once I did this k times, I just compute the average of all the errors 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).

Comparing learning algorithms L_A and L_B We would like to estimate

$$E_{S \subset D}[error_D(L_A(S)) - error_D(L_B(S))]$$

where L(S) is the hypothesis output by learner L using training set S. The expected difference in true error between hypotheses output by learners L_A and L_B, when trained using randomly selected training sets S drawn according to distribution D. This measure can be again approximated by a K-fold cross validation.

- partition data set D into k disjoint sets $S_1, ..., S_k$ ($|S_i| > 30$)
- for i = 1, ... , k do
  - use S_i as test set, and the remaining data as training set T_i
  - $T_i \leftarrow \{D - S_i\}$
  - $h_A \leftarrow L_A(T_i)$
  - $h_B \leftarrow L_B(T_i)$
  - $\delta_i \leftarrow error_{S_i}(h_A) - error_{S_i}(h_B)$
- return $\hat{\delta} \equiv \frac{1}{k} \sum_{i=1}^{k} \delta_i$

Note that if $\hat{\delta} < 0$ we can estimate that L_A is better than L_B

2.5 Other performance metrics in classification

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. Unbalanced data sets are very common in problems related to anomaly detection (e.g. malware analysis, fraud detection, etc). So we use other performance metrics: Where:

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

Error rate = | errors | / | instances | = (FN + FP) / (TP + TN + FP + FN)
Accuracy = 1- Error rate = (TP + TN) / (TP + TN + FP + FN)

- **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:

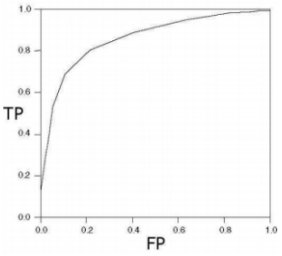
- **Error rate:** $|errors|/|instances| = (FN + FP)/(TP + FN + TN + FP)$
- **Accuracy:** 1-Error rate = $(TP + TN)/(TP + FN + TN + FP)$
- **Precision:** $TP/(TP + FP)$ ability to avoid false positives (1 if FP = 0)
- **Recall:** $TP/(TP + FN)$ ability to avoid false negatives (1 if FN = 0)
- **False positives rate:** $TP/(FP + TN)$
- **False negatives rate:** $FN/(FN + TP)$
- **F-score:** 2(Precision $*$ Recall) / (Precision $*$ Recall)

ROC curve plot of TP and FP as the threshold is changed. The ideal situation is true positive rate to 1 (no matter how we change threshold) and false positive rate to 0. The algorithm is as better as the area below the ROC curve is great.

$$Recall = TPR \cong \frac{TP}{TP+FN}$$

$$Precision = FPR \cong \frac{FP}{TN+FP}$$

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. Outside the diagonal, the errors. It is possible to see which classes are more often confused.



	C_1	C_2	C_3	C_4	C_5
C_1					
C_2					
C_3					
C_4					
C_5					

3 Decision tree

3.1 Decision tree representation

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 $h \in H$ that is consistent with D. We are going to considerate an hypotheses space made by a 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 (arc towards an internal node) represents a value of an attribute;
- each leaf assigns a classification value;

The space of instances is the set of assignments for each attribute. Each node represent a test for an attribute. We don't need to test all attributes: we search for an hypothesis represented by a decision tree that is consistent. Let's see the example of PlayTennis:

- **Instances X** = Outlook x Temperature x Humidity x Wind
- **Tuples of attributes**:
  - Outlook = $\{Sunny, Overcast, Rain\}$
  - Temperature = $\{Hot, Mind, Cold\}$
  - Humidity = $\{Normal, High\}$
  - Wind = $\{Weak, Strong\}$
- **Classification values:**
  - PlayTennis = $\{Yes, No\}$

A decision tree can be transformed in a set of rules, it represents a disjunction of conjunctions of constraints on the attribute values of instances.

Day	Outlook	Temperature	Humidity	Wind	PlayTennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

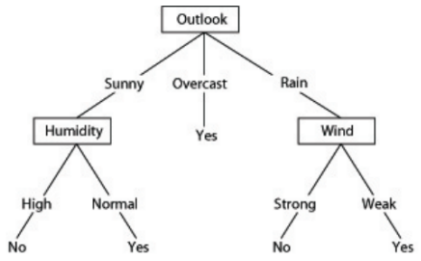


Figure 6: Left: Training data; Right: Decision tree for PlayTennis

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

Then a rule is generated for each path to a leaf node. In this way decisions are made explicit.

$$\textbf{IF} \quad (Outlook = Sunny \wedge Humidity = High)$$
$$\textbf{THEN} \quad \text{PlayTennis} = \text{No}$$

3.2 ID3 Algorithm

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 4_{th} 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 follows:

```
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}))
```

Figure 7: ID3 algorithm

3.3 Defining best attribute

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. Information gain is measured as reduction in entropy.

Entropy measures the impurity of S. Let's call $p^+ = \frac{|S^+|}{|S|}$ the proportion of positive examples in S and $p^- = 1 - p^+$ the proportion of negative examples in $S(1 - p^+)$. Entropy is defined as:

$$\text{Entropy(S)} = -p^+ \log_2 p^+ - p^- \log_2 p^-$$

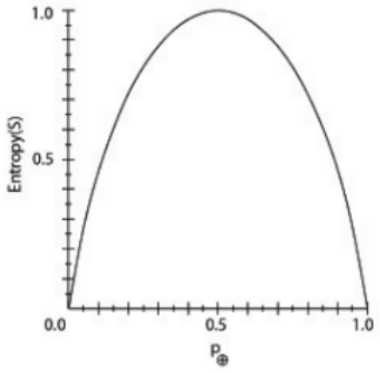


Figure 8: Entropy graph

The curve represents how S is balanced with regard to + and -. The higher point is when a number of good examples is equal to number of negative examples. So, Entropy is max when

the dataset contains exactly half positive and half negative, so when $p^+ = p^- = 0,5$ and is minimum when $p^+ = 0$ or 1.
Majority base criterion is effective when $p^+ = 1$ (when all examples are positives) or $p^- = 1$ (when all examples are negatives). In the other middle classes the majority criterion is not useful. We select attributes with **lower entropy** because they are better for classifying.

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).

$$Gain(S, A) \equiv Entropy(S) - \sum_{v \in Values(A)} \frac{|S_V|}{|S|} Entropy(S_V)$$

where *Values(A)*: set of all possible values of A

$$S_V = \{s \in S | A(s) = v\}$$

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. If we pick attributes with bigger information gain we can arrive to a leaf earlier.
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.

Some issues in Decision Tree learning are:

- Determining how deeply to grow the DT
- Handling continuous attributes
- Choosing appropriate attribute selection measures
- Handling training data with missing attribute values
- Handling attributes with different costs

3.4 Overfitting in decision trees

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;
- we start cutting some part of the tree for reduce his size (post-prune).

To determine the correct tree size i have to:

- use a separate set of examples (distinct from the training examples) to evaluate the utility of post-pruning;
- apply a statistical test to estimate accuracy of a tree on the entire data distribution;
- using an explicit measure of the complexity for encoding the examples and the decision trees.

Reduced-Error Pruning 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.
Split data into training and validation set:
Do until further pruning is harmful (decreases accuracy):

1. Evaluate impact on validation set of pruning each possible node (remove all the subtree and assign the most common classification)
2. Greedily remove the one that most improves validation set accuracy

It produces smallest version of most accurate subtree (removing sub-trees added due to coincidental irregularities). When data set is limited, reducing the set of training examples (used as validation examples) can give bad results.

Algorithm:

1. Infer the decision tree allowing for overfitting;
2. Convert the learned tree into an equivalent set of rules;
3. Prune (generalize) each rule independently of others;
4. Sort final rules into desired sequence for use.

Continuous valued attributes If Attributes have continuous values, we can use intervals in order to have discrete attributes.

Attributes with many values Whether 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.

Problem:

- If attribute has many values, *Gain* will select it
- Imagine using $Date = Jun\_3\_1996$ as attribute

One approach: use *GainRatio* instead

$$GainRatio(S, A) \equiv \frac{Gain(S, A)}{SplitInformation(S, A)}$$

$$SplitInformation(S, A) \equiv -\sum_{i=1}^{c} \frac{|S_i|}{|S|} \log_2 \frac{|S_i|}{|S|}$$

where S_i is subset of S for which A has value v_i

Figure 9: Attributes with many values

Unknown atribute values What if some examples missing values of A? Use training example anyway, sort through tree:

- If node n tests A, assign most common value of A among other examples sorted to node n
- assign most common value of A among other examples with same target value
- assign probability p_i to each possible value v_i of A
  - assign fraction p_i of example to each descendant in tree

Other algorithms based on decision tree There are other methods based on decision trees: **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. Information gain is an heuristic and doesn't give guarantees. 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.

4 Probability and Bayes network

The probability is the degree of certainty about some statements. Let's see the notation we will use:

Sample space (set of possibilities)

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

Probability space (probability model)

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

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

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

Random variables 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) $X : \Omega \to B$.
Let's see an example:
$Odd : \Omega \to Boolean$

X is a variable and a function!
$X = x_i$: the random variable X has the value $x_i \in B$
$X = x_i$ is equivalent to $\{\omega \in \Omega | X(\omega) = x_i\}$

Example: $Odd = true \equiv \{1, 3, 5\}$

P induces a probability distribution for a random variable X:

$P(X = x_i) = \sum_{\omega \in \Omega | X(\omega) = x_i} P(\omega)$

Example: $P(Odd = true) = P(1) + P(3) + P(5) = \frac{1}{6} + \frac{1}{6} + \frac{1}{6} = \frac{1}{2}$

Propositions A proposition is the event (subset of Ω) where the proposition is true. Notation for Boolean random variables: $a \equiv A = true, \neg a \equiv A = false$.
Given Boolean random variables A and B:

- event $a \equiv A = true \equiv \{\omega \in \Omega | A(\omega) = true\}$
- event $\neg a \equiv A = false \equiv \{\omega \in \Omega | A(\omega) = false\}$
- event $a \wedge b$ = points ω where $A(\omega)$ = true and $B(\omega)$ = true
- event $\neg a \vee b$ = points ω where $A(\omega)$ = false or $B(\omega)$ = true

Let's see now the syntax we will use for propositions:

- Propositional or Boolean random variables e.g., Cavity (do I have a cavity?).
  Cavity = true is a proposition, also written cavity

- Discrete random variables (finite or infinite)
  e.g., Weather is one of $<$ sunny,rain, cloudy,snow $>$.
  Weather = rain is a proposition
  Values must be exhaustive and mutually exclusive

- Continuous random variables (bounded or unbounded)
  e.g., Temp = 21.6, Temp $<$ 22.0.

- Arbitrary Boolean combinations of basic propositions
  e.g., cavity $\wedge$Weather = rain $\wedge$ Temp $\wedge$ 22.0.

Prior probability Prior or unconditional probabilities of propositions correspond to belief prior to arrival of any (new) evidence (what we believe at some point without considering any information).

Probability distribution 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. Example: $P(Odd) =< 0.5, 0.5 >$

Joint probability distribution Joint probability distribution for a set of random variables gives the probability of every atomic joint event on those random variables (i.e., every sample point in the joint sample space). If we have this table we can answer every probability question about this domain. The problem is that we don't have a lot of space to store these data so it's not so good for computation.

Joint probability distribution of the random variables *Weather* and *Cavity*:
$P(Weather, Cavity)$ = a 4×2 matrix of values:

Weather =	*sunny*	*rain*	*cloudy*	*snow*
Cavity = *true*	0.144	0.02	0.016	0.02
Cavity = *false*	0.576	0.08	0.064	0.08

Every question about a domain can be answered by the joint distribution because every event is a sum of sample points

Figure 10: Example of joint probability distribution

Conditional/Posterior probability Belief after the arrival of some evidence. I know the outcome of a random variable, how this affect probability of other random variables?
In general, conditional/posterior probabilities are different from joint probabilities and from prior probabilities.

P(Cavity = true—Weather = sunny) $\neq$ P(Cavity = true, Weather = sunny) $\neq$ P(Cavity = true)

Conditional probability distributions: representation of all the values of a conditional probabilities of random variables.

- **Conditional probability:** $P(a|b) = \frac{P(a \wedge b)}{P(b)}$ if $P(b) \neq 0$
- **Product rule:** $P(a \wedge b) = P(a|b)P(b) = P(b \setminus a)P(a)$

- **Sum rule:** $P(a \vee b) = P(a) + P(b) - P(a \wedge b)$
- **Total probability for a Boolean random variable B:** $P(a) = P(a \setminus b)P(b) + P(a|\neg b)P(\neg b)$
- **Total probability for a a random variable Y accepting mutually exclusive values y_i:** $P(X) = \sum_{y_i \in D(Y)} P(X|Y = y_i)P(Y = y_i)$ where $D(Y)$ is the set of values for variable Y.

Inference by enumeration The basic idea is to start with the joint distribution and for any proposition Φ sum the atomic events where it is true:

$$P(\Phi) = \sum_{\omega:\omega|=\phi} P(\omega)$$

	toothache		¬ *toothache*	
	catch	¬ *catch*	*catch*	¬ *catch*
cavity	**.108**	**.012**	**.072**	**.008**
¬ *cavity*	**.016**	**.064**	**.144**	**.576**

Figure 11: Example of inference for enumeration

Normalization The general idea is to compute distribution on query variable by fixing evidence variables and summing over hidden variables.

Independence The knowledge of one event doesn't affect the other event you are considering. A and B are independent iff:

$$P(A|B) = P(A) \text{ or } P(B|A) = P(B) \text{ or } P(A, B) = P(A)P(B)$$

Absolute independence powerful, but rare. Complex systems have hundreds of variables, none of which are independent.

Conditional independence X is conditionally independent from Y given Z iff:

$$P(X|Y, Z) = P(X|Z)$$

$$P(X, Y|Z) = P(X|Y, Z)P(Y|Z) = P(X|Z)P(Y|Z)$$

In most cases, the use of conditional independence reduces the size of the representation of the joint distribution from exponential in n to linear in n.

Bayes' rule

$$P(a|b) = \frac{P(b|a)}{P(a)}$$

5 Bayes Learning

5.1 Classification as probabilistic estimation

Given a target function $f : X \to V$ a dataset D and a new instance x' that is not in D, we want to compute the best prediction of the probable class it belongs to.

$$v* = argmax_{v \in V}\ P(v|x', D)$$

where $y = argmax_{x \in X} f(x) \leftarrow$ value of x among all values in X that maximize f

The general formulation is: compute the probability distribution over V:

$$P(V \setminus x', D)$$

5.2 Learning as Probabilistic estimation

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. From the Bayes rules we have that:

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

- $P(h)$ = prior probability of hypothesis h : represent the probability a priori of a hypothesis h before I get any dataset.
- $P(D)$ = prior probability of training data D: represent the probability of extracting this dataset from the entire distribution, independently from how the dataset is used in ML.
- $P(h|D)$ = probability of h given D: represent the probability that h has been generated from this dataset.
- $P(D|h)$ = probability of D given h: represent the probability that given one particular h I generate this dataset.

5.3 MAP hypotheses

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 h_{MAP} :

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

Is defined as the argmax 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 (P(D)) because it doesn't depend on h.

5.4 ML hypotheses

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

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

Note that $h_{ML} = argmax_{h \in H} P(D|h) = argmax_{h \in H} \log P(D|h)$ so we can easily derivate in order to get the max.

5.5 Brutefoce MAP hypothesis learner

For generate a MAP hypothesis we can iterate all over the possible hypotheses, and for each of this we can compute the posteriori probability and we take the hypothesis h_{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 $h \in H$ returning h_{MAP} after computing all posteriori probability.

1. For each hypothesis h in H, calculate the posterior probability

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

2. Output the hypothesis h_{MAP} with the highest posterior probability

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

If h_{MAP} is the most probable hypothesis given dataset D, given a new instance $x' \notin D$, $h_{MAP}(x')$ **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.

5.6 Bayes optimal classifier

Here we need to introduce the Bayes Optimal Classifier. Consider target function $f : X \to V, V = \{v_1, ..., v_k\}$, data set D and a new instance $x \notin D$:

$$P(v_j|x, D) = \sum_{h_i \in H} P(v_j|x, h_i)P(h_i|D)$$

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

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

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.

Bayesian learning example We have 5 kinds of bags of candies:

1. 10% are h_1: 100% cherry
2. 20% are h_2: 75% cherry, 25% lime
3. 40% are h_3: 50% cherry, 50% lime
4. 20% are h_4: 25% cherry, 75% lime
5. 10% are h_5: 100% lime

We choose a random bag (not knowing which type it is) and extract some candies from it. What kind of bag is it? What is the probability of extracting a candy of a specific flavour next? Prior probability distribution:

$$P(H) =< 0.1, 0.2, 0.4, 0.2, 0.1 >$$

Likelihood for lime candy:

$$P(l|H) =< 0, 0.25, 0.5, 0.75, 1 >$$

Probability of extracting a lime candy (without data set):

$$\sum_{h_i} P(l|h_i)P(h_i) = 0 * 0.1 + 0.25 * 0.2 + 0.5 * 0.4 + 0.75 * 0.2 + 1 * 0.1 = 0.5$$

For example let's see the case when first candy is lime: $D_1 = l$

$$P(h_i|\{d_1\}) = \alpha P(\{d_1\}|h_i)P(h_i) \text{ (Bayes Rule)}$$
$$P(H|D_1) = \alpha < 0, 0.25, 0.5, 0.75, 1 > * < 0.1, 0.2, 0.4, 0.2, 0.1 >= \alpha <$$
$$0, 0.05, 0.2, 0.15, 0.1 >=< 0, 0.1, 0.4, 0.3, 0.2 >$$

5.7 Bernoulli distribution

Probability distribution of a binary random variable $X \in 0, 1$

$$P(X = 1) = \theta P(X = 0) = 1 - \theta$$

(e.g., observing head after flipping a coin, extracting a lime candy, ...).

$$P(X = x; \theta) = \theta^x (1 - \theta)^{1-x}$$

5.8 Multi-variate Bernoulli distribution

Joint probability distribution of a set of binary random variables $X_1, ...X_n$, each random variable following Bernoulli distribution

$$P(X_1 = k_1, ..., X_n = k_n; \theta 1, ..., \theta n)$$
$$k_i \in 0, 1$$

(e.g., observing head after flipping a coin and extracting a lime candy, ...).
Under the assumption that random variables X_i are mutually independent, Multi-variate Bernoulli distribution is the product of n Bernoulli distributions:

$$\prod_{i=1}^{n} P(X_i = k_i; \theta_i)$$

5.9 Binomial distribution

Probability distribution of k outcomes from n Bernoulli trials

$$P(X = k; n, \theta) = \binom{n}{k} \theta^k (1 - \theta)^{n-k}$$

(e.g., flipping a coin n times and observing k heads, extracting k lime candies after n extractions, ...).

5.10 Multinomial distribution

Generalization of binomial distribution for discrete valued random variables with d possible outcomes. Probability distribution of k_1 outcomes for $X_1, ..., k_d$ outcomes for X_d , after n trials (with $\sum_{i=1..d} k_i = n$)

$$P(X_1 = k_1, ..., X_d = k_d; n, \theta_1, ..., \theta_d) = \frac{n!}{k_1!...k_n!}\theta_1^{k_1} * ... * \theta_d^{k_d}$$

(e.g., rolling a d-sided dice n times and observing k times a particular value, extracting k lime candies after n extractions form a bag containing d different flavors, ...).

5.11 Naive Bayes classifier

As I saw above 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 X and Y are conditionally independent if: $P(X, Y|Z) = P(X|Y, Z)P(Y|Z) = P(X|Z)P(Y|Z)$
Let's assume a target function $f : X \to Y$ where each instance x composed by attributes $(a_1, a_2, \ldots, a_n)$:

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

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**:

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

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:

$$P(a_1, ..., a_n|y, D) = \prod_i P(a_i|y, D)$$

So we obtain:

$$y_{NB} = argmax_{y \in Y} P(y|D) \prod_{i=1}^{n} P(a_i|y, D)$$

This is defined as the argmax over all the possible classification values of the probability of$y|D$ 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.

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)

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

5.12 Naive Bayes algorithm

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 $(a_1, \ldots, a_n)$:

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. $P(v_j|D)$ can be estimate as numbers of elements thatv_j as output divided by the total number of the elements in the dataset:

$$\hat{P}(v_j|D) = \frac{|\{<...,v_j>\}|}{|D|}$$

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:

$$\hat{P}(a_i|v_j, D) = \frac{|\{<...a_i,...,v_j>\}|}{|\{<...,v_j>\}|}$$

Note that if there is a missing element (if none of the training instances with target value v_j have an attribute a_i): This is not good, we must to avoid having probabilities with zero. For this we can use a prior estimates (virtual samples) so there is always non-zero probability:

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

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

5.13 Naive Bayes remarks

Conditional independence assumption is often violated

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

but it works surprisingly well anyway.

5.14 Learning to classify text

We want to use a Naive Bayes classifier that takes a set of documents (sequence of words) in input and classify them using a learn target function.
In order to achieve this we will use the bad of words representation: is a n-dimensional feature

vector. Note that the bag of words representation looses important information like the order of words in a text for example. We have two options for representing each feature:

- **boolean features:** is a vector where we record the occurrence of a word: it has 1 if the word appears or 0 otherwise. (multivariate Bernoulli distribution)
- **ordinal features:** is a vector where we record the number of occurrences of the words in the text (multinomial distribution)

Let's see the approach of the classification of documents *Docs* in classes C. The target function is:

$$f : Docs \to C, C = \{c_1, ..., c_k\}$$

$$\text{with data set } D = \{< d_i, c_i >\}$$

Given a new document d_i, compute:

$$c_{NB} = argmax_{c_j \in C} P(c_j|D) \prod_i P(d_i|c_j, D)$$

Now we use the Naive Bayes conditional independence assumption:

$$P(d_i|c_j, D) = \prod_{i}^{length(d_i)} P(a_i = w_k|c_j, D)$$

where $P(a_i = w_k|c_j)$ is probability that word in position i is w_k , given c_j.

One more assumption: $P(a_i = w_k|v_j, D) = P(a_m = w_k|v_j, D)$, $\forall i$, m, thus consider only $P(w_k|c_j, D)$. Now we have to use the multi-variate Bernoulli distribution (boolean features).

Feature vector for document d: n-dimensional vector where we have 1 if word w_k appears in document d, 0 otherwise.

$$P(d|c_j, D) = \prod_{i=1}^{n} P(w_i|c_j, D)^{I(w_i \in d)} * (1 - P(w_i|c_j, D))^{1-I(w_i \in d)}$$

$$\text{where } I(w_i \in d) = 1 \text{ if } w_i \in d, \text{ 0 otherwise.}$$

$$\hat{P}(w_i|c_j, D) = \frac{t_{i,j}+1}{t_j+2}$$

where:

- $t_{i,j}$: number of documents in D of class c_j containing word w_i
- t_j : number of documents in D of class c_j
- 1, 2: parameters for Laplace smoothing

Now the feature vector for document d: n-dimensional vector with number of occurrences of word w_i in document d.

$$P(d|c_j, D) = Mu(d; n, \theta) = ...$$

$$\hat{P}(w_i|c_j, D) = \frac{\sum_{d \in D} tf_{i,j} + \alpha}{\sum_{d \in D} tf_j + \alpha * |V|}$$

where:

- $tf_{i,j}$: term frequency (number of occurrences) of word w_i in document d of class c_j containing word w_i
- tf_j : all term frequencies of document d of class c_j
- α: smoothing parameter ($\alpha = 1$ for Laplace smoothing)

We can use estimated $\hat{P}(c_j)$ and $\hat{P}(w_i|c_j)$ to classify a new document.

Estimate $\hat{P}(c_j)$ and $\hat{P}(w_i|c_j)$ using *Bernoulli distribution*.

LEARN_NAIVE_BAYES_TEXT_BE(D, C)

$V \leftarrow$ all distinct words in D
for each target value $c_j \in C$ do
  $docs_j \leftarrow$ subset of D for which the target value is c_j
  $t_j \leftarrow |docs_j|$: total number of documents in c_j
  $\hat{P}(c_j) \leftarrow \frac{t_j}{|D|}$
  for each word w_i in V do
    $t_{i,j} \leftarrow$ number of documents in c_j containing word w_i
    $\hat{P}(w_i|c_j) \leftarrow \frac{t_{i,j}+1}{t_j+2}$

Estimate $\hat{P}(c_j)$ and $\hat{P}(w_i|c_j)$ using *multinomial distribution*.

LEARN_NAIVE_BAYES_TEXT_MU(D, C)

$V \leftarrow$ all distinct words in D
for each target value $c_j \in C$ do
  $docs_j \leftarrow$ subset of D for which the target value is c_j
  $t_j \leftarrow |docs_j|$: total number of documents in c_j
  $\hat{P}(c_j) \leftarrow \frac{t_j}{|D|}$
  $TF_j \leftarrow$ total number of words in $docs_j$ (counting duplicates)
  for each word w_i in V do
    $TF_{i,j} \leftarrow$ total number of times word w_i occurs in $docs_j$
    $\hat{P}(w_i|c_j) \leftarrow \frac{TF_{i,j}+1}{TF_j+|V|}$

CLASSIFY_NAIVE_BAYES_TEXT(d)

remove from d all words not included in vocabulary V
return

$$v_{NB} = \underset{c_j \in C}{\operatorname{argmax}} \hat{P}(c_j) \prod_{i=1}^{length(d)} \hat{P}(w_i|c_j)$$

Figure 12: Naive Bayes Text Classification algorithm

Improvements We can do more in order to improve the algorithm for example using stop words, lemming, bi-gram, ecc.

6 Probabilistic model for 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 $P(C_i|x)$ through $P(x|C_i)$ and Bayes
- **Discriminative**: estimates $P(C_i|x)$ from a model

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

6.1 Generative model

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**:

$$P(C_1|x) = \frac{P(x|C_1)P(C_1)}{P(x|C_1)P(C_1)+P(x|C_2)P(C_2)} = \frac{1}{1+e^{-\alpha}} = \sigma(\alpha)$$

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

$$\text{and: } \sigma(\alpha) = \frac{1}{1+e^{-\alpha}} \text{ the sigmoid function}$$

Now we assume that $P(x|C_i)$ can be replaced by a gaussian distribution, so we get that $P(C_i|x)$ 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.

So, in order to estimate $P(x|C_i$ let's assume:

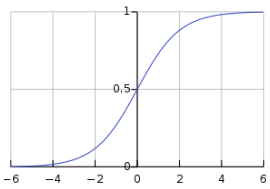


Figure 13: Sigmoid function

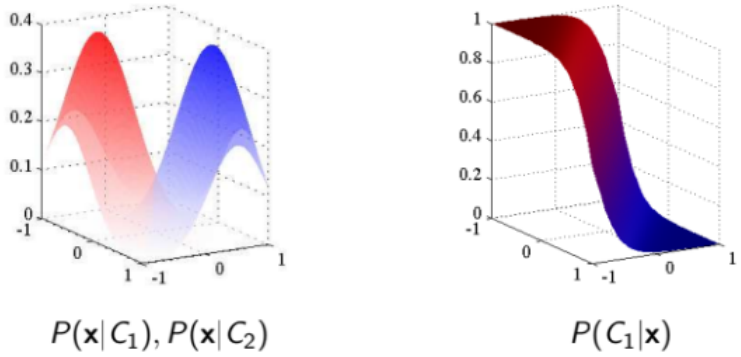
$p(x|C_i) = N(x|\mu_i, \Sigma)$ where N is the normal distribution and have the same covariance matrix

$$a = \ln \frac{p(x|C_1)P(C_1)}{p(x|C_2)P(C_2)} = \ln \frac{p(x|C_i) = N(x|\mu_1, \Sigma)P(C_1)}{p(x|C_i) = N(x|\mu_2, \Sigma)P(C_2)} = ... = w^T x + w_0$$

with: $w = \sum^{-1}(\mu_1 - \mu_2)$ so we have: $w_0 = -\frac{1}{2}\mu_1^T \sum^{-1} \mu_1 + \frac{1}{2}\mu_2^T \sum^{-1} \mu_2 + \ln \frac{P(C_1)}{P(C_2)}$

Thus:

$$P(C_1|x) = \sigma(w^T x + w_0)$$



$P(\mathbf{x}|C_1), P(\mathbf{x}|C_2)$ $P(C_1|\mathbf{x})$

Since we have two classes we take the one with the highest probability, so the decision rule is:

$$c = C_1 \iff P(c = C_1|x) > 0.5$$

6.2 Maximum likehood for two classes

Assuming:
$P(C_1) = \pi$ (thus $P(C_2) = 1 - \pi$)
$P(x|C_i) = N(x|\mu_i, \Sigma)$

Given a data set $D = \{(x_n, t_n)_{n=1}^N\}$, $t_n = 1$ if x_n belongs to class C_1, $t_n = 0$ if x_n belongs to class C_2.

Let N_1 be the number of samples in D belonging to C_1 and N_2 be the number of samples in C_2 ($N_1 + N_2 = N$)

The likehood function is:

$$P(t|\pi,\mu_1,\mu_2,\Sigma,D)=\prod_{n=1}^{N}[\pi N(x_n|\mu_1,\Sigma)]^{(t_n)}[(1-\pi)N(x_n|\mu_2,\Sigma)]^{(1-t_n)}$$

We don't know the values of π,μ_1,μ_2,Σ. The variable t_n is 1 if the instance belongs to class 1, 0 if not.

We want to maximize the log likehood function. Doing that we obtain:

$$\pi=\frac{N_1}{N}\ ;\ \mu_1=\frac{1}{N_1}\sum_{n=1}^{N}t_n x_n\ ;\ \mu_2=\frac{1}{N_2}\sum_{n=1}^{N}(1-t_n)x_n$$

$$\Sigma=\frac{N_1}{N}S_1+\frac{N_2}{N}S_2 \text{ with: } S_i=\frac{1}{N_i}\sum_{n\in C_i}(x_n-\mu_i)(x_n-\mu_i)^T,\ i=1,2$$

We assumed that all distributions N shared the same covariance matrix. If we don't assume this the number of parameters is too big (very hard to compute). Basically we must do a trade-off between computing power we need and accuracy we want. If we have only a small dataset is better to choose less parameters: with a small dataset i have high probability of overfitting and we must avoid that.

6.3 Maximum likehood for K classes

Let's remember the formulas from above: $P(C_k)=\pi_k$; $P(x|C_k)=N(x|\mu_k,\Sigma)$ Now, given a data set $D=\{(x_n,t_n)_{n=1}^{N}\}$ with t_n 1-of-k encoding, we obtain:

$$\pi_k=\frac{N_k}{N}\ ;\ \mu_k=\frac{1}{N_k}\sum_{n=1}^{N}t_{nk}x_n\ ;\ \Sigma=\sum_{K=1}^{K}\frac{N_k}{N}S_k$$

$$S_k=\frac{1}{N_k}\sum_{n=1}^{N}t_{nk}(x_n-\mu_k)(x_n-\mu_k)^T$$

6.4 Discriminative model

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. There are many approaches to estimate directly $P(C_i|x)$. We will use the logistic regression, that is a classification method based on maximum likehood.

Logistic regression Given a data set D, consider $\{x_n,t_n\}$, with $t_n\in\{0,1\}$ with $n=1,..,N$. The likehood function we obtain is:

$$p(t|w)=\prod_{n=1}^{N}y_n^{t_n}(1-y_n)^{1-t_n}$$

with $y_n=p(C_1|x_n)=\sigma(w^Tx_n)$ that indicates the probability that given x_n this belongs to class C_1.

The two approaches (general/determinative) differs from the starting point; in the general I start from a normal distribution (assumption) from $P(x|C_i)=N(x|\mu_i,\Sigma)$. In the determinative model I start from $P(C_1|x)=\sigma(w^Tx_n)$ assuming I already.

- **Generative**: passing through condition probability; it allows you to generate data samples according with the distribution.
- **Determinative**: estimate directly;

In the generative approach we have $y_n = p(C_1|x) = \sigma(w^T x_n + w_0)$ while in the discriminative we have $y_n = p(C_1|x_n) = \sigma(w^T x_n)$. Basically we just included w_0 into w^T, as follow:

$$\left(\begin{bmatrix} w_1 & \dots & w_d \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix} + w_0 \right) = \left(\begin{bmatrix} w_0 & w_1 & \dots & w_d \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_d \end{bmatrix} + w_0 \right)$$

Cross-entropy The cross-entropy error function is:

$$E(w) \equiv -\ln p(t|w) = -\sum_{n=1}^{N} [t_n \ln y_n + (1 - t_n) \ln(1 - y_n)]$$

The idea is instead of maximizing the formula we seen back, we minimize this one above.

$$w* = argmin_w E(w)$$

To solve this we can use many solvers, let's take a look at one of those, the iterative re-weighted least squares.

Iterative re-weighted squares This approach apply Newton-Raphson iterative optimization for minimizing $E(w)$. Basically we want to find the local minimum; the Newton-Raphson method pick the minimum w_1 and compute the error, if it's not good enough pick the derivative w_1' and so on. Doing this iteratively you will find a local minimum.
Gradient of the error with respect to w:

$$\bigtriangledown E(w) = \sum_{n=1}^{N} (y_n - t_n) x_n$$

The gradient descent step is:

$$w \leftarrow w - H^{-1} \bigtriangledown E(w)$$

In order to compute the equation above we need to compute first H and $\bigtriangledown E(w)$. Given $t = (t_1 \dots t_n)$, $y = (y_1 \dots y_n)^T$ and R= diagonal matrix with $R_{nn} = y_n(1 - y_n)$ and the vector $\begin{bmatrix} x_1^T \\ \vdots \\ x_N^T \end{bmatrix}$ we have:

$$\bigtriangledown E(w) = X^T(y - t)$$

$$H = \bigtriangledown \bigtriangledown E(w) = \sum_{n=1}^{N} y_n(1 - y_n) x_n x_n^T = X^T R X$$

Summing up the iterative method is:

- inizialize w
- repeat until termination condition: $w \leftarrow w - (X^T R X)^{-1} X^T (y - t)$

Multiclass logistic regression

$$P(C_k) = y_k(x) = \frac{e^{a_k}}{\sum_j e^{a_j}}$$

with $a_k = w_k^T x$. The discriminative model is:

$$P(T|w_1 \dots w_k) = \prod_{n=1}^{N} \prod_{k=1}^{K} P(C_k|x_n)^{t_{nk}} = \prod_{n=1}^{N} \prod_{k=1}^{K} y_{nk}^{t_{nk}}$$

with $y_{nk} = y_k(x_n)$ and T an $N * K$ matrix of t_{nk}.

The cross-entropy error function becomes:

$$E(w_1 \dots w_k) = -\ln P(T|w_1 \dots w_k) = \sum_{n=1}^{N} \sum_{k=1}^{K} t_{nk} \ln y_{nk}$$

The iterative algorithm with gradient is $\bigtriangledown_{w_j} E(w_1 \dots w_k) = \dots$ and Hessian matrix $\bigtriangledown_{w_k} \bigtriangledown_{w_j} E(w_1 \dots w_k)$.

The difference is in this case T becomes a matrix that for each row has 1 when that x belong to class 1, 0 otherwise:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ \dots & \dots & \dots \end{bmatrix}$$

The rows are representing $x_1 \dots x_n$ while the cols $1 \dots k$. Of course the Bayesian classifier belongs to the generative model.

Learning in feature space

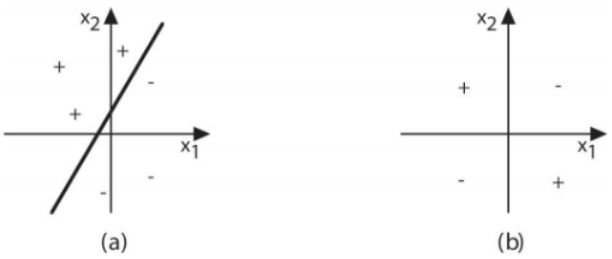
- All methods described above can be applied in a transformed space of the input (feature space).
- Given a function $\phi : X \to \phi$ (ϕ is the feature space) each sample x_n can be mappet to a feature vector $\phi_n = \phi(x_n)$
- Replacing x_n with $\phi(x_n)$ in all the equations above, makes the learning system to work in the feature space instead of the input space.

7 Linear model 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 $f : X \to Y$ with: and with $X \subset R^d$ and $Y = \{C_1 \dots C_k\}$ 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 two regions, 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.
- 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 $y : X \to \{C_1, \ldots, C_k\}$:

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

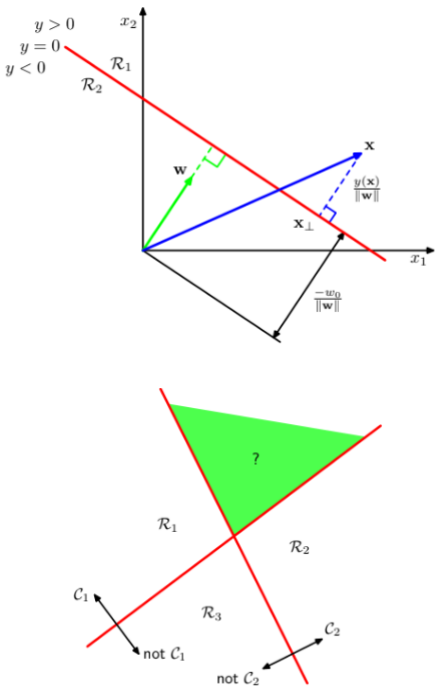
In the first case we have 2 classes like + and -, then the **discriminant function** can be given just by a linear function, where x is the input, w and w_0 are the coefficient of this function, and if you change this number you will have different functions in space.

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 observing this criterion: for $y(x) \leq 0$ then $x \in C_1$ otherwise if $y(x) > 0$ then $x \in C_2$. The red line is the intersection of the hyperplane with the plane defined by (x_1, x_2).

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-versus-one classifier** in which $K(K-1)/2$ 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:
$y_k(x) = w_k^T x + w_0$ Assigning x to C_k if $y_k(x) > y_j(x)$ 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:

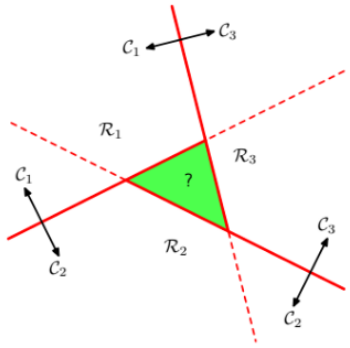
$$y(x) = \tilde{W}^T \tilde{x}$$
$$\Downarrow$$
$$\tilde{W} = \begin{Bmatrix} w_{1_0} & w_{2_0} & \dots & w_{K_0} \\ w_1 & w_2 & \dots & w_k \end{Bmatrix} \quad \tilde{x} = \begin{Bmatrix} 1 \\ x \end{Bmatrix}$$

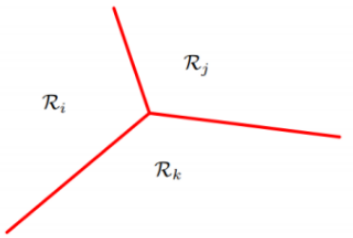
So, given a multi-class classification problem and a dataset D with linearly separable data, we need to determine **W** suck that: $y(x) = \tilde{W}^T \tilde{x}$ is the K-class discriminant. We need to compute the coefficient of the matrix, and once we compute these values, we can classify new instances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

There are 4 kinds of solution for this problem:

- Least squares
- Fisher's linear discriminant
- Perceptron
- 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 x_n is a point in our space and 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 $x_n \in C_i$.





7.1 Least squares

We define: $\tilde{X} = \begin{pmatrix} \tilde{x}_1^T \\ \vdots \\ \tilde{x}_N^T \end{pmatrix} \quad T = \begin{pmatrix} t_1^T \\ \vdots \\ t_N^T \end{pmatrix}$

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:

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

A closed form solution is:

$$\tilde{W} = (\tilde{X}^T\tilde{X})^{-1}\tilde{X}^T T \qquad \text{where } (\tilde{X}^T\tilde{X})^{-1}\tilde{X}^T = \tilde{X}^\dagger$$

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

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 (Figure 14), cause it find the line that is the best average distance from each points.

7.2 Fisher

In this case we consider two classes case, and we have to determine $y = w^T x$ and we want to classify $x \in C1$ if $y \geq -w_0, x \in C_2$ otherwise. Corresponding to the projection on a line

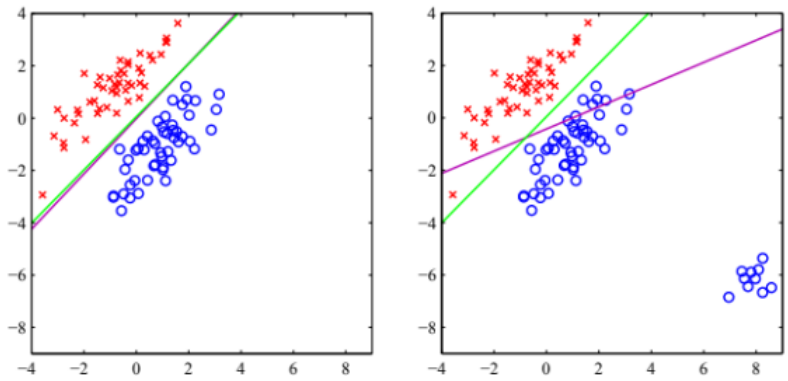


Figure 14: Assume Gaussian conditional distributions. Not robust to outliers!

determined by w. We have to adjust w to find a direction that maximizes the class separation. Consider a data set with N_1 points in C_1 and N_2 points in C_2. So we need to find the medium:

$$m_1 = \frac{1}{N_1} \sum_{n \in C_1} x_n$$

$$m_2 = \frac{1}{N_2} \sum_{n \in C_2} x_n$$

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

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

with

$$S_B = (m_2 - m_1) * (m_2 - m_1)^T$$

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

We find w that maximizes it by solving:

$$\omega^* \propto S_\omega^{-1} * (m_2 - m_1)$$

S_B is "between class scatter", SW 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.
Summarizing, given a two classes classification problem, Fisher's linear discriminant is given by the function $y = w^T x$ and the classification of new instances is given by $y \geq -w_0$ where

$$w = S_W^{-1}(m_2 - m_1)$$

$$w_0 = w^T m$$

m is the global mean of all the data set.

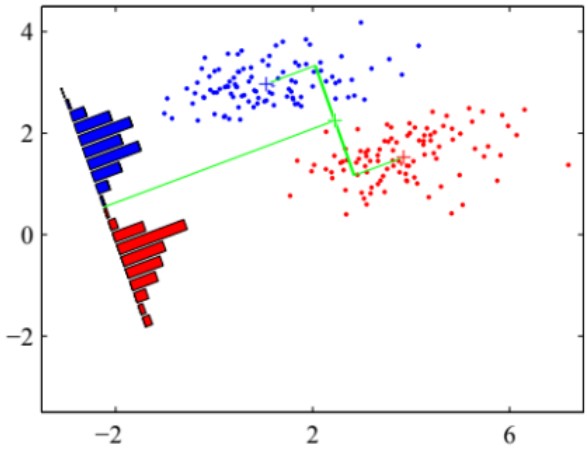


Figure 15: Fisher's linear discriminant

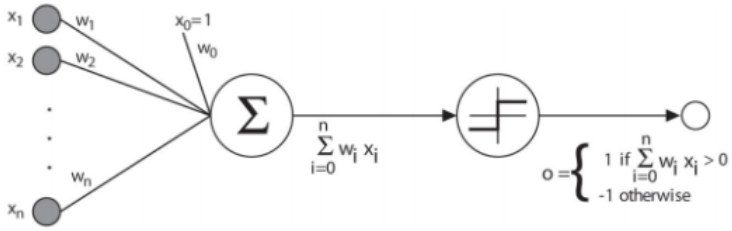


Figure 16: Perceptron

7.3 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:

$$o(x_1 \dots x_d) = \begin{cases} 1 \text{ if } w_0 + w_1 x_1 + \cdots + w_d x_d > 0 \\ \text{-1 otherwise} \end{cases}$$

Sometimes we'll use simpler notation (adding $x_0 = 1$):

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

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 te unthresholded linear unit, where:

$$o = w_0 + w_1 x_1 + \cdots + w_d x_d = w^T x$$

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.

Let's lean w_i from training examples $D = \{(x_n, t_n)_{n=1}^N\}$ that minimize the squared error (loss function):

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

The training rule is the follow:

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

Update of weights w:

$$w_i \leftarrow w_i + \triangle w_i$$

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

Where η is a small constant (e.g. 0.05) called **learning rate**.

The perceptron algorithm has those steps:

1. initialize $\hat{w}$ (e.g. small random values)
2. repeat until termination condition: $\hat{w}_i \leftarrow \hat{w}_i + \bigtriangledown w_i$
3. output $\hat{w}$

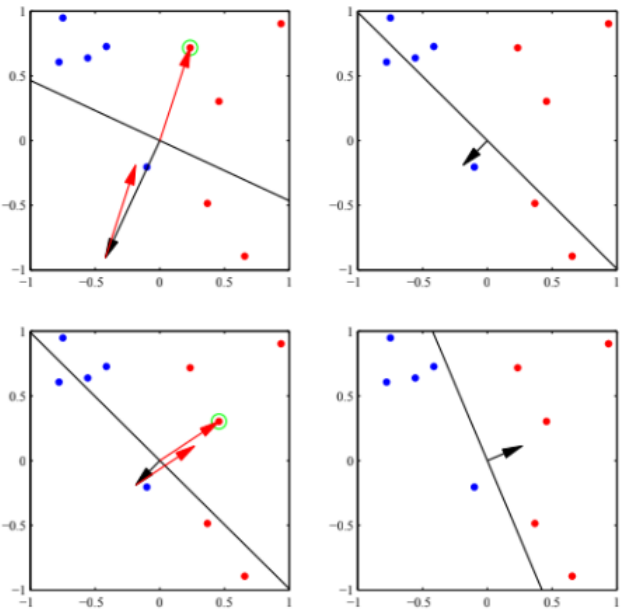
We can have some variants of the perceptron algorithm, depending on how much of the dataset we use:

- **Batch mode:** consider all dataset D
$$\triangle w_i = \eta \sum_{(x,t)\in D} (t - o(x))x_i$$
- **Mini-Batch mode:** choose a small subset S $\subset$ D
$$\triangle w_i = \eta \sum_{(x,t)\in S} (t - o(x))x_i$$
- **Incremental mode:** choose one sample $(x, t) \in D$
$$\triangle w_i = \eta(t - o(x))x_i$$

$o(x) = w^T x$ for unthresholded, $o(x) = sign(w^T x)$ for thresholded. Incremental and mini-batch modes speed up convergence and are less sensitive to local minima.

Example: $\eta = 0.1$ and $x_i = 0.8$

- if $t = 1$ and$o = -1$ then $\triangle w_i = 0.16$
- if $t = -1$ and$o = 1$ then $\triangle w_i = -0.16$



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.

7.4 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.

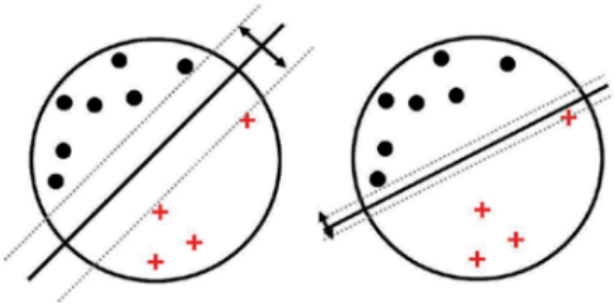


Figure 17: Support vector machines

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.

Let's consider binary classification $y : X \to \{+1, -1\}$ with data set $D = (x_n, t_n)_{n=1}^N$, $t_n \in \{+1, -1\}$ and a linear model $y(x) = w^T x + w_0$.
Assume **D** is linearly separable:

$$\exists w, w_0 \text{ s.t. } \begin{cases} y(x_n) > 0, \text{ if } t_n = +1 \\ y(x_n) < 0, \text{ if } t_n = -1 \end{cases}$$

$$t_n y(x_n) > 0 \quad \forall n = 1, \dots, N$$

Let x_k be the closest point of the data set D to the hyperplane $\bar{h} : \bar{w}^T x + \bar{w}_0 = 0$. The margin (smallest distance between x_k and $\bar{h}$) is

$$\frac{|y(x_k)|}{||w||}$$

Given data set D and hyperplane $\bar{h}$, the margin is computed as:

$$min_{n=1,\dots,N} \frac{|y(x_n)|}{||w||} = \dots = \frac{1}{||w||} min_{n=1,\dots,N} [t_n(\bar{w}^T x_n + \bar{w}_0)]$$

using the property$|y(x_n)| = t_n y(x_n)$.

Given data set D, the hyperplane $h^* : w^{*T} x + w_0^* = 0$ with maximum margin computed as:

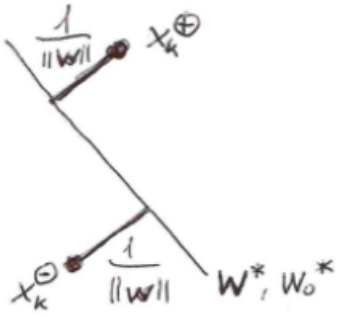
$$w^*, w_0^* = argmax_{w,w_0} \frac{1}{||w||} min_{n=1,\dots,N} [t_n(w^T x_n + w_0)]$$

Rescaling all the points does not affect the solution. Rescale in such a way that for the closet point x_k we have $t_k(w^T x_k + w_0) = 1$.

Canonical representation: $t_n(w^T x_n + w_0) \geq 1 \; \forall n = 1, \ldots, N$.

When the maxim margin hyperplane w^*, w_0^* is found, there will be at least 2 closest points $x_k^{\oplus}$ and $x_k^{\ominus}$ (one for each class).

- if $w^T x_k^{\oplus} + w_0^* = +1$
- if $w^T x_k^{\ominus} + w_0^* = -1$



In the canonical representation of the problem the maxim margin hyperplane can be found by solving the optimization problem:

$$max \frac{1}{||w||} = min \frac{1}{2}||w||^2$$

subject to

$$t_n(w^T x_n + w_0) \geq 1 \quad \forall n = 1, \ldots, N$$

Quadratic programming problem solved with Lagrangian method. Solution

$$w^* = \sum_{n=1}^{N} a_n t_n x_n$$

where a_i (Lagrange multipliers): results of the Lagrangian optimization problem:

$$\tilde{L} = \sum_{n=1}^{N} a_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} a_n a_m t_n t_m x_n^T x_m$$

subject to:

$$a_n \geq 0 \quad \forall n = 1, \ldots, N$$

$$\sum_{n=1}^{N} a_n t_n = 0$$

The Karush-Kuhn-Tucker (KKT) condition for each $x_n \in X_D$, either $a_n = 0$ or $t_n y(x_n) = 1$ x_n for which $a_m = 0$ do not contribute to the solution. Support vectors: x_k such that $a_k \neq 0$ and $t_k y(x_k) = 1$

$$SV \equiv x_k \in X_D | t_k y(x_k) = 1$$

Hyperplanes expressed with support vectors

$$y(x) = \sum_{x_j \in SV} a_j t_j x^T x_j + w_0 = 0$$

Note: other vectors $x_n \notin SV$ do not contribute ($a_n = 0$).

To compute w_0:

Support vector $x_k \in SV$ satisfies $t_k y(x_k) = 1$

$$t_k(\sum_{x_j \in SV} a_j t_j x_k^T x_j + w_0) = 1$$

Multiplying by t_k and using $t_k^2 = 1$ we obtain:

$$w_0 = t_k - \sum_{x_j \in SV} a_j t_j x_k^T x_j$$

Instead of using one particular support vector x_k to determine w_0:

$$w_0 = t_k - \sum_{x_j \in SV} a_j t_j x_k^T x_j$$

A more stable solution is obtained by averaging over all the support vectors

$$w_0 = \frac{1}{|SV|} \sum_{x_j \in SV} (t_k - \sum_{x_j \in S} a_j t_j x_k^T x_j)$$

Given the maximum margin hyperplane determined by a_k^* , w_0^* the classification of a new instance x' is:

$$sign(y(x')) = sign(\sum_{x_k \in SV} a_k^* t_k x_k'^T x_k + w_0^*)$$

Optimization problem for determining w (dimension $|X|$) is transformed in an optimization problem for determining a (dimension $|D|$) Efficient when $|X| < |D|$ (most of a_i will be zero). Very useful when $|X|$ is large or infinite.

SVM with soft margin constrains What if data are "almost" linearly separable (e.g., a few points are on the "wrong side"). Let us introduce slack variables $\xi_n \geq 0 \quad n = 1, \ldots, N$.

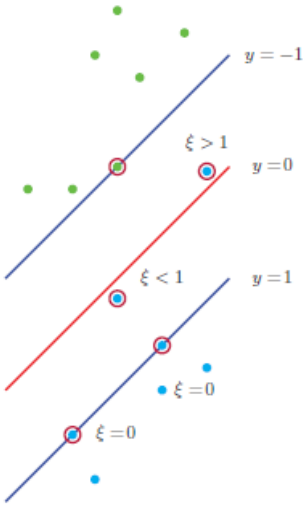
- $\xi_n = 0$ if point on or inside the correct margin boundary
- $0 < \xi_n \leq 1$ if point inside the margin but correct side
- $\xi_n > 1$ if point on wrong side of boundary

When $\xi_n = 1$, the sample lies on the decision boundary $y(x_n) = 0$ when $\xi_n > 1$, the sample will be miss-classified Soft margin constraint:

$$t_n y(x_n) \geq 1 - \xi_n, \quad n = 1, \ldots, N$$

Optimization problem with soft margin constraints:

$$min \frac{1}{2}||w||^2 + C \sum_{n=1}^{N} \xi_n$$



subject to:

$$t_n y(x_n) \geq 1 - \xi_n, \quad n = 1, \ldots, N$$
$$\xi_n \geq 0, \quad n = 1, \ldots, N$$

C is a constant (inverse of a regularization coefficient). Solution similar to the case of linearly separable data:

$$w^* = \sum_{n=1}^{N} a_n t_n x_n$$
$$w_0^* = ...$$

with an computed as solution of a Lagrangian optimization problem.

So far we considered models working directly on x. All the results hold if we consider a non-linear transformation of the inputs $\phi(x)$ (basis functions).
Decision boundaries will be linear in the feature space ϕ and non-linear in the original space x. Classes that are linearly separable in the feature space ϕ may not be separable in the input space x.

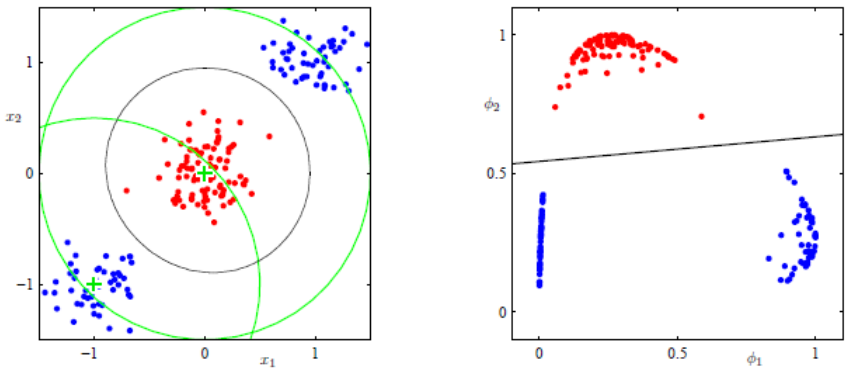


Figure 18: Basic functions example

- Linear
- Polynomial
- Radial Basis Function (RBF)
- Sigmoid
- ...

Linear models for non-linear functions Learning non-linear function

$$y : X \to C_1, \ldots, C_K$$

from data set D non-linearly separable. Find a non-linear transformation ϕ and learn a linear model:

$$y(x) = w^T \phi(x) + w_0$$

(two classes)

$$y_k(x) = w_k^T \phi(x) + w_{k0}$$

(multiple classes)

7.5 Summary

- Basic methods for learning linear classification functions
- Based on solution of an optimization problem
- Closed form vs. iterative solutions
- Sensitivity to outliers
- Learning non-linear functions with linear models using basis functions
- Further developed as kernel methods

8 Linear models for regression

8.1 Overview

- Linear models for regression
- Maximum likelihood and Least squares
- Sequential learning
- Regularization

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 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: $y(x; w)$ with parameters w to approximate the target function f .
Linear model for linear function:

$$y(x; w) = w_0 + w_1 x_1 + \cdots + w_d x_d = w^T x$$

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

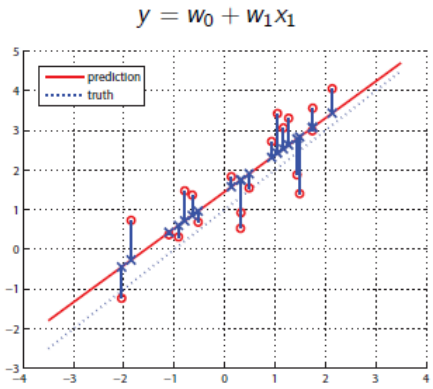


Figure 19: Example: 2D line fitting

8.2 Linear basis function models

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. Using non-linear functions of input variables:

$$y(x; w) = \sum_{j=0}^{M-1} w_j \phi_j(x) = w^T \phi(x)$$

$$w = \begin{pmatrix} w_0 \\ \vdots \\ w_{M-1} \end{pmatrix} \quad \phi(x) = \begin{pmatrix} \phi_0(x) \\ \vdots \\ \phi_{M-1}(x) \end{pmatrix} \quad \phi_0(x) = 1$$

Still linear in the parameters w! 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.

$$y = w_0 + w_1 x + w_2 x^2 + \ldots + w_M x^M = \sum_{j=0}^{M} w_j x^j$$

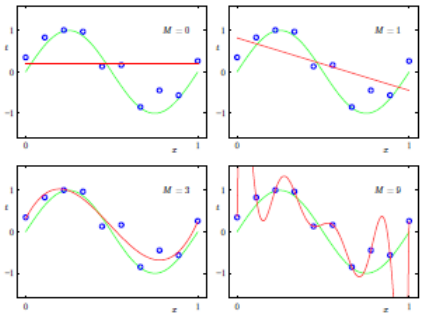


Figure 20: Example: Polynomial curve fitting

8.3 Algorithms

Maximum likelihood and least squares For minimize the error in our model we will maximise the likelihood. Now let's define our regression problem, we consider the target value

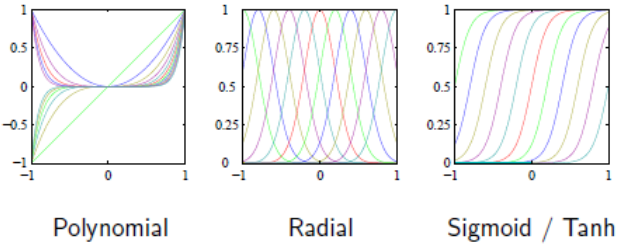


Figure 21: Examples of basis functions

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). Target value t is given by $y(x; w)$ affected by additive noise ε

$$t = y(x; w) + \varepsilon$$

Assume Gaussian noise $P(\varepsilon|\beta) = \mathcal{N}(\varepsilon|0, \beta^{-1})$, with precision (inverse variance) β. We have:

$$P(t|x, w, \beta) = \mathcal{N}(t|y(x; w), \beta^{-1})$$

Assume observations independent and identically distributed (i.i.d.). We seek the maximum of the likelihood function:

$$P(t_1, \ldots, t_N|x_1, \ldots, x_n, w, \beta) = \prod_{n=1}^{N} \mathcal{N}(t_n|w^T\phi(x_n), \beta^{-1})$$

or equivalently:

$$\ln P(\{t_1, \ldots, t_N\}|x_1, \ldots, x_N, w, \beta) = \sum_{n=1}^{N} \ln \mathcal{N}(t_n|w^T\phi(xn), \beta^{-1})$$

$$= -\beta\frac{1}{2}\sum_{n=1}^{N}[t_n - w^T\phi(x_n)]^2 - \frac{N}{2}ln(2\pi\beta^{-1})$$

The maximum likelihood

$$max \;\; P(t_1, ..., t_N|x_1, ..., x_n, w, \beta)$$

corresponds to least square error minimization:

$$min \;\; E_D(w) = min \;\; \frac{1}{2}\sum_{n=1}^{N}[t_n - w^T\phi(x_n)]^2$$

Note:

$$E_D(w) = \frac{1}{2}(t - \phi w)^T(t - \phi w)$$

where:

$$t = \begin{pmatrix} t_1 \\ \vdots \\ t_N \end{pmatrix} \qquad \phi = \begin{pmatrix} \phi_0(x_1) & \phi_1(x_1) & \ldots & \phi_{M-1}(x_1) \\ \phi_0(x_2) & \phi_1(x_2) & \ldots & \phi_{M-1}(x_2) \\ \vdots & & & \\ \phi_0(x_N) & \phi_1(x_N) & \ldots & \phi_{M-1}(x_N) \end{pmatrix}$$

Optimality condition:

$$\bigtriangledown E_D = 0 \Longleftrightarrow \phi^T\phi w = \phi^T t$$

Hence:

$$w_{ML} = (\phi^T\phi)^{-1}\phi^T t$$

Sequential learning Stochastic gradient descent algorithm:

$$\hat{w} \leftarrow \hat{w} - \eta \bigtriangledown E_n$$

with η the learning rate parameter. Therefore:

$$\hat{w} \leftarrow \hat{w} + \eta[t_n - \hat{w}^T \phi(x_n)]\phi(x_n)$$

Algorithm converges for suitable small values of η.

Regularization Regularization is a technique to control over-fitting.

$$min \;\; E_D(w) + \lambda E_W(w)$$

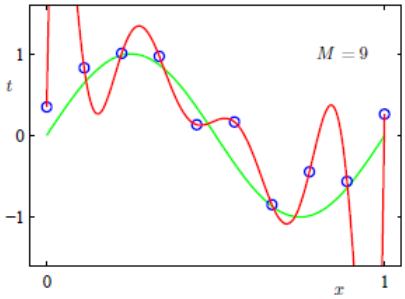
with $\lambda > 0$ being the regularization factor. A common choice is:

$$E_W(w) = \frac{1}{2} w^T w$$

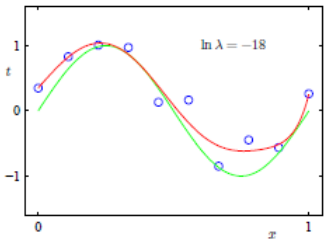
Other choice can be:

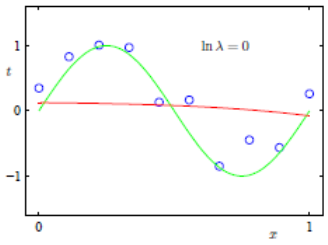
$$E_W(w) = \sum_{j=0}^{M-1} |w_j|^q$$

$\min E_D(\mathbf{w})$



$\min E_D(\mathbf{w}) + \lambda \frac{1}{2}\mathbf{w}^T\mathbf{w}$





Multiple outputs

$$y(x; W) = W^T \phi(x)$$

Target variable is given by:

$$T = y(x; W) + \varepsilon$$

with $P(\varepsilon|\beta) = \mathcal{N}(\varepsilon|0, \beta^{-1} I)$. Similarly with before we obtain:

$$W_{ML} = (\phi^T \phi)^{-1} \phi^T$$

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.

9 Kernel methods

So far:
objects represented as fixed-length feature-vectors $x \in \mathbb{R}^M$ or $\phi(x)$.
Issue:
what about objects with variable length or infinite dimensions? Examples:

- strings
- trees
- image features
- time-series

Approach: use a similarity measure $k(x, x') \geq 0$ between the instances x, x'. $k(x, x')$ is called a kernel function. Note that if we have $\phi(x)$ a possible choice is $k(x, x') = \phi(x)^T \phi(x')$.

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

- **symmetric**: $k(x, x') = k(x', x)$
- **non-negative**: $k(x, x') \geq 0$

Note that is not strictly required.

Kernel families

- **Linear**: $k(x, x') = x^T x'$
- **Polynomial**: $k(x, x') = (\beta x^T x' + \gamma)^d, \;\; d \in 2, 3, ...$
- **Radial Basis Function (RBF)**: $k(x, x') = exp(-\beta |x - x'|^2)$
- **Sigmoid**: $k(x, x') = tanh(\beta x^T x' + \gamma)$

9.1 Kernelized linear models

Consider a linear model $y(x; w) = w^T x$ with dataset $D = (x_n, t_n)_{n=1}^N$. Minimize:

$$J(w) = (t - Xw)^T (t - Xw) + \lambda ||w||^2$$

$$X = \begin{pmatrix} x_1^T \\ \vdots \\ x_N^T \end{pmatrix} \qquad t = \begin{pmatrix} t_1 \\ \vdots \\ t_N \end{pmatrix}$$

Where $\mathbf{X}$ is the design matrix and $\mathbf{t}$ is the output vector. The optimal solution is:

$$\hat{w} = (X^T X + \lambda I_N)^{-1} X^T t = X^T (X X^T + \lambda I_N)^{-1} t$$

with I_N the $N * N$ identity matrix.

Let $\alpha = (XX^T + \lambda I_N)^{-1}t$, then $\hat{w} = X^T\alpha = \sum_{n=1}^{N} \alpha_n x_n$. Hence we have $y(x;\hat{w}) = \hat{w}^T x = \sum_{n=1}^{N} \alpha_n x_n^T x$. If we consider a linear kernel $k(x, x') = x^T x'$, we can rewrite the model as

$$y(x;\hat{w}) = \sum_{n=1}^{N} \alpha_n k(x_n, x)$$

with$\alpha = (k + \lambda I_N)^{-1}t$, and $K = XX^T$.

Linear model with any kernel k:

$$y(x;\alpha) = \sum_{n=1}^{N} \alpha_n k(x_n, x)$$

Solution:

$$\alpha = (K + \lambda I_N)^{-1}t$$

Gran matrix:

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

Example Linear model with any kernel k:

$$y(x;\alpha) = \sum_{n=1}^{N} \alpha_n x_n^T x$$

Solution:

$$\alpha = (K + \lambda I_N)^{-1}t$$

Gran matrix:

$$K = \begin{pmatrix} x_1^T x_1 & \dots & x_1^T x_N \\ \vdots & & \\ x_N^T x_1 & \dots & x_N^T x_N \end{pmatrix}$$

Kernel trick or kernel substitution If input vector x appears in an algorithm only in the form of an inner product $x^T x'$, replace the inner product with some kernel $k(x^T, x')$.

- Can be applied to any x (even infinite size)
- No need to know $\phi(x)$
- Directly extend many well-known algorithms

9.2 Kernelized SVM - classification

Solution has the form:

$$\hat{w} = \sum_{n=1}^{N} \alpha_n x_n$$

Linear model (with linear kernel):

$$y(x;\alpha) = sign(w_0 + \sum_{n=1}^{N} \alpha_n x_n^T x)$$

Kernel trick

$$y(x;\alpha) = sign(w_0 + \sum_{n=1}^{N} \alpha_n k(x_n, x))$$

Note: w_0 also estimated from α.
Lagrangian problem for kernelized SVM classification:

$$\tilde{L}(a) = \sum_{n=1}^{N} a_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} a_n a_m t_n t_m k(x_n, x_m)$$

Solution:

$$a_n = \dots$$

$$w_0 = \frac{1}{|SV|} \sum_{x_i \in SV} (t_i - \sum_{x_j \in S} a_j t_j k(x_i, x_j))$$

9.3 Kernelized linear regression

Linear model for regression $y = w^T x$ and data set $D = \{(x_n, t_n)_{n=1}^N\}$. Minimize the regularized loss function:

$$J(w) = \sum_{n=1}^{N} E(y_n, t_n) + \lambda ||w||^2$$

where $y_n = w^T x_n$.

Consider $E(y_n, t_n) = (y_n - t_n)^2$: i.e., regularized linear regression.
Solution

$$\hat{(}w) = (X^T X + \lambda I_M)^{-1} X^T t = X^T \alpha$$

Predictions are made using:

$$y(x; \hat{w}) = \sum_{n=1}^{N} \hat{\alpha}_n x_n^T x$$

Apply the kernel trick:

$$y(x; \hat{w}) = \sum_{n=1}^{N} \alpha_n k(x_n, x) \quad with \;\; \alpha = (K + \lambda I_N)^{-1} t$$

Due to K all data points are involved and α is not sparse.

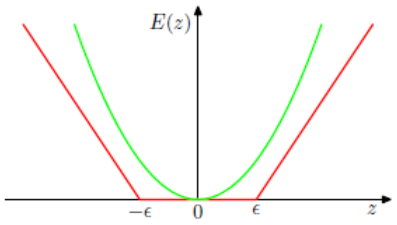
9.4 Kernelized SVM - regression

Consider

$$J(w) = C \sum_{n=1}^{N} E_\varepsilon(y_n, t_n) + \frac{1}{2} ||w||^2$$

with C inverse of λ and an ε-insensitive error function.

$$E_\varepsilon(y, t) = \begin{cases} 0 & \text{if } |y - t| < \varepsilon \\ |y - t| - \varepsilon & \text{otherwise} \end{cases}$$



Not differentiable → difficult to solve.

Figure 22: Kernelized SVM - regression

Introduce slack variables $\xi_n^+, \xi_n^- \geq 0$:

$$t_n \leq y_n + \varepsilon + \xi_n^+$$
$$t_n \geq y_n - \varepsilon - \xi_n^-$$

Points inside the ε-tube $y_n - \varepsilon \leq t_n \leq y_n + \varepsilon = 0$.

$$\xi_n^+ > 0 \Longrightarrow t_n > y_n + \varepsilon$$
$$\xi_n^- > 0 \Longrightarrow t_n < y_n - \varepsilon$$

with $y_n = y(x_n; w)$.

Loss function can be rewritten as:

$$J(w) = C \sum_{n=1}^{N} (\xi_n^+ + \xi_n^-) + \frac{1}{2}||w||^2$$

subject to the constraints:

- $t_n \leq y(x_n; w) + \varepsilon + \xi_n^+$
- $t_n \geq y(x_n; w) - \varepsilon - \xi_n^-$
- $\xi_n^+ \geq 0$
- $\xi_n^- \geq 0$

This is a standard quadratic program (QP), can be "easily" solved.

Lagrangian problem:

$$\tilde{L}(a, a') = \cdots \sum_{n=1}^{N} \sum_{m=1}^{N} a_n a_m \ldots k(x_n, x_m) \ldots$$

from which we compute $\hat{a}_n$, $\hat{a}'_m$ (sparse values, most of them are zero) and

$$\hat{w}_0 = t_n - \varepsilon - \sum_{m=1}^{N} (\tilde{a}_m - \hat{a}'_m) k(x_n, x_m)$$

from some data n such that $0 < a_n < C$.

Prediction

$$y(x) = \sum_{n=1}^{N} (\hat{a}_n - \hat{a}'_n) k(x_n, x_m) + \hat{w}_0$$

From **Karush-Kuhn-Tucker (KKT) condition** support vectors contribute to predictions:
subject to the constraints:

- $\hat{a}_n > 0 \Longrightarrow \varepsilon + \xi_n + y_n - t_n = 0$; data point lies on or above upper boundary of the ε-tube
- $\hat{a}'_n > 0 \Longrightarrow \varepsilon + \xi_n - y_n + t_n = 0$: data point lies on or below lower boundary of the ε-tube

All other data points inside the ε-tube have $\hat{a}_n = 0$ and $\hat{a}'_n = 0$ and thus do not contribute to prediction.

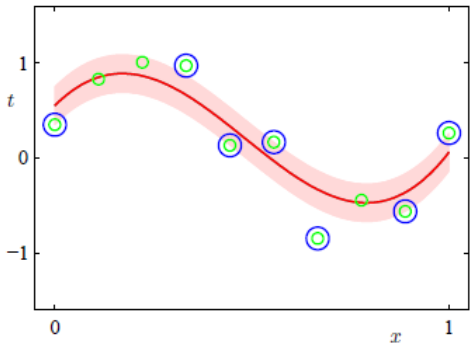


Figure 23: Example: support vectors and ε insensitive tube

9.5 Summary

- Kernel methods overcome difficulties in defining non-linear models
- Kernelized SVM is one of the most effective ML method for classification and regression
- Still requires model selection and hyper-parameters tuning

10 Instance based learning

Parametric model : has a fixed number of parameters. Examples: linear regression, logistic regression, perceptron.

Non-parametric model : number of parameters grows with amount of data.

10.1 K-nearest neighbours

It's a simple non-parametric model: instance-based learning. Classification with K-NN (target $f : X \to C$, data set $D = \{(x_i, t_i)_{i=1}^n\}$):

10.2 Summary

- find K nearest neighbours of new instance x
- assign to x the most common label among the majority of neighbours

Likehood of class c for new instance x:

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

with $N_K(x, D)$ the K nearest points to x and $\mathbb{I}(e) = \begin{cases} 1 \text{ if } e \text{ is true} \\ 0 \text{ if } e \text{ is false} \end{cases}$ Requires storage of all data set!

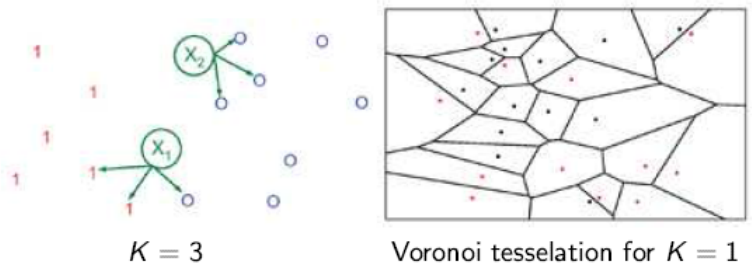


Figure 24: Example: k-nearest neighbours

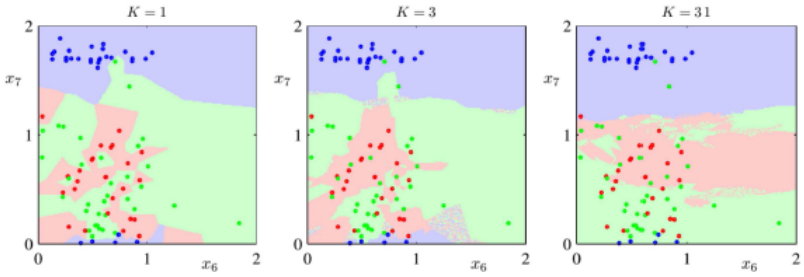


Figure 25: Increasing K brings to smoother regions (reducing overfitting).

Distance function in computing $N_K(x, D)$:

$$||x - x_i||^2 = x^T x + x_i^T x_i - 2x^T x_i$$

can be kernelized by using a kernel $k(x, x_i)$.

10.3 Locally weighted regression

Regression problem $f : X \to \mathbb{R}$ with data set $D = \{(x_i, t_i)_{i=1}^n\}$. Fit a local regression model around the query sample x_q:

1. compute $N_K(x_q, D)$: K-nearest neighbours of x_q
2. fit a regression model $y(x; w)$ on $N_K(x_q, D)$
3. return $y(x_q; w)$

10.4 Summary

- non-parametric models based on storing data (lazy approaches)
- no explicit model
- require storage of all data