Machine Learning Notes

University of Trento

a.y. 2023/2024

Contents

Chapter 1

Introduction

*A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P, if its performance at task in T, as measured by P, improves with experience E.*

Machine learning is the study of computer algorithms that improve automatically through experience. The peculiarity of machine learning algorithms is their ability to learn without being explicitly programmed. Due to these characteristics, ML is considered a component of the wider artificial intelligence field. The environments of machine learning techniques application are never ending. Nowadays hot topics include speech recognition, optical character recognition, computer vision, autonomous driving, game playing, recommender systems (e.g. Amazon's "suggested for you" sections) and so on.

Components of a well-posted learning problem

A problem solvable through machine learning techniques relies on three well defined entities:

- the **task** that we are trying to solve (e.g. recognizing handwritten characters),
- a **performance measure** with which we will evaluate the performance of the algorithm (e.g. miss-classified items, scoring system...),

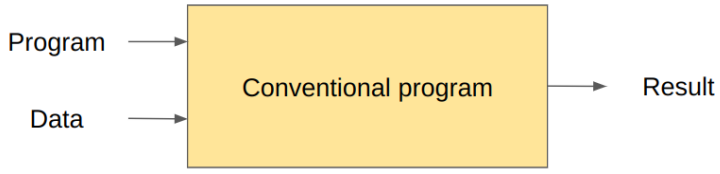


Figure 1.1: Conventional programming.

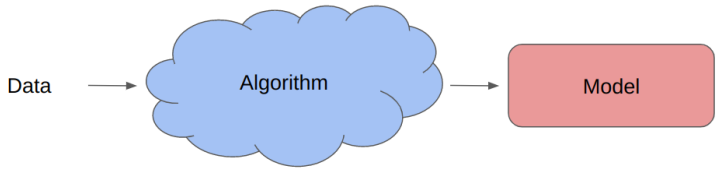


Figure 1.2: ML allows computers to **acquire knowledge**. Knowledge acquired is represented by a **model**. The model is used for **future** data.

- the **training experience** on which the system relies on to learn (e.g. data with annotated solutions, labelled handwritten characters).

1.1 Design of a machine learning system

A proper design of a machine learning system follows six steps:

1. formalize the learning task
2. collect data
3. extract features
4. choose the class of learning models
5. train the model
6. evaluate the model

Formalize the learning task

This is a overall definition of the goal we are trying to achieve. This could also be composed by a chain of separate tasks.
Along the definition of the aim, it is required to define an appropriate performance measure for evaluating the system.
With this first process, we get a formal definition of the problem.
Example: recognizing handwritten characters from images.

Collect data

A machine learning system requires data, collected in machine readable format, on which the algorithm will rely for training. This is a delicate and difficult process because could imply manual intervention for labelling data.
Not every machine learning technique requires annotation of data (semi-supervised and unsupervised learning).

Extract features

The process of feature extraction consists in representing data in a way in which a computer can work with, this implies codifying data into other representative data. Prior knowledge is often needed in order to extract significant features. The number of features chosen has a relevant impact on the performances: too many features can require a number of training data much greater, too few could miss relevant information and decrease the performance. Moreover, by taking into account too many features, there is the risk of including noisy features which can make the learning problem harder.

Choose the class of learning models

The choice of the model can heavily impact performance, a simple model classifier could be insufficient to complete the task while a complex model could lead to *overfitting*.

Overfitting
Overfitting is a modeling error that occurs when a function is too closely aligned to a limited set of data points. This happens when in training we achieve top performance while with validation sets they get lower: this could mean that the training took into account also noise among the training data.

Train the model

Training a model implies searching through the space of the possible models given a model class while performance of the model is measured. A trained model should be able to **generalize** (perform well on unseen data) and avoid overfitting 1.1. Several techniques can be exploited to improve generalization.

Evaluate the model

The evaluation of the model is based on data which the algorithm has never seen before. This process identifies weaknesses and give hints on how to improve the model. Appropriate statistical test need to be carried on to verify significance of the observed results.

1.2 Learning settings

1.2.1 Types of learning methods

Before starting to build a model, it is necessary to choose a type of learning. Often different data or goals require different learning settings.

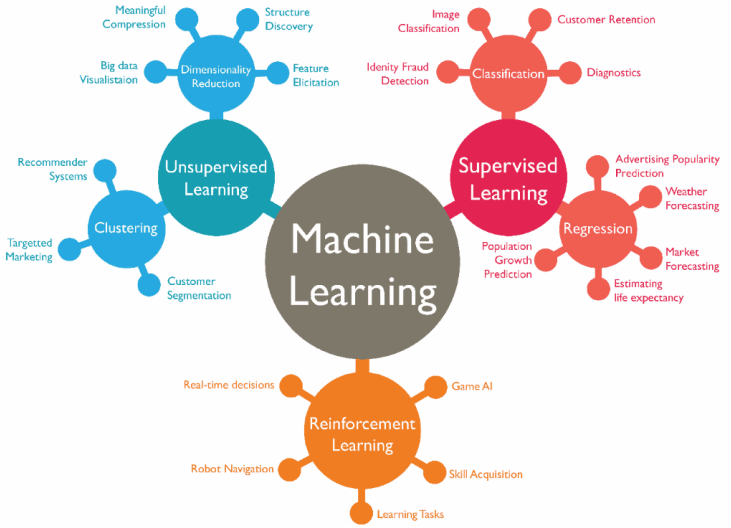


Figure 1.3: Main types of learning methods

Supervised learning
The learner is provided with a set of input and output pairs (labelled data) $(x_i, y_i) \in \mathcal{X} \times \mathcal{Y}$.
The learned model $f : \mathcal{X} \to \mathcal{Y}$ should map input samples into the proper output. A domain expert is often required in order to get labelled inputs.

Unsupervised learning
The learner is provided with a set of input but no annotation $(x_i) \in \mathcal{X}$.
The learner models training example, e.g. by grouping them into clusters according to their similarity. These algorithms discover hidden patterns or data groupings without the need for human intervention.

Semi-Supervised learning
The learner is provided with a set of input and output pairs (labelled data) $(x_i, y_i) \in \mathcal{X} \times \mathcal{Y}$ and a typical much bigger set of unlabelled data $(x_i) \in \mathcal{X}$.
The learned model $f : \mathcal{X} \to \mathcal{Y}$ should map input samples into the proper output like in supervised learning while unlabelled data can be exploited to improve performances (e.g. improve boundaries).

Reinforcement learning
The learner is provided with a set of possible states $\mathcal{S}$ and for each of them a set of possible action $\mathcal{A}$ is available in order to move through the next state. Performing an action a from a state s provides an immediate reward $r(s, a)$ that can be either immediate or delayed.
The goal is learning a policy that allows the algorithm to choose between states maximizing the overall reward. Often the model needs to face exploitation (performing moves that are known to be rewarding) and exploration (perform new moves) to avoid getting stuck in a local minimum.

1.2.2 Types of supervised learning tasks

Before starting to build a model, it is necessary to understand the goal. This implies understand what kind of classification we are trying to get once the problem is formalized.

Binary classification
The binary classification problem sets its goal into separate samples into two subset (often defined as positive and negative).

Multiclass classification
The multiclass classification assigns a sample to a class chosen between $n > 2$ different classes.

$$f : \mathbb{R}^d \to \{1, 2, \ldots, n\} \tag{1.1}$$

Multilabel classification
The multilabel classification assigns a sample to a subset of possible classes (not a unique assignment). E.g. predict the topics of a text.

Regression
This type of problems require to assign a real value to a sample.

$$f : \mathbb{R}^d \to \mathbb{R} \tag{1.2}$$

Ordinal regression or ranking
A ranking problem sets its goal into defining an order to samples according to their relevance/quality/importance.

1.2.3 Types of unsupervised learning tasks

Dimensionality reduction
In dimensionality reduction tasks we are trying to reduce the dimension of the data maintaining as much information as possible.

Clustering
For clustering we mean divide data into homologous groups according with some chosen similarity.

Novelty detection
Tasks in which we are trying to detect anomalies. It consists in studying the standard behaviour of the system in order to detect novelty or unusual events.

Probabilistic reasoning in presence of uncertainty is fundamental. Evaluating of performances related to different variables and estimate probabilities and relation between variables is often implied.

1.2.4 Types of learning algorithms

Based on the knowledge we have in the field of research, we adopt different strategies:

- **Bayesian decision theory**: when we are certain of the probability distribution of the data
- **Parameter estimation**: when we know the probability distribution but parameters need to be adjusted
- **Discriminative or generative methods**: when we have available training data but we do not know their distribution
- **Unsupervised methods**: when there is lack of data and also their distribution is unknown.

1.2.5 Discriminative and Generative Models

Machine learning models can be classified into two types of models: *discriminative* and *generative* models. These notions have not been explicitly covered during the course. However, it could be useful to briefly introduce the differences between the two in order to get a slightly better and more complete understanding of the following. In simple words, a discriminative model makes predictions on the unseen data based on conditional probability and can be used either for classification or regression problem statements ($P(y|x)$). On the contrary, a generative model focuses on the distribution of a dataset to return a probability for a given example ($P(x)$).

1.2.5.0.1 Discriminative models are not capable of generating new data points. Therefore, the ultimate objective of discriminative models is to separate one class from another (i.e. learn boundaries). Training discriminative classifiers involve estimating a function $f : X \to Y$, or probability $P(Y|X)$.

- Assume some functional form for the probability such as $P(Y|X)$.
- With the help of training data, we estimate the parameters of $P(Y|X)$.

Examples of discriminative models are:

- Support Vector Machine (SVMs)
- Traditional neural networks
- Nearest neighbor
- Decision Trees and Random Forest

1.2.5.0.2 Generative models are considered as a class of statistical models that can generate new data instances. These models are mainly used in unsupervised machine learning. Generative Adversarial Networks (GANs) are an example of generative machine learning model.
What is more, generative models are capable of finding the conditional probability $P(Y|X)$, just as the discriminative counterpart. They estimate the prior probability $P(Y)$ and likelihood probability $P(X|Y)$ with the help of the training data and uses the Bayes Theorem to calculate the posterior probability $P(Y|X)$.

$$P(Y|X) = \frac{P(Y) \cdot P(X|Y)}{P(X)} \tag{1.3}$$

As a result, generative models can tackle a more complex task than analogous discriminative models.

1.2.5.0.3 In essence, discriminative models draw boundaries in the data space, while generative models try to model how data is placed throughout the space. A generative model focuses on explaining how the data was generated, while a discriminative model focuses on predicting the labels of the data.

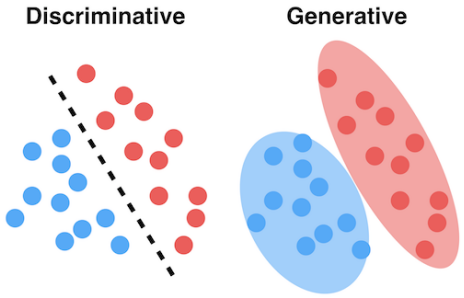


Figure 1.4: Discriminative *vs* Generative Models

Chapter 2

Decision Trees

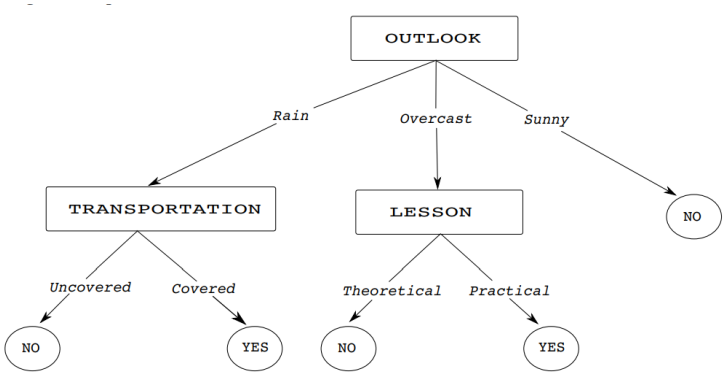


Figure 2.1: Toy example of a decision tree. *Go to lesson.*

Decision tree
A *decision tree* encodes a logical formula expressed in *disjunctive normal form*. It consists of a disjunction of conjuctions of constraints over attribute values. Each path from the root to a leaf is a conjunction of the constraints specified in the nodes along it. The leaf contains the label to be assigned to instances reaching it.

Given a decision three, such as the one reported in Figure 2.1, the disjunction of all paths is the logical formula represented by the tree.

$$(OUTLOOK = Rain \wedge TRANSPORTATION = Covered)$$
$$\vee$$
$$(OUTLOOK = Overcast \wedge LESSON = Practical)$$

Remark: the tree encodes a DNF formula for each class.

Appropriate problems for decision trees are:

- Binary or multiclass classification tasks. Extensions to regressions also exists. The most simple regression extension is to predict the average of the training examples which reached a leaf.
- Instances represented as attribute-value pairs. Indeed the idea is to split according to attribute values.
- Different explanations for the concept are possible (disjunction). For example looking at decision tree in Figure 2.1, there can be several reasons why I decide to attend a lecture. Decision trees allow to learn several explanations for a given concept.
- Some instances have missing attributes, i.e. for some attributes we do not have the value. This is typical in medical domain in which it is unlikely that a patient did all the possible tests which could be useful to make a prediction.
- There is need for an interpretable explanation of the output.

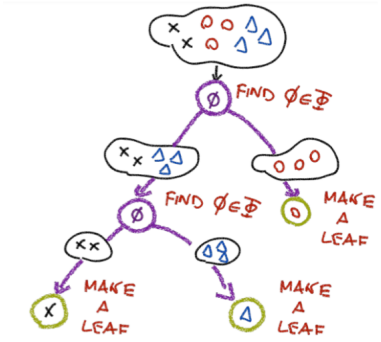


Figure 2.2: Split node training set into children according to the value of the chosen attribute

2.1 Learning decision trees

In order to learn a decision tree, we follow a *greedy* top-down strategy inspired by a *divide et impera* approach. For each node, starting from the root with full training set:

1. Choose the best attribute to be evaluated
2. Add a child for each attribute value

3. Split node training set into children according to the value of the chosen attribute
4. Stop splitting a node when a stopping condition is met. For example, the node which has been reached contains examples from a single class (all examples have the same label, they are homogeneous), or there are no more attributes to test.

Remark: if the current set of examples is not homogeneous but we stop splitting for other reasons, we assign to the leaf the majority class of the set.

In essence, the learning algorithm recursively partitions the training set and decides whether to grow leaves or non-terminal nodes.

2.1.1 Choosing the best attribute

Entropy
The *entropy* is a measure of the amount of information contained in a collection of instances (training examples) S which can take a number c of possible values (class labels). The entropy of a set of labelled examples measures its label inhomogeneity.

$$H(S) = -\sum_{i=1}^{c} p_i \log_2 p_i \tag{2.1}$$

where p_i is the fraction of S taking value i.

Information gain
Information gain is the expected reduction of entropy obtained by partitioning a set S according to the value of a certain attribute A.

$$IG(S, A) = H(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} H(S_v) \tag{2.2}$$

where $Values(A)$ is the set of possible values taken by A and S_v is the subset of S taking value v at attribute A.

- The second term represents the sum of entropies of subsets of examples obtained partitioning over A values, weighted by their respective sizes.
- An attribute with high information gain tends to produce homogeneous groups in terms of labels, thus favouring their classification.

Remark:

$$IG(S, A) \geq 0$$

Overall, the selection of the best splitting attribute is given in terms of maximization of information gain. The procedure ends when the subset of the training set which has reached a given node along the path from the root of the tree satisfies a certain criterion. For instance, the set of training examples which has reached the current node, is characterized by a homogeneity value lower then a given threshold ($H(S) < \epsilon$), has a given minimum cardinality ($|S| < k$), is completely homogeneous, there are no more splitting attributes to choose. Depending on the particular implementation, in some of these cases we grow a leaf instead of splitting the training examples and generating new intermediate nodes.

2.2 Issues in decision tree learning

2.2.1 Overfitting avoidance

Decision trees have a structure that is determined by the data. As a result they are flexible and can easily fit the training set, with high risk of overfitting.
Requiring that each leaf has only examples of a certain class can lead to very complex trees. A complex tree can easily overfit the training set, incorporating random regularities not representative of the full distribution. What is more many kinds of noise can occur in the training data: some values of attributes are incorrect because of errors in the data acquisition process, the instance was labeled incorrectly, some attributes are irrelevant to the decision making process. In order to build simpler tree structures, less prone to overfitting, it is possible to accept impure leaves, assigning them the label of the majority of their training examples.
A technique to reduce complexity is called *pruning*. There are two possible strategies to prune a decision tree:

- **pre-pruning**: decide whether to stop splitting a node even if it contains training examples with different labels.
- **post-pruning**: learn a full tree and successively prune it removing subtrees, replacing whole subtrees with leaf nodes.

With the aim of improving generalization and as a consequence, reducing overfitting, if a *validation set* is available it is possible to implement a post-pruning strategy. The procedure is described in Algorithm 1.

2.2.2 Dealing with continuous-valued attributes

Continuous valued attributes need to be discretized in order to be used in internal nodes tests. Discretization threshold can be chosen to maximize the information gain. A possible procedure is the following:

1. Examples are sorted according to their continuous attribute values.

Algorithm 1 Post pruning

```
1: for each node n in the tree do{
2:     Evaluate the performance on the validation set
3:     when removing the subtree rooted at n
4: }
5: if all node removals worsen performance{
6:     STOP
7: }
8:
9: n ← node whose removal has the best performance improvement
10:
11: Replace the subtree rooted at n with a leaf
12:
13: Assign to the leaf the majority label of all examples in the subtree
14:
15: return to line 1
```

2. For each pair of successive examples having different labels, a candidate threshold is placed as the average of the two attribute values.

3. For each candidate threshold, the information gain achieved by splitting the examples is computed.

4. The threshold producing the higher information gain is used to discretize the attribute.

2.2.3 Alternative attribute test measures

The information gain criterion tends to prefer attributes with a large number of possible values. As an extreme, the unique ID of each example is an attribute which perfectly splits the data into singletons, but it will be of no use on new examples. Moreover, this would lead to overfitting.

2.2.3.0.1 In order to deal with this problem a new measure of entropy is introduced. In this case we do not compute entropy taking into account class values, but we calculate the spread with respect to the values of the attributes.

$$H_A(S) = -\sum_{v \in Values(A)} \frac{|S_v|}{|S|} \log_2 \frac{|S_v|}{|S|} \tag{2.3}$$

This quantity is maximized when there exists an attribute which splits the dataset homogeneously in many subsets. The aim is to discourage the choice of these kind of attributes to perform the splits. To do this *gain ratio* is introduced as an alternative of information gain measure introduced above.

$$IGR(S, A) = \frac{IG(S, A)}{H_A(S)} \tag{2.4}$$

In this case, splitting according to unique identifiers would result in high values of entropy with respect to attribute values ($H_A(S)$), which leads to low values of gain ratio ($IGR(S, A)$).

2.2.4 Handling attributes with missing values

Assume that an example x with class $c(x)$ has missing value for attribute A. When attribute A has to be tested at node n, the following solutions could be adopted:

- *Simple solution.* Assign to x the most common attribute value among the examples in n or (during training) the most common attribute value among the examples in n with class $c(x)$.
- *Complex solution.* Propagate x to each of the children of n, with a fractional value equal to the proportion of examples with the corresponding attribute value.
  At training time, you assign the leaf the majority vote among the training examples which reach that leaf considering their weight.
  At test time, if we process an example with missing attributes it ends up in multiple leaves with a given probability. Each leaf votes with a weight which depends on the fraction of the test example which reached that leaf.

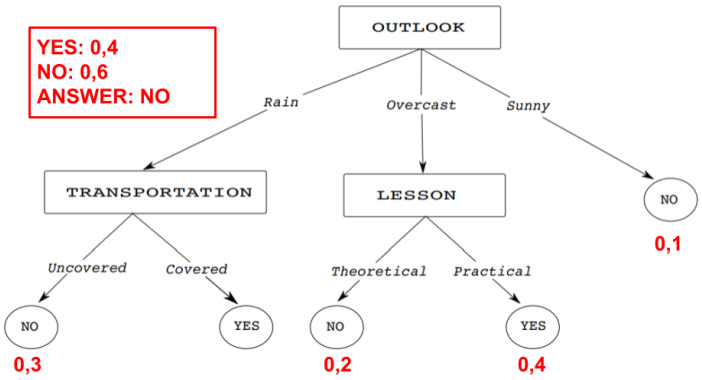


Figure 2.3: Dealing with missing attributes at test time. Complex solution.

2.3 Random forest

A common technique to reduce overfitting is *ensemble learning*. Following this method we learn multiple models at training time. At test time we average the results or take the majority vote of the models.

2.3.1 Bootstrap sampling and bagging

2.3.1.0.1 Bootstrap sampling: Given a set D containing N training examples, create D' by drawing N examples at random with replacement (i.e. same example can be selected multiple times) from D.

2.3.1.0.2 Bagging:

1. Create k bootstrap datasets D_i
2. Train distinct classifier on each D_i
3. Classify new instance by majority vote/average

2.3.2 Training

Random forests are ensembles of decision trees. Each tree is typically trained on a bootstrapped version of the training set (sampled with replacement, i.e. independent samples). Each decision tree is independently trained with its bootstrapped version of the training set. At each node the splitting function is optimized on m randomly sampled features. This helps obtaining decorrelated decision trees At the end of the training a forest with M trees is generated.

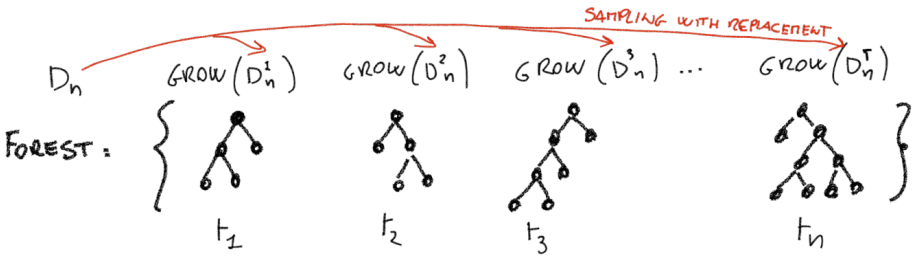


Figure 2.4: Random forest

2.3.3 Testing

1. Test the example with each tree in the forest.
2. Return the majority class among the predictions.

More formally, given a set of trees $Q = \{t_1, ..., t_T\}$, the final prediction of the forst is obtained by averaging the prediction of each tree in the ensemble:

$$f_Q(x) = \frac{1}{T} \sum_{j=1}^{T} f_t(x) \tag{2.5}$$

Chapter 3

K-nearest neighbors

K-Nearest Neighbors (kNN)
K-nearest neighbors algorithm is a non-parametric supervised learning method. It allows to define the belonging of an example to an area of influence of the data used for learning.

Remark: kNN is non-parametric since it does not make any assumptions on the data being studied, i.e., the model is distributed from the data.

Remark: kNN is a lazy algorithm because it does not use the training data points to make any generalisation. Which implies:

- You expect little to no explicit training phase,
- The training phase is pretty fast,
- kNN keeps all the training data since they are needed during the testing phase.

Given a training set, each point has its own area of influence, as the area in which is most likely to position new samples that can be labelled as the main point. Training data can be positioned in a multidimensional space, so the concept of distance we use is not the "trivial" one: euclidean distance is not always the best option. The k in **kNN** means that the number of closest neighbors we consider is equal to k (e.g. if $k = 3$ and two of them are points of a class A and only one belongs to a class B, then the point we are trying to classify will belong to class A, regardless of the position of the three points in training set).

Concept of distance (or metric)
Given a set $\mathcal{X}$ a function $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}_0^+$ is a metric for $\mathcal{X}$ if for any $x, y, z \in \mathcal{X}$ the following proprieties are satisfied:

- **reflexivity**: $d(x, y) = 0 \Leftrightarrow x = y$
- **symmetry**: $d(x, y) = d(y, x)$
- **triangle inequality**: $d(x, y) + d(y, z) \geq d(x, z)$

An example of a metric distance definition is the euclidean distance:

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

The pseudo-algorithm for classification problems can be formalized with Algorithm 2.

Algorithm 2 kNN for classification

```
for all test examples x do
  for all training examples (x_i, y_i) do
    compute distance d(x, x_i)
  end for
  select the k-nearest neighbors
  return class of x as majority class among k-neighbors
end for
```

The selection of the majority vote can be formalized as:

$$argmax_y \sum_{i=1}^{k} \delta(y, y_i) \tag{3.1}$$

The pseudo-algorithm for regression problems can be formalized with Algorithm 3.

Algorithm 3 kNN for regression

```
for all test examples x do
  for all training examples (x_i, y_i) do
    compute distance d(x, x_i)
  end for
  select the k-nearest neighbors
  return the average output value among neighbors
end for
```

The computation of the average value can be formalized as:

$$\frac{1}{k}\sum_{i=1}^{k} y_i$$

3.1 Characteristic of kNN learning

Here are reported the main features of the kNN algorithm:

- **instance-based learning**: the models results calibrated for the test example to be processed.
  Instance-based learning (sometimes called *memory-based learning*) is a family of learning algorithms that, instead of performing explicit generalization, compare new problem instances with instances seen in training, which have been stored in memory. In other words, this methods construct hypotheses directly from the training instances themselves. This means that the hypothesis complexity can grow with the data: in the worst case, a hypothesis is a list of n training items and the computational complexity of classifying a single new instance is $\mathcal{O}(n)$. One advantage that instance-based learning has over other methods of machine learning is its ability to adapt its model to previously unseen data. Instance-based learners may simply store a new instance or throw an old instance away.
  Because computation is postponed until a new instance is observed, these algorithms are sometimes referred to as *lazy*.

- **lazy learning**: computation is mostly deferred to the classification phase (speed up research)

- **local learner**: assumes prediction should be mainly influenced by nearby instances (an area is affected by a class)

- **uniform feature weighting**: all features are uniformly weighted in computing distances, no data point or distance affect the classification in any way since the given data point in the training set is considered in the set of nearest neighbours.

Anyhow, for the last point, a further implementation of the *kNN algorithm* is available: its upgrade consists in giving the data a different weight proportional to the inverse of the distance.

For the classification problem, the selection of the majority vote is:

$$argmax_y \sum_{i=1}^{k} w_i \delta(y, y_i) \tag{3.2}$$

while for the regression problem is:

$$\frac{\sum_{i=1}^{k} w_i y_i}{\sum_{i=1}^{k} w_i} \tag{3.3}$$

where

$$w_i = \frac{1}{d(x, x_i)} \tag{3.4}$$

Chapter 4

Linear algebra

Group
In abstract algebra, a group is a set G where an operation $*$ is defined. Moreover $*$ satisfies the following properties:

1. *associative*: $(a * b) * c = a * (b * c) \forall a, b, c \in G$
2. *identity element*: $\exists e \in G : a * e = e * a = a \forall a \in G$
3. *inverse element*: $\forall a \in G \; \exists a' \in G : a * a' = a' * a = e$

Commutative group
A group $(G, *)$ is *commutative* iff $a * b = b * a \; \forall a, b \in G$.

Vector space
A *vector space* over the field $\mathcal{R}$ is an algebraic structure composed by a commutative group $(V, +)$, whose elements are called *vectors*, and a function $f : \mathcal{R} \times V \to V$ called *scalar multiplication* which satisfies the properties listed below. It is common to use kv, instead of $f(k, v)$ to represent the scalar multiplication between a scalar $k \in \mathcal{R}$ and a vector $v \in V$.

- *distributive over elements* $k(v_1 + v_2) = kv_1 + kv_2$
- *distributive over scalars* $(k_1 + k_2)v = k_1 v + k_2 v$
- *associative over scalars* $(k_1 k_2)v = k_1(k_2 v)$
- *identity element* $1v = v$

$\forall k, k_1, k_2 \in \mathcal{R}, \forall v, v_1, v_2 \in V$.

Vector space examples are: the set of two-dimensional vectors V^2, the set of three-dimensional vectors V^3, the matrix space $M_{m,n}(\mathcal{R})$.

Subspace
Given a vector space V defined over the field R, a non-empty set $W \subseteq V$ is a *subspace* of V if it is close w.r.t. the sum of vectors and the scalar product:

$$\forall w, w' \in W, \forall k \in K, w + w' \in W \wedge kw \in W$$

Each subspace is in turn a vector space with respect to the operations defined in V.

Linear combination
Given a vector $v \in V$, v is a linear combination of the vectors $v_1, ..., v_k \in V$ with coefficients $c_i \in \mathcal{R}$ if:

$$v = c_1 v_1 + ... + c_k v_k = \sum_{i=1}^{K} c_i v_i$$

Span
The span of vectors $v_1, ..., v_k$ is defined as the set of their linear combinations:

$$\{\sum_{i=1}^{K} c_i v_i, c_i \in \mathcal{R}\} =< v_1, ..., v_k >$$

The span is a *vector subspace* (also known as *linear subspace*) of V. In particular the span $< v_1, ..., v_k >$ is the subset generated by $v_1, ..., v_k$.

Linear independency and linear dependency
A set of vectors $v_1, ..., v_k$ is *linearly independent* if the null vector can be written as a linear combination of the elements v_i of the set

$$a_1 v_1 + a_2 v_2 + ... + a_k v_k = 0$$

if and only if all the coefficients $a_1, ..., a_k$ are null. Intuitively, a set of vectors $v_1, ..., v_k$ is linearly independent if none of them can be written as a linear combination of the others.
A set of vectors $v_1, ..., v_k$ is *linearly dependent* is it is not linearly independent. As a consequence, a set of vectors $v_1, ..., v_k$ is linearly dependent if there exist some not-null scalars $a_1, ..., a_k$ such that:

$$a_1 v_1 + a_2 v_2 + ... + a_k v_k = 0$$

Basis
A set of vectors $\mathcal{B} = \{v_1, ..., v_k\}$ is a *basis* for V if any element in V can be uniquely written as a linear combination of vectors v_i. A necessary condition is that vectors v_i are linearly independent. All bases of V have the same number of elements, called the *dimension* of the vector space.

Linear maps
Given two vector spaces V and V', a function $f : V \to V'$ is a *linear map* if $\forall a_1, a_2 \in \mathcal{R}, v_1, v_2 \in V$:

$$f(a_1 v_1 + a_2 v_2) = a_1 f(v_1) + a_2 f(v_2)$$

A linear map between two finite-dimensional spaces V, V' of dimensions n,m can always be written as a matrix A. Consider a linear map $T : V \to V'$ and two sets $\mathcal{B} = \{u_1, ..., u_n\}$, $\mathcal{C} = \{v_1, ..., v_m\}$ which are basis of V and V' respectively.

Consider also two isomorphisms $T_{\mathcal{B}} : V \to \mathcal{R}^n$, $T_{\mathcal{C}} : V' \to \mathcal{R}^m$ which associate to each vector its coordinates with respect to a given base. These two isomorphisms uniquely identify a matrix $A \in M_{m,n}(\mathcal{R})$. The n columns of A are the vectors $T_{\mathcal{C}}(T(u_j))$ obtained from the coordinates of the images $T(u_1), ..., T(u_n)$, with respect to basis $\mathcal{C}$.

For any $x \in V$ we have:

- Each element $x \in V$ can be written as a linear combination of the elements of the basis $\mathcal{B}$.

$$f(x) = f(\sum_{i=1}^{n} \lambda_i u_i) = \sum_{i=1}^{n} \lambda_i f(u_i)$$

- Given that the n columns of A are the vectors $T_{\mathcal{C}}(T(u_j))$ obtained from the coordinates of the images $T(u_1), ..., T(u_n)$, with respect to basis $\mathcal{C}$.

$$f(u_i) = \sum_{j=1}^{m} a_{ji} v_j$$

- Putting all together we notice that actually the image of $f(x)$ can be written as a linear combination of the elements of the basis $\mathcal{C}$. As a consequence, $f(x)$ belongs to the vector space V'.

$$f(x) = \sum_{i=1}^{n} \lambda_i f(u_i) = \sum_{i=1}^{n} \sum_{j=1}^{m} \lambda_i a_{ji} v_j = \sum_{j=1}^{m} (\sum_{i=1}^{n} \lambda_i a_{ji}) v_j = \sum_{j=1}^{m} \mu_j v_j$$

The matrix $A \in M_{mn}(\mathcal{R})$ defined above is called the matrix associated with the linear map T with respect to the basis $\mathcal{B}$, $\mathcal{C}$. A is typically indicated with the symbol $M_{\mathcal{B}}^{\mathcal{C}}(T)$. If $V = V'$ and $\mathcal{B} = \mathcal{C}$, then we write $M_{\mathcal{B}}(T)$. If $V = \mathcal{R}^n$, $V' = \mathcal{R}^m$ and we consider the canonical basis of the two vector spaces, then we simply write $M(T)$.

Consider a linear map T and a matrix $A = M_{\mathcal{B}}^{\mathcal{C}}(T)$ defined as described above. Consider also a vector $v \in V$ and a vector x. This latter is the column vector of the coordinates $x_1, ..., x_n$ of v with respect to the basis $\mathcal{B}$. Given that, the image of $T(v)$ has a corresponding vector of coordinates with respect to basis $\mathcal{C}$ which can be calculated by the matrix multiplication Ax. We propose an example in Figure 4.1.

$T(x) = (2x_1 - x_2 + x_4,\ x_2 - 2x_3 + x_4,\ x_1 + x_2 - 2x_3)$

- B basis of $\mathbb{R}^4$: $B = \{(1,0,0,0), (0,1,-1,0), (0,0,0,1), (0,1,1,1)\}$
- C basis of $\mathbb{R}^3$: $C = \{(1,1,1), (1,1,0), (1,0,0)\}$

$$A = \begin{bmatrix} 2 & -1 & 0 & 1 \\ 0 & 1 & -2 & 1 \\ 1 & 1 & -2 & 0 \end{bmatrix} \equiv \text{matrix associated with } T$$

TASK: COMPUTE THE MATRIX $M_B^C(T)$

i. CALCULATE THE FOUR IMAGES

$T(1,0,0,0) = (2,0,1)$; $T(0,1,-1,0) = (-1,3,3)$
$T(0,0,0,1) = (1,1,0)$; $T(0,1,1,1) = (0,0,-1)$

ii. FIND THE COORDINATES OF $(2,0,1)$, $(-1,3,3)$, $(1,1,0)$, $(0,0,-1)$ WITH RESPECT TO BASIS C

Solving a linear system we get: $T_C(2,0,1) = (1,-1,2)$
$T_C(-1,3,3) = (3,0,-4)$
$T_C(1,1,0) = (0,1,0)$
$T_C(0,0,-1) = (-1,1,0)$

iii. Put the results obtained in ii inside column vectors

$$M_B^C(T) = M = \begin{bmatrix} 1 & 3 & 0 & -1 \\ -1 & 0 & 1 & 1 \\ 2 & 4 & 0 & 0 \end{bmatrix}$$

Figure 4.1: Example of a computation of the matrix associated with a linear function

Matrix of basis transformation
Consider two basis of the vector space V: $\mathcal{B} = \{v_1, ..., v_n\}$ and $\mathcal{B}' = \{v'_1, ..., v'_n\}$. The *transition matrix* (or *matrix of basis transformation*) from $\mathcal{B}$ to $\mathcal{B}'$ is the invertible matrix $P = [p_{ij}]$, with order n, whose columns are the coordinates of the vectors v_j of the basis $\mathcal{B}$ with respect to the basis $\mathcal{B}'$.

$$v_j = \sum_i p_{ij} v'_i \quad (j = 1, ..., n)$$

Remark: The matrix P is the matrix associated with the identity function with respect to the basis $\mathcal{B}$ and $\mathcal{B}'$: $P = M_{\mathcal{B}}^{\mathcal{B}'}(id_V)$. From this observation we can derive the following consequence. If the vector $v \in V$ has coordinates $x_1, ..., x_n$ with respect to basis $\mathcal{B}$, and coordinates $x'_1, ..., x'_n$ with respect to basis $\mathcal{B}'$, then:

$$x' = Px \tag{4.1}$$

where P is the matrix of basis transformation from $\mathcal{B}$ to $\mathcal{B}'$. We propose an example of basis transformation in $\mathcal{R}^2$ in Figure 4.2.

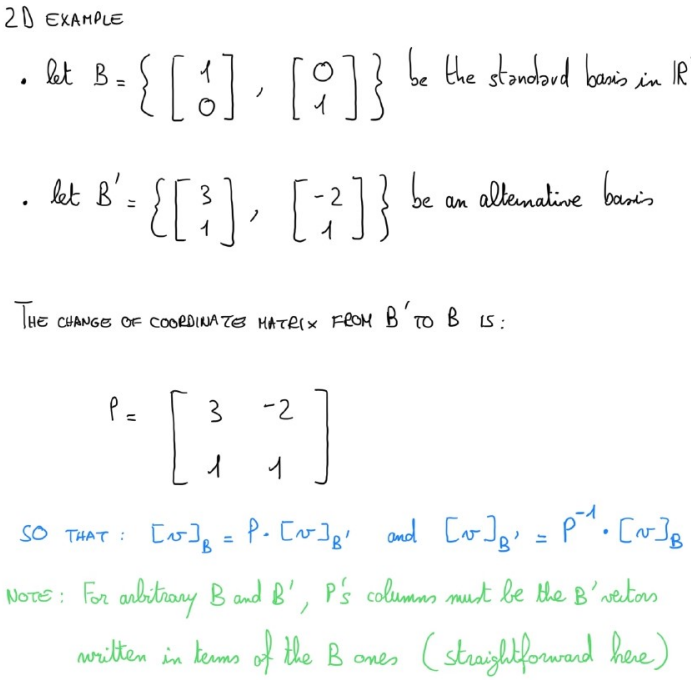


Figure 4.2: 2D example of basis transformation.

Transpose
Matrix obtained exchanging rows with columns (indicated with M^T). Given two matrices M,N and their transposes M^T, N^T, the following relevant property holds:

$$(MN)^T = N^T M^T$$

Trace
Sum of diagonal elements of a matrix.

$$tr(M) = \sum_{i=1}^{n} M_{ii}$$

Inverse
The matrix which multiplied with the original matrix gives the identity.

$$MM^{-1} = I$$

Rank
The rank of a $n \times m$ matrix is the dimension of the space spanned by its columns. Consider a matrix A $m \times n$. In the matrix there are n columns $A^1, ..., A^n$. It is possible to demonstrate that the column space $< A^1, ..., A^n >$ has dimension equal to the rank of the matrix A.

$$rg(A) = dim(< A^1, ..., A^n >)$$

In particular the columns of A are linearly independent if and only if $rg(A) = n$.

4.1 Matrix derivatives

In this section, there are reported some useful properties of matrix calculus.

$$\begin{aligned}
\frac{\partial M\mathbf{x}}{\partial \mathbf{x}} &= M \\
\frac{\partial \mathbf{y}^T M\mathbf{x}}{\partial \mathbf{x}} &= M^T\mathbf{y} \\
\frac{\partial \mathbf{x}^T M\mathbf{x}}{\partial \mathbf{x}} &= \left(M^T + M\right)\mathbf{x} \\
\frac{\partial \mathbf{x}^T M\mathbf{x}}{\partial \mathbf{x}} &= 2M\mathbf{x} \quad \text{if } \mathbf{M} \text{ is symmetric} \\
\frac{\partial \mathbf{x}^T\mathbf{x}}{\partial \mathbf{x}} &= 2\mathbf{x}
\end{aligned}$$

Note: Results are column vectors. Transpose them if row vectors are needed instead.

4.2 Metric structure

Norm
The *norm* defined in $\mathcal{R}^n$ is a function $||\cdot|| : \mathcal{X} \to \mathcal{R}_0^+$ which associates to the vector x the non-negative real number:

$$||x|| = \sqrt{x \cdot x} = \sqrt{x_1^2 + ... + x_n^2} \tag{4.2}$$

The norm has the following properties:

- $||x + y|| \leq ||x|| + ||y||$
- $||\lambda x|| = |\lambda| ||x||$
- $||x|| > 0$ if $x \neq 0$

where $x, y \in \mathcal{X}$, $\lambda \in \mathcal{R}$

Metric
A norm defines a *metric* $d : \mathcal{X} \times \mathcal{X} \to \mathcal{R}_0^+$ such that:

$$d(x, y) = ||x - y|| \tag{4.3}$$

This metric is commonly referred to as *distance*. This latter is a function which associates to each couple of vectors $x, y \in \mathcal{R}^n$ the real number $d(x, y)$.

Remark: The concept of norm is stronger than that of metric: not any metric gives rise to a norm.

Unit vector
A *unit vector* is a vector $v \in \mathcal{R}^n$ such that:

$$||v|| = 1$$

Remark: each vector $v \neq \mathbf{0}$ can be normalized:

$$v' = \frac{v}{||v||}$$

where v' is a unit vector.

4.3 Dot product

Bilinear form
A function $Q : \mathcal{X} \times \mathcal{X} \to \mathcal{R}$ on a vector space X is a *bilinear form* if it is linear in each argument separately:

$$Q(\lambda x + \mu y, z) = \lambda Q(x, z) + \mu Q(y, z)$$

$$Q(x, \lambda y + \mu z) = \lambda Q(x, y) + \mu Q(x, z)$$

where $x, y, z \in \mathcal{X}$, $\lambda, \mu \in \mathcal{R}$.

A bilinear form is *symmetric* if for all $x, y \in \mathcal{X}$:

$$Q(x, y) = Q(y, x)$$

Dot product
The *dot product* on $\mathcal{R}^n$ is a function $< \cdot, \cdot >: \mathcal{X} \times \mathcal{X} \to \mathcal{R}$ which associate to a couple of vectors $x, y \in \mathcal{R}^n$ the real number:

$$< x, y >= x \cdot y = \sum_{i=1}^{n} x_i y_i = x^T y \tag{4.4}$$

The dot product has the following properties:

1. *symmetric*: $x \cdot y = y \cdot x$ for all $x, y \in \mathcal{R}^n$
2. *bilinear*: $(\alpha x + \beta y) \cdot z = \alpha(x \cdot z) + \beta(y \cdot z)$ and $x \cdot (\alpha y + \beta z) = \alpha(x \cdot y) + \beta(x \cdot z)$, for all $x, y, z \in \mathcal{R}^n$ and $\alpha, \beta \in \mathcal{R}$
3. *symmetric bilinear*: this property follows from the previous ones
4. *positive semi-definite*: $x \cdot x \geq 0$ for all $x \in \mathcal{R}^n$
5. The dot product could also be *positive definite* if it satisfies: $x \cdot x = 0$ iff $x = 0$ for all $x \in \mathcal{R}^n$

Any dot product defines a corresponding *norm* via:

$$||x|| = \sqrt{< x, x >}$$

- The angle θ between two vectors is defined as:

$$\cos\theta = \frac{< x, z >}{||x||||z||} \tag{4.5}$$

- Two vectors are *orthogonal* if $< x, y >= 0$.

- A set of vectors $\{x_1, ..., x_n\}$ is orthonormal if all vectors x_i in the set are mutually orthogonal and are all unit vectors:

$$< x_i, x_j >= \delta_{ij}$$

where $\delta_{ij} = 1$ if $i = j$, 0 otherwise.

4.4 Eigenvalues and eigenvectors

Eigenvalues and eigenvectors
Given a $n \times n$ matrix M, the real value λ and (non-zero) vector x are an *eigenvalue* and corresponding *eigenvector* of M if:

$$Mx = \lambda x \tag{4.6}$$

Properties:

- A $n \times n$ matrix has n eigenvalues
- A $n \times n$ matrix can have less than n distinct eigenvalues
- A $n \times n$ matrix can have less than n linear independent eigenvectors (also fewer then the number of distinct eigenvalues)

Singular matrices
A matrix is *singular* if it has a zero eigenvalue.

$$Mx = 0x = 0$$

A singular matrix has linearly dependent columns (see Definition 4):

$$[M_1 ... M_{n-1} M_n] \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = 0$$

$$M_1 x_1 + \ldots + M_{n-1} x_{n-1} + M_n x_n = 0$$

$$M_n = M_1 \frac{-x_1}{x_n} + \ldots + M_{n-1} \frac{-x_{n-1}}{x_n}$$

Determinant
The *determinant* $|M|$ of a $n \times n$ matrix M is the product of its eigenvalues.

Remark: a matrix is *invertible* if its determinant is not zero (i.e. it is not singular).

Symmetric matrix
A *symmetric matrix* is a square matrix that is equal to its transpose.

$$A = A^T$$

Because equal matrices have equal dimensions, only square matrices can be symmetric. The entries of a symmetric matrix are symmetric with respect to the main diagonal. So if a_{ij} denotes the entry in the ith row and jth column then:

$$a_{ji} = a_{ij}$$

for all indices i and j.

If A is a real symmetric matrix, then any two eigenvectors (x, z) corresponding to distinct eigenvalues $(\lambda \neq \mu)$ are orthogonal. (x, λ) and (z, μ) IF $\lambda \neq \mu \Rightarrow < x, z >= 0$. The proof of this preposition is reported in Figure 4.3.

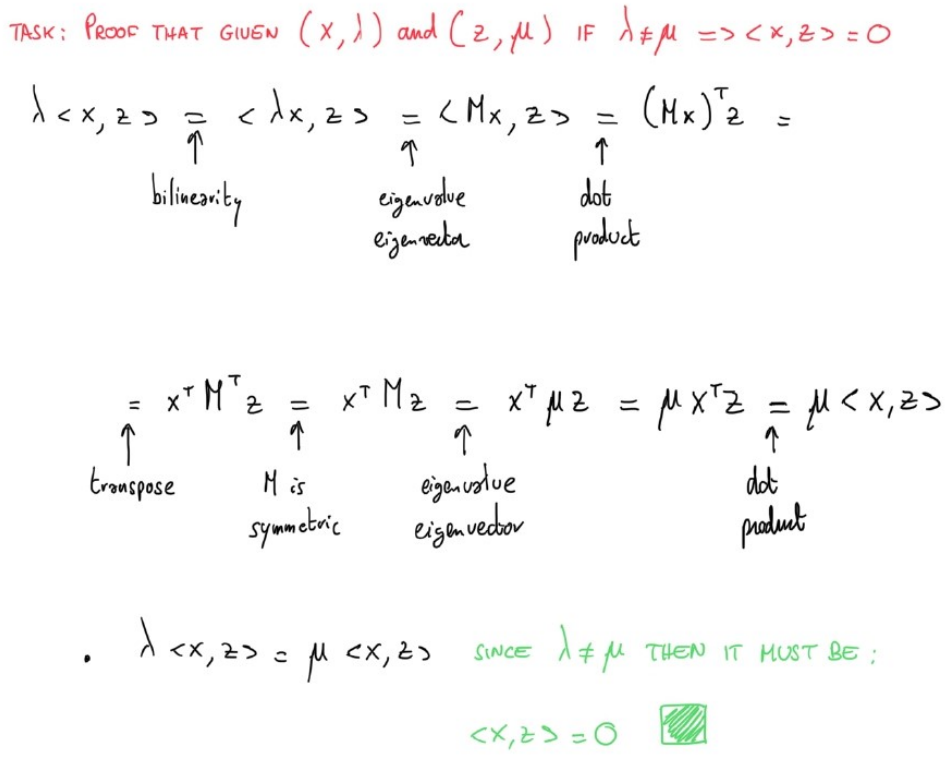


Figure 4.3: Proof that (x, λ) and (z, μ) IF $\lambda \neq \mu \Rightarrow < x, z >= 0$.

4.5 Eigen-decomposition

The purpose of this section is to describe an iterative procedure to find eigenvalues and eigenvectors for a matrix A. Given A we want to find λ and x values such that:

$$Ax = \lambda x$$

We multiply both sides of the equation by x^T and we divide both sides of the equation by $x^T x$. We are not interested in the 0 vector, so the dot product $x^T x$ is not 0.

$$\frac{x^T Ax}{x^T x} = \frac{x^T \lambda x}{x^T x}$$

For linearity we can move λ.

$$\frac{x^T Ax}{x^T x} = \lambda \frac{x^T x}{x^T x} = \lambda$$

The fraction $\frac{x^T Ax}{x^T x}$ is called *Raleigh quotient*.

4.5.0.0.1 At this point we want to find the eigenvector which maximize the *Raleigh quotient*. In this way we get the eigenvector (x) which corresponds to the maximal eigenvalue.

$$x = argmax_v \frac{v^T Av}{v^T v}$$

Now we normalize the obtained eigenvector x:

$$x \leftarrow \frac{x}{||x||}$$

Note that a normalized eigenvector is still an eigenvector.

In this way we have obtained the first eigenvalue and the first normalized eigenvector.

4.5.0.0.2 In order to compute the other eigenvectors and eigenvalues, we modify A such that, if we repeat the previous procedure, we find another eigenvector. This modification is called *deflation*.

$$\tilde{A} = A - \lambda x x^T$$

Note that xx^T is not $x^T x$ which is a scalar (dot product). Indeed xx^T is a matrix.

Deflation turns x into an eigenvector for $\tilde{A}$ which has zero-eigenvalue:

$$\tilde{A}x = (A - \lambda x x^T)x = Ax - \lambda x x^T x$$

Since x is normalized $x^T x = 1$, so:

$$\tilde{A}x = Ax - \lambda x$$

Since λ and x is an eigenvalue-eigenvector pair $Ax = \lambda x$, so:

$$\tilde{A}x = Ax - \lambda x = 0$$

We can conclude that x is an eigenvector with 0 eigenvalue:

$$\tilde{A}x = 0x$$

4.5.0.0.3 At this point, we can repeat the maximization procedure on the deflated matrix. In particular, we maximize the Raleigh quotient $\frac{v^T \tilde{A} v}{v^T v}$. In this way we can find the maximal possible eigenvalue. For sure, we will not find x since its corresponding eigenvalue is 0. We can demonstrate that after the deflation operation, other eigenvalues-eigenvectors are unchanged:

$$\tilde{A}z = (A - \lambda x x^T)z = Az - \lambda x x^T z$$

As stated in Section 4.4, in the case of symmetric matrices, eigenvectors with distinct eigenvalues are orthogonal, so:

$$\tilde{A}z = Az - \lambda x x^T z = Az$$

As a conclusion, for a symmetric matrix, dealing with $\tilde{A}$ is like working with A.

4.5.0.0.4 At the end of the day:

$$x' = argmax_v \frac{v^T \tilde{A} v}{v^T v}$$

where x' is the eigenvector with the second largest eigenvalue. The procedure is iteratively repeated on the deflated matrix until solution is zero, which means that there are no more positive eigenvalues. If the matrix has negative eigenvalues, I can recover them minimizing the Raleigh quotient instead of maximizing it. At some point the solution of the iterative procedure will be 0 again. If the obtained set of eigenvalues is not full rank yet, we have to take into consideration the eigenvectors with zero eigenvalues. The eigenvectors with zero eigenvalues are obtained extending the obtained set to an orthonormal basis.

4.5.0.0.5 Eigen-decomposition allows to diagonalize a matrix.

- Let $V = [v_1 \dots v_n]$ be a matrix with orthonormal eigenvectors as columns
- Let Λ be the diagonal matrix of corresponding eigenvalues
- A square simmetric matrix can be *diagonalized* as:

$$V^T A V = \Lambda \tag{4.7}$$

Remark: a diagonalized matrix is much simpler to manage and has the same properties as the original one (e.g. same eigen-decomposition).

Proof:
From eigenvalue-eigenvector definition (Definition 4.4):

$$A[v_1 \dots v_n] = [v_1 \dots v_n] \begin{bmatrix} \lambda_1 & & 0 \\ & \ddots & \\ 0 & & \lambda_n \end{bmatrix}$$

$$AV = V\Lambda$$

We multiply both sides by V^{-1}.

$$V^{-1}AV = V^{-1}V\Lambda$$

We have that $V^{-1}V = I$. What is more V is a *unitary matrix*, which means that its columns are orthonormal, for which: $V^{-1} = V^T$. As a consequence:

$$V^T A V = \Lambda$$

Positive semi-definite matrix
An $n \times n$ symmetric matrix M is *positive semi-definite* if all its eigenvalues are non-negative. Alternative sufficient and necessary conditions are:

- for all $x \in \mathcal{R}^n$

$$x^T M x \geq 0$$

- there exists a real matrix B such that:

$$M = B^T B$$

Positive definite matrix
An $n \times n$ symmetric matrix M is *positive definite* if all its eigenvalues are positive.

4.6 Understanding eigendecomposition

4.6.1 Scaling transformation in standard basis

Consider Figure 4.4.

- Let $x_1 = [1, 0]$, $x_2 = [0, 1]$ be the standard orthonormal basis in $\mathcal{R}^2$

- Let $x = [x_1, x_2]$ be an arbitrary vector in $\mathcal{R}^2$
- A linear transformation is a *scaling* transformation if it only stretches x along its directions

In Figure 4.4, matrix A encodes a scaling transformation.

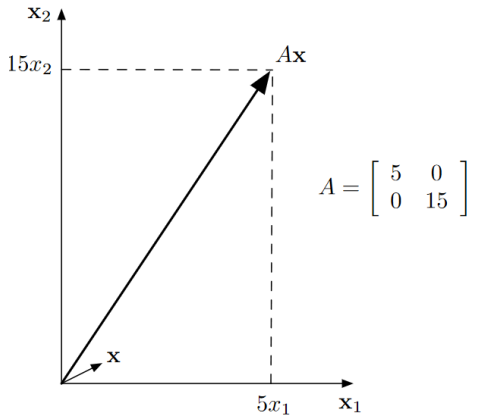


Figure 4.4: Scaling transformation in standard basis.

4.6.2 Scaling transformation in eigenbasis

Consider Figure 4.5.

- Let A be a non-scaling linear transformation in $\mathcal{R}^2$. Actually, transformation Av is not a linear transformation, since v has no components along the xaxis.
- Let $\{v_1, v_2\}$ be an *eigenbasis* for A. An eigenbasis is a basis in which every vector is an eigenvector.
- By representing vectors in $\mathcal{R}^2$ in terms of the $\{v_1, v_2\}$ basis (instead of the standard $\{x_1, x_2\}$), A becomes a scaling transformation.

Eigendecomposition is useful since we can compute an eigenbasis for a matrix A so that A becomes a scaling transformation with respect to the coordinate system defined by the eigenbasis.

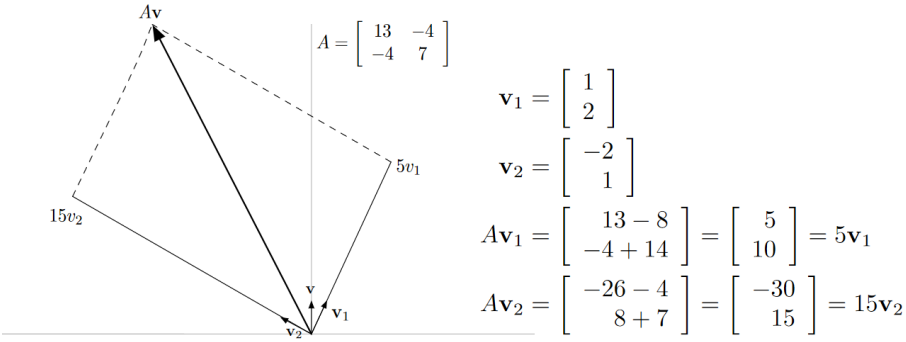


Figure 4.5: Scaling transformation in eigenbasis.

4.7 Principal Component Analysis (PCA)

Consider Figure 4.6. X,Y could be the height and the weight of a person. These two dimensions are correlated. As a result, if we collect some data from a population, we would obtain a distribution similar to the one represented on the left side of Figure 4.6. On the other hand, the graphic on the right of Figure 4.6, represents two quantities $P1, P2$ which are apparently uncorrelated. Indeed, the data spreads along the two dimensions.

- Let A be a data matrix with correlated coordinates.
- PCA is a linear transformation mapping data to a system of uncorrelated coordinates.
- It corresponds to fitting an ellipsoid to the data, whose axes are the coordinates of the new space.

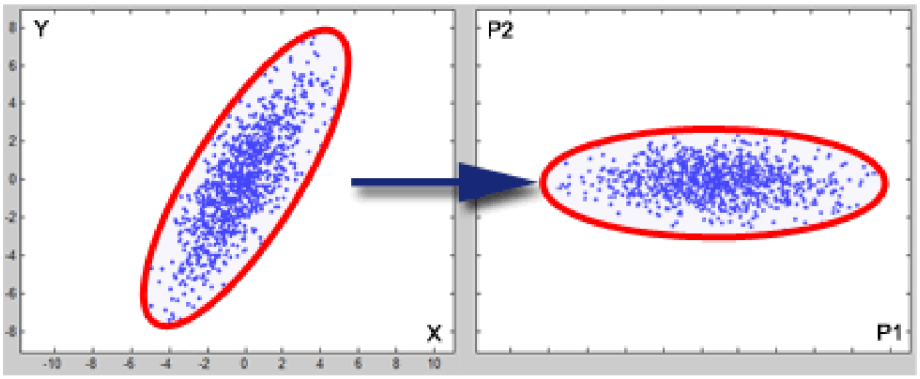


Figure 4.6: Principal Component Analysis (PCA).

Given a dataset $X \in \mathcal{R}^{n \times d}$ with n examples and d dimensions.

1. Compute the mean of the data (X_i is the i^{th} row vector of X):

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} X_i$$

2. Center the data into the origin:

$$X = X - \begin{bmatrix} \bar{x} \\ \vdots \\ \bar{x} \end{bmatrix}$$

3. Compute the data covariance:

$$C = \frac{1}{n} X^T X$$

   The covariance information highlights the correlations between the dimensions.

4. Compute the (orthonormal) eigendecomposition of C:

$$V^T C V = \Lambda$$

5. Use the set of eigen vectors as the new coordinate system:

$$x' = V^{-1} x = V^T x$$

   ($V^{-1} = V^T$ as V is unitary). In this new space (represented by the axis of the ellipse) the data are uncorrelated.

Remark: this procedure assumes linear correlations (and Gaussian distributions).

4.7.1 Dimensionality reduction

Assume that we have computed the covariance matrix C of a certain data distribution. What is more, we have also performed eigen value - eigen vector decomposition. At this point, it is possible to proof that each eigenvalue corresponds to the amount of variance in the direction of the corresponding eigenvector. The eigen vector which corresponds to the largest eigen values represents the directions of largest spread. The dimensions with the largest spread are the dimensions with the largest information. In order to perform *dimensionality reduction* (e.g. visualization), we select only the k eigenvectors with largest eigenvalues.
Suppose that we want to map the data in a d dimensional space in a lower dimensional space with k dimensions. To do this, we execute the same procedure

described above for PCA, but this time we take into consideration only the first k eigenvectors of the covariance matrix decomposition.

$$W = [v_1, \ldots, v_k]$$

Then, it is straightforward to map a point in the primary space to the new lower k-dimensional space (which retains the most information in terms of linear correlations):

$$x' = W^T x$$

Dimensionality reduction could be a valid pre-computation before applying other machine learning algorithms which perhaps performs poorly on high dimensional scenarios.

Chapter 5

Probability theory

5.1 Discrete random variables

A discrete variable can assume a value in a range of possible values, it can not get values in between.

Probability mass function:
Given a discrete random variable X taking values in $\mathcal{X} = \{v_1, ..., v_n\}$, its probability mass function $P : \mathcal{X} \to [0, 1]$ is defined as:

$$P(V_i) = Pr(X = v_i)$$

and satisfies the following conditions:

- $P(x) \geq 0$
- $\sum_{x \in \mathcal{X}} P(x) = 1$

Expected value:
The expected value (mean or average) of a random variable is:

$$E[X] = \mu = \sum_{x \in \mathcal{X}} xP(x) = \sum_{i=1}^{m} v_i P(v_i)$$

This is a linear operator, that implies:

$$E[\lambda x + \lambda' y] = \lambda E[y] + \lambda' E[y]$$

Variance:
The variance of a random variable is the moment of inertia of its probability mass function:

$$Var[x] = \sigma^2 = E[(x-\mu)^2] = \sum_{x \in \mathcal{X}} (x-\mu)^2 P(x)$$

The standard deviation σ indicates the typical amount of deviation from the mean one should expect for a randomly drawn value of x.

Variance is **not** a linear operator. Therefore:

$$Var[x] = E[x^2] = E[x]^2$$
$$Var[\lambda x] = \lambda^2 Var[X]$$

And for **uncorrelated variables** x, y it is true that

$$Var[x+y] = Var[x] + Var[y]$$

5.1.1 Probability distributions for discrete variables

Bernoulli distribution:
The Bernoulli distribution applies on variables that can assume binary outcomes (0/1, true/false, success/failure). Being p the parameter of success, its probability mass function is defines as:

$$P(x;p) = \begin{cases} p & if\ x = 1 \\ 1-p & if\ x = 0 \end{cases}$$

The expected value for a Bernoulli distribution is p and the variance is equal to $p(1-p)$.

Binomial distribution:
The binomial distribution is a Bernoulli distribution extended to a certain number n of trials. Its probability mass function is:

$$P(x;p,n) = \binom{n}{x} p^x (1-p)^{n-x}$$

The expected value for a Binomial distribution is np and the variance is equal to $np(1-p)$.

It is easy to see that a Bernoulli distribution is per se a Binomial distribution in the trivial case where $n = 1$.
A Bernoulli distribution can be used for modelling the toss of a coin, whereas the Binomial distribution describes the event of the coin has been tossed n times.

Multinomial distribution
The multinomial distribution models the probability of an event that can have m different outcomes, each with (possibly) a different probability. Its probability mass function is:

$$P(x_1, \ldots, x_m; p_1, \ldots p_m) = \prod_{i=1}^{m} p_i^{x_i}$$

where

- $x_1, \ldots, x_m$ is a vector that represents the outcome,
- $E[x_i] = p_i$
- $Var[x_i] = p_i(1 - p_i)$
- $Cov[x_i, x_j] = -p_i p_j$

Examples of outcome vectors for a 6-faces dice (d6) are:

$$x_1 = [1, 0, 0, 0, 0, 0],$$
$$x_2 = [0, 1, 0, 0, 0, 0],$$
$$x_3 = [0, 0, 1, 0, 0, 0],$$
$$x_4 = [0, 0, 0, 1, 0, 0],$$
$$x_5 = [0, 0, 0, 0, 1, 0],$$
$$x_6 = [0, 0, 0, 0, 0, 1]$$

in which the first column represents the success of the event of getting one as result, the second column the event of getting two and so on so forth. Since the event are mutually exclusive, just one column is signed as 1. For a fair dice, the vector or probabilities would be:

$$p = \left[\frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}\right]$$

Multinomial distribution (general case)
Given n repetition of an event which can end in m possible outcome, $p = \{p_1, \ldots, p_m\}$ the probability of each of the outcomes, the probability mass function is

$$P(x_1, \ldots, x_m; p_1, \ldots, p_m, n) = \frac{n!}{\prod_{i=1}^{m} x_i!} \prod_{i=1}^{m} p_i^{x_i}$$

In this distribution $E[x_i] = np_i$, $Var[x_i] = np_i(1 - p_i)$ and $Cov[x_i, x_j] = -np_i p_j$

Also this last case is a generalization of every case described above, but four our purposes will not be used, it is reported for completeness.

5.1.2 Pairs of discrete random variables

Joint probability mass function:
Given a pair of discrete random variables X, Y taking values $\mathcal{X} = \{v_1, ..., v_n\}$ and $\mathcal{Y} = \{y_1, ..., y_n\}$, the joint probability mass function is defined as:

$$P(v_i, w_j) = Pr[X = v_i, Y = w_j]$$

This function has the following properties:

- $P(x, y) \geq 0$
- $\sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} P(x, y) = 1$

Its expected values are:

$$\mu_x = E[x] = x \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} P(x, y)$$

$$\mu_y = E[y] = y \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} P(x, y)$$

The variances are:

$$\sigma_x^2 = Var[(x - \mu_x)^2] = \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} (x - \mu_x)^2 P(x, y)$$

$$\sigma_y^2 = Var[(y - \mu_y)^2] = \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} (y - \mu_y)^2 P(x, y)$$

The covariance (expectation of how much they can co-vary) is:

$$\sigma_{xy} = E[(x - \mu_x)(y - \mu_y)] = \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} (x - \mu_x)(y - \mu_y) P(x, y)$$

From these values we can get a coefficient of correlation as:

$$\rho = \frac{\sigma_{xy}}{\sigma x \sigma y}$$

This measure is the ratio between the covariance of two variables and the product of their standard deviations; thus, it is essentially a normalized measurement of the covariance, such that the result always has a value between -1 and 1.

5.2 Conditional probabilities

If two variables are not independent, we can apply some laws and rules to get their probability and conditional probabilities.

Law of total probability
The marginal distribution of a variable is obtained from a joint distribution summing over all possible values of the other variable.

$$P(x) = \sum_{y \in \mathcal{Y}} P(x, y)$$

and

$$P(y) = \sum_{x \in \mathcal{X}} P(x, y)$$

Produt rule, Bayes' rule
Given two variables x, y we can get the conditional probability from Bayes' rule as:

$$P(x|y)P(y) = p(y|x)P(x)$$

This is important because it allows to get information about a posterior probability given the initial conditions and the likelihood of the linkage.

$$posterior = \frac{likelihood \times prior}{evidence}$$

Prior and evidence are the data related to the single variables and likelihood is the effect produced by the cause.
These rules apply also to more than two variables:

$$P(y) = \sum_{x} \sum_{z} P(x, y, z)$$

$$P(y) = \sum_{x} \sum_{z} P(y|x, z)P(x, z)$$

also for conditional probability:

$$P(y) = \sum_{x} \sum_{y} \frac{P(x|y, x)P(y|z)P(x, z)}{P(x|z)}$$

Overall it is true that:

$$P(y|x, z) = \frac{P(x|z, y)P(y|z)}{P(x|z)}$$

5.3 Continuous random variables

Continuous variables allows us to assign non-discrete values, for example the height of a range of people, what we do is assigning intervals. The probability mass function can be generalized on continuous domains.
Given $W = (a < X \leq b)$ then $A = (X \leq a)$ and $B = (X \leq b)$. W and A end up to be mutually exclusive, so

$$P(B) = P(A) + P(W)$$

Cumulative distribution
Given a continuous variable X we define

$$F(q) = P(X \leq q)$$

the cumulative distribution of X. F is a monotonic function.
Given and interval $a < X \leq b$, the probability of X is described as the difference of the edge values:

$$P(a < X \leq b) = F(b) - F(a)$$

Probability density function
The derivative of the cumulative distribution is called density function.

$$p(x) = \frac{d}{dx}F(x)$$

This corresponds to the mass function for continuous values. Therefore is true that

$$F(q) = P(X \leq q) = \int_{-\infty}^{q} p(x)dx$$

The probability density function has these properties:

- $p(x) \geq 0$
- $\int_{-\infty}^{\infty} p(x)dx = 1$ which stands for the sum of all possible cases.

Also mean and variance 5.1 5.1 are computed in the same way, but taking into account the infinite number of small interval, therefore computing an integral as well.

$$E[x] = \mu = \int_{-\infty}^{\infty} xp(x)dx$$

$$Var[x] = \sigma^2 = \int_{-\infty}^{\infty} (x-\mu)^2 p(x)dx$$

5.3.1 Probability distributions for continuous variables

Gaussian or Normal distribution ($\mathcal{N}$)
The Gaussian distribution is a bell-shaped function that is the standard for fitting continuous data. The mean is the highest point in the graph while the variance is the spreading of the data from the mean.
The probability density function is

$$p(x;\mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma} \exp^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

A standard distribution is denoted as $\mathcal{N}(0,1)$ where $\mu = 0$ and $\sigma = 1$, whereas a standardization of a normal distribution is $\mathcal{N}(\mu,\sigma^2)$, it is carried on through this formula:

$$z = \frac{x-\mu}{\sigma}$$

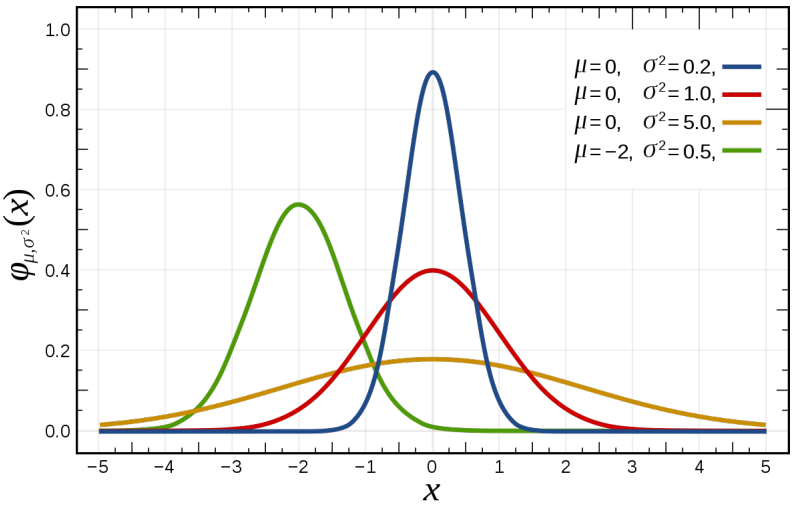


Figure 5.1: Set of Gaussian distributions

Beta distribution
The beta distribution is a second degree order of probability distribution (as the probability of a probability) that is defined in the interval $[0, 1]$ with parameters α and β. Its probability density function is

$$p(x; \alpha, \beta) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1-x)^{\beta-1}$$

where

- $\mathbb{E}[x] = \frac{\alpha}{\alpha+\beta}$
- $Var[x] = \frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$
- $\Gamma(x+1) = x\Gamma(x)$
- $\Gamma(1) = 1$

This distribution focuses on describing the distribution of a parameter p of a binomial distribution after observing $\alpha - 1$ independent events with probability p and $\beta - 1$ with probability $1 - p$.

Multivariate normal distribution
This distribution aims to define a Gaussian distribution extended to d dimensions. In order to define such distribution we need a mean μ and Σ covariance matrix. Its probability density function is described as

$$p(\vec{x}, \vec{\mu}, \Sigma) = \frac{1}{(2\pi)^{\frac{d}{2}}|\Sigma|^{\frac{1}{2}}} e^{-\frac{1}{2}(\vec{x}-\vec{\mu})^T\Sigma^{-1}(\vec{x}-\vec{\mu})}$$

Relevant features are:

- $E[x] = \mu$
- $Var[x] = \Sigma$
- the squared Mahalanobis distance from x to μ is $r^2 = (x-\mu)^T\Sigma^{-1}(x-\mu)$

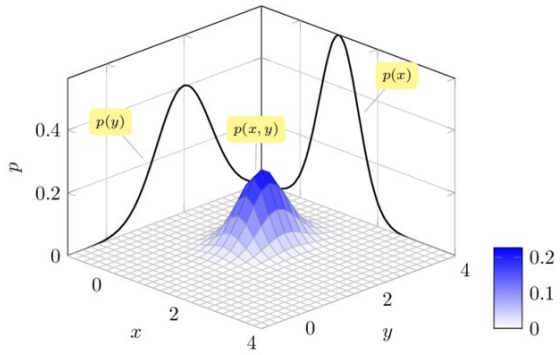


Figure 5.2: Multivariate normal distribution for $d = 3$

Dirichlet distribution
This distribution is defined for $x \in [0,1]^m$, $\sum_{i=1}^{m} x_i = 1$, the parameters it is based on are $\alpha = \alpha_1, \ldots, \alpha_m$ and its probability density function is

$$p(x_1, \ldots, x_m; \vec{\alpha}) = \frac{\Gamma(\alpha_0)}{\prod_{i=1}^{m} \Gamma(\alpha_i)} \prod_{i=1}^{m} x_i^{a_i - 1}$$

Where $\alpha_0 = \sum_{j=1}^{m} \alpha_j$.

Characterizing values of this distribution are:

- $E[x_i] = \frac{\alpha_1}{\alpha_0}$
- $Var[x_i] = \frac{\alpha_i(\alpha_0 - \alpha_i)}{\alpha_0^2(\alpha_0+1)}$
- $Cov[x_i, x_j] = \frac{-\alpha_i \alpha_j}{a_0^2(\alpha_0+1)}$

This distribution can model a posterior distribution of parameters p of a multinomial distribution after observing $\alpha_i - 1$ times each mutually exclusive events.

5.4 Probability laws

Let's consider a sample of $X_1, \ldots, X_n$ **independent** instances from a distribution with mean μ and variance σ^2. We can get a **mean of the obervated values** as

$$\bar{X}_n = \frac{X_1 + \cdots + X_n}{n}$$

and his expectation will be equals to μ

$$E[\bar{X}_n] = \mu$$

Its variance (iff they are **independent**) will be computed as

$$Var[a(X+Y)] = a^2(Var[X] + Var[Y])$$

and the mean of the computed variance will be

$$Var[\bar{X}_n] = \frac{1}{n^2}(Var[X_1] + \cdots Var[X_n]) = \frac{\sigma^2}{n}$$

that means that the variance of the average decreases as the number of observation increases (the higher the number of the observation, the higher the precision of the estimate).

Chebyshev's inequality
Given a random variable X with mean μ and variance σ^2, for all $a > 0$

$$Pr[|X - \mu| \geq a] \leq \frac{\sigma^2}{a^2}$$

this implies the more X differs from the mean more than a the more σ^2 gets bigger and vice versa.
Also, replacing $a = k\sigma$ for $k > 0$

$$Pr[|X - \mu| \geq k\sigma] \leq \frac{1}{k^2}$$

This inequality shows that most of the probability mass of a random variable stays within few standard deviations from its mean.

Law of large numbers
Considering a sample of $X_1, \ldots, X_n$ independent instances from a distribution with mean μ and variance σ^2, for any $\epsilon > 0$ it is true that the object $\bar{X}_n$ obeys:

$$\lim_{n\to\infty} Pr[|\bar{X}_n| > \epsilon] = 0$$

Using Chebyschev's inequality and knowing that $E[\bar{X}_n] = \mu$ and also $Var[\bar{X}_n] = \frac{\sigma^2}{n}$ it comes that

$$Pr[|\bar{X}_n - E[\bar{X}_n]| > \epsilon] \leq \frac{\sigma^2}{n\epsilon^2}$$

Overall this means that increasing the number of observation of a certain events allows us to get closer to the actual mean μ so the accuracy increases. Also this means that the variance gets closer (less uncertainty, relatively less values far away from the actual mean).

Central limit theorem
Considering a sample $X_1, \ldots, X_n$ of idipendent instances drawn from a distribution with mean μ and variance σ^2, then

1. regardless the distribution of X_i, for $n \to \infty$ the distribution of the sample average $\bar{X_n}$ approaches to a Normal distribution $\mathcal{N}$
2. Its mean approaches to μ and its variance approaches to $\frac{\sigma^2}{n}$
3. Thus the normalized sample average:

$$z = \frac{\bar{X_n} - \mu}{\frac{\sigma}{\sqrt{n}}}$$

approaches to a normal distribution $\mathcal{N}(0, 1)$

The Central Limit theorem implies that:

- the sum of a sufficiently large sample of independent random measurements is **approximately normally distributed**
- there is **no need to know the actual distribution**
- justifies the **importance of the Gaussian** distribution in the real world applications

5.5 Information theory

Consider a discrete set of symbols $\mathcal{V} = \{v_1, \ldots, v_n\}$ with mutually exclusive probabilities $P(V_i)$. We aim designing a binary codification for each symbol that minimize the overall length of messages encoded in binary. This optimal code assign to each symbol v_i a number of bits equals to $-\log P(v_i)$.

Entropy
The entropy of the set of symbols is the expected length of a message encoding a symbol assuming each such optimal coding

$$H[\mathcal{V}] = E[-\log P(v)] = -\sum_{i=1}^{n} P(v_i) \log P(v_i)$$

Cross entropy
Considering two distributions P and Q over a variable X, the cross entropy between the two distributions measures the expected number of bits needed to code a symbol sampled from P using Q instead:

$$H(P;Q) = \mathbb{E}_P[-\log Q(v)] = -\sum_{i=1}^{n} P(v_i) \log Q(v_i)$$

This is also used as a loss in binary classification (with P as empirical true distribution and Q ad empirical predicted distribution).

From these two definition we get the concept of **relative entropy**.

Relative entropy
Consider two distribution P and Q over a variable X, the relative entropy divergence (KL) measures the expected length difference when coding instances samples from P using Q instead:

$$D_{KL}(p||q) = H(P;Q) - H(P)$$

therefore

$$D_{KL}(p||q) = \sum_{i=1}^{n} P(v_i) \log \frac{P(v_i)}{Q(v_i)}$$

Conditional entropy
Consider two variables V, W with (possibly different) distribution P, then the conditional entropy of the entropy remaining for variable W once V is known:

$$H(W|V) = \sum_{v} P(v) H(W|V = v)$$

$$H(W|V) = \sum_{v} P(v) \sum_{w} P(w|v) \log P(w|v)$$

The conditional entropy allows to get insight of the information gain (reduction of entropy for W).

Mutual information
Given two variables V, W with (possibly different) distribution P. The mutual information (or *information gain*) is the reduction in entropy for W once V is known:

$$I(W;V) = H(W) - H(W|V)$$

$$I(W;V) = -\sum_{w} P(w) \log P(w) + \sum_{v} P(v) \sum_{w} P(w|v) \log P(w|v)$$

Chapter 6

Evaluation

Evaluation requires to define performance measures to be optimized. The evaluation process differs from solving a classic optimization problem. Indeed, performance of learning algorithms cannot be evaluated on entire domain. The purpose of machine learning is to learn a model from a restricted training set and perform generalization from this knowledge. Performance evaluation is needed for:

- tuning hyperparameters of learning methods (e.g. type of kernel and kernel parameters, learning rate of perceptron)
- evaluating quality of learned predictor
- computing statistical significance of difference between learning algorithms

The training loss function measures the cost paid for predicting $f(x)$ for output y ($f(x)$ is the predicted value; y is the ground truth). It is designed to boost effectiveness and efficiency of learning algorithms (e.g. *hinge loss* for SVM). Typically the purpose of the training is to minimize a training loss. As a consequence, ideal training loss functions are smooth, differentiable. As a result, not all the performance measures can be used as loss functions. Actually there is a difference between a loss function and the final performance measure. For example, *accuracy* (i.e. fraction of examples correctly classified) is a performance measure which can not be used as a loss function. More in depth, accuracy (or misclassification cost) is never used as a loss function, since it is *piecewise constant* and so not amenable to gradient descent. A function is said to be piecewise constant if it is locally constant in connected regions separated by a possibly infinite number of lower-dimensional boundaries. Square wave (Figure 6.1) is an example of piecewise constant function. This latter is not differentiable in correspondence of the jumps and has first derivative equals to zero when it is constant. The gradient cannot be computed or it is zero and so it does not indicate where to move in order to find the optimum.

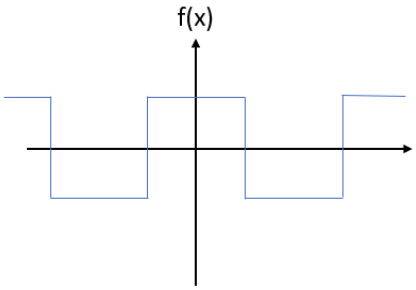


Figure 6.1: Square wave function is an example of piecewise constant function.

An example of training loss function is *cross entropy loss*.

Typically it is useful to consider multiple performance measures which can highlight complementary information about the model under study.

6.1 Binary classification

The typical way to visualize the performance of a classification algorithm is by means of a *confusion matrix*. This last is a table which has the possible labels as rows and columns. In particular, the rows represent the true labels for the data, while the columns report the predicted labels. In the case of binary classification, the confusion matrix can be exemplified as illustrated in Figure 6.1.

- TP → true positives: positives predicted as positives
- TN → true negatives: negatives predicted as negatives
- FP → false positives: negatives predicted as positives
- FN → false negatives: positives predicted as negatives

		prediction	
		P	N
ground truth	P	TP	FN
	N	FP	TN

Table 6.1: Confusion matrix for binary classification

From the confusion matrix it is possible to compute several performance measures.

Accuracy
Accuracy is the fraction of correctly labelled examples among all predictions.

$$Acc = \frac{TP + TN}{TP + TN + FP + FN} \tag{6.1}$$

Remark: accuracy is one minus the misclassification cost.

Accuracy is not informative for strongly unbalanced datasets (typically negatives much more than positives). For instance, if we monitor the presence of a rare pathology in a set of people, negatives example would be much more than positives. In this context, predictions are dominated by the larger class. As a consequence, predicting everything as negative often maximizes the accuracy without learning nothing. A possible solution consists of *rebalancing* costs. Following this technique, a positive example does not count as one but it counts as $\frac{N}{P}$ where $N = TN + FP$ and $P = TP + FN$. In this way, if we have 10 times more negatives, when the model predicts a positive correctly, it does not count as one point but it counts as 10 points.

Precision
Precision is the fraction of positives among examples predicted as positives.

$$Pre = \frac{TP}{TP + FP} \tag{6.2}$$

It measures the precision of the learner when predicting positive.

Remark: a problem of this measure is that I can improve precision by predicting positive very rarely. For this reason a the following complementary measure is introduced.

Recall or Sensitivity
Recall (or *sensitivity*) is the fraction of positive examples predicted as positives.

$$Rec = \frac{TP}{TP + FN} \tag{6.3}$$

It measures the coverage of the learner in returning positive examples.

Since these last two measures are complementary, i.e. when precision increases, the recall typically decreases, it is not usually appropriate to consider them separately. Nevertheless, in some contexts it could be reasonable to consider recall and precision separately. Consider an in information retrieval task like: "*Give me the pages about machine learning from DISI website*". In this case it is crucial to achieve a good precision in the first results of the searching algorithm output. Vice versa, if we aim to develop a cancer detection application, recall has to be prioritized.
For standard classification tasks, there is a measure which combines recall and precision balancing the two aspects.

F-measure
F-measure combines precision and recall balancing the two aspects. β is a parameter tradingoff precision and recall.

$$F_\beta = \frac{(1+\beta^2)(Pre \cdot Rec)}{\beta^2 Pre + Rec} \tag{6.4}$$

The most common version of F-measure is the F_1-measure which sets $\beta = 1$. F_1 is the *harmonic mean* of precision and recall. A good trade off between precision and recall is needed in order to have a high value of F_1. F_1 is a good performance measure (better then accuracy) in the case of unbalanced datasets.

$$F_1 = \frac{2(Pre \cdot Rec)}{Pre + Rec} \tag{6.5}$$

Classifiers often provide a confidence in the prediction (e.g. random forests, margin or SVM). In these cases it is possible to improve precision and/or recall by moving the confidence threshold for deciding if an element has to be classified positively or negatively. With a high threshold we would maximize precision. On the other hand, low values of threshold tend to maximize recall. By varying threshold from min to max possible value, we obtain a curve of performance measures. This curve can be shown plotting one measure (recall) against the complementary one (precision) (see Figure 6.2).

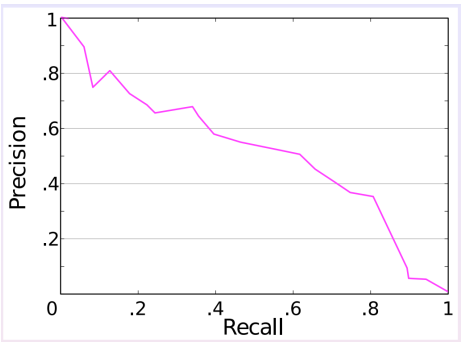


Figure 6.2: Precision-recall curve. When the threshold is 0, we obtain very high precision, since in this case the model predicts positive only if it has very high confidence. On the right the graphic highlights high recall since the model predicts always positive with such a configuration.

With this tool, we can compare different algorithms by plotting them on the same graphic. Then, we can decide the algorithm to use according to what we are interested in.
A single aggregate value can be obtained taking the area under the curve (see Figure 6.3). It combines the performance of the algorithm for all possible thresholds (without preference for a specific value of threshold). The area under the

precision-recall curve is a good general way to compare an algorithm with respect to another.

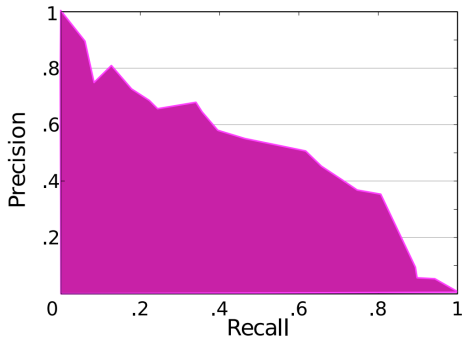


Figure 6.3: Area under precision-recall curve.

6.2 Multiclass classification

The notion introduced for binary classification in Section 6.1 can be extended to the domain of multiclass classification.
The confusion matrix is a generalized version of the binary one. The rows are the ground truth labels, while the columns represents the predicted labels. The structure of a confusion matrix in the case of multiclass classification is schematized in Figure 6.2.

		prediction		
		y_1	y_2	y_3
ground truth	y_1	n_{11}	n_{12}	n_{13}
	y_2	n_{21}	n_{22}	n_{23}
	y_3	n_{31}	n_{32}	n_{33}

Table 6.2: Multiclass classification confusion matrix.

- n_{ij} is the number of examples with class y_i predicted as y_j
- the main diagonal contains true positives for each class
- the sum of off-diagonal elements along a column is the number of false positives for the column label:

$$FP_i = \sum_{j \neq i} n_{ji}$$

- the sum of off-diagonal elements along a row is the number of false negatives for the row label:

$$FN_i = \sum_{j \neq i} n_{ij}$$

At this point we can compute the performance measures discussed above in Section 6.1. Accuracy, precision, recall, F_1-measure, are defined "*per-class*" (e.g. Pre_i indicates the precision of class i) considering as negatives examples from other classes.

- Precision of class i
$$Pre_i = \frac{n_{ii}}{n_{ii} + FP_i}$$
- Recall of class i
$$Rec_i = \frac{n_{ii}}{n_{ii} + FN_i}$$

An extension of the accuracy in a multiclass setting is the concept of *multiclass accuracy*.

Multiclass accuracy
Multiclass accuracy is the overall fraction of correctly classified examples.

$$MAcc = \frac{\sum_i n_{ii}}{\sum_i \sum_j n_{ij}} \tag{6.6}$$

In the same manner, we can compute F_1 for class i and the average F_1 across classes.

From the confusion matrix we can infer the pair of classes which are often confused by the classifier. This kind of observations could for example highlight the need of considering additional features in order to disambiguate classes.

6.3 Regression

Root mean squared error is the typical performance measure in the regression domain. Intuitively, it represents the distance between the predicted value and the real value.

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(f(x_i) - y_i)^2} \tag{6.7}$$

for dataset $\mathcal{D}$ with $n = |\mathcal{D}|$.

Another useful performance parameter for regression is the *Pearson correlation coefficient*:

$$\rho = \frac{cov(X,Y)}{\sigma_X \sigma_Y} = \frac{E[(X - \bar{X})(Y - \bar{Y})]}{\sqrt{E[(X - \bar{X})^2 E[(Y - \bar{Y})^2])]}}$$

The random variable X represents the predicted value, the random variable Y represents the ground truth label. However, the expectation values can

be computed only knowing the probability distribution. This last is typically unknown in machine learning tasks. As a consequence we approximate the expected values considering the average value over the entire dataset.

$$\rho = \frac{\sum_{i=1}^{n}(f(x_i) - \bar{f}(x_i))(y_i - \bar{y_i})}{\sqrt{\sum_{i=1}^{n}(f(x_i) - \bar{f(x_i)})^2 \sum_{i=1}^{n}(y_i - \bar{y_i})^2}} \tag{6.8}$$

6.4 Performance estimation

Computing performance measures on training set would be optimistically biased. We need to rely on different approaches. A typical method to deal with these circumstances is the *Hold-out procedure*.

6.4.1 Hold-out procedure

Given a labelled training set $\mathcal{D}$, we split it in three independent sets:

- *Training set* (40% $\mathcal{D}$): used to train the model
- *Validation set* (30% $\mathcal{D}$): used to estimate the performance of different algorithmic settings (i.e. hyperparameters). This set is used to make decisions.
- *Test set* (30% $\mathcal{D}$): used to estimate final performance of the selected model

The problematic aspect of this method is that the split of the original dataset into the training set, validation set and test set is constant. As a consequence, this procedure is used when a lot of data are available. On the contrary, if the dataset is small the prediction would be inaccurate. In the case of small datasets we need an approach whose results do not depend on the particular split.

6.4.2 k-fold cross validation

The *k-fold cross validation* approach splits the dataset $\mathcal{D}$ in k equal sized disjoint subsets $\mathcal{D}_i$. For each $i \in [1, k]$:

- train a predictor on $\mathcal{T}_i = \mathcal{D} \setminus \mathcal{D}_i$
- compute score S (e.g. accuracy) of predictor $L(\mathcal{T}_i)$ on test set $\mathcal{D}_i$:

$$S_i = S_{\mathcal{D}_i}[L(\mathcal{T}_i)]$$

After this iterative procedure, return the average score across folds.

$$\bar{S} = \frac{1}{k}\sum_{i=1}^{k} S_i$$

In this way, the performance doesn't depend on the specific choice of the split, because we perform multiple splits. In real application a typical value of k is $k \geq 10$.

The variance of the average score is computed assuming independent folds as follows:

$$Var[\bar{S}] = Var[\frac{S_1 + \ldots + S_k}{k}] \approx \frac{1}{k^2} \sum_{j=1}^{k} Var[S_j] \tag{6.9}$$

Notice that, the last part of the equation is not an equality. Indeed, the sets $S_1, \ldots, S_k$ share a lot of components. At this point the variance of the performance score on a specific fold can not be directly computed. Since we cannot exactly compute $Var[S_j]$, we approximate it with the *unbiased variance* across folds:

$$Var[S_j] = Var[S_h] \approx \frac{1}{k-1} \sum_{i=1}^{k} (S_i - \bar{S})^2 \tag{6.10}$$

It is not important to understand why we divide the result by $k - 1$. In essence, it is necessary because an unbiased estimate, is an estimate such that, if we consider an infinite number of cases converges to the true prediction. If we plug Equation 6.10 into Equation 6.9, we get:

$$Var[\bar{S}] \approx \frac{1}{k^2} \sum_{j=1}^{k} \frac{1}{k-1} \sum_{i=1}^{k} (S_i - \bar{S})^2 = \frac{1}{k^2} \frac{k}{k-1} \sum_{i=1}^{k} (S_i - \bar{S})^2 = \frac{1}{k} \frac{1}{k-1} \sum_{i=1}^{k} (S_i - \bar{S})^2 \tag{6.11}$$

6.5 Hypothesis testing

We want to compare generalization performance of two learning algorithms. We want to know whether observed difference in performance is *statistically significant* (and not due to some noisy evaluation). Hypothesis testing allows to test the statistical significance of a hypothesis (e.g. the two predictors have different performance).

Null hypothesis: H_0 default hypothesis, for rejecting which evidence should be provided

Test statistic: Given a sample of k realizations of random variables $X_1, \ldots, X_k$, a *test statistic* is a statistic $T = h(X_1, \ldots, X_k)$ whose value is used to decide whether to reject H_0 or not.

Example: given a set measurements $X_1, \ldots, X_k$, decide whether the actual value to be measured is zero.

- *Null hypothesis:* the actual value is zero

- *Test statistic:* sample mean:

$$T = h(X_1, \ldots, X_k) = \frac{1}{k} \sum_{i=1}^{k} X_i = \bar{X}$$

6.5.1 Glossary

- *Tail probability*: probability that T is at least as great (right tail) or at least as small (left tail) as the observed value t.
- *p-value*: the probability of obtaining a value T at least as extreme as the one observed t, under the assumption that the null hypothesis is correct. A very small p-value means that such an extreme observed outcome would be very unlikely under the null hypothesis. (Figure 6.4)
- *Type I error*: reject the null hypothesis when it's true. This is the most serious error. In order to minimize the type I it is common to put a threshold on the p-value.
- *Type II error*: accept the null hypothesis when it's false.
- *Critical region*: the set of random variables $X_1, \ldots, X_k$ taken into account in the test statistic, can be visualized as a vector in a n-dimensional space. In this latter, we define a region C such that if the vector lies in C then the null hypothesis is rejected. In orther words the critical region represents the set of values of T for which we reject the null hypothesis.
- *Critical values*: values on the boundary of the critical region.

  Example:
  A common null hypothesis is the one which states that a Gaussian distribution $\mathcal{N}(\theta, 1)$ with variance 1 has mean value equals to one ($\theta = 1$). The adopted critical region to face this test statistic is the following one:

$$C = \{(X_1, X_2, \ldots, X_n) : |1 - \frac{1}{n} \sum_{i=1}^{n} X_i| > \frac{1.96}{\sqrt{n}}\}$$

  So, the null hypothesis "$\theta = 1$" has to be rejected when the distance between the sample mean and 1 is larger then $\frac{1.96}{\sqrt{n}}$.

- *Significance level*: The type 1 error and the type 2 error are not symmetric. Actually, the purpose of the test statistics is not to state that the null hypothesis is true or false, but the purpose of the test is to understand if the sampled data are compatible with the null hypothesis. As a consequence, there is high tolerance for accepting H_0, while it is rejected only if the sampled data are really improbable assuming H_0 is satisfied. This kind of balance is regulated according to a parameter α. This latter is referred to as *significance level*. The test must fulfil the property for which,

the probability of rejecting the null hypothesis when it is true is lower then α. In other words, the significance level α is the largest acceptable probability for committing a type 1 error. Typical values of α are $0.1, 0.05, 0.005$.

Example:
Suppose we want to verify the following null hypothesis:

$$H_0 : \theta \in w$$

where w is a set of possible values for parameter θ. For this purpose, a point estimator $d(\boldsymbol{X})$ is considered. The null hypothesis H_0 is rejected when $d(\boldsymbol{X})$ is "far" from region w. In order to understand how much "far" w and $d(\boldsymbol{X})$ should be in order to reject H_0 with a significance level α, it is necessary to know the distribution of the estimator $d(\boldsymbol{X})$ when H_0 is true. This notion would allow us to use the property such that the probability of the type 1 error is lower than α, to understand when the estimator is "far" enough from w to reject the null hypothesis, and so defining the critical region of the test.

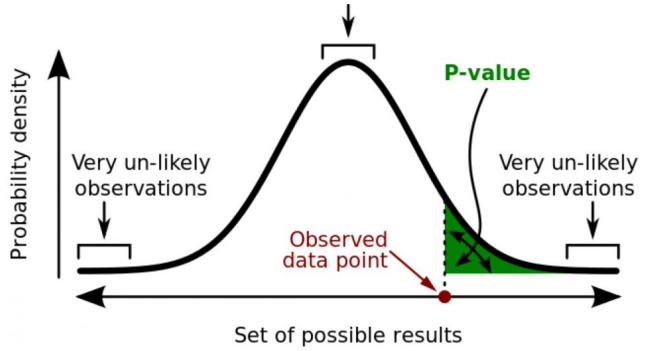


Figure 6.4: p-value

6.5.2 A useful digression to better understand hypothesis testing

Suppose that $X_1, X_2, \ldots, X_n$ is a random sample sampled from a normal distribution with parameters μ and σ^2. The variance of the normal distribution is known, while the mean value is unknown. Given a plausible mean value μ_0, we want to test the null hypothesis:

$$H_0 : \mu = \mu_0$$

For example, if we are testing the performance differences between two machine learning algorithms and the null hypothesis states that there is no difference between the performance of the two algorithms, then the mean value of

their difference in performance would be 0.

In this case, $\bar{X} := \frac{1}{n}\sum_{i=1}^{n} X_i$ is the natural punctual estimator for μ. We accept H_0 when $\bar{X}$ is not too far from μ_0. As a consequence, for a proper choice of the constant c, the critical region is:

$$C = \{(X_1, X_2, \ldots, X_n) : |\bar{X}| - \mu_0| > c\} \tag{6.12}$$

If our intention is to build a test with significance level α, we need to find the value of c in Equation 6.12 such that the probability of commit a first type error is α. This means that c has to verify the following relation:

$$\alpha = P(I \textit{ type error}) = P_{\mu_0}(|\bar{X} - \mu_0| > c) \tag{6.13}$$

We write P_{μ_0} to indicate the value of probability computed under the assumption that $\mu = \mu_0$. Indeed, by definition, first type error occurs when the sampled data lead us to reject H_0 (i.e. $(X_1, X_2, \ldots, X_n \in C)$) when indeed the null hypothesis is correct (i.e. $\mu = \mu_0$).
When $\mu = \mu_0$, we know that $\bar{X}$ is characterized by a normal distribution with mean value μ_0 and variance $\frac{\sigma^2}{n}$. Considering a $\mathcal{N}(0,1)$ random variable $\mathcal{Z}$, we obtain:

$$\frac{\bar{X} - \mu_0}{\frac{\sigma}{\sqrt{n}}} \sim_{\mu_0} \mathcal{Z} \tag{6.14}$$

Where the relation $\sim$ is conditioned to the hypothesis that: $H_0 : \mu = \mu_0$. The Equation 6.13 can now be written as:

$$\alpha = P_{\mu_0}(|\frac{\bar{X} - \mu_0}{\frac{\sigma}{\sqrt{n}}}| > \frac{c\sqrt{n}}{\sigma}) = P(|Z| > \frac{c\sqrt{n}}{\sigma}) = 2P(Z > \frac{c\sqrt{n}}{\sigma})$$

From this formula, we can immediately conclude that:

$$P(Z > \frac{c\sqrt{n}}{\sigma}) = \frac{\alpha}{2}$$

By definition of $z_{\frac{\alpha}{2}}$ we write:

$$P(Z > z_{\frac{\alpha}{2}}) = \frac{\alpha}{2}$$

From this we get:

$$\frac{c\sqrt{n}}{\sigma} = z_{\frac{\alpha}{2}}$$

$$c = z_{\frac{\alpha}{2}} \frac{\sigma}{\sqrt{n}}$$

We can conclude that:

- if $|\frac{\bar{X} - \mu_0}{\frac{\sigma}{\sqrt{n}}}| > z_{\frac{\alpha}{2}}$ we reject H_0

- if $|\frac{\bar{X}-\mu_0}{\frac{\sigma}{\sqrt{n}}}| \leq z_{\frac{\alpha}{2}}$ we accept H_0

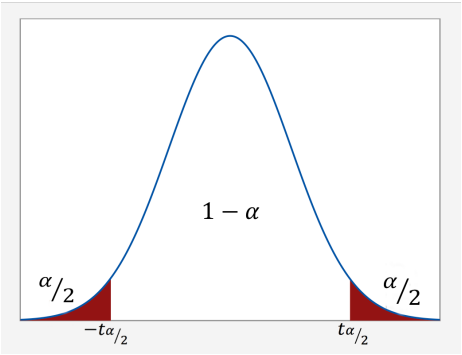


Figure 6.5: Confidence intervals

The acceptance region corresponds to the interval symmetric with respect to 0 labeled with $1-\alpha$ in Figure 6.5.

6.5.3 Hypothesis testing when the variance is unknown

Up to this point we have assumed that the only unknown parameter of the distribution is the mean value. However, really often both the mean μ and the variance σ^2 are unknown. Under these new assumptions, we study how to test the null hypothesis about the mean value.

$$H_0 : \mu = \mu_0$$

Analogously to what we have done before, it is reasonable to reject the null hypothesis when the sample mean $\bar{X}$ is far from μ_0. However, in this case the problem is more difficult since the distance between $\bar{X}$ and μ_0 which determines a rejection of the null hypothesis, depends on the standard deviation σ, which in this new setting is unknown. In particular, in the previous case, the null hypothesis is rejected when $|\bar{X} - \mu_0|$ is larger then $z_{\frac{\alpha}{2}} \cdot \frac{\sigma}{\sqrt{n}}$:

$$|\frac{\bar{X} - \mu_0}{\frac{\sigma}{\sqrt{n}}}| > z_{\frac{\alpha}{2}}$$

Since σ is unknown, we replace it with the sample standard deviation S.

$$S = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{X})^2} \tag{6.15}$$

Given that, we reject the null hypothesis when $|\frac{\bar{X}-\mu_0}{\frac{S}{\sqrt{n}}}|$ is "too large". In order to understand when this quantity is "too large" we need to know the distribution of the statistic test when H_0 is verified.

Probability distribution of a sample sampled from a Gaussian distribution with unknown variance

$X_1, X_2, \ldots, X_n$ is a sample sampled from a Gaussian distribution with mean value μ. If $\bar{X}$ and S^2 denotes the sample average and the sample variance respectively, then:

$$\frac{\bar{X} - \mu}{\frac{S}{\sqrt{n}}} \sim t_{n-1} \tag{6.16}$$

- When we normalize $\bar{X}$ by subtracting the mean value μ and dividing by the standard deviation $\frac{\sigma}{\sqrt{n}}$, we obtain a **standard normal distribution**.
- When we normalize $\bar{X}$ by subtracting the mean value μ and dividing by the sample standard deviation $\frac{S}{\sqrt{n}}$, we obtain a **Student's t-distribution** with $n-1$ degrees of freedom.

We denote the statistic of the test with T such that:

$$T = \frac{\bar{X} - \mu}{\frac{S}{\sqrt{n}}}$$

Under the assumption that H_0 is true ($\mu = \mu_0$), T is a t-distribution with $n-1$ degrees of freedom. We impose that the probability of a type one error is α:

$$P_{\mu_0}(-c \leq \frac{\bar{X} - \mu_0}{\frac{S}{\sqrt{n}}} \leq c) = 1 - \alpha$$

$$\alpha = 1 - P(-c < T < c) = P(T \leq -c) + P(T \geq c) = 2P(T \geq c)$$

$$P(T > c) = \frac{\alpha}{2}$$

$$c = t_{\frac{\alpha}{2}, n-1}$$

In conclusion:

- H_0 is rejected at a significance level α if:

$$|\frac{\bar{X} - \mu_0}{\frac{S}{\sqrt{n}}}| > t_{\frac{\alpha}{2}, n-1}$$

- H_0 is accepted if:

$$|\frac{\bar{X} - \mu_0}{\frac{S}{\sqrt{n}}}| \leq t_{\frac{\alpha}{2}, n-1}$$

In this case, the p-value is the probability that a Student's t-distribution with $n-1$ degrees of freedom has a value larger then $|t|$ under the assumption that H_0 is verified, where t is the test statistic calculated from the sample data.

6.5.4 Some properties of the t distribution

In the following we report some important properties of the t_{k-1} distribution.

- The Student's t-distribution is a bell-shaped distribution similar to the Normal one
- The shape is wider and shorter (fatter tails): reflects greater variance due to using its approximation $\tilde{Var}[\bar{X}]$ instead of the true unknown variance of the distribution
- $k-1$ is the number of degrees of freedom of the distribution (related to the number of independent events observed)
- t_{k-1} tends to the standardized normal Z for $k \to \infty$

6.6 Comparing learning algorithms

The purpose of this section is to understand how to compare two learning algorithms using hypothesis testing.
First of all we run k-fold cross validation procedure for algorithms A and B. Then we compute the mean performance difference for the two algoritms:

$$\hat{\delta} = \frac{1}{k}\sum_{i=1}^{k}\delta_i = \frac{1}{k}\sum_{i=1}^{k} S_{\mathcal{D}_i}[L_A(\mathcal{T}_i)] - S_{\mathcal{D}_i}[L_B(\mathcal{T}_i)] \tag{6.17}$$

Where $S_{\mathcal{D}_i}[L_A(\mathcal{T}_i)]$ is the performance of algorithm A on fold i. In particular the learning algorithm L_A is executed on a subset of the dataset ($\mathcal{T}_i$) while the performance (S) of the algorithm is measured considering the rest of the dataset ($\mathcal{D}_i$).

The null hypothesis is that the mean difference is zero (i.e. there is no difference between the two algorithms).

$$H_0 : \mu = 0$$

Clearly, the variance of the distribution is unknown. As a consequence we implement a t-test which rejects the null hypothesis at significance level α when:

$$\frac{\bar{\delta}}{\sqrt{\tilde{Var}[\bar{\delta}]}} \leq -t_{k-1,\frac{\alpha}{2}}$$

or

$$\frac{\bar{\delta}}{\sqrt{\tilde{Var}[\bar{\delta}]}} \geq t_{k-1,\frac{\alpha}{2}}$$

where $\bar{\delta}$ is the sample mean of the difference and:

$$\sqrt{\tilde{Var}[\bar{\delta}]} = \sqrt{\frac{1}{k(k-1)} \sum_{i=1}^{k} (\delta_i - \bar{\delta})^2}$$

Notice the the null hypothesis assumes zero means, that is why $\bar{\delta}$ is alone in the numerator.
We perform a two-tailed test if no prior knowledge can tell the direction of the difference. Otherwise it is reasonable to use one-tailed test.

6.6.1 t-test example: 10-fold cross validation

In Figure 6.6 there are reported the test performances computed on ten folds. The first column is the number of the fold $(\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_{10})$ of the 10-fold cross validation. The second and the third columns reports the performances (e.g. accuracy) achieved by algorithm A and B respectively. Finally, the fourth columns indicates the difference between the two scores.
Firstly, we can easily compute the mean value of the difference:

$$\bar{\delta} = \frac{1}{10} \sum_{i=1}^{10} \delta_i = 0.046$$

$\mathcal{D}_i$	$S_{\mathcal{D}_i}[L_A(\mathcal{T}_i)]$	$S_{\mathcal{D}_i}[L_B(\mathcal{T}_i)]$	δ_i
$\mathcal{D}_1$	0.81	0.80	0.01
$\mathcal{D}_2$	0.82	0.77	0.05
$\mathcal{D}_3$	0.84	0.70	0.14
$\mathcal{D}_4$	0.78	0.83	-0.05
$\mathcal{D}_5$	0.85	0.80	0.05
$\mathcal{D}_6$	0.86	0.78	0.08
$\mathcal{D}_7$	0.82	0.75	0.07
$\mathcal{D}_8$	0.83	0.80	0.03
$\mathcal{D}_9$	0.82	0.78	0.04
$\mathcal{D}_{10}$	0.81	0.77	0.04

Figure 6.6: 10-fold cross validation used to implement a t-test example on the performance difference between two algorithms A and B.

At this point we compute the unbiased estimate of the standard deviation:

$$\sqrt{\tilde{Var}[\bar{\delta}]} = \sqrt{\frac{1}{10 \cdot 9} \sum_{i=1}^{10} (\delta_i - \bar{\delta})^2} = 0.0154344$$

With this value we can calculate the standardized mean error difference:

$$\frac{\bar{\delta}}{\sqrt{\tilde{Var}[\bar{\delta}]}} = \frac{0.046}{0.0154344} = 2.98$$

We compute the t distribution for $\alpha = 0.05$ (i.e. we are happy to have a type one error probability of 5%) (it is a two-tailed test so we consider $\frac{\alpha}{2}$ in the following formula) and $k = 10$:

$$t_{k-1,\frac{\alpha}{2}} = t_{9,0.025} = 2.262$$

Finally, since $2.262 < 2.98$, the null hypothesis is rejected. We can conclude that the two classifiers are different. Indeed, my observation (i.e. my test statistic) is further to the right with respect to $t_{k-1,\frac{\alpha}{2}}$. In order to calculate the value of $t_{9,0.025}$ we have looked at the entry $[df = 9, t_.025]$ of the table in Figure 6.7.

df	$t_{.100}$	$t_{.050}$	$t_{.025}$	$t_{.010}$	$t_{.005}$
1	3.078	6.314	12.706	31.821	63.657
2	1.886	2.920	4.303	6.965	9.925
3	1.638	2.353	3.182	4.541	5.841
4	1.533	2.132	2.776	3.747	4.604
5	1.476	2.015	2.571	3.365	4.032
6	1.440	1.943	2.447	3.143	3.707
7	1.415	1.895	2.365	2.998	3.499
8	1.397	1.860	2.306	2.896	3.355
9	1.383	1.833	2.262	2.821	3.250
10	1.372	1.812	2.228	2.764	3.169
11	1.363	1.796	2.201	2.718	3.106
12	1.356	1.782	2.179	2.681	3.055
13	1.350	1.771	2.160	2.650	3.012
14	1.345	1.761	2.145	2.624	2.977
15	1.341	1.753	2.131	2.602	2.947
16	1.337	1.746	2.120	2.583	2.921
17	1.333	1.740	2.110	2.567	2.898
18	1.330	1.734	2.101	2.552	2.878
19	1.328	1.729	2.093	2.539	2.861
20	1.325	1.725	2.086	2.528	2.845
21	1.323	1.721	2.080	2.518	2.831
22	1.321	1.717	2.074	2.508	2.819
23	1.319	1.714	2.069	2.500	2.807
24	1.318	1.711	2.064	2.492	2.797
25	1.316	1.708	2.060	2.485	2.787
26	1.315	1.706	2.056	2.479	2.779
27	1.314	1.703	2.052	2.473	2.771
28	1.313	1.701	2.048	2.467	2.763
29	1.311	1.699	2.045	2.462	2.756
30	1.310	1.697	2.042	2.457	2.750
32	1.309	1.694	2.037	2.449	2.738
34	1.307	1.691	2.032	2.441	2.728
36	1.306	1.688	2.028	2.434	2.719
38	1.304	1.686	2.024	2.429	2.712
∞	1.282	1.645	1.960	2.326	2.576

Figure 6.7: Computing Student's t distribution.

Remarks:

- The test we have described in this section is a *paired test* since both the algorithms are evaluated over identical samples.
- The test we have described is a *two-tailed test*. Indeed we can not tell a priory if algorithm A is better then B or vice versa. Otherwise, if we had prior knowledge about the direction of the difference, we would use a one-tailed test (dividing α by two wouldn't be required).

Chapter 7

Parameter estimation

The settings of the environment we are now working in rely on:

- a collection of data sampled from a probability distribution $p(\boldsymbol{x}, y)$
- the probability distribution $p(\boldsymbol{x}, y)$ from which data are driven is known, but the parameters that describe this distribution are unknown (e.g. we know that p is a Gaussian distribution but we do not know the actual mean μ and variance σ (or mean vector and covariance matrix in the multivariate case)
- the data coming from the training set $\mathcal{D} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_m, y_m)\}$ are independent and identically distributed samples (i.i.d.) according to $p(\boldsymbol{x}, y)$
- the training set $\mathcal{D}$ can be divided into $\mathcal{D}_1, \ldots, \mathcal{D}_c$ subsets, with c equal to the number of classes. For each subset we have n examples i.i.d. $\mathcal{D}_i = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$
- for any *new* example x not in the training set, we compute the posterior probability of the class given the example and the full training set $\mathcal{D}$:

$$P(y_i|\boldsymbol{x}, \mathcal{D}) = \frac{p(\boldsymbol{x}|y_i, \mathcal{D})p(y_i|\mathcal{D})}{p(\boldsymbol{x}|\mathcal{D})} \tag{7.1}$$

  Computing this process for every class, we can take as predicted label the maximum probability among the classes.

We are trying to find the parameters that describe the actual distribution p from the training data $\mathcal{D}$, therefore $p(\boldsymbol{X}|\mathcal{D}_i, y_i)$.

From now on, whenever we are referring to a vector of features, we will use the bold notation $\boldsymbol{x}$.

By *independent* we mean that each example is sampled independently from the others.

By *identically distributed* we mean that all examples are sampled from the same distribution $p(\boldsymbol{x}, y)$.

From these settings we can further assume some simplifications:

- $\boldsymbol{x}$ is independent of $\mathcal{D}_j (j \neq i)$ given y_i and $\mathcal{D}_i$

$$P(y_i|\boldsymbol{x}, \mathcal{D}) = \frac{p(\boldsymbol{x}|y_i, \mathcal{D}_i)p(y_i|\mathcal{D})}{p(\boldsymbol{x}|\mathcal{D})}$$

- without further knowledge

$$p(y_i|\mathcal{D}) = \frac{\#y_i}{\#total} = \frac{|\mathcal{D}_i|}{|\mathcal{D}|}$$

- the normalizing factor

$$p(\boldsymbol{x}|\mathcal{D}) = \sum_{i=1}^{c} p(\boldsymbol{x}|y_i, \mathcal{D}_i)p(y_i|\mathcal{D})$$

At this point, the factor of the Equation 7.1 we are missing is $p(\boldsymbol{x}|y_i, \mathcal{D}_i)$. In order to do this, we need to estimate the parameters θ_i of the given distribution. There are two main option in order to get an estimation of the parameters θ_i:

1. **Maximum likelihood**: we assume that Θ_i have fixed but unknown values, then we assume that the parameters Θ_i are the ones that maximize the probability of the observed examples $\mathcal{D}_i$ (coming from the training set) of being part of $\mathcal{D}_i$.
   The values Θ_i are used to compute probability for new unseen examples:

$$P(\boldsymbol{x}|y_i, \mathcal{D}_i) \approx p(\boldsymbol{x}|\Theta_i)$$

2. **Bayesian estimation**: we assume that Θ_i are **not** fixed, but random variables with some known *prior distribution*, as we observe examples, the prior distribution become a *posterior* distribution. The prediction for the new examples are obtaining by an integral over the possible values of Θ_i

$$p(\boldsymbol{x}|y_i, \mathcal{D}_i) = \int_{\Theta_i} p(\boldsymbol{x}, \Theta_i|y_i, \mathcal{D}_i) d\Theta_i$$

7.1 Maximum-likelihood estimation

This method of parameter estimation is based on an analysis *a posteriori*, aka the data have already been seen.
The estimation of the parameters Θ^* if the distribution $p(\boldsymbol{x}, y)$ is based on finding the combination that fits at best the distribution of the data in each true class.

$$\Theta_i^* = argmax_{\Theta_i} p(\Theta_i|\mathcal{D}_i, y_i) = argmax_{\Theta_i} \frac{p(\mathcal{D}_i, y_i|\Theta_i)p(\Theta_i)}{p(\mathcal{D}_i, y_i)}$$

but $p(\mathcal{D}_i, y_i)$ does not depend on Θ, so:

$$\Theta_i^* = argmax_{\Theta_i} p(\Theta_i|\mathcal{D}_i, y_i) = argmax_{\Theta_i} p(\mathcal{D}_i, y_i|\Theta_i)p(\Theta_i)$$

$p(\mathcal{D}_i, y_i|\Theta_i)$ is called the likelihood. $p(\Theta_i)$ is called the prior.
This estimation assumes that a prior distribution for the parameters $p(\Theta_i)$ is available.
The most common formulation for the maximum likelihood estimation is:

$$\Theta_i^* = argmax_{\Theta_i} p(\mathcal{D}_i, y_i|\Theta_i) \tag{7.2}$$

which maximizes the likelihood of the parameters with respect to the training samples **without** prior knowledge distribution of the parameters.

Since every class is treated independently, we will drop the redundant notation $\mathcal{D}_i, y_i$ with $\mathcal{D}$ since it is not ambiguous.

With these assumption, we are trying to find θ^* as the combination of parameters for the distribution $p(x, y)$ that maximize the probability of a data coming for that same distribution:

$$\begin{aligned}\Theta^* &= argmax_{\Theta} p(\mathcal{D}|\Theta) \\ &= argmax_{\Theta} \prod_{j=1}^{n} p(\boldsymbol{x}_j|\Theta)\end{aligned}$$

Remark: $p(\mathcal{D}|\Theta) = \prod_{j=1}^{n} p(\boldsymbol{x}_j|\Theta)$ since the elements $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ in the training data are i.i.d..

7.1.1 Maximizing the log-likelihood

In order to maximize the probability previously described, is it clever to consider to maximize the logarithm of the probability. This is considered useful because the log function is monotonic and has several properties that simplify the procedure. For instance, computing the derivatives of a product is a pain, while it is easier to derive a summation.

$$\begin{aligned}\Theta^* &= argmax_\Theta \log p(\mathcal{D}|\Theta) \\ &= argmax_\Theta \sum_{j=1}^{n} \log p(\boldsymbol{x}_j|\Theta)\end{aligned}$$

In order to get the maximum, we can apply the derivative operation and look for the null point of this function. Since there are several parameters we are looking for, we will apply a gradient to the equation:

$$\nabla_\Theta \sum_{j=1}^{n} \log p(\boldsymbol{x}_j|\Theta) = 0 \tag{7.3}$$

These point will be local or global maxima depending on the distribution.

The distribution we take into account is a Gaussian 5.3.1 distribution of the parameters μ and σ (*univariate* case).

$$p(\boldsymbol{x}_j|\Theta) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(\boldsymbol{x}-\mu)^2}{2\sigma^2}\right)$$

Considering the logarithm:

$$\sum_{i=1}^{n} \log p(\boldsymbol{x}_j|\Theta) = \sum_{i=1}^{n} \log\left[\frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(\boldsymbol{x}-\mu)^2}{2\sigma^2}\right)\right] = \sum_{i=1}^{n} \log\frac{1}{\sqrt{2\pi}\sigma} - \frac{(\boldsymbol{x}-\mu)^2}{2\sigma^2}$$

Let's begin with the derivative of the log of the probability with respect to μ.

$$\begin{aligned}\frac{\partial p(\boldsymbol{x}_j|\Theta)}{\partial \mu} &= \frac{\partial}{\partial \mu} \sum_{i=1}^{n} \log\frac{1}{\sqrt{2\pi}\sigma} - \frac{(\boldsymbol{x}-\mu)^2}{2\sigma^2} \\ &= \sum_{i=1}^{n} \frac{(\boldsymbol{x}_i-\mu)}{\sigma^2}\end{aligned}$$

If we set this quantity to be equal to zero we get

$$\begin{aligned}\sum_{i=1}^{n} \frac{(\boldsymbol{x}_i-\mu)}{\sigma^2} &= 0 \\ \sum_{i=1}^{n} (\boldsymbol{x}_i-\mu) &= 0\end{aligned}$$

$$\sum_{i=1}^{n} \boldsymbol{x}_i = \sum_{i=1}^{n} \mu$$
$$\sum_{i=1}^{n} \boldsymbol{x}_i = n\mu$$
$$\mu = \frac{1}{n} \sum_{i=1}^{n} \boldsymbol{x}_i$$

Remark: this is the sample mean.

Repeating these operation with respect to σ we get

$$\begin{aligned}
\frac{\partial p(\boldsymbol{x}_j|\Theta)}{\partial \sigma} &= \frac{\partial}{\partial \sigma} \sum_{i=1}^{n} \log \frac{1}{\sqrt{2\pi}\sigma} - \frac{(\boldsymbol{x}_i - \mu)^2}{2\sigma^2} \\
&= \sum_{i=1}^{n} \sqrt{2\pi}\sigma \frac{1}{\sqrt{2\pi}}(-\sigma^{-2}) - \frac{(\boldsymbol{x}_i - \mu)^2}{2}(-2\sigma^{-3}) \\
&= \sum_{i=1}^{n} -\frac{1}{\sigma} + \frac{(\boldsymbol{x}_i - \mu)^2}{\sigma^3}
\end{aligned}$$

If we set this quantity to be equal to zero we get

$$\begin{aligned}
\sum_{i=1}^{n} -\frac{1}{\sigma} + \frac{(\boldsymbol{x}_i - \mu)^2}{\sigma^3} &= 0 \\
\sigma^3 \cdot \sum_{i=1}^{n} -\frac{1}{\sigma} + \frac{(\boldsymbol{x}_i - \mu)^2}{\sigma^3} &= 0 \cdot \sigma^3 \\
\sum_{i=1}^{n} -\sigma^2 + \sum_{i=1}^{n} (\boldsymbol{x}_i - \mu^2) &= 0 \\
\sum_{i=1}^{n} (\boldsymbol{x}_i - \mu^2) &= \sum_{i=1}^{n} \sigma^2 \\
\sum_{i=1}^{n} (\boldsymbol{x}_i - \mu^2) &= n\sigma^2 \\
\sigma^2 &= \frac{\sum_{i=1}^{n} (\boldsymbol{x}_i - \mu)^2}{n}
\end{aligned}$$

Remark: this is the sample variance.

By this we can get to the conclusion that if take into account Gaussian distribution, then the parameters that maximize the fitting of the training data into the dataset divided per class correspond to the sample mean and to the sample variance.

If this example is valid for a univariate (single feature) Gaussian distribution, we can extend the concept to a multivariate Gaussian distribution 5.3.1 and get as a result that:

- the log of the likelihood is

$$\sum_{j=1}^{n} -\frac{1}{2}(x_j - \mu)^t \Sigma^{-1}(x_j - \mu) - \frac{1}{2} log(2\pi)^d |\Sigma|$$

- the maximum likelihood estimates are the vector of the sample means

$$\mu = \frac{1}{n}\sum_{j=1}^{n} x_j$$

and the covariances matrix

$$\Sigma = \frac{1}{n}\sum_{j=1}^{n}(x_j - \mu)(x_j - \mu)^t$$

In the general case of a Gaussian distribution the maximum likelihood parameters are simply their empirical estimates over the samples.
The Gaussian mean is the sample mean while the covariance matrix is the mean of the sample covariances.

7.2 Bayesian estimation

With this approach we focus on the estimation of the parameters of our distribution $p(x, y)$ a *priori*, meaning that we update our values while getting new classified data.
In this case, the parameters Θ_i are related to a random variables (with some known prior distribution) that also need to be modeled. The prediction of the new samples is obtained through the integral over every possible parameter.

$$p(x|y_i, \mathcal{D}_i) = \int_{\Theta_i} p(x, \Theta_i|y_i, \mathcal{D}_i) d\Theta_i$$

The probability of $\boldsymbol{x}$ given each class y_i is independent of the other classes y_j with $i \neq j$. For simplicity we write:

$$p(\boldsymbol{x}|\mathcal{D}) = \int_{\Theta} p(\boldsymbol{x}, \Theta|\mathcal{D}) d\Theta$$

$$p(\boldsymbol{x}|\mathcal{D}) = \int_{\Theta} p(\boldsymbol{x}|\Theta) p(\Theta|\mathcal{D}) d\Theta$$

where $\mathcal{D}$ is the dataset for a certain class y and Θ the parameters of its distribution.

Let's proceed also in this case considering a Gaussian 5.3.1 distribution with unknown parameters Θ.
In computing $p(\boldsymbol{x}|\Theta, \mathcal{D})$, we do not need $\mathcal{D}$ since the probability of an example depends only on the parameters Θ. As a result, hereafter we write $p(\boldsymbol{x}|\Theta)$. Since the prediction is conditioned on the parameters Θ

$$p(\boldsymbol{x}|\mathcal{D}) = \int p(\boldsymbol{x}|\Theta)p(\Theta|\mathcal{D})d\Theta$$

The member of the equation $p(\boldsymbol{x}|\Theta)$ can be easily compute since we have the type of distribution and its parameters. Therefore we need to estimate the parameter posterior density given the training set:

$$p(\Theta|\mathcal{D}) = \frac{p(\mathcal{D}|\Theta)p(\Theta)}{p(\mathcal{D})}$$

In this formula $p(\mathcal{D})$ is a constant independent on Θ (therefore will not influence the Bayesian decision). If the final probability is needed we can compute it through

$$p(\mathcal{D}) = \int_{\Theta} p(\mathcal{D}|\Theta)p(\Theta)d\Theta$$

Remark: Calculating $p(\mathcal{D})$ is useful only if we want the final probability. Otherwise, if we only need to compute the most probable class then $p(\mathcal{D})$ can be treated as a constant and it is not necessary to compute it.

7.2.1 Univariate normal case: unknown μ and known σ

In the following we consider a univariate normal case such that the mean μ is unknown and the standard deviation is known. In other words $\Theta = \{\mu\}$. Let's say that we are trying to define μ from a given Gaussian distribution $p(x|\Theta) = p(x|\mu) \sim \mathcal{N}(\mu, \sigma^2)$. For the Bayesian settings in which we are working, also the mean μ belongs to a Gaussian distribution $p(\mu) \sim \mathcal{N}(\mu_0, \sigma_0^2)$: this distribution derives from **prior** knowledge, μ is at this time a random variable. For example, μ could be a random variable representing the mean height of the German student at the University of Trento while the prior μ_0, σ_0^2 could be the mean and the variance of the world German population.
The Gaussian mean posterior given the dataset is computed as

$$\begin{aligned} p(\mu|\mathcal{D}) &= \frac{p(\mathcal{D}|\mu)p(\mu)}{p(\mathcal{D})} \\ &= \alpha \prod_{j=1}^{n} p(x_j|\mu)p(\mu) \end{aligned}$$

where $\alpha = \frac{1}{p(\mathcal{D})}$ is independent of μ.

$p(\mathcal{D}|\mu) = \prod_{j=1}^{n} p(\boldsymbol{x}_j|\mu)$ since the examples are i.i.d..

From here we compute $p(\mu|\mathcal{D})$ taking into account two Gaussian distributions: the Gaussian distribution (of the data) $p(\boldsymbol{x}_j|\mu) \sim \mathcal{N}(\boldsymbol{x}_j; \mu, \sigma^2)$ and the one for of the mean, $p(\mu) \sim \mathcal{N}(\mu; \mu_0, \sigma_0^2)$.
After some mathematical computation (substituting the definition of normal distribution we get), we get that $p(\mu; \mathcal{D})$ is equal to

$$p(\mu|D) = \alpha \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma} exp - \frac{(\boldsymbol{x}_i - \mu)^2}{2\sigma^2} \frac{1}{\sqrt{2\pi}\sigma_0} exp - \frac{(\mu - \mu_0)^2}{2\sigma_0^2}$$

Which, after some manipulations, becomes:

$$\begin{aligned} p(\mu \mid \mathcal{D}) &= \alpha \prod_{j=1}^{n} \overbrace{\frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{1}{2}\left(\frac{x_j - \mu}{\sigma}\right)^2\right]}^{p(x_j|\mu)} \overbrace{\frac{1}{\sqrt{2\pi}\sigma_0} \exp\left[-\frac{1}{2}\left(\frac{\mu - \mu_0}{\sigma_0}\right)^2\right]}^{p(\mu)} \\ &= \alpha' \exp\left[-\frac{1}{2}\left(\sum_{j=1}^{n}\left(\frac{\mu - x_j}{\sigma}\right)^2 + \left(\frac{\mu - \mu_0}{\sigma_0}\right)^2\right)\right] \\ &= \alpha'' \exp\left[-\frac{1}{2}\left[\left(\frac{n}{\sigma^2} + \frac{1}{\sigma_0^2}\right)\mu^2 - 2\left(\frac{1}{\sigma^2}\sum_{j=1}^{n} x_j + \frac{\mu_0}{\sigma_0^2}\right)\mu\right]\right] \end{aligned}$$

From the last step, we can see that this composition behaves like a normal distribution itself

$$p(\mu|\mathcal{D}) = \frac{1}{\sqrt{2\pi}\sigma_n} e^{\frac{1}{2}(\frac{(\mu - \mu_n)}{\sigma_n})^2}$$

then we can put the last two equation as equal (the step by step explanation is omitted).

Solving for μ_n and σ_n^2 we get

$$\mu_n = \left(\frac{n\sigma_0^2}{n\sigma_0^2 + \sigma^2}\right)\hat{\mu_n} + \frac{\sigma^2}{n\sigma_0^2 + \sigma^2}\mu_0$$

and

$$\sigma_n^2 = \frac{\sigma_0^2\sigma^2}{n\sigma_0^2 + \sigma^2}$$

where $\hat{\mu_n}$ is the sample mean

$$\hat{\mu_n} = \frac{1}{n}\sum_{j=1}^{n} x_i$$

The interpretation of these results are the following:

- the mean is a **linear combination** of the *prior* μ_0 and the sample mean $\hat{\mu}_n$
- the higher the number of the training sample n, the more importance gets the sample mean $\hat{\mu}_n$, while μ_0 loses importance
- the more training samples n the more the variance decreases making the distribution less uncertain and more sharp.

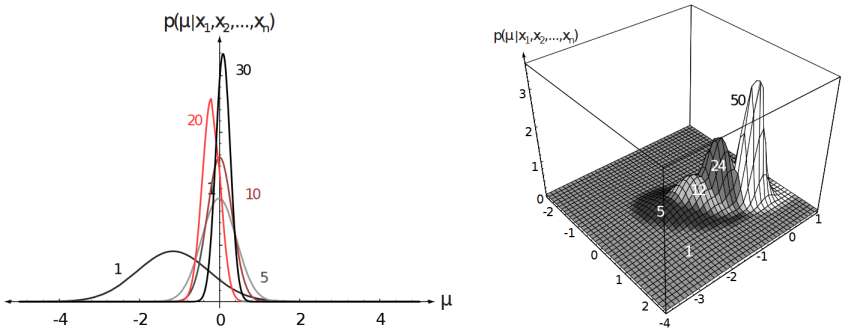


Figure 7.1: 2D and 3D graphics of increasing number of n samples seen for a Bayesian estimation of the mean μ

By this, we get that

$$p(x|\mathcal{D}) \sim \mathcal{N}(\mu_n, \sigma^2 + \sigma_n^2)$$

therefore the Gaussian distribution of the probability for the new samples in the model is built with parameters Θ that are the mean (as the posterior mean) and the sum of the known variance and the variance of the mean as a random variable (aka the uncertainty over the mean.

7.2.2 Generalization of univariate case

Since we know that $p(x|\mu) \sim \mathcal{N}(\mu, \Sigma)$
$p(\mu) \sim \mathcal{N}(\mu_n, \Sigma_n)$
therefore $p(\mu|\mathcal{D}) \sim \mathcal{N}(\mu_n, \Sigma_n)$
and $p(x|\mathcal{D}) \sim \mathcal{N}(\mu_n, \Sigma + \Sigma_n)$

7.3 Sufficient statistics

Statistic
Any function on a set of samples $\mathcal{D}$ is a statistic, a sufficient statistic

$$\mathbf{s} = \Phi(\mathcal{D})$$

if respects the following statement:

$$P(\mathcal{D}|\mathbf{s}, \Theta) = P(\mathcal{D}|\mathbf{s})$$

If Θ is a random variable, a sufficient statistic $\mathbf{s}$ contains all the relevant information about $\mathcal{D}$ in order to get the same and correct probability:

$$p(\Theta|\mathcal{D}, \mathbf{s}) = \frac{p(\mathcal{D}|\Theta, \mathbf{s})p(\Theta|\mathbf{s})}{p(\mathcal{D}|\mathbf{s})} = p(\Theta|\mathbf{s})$$

A sufficient statistic for a Gaussian distribution are sample mean and covariance.

7.4 Conjugate priors

Conjugate prior
Given the likelihood function $p(x|\Theta)$ and the prior distribution $p(\Theta)$, then

$$p(\Theta)$$

is a conjugate prior for $p(x|\Theta)$ if the posterior distribution $p(\Theta|x)$ is in the same family as the prior $p(\Theta)$

likelihood	**Parameters**	**Conjugate prior**
Binomial	p (*probability*)	Beta
Multinomial	p (*probability vector*)	Dirichlet
Normal	μ (*mean*)	Normal
Multivariate Normal	μ_i (*mean vector*)	Normal

Table 7.1: examples of distributions and their conjugate priors

In Table 7.1 we report some examples of distributions and their conjugate priors.

7.4.1 Example of a Bernoulli distribution

Given a Bernoulli distribution 5.1.1 with events $x = 1$ for *success* and $x = 0$ for *failure*, parameters Θ for success and $1 - \Theta$ for failure and probability mass

distribution

$$P(x|\Theta) = \Theta^x(1-\Theta)^{1-x}$$

then the conjugate prior is a Beta distribution 5.3.1 that depends on α_h, α_t

$$\begin{aligned} P(\Theta|\psi) &= P(\Theta|\alpha_h, \alpha_t) \\ &= \frac{\Gamma(\alpha)}{\Gamma(\alpha_h)\Gamma(\alpha_t)}\Theta^{\alpha_h-1}(1-\Theta)^{\alpha_t-1} \end{aligned}$$

7.4.1.1 Maximum likelihood estimation

For example, we can prove that given a dataset $\mathcal{D} = \{H, H, T, T, T, H, H\}$ of N iterations (like head/tail toss results), then the likelihood function becomes

$$p(\mathcal{D}|\Theta) = \Theta^h(1-\Theta)^t$$

The maximum likelihood parameter (maximizing the probability means compute the gradient and zeroing it), we are interested in the derivation with respect to Θ, then the estimation becomes

$$\begin{aligned} \frac{\partial}{\partial\Theta}\log p(\mathcal{D}|\Theta) \longrightarrow \frac{\partial}{\partial\Theta}h\log\Theta + t\log(1-\Theta) &= 0 \\ h\frac{1}{\Theta} - t\frac{1}{1-\Theta} &= 0 \\ h(1-\Theta) &= t\Theta \\ \Theta &= \frac{h}{h+t} \end{aligned}$$

By this we mean that t, h are a sufficient statistic 7.3. The best maximum likelihood parameters is the number of successes in the data among the total.

7.4.1.2 Bayesian estimation

If we try to estimate the parameters through a Bayesian estimation 7.2, then the conjugate prior will still be a Beta distribution 5.3.1.
The posterior is proportional to $P(\mathcal{D}|\Theta)P(\Theta|\psi)$ which comes from the Bayes' theorem 5.2 where $P(\mathcal{D})$ is omitted (therefore the $\propto$ property and not $=$), but it is also proportional to $P(\mathcal{D}|\Theta)P(\Theta,\psi)$ which is the prior Beta distribution

$$\begin{aligned} P(\Theta|\mathcal{D},\psi) &\propto P(\mathcal{D}|\Theta)P(\Theta|\psi) \\ &\propto \Theta^h(1-\Theta)^t\Theta^{\alpha_h-1}(1-\Theta)^{\alpha_t-1} \end{aligned}$$

This proportions gets together very easily collecting parameters until

$$P(\Theta|\mathcal{D},\psi) \propto \Theta^{h+\alpha_h-1}(1-\Theta)^{t+\alpha_t-1}$$

that is a Beta distribution $\sim Beta(\alpha'_h, \alpha'_t)$ where parameters $\alpha'_h = h + \alpha_h$ and $\alpha'_t = t + \alpha_t$. The posterior of my parameters is again a Beta with a new pair of

parameters α'_h and α'_t which are derived from the sum of the real counts (h, t) and the *imaginary* counts (α_t, α_h).

Then the prediction for a new event given the data is the expected value of the posterior Beta:

$$\begin{aligned} P(x|\mathcal{D}) &= \int_\Theta P(x, \Theta|\mathcal{D}) d\Theta \\ &= \int_\Theta P(\Theta|\mathcal{D}) P(\Theta|\mathcal{D}) d\Theta \end{aligned}$$

Since we need to integrate on a binary output (success, fail), then let's focus of the success event ($P(x = 1) = \Theta$) keeping in mind that Θ is a Beta distribution (therefore its expected value is $E[\Theta] = \frac{\alpha'_h}{\alpha'_h + \alpha'_t}$):

$$\begin{aligned} P(x = 1|\mathcal{D}) &= \int_\Theta \Theta P(\Theta|\mathcal{D}, \psi) d\Theta \\ &= E[\Theta] \\ &= \frac{h + \alpha_h}{h + t + \alpha_h + \alpha_t} \end{aligned}$$

by this we get that:

- with high t, h (seen results) then the parameters α_t, α_h become irrelevant, so this estimation becomes closer to the maximum likelihood estimation,
- with small t, h (seen results) that the prior estimate counts more.

7.4.2 Example with a multinomial distribution

Let's remind the reader the settings for a multinomial distribution: we have a set with r possible outcomes $x \in \{x^1, \ldots, x^r\}$ and for each outcome a (possibly different) probability (the probability over the dataset given the parameters of the distribution).
The one-hot encoding is defined as $x(x) = [z_1(x), \ldots, z_r(x)]$ with $z_k(x) = 1$ if $x = x^k$, 0 otherwise.
The probability mass function for this distribution is

$$P(x|\Theta) = \prod_{k=1}^{r} \Theta_k^{z_k(x)}$$

while its conjugate prior is a Dirichlet distribution 5.3.1:

$$\begin{aligned} P(\Theta|\psi) &= P(\Theta|\alpha_1, \ldots, \alpha_r) \\ &= \frac{\Gamma(\alpha)}{\prod_{k=1}^{r} \Gamma(\alpha_k)} \prod_{k=1}^{r} \Theta_k^{\alpha_k - 1} \end{aligned}$$

Starting from an example: we have a series of data representing the forecast of the past r days as $\mathcal{D} = \{R, C, C, S, S, S, R, C\}$ with R as rainy, C as cloudy and S as sunny. Each of the outcome has its own probability $\Theta_R, \Theta_C, \Theta_S$. Since we have the number of times that R, C and R were detected, overall we have probabilities $\Theta_C^{\#C}, \Theta_R^{\#R}, \Theta_S^{\#S}$ therefore $\Theta_C^3, \Theta_R^2, \Theta_S^3$.

7.4.2.1 Maximum likelihood estimation

We now try to estimate parameters for the multinomial distribution with the maximum likelihood method 7.1. We are trying to maximize the probability $P(\mathcal{D}|\Theta)$, therefore we look for the maximum of the logarithmic function that describes the probability.

$$\nabla_\Theta \log P(\mathcal{D}|\Theta) = 0$$

taking into account also that the sum of the probabilities need to be equal to 1

$$\sum_{i=1}^{n} \Theta_i = 1$$

From these assumptions we get:

$$\Theta_i = \frac{N_i}{\sum_{i=1}^{n} N_i}$$

which stands for the fraction of the outcomes in $\mathcal{D}$. Also, the numbers of the different outcomes in $\mathcal{D}$ are a sufficient statistic 7.3.

7.4.2.2 Bayesian estimation

Let's now try to estimate the parameters with the Bayesian method 7.2. In this case we have to consider that

$$\begin{aligned} P(\Theta|\mathcal{D}) &\propto P(\mathcal{D}|\Theta)P(\Theta) \\ &\propto \prod_{i=1}^{r} \Theta_i^{N_i} \prod_{i=1}^{r} \Theta_i^{\alpha_i - 1} \end{aligned}$$

where $P(\Theta|\mathcal{D})$ is the posterior, $P(\Theta)$ is the prior (Dirichlet distribution 5.3.1) and $P(\mathcal{D}|\Theta)$ is the likelihood.
Then we have that

$$\begin{aligned} \prod_{i=1}^{r} \Theta_i^{N_i} \prod_{i=1}^{r} \Theta_i^{\alpha_i - 1} &= \prod_{i=1}^{r} \Theta^{N_i + \alpha_i - 1} \\ &= \prod_{i=1}^{r} \Theta^{\alpha' - 1} \end{aligned}$$

Fixing $\alpha' = N_i + \alpha_i$.

Let's take for example the computation for $x = R$, then

$$\begin{aligned} P(x = R|\mathcal{D}) &= \int_{\Theta} P(x = R|\Theta)P(\Theta|\mathcal{D})d\Theta \\ &= \int_{\Theta} \Theta_R P(\Theta_R|\mathcal{D})d\Theta \\ &= E[\Theta_R] \end{aligned}$$

we end up with the expected value for R.

$$\begin{aligned} E[\Theta_R] &= \frac{\alpha'_R}{\sum_{i=1}^{r} \alpha'_i} \\ &= \frac{N_r + \alpha_R}{\sum_{i=1}^{r} N_i + \alpha_i} \end{aligned}$$

Also in this case, the prediction ends up being influenced by the actual seen values in N_R but also partially driven by the *imaginary* value for the parameter estimation α_R.

Chapter 8

Bayesian Networks

All probabilistic inference (i.e. finding the probability of a variable of interest given what we know in terms of the other variables) and learning amount at repeated applications of the basic rules of probability (sum and product rules). Hence, in principle we do not need an additional framework to deal with relationships between variables. However, *probabilistic graphical models* simplify a lot the management of multiple interconnected variables. Probabilistic graphical models are graphical representations of the *qualitative* aspects of probability distributions allowing to:

- visualize the structure of a probabilistic model (i.e. relationships between variables) in a simple and intuitive way
- discover properties of the model, such as conditional indipendencies, by visually inspecting the graph
- express complex computations for inference and learning in terms of graphical manipulations (e.g. information flows inside the graphical model)
- represent multiple probability distributions with the same graph, abstracting from their quantitative aspects (e.g. discrete vs continuous distributions). In a sense probabilistic graphical models are able to detach the qualitative aspects of probabilistic relationships from the quantitative aspects (e.g. the shape of the probability distribution which relates two different variables)

A *Bayesian Network* (BN) structure ($\mathcal{G}$) is a *directed graphical model*, so the connections between variables are characterized by a direction. Each node of the graph represents a random variable x_i. Each edge of the graph represents a direct dependency between two variables. Notice that if there is not an edge between two random variables x_i, x_j we can not conclude that there is no probabilistic relationship between x_i and x_j. Indeed, we have to take into account

indirect relationships. In Figure 8.1 it is reported an example of Bayesian Network.

Remark: $\mathcal{G}$ has no cycles, i.e. it is a *directed acyclic graph*.

The structure encodes these independence assumptions:

$$\mathcal{I}_l(\mathcal{G}) = \{\forall i \; x_i \perp NonDescendants_{x_i} | Parents_{x_i}\} \tag{8.1}$$

each variable is independent ($\perp$) of its non-descendants given its parents.

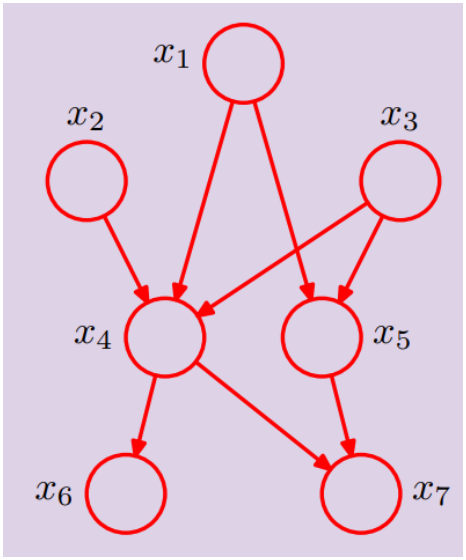


Figure 8.1: Bayesian Network example.

The *non-descendants* of a variable are the nodes which can not be reached following arrows from the variable. The subscript l in $\mathcal{I}_l(\mathcal{G})$ stands for *local* because, as discussed in the following, the model can also encode some global indipendences.

Now we explain how to represent probabilistic relationships among a set of variables by means of a Bayesian Network.

- Let $\mathcal{X}$ be a set of variables
- Let p be a joint probability distribution over variables $\mathcal{X}$. Here we assume that p is known, in practice we typically have to learn it from data
- Let $\mathcal{I}(p)$ be the set of independence assertions holding in p

- $\mathcal{G}$ is an *independency map* (I-map) for p if p satisfies the local independences of $\mathcal{G}$:

$$\mathcal{I}_l(\mathcal{G}) \subseteq \mathcal{I}(p)$$

Remark: Note that given $\mathcal{I}_l(\mathcal{G}) \subseteq \mathcal{I}(p)$, the reverse is not necessarily true: there can be independences in p that are not modelled by $\mathcal{G}$, perhaps we do not know them or the specific graphical model does not allow to encode them. Simmetrically, if we encode an independency in $\mathcal{G}$, this independecy should also hold in p.

Suppose we have to model a joint probability distribution $p(x_1, \ldots, x_m)$ among m binary variables, we have to consider 2^m possible configurations which can soon become intractable. Our aim is to break down this joint probability into pieces. We say that p *factorizes* according to $\mathcal{G}$ if:

$$p(x_1, \ldots, x_m) = \prod_{i=1}^{m} p(x_i | Pa_{x_i})$$

where Pa_{x_i} stands for the parents of x_i. Consider for example the Bayesian Network illustrated in Figure 8.1:

$$p(x_1, \ldots, x_7) = p(x_1)p(x_2)p(x_3)p(x_4|x_1, x_2, x_3)p(x_5|x_1, x_3)p(x_6|x_4)p(x_7|x_4, x_5)$$

Factorization is useful beacuse it breaks down a formula involving many variables into a product of formulas typically involving less values. In the example above, modeling $p(x_1, \ldots, x_7)$ as a single joint distribution, it would have required taking into consideration 2^7 possible parameters (assuming only binary variables for simplicity), one for each possible configuration. On the other hand, the pieces on the left side of the equation involve significantly less parameters:

- $p(x_1), p(x_2), p(x_3)$ have 2 possible configurations
- $p(x_4|x_1, x_2, x_3)$ has 2^4 possible configurations because it involves 4 variables
- $p(x_5|x_1, x_3), p(x_7|x_4, x_5)$ have 2^3 possible configurations
- $p(x_6|x_4)$ has 2^4 possible configurations

The order of complexity is around 2^4 against 2^7.

This important result about factorization holds:

- If $\mathcal{G}$ is an I-map for p, then p factorizes according to $\mathcal{G}$
- If p factorizes according to $\mathcal{G}$, then $\mathcal{G}$ is an I-map for p

The proof that this is the case follows:

I-map $\Rightarrow$ factorization

1. If $\mathcal{G}$ is an I-map for p, then p satisfies (at least) these (local) independences:

$$\{\forall i \; x_i \perp NonDescendants_{x_i} | Parents_{x_i}\}$$

2. Let us order variables in a *topological order* relative to $\mathcal{G}$, i.e.:

$$x_i \to x_j \Rightarrow i < j$$

The variables in the Bayesian Network in Figure 8.1 are already ordered topologically.

3. Let us decompose the joint probability using the chain rule as:

$$\begin{aligned} p(x_1, \ldots, x_m) &= p(x_m | x_1, \ldots, x_{m-1}) P(x_1, \ldots, x_{m-1}) \\ &= \prod_{i=1}^{m} p(x_i | x_1, \ldots, x_{i-1}) \end{aligned}$$

4. Since nodes are topologically ordered, in $p(x_m | x_1, \ldots, x_{m-1})$ x_m is conditioned on non-descendants. Local indipendences imply that for each x_i, in calculating $p(x_i | x_1, \ldots, x_{i-1})$, the only non-descendants $x_j \in \{x_1, \ldots, x_{i-1}\}$ which are relevant are the parents of x_i:

$$p(x_i | x_1, \ldots, x_{i-1}) = p(x_i | Pa_{x_i})$$

factorization $\Rightarrow$ I-map

1. If p factorizes according to $\mathcal{G}$, the joint probability can be written as:

$$p(x_1, \ldots, x_m) = \prod_{i=1}^{m} p(x_i | Pa_{x_i})$$

2. Let us consider the last variable x_m (repeat steps for the other variables). By the product (chain) and sum rules:

$$p(x_m | x_1, \ldots, x_{m-1}) = \frac{p(x_1, \ldots, x_m)}{p(x_1, \ldots, x_{m-1})} = \frac{p(x_1, \ldots, x_m)}{\sum_{x_m} p(x_1, \ldots, x_m))}$$

3. Applying factorization and isolating the only term containing x_m (the other terms which do not contain m are constant with respect to the summation and can be taken out from the summation) we get:

$$\begin{aligned} \frac{p(x_1, \ldots, x_m)}{\sum_{x_m} p(x_1, \ldots, x_m))} &= \frac{\prod_{i=1}^{m} p(x_i | Pa_{x_i})}{\sum_{x_m} \prod_{i=1}^{m} p(x_i | Pa_{x_i})} \\ &= \frac{p(x_m | Pa_{x_m}) \cancel{\prod_{i=1}^{m-1} p(x_i | Pa_{x_i})}}{\cancel{\prod_{i=1}^{m-1} p(x_i | Pa_{x_i})} \cancel{\sum_{x_m} p(x_m | Pa_{x_m})}} \end{aligned}$$

Remark: $\sum_{x_m} p(x_m|Pa_{x_m}) = 1$ since we are summing over all possible values of x_m.
Remark: In $p(x_m|x_1, \ldots, x_{m-1})$ all the variables in $\{x_1, \ldots, x_{m-1}\}$ are necessarily non-descendants of x_m since we have defined a topological order.
At the end of the day, we can conclude that the probability of x_m given the non-descendants is equal to the probability of x_m given the parents.

4. If this property holds for x_m, for the same reasoning the property must holds for all the other variables. In other words, following this procedure we recover the set of independences that define an I-map.

At this point we can conclude: factorization $\Leftrightarrow$ I-map.

Bayesian Network
A *Bayesian Network* is a pair $(\mathcal{G}, p)$ where p factorizes over $\mathcal{G}$ (which is a Bayesian network structure) and it is represented as a set of conditional probability distributions (cpd) associated with the nodes of $\mathcal{G}$ $(p(x_1|Pa_{x_1}), p(x_2|Pa_{x_2}), \ldots, p(x_m|Pa_{x_m}))$.

$$p(x_1, \ldots, x_m) = \prod_{i=1}^{m} p(x_i|Pa_{x_i})$$

8.1 Example of Bayesian Network: toy regulatory network

They are given three genes such that:

- genes A and B have independent prior probabilities
- gene C can be enhanced by both A and B (if A and B are active, i.e. they are producing proteins, then it is more probable that also C is active)

We can represent this probabilistic setting with a Bayesian network as illustrated in Figure 8.2. For the sake of simplicity, we assume that the variables are binary (active or inactive). In the context of binary variables, the conditional probability distribution becomes a conditional probability table, i.e. for each possible value of the variables there is a probability value. According to the structure of the network, the joint probability decomposes as:

$$P(A, B, C) = P(C|A, B)P(A)P(B)$$

Since A and B have no parents, we simply build the two tables as follows:

gene	value	P(value)
A	active	0.3
A	inactive	0.7

gene	value	P(value)
B	active	0.3
B	inactive	0.7

On the other hand, mapping $P(C|A, B)$ is slightly more difficult. The conditional probability table has indeed 3 variables. The table is illustrated in Figure 8.3.
Remark: of course the columns of the conditional probability table proposed in Figure 8.3 sum to one.

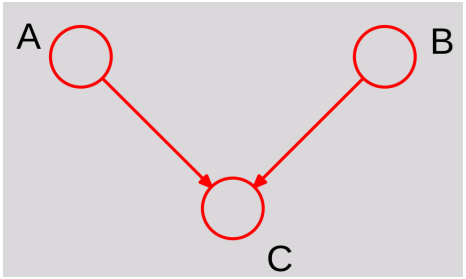


Figure 8.2: Toy regulatory network.

		A			
		active		inactive	
		B		B	
		active	inactive	active	inactive
C	active	0.9	0.6	0.7	0.1
C	inactive	0.1	0.4	0.3	0.9

Figure 8.3: Conditional probability table of gene C in the toy regulatory network proposed in Figure 8.2.

8.2 Conditional independence

A graphical model is a way to make independences assumptions explicit in the structure of the model.

Two variables a, b are independent (written $a \perp b|\emptyset$) if:

$$p(a, b) = p(a)p(b) \tag{8.2}$$

Two variables a, b are conditionally independent given c (written $a \perp b|c$) if:

$$p(a, b|c) = p(a|c)p(b|c) \tag{8.3}$$

Independence assumptions can be verified by repeated applications of sum and product rules in order to see whether relations like 8.3 are actually true. However, doing these kind of mathematical derivations can become tedious. Whereas, with a graphical model which encodes the probability independences which we assume the distribution has, the procedure becomes simplier. Graphical models allow to directly verify independence assumptions through the *d-separation* criterion.

8.2.1 d-separation

In order to introduce the process of d-separation, we start analyzing the base cases. The first relevant base case involves three variables. Indeed, the two variables case is trivial, since either the two variables are linked together and so they are related or there is no connection between the two variables and so they are not related. In a similar manner, also the case with three variables illustrated in Figure 8.4 is trivial since each variable is related (i.e. there is a link) to the others. As a consequence everybody is dependent with respect to everybody else. There is nothing non-trivial to find out. The non-trivial case is when not all the three variables are connected to all the others (e.g. one link is missing). An example is illustrated in Figure 8.5.

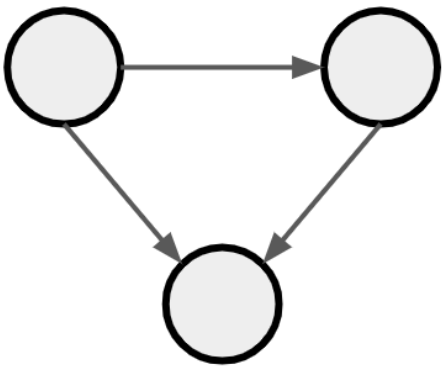


Figure 8.4: In this case although three variables are involved, verifying independency assumptions is trivial since each variable is related to the others.

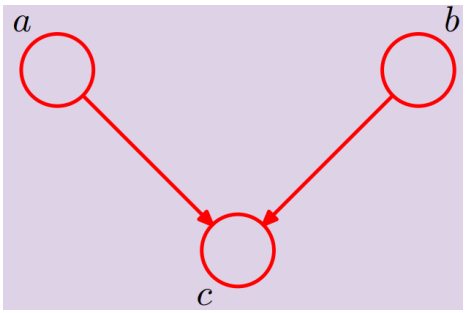


Figure 8.5: In this case the independences among the three variables are not trivial.

In the following we discuss about the possible non trivial configurations of three variables. For consistency, we assume that the variable in the middle is identified with c, while the variable on the left and the variable on the right are named a and b respectively.

8.2.1.1 Tail-to-tail

An example of this configuration is illustrated in Figure 8.6.
The pattern is called *tail-to-tail* because the central node c is connected to the other variables by means of the tails of the two arrows.
Looking at the structure in Figure 8.6, the joint probability distribution is:

$$p(a, b, c) = p(a|c)p(b|c)p(c)$$

At this point, our purpose is to understand whether a and b are independent ($a \perp b|\emptyset$) or not ($a \top\!\!\!\top b|\emptyset$). a and b are independent if:

$$p(a, b) = p(a)p(b)$$

Given $p(a, b, c)$ we get:

$$p(a, b) = \sum_c p(a, b, c)$$

According to the structure in Figure 8.6, we can write:

$$p(a, b) = \sum_c p(a|c)p(b|c)p(c)$$

Actually, this formula does not simplify into $p(a)p(b)$:

$$p(a, b) = \sum_c p(a|c)p(b|c)p(c) \neq p(a)p(b)$$

as a result a and b are **not independent**.
However, a and b are **conditionally independent given** c (Figure 8.6b):

$$p(a, b|c) = \frac{p(a, b, c)}{p(c)} = \frac{p(a|c)p(b|c)p(c)}{p(c)} = p(a|c)p(b|c)$$

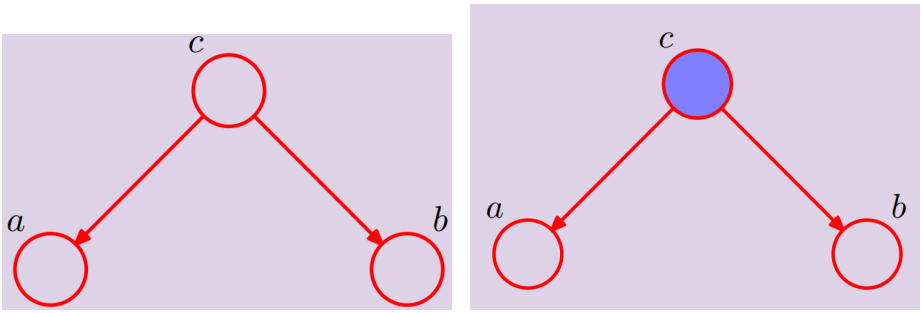
In other words, once c is observed, a and b become independent.

Example 1:

- a =Bike
- b =Bus delay
- c =Rain

Example 2:

- a =Cough
- b =Temperature
- c =Covid



(a) a and b are not independent.

(b) a and b are conditionally independent given c.

Figure 8.6: Tail-to-tail

8.2.1.2 Head-to-tail

In this case node c is in the middle of a chain, Figure 8.7. Similarly to the previous case, this configuration is called *head-to-tail* because variable c is connected to a and b by means of the head of an arrow and a tail of another arrow.

Remark: head-to-tail and tail-to-head are the same configuration because the labels a, b, c are arbitrary. The only thing which matters is the chain that characterizes the structure.

The joint probability $p(a, b, c)$ decomposes as follows:

$$p(a, b, c) = p(b|c)p(c|a)p(a)$$

We use this result to compute $p(a, b)$:

$$p(a, b) = \sum_c p(a, b, c) = p(a) \sum_c p(b|c)p(c|a)$$

There is no way applying probability rules to satisfy the equality $p(a)\sum_c p(b|c)p(c|a) = p(a)p(b)$. Given this, we can conclude that a and b are **not independent**:

$$p(a,b) = p(a)\sum_c p(b|c)p(c|a) \neq p(a)p(b)$$

On the other hand a and b are **conditionally independent given** c:

$$p(a,b|c) = \frac{p(a,b,c)}{p(c)} = \frac{p(b|c)p(c|a)p(a)}{p(c)}$$

Applying the Bayes rule we notice that:

$$\frac{p(c|a)p(a)}{p(c)} = p(a|c)$$

So:

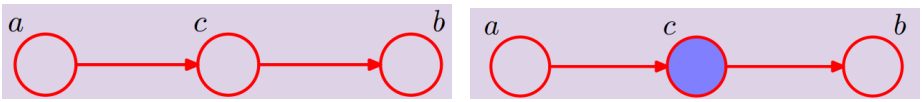
$$p(a,b|c) = \frac{p(b|c)p(c|a)p(a)}{p(c)} = p(b|c)p(a|c)$$

Intuitively, if you have already observed the effects of a certain cause, you don't really care about the cause anymore in order to update the probability of the consequence.

Example:

- a =Cloudy
- c =Rainy
- b =Get wet

Assume that a student of the DISI department of the University of Trento named Giovanni Valer wakes up and notices that it's a cloudy day. He realizes that, if he goes to the university by bike, he could get wet. Indeed, since it is cloudy, it could rain. The fact that it is cloudy influences the chance of getting wet (causality reasoning).
Suppose not that Giovanni wakes up and it is raining but at the same time there is the sun. Even if it is not cloudy, Giovanni gets wet if he travels by bike to the university. Evidently, given that it is raining, the fact that it is not cloudy does not influence the probability of getting wet.



(a) a and b are not independent.

(b) a and b are conditionally independent given c.

Figure 8.7: Head-to-tail

8.2.1.3 Head-to-Head

For the same reasons of the previous cases, the structure exemplified in Figure 8.8 is called *head-to-head*. In this case the joint distribution decomposes as:

$$p(a, b, c) = p(c|a, b)p(a)p(b)$$

Calculating $p(a, b)$ summing over c we notice that $p(a)$ and $p(b)$ can be extracted from the summation since they do not depend on c. Moreover, $\sum_c p(c|a, b) = 1$:

$$p(a, b) = \sum_c p(c|a, b)p(a)p(b) = p(a)p(b)$$

Therefore, we can conclude that a and b are **independent**.

In this case we are not able to simplify $p(a, b|c)$ in order to verify the equality $p(a, b|c) = p(a|c)p(b|c)$. Actually, a and b are **not conditionally independent given** c:

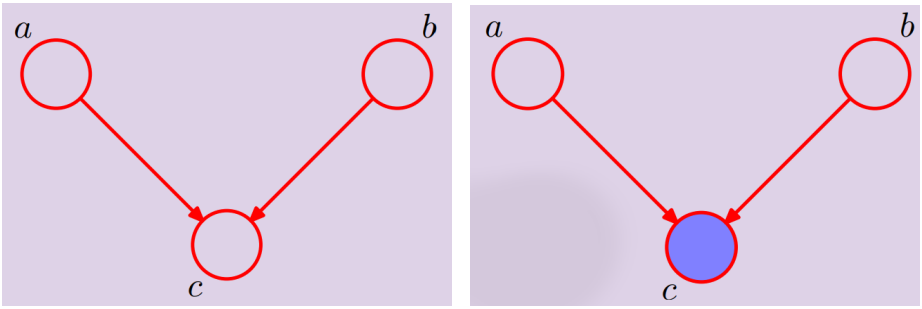
$$p(a, b|c) = \frac{p(c|a, b)p(a)p(b)}{p(c)} \neq p(a|c)p(b|c)$$

Intuitively in this case a and b are competing causes of the effect c. At the beginning a and b are independent causes, but once the effect c is observed they compete for the explanation. As a result, observing the result of one of them, reduces the probability of the other. This situation is usually referred to as *explaining away effect*.

Example:

- a =Burglar
- b =Earthquake
- c =Alarm

Burglar and earthquake can be both possible causes for the alarm to ring. Of course, the burglar can decide to steal something from the house of Giovanni Valer regardless of the probability of an earthquake.
However, if the alarm rings Giovanni immediately thinks about a burglar. But if there is an earthquake in that moment, then the probability of a burglar in the mind of Giovanni decreases to 0.



(a) a and b are independent.

(b) a and b are not conditionally independent given c.

Figure 8.8: Head-to-head

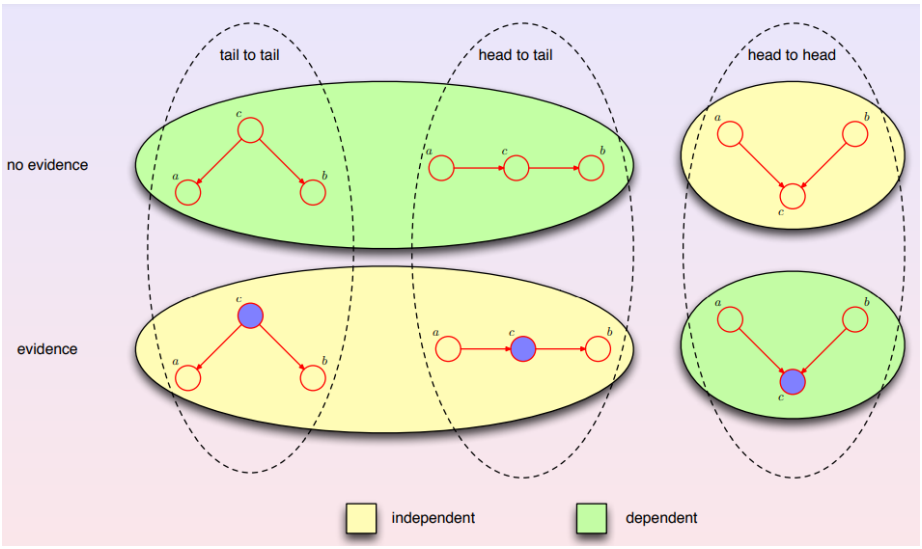


Figure 8.9: d-separation: basic rules summary.

8.2.1.4 Example of head-to-head connection

In this example we take into consideration a fuel system in a car. Our problem setting studies three random binary variables:

- *Battery*[B]: either charged (B=1) or flat (B=0)
- *Fuel tank*[F]: either full (F=1) or empty (F=0)
- *Electric fuel gauge*[G]: it indicates either full tank (G=1) or empty tank (G=0)

The probabilistic dependences among these variables are represented by the Bayesian Network in Figure 8.10a.

Conditional probability tables (CPT)

- Battery and tank have independent prior probabilities:

$$P(B = 1) = 0.9$$

$$P(F = 1) = 0.9$$

- The fuel gauge is conditioned on both (unreliable):

$$P(G = 1|B = 1, F = 1) = 0.8$$

$$P(G = 1|B = 0, F = 1) = 0.2$$

$$P(G = 1|B = 1, F = 0) = 0.2$$

$$P(G = 1|B = 0, F = 0) = 0.1$$

Probability reasoning

In this case the joint probability decomposes as follows:

$$P(F, G, B) = P(G|B, F)P(B)P(F)$$

Suppose that we are interesting in reasoning about the probability of empty tank.

- The prior probability, without observing anything is:

$$P(F = 0) = 1 - P(F = 1) = 0.1$$

- The posterior after observing empty fuel gauge (Figure 8.10b) is different:

$$\begin{aligned} P(F = 0|G = 0) = & \\ & = \frac{P(G = 0|F = 0)P(F = 0)}{P(G = 0)} = \\ & = \frac{\sum_B [P(G = 0, B|F = 0)]P(F = 0)}{P(G = 0)} = \\ & = \frac{\sum_B [P(G = 0|F = 0, B)P(B|F = 0)]P(F = 0)}{P(G = 0)} \end{aligned}$$

F is not a descendant of B, as a result $P(B|F = 0) = P(B)$. B is independent with respect to its non descendants given the parents and B has no parents.

$$P(F=0|G=0) = \\ = \frac{\sum_B [P(G=0|F=0,B)P(B|F=0)]P(F=0)}{P(G=0)} = \\ = \frac{\sum_B [P(G=0|F=0,B)P(B)]P(F=0)}{P(G=0)}$$

Now, we compute the denominator:

$$P(G=0) = \\ = \sum_{B\in\{0,1\}} \sum_{F\in\{0,1\}} P(G=0,B,F) = \\ = \sum_{B\in\{0,1\}} \sum_{F\in\{0,1\}} P(G=0|B,F)P(B,F) = \\ = \sum_{B\in\{0,1\}} \sum_{F\in\{0,1\}} P(G=0|B,F)P(F|B)P(B)$$

Given the network, F is independent with respect to its non descendants given its parents and F has no parents.

$$P(G=0) = \\ = \sum_{B\in\{0,1\}} \sum_{F\in\{0,1\}} P(G=0|B,F)P(F|B)P(B) \\ = \sum_{B\in\{0,1\}} \sum_{F\in\{0,1\}} P(G=0|B,F)P(B)P(F)$$

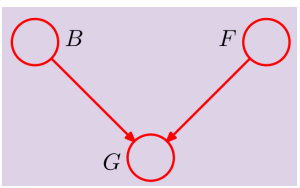
At the end of the day:

$$P(F=0|G=0) = \\ = \frac{\sum_B [P(G=0|F=0,B)P(B)]P(F=0)}{P(G=0)} = \\ = \frac{\sum_B [P(G=0|F=0,B)P(B)]P(F=0)}{\sum_{B\in\{0,1\}} \sum_{F\in\{0,1\}} P(G=0|B,F)P(B)P(F)} \simeq 0.257$$

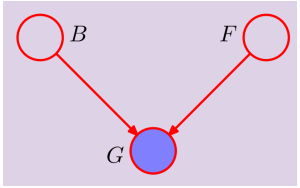
The probability that the tank is empty increases from observing that the fuel gauge reads empty (not as much as expected because a strong prior and unreliable gauge).

Remark: In real world applications, we typically observe the effects rather than the causes (e.g. we observe symptoms, not the pathology). We have to apply probability rules and in particular the Bayes theorem in order to reason about the probability of the causes. This is why this structures are named

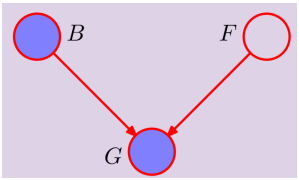
Bayesian Networks.



(a) Example of head-to-head connection.



(b) The probability that the tank is empty increases from observing that the fuel gauge reads empty.



(c) The probability that the tank is empty decreases after observing that the battery is also flat.

Figure 8.10: Head-to-head

Suppose that now, we are interesting in the posterior probability after observing that the battery is also flat (Figure 8.10c).

$$P(F=0|G=0,B=0) = \\ = \frac{P(G=0|F=0,B=0)P(F=0|B=0)}{P(G=0|B=0)} \simeq 0.111$$

From this we understand that F and B are related once G is known. If they were unrelated, knowing something about B would not change the probability value. On the contrary, the probability that the tank is empty decreases after observing that the battery is also flat. The battery condition *explains away* the observation that the fuel gauge reads empty. The probability is sill greater than the prior one, because the fuel gauge observation still gives some evidence in favour of an empty tank.

8.2.1.5 d-separation in networks with more than three variables

Naturally, it is interesting to study more complex configurations involving more than three random variables.

General Head-to-head

- Let a descendant of a node x be any node which can be reached from x with a path following the direction of the arrows.
- A head-to-head node c unblocks the dependency path between its parents if either itself or any of its descendants receives evidence.

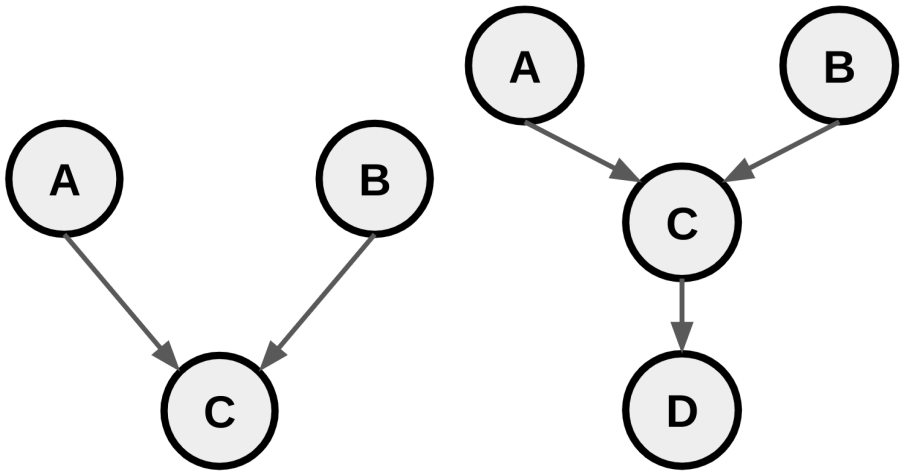
Consider the example in Figure 8.11a. As we explained above, A and B are independent in principle. However, once C receives evidence, then A and B are dependent one with respect to the other. Analogously, in Figure 8.11b

variables A and B are initially independent, but when D, which is a child of C, is observed, then A and B becomes mutually dependent. For instance, the four variables could represent the following events:

- A = burglar
- B = earthquake
- C = alarm
- D = phone call from the alarm company

If Giovanni Valer is not at home but he receives a phone call from the alarm company he immediately thinks about a burglar even if he does not hear the alarm. Evidently, burglar and earthquake both compete for an indirect effect.

This rules is valid for every descendant, though probably the further the descendant the less influent is the indirect connection.



(a) Head-to-head connection among three random variables.

(b) Head-to-head connection among more than three random variables.

Figure 8.11: General Head-to-head

8.2.1.6 General d-separation criterion

Given a generic Bayesian network. Given A, B, C arbitrary nonintersecting sets of nodes. The sets A and B are *d-separated* by C ($dsep(A; B|C)$) if all paths from any node in A to any node in B are *blocked* (there is no information flow along the path). Actually, if there is no information flow between A and B, then A and B are independent given C.
A path is blocked if it includes at least one node s.t. either:

- the arrows on the path meet tail-to tail (Figure 8.6b) or head-to-tail (Figure 8.7b) at the node and it is in C, or
- the arrows on the path meet head-to-head at the node and neither it nor any of its descendants is in C (Figure 8.8b).

d-separation implies conditional independence
The sets A and B are independent given C ($A \perp B|C$) if they are d-separated by C.

In a sense, we are using the basic rules in Figure 8.9 as building blocks to define the more general d-separation principle. The basic rules allow to understand if individual paths are blocked somewhere. It is sufficient that a path is blocked in one place to be completely blocked.

Example of general d-separation 1
In this second example we consider the Bayesian Network in Figure 8.12. In this case nodes a and b are not d-separated by c:

- Node f is tail-to-tail and not observed.
- Node e is head-to-head and its child c is observed.

We can conclude that $a \top\!\!\!\top b|c$. In other words, there is information flow from a to b.

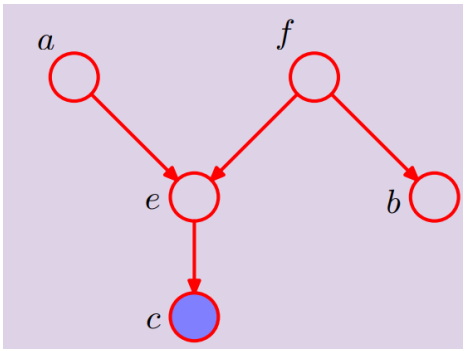


Figure 8.12: Example of general d-separation. $a \top\!\!\!\top b|c$

Example of general d-separation 2
In this second example we consider the Bayesian Network in Figure 8.13. In this case a and b are d-separated by e:

- e is head-to-head connected.
- e is not observed.

- no children of e are observed.

As a result there is no information flow via e. We can conclude that $dsep(A, B|\emptyset)$.

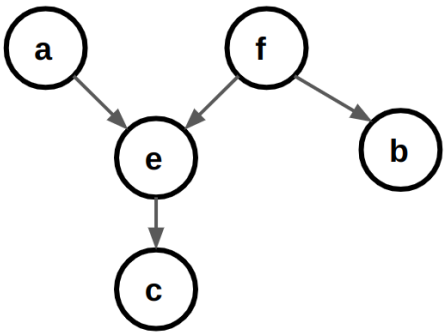


Figure 8.13: Example of general d-separation. $a \perp b|c$

Example of general d-separation 3
In this third example we consider the Bayesian Network in Figure 8.14. In this case a and b are d-separated by f:

- Node f is tail-to-tail and observed.

In this case node f blocks the flow of information. We can conclude that $a \perp b|f$. Intuitively, a and f are two possible causes for b. If f is observed than the cause is already known.

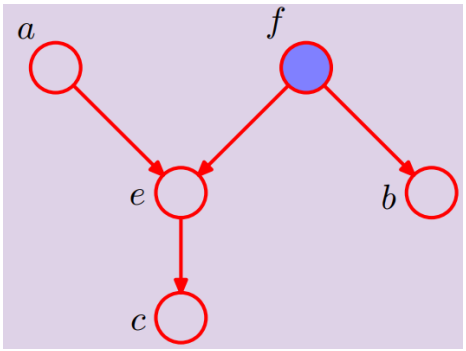


Figure 8.14: Example of general d-separation. $a \perp b|c$

Example of general d-separation 4
In this fourth example, we consider the network in Figure 8.15 and we investigate if A and B are d-separated or not ($dsep(A, B|C)$). In order to answer to

this question we have to examine all the possible paths between A and B. In correspondence of α there is a head-to-head connection and node α is observed. Hence, we can conclude that information flows from A to B and the statement $dsep(A, B|C)$ is not true.

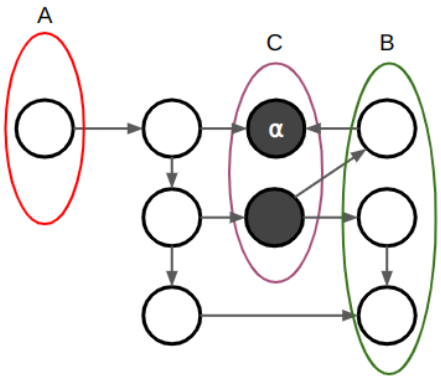


Figure 8.15: Example of general d-separation. $A \top B|C$

Example of general d-separation 5

In this fifth example, we consider the network illustrate in Figure 8.16. In this case the path 1-2-3-4 is blocked because in correspondence of node 3 there is a head-to-head connection but node 3 is not observed and it has no children. As a consequence we have to check other paths in order to understand whether or not $dsep(A, B|C)$. Moreover also paths 1-2-5-β-6 and 1-2-5-β-4 are blocked. However, along the chain 1-2-5-7-8 there are no observations. So along 1-2-5-7-8 information flows. We can conclude that A and B are not d-separated.

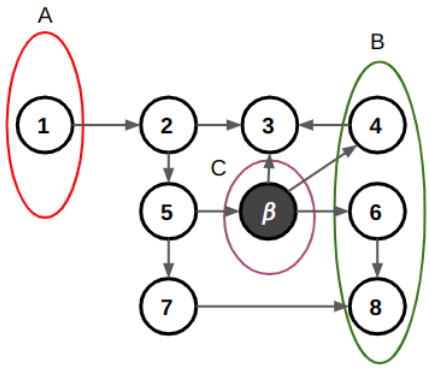


Figure 8.16: Example of general d-separation. $A \top B|C$

Example of general d-separation 6
In this sixth example we consider the network illustrated in Figure 8.17. Following the same reason as before, we understand that A and B are d-separated by C this time.

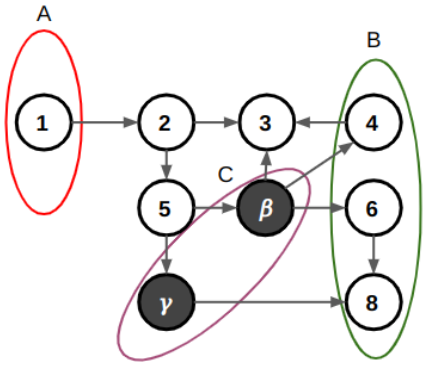


Figure 8.17: Example of general d-separation. $A \perp B|C$

8.2.2 BN independences revisited

- A BN structure $\mathcal{G}$ encodes a set of *local* independence assumptions:

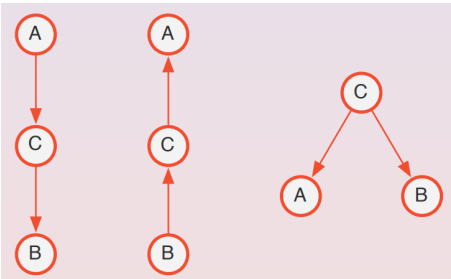
$$I_l(\mathcal{G}) = \{\forall ix_i \perp NonDescendants_{x_i}|Parents_{x_i}\}$$

- A BN structure $\mathcal{G}$ encodes a set of *global* (Markov) independence assumptions:

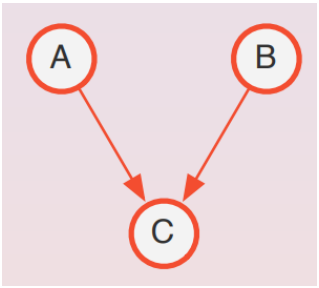
$$I(\mathcal{G}) = \{(A \perp B|C) : dsep(A; B|C)\}$$

8.2.3 BN equivalence classes

Quite different BN structures can actually encode the exact same set of independence assumptions. For example the three BNs in Figure 8.18a constitutes an equivalence class whose components encode the same independence assumptions. Figure 8.18b represents the second independence class which encodes the opposite (A and B are not independent, they become independent when C is observed) of the structures in Figure 8.18a. This concept can be generalized to arbitrary Bayesian Networks with more than three nodes. In general two BN structures (of course they have to be on the same set of nodes) $\mathcal{G}$ and $\mathcal{G}'$ are *I-equivalent* if $I(G) = I(\mathcal{G}')$. Following this relation, the space of BN structures over $\mathcal{X}$ is partitioned into a set of mutually exclusive and exhaustive *I-equivalence classes*.



(a) The three BNs constitutes an equivalence class whose components encode the same independence assumptions.



(b) A second independence class which encodes the opposite (A and B are not independent, they become independent when C is observed) of the structures in Figure 8.18a.

Figure 8.18: BN equivalence classes

8.3 I-maps vs distributions

As explained above, for a structure $\mathcal{G}$ to be an I-map for p, it does not need to encode all its independences. As a consequence a fully connected graph is an I-map of any p_i defined over its variables. Indeed, a fully connected graph does not encode any independency and so, for sure it does not encode an independency which is not in p. Obviously, a fully connected graph is not useful for our purposes.

Minimal I-map
A *minimal I-map* for p is an I-map $\mathcal{G}$ which can't be "reduced" into a $\mathcal{G}' \subset \mathcal{G}$ (by removing edges) that is also an I-map for p. In other words it is no more possible to remove edges from $\mathcal{G}$ without introducing independences which do not hold in p.

Remark: a minimal I-map for p does not necessarily capture all the independences in p.

Perfect Maps (P-maps)
A structure $\mathcal{G}$ is a *perfect map* (P-map) for p it captures all (and only) its independences:

$$I(\mathcal{G}) = I(p)$$

Remark: not all distributions have a P-map. Some cannot be modelled exactly by the BN formalism. In particular there are some independences which cannot be modelled by directed edges.

There exists an algorithm for finding a P-map of a distribution which is exponential in the in-degree of the P-map. The algorithm returns an equivalence class of structures which are a perfect map rather than a single structure.

8.4 Practical suggestions for building a Bayesian Network

- Get together with a domain expert.
- Define variables for entities that can be *observed* (e.g. symptoms, age, weight) or that you can be interested in *predicting* (e.g. pathology). Latent variables can also be sometimes useful. Latent variables are variables which we do not observe and we do not want to predict. However, they are mediators for what we need to predict. For instance, a certain pathology could depend on some groups of behaviours which could facilitate the prediction of the pathology.
- Try following *causality* considerations in adding edges (edges which encode causalities are more interpretable and they tend to produce sparser networks).
- In defining probabilities for configurations (almost) never assign zero probabilities. Otherwise, every example with that configuration which we will observe in the future will have zero probability.
- It is usually difficult in real world applications to assign probabilities to certain variables configurations. As a result we typically refine or estimates these parameters using data. If data are available, use them to help in *learning* parameters and structure of the network.

Chapter 9

Inference in BN

The criteria with which a Bayesian Network (Chapter 8) is built, allows us to get information about the data.

We assume that we have evidence e (what I observe) on the state of a subset of variables E in the model. Therefore Inference amounts as computing the posterior probability of a subset X of the non-observed variables given the observation:

$$P(X|E = e)$$

Inference is computing the probability of variable X (one of more non-observed variables) given the observed variables (posterior probability of X given the evidences).

A graphical model is designed as a network of joint probability, we can use messages to compute joint probability of the variable of interest and the evidence. Typically I need the conditional probability, then I need to compute (Bayes' theorem 5.2):

$$P(X|E = e) = \frac{P(X, E = e)}{P(E = e)}$$

Then I need to compute $P(X, E = e)$ and $P(E = e)$, the denominator is easy (once I compute the numerator $P(E = e) = \sum_X P(X, E = e)$ (through a marginalization)), therefore I need to compute the nominator.
The nominator is a joint probability, I want to exploit the BN structure to compute this numerator easily, otherwise it will explode exponentially. For this reason, we can exploit the structure of the Bayesian Network to get inference more easily, this will end up in a problem that can be solved using dynamic programming.

9.1 Inference in chain-like structure

Let's start from a linear chain of nodes, which will be our BN (Bayesian Network).

We want to compute the probability for a certain node given some evidence:

$$P(X|E=e) = \frac{P(X, E=e)}{P(E=e)}$$

9.1.1 Inference without evidence

Let's ignore the evidence first, then come back to that later. The probability of any of the nodes depends on the previous nodes as The joint distribution modelled on the BN is $P(X_1, \ldots, X_n)$, then if I am interested in $P(X_n)$ I know that it possibly depends on all the other possible variables (marginalization) This is where the exponential problem would come up.

$$P(X_n) = \sum_{X_i \neq X_n} P(X_1, \ldots, X_n)$$

Considering joint probability decomposed according to the chain is

$$P(X_n) = \sum_{X_1}\sum_{X_2}\cdots\sum_{X_{n-1}}\sum_{X_{n+1}}\sum_{X_n} P(X_1)P(X_2|X_1)P(X_3|X_2)\ldots P(X_n|X_{n-1})$$

But we notice that the sum over X_n has only a value in the decomposition which depends on it ($P(X_n|X_{n-1})$), the other probabilities are constant with respect to X_n. Therefore we represent this value as a function depending on X_{n-1}

$$\mu_\beta(X_{n-1}) = \sum_{X_n} P(X_n|X_{n-1})$$

So we get

$$P(X_N) = \sum_{X_1}\sum_{X_2}\cdots\sum_{X_{n-1}} P(X_1)P(X_2|X_1)P(X_3|X_2)\ldots P(X_{n-1}|X_{n-2})\mu_\beta(X_{n-1})$$

We can proceed with the same intuition for all the previous members

$$\mu_\beta(X_i) = \sum_{i+1} P(X_{i+1}|X_i)\mu_\beta(X_{i+1})$$

I go back with this procedure until $n+1$.

We can compute a similar procedure from the top to the bottom of the chain. Given

$$P(X_n) = \sum_{X_1}\sum_{X_2}\cdots\sum_{X_{n-1}} P(X_1)P(X_2|X_1)P(X_3|X_2)\dots P(X_n|X_{n-1})$$

notice that the terms that depend on the first summation $\sum_{X_1}$ are only $P(X_1)P(X_2|X_1)$, then we can carry on the same reasoning

$$\mu_\alpha(X_2) = \sum_{X_1} P(X_1)P(X_2|X_1)$$

where α stands for forward messages while β for reverse messages.

$$\sum_{X_2}\sum_{X_2}\cdots\sum_{X_{n-1}} \mu_\alpha(X_2)P(X_3|X_2)\dots P(X_n|X_{n-1})$$

In general:

$$\mu_\alpha(X_i) = \sum_{X_{i-1}} \mu_\alpha(X_{i-1})P(X_i|X_{i-1})$$

For the forward message we can stop when

$$\mu_\alpha(X_n) = \sum_{X_{n-1}} \mu_\alpha(X_{n-1})P(X_n|X_{n-1})$$

The only term that is left from the two chain is $P(X_n)$ that can be written as

$$P(X_n) = \mu_\alpha(X_n)\mu_\beta(X_n)$$

this means that computing the local probabilities and summing them independently, using (for a binary problem) a number of operation equal to $2(n-1)$ instead of 2^n. μ_α and μ_β are message sent backwards or forward from node to node. The message is always a function of the destination node: I am interested in computing the probability of the i^{th} node and I collect the messages coming from the adjacent nodes.

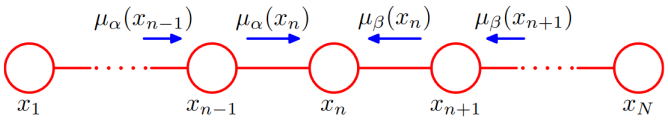


Figure 9.1: Inference for the i^{th} node based on the messages from the previous and next node passing

Nodes compute messages to be sent to the next node, as generalization, the message computed is a combination of a local probability of that node and the

message received from the previous (or next) nodes: this is a standard way to send messages also for non-chain structures.

It is clever to store the partial computations we can do the *full message passing*: sending a message from the top to the end of the chain and vice versa, so we have already computed all the partial computation that we could need.

9.1.2 Inference adding evidence

Typically we have some observed values, and then we want to compute the probability: instead of the sum over the values for the variable that I observed we can plug in the value I actually observed.
Let's say I observed $X_1 = x_{e1}, X_3 = x_{e3}$ in a chain of 4 elements and I what to compute X_2. This can be done as

$$P(X_2|X_3 = x_{e3}, X_1 = x_{e1}) = \frac{P(X_1, X_2, X_3, X_4)}{P(X_3 = x_{e3}, X_1 = x_{e1})}$$

the joint probability can be computed as

$$\sum_{X_4} P(X_1, X_2, X_3, X_4) = P(X_1 = x_{e1}, X_2, X_3 = x_{e3}, X_4)$$

where for the observed values I can plug in the values

$$\sum_{X_4} P(X_1 = x_{e1})P(X_2|X_3 = x_{e3}, X_1 = x_{e1})P(X_3 = x_{e3}))P(X_4|X_3 = x_{e3})$$

The only term that depends of X_4 is the last probability, therefore

$$\sum_{X_4} P(X_4|X_3 = x_{e3}) = \mu_\beta(X_3 = x_{e3})$$

and for the latter I do have the value observed. The inference procedure is actually the same, having evidence implies plugging in values instead of computing a summation.

After all it is true that:

$$P(X_n|\mathbf{X}_e = \mathbf{x}_e) = \frac{P(X_n, \mathbf{X}_e = \mathbf{x}_e)}{\sum_{X_n} P(X_n, \mathbf{X}_e = \mathbf{x}_e)} \tag{9.1}$$

with bold values as evidence.

9.2 Inference in trees-like structure

The solution highlighted for chains can be extended to trees. At this point we should consider different kind of structures:

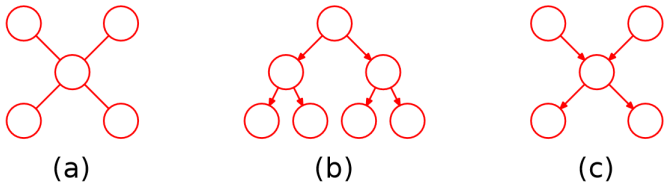


Figure 9.2: a) undirected trees b) directed trees c) directed poly-trees

(a) **undirected trees**: undirected graph with a single path for each pair of nodes

(b) **directed trees**: directed graph with a single node (root) with no parents, all other nodes with a single parent

(c) **directed poly-trees**: directed graphs with multiple parents, for node and multiple roots but still a single (undirected) path between each pair of nodes.

The procedure that we will described will work for directed (BN) and undirected models (Markov's Models, *MM*).

In order to facilitate the description (and the implementation) we need to go through an alternative representation of the graphical model: a *factor graph*.

Factor graph
It is a graphical representation of a graphical model highlighting its factorization. The factor graph is an undirected graph that has one node for each node in the original graph and also additional nodes for each factor (a factor node had undirected links to each of the node variables in the factor).

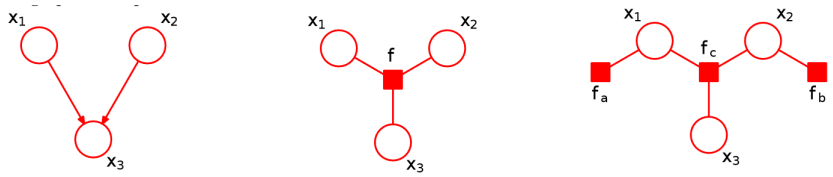


Figure 9.3: Example of two different factor graphs for the first directed graph.

A factor is a component of the joint distribution we are trying to represent. The value associated to the factor is a probability distribution. In Figure 9.3 we can look at two different realization of factor graphs for the easy directed graph on the left. The factor graph in the middle represent the following factorialization: $f(x_1, x_2, x_3) = p(x_3|x_1, x_2)P(x_1)P(x_2)$

while the most right one stands for
$f_a(x_1) = P(x_1)$
$f_b(x_2) = P(x_2)$
$f_c(x_1, x_2, x_3) = p(x_3|x_1, x_2)$
This last one means that
$P(x_1, x_2, x_3) = f_c(x_1, x_2, x_3) f_a(x_1) f_b(x_2)$
The latter is the one that we will use ad default option since is the most complete.

9.2.1 The sum-product algorithm

If we think of the inference algorithm on chain, we took the incoming message, multiply over the local probability, summing over one variable and sending the message to another variable (*sum-product alternation*): this procedure is efficient for tree-like structure (also chain as edge case).
On factor graphs it still can be applied if the factor graph it belongs to one of the classes described in Figure 9.2 (a factor graph will not be directed).

9.2.1.1 Inference without evidence

At this point I have no evidence, and no sequence. I have a generic joint probability (marginalize on everything except X in which I am interested)

$$P(X) = \sum_{\mathbf{X} \neq X} p(\mathbf{X})$$

Let us remind that in the chain case

$$P(X_n) = \mu_\alpha(X_n)\mu_\beta(X_n)$$

This version is the same but generalized to all the messages reaching the node (incoming), the node are only connected to factors (not nodes) therefore it is the product for all factors in the neighbourhood from the factor to X

$$P(X) = \prod_{f_s \in adj(X)} \mu_{f_s \to X}$$

where $f_s \to X$ stand for f_s as source and X as destination. The message is always a function of the variable *node* because the factor has more variables.

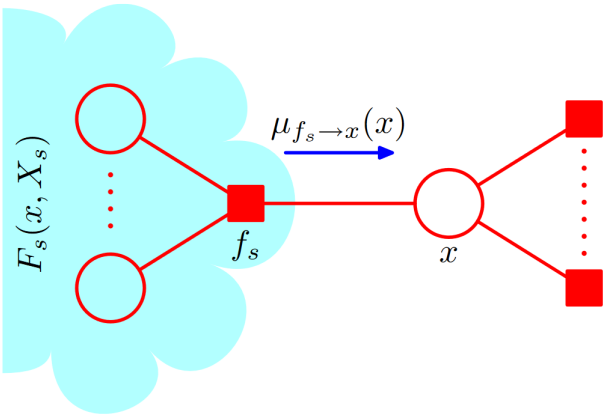


Figure 9.4: Collecting messages from the adjacent nodes of X (fwd message passing)

The value corresponding to the message that $f_s \to X$ is the multiplication of the incoming messages from its adjacent nodes multiplied by the local probability of the factor.

$$\mu_{f_s \to X} = \sum_{X_1} \cdots \sum_{X_m} f(X, X_1, \ldots, X_m) \prod_{X_i \in adj(f_s) \neq X} \mu_{X_i \to f_s}$$

with $x_1, \ldots, x_m$ the nodes that f_s receives messages from (but the destination X).

Computing the message from the node to a factor becomes simple: I have a node that needs to send a message to a factor, then it will be connected only to factors node $f_1, \ldots, f_n$. The message from $f \to X$ is always a function of the variable node, then we

- collect messages from the adjacent nodes, therefore multiply for all factors that are not the destination (this is the **product of the incoming messages** for all neighbours)

$$\mu_{X \to f}(X) = \prod_{f_i \in adj(X) \neq X} \mu_{f_i \to X}(X)$$

- if the messages is at the extremes, then we need to take care of leaves. We start sending messages from the leaves X, then the message will be equal to one since it has no incoming messages

$$\mu_{X \to f}(X) = 1$$

- if the extreme is a factor, then we will not have any incoming messages and from this

$$\mu_{f \to X} = \sum_{X_1} \cdots \sum_{X_m} f(X, X_1, \ldots, X_m) \prod_{X_i \in adj(f) \neq X} \mu_{X_i \to f}$$

we get that it is only function of one variables

$$\mu_{f \to X} = f(X)$$

We can start sending messages like this, and then things come together as sums and products.

The message computation in a tree structure gets more complicated than a chain (in which it is enough to start from top and end of the chain). Given a tree, we have a root already defined, but in the factor graph we do not.
Therefore we take one node (whatever) and use it as a root so we have leaves defined. From the leaves we send messages and each node will send messages further up. At some point the messages will reach the root. At this point we can compute the probability of the root as the product of the messages coming to the root.

Therefore the procedure for computing the probability for a generic node, is using it as root, making leaves send messages and eventually getting the messages from the adjacent factors of the node we are interested in.

In order to avoid useless computation, we send all messages up to the root, then from the root we send messages all down to the leaves. The same for the intermediate nodes: this is a *full messages passing scheme*. At this point we do not care about who the root is because we will have received all partial messages to all nodes and factors in each direction. Therefore we can compute whatever we want.

9.2.1.2 Example of inference without evidence

Let's consider the following graph as an example:

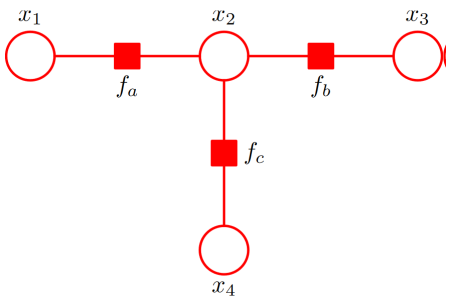


Figure 9.5: Example graph

The joint probability encoded in this factor graph is the product of the three factors, with $f_a(x_1, x_2), f_b(x_2, x_3), f_c(x_4, x_2)$. Therefore it is the multiplication

$$P(X) = f_a(x_1, x_2) f_b(x_2, x_3) f_c(x_4, x_2)$$

Let's do inference choosing x_3 as root (by chance), then x_1, x_4 are leaves. x_1, x_4 can start sending messages to the root x_3. Since x_1, x_4 are (leaf) nodes, then their message is equal to 1

$$\mu_{x_1 \to f_a}(x_1) = 1$$
$$\mu_{x_4 \to f_c}(x_4) = 1$$

Once these messages are received by f_a, F_c, then the factors can send messages to the next nodes.

$$\mu_{f_a \to x_2}(x_2) = \mu_{x_1 \to f_a}(x_1) \sum_{x_1} f_a(x_1, x_2)$$
$$= \sum_{x_1} f_a(x_1, x_2)$$

and

$$\mu_{f_c \to x_2}(x_2) = \mu_{x_4 \to f_c}(x_4) \sum_{x_4} f_c(x_4, x_2)$$
$$= \sum_{x_4} f_c(x_4, x_2)$$

At this point x_2 received two messages and can compute its own message to send to f_b as

$$\mu_{x_2 \to f_b}(x_2) = \mu_{f_a \to x_2}(x_2) \mu_{f_c \to x_2}(x_2)$$

And now f_b sends its message to x_3 multiplying it by its internal factor

$$\mu_{f_b \to x_3}(x_3) = \mu_{x_2 \to f_b}(x_2) \sum_{x_2} f_b(x_2, x_3)$$

This is the message reaching x_3, which has only one neighbour, so its probability it is only the incoming message $\mu_{f_b \to x_3}(x_3)$. Replacing bits of equations we get

$$P(x_3) = \sum_{x_1} \sum_{x_2} \sum_{x_4} f_a(x_1, x_2) f_b(x_2, x_3) f_c(x_4, x_2)$$

Please notice that the result is as expected the sum over all the variables except the destination. At this point we can propagate backwards until every node received messages from all neighbours.

9.2.1.3 Another example

Let's consider now the following graph with its factor graph

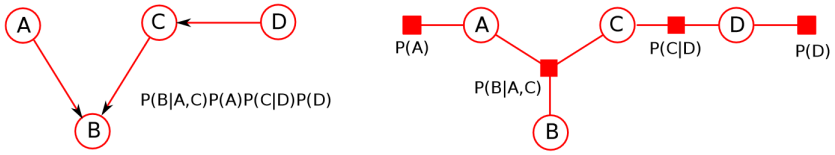


Figure 9.6: Example graph with possible factor graph

Suppose we want to compute the probability for B. What we do is sending messages towards B (both factors and intermediate nodes). Nodes A, D propagate messages, while C needs to wait the message from the previous factor $f_{C,D\to C}$ and then propagate.
Once the middle factor received both messages from A, C then can propagate the message to B after multiplying by its factor over the adjacent variables that are not the destination node.

$$\begin{aligned}\mu_{f_{A,B,C}\to B}(B) &= \sum_A \sum_C P(B|A,C)\mu_{C\to f_{A,B,C}}(C)\mu_{A\to f_{A,B,C}}(A) \\ &= \sum_A \sum_C P(B|A,C)P(A)\sum_D P(C|D)P(D) \\ &= \sum_A \sum_C \sum_D P(A,B,C,D)\end{aligned}$$

9.2.2 Most probable configuration, sum-product algorithm

Suppose that we have a BN network describing the probability of having a certain disease. I want to know if a person has a certain pathology. Suppose we have a number of pathologies that can be correlated to the same sets of symptoms. Maybe some symptom is indicative of a certain disease over another. I want to get the **most probable configuration** as the most probable configuration for the variable I do not know (having the disease). It is highly unlikely that the symptoms are caused from more than one disease.

$$\mathbf{X}^{max} = argmax_{\mathbf{X}} p(X)$$

this is the most probable explanation that *jointly* maximizes the probability. Therefore we need to maximize on all possible causes.

$$\begin{aligned}p(\mathbf{X}^{max}) &= max_{\mathbf{X}} p(\mathbf{X}) \\ &= max_{x_1} max_{x_2} \dots max_{x_m} p(\mathbf{X})\end{aligned}$$

This means jointly maximizing over all variables. Depending on how $p(\mathbf{X})$ gets decomposed, I can do some local computation independently from the rest. I can apply the very same algorithm where I only replace $\sum \to max$.
For the chain-like structure, we end up with the very same reasoning proposed in the previous sections:

$$\begin{aligned}max_{\mathbf{X}} p(\mathbf{X}) &= max_{x_1} \dots max_{x_n} [p(x_1)p(x_2|x_1)\dots p(x_n|x_{n-1})] \\ &= max_{x_1}[p(x_1)p(x_2|x_1)] \dots max_{x_n}[p(x_{n-1}|x_n)]\end{aligned}$$

We can call these local computations, again, *messages*:

$$\mu_\beta(x_{n-1}) = max_{x_n}[p(x_n|x_{n-1})$$

which won't be equal to one, but it is the probability. In the general message passing algorithm (sum-product algorithm), we replace the sum with a max:

$$\mu_{f\to x}(x) = max_{x_1,\dots,x_m}\left[f(x,x_1,\dots,x_m)\prod_{x_m\in adj(x)\neq x}\mu_{x_m\to f}(x_m)\right]$$
$$\mu_{x\to f}(x) = \prod_{f_j\in adj(x)\neq f}\mu_{f_j\to x}(x)$$

There is however a problem with this operation: if I compute the maximization locally (starting from the end and going to to the top) I do not know the configuration that maximize the configuration. What we need to do is backtracking the information to recover the values of the previous variables. Here is how we do it on a tree structure given a node that we decided to be the root. The probability with the maximum configuration is obtained as

$$p(\mathbf{X}^{max}) = max_{X_r}\left[\prod_{f_j\in adj(X_r)}\mu_{f_j\to X_r}(X_r)\right]$$

The maximal configuration *only* for a root r is:

$$X_r^{max} = argmax_{X_r}\left[\prod_{f_j\in adj(X_r)}\mu_{f_j\to X_r}(X_r)\right]$$

I need the configuration also for the other nodes. When I compute the messages, in principle I also know what is the configuration of the variables I am maximizing over that will give me a certain configuration for x. I need to store the information of the argmax message that is the configuration of the other variables that gave me the configuration. I store another message $\Phi_{f\to X}(X)$ for this reason. For each step I store the max message in μ and its configuration in Φ in the argmax message.

9.2.2.1 Example

When I get to the end of the chain, I have a message from the node x_{n-1} coming from the last factor. If I maximize over x_n I get the maximum probability, while if I compute the argmax I get the maximal configuration for x_n ($f_{n-1,n}\to x_n(x_n)$ stands for the factor that connects $n-1$ to n nodes and that sends message to node x_n).

$$x_n^{max} = argmax_{x_n}[\mu_{f_{n-1,n}\to x_n}(x_n)]$$

If I use this other message Φ (configuration message) plugging x_n^{max} then I will get the configuration of the variable that maximized the value (therefore the most probable)

$$x_{n-1}^{max} = \Phi_{f_{n-1,n}\to x_n}(x_n^{max})]$$

The I do the same for the previous messages

$$x_{n-2}^{max} = \Phi_{f_{n-2,n-1} \to x_{n-1}}(x_{n-1}^{max})]$$

$$\dots$$

$$x_1^{max} = \Phi_{f_{1,2} \to x_2}(x_2^{max})]$$

After all the variables, I got the most probable configuration on all the variables. The back propagation of the last value gives you the most probable configuration of the previous nodes.
For better understanding, we now consider an example with three state of variables (*trellis for linear chain*).

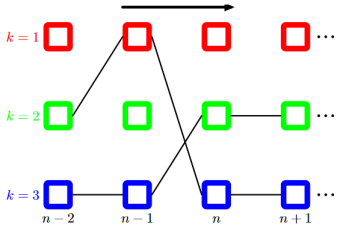


Figure 9.7: Trellis for linear chain

A trellis diagram shows k possible states of a variable (in this case $k = 3$), the states for each node are represented in n columns, times goes left to right. For each state k of a variable X_n, $\Phi_{f_{n-1,n} \to X_n}(X_n)$ defines a unique state which is maximal given the previous states (in the diagram is linked by an edge). Once the maximal states for the last variable X_n is chosen, that I back-propagate through the links in order to get the most probable configuration of the previous nodes.
This procedure works also for trees with the exception that we need to back-propagate multiple times. The reasoning is still the same.

This is needed because if I want the most probable explanation, I do not just want the probability but I am also interested in the most probable configuration for that to happen. Therefore I propagate the probability message μ and the configuration message Φ.

9.2.2.2 Underflow issue

When we compute products of probabilities ($0 < p < 1$) the number becomes smaller and smaller: therefore we can get across the *underflow issue*. In order to avoid this problem we typically use the log since it is monotonic and allows to replays multiplication with summation.

9.2.3 Exact inference on general graphs

These algorithm can not be applied on graphs that do not produce tree-structure factor graphs. If we have BN that do not satisfy these conditions (like containing undirected loops) these algorithms can not be applied anymore but there are very similar algorithms (that use slightly similar intermediate structures as *junction trees* instead of factor graphs). In junction trees the nodes contain clusters of variables (which can contain loops) and the apply the very same idea described in the previous paragraphs. This algorithm becomes exponential with the size of the cluster. What happens in reality is that we need to approximate because we can not afford exact inference in these cases. There is a lot of research going on that looks for the best way to approximate inference such as:

- **looply belief propagation** that consists in message passing on the original graph even if it contains loops
- **variational methods** that are deterministic approximations assuming the posterior probability factorizes in a particular way
- **sampling methods** that approximate posterior with getting samples from the network

In the looply belief propagation we build a factor graph, the message flows indefinitely. This methods allows the message passing through loops deciding a scheme of message sending. The node possibly recomputes some messages, this implies that for every iteration, values are recomputed. If the value shrinks, the network is converging (all messages do not change), but there is no guarantee that this will happen. The maximal configurations and maximum probabilities will not be exact but approximations.

In the sampling methods, I get a set of configuration of this network which is compatible with the probability. Then we can sample from the distribution and check if they are compatible with the configuration we intend to verify. It is tricky to sample from the distribution. So what we do is Monte Carlo Markov Chain (MCMC): we do not sample from posterior, but from a random process that get progressively closer to the posterior. This means that the state of the process is a configuration, and a new state is generated from moving some variables (change of state). We move in this space according to a probabilistic model, given certain conditions, it converges to a configuration that can be used as samples.

Chapter 10

Learning BN

As explained in Section 8.4 when building a Bayesian Network it is crucial to collaborate with a domain expert in order to follow as much as possible causality relationships among variables. At the beginning of this section we assume that the structure of the model is given.
We are given a dataset (training set) of examples $\mathcal{D} = \{\boldsymbol{x}(1), ..., \boldsymbol{x}(N)\}$. Each example $\boldsymbol{x}(i)$ is a configuration for *all* (complete data) or *some* (incomplete data) variables in the model (which is the Bayesian network).

Task: we need to estimate the parameters of the model (conditional probability distributions) from the data.

The simplest approach consists of learning the parameters maximizing the likelihood of the data:

$$\boldsymbol{\theta}^{max} = argmax_{\boldsymbol{\theta}} p(\mathcal{D}|\boldsymbol{\theta}) = argmax_{\boldsymbol{\theta}} \mathcal{L}(\mathcal{D}, \boldsymbol{\theta})$$

As we have discussed in Section 7.4.1.1 this method is referred to as *Maximum likelihood estimation*. In the formula $\mathcal{L}(\mathcal{D}, \boldsymbol{\theta})$ is the likelihood function to maximize with respect to the parameters.

Probabilistic graphical models allow to graphically represent relationships among examples and parameters. Suppose we have three variables linked together with a tail-to-tail configuration as illustrated in Figure 10.1.

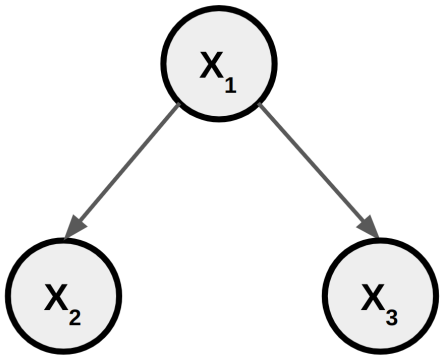


Figure 10.1: Three variables linked together with a tail-to-tail configuration.

What is more, we have N examples of this Bayesian network as schematized in Figure 10.2.

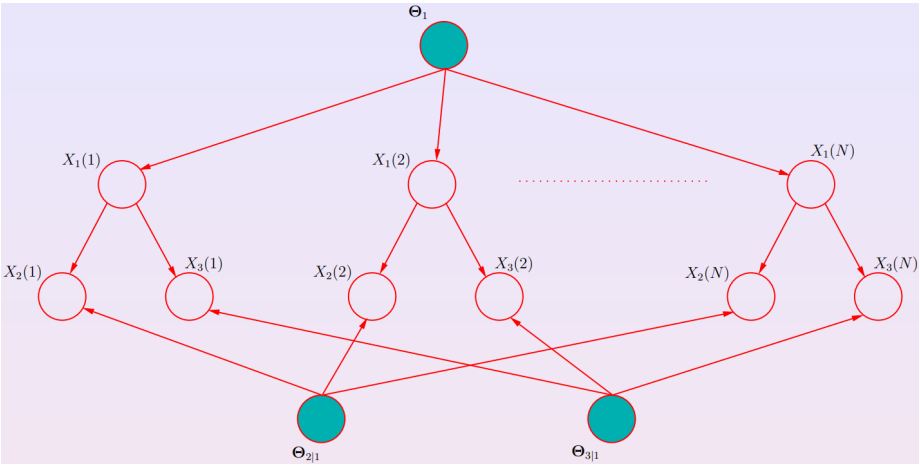


Figure 10.2: N examples of the Bayesian network reported in Figure 10.1.

At this point, our aim is to write down an expression for $p(\mathcal{D}|\boldsymbol{\theta})$. Since data are iid given the parameters $\boldsymbol{\theta}$, then we can write:

$$p(\mathcal{D}|\boldsymbol{\theta}) = \prod_{i=1}^{N} p(\boldsymbol{x}(i)|\boldsymbol{\theta})$$

Now, we can go through the factorization of the BN to rewrite $p(\boldsymbol{x}(i)|\boldsymbol{\theta})$:

$$p(\mathcal{D}|\boldsymbol{\theta}) = \prod_{i=1}^{N} \prod_{j=1}^{m} p(\boldsymbol{x}_j(i)|pa_j(i), \boldsymbol{\theta})$$

Notice that in Figure 10.2, we replicate the model N times. Each instance of the BN is related to a different example.
When computing $p(\boldsymbol{x}_j(i)|pa_j(i))$, this probability depends only on a subset of the parameters $\boldsymbol{\theta}_{X_j|pa_j}$ which are interesting (are part of the table) in that particular conditional probability distribution. So, the probability $p(\boldsymbol{x}_j(i)|pa_j(i))$ depends only on $\boldsymbol{\theta}_{X_j|pa_j}$ and does not depend on other parameters in the network.

$$p(\mathcal{D}|\boldsymbol{\theta}) = \prod_{i=1}^{N}\prod_{j=1}^{m} p(\boldsymbol{x}_j(i)|pa_j(i), \boldsymbol{\theta}_{X_j|pa_j}) \tag{10.1}$$

This concept is described graphically in Figure 10.2 as follows:

- $\boldsymbol{\Theta}_1$ are the parameters associated to X_1. Evidently, X_1 is not a conditional probability distribution.
- $\boldsymbol{\Theta}_{2|1}$ are the parameters associated to X_2.
- $\boldsymbol{\Theta}_{3|1}$ are the parameters associated to X_3.

It is important to notice that the directional arrows are from the parameters to the nodes in the network. Indeed, at a high level the structure models $p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})$. So parameters are parents with respect to the data.

Remark: If we try to build a Bayesian network which has the factorization of Equation 10.1 we would recover the network structure illustrated in Figure 10.2. For instance, suppose we have three nodes X_1, X_2, X_3 and $N = 2$ examples. Then from Equation 10.1 we get:

$$\begin{aligned} p(\mathcal{D}|\boldsymbol{\theta}) &= \prod_{i=1}^{N}\prod_{j=1}^{m} p(\boldsymbol{x}_j(i)|pa_j(i), \boldsymbol{\theta}_{X_j|pa_j}) \\ &= p(X_1(1)|\theta_{X_1})p(X_2(1)|X_1(1), \theta_{X_2|X_1})p(X_3(1)|X_1(1), \theta_{X_3|X_1}) \\ &\quad p(X_1(2)|\theta_{X_1})p(X_2(2)|X_1(2), \theta_{X_2|X_1})p(X_3(2)|X_1(2), \theta_{X_3|X_1}) \end{aligned}$$

Notice that the parameters $\theta_{X_1}, \theta_{X_2|X_1}, \theta_{X_3|X_1}$, do not depend on the specific example.
If we try to draw the corresponding network we get the model illustrated in Figure 10.3.

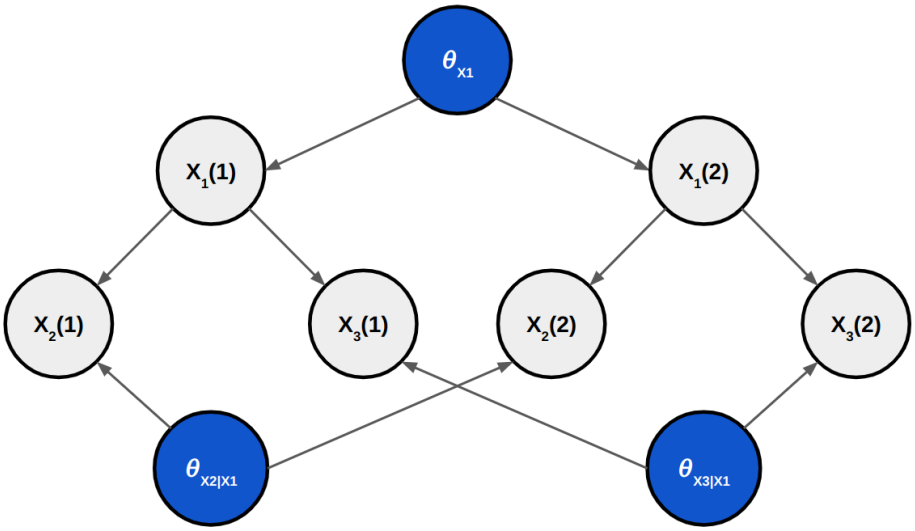


Figure 10.3: $N = 2$ examples of the Bayesian network reported in Figure 10.1.

10.1 Maximum likelihood estimation, complete data

The parameters of each CPD (conditional probability distribution) can be estimated independently:

$$\boldsymbol{\theta}^{max}_{X_j|pa_j} = argmax_{\boldsymbol{\theta}_{X_j|pa_j}} \prod_{i=1}^{N} p(\boldsymbol{x}_j(i)|pa_j(i), \boldsymbol{\theta}_{X_j|pa_j}) = \mathcal{L}(\boldsymbol{\theta}_{X_j|pa_j}, \mathcal{D}) \qquad (10.2)$$

In other words, if we want to maximize with respect to $\boldsymbol{\theta}_{X_j|pa_j}$, we do not take care about other $\boldsymbol{\theta}_{X_{j'}|pa_{j'}}$ such that $j \neq j'$. Indeed, we are maximizing $\prod_{i=1}^{N}\prod_{j=1}^{m} p(\boldsymbol{x}_j(i)|pa_j(i), \boldsymbol{\theta}_{X_j|pa_j})$ with respect to the parameters related to the j^{TH} node. In the product, everything which is not j is constant with respect to j, as a result we can ignore everything which is not j when maximizing. In Equation 10.2 we remove the product over j, since we concentrate on a particular value of j.
We have to compute a maximization of this kind for each node of the network and so for each conditional probability distribution. A discrete CPD $P(X|\boldsymbol{U})$, can be represented as a table (an example is reported in Figure 10.4) with:

- a number of rows equal to the number $Val(X)$ (2 in the case of Figure 10.4) of configurations for X
- a number of columns equal to the number $Val(\boldsymbol{U})$ (3 in the case of Figure 10.4) of configurations for its parents $\boldsymbol{U}$ (one value if there is one parent, multiple values if there is more than one parent)

- each table entry $\theta_{X|\boldsymbol{u}}$ indicating the probability of a specific configuration of $X = x$ and its parents $\boldsymbol{U} = \boldsymbol{u}$

Replacing $p(x(i)|pa(i))$ with $\theta_{x(i)|\boldsymbol{u}(i)}$, the local likelihood of a single CPD becomes:

$$\begin{aligned}\mathcal{L}(\boldsymbol{\theta}_{X|Pa}, \mathcal{D}) = \\ &= \prod_{i=1}^{N} p(x(i)|pa(i), \boldsymbol{\theta}_{X|Pa}) &= \\ &= \prod_{i=1}^{N} \theta_{x(i)|\boldsymbol{u}(i)} &= \\ &= \prod_{\boldsymbol{u} \in Val(\boldsymbol{U})} [\prod_{x \in Val(X)} \theta_{x|\boldsymbol{u}}^{N_{\boldsymbol{u},x}}]\end{aligned}$$

where $N_{\boldsymbol{u},x}$ is the number of times the specific configuration $X = x, \boldsymbol{U} = \boldsymbol{u}$ was found in the data.

Remark: in principle we should keep the j index. However, we omit it since node j is independent with respect to all other nodes. As a result we can present results for a generic node. Following this approach, each node is treated independently.

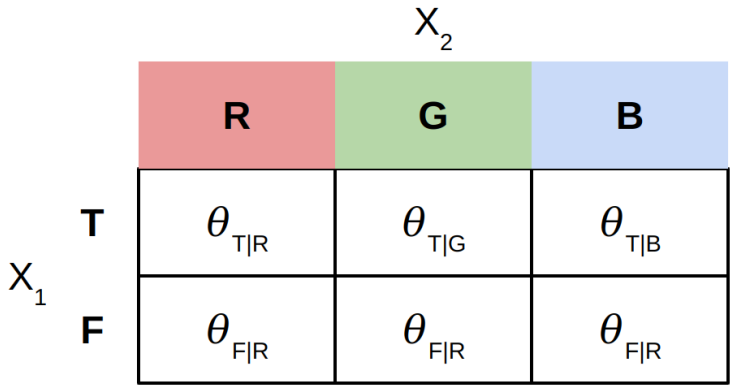


Figure 10.4: A discrete CPD represented as a table.

In essence, for all possible parameter in the conditional probability distribution, the likelihood becomes $parameter^{numberoftimesIseethatconfigurationinmydataset}$

A column in the CPD table contains a multinomial distribution over values of X for a certain configuration of the parents $\boldsymbol{U}$. Thus, each column should sum to one:

$$\sum_{x} \theta_{x|\boldsymbol{u}} = 1$$

Parameters of different columns can be estimated independently. For each multinomial distribution, zeroing the gradient of the maximum likelihood and considering the normalization constraint, we obtain:

$$\theta^{max}_{x|\boldsymbol{u}} = \frac{\boldsymbol{N}_{\boldsymbol{u},x}}{\sum_x \boldsymbol{N}_{\boldsymbol{u},x}}$$

At the end of the day, the maximum likelihood parameters are simply the fraction of times in which the specific configuration was observed in the data.

Example
In this example we consider the situation illustrated in Figure 10.4. Suppose that our dataset is populated as follows:

X_1	X_2	X_3
T	R	F
T	R	F
T	G	T
F	R	F
T	B	T
T	B	T
T	B	T

In this case the likelihood function which we want to maximize becomes:

$$\mathcal{L}(\boldsymbol{\theta}_{X|Pa}, \mathcal{D}) = \theta^2_{T|R}\theta_{T|G}\theta^3_{T|B}\theta_{F|R} \tag{10.3}$$

Remember that the columns in the table must sum to one. As a consequence, the purpose of the problem is to maximize Equation 10.3 subject to the fact that the columns in the table sum to one. As a result, there are no connections among columns. So, in 10.3 we consider $\theta^2_{T|R}$ and $\theta_{F|R}$ together because they have to sum to one. On the other hand, for $\theta_{T|G}$ and $\theta^3_{T|B}$ we have no constraints with respect to the other variables in the formula, so we can maximize them independently. In essence, in the maximization, each column (i.e. a configuration for the parents) can be treated independently. Maximizing a single column means maximizing over a single distribution which corresponds to the column. So, we face the maximization problem, dividing it into pieces until we end up maximizing individual distributions.
In this case:

$$\theta^{max}_{x|\boldsymbol{u}} = \frac{\boldsymbol{N}_{\boldsymbol{u},x}}{\sum_x \boldsymbol{N}_{\boldsymbol{u},x}} \tag{10.4}$$

- $\theta_{T|R} = \frac{2}{3}$
- $\theta_{T|G} = \frac{1}{1}$
- $\theta_{T|B} = \frac{3}{3}$
- $\theta_{F|R} = \frac{1}{3}$

- $\theta_{F|G} = 0$
- $\theta_{F|B} = 0$

Remark: this is a toy example. In real world scenarios, it is not ideal to have probability 1 or probability 0. For example, if the estimation of a probability is zero it nullify the joint probability of the whole network. In order to deal with this problem, we introduce priors.

10.1.1 Adding priors

Maximum likelihood estimation tends to overfit the training set. Configuration not appearing in the training set will receive zero probability. A common approach consists of combining maximum likelihood with a prior probability on the parameters, achieving a maximum-a-posteriori estimate:

$$\boldsymbol{\theta}^{max} = argmax_{\boldsymbol{\theta}} p(\mathcal{D}|\boldsymbol{\theta}) p(\boldsymbol{\theta}) \tag{10.5}$$

In this case, our aim is to compute the configuration $\boldsymbol{\theta}^{max}$ which maximizes not the likelihood but the posterior.

Remark: the posterior is $p(\boldsymbol{\theta}|\mathcal{D})$:

$$p(\boldsymbol{\theta}|\mathcal{D}) = \frac{p(\mathcal{D}|\boldsymbol{\theta}) p(\boldsymbol{\theta})}{p(\mathcal{D})}$$

$p(\mathcal{D})$ does not depend on $\boldsymbol{\theta}$. So, maximizing $p(\boldsymbol{\theta}|\mathcal{D})$ is the same as maximizing $p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})$ since $p(\mathcal{D})$ is constant ($p(\boldsymbol{\theta}|\mathcal{D}) \propto p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})$).

The conjugate (read natural) prior for a multinomial distribution is a Dirichlet distribution with parameters $\alpha_{x|\boldsymbol{u}}$ for each possible value of x. The resulting maximum-a-posteriori estimate is:

$$\theta^{max}_{x|\boldsymbol{u}} = \frac{\boldsymbol{N}_{\boldsymbol{u},x} + \alpha_{x|\boldsymbol{u}}}{\sum_x (\boldsymbol{N}_{\boldsymbol{u},x} + \alpha_{x|\boldsymbol{u}})} \tag{10.6}$$

The prior is like having observed $\alpha_{x|\boldsymbol{u}}$ imaginary samples with configuration $X = x$, $\boldsymbol{U} = \boldsymbol{u}$.
For example, we could assume to have seen each configuration 1 (or a fraction of 1) times by coherently setting each $\alpha_{x|\boldsymbol{u}}$. In this way we avoid to get zero probabilities.

10.2 Maximum likelihood estimation, incomplete data

Up to this point we have considered only complete data such that, for each example we have fully observed data (we know the value for every variable). With

incomplete data, some of the examples miss evidence on some of the variables. Counts of occurrences of different configurations cannot be computed if not all data are observed. The full Bayesian approach of integrating (in the discrete case, sum up) over missing variables is often intractable in practice. We need approximate methods to deal with the problem.

Example: for example we could have question marks for some configurations:

X_1	X_2	X_3
T	R	F
?	R	F
T	G	T
F	?	F
T	B	T
T	?	T
T	B	T

This is the case of the medical domain where it is unlikely that a person does all the possible exams. This of course results in a lot of missing data.

The idea is to take advantage of the graphical model in order to compute the probability of what we do not know given what we know. For instance, in the context of the table above, we could use the graphical model to compute $P(X_1 = T|X_2 = R, X_3 = F)$. This latter becomes a replacement for the corresponding entry of the table. The question mark is replaced by a probability of being true or false.
However, in order to compute the probability $P(X_1 = T|X_2 = R, X_3 = F)$ we need the parameters $\theta_{T|R}, \theta_{T|G}, \theta_{T|B}, \theta_{F|R}, \theta_{F|G}, \theta_{F|B}$ which we are trying to estimate. This chicken and egg problem is solved using an iterative procedure. This last is called *Expectation-Maximization*.

10.2.1 Expectation-Maximization

Sufficient statistics (counts) cannot be computed due to missing data. Remember that the counts are the sufficient statistics, i.e. all we need of the data. At the beginning we can initialize the parameters in some ways as we discuss in the following. After that, we fill-in missing data inferring them using current parameters by solving inference problem to get expected counts. In other words, we compute the probability of a certain configuration given the current parameters and what we already know (the non-missing data). Using these probabilities we can compute the so called *expected counts*. These lasts are named "expected" since we do not have certain values for some variables. Then, we compute parameters maximizing likelihood (or posterior) of such expected counts (using the expected counts as if they are the exact counts). Finally, we iterate the procedure to improve the quality of the parameters. Indeed, with the updated parameters we can re-compute the expected counts. The procedure is iterated

until nothing changes anymore (i.e. there is no parameters update). This procedure is guaranteed to converge to a local maximum of the likelihood. The quality of this local optimum depends significantly on the initial point which is the initial configuration of the parameters.

10.2.2 Expectation-Maximization algorithm: e-step

Our aim is to compute the expected sufficient statistics (i.e. the expected counts) for the complete dataset, with expectation taken with respect to the joint distribution for $\boldsymbol{X}$ ($\boldsymbol{X}_l$ is what I know about the l^{TH} example, i.e. the evidence) conditioned of the current value of $\boldsymbol{\theta}$ (current estimate of the parameter) and the known data $\mathcal{D}$:

$$E_{p(\boldsymbol{X}|\mathcal{D},\boldsymbol{\theta})}[N_{ijk}] = \sum_{l=1}^{n} p(X_i(l) = x_k, Pa_i(l) = pa_j|\boldsymbol{X}_l, \boldsymbol{\theta}) \tag{10.7}$$

- In the formula N_{ijk} is the count for the configuration where example X_i takes value x_k and the parents Pa_i of the example X_i takes value pa_j. So, i is the node in the network, k is the value for that node, j is the value of the parent. i identifies the table, j and k identify the entry of the table.
- If $X_i(l)$ and $Pa_i(l)$ are observed for $\boldsymbol{X}_l$, it is either zero or one. (In the case of complete data, for each example we check if the particular configuration holds for that example and we sum up obtaining an integer which is the real count).
- Otherwise, run Bayesian inference to compute probabilities from observed variables. ($\sum_{l=1}^{n}$ is the sum over the set of examples) ($X_i(l)$ represents node i in the l^{TH} example) ($Pa_i(l)$ represents the parent of node i in the l^{TH} example)

This step is called *expectation step* (e-step).

10.2.3 Expectation-Maximization algorithm: m-step

The second step is called *maximization step* (m-step). The purpose of this step is to compute parameters maximizing the likelihood of the complete dataset D_c using the expected counts.

$$\boldsymbol{\theta}^* = argmax_{\boldsymbol{\theta}} p(D_c|\boldsymbol{\theta}) \tag{10.8}$$

In the formula, D_c represents the dataset filled with the expected counts. For each multinomial parameter θ^*_{ijk} evaluates to:

$$\theta^*_{ijk} = \frac{E_{p(\boldsymbol{X}|\mathcal{D},\boldsymbol{\theta})}[N_{ijk}]}{\sum_{k=1}^{r_i} E_{p(\boldsymbol{X}|\mathcal{D},\boldsymbol{\theta})}[N_{ijk}]} \tag{10.9}$$

Actually, this is an extension of Equation 10.4 using expected counts instead of actual counts.

Remark: maximum likelihood estimation can be replaced by maximum a-posteriori (MAP) estimation giving:

$$\theta^*_{ijk} = \frac{\alpha_{ijk} + E_{p(\boldsymbol{X}|\mathcal{D},\boldsymbol{\theta})}[N_{ijk}]}{\sum_{k=1}^{r_i}(\alpha_{ijk} + E_{p(\boldsymbol{X}|\mathcal{D},\boldsymbol{\theta})}[N_{ijk}])} \tag{10.10}$$

At the end of this step, we get a new estimate of the parameters $\boldsymbol{\theta}$. We iteratively execute the e-step with the updated parameter estimation. The iterative procedure proceeds in this manner until convergence.

10.3 Learning structure of graphical models

So far, we have assumed that the structure of the network is given (and we try to learn the parameters). However, this is not always the case of course. If we have domain knowledge we could try to construct the graphical model. However, this is not always possible. Typically, we know some relationships, but we do not know everything. In order to face this problem, the literature suggests three main approaches:

- **Constraint-based:** test conditional independencies on the data and construct a model satisfying them. With this approach we rely on hypothesis testing techniques to verify dependency or independency between pairs of variables. Then based on the result of these tests, we add connections to the network. Typically this method does not allow to learn the directions of the connections.
- **Score-based:** assign a (probabilistic) score to each possible structure, define a search procedure looking for the structure maximizing the score. With this approach we move in the space of possible structures guided by the score.
- **Model-averaging:** given that we do not know the network structure, we treat it as a random variable. We assign a prior probability to each structure, and average prediction over all possible structures weighed by their probabilities (full Bayesian, intractable since the number of possible structures is exponential with respect to the number of variables). This approach is a kind of ensemble learning method where we generate more than one structure and average the prediction in a similar way as we have seen for random forests. This could increase the robustness of the predictions but leads often to less explainable results.

Chapter 11

Naive Bayes

Let's now consider a simple classifier that can be used to exploit a BN.

11.1 Naive Bayes

We are facing the problem of classification. BN try to model the domain with variables and relationship that make sense of the data. When we have a set of input and an output to be computed (standard setting for supervised learning) starting from the knowledge. In plain classification we know that a set of known variables in input $\vec{x}$ and a target y with no ambiguity.
The idea of naive bayes classifier is a probabilistic algorithm for classification. We decide we want to compute the argmax given the data.

$$y* = argmax_{y \in Y} P(y|\vec{x}) = argmax_{y \in Y} \frac{P(\vec{x}|y)P(y)}{P(\vec{x})}$$

but since $P(\vec{x})$ is constant (given) then we can simply consider

$$y* = argmax_{y \in Y} P(\vec{x}|y)P(y)$$

Now, here we are not able to model $P(\vec{x}|y)$ the joint probability (when $\vec{x}$ increases dramatically in dimension).

$$P(\vec{x}|y) = P(x_1, \ldots, x_n|y)P(y)$$

When everything is binary I get 2^n configuration to be modelled, which is pretty complex. I would need tons of data in order to learn every configuration. This is not always feasible.

We consider a simplification as assumption: **a naive bayes assumes that the different features $(x_1, \ldots, x_n)$ of the input are independent given**

the class. This means that I can simplify the probability into:

$$P(y|\vec{x}) = \prod_{i=1}^{n} P(x_i|y)P(y)$$

with n number of features. This is rather simple because we consider separate table for each features, therefore we can estimate parameters locally and this allows to have much less parameters. This classifier is not always accurate (it is indeed naive) but it is a good starting point. The bayesian network will look like this:

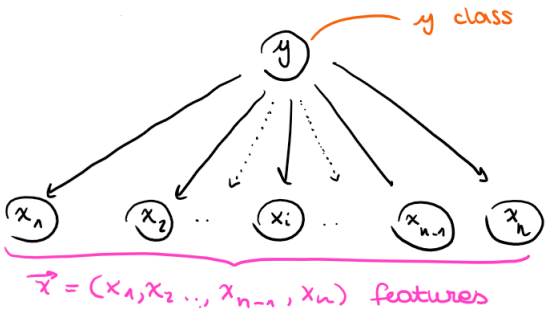


Figure 11.1: basic representation of a naive NB

where we have a connection from the class y to each of the features $x_1, \ldots, x_n$ but features are not connected to each other.

11.2 Text classification

Let' consider having a piece of text x and we consider words as features

$$\vec{x} = (w_1, \ldots, w_n)$$

The naive bayes assumption says that the probability of finding a word or another is the same (so the probabilities are independent). The piece of text is turned into a *bag of words*, therefore we forget about the position.
This is a simplifying assumption because words that are often associated together (machine + learning, artificial + intelligence ...) are learned independently but as a matter of fact they are not. The way to model a naive bayes classifier is by saying that each word is modelled as independent items belonging to a greater set of words $\mathcal{V}$ (as vocabulary). Each word can take a number of values corresponding at most with the size of the vocabulary. The naive classifier will be

$$\prod_{i=1}^{|\vec{x}|} P(w_i|y)P(y)$$

This is a multinomial distribution that is specific for every types of topic y (biology, machine learning, mathematics, physics...) so we can get all possible words for all possible class.

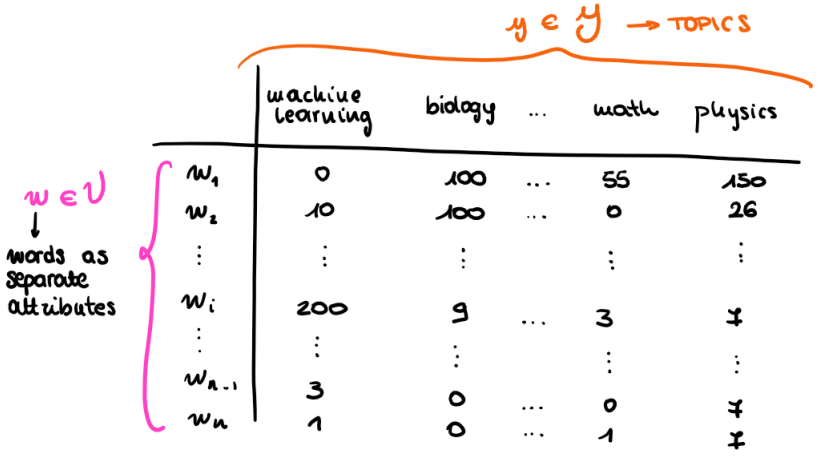


Figure 11.2: Example of recurrences of each word in the vocabulary for each class (topic) as independent variables

Each column is a multinomial distribution over size of $|\mathcal{V}|$ possible values. In this way the different elements in the product are modelled in the same distribution. Typically we have multiple times the same word in the document. The naive bayes assumption can not always be applied as we have seen so far since in some cases the dataset is made as separate attributes (height, weight, hair color...) and need to be modelled with different models (continuous/discrete values).

Chapter 12

Linear discriminant functions

We can distinguish learning models between *generative learning models* and *discriminative learning models*.

- **Generative learning** assumes knowledge of the distribution governing the data. In essence the purpose of generative models is to model the distribution governing the data. For example, Bayesian network is a generative model since it models the joint probability among the variables in the network. In this manner it is possible to make prediction about the variables in the network. Moreover, with a generative model we can generate new data (samples) according to the distribution.
- **Discriminative learning** focuses on directly modeling the discriminant function. For example, for classification, the aim is to directly modeling the decision boundaries rather than inferring them from the modelled data distributions. We use discriminative learning models when we know that the purpose of our application is always to predict the output given the input (e.g. supervised learning). In this case we model the relationship between input and output, without taking into account the underlying generative process. In the context of classification, this means modeling the boundaries directly rather than the data distribution.

These distinction is graphically illustrated in Figure 12.1. In the case of Gaussian classifiers, the aim is to learn a Gaussian probabilistic distribution for each class. At this point, the Gaussian distributions allow to model the relationship between X and Y, i.e. the joint probability $P(X,Y) = P(X|Y)P(Y)$, where $P(X|Y)$ are the Gaussian distributions and Y is the set of all possible classes. This kind of model represents the joint probability between input and output. On the other hand, discriminative functions track boundaries between the possible classes. Indeed, in order to make predictions of Y given X it is not necessary

to model the full joint probability. We can focus only on learning the boundaries. In essence, we learn a function (*disicriminative function*) $f : X \to Y$ which directly models $P(Y|X)$.

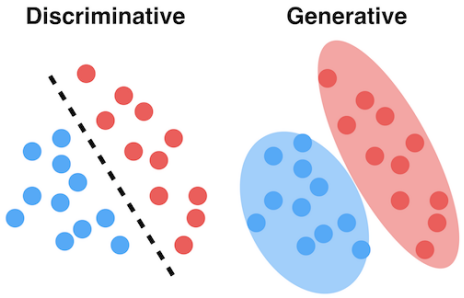


Figure 12.1: Discriminative *vs* Generative Models

12.1 Advantages and disadvantages of discriminative learning

In the following we discuss about some pros and cons related to the adoption of discriminative learning models.

Pros:

- When data are complex, modeling their underlying distribution can be very difficult (e.g. $P(X|Y)$ where X represents high resolution images)
- If data discrimination is the goal, data distribution modeling is not needed
- As a consequence we can focus only on learning (using the available training examples) the parameters which are interesting to understand the mapping between input and output

Cons:

- The learned model is less flexible in its usage. Indeed, discriminative models have fixed input and fixed output.
- It does not allow to perform arbitrary inference tasks. Really often in these scenarios there is not a precise distinction between inputs and outputs.
- It is not possible to efficiently generate new data from a certain class.

12.2 Linear discriminant functions

A linear discriminative model is a discriminative model which use a linear function to make predictions.

The *discriminant function* is a linear combination of example features ($\boldsymbol{x}$).

$$f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + w_0 \tag{12.1}$$

In the formula, w_0 is called *bias* or *threshold*. Equation 12.1 is the simplest possible discriminant function. Depending on the complexity of the task and amount of data, it can be the best option available (at least it is the first to try).

12.2.1 Linear binary classifier

In binary classification the predictied class is obtained taking the sign of the linear function.

$$f(\boldsymbol{x}) = sign(\boldsymbol{w}^T\boldsymbol{x} + w_0) \tag{12.2}$$

The decision boundary, which is a hyperplane (H), is achieved when $f(\boldsymbol{x}) = 0$.

Remark: the weight vector $\boldsymbol{w}$ is orthogonal to the decision hyperplane (Figure 12.2). **Proof:** we take two points on the decision boundary:

$$\forall \boldsymbol{x}, \boldsymbol{x}' : f(\boldsymbol{x}) = f(\boldsymbol{x}') = 0$$

We can replace $f(\boldsymbol{x})$ and $f(\boldsymbol{x}') = 0$ according to Equation 12.1:

$$\boldsymbol{w}^T\boldsymbol{x} + w_0 = \boldsymbol{w}^T\boldsymbol{x}' + w_0 = 0$$

$$\boldsymbol{w}^T\boldsymbol{x} + w_0 - \boldsymbol{w}^T\boldsymbol{x}' - w_0 = 0$$

$$\boldsymbol{w}^T(\boldsymbol{x} - \boldsymbol{x}') = 0$$

This means that vector $\boldsymbol{w}^T$ and vector $(\boldsymbol{x} - \boldsymbol{x}')$ are orthogonal.

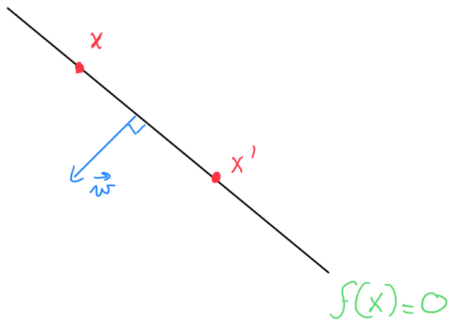


Figure 12.2: The weight vector $\boldsymbol{w}$ is orthogonal to the decision hyperplane.

Functional margin
The value $f(\boldsymbol{x})$ of the function for a certain point $\boldsymbol{x}$ is called *functional margin*. It can be seen as a confidence in the prediction.

In a sense, the functional margin represents the margin before predicting the opposite class. The closer $f(\boldsymbol{x})$ is to zero, the smaller the confidence of our prediction.

Geometric margin
The distance from $\boldsymbol{x}$ to the hyperplane is called *geometric margin*:

$$r^x = \frac{f(\boldsymbol{x})}{||\boldsymbol{w}||} \tag{12.3}$$

It is a normalized version of the functional margin.

Remark: the distance from the origin to the hyperplane is:

$$r^0 = \frac{f(\mathbf{0})}{||\boldsymbol{w}||} = \frac{w_0}{||\boldsymbol{w}||} \tag{12.4}$$

These concepts can be further clarified looking at Figure 12.3. Given an arbitrary point $\boldsymbol{x}$, it can be expressed by its projection on H ($\boldsymbol{x}^p$) plus its distance to H (r^x) times the unit vector in that direction ($\frac{\boldsymbol{w}}{||\boldsymbol{w}||}$):

$$\boldsymbol{x} = \boldsymbol{x}^p + r^x \frac{\boldsymbol{w}}{||\boldsymbol{w}||}$$

At this point we can replace this value of x into Equation 12.1:

$$\begin{aligned} f(\boldsymbol{x}) &= \boldsymbol{w}^T\boldsymbol{x} + w_0 = \\ &= \boldsymbol{w}^T(\boldsymbol{x}^p + r^x \frac{\boldsymbol{w}}{||\boldsymbol{w}||}) + w_0 = \\ &= \boldsymbol{w}^T\boldsymbol{x}^p + w_0 + r^x \boldsymbol{w}^T \frac{\boldsymbol{w}}{||\boldsymbol{w}||} \end{aligned}$$

We can notice that $\boldsymbol{w}^T\boldsymbol{x}^p + w_0 = f(\boldsymbol{x}^p)$. We know that $f(\boldsymbol{x}^p) = 0$ because $\boldsymbol{x}^p$ is on the decision boundary. Moreover:

$$\frac{\boldsymbol{w}^T\boldsymbol{w}}{||\boldsymbol{w}||} = \frac{\boldsymbol{w}^T\boldsymbol{w}}{\sqrt{\boldsymbol{w}^T\boldsymbol{w}}} = \sqrt{\boldsymbol{w}^T\boldsymbol{w}} = ||\boldsymbol{w}||$$

Given these, we can write:

$$f(\boldsymbol{x}) = r^x||\boldsymbol{w}||$$
$$\frac{f(\boldsymbol{x})}{||\boldsymbol{w}||} = r^x$$

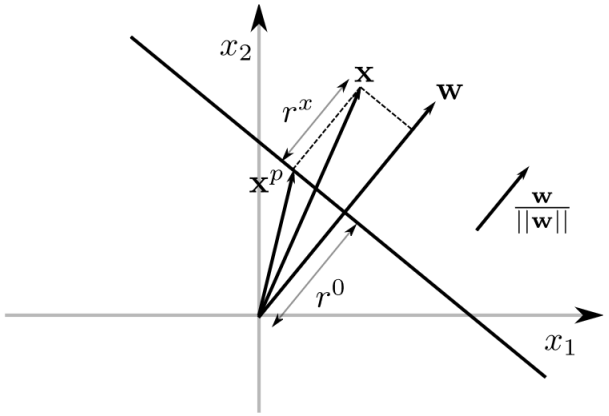


Figure 12.3: A point ($\mathbf{x}$) can be expressed by its projection on H plus its distance to H times the unit vector in that direction: $\boldsymbol{x} = \boldsymbol{x}^p + r^x \frac{\boldsymbol{w}}{||\boldsymbol{w}||}$

12.3 Perceptron

In order to introduce the perceptron, in the following we write about the biological motivation behind this concept.

12.3.1 Biological motivation

The human brain is composed of densely interconnected network of *neurons* (Figure 12.4). A neuron is made of:

- *soma*: a central body containing the nucleus
- *dendrites*: a set of filaments departing from the body
- *axon*: a longer filament (up to 100 times body diameter)
- *synapses*: connections between dendrites and axons from other neurons

Electrochemical reactions allow signals to propagate along neurons via axons, synapses and dendrites. Synapses can either **excite** or **inhibit** a neuron potential. Each neuron receives (and "accumulates") signals from other neurons. Once a neuron potential exceeds a certain **threshold**, a signal is generated and transmitted along the axon.

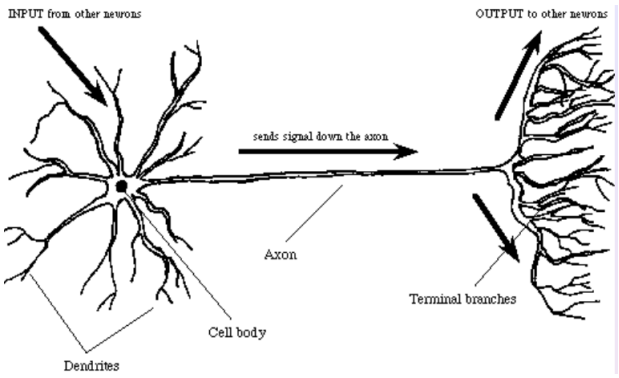


Figure 12.4: The human brain is composed of densely interconnected network of *neurons*.

12.3.2 Single neuron architecture

Mathematically, the neuron structure can be implemented by the *perceptron* (Figure 12.5).

$$f(x) = sign(\boldsymbol{w}^T \boldsymbol{x} + w_0) \tag{12.5}$$

The function is a linear combination of input features. The coefficients of the linear combination are the weights which can be seen as the synapses. Positive weights excite, negative weights inhibit. The weighted signals are summed up. The result of the summation is processed by an activation function to decide whether to fire or not. In this context "firing or not" means giving output 0 or 1.

Remark: we always append to the values $\{x^1 \dots x^m\}$ a bias $x^0 = 1$.

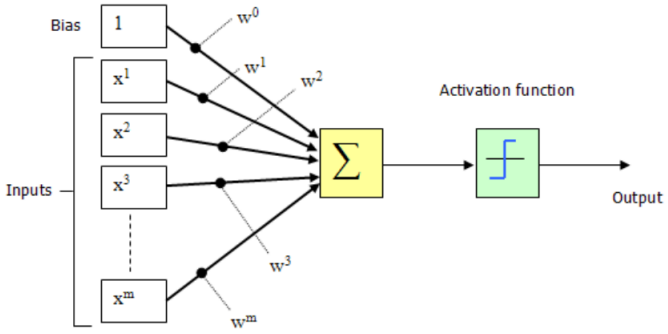


Figure 12.5: Single neuron architecture.

12.3.3 Representational power

A single linear classifier can represent *linearly separable* sets of examples (e.g. primitive boolean functions (AND, OR, NAND, NOT) - Figure 12.6 and Figure

12.7).

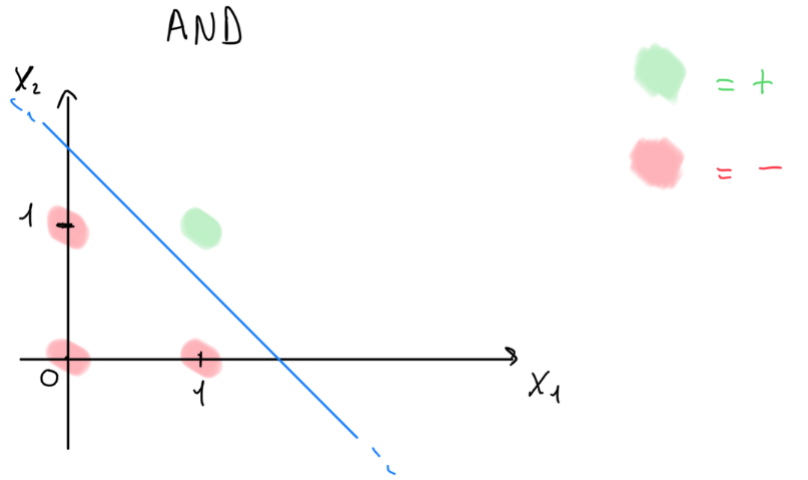


Figure 12.6: A single linear classifier can represent *linearly separable* sets of examples. For example the AND boolean function.

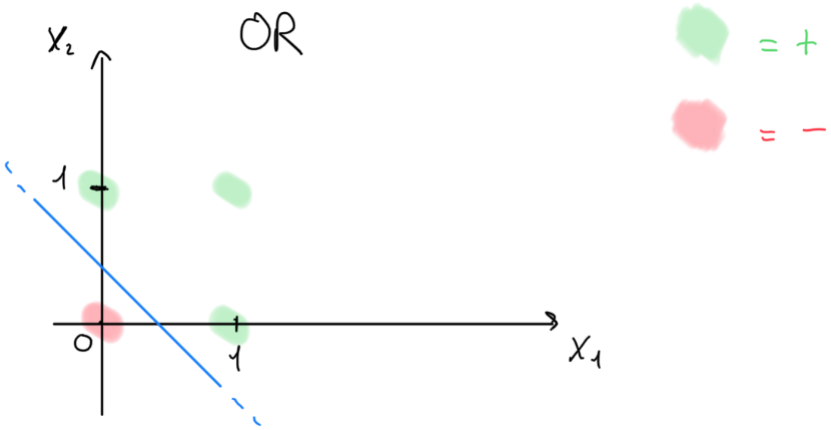


Figure 12.7: A single linear classifier can represent *linearly separable* sets of examples. For example the OR boolean function.

A linear classifier can not learn more complex boolean formula like the XOR (Figure 12.8).

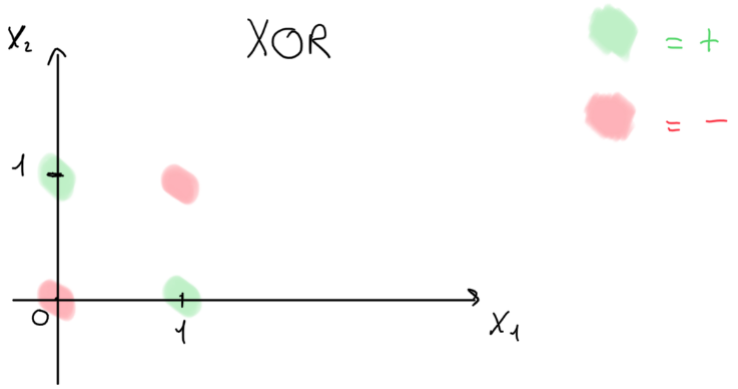


Figure 12.8: A linear classifier can not learn more complex boolean formula like the XOR.

However, the XOR can be represented in terms of AND and OR:

$$x_1 \oplus x_2 = (x_1 \wedge \bar{x_2}) \vee (\bar{x_1} \wedge x_2)$$

At this point $A = (x_1 \wedge \bar{x_2})$ and $B = (\bar{x_1} \wedge x_2)$ and $C = A \vee B$ can be represented by a neuron. In particular the input of neuron C are the output of the neurons A and B. This means that the XOR can be learnt by a network with two layers.

In general, any logic formula ϕ can be written in DNF (disjunctive normal form) or CNF (conjunctive normal form). This means that any logic formula can be represented and learnt by a network of two levels of perceptrons. Unfortunately, it is extremely expensive to represent complex logic formula with two levels of perceptron.

Problems

- *non-linearly separable* sets of examples cannot be separated
- representing complex logic formulas can require a number of perceptrons exponential in the size of the input. Having an exponential number of perceptron is definitely prohibitive for learning. This models are usually referred to as *shallow models*. The solution is to build deeper models with several perceptron layers.

It is useful to write down Equation 12.5 in a shorter way. To do this, *augmented/weight vectors* are used. The idea is to incorporate the bias in the augmented vector.

$$f(x) = sign(\hat{\boldsymbol{w}}^T \hat{\boldsymbol{x}}) \tag{12.6}$$

$$\hat{\boldsymbol{w}} = \begin{pmatrix} w_0 \\ \boldsymbol{w} \end{pmatrix} \tag{12.7}$$

$$\hat{\boldsymbol{x}} = \begin{pmatrix} 1 \\ \boldsymbol{x} \end{pmatrix} \tag{12.8}$$

For the sake of brevity, we skip the hat over $\boldsymbol{x}$ and $\boldsymbol{w}$ in the following, replacing the search for weight vector + bias with the search for the augmented weight vector.

12.4 Parameter learning

Similarly to what we have studied for maximum likelihood for probability distributions, we need to find a function of the parameters to be optimized. A reasonable function is the measure of the error on the training set $\mathcal{D}$. This function is called *loss function* and it compares the ground truth with the output of our model. In few words, $l(y, f(\boldsymbol{x}))$ represents the loss that we have to pay for predicting $f(\boldsymbol{x})$ instead of y. As a consequence, the loss function is something to be minimized.

$$E(\boldsymbol{w}; \mathcal{D}) = \sum_{(\boldsymbol{X}, y) \in \mathcal{D}} l(y, f((x)) \tag{12.9}$$

Finding the parameters to minimize the error function can easily lead to overfitting. However, keep in mind that linear classifier are simple models less prone to overfitting with respect to deep neural networks.

12.4.1 Gradient descent

The common approach to minimize the error function is the *gradient descent*. In the context of optimization problems, the interesting points are where the gradient of the function becomes zero. Unfortunately, once we have this equation, there is no closed formula to solve and get the parameters. In order to face this situation we rely on gradient descent approach in order to progressively find a good local minima of the error function.
Gradient descent works as follows:

1. initialize $\boldsymbol{w}$ (e.g. $\boldsymbol{w} = 0$) (randomly)
2. iterate until gradient is approximately zero:

$$\boldsymbol{w} = \boldsymbol{w} - \eta \nabla E(\boldsymbol{w}; \mathcal{D}) \tag{12.10}$$

Remark: the gradient points to the maximum of the function. As a result we have to explore the function in the opposite direction $-\eta \nabla E(\boldsymbol{w}; \mathcal{D})$.

- η is called *learning rate* and controls the amount of movement at each gradient step. In essence, the gradient gives us information about the *direction* while the learning rate quantifies the *size of the step* along the suggested direction.

- the algorithm is guaranteed to converge to a local optimum of $E(\boldsymbol{w}; D)$ (for small enough η)
- too low η implies slow convergence. On the other hand, with too high values of η we end up oscillating.
- techniques exist to adaptively modify η

In the case of the linear classifier, the misclassification loss is piecewise constant and so it is a poor candidate for gradient descent. Indeed, applying gradient descent with a function like the one represented in Figure 12.9 the gradient is either 0 or it does not exist. In order to apply gradient descent approach, we need functions with a smoother behaviour.

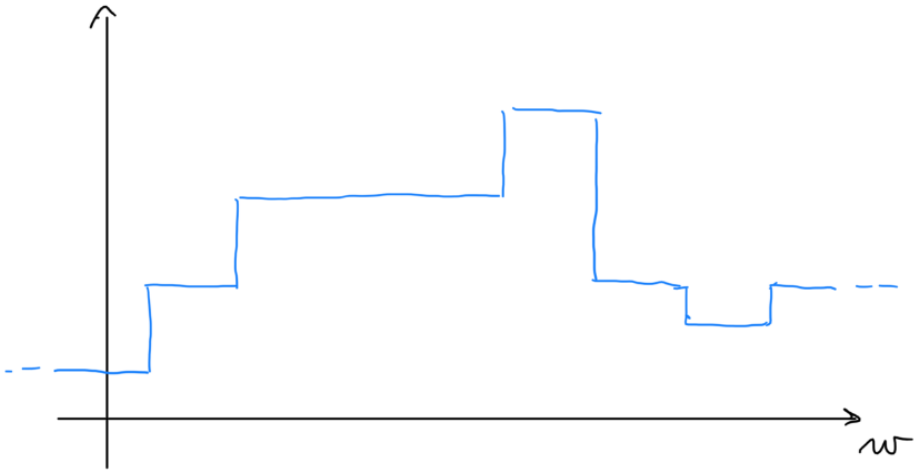


Figure 12.9: The misclassification loss is piecewise constant and so it is a poor candidate for gradient descent.

As a consequence we adopt a different training rule, the *perceptron training rule*. Following this approach, the error is the sum of the functional margins ($f(\boldsymbol{x})$) of incorrectly classified examples:

$$E(\boldsymbol{w}; \mathcal{D}) = \sum_{(\boldsymbol{X}, y) \in \mathcal{D}_E} -yf(\boldsymbol{x})s \tag{12.11}$$

where $\mathcal{D}_E$ is the set of current training errors for which:

$$yf(\boldsymbol{x}) \leq 0$$

Indeed, we do not consider the right predictions but we take into consideration only the wrong predictions.

With this approach we are taking into consideration the confidence of our predictions. For this reason, we typically refer to this kind of loss functions as *confidence loss*. We compare the confidence of the prediction with respect to the actual class.

Remark: We write $-yf(\boldsymbol{x})$ because we want to maximize the confidence of the correct predictions (or analogously minimize the incorrect predictions characterized by high confidence).

We apply gradient descent on Equation 12.11:

$$\begin{aligned} \nabla E(\boldsymbol{w}; \mathcal{D}) = \\ \nabla \sum_{(\boldsymbol{X},y)\in\mathcal{D}_E} -yf(\boldsymbol{x}) = \\ \nabla \sum_{(\boldsymbol{X},y)\in\mathcal{D}_E} -y(\boldsymbol{w}^T\boldsymbol{x}) \end{aligned}$$

The gradient of $\boldsymbol{w}^T\boldsymbol{x}$ with respect to $\boldsymbol{w}$ is just $\boldsymbol{x}$.

$$\begin{aligned} \nabla E(\boldsymbol{w}; \mathcal{D}) = \\ \nabla \sum_{(\boldsymbol{X},y)\in\mathcal{D}_E} -y(\boldsymbol{w}^T\boldsymbol{x}) = \\ \sum_{(\boldsymbol{X},y)\in\mathcal{D}_E} -y\boldsymbol{x} \end{aligned}$$

The amount of update is:

$$-\eta\nabla E(\boldsymbol{w}; \mathcal{D}) = \eta \sum_{(\boldsymbol{X},y)\in\mathcal{D}_e} y\boldsymbol{x}$$

This last is the update of $\boldsymbol{w}$ at each iteration.

12.4.2 Stochastic perceptron training rule

Following the approach we have presented so far, in order to update the vector of the weights, we have to iterate over all the training set at each iteration to compute the gradient of the error for each example. This results in slow convergence if we have many examples (e.g. deep learning). *Stochastic perceptron training rule* is an alternative to this model.

1. Initialize weights randomly
2. Iterate until all examples are correctly classified:
   (a) for each incorrectly classified training example ($\boldsymbol{x}$,y) update the weight vector:
   $$\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta y\boldsymbol{x}$$

Remark: the update formula $\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta y \boldsymbol{x}$ is the same as before but there is not the summation over all the training data. We update the weight vector considering only one example.

With this method, we make a gradient step for each training error, rather than on the sum of them in *batch* learning. In this way, each gradient step is very fast. Moreover, stochasticity can sometimes help to avoid local minima (i.e. it could be more robust), being guided by various gradients for each training example (which won't have the same local minima in general). At each iteration we compute the gradient on a different error function. Following this approach, we stochastically optimize the error function.

Remark: instead of considering one single example at each iteration, some implementations of stochastic gradient discent consider a *minibatch* of training data.

12.5 Perceptron regression

Linear models are not used only for linear classification but also for *linear regression*.

Let $X \in \mathbb{R}^n \times \mathbb{R}^d$ be the input training matrix (i.e. $X = \left[\boldsymbol{x}^1 \dots \boldsymbol{x}^n\right]^T$ for $n = |\mathcal{D}|$ and $d = |\boldsymbol{x}|$).
Let $\boldsymbol{y} \in \mathbb{R}^n$ be the output training matrix (i.e. y_i is output for example $\boldsymbol{x}^i$)

Regression learning could be stated as a set of linear equations:

$$X\boldsymbol{w} = \boldsymbol{y} \tag{12.12}$$

Giving as solution:

$$\boldsymbol{w} = X^{-1}\boldsymbol{y} \tag{12.13}$$

Unfortunately, this approach does not work:

- X is not square in general because not necessarily the number of examples is equal to the number of features. In general, matrix X is rectangular, usually more rows than columns
- System of equations is overdetermined (more equations than unknowns)
- No exact solution typically exists

So, even if the matrix is square, it is not typically invertible. However, our aim is not necessarily to solve the system of equation exactly. It is enough to do the best that we can. Again, we are going to minimize an error function. When

dealing with regression and loss minimization, the most common error function which is considered is the *mean squared error* (MSE).

$$E(\boldsymbol{w};\mathcal{D}) = \sum_{(\boldsymbol{X},y)\in\mathcal{D}} (y - f(\boldsymbol{x}))^2 = (\boldsymbol{y} - X\boldsymbol{w})^T(\boldsymbol{y} - X\boldsymbol{w}) \tag{12.14}$$

For this equation, there is a closed form solution. Moreover, it can also be always solved by gradient descent. This latter approach can be faster.

Remark: Mean squared error can also be used as a classification loss.

12.5.1 Closed form solution

In order to calculate a closed form solution to our problem, the idea is to compute the gradient and make it equal to zero.

$$\begin{aligned}
\nabla E(\boldsymbol{w};\mathcal{D}) = \\
\nabla(\boldsymbol{y} - X\boldsymbol{w})^T(\boldsymbol{y} - X\boldsymbol{w}) = \\
2(\boldsymbol{y} - X\boldsymbol{w})^T(-X) = 0 \\
-2\boldsymbol{y}^T X + 2\boldsymbol{w}^T X^T X = 0 \\
\boldsymbol{w}^T X^T X = \boldsymbol{y}^T X \\
(\boldsymbol{w}^T X^T X)^T = (\boldsymbol{y}^T X)^T \\
X^T X\boldsymbol{w} = X^T\boldsymbol{y} \\
\boldsymbol{w} = (X^T X)^{-1} X^T\boldsymbol{y}
\end{aligned}$$

Remark: $\boldsymbol{w}$ is a column vector.

$(X^T X)^{-1} X^T$ is a pseudo inverse which is called *left-inverse*.

Remark:

- The left-inverse exists provided $(X^T X) \in \mathbb{R}^{d\times d}$ is full rank, otherwise we can't invert it. If $(X^T X)$ is not full rank is because features are linearly dependent. In order to make it full rank, we just remove the redundant features. In this way, all the features are linearly independent and $(X^T X)$ is full rank.
- If X is square and nonsingular (i.e. invertible), inverse and left-inverse coincide ($(X^T X)^{-1} X^T = X^{-1}$) and the MSE solution corresponds to the exact one.

  $$\boldsymbol{w} = X^{-1}\boldsymbol{y}$$

  So, if there is an exact solution, MSE finds it, otherwise it will provide an approximation which minimizes the squared error.

In a context with a lot of features, inverting $(X^T X)$ could be very expensive. As a result, instead of solving the equation with the gradient equal to zero, as an alternative, we can compute the gradient and perform gradient descent. For a single weight, the gradient step is computed as illustrated:

$$\begin{aligned}\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{(\boldsymbol{x},y)\in\mathcal{D}}(y-f(\boldsymbol{x}))^2 \\ &= \frac{1}{2}\sum_{(\boldsymbol{x},y)\in\mathcal{D}}\frac{\partial}{\partial w_i}(y-f(\boldsymbol{x}))^2 \\ &= \frac{1}{2}\sum_{(\boldsymbol{x},y)\in\mathcal{D}}2(y-f(\boldsymbol{x}))\frac{\partial}{\partial w_i}\left(y-\boldsymbol{w}^T\boldsymbol{x}\right) \\ &= \sum_{(\boldsymbol{x},y)\in\mathcal{D}}(y-f(\boldsymbol{x}))\,(-x_i)\end{aligned}$$

12.6 Multiclass classification

All the methods illustrated in this section, solve *multiclass classification* by means of combination of binary classification tasks.

Remark: the distinction between multiclass classification and binary classification is necessary in the case of discriminant models. On the other hand, in the generative model domain (e.g. Naive Bayes), there is not a different solution between binary classification and multiclass classification.

12.6.1 ONE vs ALL

The idea of this multiclass classification solution is to learn one binary classifier for each class such that:

- positive examples are examples of the class
- negative examples are examples of all other classes

At this point we have a model with m decision hyperplanes, one per class. Each classifier provides a confidence (score), and we take the class with maximal score. Figuratively, it is as if each classifier votes a class. More formally, the model predicts a new example in the class with maximum functional margin.

Decision boundaries for which $f_i(\boldsymbol{x}) = f_j(\boldsymbol{x})$ are pieces of hyperplanes such that the confidence for class i is equal to the confidence for class j. In these configurations, we do not know which of the two classes to choose.

$$\boldsymbol{w}_i^T\boldsymbol{x} = \boldsymbol{w}_j^T\boldsymbol{x}$$

$$(\boldsymbol{w}_i - \boldsymbol{w}_j)^T\boldsymbol{x} = 0$$

From this last equation we observe that the boundary is actually a hyperplane (Figure 12.10).

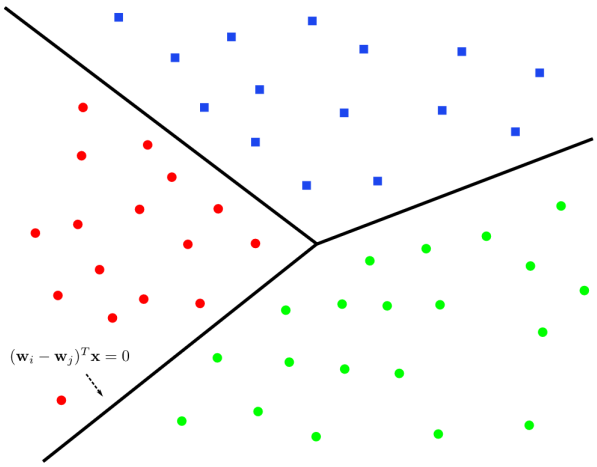


Figure 12.10: Multiclass classification decision boundaries.

12.6.2 ONE vs ONE (all pairs)

The aim of this multiclass classification solution is to learn one binary classifier for each pair of classes:

- positive examples from one class
- negative examples from the other

If there are m classes, there will be $\frac{m(m-1)}{2}$ binary classifier. The model predicts a new example in the class winning the largest number of pairwise classifications as in a tournament.

12.6.3 ONE vs ONE - ONE vs ALL comparison

In ONE vs ALL we have to train m classifiers each with all examples. On the other hand, in ONE vs ONE we have to train $\mathcal{O}(m^2)$ classifiers, each with much fewer examples (only the examples of the two classes). If the training procedure complexity is high with respect to the number if examples, all pairs is faster.

12.7 Generative linear classifiers

What kind of generative models produce linear classifiers? At the beginning of this section, we write about *Gaussian classifier*: we have a Gaussian distribution $P(\boldsymbol{x}|y)$ for each class. The task is to compute the posterior $P(y|\boldsymbol{x})$. In the case of Gaussian distributions, linear decision boundaries are obtained when

covariance is shared among classes ($\Sigma_i = \Sigma$). This defines a general spread in the space which does not depend on the class. So, only under this assumption the Gaussian classifier becomes a linear classifier.

Remark: Gaussian has the exponential in its formulation, so we typically adopt log-linear Gaussian classifiers. Log-linear classifiers are characterized by the same expressive capability with respect to linear classifiers since the logarithm is a monotonic function.

Another generative model which is also linear is Naive Bayes classifier. In this case, it is required to assume that all features are independent one with respect to the other given the class.

$$\begin{aligned} f_i(\boldsymbol{x}) = P(\boldsymbol{x}|y_i)P(y_i) = \\ = \prod_{j=1}^{|\boldsymbol{x}|} P(x_j|y_i)P(y_i) = \\ = \prod_{j=1}^{|\boldsymbol{x}|} \prod_{k=1}^{K} \theta_{ky_i}^{z_k(x[j])} \frac{|\mathcal{D}_i|}{|\mathcal{D}|} = \\ = \prod_{k=1}^{K} \theta_{ky_i}^{N_{k\boldsymbol{x}}} \frac{|\mathcal{D}_i|}{|\mathcal{D}|} \end{aligned}$$

where $N_{k\boldsymbol{x}}$ is the number of times feature k (e.g. a word) appears in $\boldsymbol{x}$.

Remark:

- $\prod_{j=1}^{|\boldsymbol{x}|}$ product over all the possible features
- $\prod_{k=1}^{K}$: x_j is a discrete variable with k possible values
- $z_k(x[j])$ is a hot encoding of a categorical variable (a categorical variable is a variable which can be represented as a hot encoding). $z_k(x[j]) = 1$ if the variable takes the k value and zero otherwise
- θ_{ky_i} parameter of the kth value for y_i class. It is the probability that feature k of x_j is true given y_i. In essence θ_{ky_i} is raised to the power of 1 when x_j has the kth feature, otherwise θ_{ky_i} is raised to the power of 0. (Figure 12.11)
- $\frac{|\mathcal{D}_i|}{|\mathcal{D}|}$ is equal to $P(y_i)$

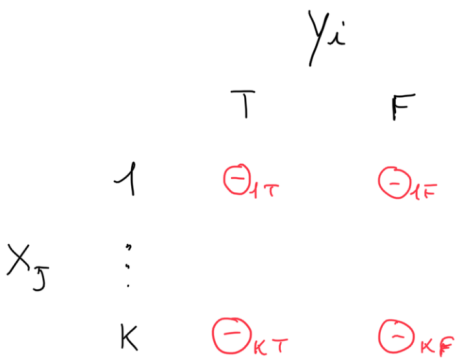


Figure 12.11: Parameter of the kth value for y_i class. In this figure we assume y_i is a binary class.

In the formula there are too many expensive exponents. As result we take the log:

$$\log f_i(\boldsymbol{x}) = \sum_{k=1}^{K} N_{k\boldsymbol{x}} \log \theta_{ky_i} + \log\left(\frac{|\mathcal{D}_i|}{\mathcal{D}}\right)$$

Note that the term $\log\left(\frac{|\mathcal{D}_i|}{\mathcal{D}}\right)$ is a scalar which does not depend on x. We can replace it with w_0 (bias). Moreover, we can define:

- $\boldsymbol{x}' = \left[N_{1\boldsymbol{x}} \dots N_{k\boldsymbol{x}}\right]^T$: vector of the number of times I have seen each feature (e.g. word in a document).
- $\boldsymbol{w} = \left[\log \theta_{1y_i} \dots \log \theta_{ky_i}\right]^T$

So, we can rewrite the above formula as:

$$\log f_i(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x}' + w_0$$

Naive Bayes is a *log-linear* model (as Gaussian distributions with shared Σ).

Chapter 13

Support vector machines

Support Vector Machines (SVM) are a very popular method for linear (and non-linear) classification. They have some properties:

- they are **large margin classifiers**: the separating hyperplane is setted withing the largest margin possible between classes
- the **support vector** are the training examples (also only a small subset), the decision boundary is identified on few support vectors
- the **large margin** properties are formally showed to imply some generalization properties
- can easily be extended to non-linear separation (also with *kernel machines* 15)

13.1 Hyperplanes

All the hyperplanes that can separate the two subsets in Figure 13.1 classify correctly every example. If I use a perceptron I could end up with every of those planes, depending on the starting position.

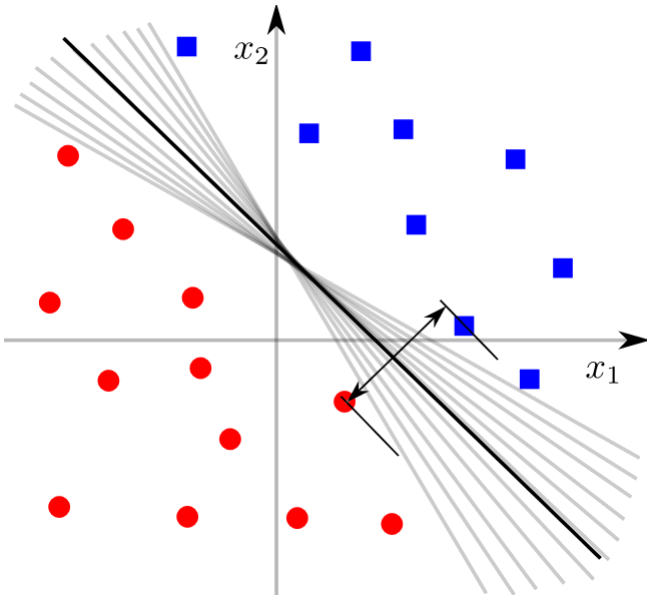


Figure 13.1: Possible hyperplanes separating two subsets

SVMs learns the central hyperplane, the one that has the greater *margin*. The margin is the distance of the closest point to the hyperplane. SVM tries to maximize this distance from both classes. This is intuitively a good choice: leaves the most space between samples.

Classifier margin
Given a training set $\mathcal{D}$, a classifier confidence margin is

$$\rho = min_{(\boldsymbol{x},y)\in\mathcal{D}} yf(\boldsymbol{x})$$

This is the minimal confidence of the classifier in a correct prediction (which corresponds to $yf(\boldsymbol{x})$ as for perceptrons) and has to be positive in order to correctly separate the classes.
The *geometric margin* is the same value divided by the norm of $\boldsymbol{w}$:

$$\frac{\rho}{||\boldsymbol{w}||} = min_{(\boldsymbol{x},y)\in\mathcal{D}} \frac{yf(\boldsymbol{x})}{||\boldsymbol{w}||}$$

13.2 Hard margin SVMs

We want intuitively learn a classifier that learns correct prediction with a large margin. We need to fix degrees of freedom of hyperplanes in the space.

Canonical hyperplane
There is a infinite number of equivalent hyperplanes that encode for the same hyperplane:

$$\alpha(\boldsymbol{w}^T\boldsymbol{x} + w_0) = 0$$

for any $\alpha \neq 0$. The *canonical hyperplane* is the hyperplane having the confidence (classifier margin 13.1) equal to 1

$$\rho = min_{(\boldsymbol{x},y)\in\mathcal{D}} yf(\boldsymbol{x}) = 1$$

and its geometric margin is

$$\frac{\rho}{||\boldsymbol{w}||} = \frac{1}{||\boldsymbol{w}||}$$

We basically want to remove the degree of freedom because it would lead us to compute for the same hyperplane multiple times. Therefore we will set $\rho = 1$ in order to use only the canonical hyperplane. We can always do it until ρ it is not negative and provided that the samples are linearly separable.

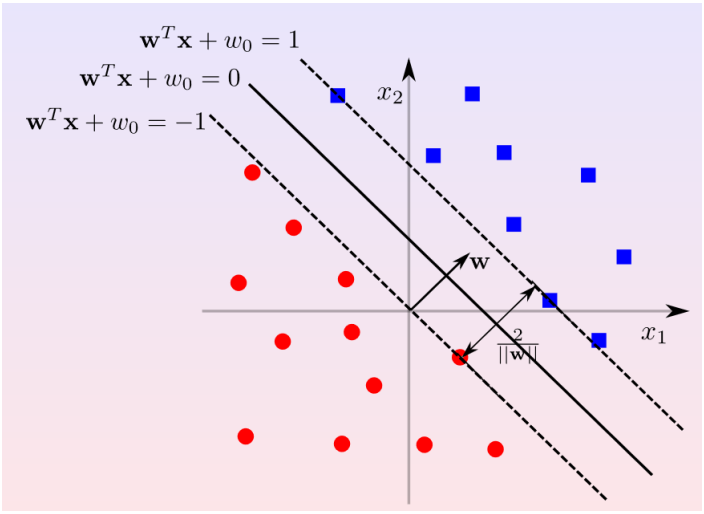


Figure 13.2: Canonical hyperplane for each subsets

As Figure 13.2 shows, the dashed lines are the hyperplanes for which the confidence margins are smallest and equal to 1 (and -1). They are the hyperplanes with confidence 1 for class red (labelled as -1) and class blue (as +1). Overall it always mean that $yf(x) = 1$ (taking into account the labels of the classes). The geometric margin becomes $\frac{2}{||w||}$.

Margin error bound (theorem)
Hard margin SVM Theorem (Margin Error Bound) Consider the set of decision functions $f(\mathbf{x}) = \text{sign}^T \mathbf{x}$ with $\|\mathbf{w}\| \leq \Lambda$ and $\|\mathbf{x}\| \leq R$, for some $R, \Lambda > 0$. Moreover, let $\rho > 0$ and ν denote the fraction of training examples with margin smaller than $\rho/\|\mathbf{w}\|$, referred to as the margin error. For all distributions P generating the data, with probability at least $1 - \delta$ over the drawing of the m training patterns, and for any $\rho > 0$ and $\delta \in (0, 1)$, the probability that a test pattern drawn from P will be misclassified is bound from above by

$$\nu + \sqrt{\frac{c}{m}\left(\frac{R^2\Lambda^2}{\rho^2}\ln^2 m + \ln(1/\delta)\right)}.$$

Here, c is a universal constant. Support Vector Machine

(you are not supposed to know the theorem by heart).
Basically, this theorem shows a bound on the generalization error of a classifier which is trained to be a hard margin SVM maximized. It happens that the generalization error (expected error) for unseen example is a combination of:

1. **sum of margins errors** ν that stands for the component of all margin errors (training errors or examples with not enough confidence)

2. **the second term** depends on number of training examples m (the greater is m, the smaller is this factor) and ρ^2 which has to do with the margin (the larger the margin, the smaller the error)

Remark: if ρ is fixed to 1 (canonical hyperplane), maximizing margin corresponds to minimizing $||\boldsymbol{w}||$.

13.2.1 Learning problem

The learning objective of SVMs is indeed:

$$min_{\boldsymbol{w},w_0}\frac{1}{2}||\boldsymbol{w}||^2 \tag{13.1}$$

since when ρ is fixed to 1 (canonical hyperplane), maximizing margin corresponds to minimizing $||\boldsymbol{w}||$. This is subject to

$$y_i(\boldsymbol{w}^T\boldsymbol{x}_i) + w_0 \geq 1$$

for all $(\boldsymbol{x}_i, y_i) \in \mathcal{D}$. The hard margin is called this way because it require all the example to have a confidence at least equal to 1. Hard because it is not always possible. It there is no hyperplane that has this property, the problem has no solution. Hard stands for the hard constraint, difficult to obtain. This problem is a quadratic optimization problem (in w), with quadratic objective

so overall it is a quadratic optimization problem (reasonably efficient with no local optima, but only one overall optima).
We could solve it for standard tool for quadratic optimization, since it is a convex problem, it is guaranteed that when we find a solution, is globally optimal.

13.2.1.1 Techniques for solving the problem

Karush-Kuhn-Tucker approach (KKT)
A constrained optimization problem (of which I can minimize the objective but not guaranteeing that it satisfy the constraints) can be addresses by converting it into an *unconstrained* problem with the same solution. Let's have a general constrainted optimization problem with objective

$$min_z f(x)$$

subject to (some constraint, in this case non-negativity constraints)

$$g_i(z) \geq 0 \quad \forall i$$

Let's introduce a non-negative variable $\alpha_i \geq 0$ (called *Lagrange multiplier*) for each constraint and rewrite the optimization problem as Lagrangian as putting constraint inside the objective in this way:

$$min_z max_{\alpha \geq 0} f(z) - \sum_i \alpha_i g_i(x)$$

where i are the possible constraints and α_i varies depending on the constraint. Basically we remove the constraint by adding it into the objective. This problem has z parameter that we want to minimize and αs. We want to minimize z and maximize with respect to α which has to be non-negative.
The optimal solution(s) x^* for this problem are also optimal solution for the original constrained problem. x^* will minimize the original problem and satisfy the constraints.

Suppose that we find x', let's check whether it is an optimal solution if it does **not** satisfy the constraints. If there is at least one constraint that x' does not satisfy, then there is some i for which $g_i(z) < 0$ (it is negative). If g_i is negative, therefore the summation becomes positive and I can set α to an infinite value. This leads to a non-valid solution for the original problem.

Suppose now that x' does satisfy all the constraints. Then the equivalent problem formalized by KKT results in a finite number and therefore we know for sure that the constraints are satisfied.
If we set all αs to zero, then the summation goes to zero, therefore we have all constraint satisfied and also z' will be a solution of $min_z f(z)$ (which is what we were looking for at the beginning).

Applying the KKT approach to SVMs, then we can formalize out learning objective as:

$$min_{\boldsymbol{w},w_0}\frac{1}{2}||\boldsymbol{w}||^2$$

subject to:

$$y_i(\boldsymbol{w}^T\boldsymbol{x}_i + w_0 \geq 1)$$
$$\forall(\boldsymbol{x}_i, y_i) \in \mathcal{D}$$

We now apply KKT and bring 1 to the other side.

$$L(\boldsymbol{w}, w_0, \alpha) = \frac{1}{2}||\boldsymbol{w}||^2 - \sum_{i=1}^{m}\alpha_i(y_i(\boldsymbol{w}^T\boldsymbol{x}_i + w_0) - 1) \tag{13.2}$$

with L as *Lagrangian* equivalent problem.
Again, L gets minimized with respect to $\boldsymbol{w}$, w_0 and maximized from α_i (then we get the solution for a saddle point).

We now try to solve the Lagrangian equivalent form. Let's take the gradient with respect to $\boldsymbol{w}, w_0$ and set it to zero.

$$\begin{aligned}
L &= \frac{1}{2}||\boldsymbol{w}||^2 - \sum_{i=1}^{m}\alpha_i(y_i(\boldsymbol{w}^T\boldsymbol{x}_i + w_0) - 1) \\
\nabla_{\boldsymbol{w}}L &= \nabla_{\boldsymbol{w}}\left(\frac{\boldsymbol{w}^T\boldsymbol{w}}{2}\right) - \nabla_{\boldsymbol{w}}\sum_{i=1}^{m}\alpha_i y_i \boldsymbol{w}^T\boldsymbol{x}_i \\
&= \frac{2\boldsymbol{w}}{2} - \sum_{i=1}^{m}\alpha_i y_i \boldsymbol{x}_i \\
&= \boldsymbol{w} - \sum_{i=1}^{m}\alpha_i y_i \boldsymbol{x}_i = 0 \\
\boldsymbol{w} &= \sum_{i=1}^{m}\alpha_i y_i \boldsymbol{x}_i
\end{aligned}$$

Remark:

$$\nabla_{\boldsymbol{w}}\sum_{i=1}^{m}\alpha_i(y_i(\boldsymbol{w}^T\boldsymbol{x}_i + w_0) - 1) = \nabla_{\boldsymbol{w}}\sum_{i=1}^{m}\alpha_i y_i \boldsymbol{w}^T\boldsymbol{x}_i$$

because $\alpha_i y_i \boldsymbol{w}^T\boldsymbol{x}_i$ is the only term which depends on $\boldsymbol{w}$.

Now we get

$$\boldsymbol{w} = \sum_{i=1}^{m}\alpha_i y_i \boldsymbol{x}_i \tag{13.3}$$

defined in terms of alphas, but this is a way to write primal variables in terms of secondary (dual) variables (the αs). Let's also take the derivative of the Lagrangian formulation with respect to w_0

$$L = \frac{1}{2}||\boldsymbol{w}||^2 - \sum_{i=1}^{m} \alpha_i (y_i(\boldsymbol{w}^T \boldsymbol{x}_i + w_0) - 1)$$
$$\frac{\partial L}{\partial w_0} = \frac{\partial(-\sum_{i=1}^{m} \alpha_i y_i w_0)}{\partial w_0}$$

Remark: also in this case we take into account only the terms where w_0 appears.

Since it is linear, it becomes

$$\sum_{i=1}^{m} \alpha_i y_i = 0 \tag{13.4}$$

What we can do, is replacing $\boldsymbol{w}$ with the formulation in terms of alphas ($w = \sum_{i=1}^{m} \alpha_i y_i \boldsymbol{x}_i$), getting

$$L = \frac{\boldsymbol{w}^T \boldsymbol{w}}{2} - \sum_{i=1}^{m} \alpha_i (y_i(\boldsymbol{w}^T \boldsymbol{x}_i + w_0) - 1)$$
$$= \frac{1}{2} \left(\sum_{i=1}^{m} \alpha_i y_i \boldsymbol{x}_i \right)^T \sum_{j=1}^{m} \alpha_j y_j \boldsymbol{x}_j - \sum_{i=1}^{m} \alpha_i y_i (\sum_{j=1}^{m} \alpha_j y_j \boldsymbol{x}_j)^T \boldsymbol{x}_i - \sum_{i=1}^{m} \alpha_i y_i w_0 + \sum_{i=1}^{m} \alpha_i$$

Considering the first term, α, y are scalars (their transpose is still a scalar) only $\boldsymbol{x_i}, \boldsymbol{x_j}$ are vectors. We compute the product excluding the scalar part, therefore

$$L = \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j x_i^T x_j \dots$$

But the second term $-\sum_{i=1}^{m} \alpha_i y_i (\sum_{i=j}^{m} \alpha_j y_j \boldsymbol{x}_j)^T \boldsymbol{x}_i$ it is actually twice the quantity described by the first term. In the third term, we get that w_0 does not depend on i, we can take it outside. At this point we get:

$$L = \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \boldsymbol{x_i}^T \boldsymbol{x_j} - \sum_i \sum_j \alpha_i \alpha_j y_i y_j \boldsymbol{x_i}^T \boldsymbol{x_j} - w_0 \sum_{i=1}^{m} \alpha_i y_i + \sum_{i=1}^{m} \alpha_i$$

Since we defined $\boldsymbol{w} = \sum_{i=1}^{m} \alpha_i y_i \boldsymbol{x}_i$ (13.3), and also $\sum_{i=1}^{m} \alpha_i y_i = 0$ 13.4, therefore our objective becomes:

$$L(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \boldsymbol{x_i}^T \boldsymbol{x_j}$$

Which is the final formulation for the Lagrangian. This has to be minimized with respect to $\boldsymbol{w}, w_0$ and maximized with respect to α. We did get rid of the minimization, we are now going to maximize with respect to α.

Our objective becomes:

$$max_{\alpha \in \mathbb{R}^m} \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \boldsymbol{x_i}^T \boldsymbol{x_j}$$

and our constraints:

$$\alpha_i \geq 0 \quad i = 1, \ldots, m$$

$$\sum_{i=1}^{m} \alpha_i y_i = 0$$

This is a dual formulation (αs are in this case the dual variables, because they did not appear in the original problem but introduced in the Lagrangian), while the original formulation was only in terms of the primal variables. So we can formulate the constrained optimization problem in two ways (as primal or dual), there forms are mediated via the Lagrangian. This is still a quadratic optimization problem, the constraints are simpler then in the primal problem: we had linear constraints which were not easy to keep updated. In the dual problem we have easier constraint (non-negativity and bounding box constraint). The dual formulation is easier to deal with. Overall there are many ways in which we can solve it. Depending on the number of feature and number of samples (alphas depend on m number of examples, and $\boldsymbol{w}$ depends on number of features d) we can address this problem more easily from the primal or from the dual.

We can write $f(\boldsymbol{x})$ (the decision function) in term of the primal or the dual (remember 13.3)

$$f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} + w_0 = \sum_{i=1}^{m} \alpha_i y_i \boldsymbol{x}_i^T \boldsymbol{x} + w_0$$

Why do we call this method *support vector*?
By the KKT conditions, we get a Lagrangian and the Lagrangian multipliers (αs). The Lagrangian 13.2 contains a combination that stand for the margin (the inverse of the margin) and the constraints with Lagrangian multipliers. This is a min-max problem (minimize wrt $\boldsymbol{w}$ and maximize wrt αs), the results is a *saddle point*. What we know is that at this point (optimum), each of the conditions (constraint) should be equal to zero, therefore each element in the summation should be zero. This happens in two ways:

1. α_i are equal to zero, which means that the example x_i does not contribute to the final solution.

2. $y_i(\boldsymbol{w}^T\boldsymbol{x}_i+w_0)-1=0$ (the other component of the product is equal to zero), which means that $y_i(\boldsymbol{w}^T\boldsymbol{x}_i+w_0)=1$. This means that the confidence for the examples should be one. There are two hyperplanes with confidence equal to one (positive and negative 13.2) but both satisfy this condition (**positive examples** with confidence $+1$ and **negative examples** with confidence -1). These are the only points (with exact confidence $=+1$ or $=-1$) with $\alpha_i>0$ and these are the **support vectors** (SV).

My decision hyperplane $f(x)=0$ is defined only in terms of the support vectors. All other points do not contribute in defining the decision function. This mean that if I remove every other example from the training set (leaving only the support vectors) I would get the very same classifier. SVMs are *sparse*, this means that only few of the training example will end up in SV. In many cases the number of SV become much less than the training examples.

If we look at $f(\boldsymbol{x})$ (which is the decision on x), is basically taken as a linear combination of dot products between training data and $\boldsymbol{x}$ this value is high if the two values are similar. If $\boldsymbol{x_i}$ is similar to $\boldsymbol{x}$, the high value will be multiplied by y_i (class of $\boldsymbol{x_i}$) and α_i. Basically each train example pulls towards its own class based on how similar the new example is to him and proportionally to α_i (as weight). This prediction is therefore performed on weighted similarity.

We previously calculated $\boldsymbol{w}$ but we still need to compute w_0. I can use the condition $y_i(\boldsymbol{w}^T\boldsymbol{x}_i+w_0)=1$ considering cases in which $\alpha_i>0$. Therefore the bias w_0 is:

$$\begin{aligned} y_i(\boldsymbol{w}^T\boldsymbol{x}_i+w_0)&=1\\ y_i\boldsymbol{w}^T\boldsymbol{x}_i+y_iw_0&=1\\ w_0&=\frac{1-y_i\boldsymbol{w}^T\boldsymbol{x}_i}{y_i}\end{aligned}$$

For each of the samples, we get a different w_0, therefore, out final value will be their average.

13.3 Soft margin SVMs

There is one big problem with hard margin SVMs.

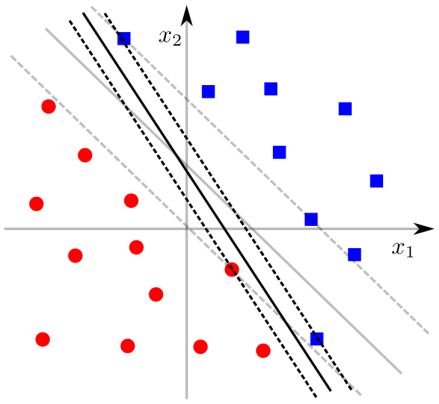


Figure 13.3: By adding an additional blue point, the previous hyperplane is no longer a valid solution, by computing it again we get the black new hyperplane

By adding an additional blue point as shown in Figure 13.3, the previous hyperplane is no longer a valid solution, by computing it again we get the black new hyperplane which has a significantly smaller confidence and it is tilted. Is it worth it? We should not assume that all labels are correct, we are maybe **overfitting** training examples. A solution is using **soft margin** SVMs.

We want to maximize the margin but with a soft constraint: some exceptions of falsely classified samples are allowed. This intuition can be formalized as minimizing the inverse of the margin subject to all examples being predicted in the correct class with a confidence at least one, **but** we add a slack variable ξ_i for each sample in the dataset, then we will sum them up according to a multiplicative factor C which is a hyperparameter.

$$min_{\boldsymbol{w}\in\mathcal{X},w_0\in\mathbb{R},\xi\in\mathbb{R}^m} \frac{||\boldsymbol{w}||^2}{2} + C\sum_{i=1}^{m}\xi_i$$

subject to:

$$y_i(\boldsymbol{w}^T\boldsymbol{x}_i + w_0) \geq 1 - \xi_i \quad i = 1,\ldots,m$$
$$\xi_i \geq 0 \quad i = 1,\ldots,m$$

$-\xi_i$ the slack is measuring how far we are from satisfying the constraint:

- if it is equal to zero we are correctly classifying the example
- if it is greater than zero we are miss-classifying that example
- the larger is $\sum_{i=1}^{m}\xi_i$, the larger is the number of miss-classified examples

By collecting all the slack variables $C\sum_{i=1}^{m}\xi_i$ (sum of penalties) we are trying to combine margin maximization and penalty minimization. The summation

is weighted based on C a regularization parameter constant greater than zero, that gives a trade off between maximizing the margin and fitting the examples. If $C = \infty$ we will have to correctly predict every example (we are back to hard margins SVMs). The smaller the C, the more exception we allow.

The idea is that in this objective we are combining a large margin and having few exception to the rule of large margin separation. Of course they are conflicting objectives. There is a more general theory under this concept: **regularization theory**. In regularization theory we have objectives to be learned that combine a **complexity term** (in this case the norm of the weight vectors, the margin but in general is a complexity of the solution) and **training errors** (in term of a loss function, measure of the error is the case of SVMs).

$$min_{\boldsymbol{w}\in\mathcal{X},w_0\in\mathbb{R},\xi\in\mathbb{R}^m} \frac{||\boldsymbol{w}||^2}{2} + C\sum_{i=1}^{m} loss(y_i, f(\boldsymbol{x}_i))$$

In soft margin SVMs, then we need to specify what is the loss function $loss(y_i, f(\boldsymbol{x}_i))$. Basically ξ_i is the loss of false predictions

$$\xi_i = loss(y_i, f(\boldsymbol{x}_i))$$

considering the constraint

$$y_i(\boldsymbol{w}^T\boldsymbol{x}_j + w_0) \geq 1 - \xi_i$$
$$y_i f(\boldsymbol{x}_i) \geq 1 - \xi_i$$

therefore

$$\xi_i \geq 1 - y_i(\boldsymbol{w}^T\boldsymbol{x}_i + w_0)$$

remember also that ξ is non-negative, so if we combine these things basically we get that

$$loss(y_i, f(\boldsymbol{x}_i)) = |1 - y_i f(\boldsymbol{x}_i)|_+ = |1 - y_i(\boldsymbol{w}^T\boldsymbol{x}_j + w_0)|_+ \tag{13.5}$$

where $|\ldots|_+$ stands for the value itself if it is positive and zero otherwise. If we look at this loss function graphically we can plot it in terms of $yf(\boldsymbol{x})$

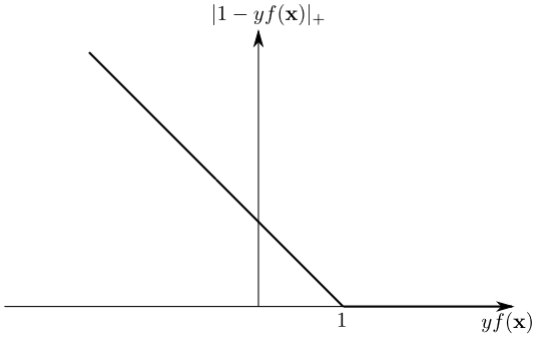


Figure 13.4: Plotting loss function (hinge loss)

the loss function is zero if the confidence in the correct prediction is at least one. If the confidence is less than one, then it is $|1 - y_i f(\boldsymbol{x}_i)|_+$.
This loss function is called **hinge loss function** since it is not a smooth loss function. It also shows why SVMs are sparse: the loss function is zero in a part of a space helps having a sparse solution (means that you don't care about differences, so the complexity term will dominate). Still, this loss function can be minimized.

13.3.1 Lagrangian for soft margin SVMs

Let's compute the same calculation described before also for soft margin SVMs. Our KKT formulation will be

$$L = \frac{1}{2}||\boldsymbol{w}||^2 + C\sum_{i=0}^{m}\xi_i - \sum_{i=1}^{m}\alpha_i(y_i(\boldsymbol{w}^T\boldsymbol{x}_i + w_0) - 1 + \xi_i) - \sum_{i=0}^{m}\beta_i\xi_i$$

where $\sum_{i=0}^{m}\beta_i\xi_i$ are the Lagrangian multipliers for the constraints $\xi_i \geq 0$. In this case, the primal variable are again $\boldsymbol{w}, w_0$, while the dual are α, β. Let's now compute the gradient of the Lagrangian

$$\begin{aligned}
\nabla_{\boldsymbol{w}} L &= \nabla_{\boldsymbol{w}} \frac{1}{2}||\boldsymbol{w}||^2 + C\sum_{i=0}^{m}\xi_i - \sum_{i=1}^{m}\alpha_i(y_i(\boldsymbol{w}^T\boldsymbol{x}_i + w_0) - 1 + \xi_i) - \sum_{i=0}^{m}\beta_i\xi_i \\
&= \nabla_{\boldsymbol{w}}\left(\frac{\boldsymbol{w}^T\boldsymbol{w}}{2} - \sum_{i=1}^{m}\alpha_i y_i \boldsymbol{w}^T\boldsymbol{x}_i\right) \\
&= \boldsymbol{w} - \sum_{i=1}^{m}\alpha_i y_i x_i
\end{aligned}$$

therefore it does not change from hard margin

$$\boldsymbol{w} = \sum_{i=1}^{m}\alpha_i y_i x_i \tag{13.6}$$

Then, we compute the derivative wrt w_0, but again does not change, since there is only one term containing w_0, then zeroing it we get.

$$\sum_{i=1}^{m}\alpha_i y_i = 0 \tag{13.7}$$

Then, we need to compute also for slack variables. Here we compute gradients wrt each of the ξ_i

$$\frac{\partial L}{\partial \xi_i} = \frac{\partial}{\partial \xi_i}(C\xi_i - \alpha_i\xi_i - \beta_i\xi_i)$$

By zeroing we get

$$c - \alpha_i - \beta_i = 0 \tag{13.8}$$

If we replace this result 13.8 in the Lagrangian (missing calculations), for $\boldsymbol{w}$ we get simplified results:

$$L(\alpha) = -\frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j + \sum \alpha_i$$

that fully correspond exactly to the hard margin case. Therefore the dual formulation is:

$$max_{\alpha_{i=1}^m \in \mathbb{R}^m} \sum_{i=1}^m \alpha_i - \frac{1}{2}\sum_{i,j=1}^m \alpha_i \alpha_j y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j \tag{13.9}$$

subject to

$$0 \leq \alpha_i \leq C \quad i = 1, \ldots, m$$

$$\sum_{i=1}^m \alpha_i y_i = 0$$

In this soft constraint, α_i needs to be non-negative, but also at most equal C (from 13.8). This box constraint is the only difference from the hard case when we transform the problem in the dual form.

The interesting fact is that the support vectors: in the Lagrangian (KKT conditions), when we get to the saddle point it holds that

$$\alpha_i(y_i(\boldsymbol{w}^T \boldsymbol{x}_i + w_0) - 1 + \xi_i) = 0 \quad \forall i$$

and

$$\beta_i \xi_i = 0 \quad \forall i$$

This implies that the support vectors, if $\alpha_i = 0$, the example does not contribute to the decision. If $\alpha_i > 0$ than it is necessarily true that $(y_i(\boldsymbol{w}^T \boldsymbol{x}_i + w_0) - 1 + \xi_i) = 0$. This happens because we have and additional constraint (13.8) that is not a KKT condition but comes directly from the derivative. Suppose that $\alpha_i < C$, then

$$C - \alpha_i - \beta_i = 0 \implies \beta_i > 0$$

If $\beta_i > 0$, then to satisfy $\beta_i \xi_i = 0$ we need $\xi_i = 0$. Therefore

$$y_i(\boldsymbol{w}^T \boldsymbol{x}_i + w_0) = 1$$

which is the same condition of the hard margin SVMs. The examples stay on the hyperplane with confidence one. For any example that has a confidence greater than zero but smaller than C, this example stays on the hyperplane with confidence equal to one.

For the examples for which $\alpha_i = C$, it means that $\beta_i = 0$, then ξ_i does not have to be equal to zero (typically it is not). Therefore the example won't be on the confidence one hyperplane, but closer to the decision hyperplane (the

confidence is less than one). These are called *margin errors*.

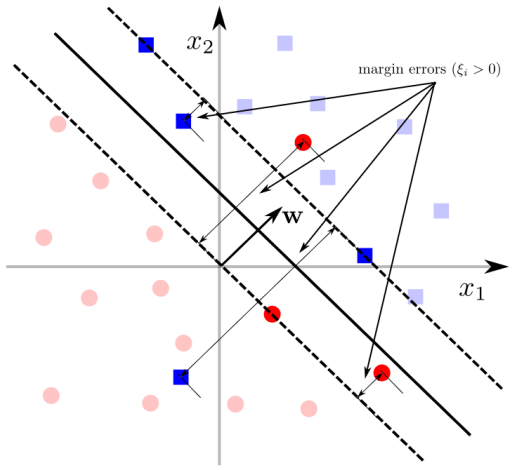


Figure 13.5: soft margin SVM with decision hyperplane, hyperplanes with confidence equal to one, bounded (on the hyperplanes) and unbounded support vectors (miss-labelled that training and margin errors or closer to the decision hyperplane that are margin error)

Both the examples with $\alpha_i \leq 0$ belong to the so called *support vector* because they contribute to the definition of the separation hyperplane (bound and unbound support vectors). All other having $\alpha_i > 0$ do not contribute to the decision hyperplane.

13.4 Large scale SVM learning

Training SVMs is not very efficient (it is more than quadratic problem) for large dataset. If we consider millions of example, SVMs training is not something feasible. People have developed also procedures for large-scale SVMs which is a stochastic gradient descent (GD).

The objective for SVMs is the following minimization

$$min_{\boldsymbol{w}\in\mathcal{X}}\frac{\lambda}{2}||\boldsymbol{w}||^2 + \frac{1}{m}|1 - y_i\langle\boldsymbol{w},\boldsymbol{x}_i\rangle|_+$$

where

- the notation $\langle\dots\rangle$ stands for a dot product.
- in addition, $C = \frac{1}{m}$ for a large scale problem, since the training error could dominate. We normalize by the number of examples.

- $\lambda = \frac{1}{C}$
- there is no w_0, since with high dimensional problem this would mean preventing passing through the origin (not needed)
- we perform feature augmenting via adding 1 in the sum.

Performing stochastic GD means that we compute gradient on single examples (not all together). Therefore we are only considering $x_i y_i$

$$E(\boldsymbol{w}, (\boldsymbol{x}_i, y_i)) = \frac{\lambda}{2}||\boldsymbol{w}||^2 + |1 - y_i\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle|_+$$

This is the error only on **one** example. By computing the gradient on the error

$$\nabla_{\boldsymbol{w}} E(\boldsymbol{w}, (\boldsymbol{x}_i, y_i)) = \lambda \boldsymbol{w} + \mathbb{1}[y_i\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle < 1] y_i \boldsymbol{x}_i$$

At this point we do not know how to carry on, so we will compute a **subgradient**. Whenever we do not have a single gradient in a point, we can perform a subgradient. Its indicator function will be

$$\mathbb{1}[y_i\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle < 1] = \begin{cases} 1 & \text{if } y_i\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle < 1 \\ 0 & \text{otherwise} \end{cases}$$

The subgradient of a function f in a point $\boldsymbol{x}_0$ is any vector $\boldsymbol{v}$ such that for any $\boldsymbol{x}$ hold the condition:

$$f(\boldsymbol{x}) - f(\boldsymbol{x}_0) \geq \boldsymbol{v}^T(\boldsymbol{x} - \boldsymbol{x}_0)$$

This means that when we have discontinuity for the derivative (gradient) we could find subgradients.

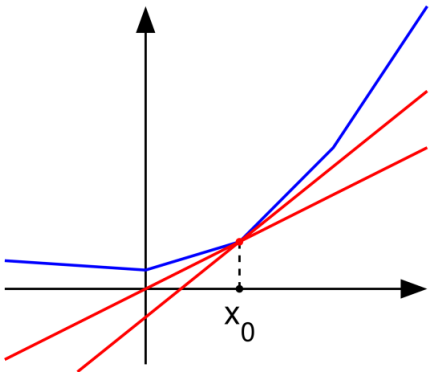


Figure 13.6: Visual representation of a subgradient for a point x_0, the voice of $\boldsymbol{v}$ is among the red lines

We could use any of the (red) vectors that satisfy the condition. The selection of $\boldsymbol{v}$ is defined by the indicator function that tells me if the confidence is smaller

than one. We then can get the exact gradient in each point and then perform subgradient descent.

This means that we update along with the error rate. The only additional piece, is that the learning rate is not constant, but decreases with $\lambda = \frac{1}{C}$ and on t, which is pretty common in GD. This has some theoretical guarantees, but there is no need for theoretical details.

Chapter 14

Non-linear SVMs

Up to this point we have written about linear discriminative machine learning model. In this section we introduce a non-linear discriminative model. One of the good properties of SVMs is that they have a theoretically grounded extension to the non-linear domain.

Hard-margin SVM can address linearly separable problems. Using these kind of SVMs we must assume that training data are linearly separable.

Soft-margin SVM can address linearly separable problems with outliers.

Non-linearly separable problems need a higher expressive power. These problems are characterized for example by more complex feature combinations.

Facing the problem of non-linearly separable problems, we do not want to loose the advantages of linear separators (i.e. large margin, sparsity of the support vectors, theoretical guarantees). To achieve these purposes, the solution is to map input examples (input space) in higher dimensional feature space (this procedure is called *feature mapping*). Once data are represented in the feature space, we perform linear classification in this higher dimensional space.

Feature map

$$\Phi : \mathcal{X} \to \mathcal{H} \tag{14.1}$$

Φ is a function mapping each example to a higher dimensional space $\mathcal{H}$ (potentially infinite dimensional).

Examples $\boldsymbol{x}$ are replaced with their feature mapping $\Phi(\boldsymbol{x})$. The feature mapping should increase the expressive power of the representation (e.g. introducing features which are combinations of input features). Examples should be (approximately) linearly separable in the mapped space.

14.1 Example: polynomial mapping

Sometimes we are interested in considering jointly more variables when building a predictor on top of them. This can be done with polynomial mapping which maps features (input vectors) to all possible conjunctions (i.e. products) of features. There are two types of polynomial mapping:

1. *Homogeneous mapping*: maps features to all possible conjunctions of features of a certain degree d. For example if $d = 2$ and we consider two features:
$$\Phi\binom{x_1}{x_2} = \begin{pmatrix} x_1^2 \\ x_1 x_2 \\ x_2^2 \end{pmatrix}$$

2. *Inhomogeneous mapping*: maps features to all possible conjunctions of features up to a certain degree. For example if $d = 2$ and we consider two features:
$$\Phi\binom{x_1}{x_2} = \begin{pmatrix} x_1 \\ x_2 \\ x_1^2 \\ x_1 x_2 \\ x_2^2 \end{pmatrix}$$

Remark: the higher the degree, the larger the feature space, the higher the degree of interactions that we are able to model with the mapping.

In Figure 14.1 there is no chance to linearly separate the examples in the input space. However, if we apply a polynomial ($d = 2$) mapping representing the data in a three dimensional space, examples become linearly separable.

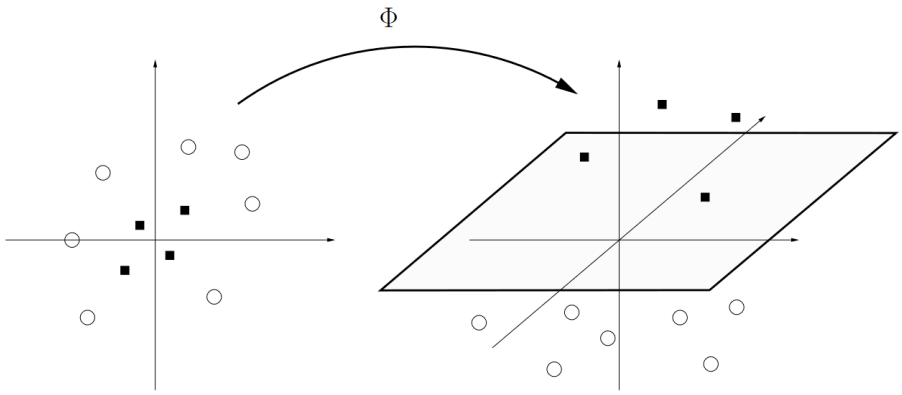


Figure 14.1: If we apply a polynomial mapping representing the data in a three dimensional space, examples become linearly separable.

SVM algorithm is applied just replacing $\boldsymbol{x}$ with $\Phi(\boldsymbol{x})$:

$$f(\boldsymbol{x}) = \boldsymbol{w}^T \Phi(\boldsymbol{x}) + w_0 \tag{14.2}$$

Remark: $\boldsymbol{w}$ in this case is a vector in a three dimensional space.

A linear separation (i.e. hyperplane) in a feature space corresponds to a non-linear separation in an input space (Figure 14.2) :

$$f(\begin{matrix} x_1 \\ x_2 \end{matrix}) = \text{sign}(w_1 x_1^2 + w_2 x_1 x_2 + w_3 x_2^2 + w_0)$$

Indeed $w_1 x_1^2 + w_2 x_1 x_2 + w_3 x_2^2 + w_0$ (obtained solving the dot product between $\boldsymbol{w}^T$ and $\Phi(\boldsymbol{x})$) in terms of input features, is an ellipsoid.

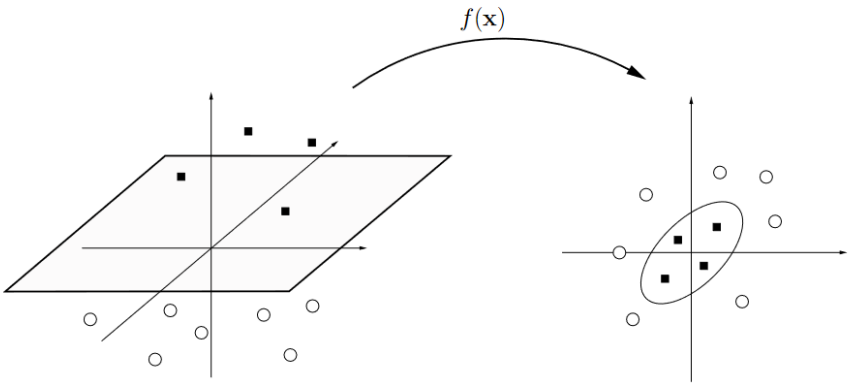


Figure 14.2: A linear separation (i.e. hyperplane) in a feature space corresponds to a non-linear separation in an input space.

Once data are mapped in a higher dimensional space, we can apply the same SVM solvers that we saw in the linearly separable case.

14.2 Non-linear SVM in the case of regression

In the previous chapter we introduced the regularization theory. In that context, our aim was to combine fitting of the training data, which was measured by means of weighted penalties, and model complexity which in the classification problem was the size of the margin.
The purpose is the same in the regression case. The aim is to retain combination of regularization and data fitting (i.e. correctly approximating the output given the input). Analogously to the classification case, the result is a trade-off between model complexity (smoothness of the function) and data fitting. In essence, regularization means smoothness (i.e. smaller weights, lower complexity) of the learned function. Moreover, the focus is to search for sparsifying loss

to have few support vector. For example, in the previous chapter, the fact that support vectors were sparse was a consequence of the hinge loss.

In order to achieve this property in support vector regression, we adopt the ϵ-*intensive loss*. A commonly used loss function in regression, for example in the deep learning context, is the square loss ($l(f(\boldsymbol{x}), y) = (f(\boldsymbol{x}) - y)^2$) or the mean square error (sum up all the square losses which characterizes all the training examples and divide by the number of examples) due to its useful properties (e.g. smoothness). However, if we apply this loss function in the case of SVM we miss an area where we do not pay anything. The only case when we do not pay anything is when $f(\boldsymbol{x})$ is exactly equal to y, but when dealing with a regression problem we are considering the continuous domain. The idea is to adopt a loss function which is more tolerant than this in order to achieve the sparsity property. We introduce a regression version of the hinge loss which is called ϵ-*intensive loss* (Figure 14.3).

ϵ-insensitive loss

$$\ell(f(\boldsymbol{x}, y)) = |y - f(\boldsymbol{x})|_\epsilon = \begin{cases} 0 & \text{if} |y - f(\boldsymbol{x})| \leq \epsilon \\ |y - f(\boldsymbol{x})| - \epsilon & \text{otherwise} \end{cases} \tag{14.3}$$

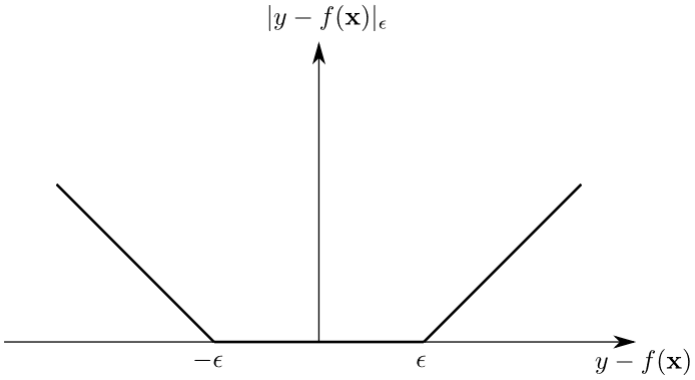


Figure 14.3: The idea is to adopt a loss function which is more tolerant than this in order to achieve the sparsity property. We introduce a regression version of the hinge loss which is called ϵ-*intensive loss*.

The ϵ-intensive loss depends on the difference between y and $f(\boldsymbol{x})$ because we are speaking about regression and not classification.

Remark: in $|y - f(\boldsymbol{x})|_\epsilon$, the ϵ reminds the positive sign which we have in the hinge loss $|1 - yf(\boldsymbol{x})|_+$.

The result is that we are able to tolerate small (ϵ) deviations from the true value (i.e. in this tolerance region we pay no penalty). This is reasonable, remember that in regression the data are typically the result of measurements.

On the other hand if $|y - f(\boldsymbol{x})| > \epsilon$ we pay a linear penalty as in the hinge loss. The obtained structure is in some ways a two sided hinge loss.

In the middle of the function it is defined an ϵ-tube of insensitiveness around true values. If the difference stays within this tube we do not pay anything.

The larger the ϵ the more tolerant is our model. As well as C in the linearly separable SVM, also ϵ is something that we have to configure beforehand (hyperparameter). Hyperparameters are configured in the model selection stage (basically, try out different values and evaluate the performance on a validation set). Playing on ϵ value allows to trade off function complexity with data fitting.

14.2.1 Support Vector Regression optimization problem

Let's start with the constraints:

- $|y-f(\boldsymbol{x})|_\epsilon$ means that for all i it must hold $y_i - f(\boldsymbol{x_i}) \leq \epsilon$ and $f(\boldsymbol{x_i}) - y_i \leq \epsilon$

The complexity term is again related to the minimization of the norm of the weights.

$$\min \frac{||w||^2}{2}$$

In the SVM for classification, the complexity term, the norm of the weights, corresponds to the inverse of the margin. In regression, the norm of the weights corresponds to the complexity of the function (the number of bumps which we have, a lot of bumps require a lot of weights).

With this formulation, we are assuming that all the training examples has to lie in the ϵ-tube. However, as well as in the case of hard margin SVMs, in order to be learnable, this formulation could require really large values of ϵ. As a result, as in the soft margin SVMs, we can introduce slack variables and penalties for non-satisfied constrains. If we want to relax the condition $y_i - f(\boldsymbol{x_i}) \leq \epsilon$, we can write:

$$y_i - f(\boldsymbol{x_i}) \leq \epsilon + \xi$$

and symmetrically:

$$f(\boldsymbol{x_i}) - y_i \leq \epsilon + \xi^*$$

Remark: we write ξ^* in this latter formula to highlight the fact that $\xi \neq \xi^*$.

Overall, the resulting learning problem is:

$$\min_{\boldsymbol{w} \in \mathcal{X}, w_0 \in \mathbb{R}, \boldsymbol{\xi}, \boldsymbol{\xi^*} \in \mathbb{R}^m} \frac{1}{2}||\boldsymbol{w}||^2 + C \sum_{i=1}^{m} (\xi_i + \xi_i^*) \tag{14.4}$$

subject to

$$\boldsymbol{w}^T\Phi(\boldsymbol{x_i}) + w_0 - y_i \leq \epsilon + \xi_i$$
$$y_i - (\boldsymbol{w}^T\Phi(\boldsymbol{x_i}) + w_0) \leq \epsilon + \xi_i^*$$
$$\xi_i, \xi_i^* \geq 0$$

Remark: we get two constraints for each example for the upper and lower sides of the tube. Slack variables ξ_i, ξ_j^* penalize predictions out of the ϵ-insensitive tube.

Remark: again the obtained formula is a quadratic optimization problem with linear constraints.

14.2.2 Lagrangian in the regression case

As well as in the classification case, we can address the regression optimization problem using Lagrange multipliers. We include constraints in the minimization function using Lagrange multipliers ($\alpha_i, \alpha_i^*, \beta_i, \beta_i^*, \beta_i^* \geq 0$).

- **constraint 1:**
$$f(\boldsymbol{x_i}) - y_i \leq \epsilon + \xi_i$$
$$\epsilon + \xi_i - f(\boldsymbol{x_i}) + y_i \geq 0$$

- **constraint 2:**
$$y_i - f(\boldsymbol{x_i}) \leq \epsilon + \xi_i$$
$$\epsilon + \xi_i - y_i + f(\boldsymbol{x_i}) \geq 0$$

- **constraint 3:**
$$\xi_i \geq 0$$

- **constraint 4:**
$$\xi_i^* \geq 0$$

The Lagrangian is:

$$L = \frac{||\boldsymbol{w}||^2}{2} + C\sum_i(\xi_i + \xi_i^*)$$
$$-\sum_i \alpha_i(\epsilon + \xi_i - y_i + f(\boldsymbol{x_i}))$$
$$-\sum_i \alpha_i^*(\epsilon + \xi_i^* - f(\boldsymbol{x_i}) + y_i)$$
$$-\sum_i \beta_i\xi_i$$
$$-\sum_i \beta_i^*\xi_i^*$$

In this case the primal variables are: $\boldsymbol{w}$, w_0, $\boldsymbol{\xi}$, $\boldsymbol{\xi}^*$.

I compute the gradient of the Lagrangian with respect to $\boldsymbol{w}$. Some terms disappear because they do not depend on $\boldsymbol{w}$.

$$\nabla_{\boldsymbol{w}} L = \boldsymbol{w} - \nabla_{\boldsymbol{w}} \sum_i \alpha_i (\boldsymbol{w}^T \Phi(\boldsymbol{x_i}) + w_0) + \nabla_{\boldsymbol{w}} \sum_i \alpha_i^* (\boldsymbol{w}^T \Phi(\boldsymbol{x_i}) + w_0) =$$

$$\boldsymbol{w} - \sum_i \alpha_i \Phi(\boldsymbol{x_i}) + \sum_i \alpha_i^* \Phi(\boldsymbol{x_i})$$

We now vanish the gradient setting it to zero.

$$\nabla_{\boldsymbol{w}} L = \boldsymbol{w} - \sum_i \alpha_i \Phi(\boldsymbol{x_i}) + \sum_i \alpha_i^* \Phi(\boldsymbol{x_i}) = 0$$

$$\boldsymbol{w} = \sum_i (\alpha_i - \alpha_i^*) \Phi(\boldsymbol{x_i})$$

Again $\boldsymbol{w}$ is a linear combination of training examples with coefficients $(\alpha_i - \alpha_i^*)$.

Now we compute the derivative of the Lagrange with respect to w_0.

$$\frac{\partial L}{\partial w_0} = \frac{\partial}{\partial w_0} - \sum_i \alpha_i (w_0) + \frac{\partial}{\partial w_0} \sum_i \alpha_i^* (w_0)$$

Setting this latter result equal to zero we get:

$$\frac{\partial L}{\partial w_0} = - \sum_i \alpha_i + \sum_i \alpha_i^* = 0$$

$$\sum_i (\alpha_i^* - \alpha_i) = 0$$

The result is a constraint on the sum of the alphas.

Now we compute the derivative with respect to $\boldsymbol{\xi}$.

$$\frac{\partial L}{\partial \xi_i} = \frac{\partial}{\partial \xi_i} (C\xi_i - \alpha_i \xi_i - \beta_i \xi_i)$$

If we set this equal to zero we obtain the same result as in the classification case:

$$C - \alpha_i - \beta_i = 0$$

The derivative with respect to $\boldsymbol{\xi}^*$ is the more or less the same.

$$\frac{\partial L}{\partial \xi_i^*} = C - \alpha_i^* - \beta_i^* = 0$$

At this point we have to replace the obtained result in the Lagrangian.

First of all we think about $\boldsymbol{w}$ highlighting where it appears:

$$\frac{\boldsymbol{w}^T\boldsymbol{w}}{2} - \sum_i \alpha_i \boldsymbol{w}^T \Phi(\boldsymbol{x_i}) + \sum_i \alpha_i^* \boldsymbol{w}^T \Phi(x_i)$$

Now we replace $\boldsymbol{w}$ with the expression discovered above.

$$(\frac{1}{2}\sum_i(\alpha_i - \alpha_i^*)\Phi(x_i))^T(\sum_j(\alpha_j - \alpha_j^*)\Phi(x_j)) - \sum_i(\alpha_i - \alpha_i^*)(\sum_j(\alpha_j - \alpha_j^*)\Phi(x_j))^T\Phi(x_i)$$

This expression contains two pieces which are the same only one is twice with respect to the other:

$$(\frac{1}{2}\sum_i(\alpha_i - \alpha_i^*)\Phi(x_i))^T(\sum_j(\alpha_j - \alpha_j^*)\Phi(x_j)) - \sum_i(\alpha_i - \alpha_i^*)(\sum_j(\alpha_j - \alpha_j^*)\Phi(x_j))^T\Phi(x_i) =$$

$$-\frac{1}{2}\sum_i\sum_j(\alpha_i - \alpha_i^*)(\alpha_j - \alpha_j^*)\Phi(x_i)^T\Phi(x_j)$$

The obtained result is really similar to the one obtained in the classification case.

Now we do the same thing for w_0. First of all we highlight where it appears:

$$-\sum_i \alpha_i w_0 + \sum_i \alpha_i^* w_0 = w_0 \sum_i (\alpha_i^* - \alpha_i)$$

Since $\sum_i(\alpha_i^* - \alpha_i) = 0$ we get:

$$w_0 \sum_i (\alpha_i^* - \alpha_i) = 0$$

Then we examine the component with ξ_i.

$$\sum_i \xi_i (C - \alpha_i - \beta_i) = 0$$

The same holds for ξ_i^*.

$$\sum_i \xi_i^* (C - \alpha_i^* - \beta_i^*) = 0$$

We now consider the pieces where ϵ appears:

$$-\sum_i \alpha_i \epsilon - \sum_i \alpha_i^* \epsilon + \sum_i \alpha_i y_i - \sum_i \alpha_i^* y_i$$

At the end of the day we get the dual formulation:

$$\begin{aligned} \max_{\boldsymbol{\alpha}\in\mathbb{R}^m} \quad & -\tfrac{1}{2}\sum_{i,j=1}^{m}\left(\alpha_i^*-\alpha_i\right)\left(\alpha_j^*-\alpha_j\right)\Phi\left(\mathbf{x}_i\right)^T\Phi\left(\mathbf{x}_j\right) \\ & -\epsilon\sum_{i=1}^{m}\left(\alpha_i^*+\alpha_i\right)+\sum_{i=1}^{m}y_i\left(\alpha_i^*-\alpha_i\right) \\ \text{subject to} \quad & \sum_{i=1}^{m}\left(\alpha_i-\alpha_i^*\right)=0 \\ & \alpha_i,\alpha_i^*\in[0,C] \quad \forall i\in[1,m] \end{aligned}$$

$\alpha_i, \alpha_i^* \in [0, C]$ because of $C - \alpha_i - \beta_i = 0$ and $C - \alpha_i^* - \beta_i = 0$

As well as in the classification case, the obtained result is still a quadratic optimization problem with box constraints.

It we replace $\boldsymbol{w}$ in the decision function with its dual formulation $\boldsymbol{w} = \sum_i(\alpha_i - \alpha_i^*)\Phi(\boldsymbol{x_i})$ we get:

$$f(\boldsymbol{x}) = \boldsymbol{w}^T\Phi(\boldsymbol{x}) + w_0 = \sum_{i=1}^{m}(\alpha_i - \alpha_i^*)\Phi(\boldsymbol{x_i})^T\Phi(\boldsymbol{x}) + w_0$$

Remark: if Φ is the identity function then this is linear regression. So, we can do linear regression with SVM. If Φ is not the identity function we are doing non linear regression.

14.2.3 Karush-Khun-Tucker conditions (KKT) in the case of support vector regression

In this subsection we reason about the support vectors properties in the dual formulation which we have just introduced. The KKT conditions requires that all terms where the Lagrange multipliers appear should be equal to zero. So, at the saddle point it holds that for all i:

$$\alpha_i(\epsilon + \xi_i + y_i - \boldsymbol{w}^T\Phi(\boldsymbol{x_i}) - w_0) = 0$$

$$\alpha_i^*(\epsilon + \xi_i^* - y_i + \boldsymbol{w}^T\Phi(\boldsymbol{x_i}) + w_0) = 0$$

$$\beta_i\xi_i = 0$$

$$\beta_i^*\xi_i^* = 0$$

Combined with $C - \alpha_i - \beta_i = 0$, $\alpha_i \geq 0$, $\beta_i \geq 0$ and $C - \alpha_i^* - \beta_i^* = 0$, $\alpha_i^* \geq 0$, $\beta_i^* \geq 0$ we get:

$$\alpha_i \in [0, C]$$

$$\alpha_i^* \in [0, C]$$

and

$$\alpha_i = C \quad \text{if } \xi_i > 0$$

$$\alpha_i^* = C \quad \text{if } \xi_i^* > 0$$

Notice that in $f(\boldsymbol{x}) = \sum_{i=1}^{m}(\alpha_i - \alpha_i^*)\Phi(\boldsymbol{x_i})^T\Phi(\boldsymbol{x}) + w_0$ in order for an example to not be a support vector both α_i and α_i^* must be zero. Indeed, an example could be over the tube or below the tube and not satisfy the constraints. To satisfy the constraint the example should be inside the tube. If this is not the case, only one of the two alphas can be non zero because the example cannot be over and below the tube at the same time.
If for example $\alpha_i \neq 0$ then $(\epsilon + \xi_i + y_i - \boldsymbol{w}^T\Phi(\boldsymbol{x_i}) - w_0) = 0$.
If also $\xi_i = 0$, than it means that the difference between $f(\boldsymbol{x_i})$ and y_i is exactly ϵ. In this case there is no penalty because we are on the boundary of the tube. If instead $\xi_i > 0$ the difference between y_i and $f(\boldsymbol{x_i})$ is larger than ϵ. In this case $\beta_i = 0$ and so $\alpha_i = C$.
The same reasoning works symmetrically for the $*$ case.

Remark: according to the professor $\alpha_i \in [0, C]$ is not only a consequence of $C - \alpha_i - \beta_i = 0$, $\alpha_i \geq 0$, $\beta_i \geq 0$, but also of $\beta_i\xi_i = 0$. At this moment, the author is not convinced about this claim. Maybe in the future I will understand.

We can sum up the obtained results as follows:

- All patterns within the ϵ-tube, for which $|f(\boldsymbol{x_i}) - y_i| < \epsilon$, have $\alpha_i, \alpha_i^* = 0$ and thus don't contribute to the estimated function f. All the examples which lie inside the tube are not support vectors.
- Patterns for which either $0 < \alpha_i < C$ or $0 < \alpha_i^* < C$ (they cannot be both non-zero at the same time) are on the border of the ϵ-tube, that is $|f(\boldsymbol{x_i}) - y_i| = \epsilon$. They are the unbound support vectors. (It is like staying exactly on the confidence one hyperplane).
- The remaining training patterns are margin errors (either $\xi_i > 0$ or $\xi_i^* > 0$), and reside out of the ϵ-insensitive region. They are bound support vectors, with corresponding $\alpha_i = C$ or $\alpha_i^* = C$.

For the sake of clarity, Figure 14.4 illustrates the concept graphically.

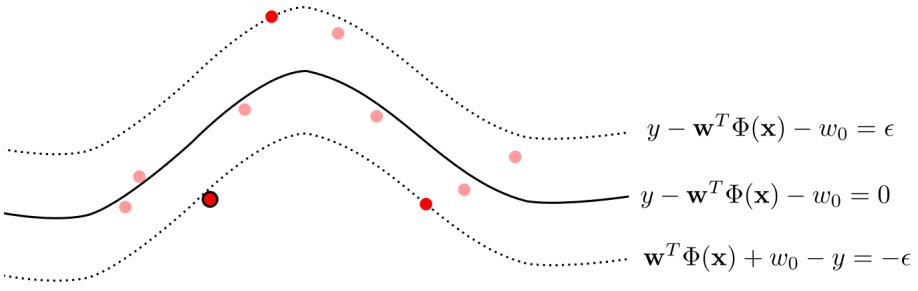


Figure 14.4: The solid curve corresponds to the regions where $y - \boldsymbol{w}^T\Phi\boldsymbol{x} - w_0 = 0$. The two parallel curves are where $y - \boldsymbol{w}^T\Phi\boldsymbol{x} - w_0 = \epsilon$ and $\boldsymbol{w}^T\Phi\boldsymbol{x} - w_0 - y = -\epsilon$.

The solid curve corresponds to the regions where $y - \boldsymbol{w}^T \Phi \boldsymbol{x} - w_0 = 0$. The two parallel curves are where $y - \boldsymbol{w}^T \Phi \boldsymbol{x} - w_0 = \epsilon$ and $\boldsymbol{w}^T \Phi \boldsymbol{x} - w_0 - y = -\epsilon$. The space between the two parallel curves with the solid curve in the middle is the ϵ-insensitive tube. This structure reminds the classification case. In this last, examples must be outside the margin. In this case we look for examples which stay inside the tube.
All the shaded points, which correspond to the points inside the tube, are not support vectors.
The examples which stay exactly on the border of the tube are unbound support vectors.
The red point which is slightly out of the tube is a bound support vector.

Looking at Figure 14.5 we notice that reducing ϵ the tube becomes smaller. The function can accommodate more bumps if needed (it has more flexibility). As a result the number of support vector increases. Indeed, if the tube becomes smaller it is difficult that examples stay inside the tube. In the third case the tube is really small, as a result almost all the points are support vectors.
In conclusion we understood that the complexity of the function is regulated by the minimization of the weights and by the configuration of the hyperparameter ϵ which corresponds to the size of the tube.

Remark: in the last case of Figure 14.5 it is clear that we are allowing too much flexibility. In this case we tend to interpolate all the examples including their noise.

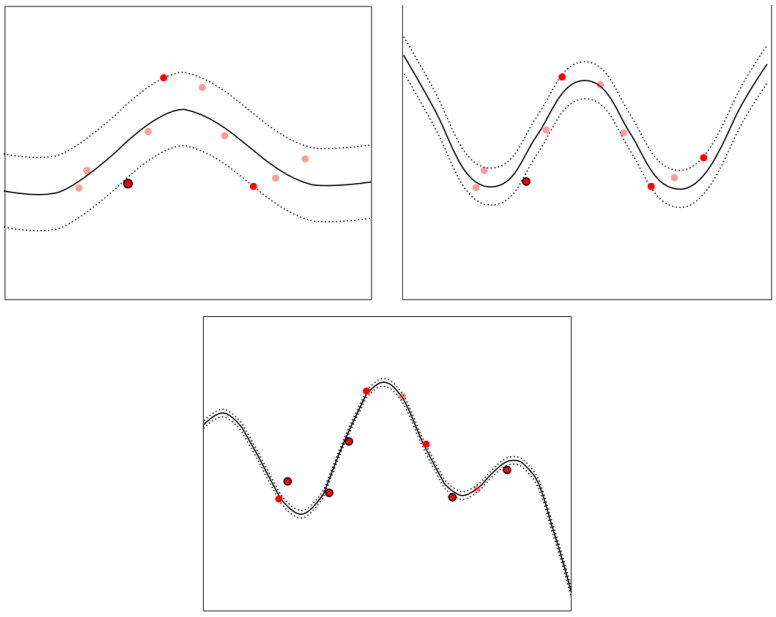


Figure 14.5: Reducing ϵ, the size of the tube reduces.

Chapter 15

Kernel machines

SVMs allow both linear and non linear problems. We saw a solution for non-linear SVMs that consists in a map in non-linear features as combination of features. This is a solution that is valid also for perceptrons. Therefore is not specific to SVMs but it becomes computationally unfeasible in high dimensional cases. Also you need to find the correct map and it is a non trivial procedure. The dual formulation of the SVMs (both classification and regression), the feature map only appear in dot products, which gets really expensive.

For SVMs we can do non-linear learning in a different way, this allows with infinite dimension spaces (the *kernel* trick).

Kernel trick
The *kernel trick* consists in replacing the dot products that appear in the dual formulation of non-linear SVMs with an equivalent kernel function:

$$k(\boldsymbol{x}, \boldsymbol{x'}) = \Phi(\boldsymbol{x})^T \Phi(\boldsymbol{x'})$$

The kernel function uses example in *input space* (not feature). We can build a function that can compute this without to explicitly build the map.

As before, the dual optimization problem is

$$max_{\alpha \in \mathbb{R}^m} \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i y_i \alpha_j y_j \Phi(\boldsymbol{x_i})^T \Phi(\boldsymbol{x_j})$$

in which Φ only appears as dot product. subject to:

$$\alpha_i \geq 0 \quad i = 1, \ldots, m$$

$$\sum_{i=1}^{m} \alpha_i y_i = 0$$

The dual decision function is

$$f(\boldsymbol{x}) = \sum_{i=1}^{m} \alpha_i y_i \Phi(\boldsymbol{x}_i)^T \Phi(\boldsymbol{x'}) = \sum_{i=1}^{m} \alpha_i y_i k(\boldsymbol{x}_i, \boldsymbol{x'})$$

in which $\Phi(\boldsymbol{x}_i)^T \Phi(\boldsymbol{x'})$ is in fact the dot product. I can replace the dot product with $k(\boldsymbol{x}, \boldsymbol{x'}) = \Phi(\boldsymbol{x}_i)^T \Phi(\boldsymbol{x'})$, since $\boldsymbol{x}$ does not appear anywhere else, we are done.

15.1 What are kernels

Let's see an example of a polynomial mapping (both homogeneous and inhomogeneous).

- Homogeneous:

$$k(\mathbf{x}, \mathbf{x}') = \left(\mathbf{x}^T \mathbf{x}'\right)^d$$

- E.g. $(d = 2)$

$$\begin{aligned} k\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \begin{pmatrix} x_1' \\ x_2' \end{pmatrix}\right) &= (x_1 x_1' + x_2 x_2')^2 \\ &= (x_1 x_1')^2 + (x_2 x_2')^2 + 2 x_1 x_1' x_2 x_2' \\ &= \underbrace{\begin{pmatrix} x_1^2 & \sqrt{2} x_1 x_2 & x_2^2 \end{pmatrix}}_{\Phi(\mathbf{x})^T} \underbrace{\begin{pmatrix} x_1'^2 \\ \sqrt{2} x_1' x_2' \\ x_2'^2 \end{pmatrix}}_{\Phi(\mathbf{x}')} \end{aligned}$$

We can substitute the $k(\boldsymbol{x}, \boldsymbol{x'})$ with $(\boldsymbol{x}^T, \boldsymbol{x'})^d$ (dot product in input space) at the power of d, whose result is a scalar. The number of operations depends on the number of input features, not on the size of the features space.
A kernel it always correspond to a dot product in some feature space. For each piece of the sum of the second line, we need to to separate $\boldsymbol{x}$ from $\boldsymbol{x'}$. Therefore, in this case the dot product corresponds to $\Phi(\boldsymbol{x})^T \Phi(\boldsymbol{x'})$, which makes it a dot product (polynomial mapping, aside from the coefficient). We did not explicitly compute Φ but we computed only at input-space level.
In this case we compute two summation instead of three, but with greater feature spaces and powers then we get a large advantage. We are computing a polynomial mapping in a non-explicit way.

It is also more effective in the case of an inhomogeneous polynomial kernel, in which not only we have combination of degree d but also all combination with degree *up to* $\mathrm{d} \leq d$. This increases a lot the number of features. We can compute it by summing 1 to the dot product in input space before raising to the power of d. Adding 1 corresponds to a dot product in a inhomogeneous polynomial mapping. The procedure is similar. Again, we separate $\boldsymbol{x}$ from $\boldsymbol{x'}$

and we get two Φ functions, which contain term with degree up to d. Also in this case we did explicitly computed in feature-space level.

- Inhomogeneous:

$$k(\mathbf{x}, \mathbf{x}') = \left(1 + \mathbf{x}^T \mathbf{x}'\right)^d$$

- E.g. $(d = 2)$

$$\begin{aligned} k\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \begin{pmatrix} x_1' \\ x_2' \end{pmatrix}\right) &= (1 + x_1 x_1' + x_2 x_2')^2 \\ &= (1) + (x_1 x_1')^2 + (x_2 x_2')^2 + 2x_1 x_1' + 2x_2 x_2' + 2x_1 x_1' x_2 x_2' \\ &= \underbrace{\begin{pmatrix} 1 & \sqrt{2}x_1 & \sqrt{2}x_2 & x_1^2 & \sqrt{2}x_1 x_2 & x_2^2 \end{pmatrix}^T}_{\Phi(\mathbf{x})^T} \underbrace{\begin{pmatrix} 1 \\ \sqrt{2}x_1\prime \\ \sqrt{2}x_2\prime \\ x_1'^2 \\ \sqrt{2}x_1' x_2' \\ x_2'^2 \end{pmatrix}}_{\Phi(\mathbf{x}')} \end{aligned}$$

The fact that we rise in an $\mathbb{R}^n$ dimensional space does not change complexity since we are raising scalars. Usual we do not use high complexity spaces because we would get too many features and we risk overfitting.

This is possible only because we are working in the *dual form* and we get $\boldsymbol{x}$ only as dot products.

Kernel
A valid **kernel** what works properly, is a function defined over the cartesian product of the input space

$$k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$$

that correspond to a dot product in *some* feature space

$$k(\boldsymbol{x}, \boldsymbol{x}') = \Phi(\boldsymbol{x})^T \Phi(\boldsymbol{x}')$$

Kernels can be seen as similarities between objects.

Indeed, you can apply kernels to input that are not vectors at all (as sequences or graphs). We can define kernels on arbitrary structures. In practice, we can apply several algorithms to these object via this kernel that implicitly maps these object in some vectors.

15.2 Validity of a kernel

In many cases, if we build learning systems on non-vector object we have to invent the kernel ourselves, therefore we need to verify that it is a valid one.

Gram matrix
Given examples $\{\boldsymbol{x}_1, \dots, \boldsymbol{x}_m\}$ and a kernel function k, the *Gram matrix* K is the symmetric matrix of pairwise kernels between examples:

$$K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j) \quad \forall i, j$$

This matrix is symmetric.

Positive definite matrix
A symmetric $m \times m$ matrix K is a positive (*semi-*)definite if

$$\sum_{i,j=1}^{m} \boldsymbol{c}_i \boldsymbol{c}_j K_{ij} \geq 0 \quad \forall \boldsymbol{c} \in \mathbb{R}^m$$

If equality only holds for $\boldsymbol{c} = 0$, the matrix is *strictly positive definite*. All eigenvalues of K are non negative, or are positive (for strictly positive definite).

It is not enough to see what happens only on the training set matrix: we would like to show that the *function* is positive definite, that it is indeed a function. There is a direct way to do it which is an eigen-decomposition for functions, which will not be discussed and could be tricky.

The easier way to show this validity is checking the satisfaction of at least one of these conditions and requisites:

- prove its positive definiteness (difficult)
- find corresponding feature map: explication of Φ dot-product combination by making the feature map explicit
- use kernel combination properties in order to build a new kernel, the operation that we can perform on kernels will preserve their properties.

Kernelizing different SVMs
This procedures also works for different types of SVMs, therefore we can make appear Φ only in dot products. Example support vector regression:

- Dual problem:

$$\max_{\alpha \in \mathbb{R}^m} -\frac{1}{2}\sum_{i,j=1}^{m}(\alpha_i^* - \alpha_i)\left(\alpha_j^* - \alpha_j\right)\underbrace{\Phi\left(\mathbf{x}_i\right)^T\Phi\left(\mathbf{x}_j\right)}_{k(\mathbf{x}_i,\mathbf{x}_j)}$$
$$-\epsilon\sum_{i=1}^{m}(\alpha_i^* + \alpha_i) + \sum_{i=1}^{m} y_i\left(\alpha_i^* - \alpha_i\right)$$
$$\text{subject to} \quad \sum_{i=1}^{m}(\alpha_i - \alpha_i^*) = 0 \quad \alpha_i, \alpha_i^* \in [0, C] \quad \forall i \in [1, m]$$

- Regression function:

$$f(\mathbf{x}) = \mathbf{w}^T\Phi(\mathbf{x}) + w_0 = \sum_{i=1}^{m}(\alpha_i - \alpha_i^*)\underbrace{\Phi\left(\mathbf{x}_i\right)^T\Phi(\mathbf{x})}_{k(\mathbf{x}_i,\mathbf{x})} + w_0$$

Kernelizing a perceptron

The kernel procedure can be also be applied on perceptrons: a linear function of a perceptron *kernelized* becomes a non-linear function.
We can take the dual formulation of the perceptron. Remember that in kernel machines (in both regression and classification), $f(x)$ is a combination of the sum over the train examples coefficient times the kernel function. For a classic stochastic perceptron, the procedure is the following: at first we set

$$\boldsymbol{w} = 0$$

then we iterate until all examples are correctly classified (stochastic perceptron) and we update each incorrectly classified examples:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta y_i \boldsymbol{x}_i$$

We learn this coefficient going through the data.

For a kernel perceptron it becomes: at first we initialize

$$\alpha_i = 0 \quad \forall i$$

then we iterate through the examples and for each miss-classified example we perform an update:

$$\alpha_i \leftarrow \alpha_i + \eta y_i$$

The kernel perceptron classification function becomes

$$f(\boldsymbol{x}) = \sum_{i=1}^{m}\alpha_i k(\boldsymbol{x}_i, \boldsymbol{x})$$

15.3 Types of kernels

Two principal types of kernels are

- linear
$$k(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{x}^T \boldsymbol{x}'$$
- polynomial, that can be parameterized
$$k_{c,d}(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^T \boldsymbol{x}' + c)^d$$
- Gaussian kernel 15.3.1

15.3.1 Gaussian kernel

Apart from the coefficients typical of the normal, it is the same as a Gaussian distribution. We can assume that one of the two input is the sample and the other the mean (the Gaussian is symmetric, so it does not matter the order).

$$\begin{aligned} k_\sigma(\boldsymbol{x}, \boldsymbol{x}') &= exp\left(-\frac{||\boldsymbol{x} - \boldsymbol{x}'||^2}{2\sigma^2}\right) \\ &= exp\left(-\frac{\boldsymbol{x}^T\boldsymbol{x} - 2\boldsymbol{x}^T\boldsymbol{x}' + \boldsymbol{x}'^T\boldsymbol{x}'}{2\sigma^2}\right) \end{aligned}$$

This is an exponential detection of the difference between the values. This is an exponentially decaying kernel.

First: if we unroll the square norm, we get a Gaussian kernel that can be computed as dot products in input space, and then we operate scalar operation that produce a Gaussian kernel. Once again, the complexity is bounded to the input space.
A Gaussian kernel has an infinite dimension feature space, this determines a useful property called *universality*: this means that it can uniformly approximate any function (provided it is continuous). In principle, with a Gaussian kernel, we can approximate any (continuous) function.

The problem of this kernel is finding the correct value of σ, the larger the value, the more tolerance we allow (on the other hand: the higher, the stricter, then only the closest training sample will impact the prediction, more similar to kNN).

15.3.2 How to choose

The choice of the kernel (also the choice of σ for a Gaussian kernel) has to be made before the training, so we need to specify them in term of cross-validation.

15.4 Kernels on structured data

We previously said that kernel machines come with the nice feature of the possibility of being applied on data structures different than vectors. By working in the dual we can replace $\boldsymbol{x}$ with the kernel function, I can apply the learning algorithm and I am sure to compute large margin solution in feature space, even if the input space is a non-vector space. In this case, the kernel can be seen as a way of generalizing the dot product in arbitrary spaces. This is the driving principle of designing a kernel.

A kernel of a structure is typically built in terms of *combination of kernels over pieces* (structures have lot of pieces, and then combined). Here some types of kernels on structures (x are not vectors anymore)

- **match kernel** (or *delta*) which consists in a delta function:

$$k_\sigma(x, x') = \delta(x, x') = \begin{cases} 0, & \text{if } x = x' \\ 1, & \text{otherwise} \end{cases} \tag{15.1}$$

  This does not make sense with vectors (low generalization). It makes sense indeed when we use this in combination.

- **string kernel 3-gram spectrum kernel** which basically looks at frequencies.

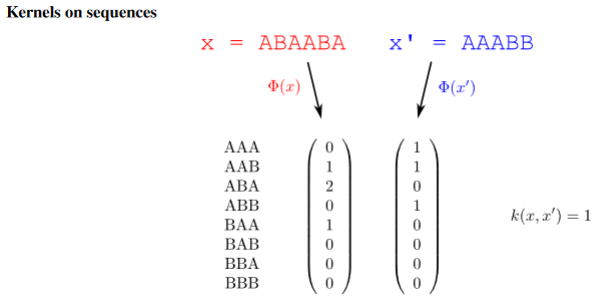


Figure 15.1: example of kernel applied on sequences

It looks at how many times the triplet appears as continuous sub-sequences in x and compare them with same sub-sequences in x'. Then the kernel does a dot-product in this feature space.

15.4.1 Building kernel as combination

Build a kernel by combining pieces means build a valid kernel by following some rules that ensure validity.

1. simpler kernel can be combined using certain operation $(+, \times, \dots)$

2. Kernel combination allows to design complex kernels on structures from simpler ones
3. correctly using combination operators guarantees that complex kernels are preserved

15.4.1.1 Kernel summation

The sum of two kernels corresponds to the *concatenation* of their respective feature space.

$$\begin{aligned}(k_1 + k_2)(x, x') &= k_1(x, x') + k_2(x, x') \\ &= \Phi_1(x)^T \Phi_1(x') + \Phi_2(x)^T \Phi_2(x') \\ &= (\Phi_1(x)\Phi_2(x)) \begin{pmatrix} \Phi_1(x') \\ \Phi_2(x') \end{pmatrix}\end{aligned}$$

The assumption is of course the combination of valid kernels. k_1, k_2 could look at different part of the input, not necessarily the same (example two different portion or characteristic of strings).

15.4.1.2 Kernel multiplication

The product of valid kernel is a valid kernel.

$$\begin{aligned}(k_1 \times k_2)(x, x') &= k_1(x, x')\, k_2(x, x') \\ &= \sum_{i=1}^{n} \Phi_{1i}(x)\Phi_{1i}(x') \sum_{j=1}^{m} \Phi_{2j}(x)\Phi_{2j}(x') \\ &= \sum_{i=1}^{n}\sum_{j=1}^{m} (\Phi_{1i}(x)\Phi_{2j}(x))\,(\Phi_{1i}(x')\,\Phi_{2j}(x')) \\ &= \sum_{k=1}^{nm} \Phi_{12k}(x)\Phi_{12k}(x') = \Phi_{12}(x)^T \Phi_{12}(x')\end{aligned}$$

where $\Phi_{12}(x) = \Phi_1(x) \times \Phi_2(x)$ is the cartesian product between the spaces of the two kernels. We have to compute over every combination of features. The feature space that comes out of it is more complex. This already explodes the feature spaces.
Still, this is a valid combination.

15.4.1.3 Linear combination

We can always multiply a kernel by a scalar, provided that the scalar is non-negative (would invert every prediction). We can define a kernel as linear combination of kernels.

$$k_{sum}(x, x') = \sum_{k=1}^{K} \beta_k k_k(x, x')$$

Since understanding which kernel to use for a specific problem is non trivial, we can use a linear combination of different kernels and then learn parameters of kernels β along with learning α. We can learn both coefficient jointly to select which kernel is more useful (**kernel learning**).

15.4.1.4 Decomposition kernels

We need to decompose our structure in some way we can apply different kernels. We can do it hierarchically, another way to formalizing the decomposition is using a **decomposition kernel** or **convolutional** kernels. This works by having a *decomposition relationship* that takes one object and brakes it into *parts*

$$R(x) = (x_1, x_2, \dots, x_D)$$

A decomposition kernel is

$$(k_1 * \cdots * k_D)(x, x') = \sum_{(x_1,\dots,x_D)\in R(x)} \sum_{(x'_1,\dots,x'_D)\in R(x')} \prod_{d=1}^{D} k_d(x_d, x'_d)$$

We compute a kernel, we sum over every possible decomposition of the two objects according to their decomposition function and then we compare the two decomposition (in the convolution case) as the products of the single pieces. k_d is a kernel on pieces that could be hierarchically defined (or simply the match kernel).

Set kernel
This is a very simple example, basically the decomposition relationship is the set - membership. Therefore the overall kernel is the sum of overall combinations of the kernel between them.

$$k_{set}(X, X') = \sum_{\xi\in X} \sum_{\xi'\in X'} k_{member}(\xi, \xi')$$

This becomes easy when the k_{member} function is a delta function.

15.4.1.5 Kernel normalization

When working with structures, a problem could be that you use different dimensional objects (the similarity of a long sequence will be higher than the one computed on a short sequence). Therefore we need to normalize our kernels by taking into account the size of the structure

$$\hat{k}(x, x') = \frac{k(x, x')}{\sqrt{k(x, x)k(x', x')}}$$

At the denominator we get exactly the norm. This particular normalization is the *cosine normalization* because it takes into account the angle between the two (we are getting rid of the size of the object).

15.4.1.6 Kernel composition

We can build not only by combining, but also via composition. Instead of having $x^T x$ in the Gaussian kernel, we can plug another kernel and put it as input of the Gaussian kernel.

$$(k_{d,c} \circ k)(x, x') = (k(x, x') + c)^d$$

$$(k_\sigma \circ k)(x, x') = exp\left(-\frac{k(x, x) - 2k(x, x') + k(x', x')}{2\sigma^2}\right)$$

This gets us to a composition of kernels. This project us in higher dimensional space.

15.4.2 Kernel on graphs (WL kernel)

We can also exploit kernel on structured data as graphs. The Weistfeiler-Lehman (WL) graph kernel relies in the WL approximation for graphs isomorphism. This is an efficient way of dealing with kernel on graphs.

15.4.2.1 WL isomorphism test

This test is conducted on two graphs $\mathcal{G} = \{E, V\}, \mathcal{G}' = \{E', V'\}$ with equal number of vertices. Let $\mathcal{L}(\mathcal{G})$ a label function for each node in $\mathcal{G}$ such that contains $l(v)$ for each node $v \in V$. There is also a label function for $\mathcal{G}'$ and they need to contain the same labels. The algorithm goes like this:

1. set $l_0(v) = l(v)$ for all v (set initial labels of the graph $\mathcal{G}$ as the given labels for that graph).
2. for each $i \in [1 \dots |V|]$
   (a) for each node v in $\mathcal{G}, \mathcal{G}'$
      i. construct a list of sorted labels $M_i(v)$ of the adjacent nodes of v of the previous step

$$M_i(v) = \{l_{i-1}(u)\} \qquad \forall u \text{ adjacent to } v$$

      ii. concatenate the label for the node v at previous step with the $m_i(v)$ list (Figure 15.2), also use a function *label* that assigns unique labels to the string (Figure 15.3) (could be for example an hash function)

$$l_i(v) = label(l_{i-1}(v), M_i(v))$$

   (b) if $l_i(\mathcal{G}) \neq l_i(\mathcal{G}')$
      i. **return FALSE**
3. **return TRUE**

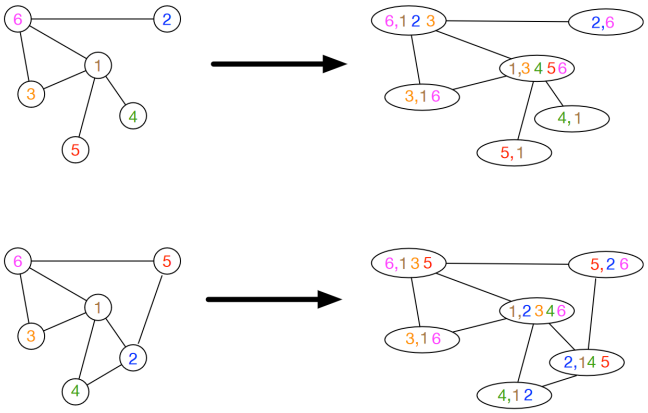


Figure 15.2: WL-isomorphism test, we are assigning to each node a string that contains the labels of all adjacent nodes

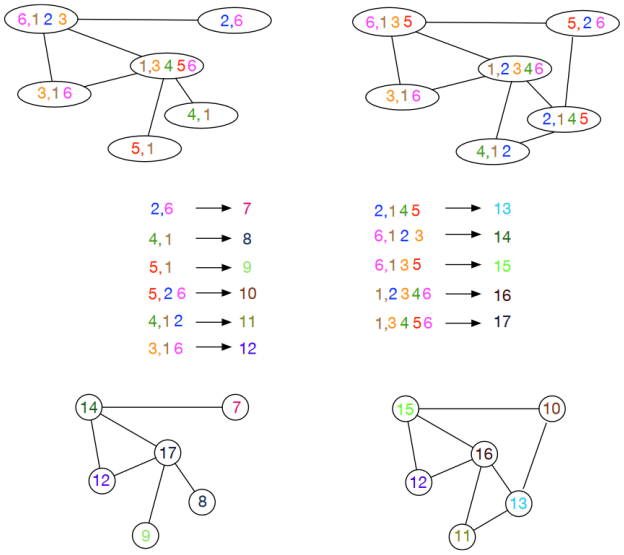


Figure 15.3: WL-isomorphism test, we using the *label* function to convert the string into unique labels

15.4.2.2 WL graph kernel

Let $\{\mathcal{G}_0, \ldots, \mathcal{G}_h\} = \{(V, E, l_0), \ldots, (V, E, l_h)\}$ a sequence of graphs made from $\mathbf{G}$, where we are identifying the label i as the i-*th* iteration of the WL algorithm. Let $k : \mathcal{G} \times \mathcal{G}' \to \mathbb{R}$ any kernel on graph. Then the WL graph kernel is defined

as

$$k_{WL}^{h}(\mathcal{G}, \mathcal{G}') = \sum_{i=0}^{h} k(\mathcal{G}_i, \mathcal{G}_i')$$

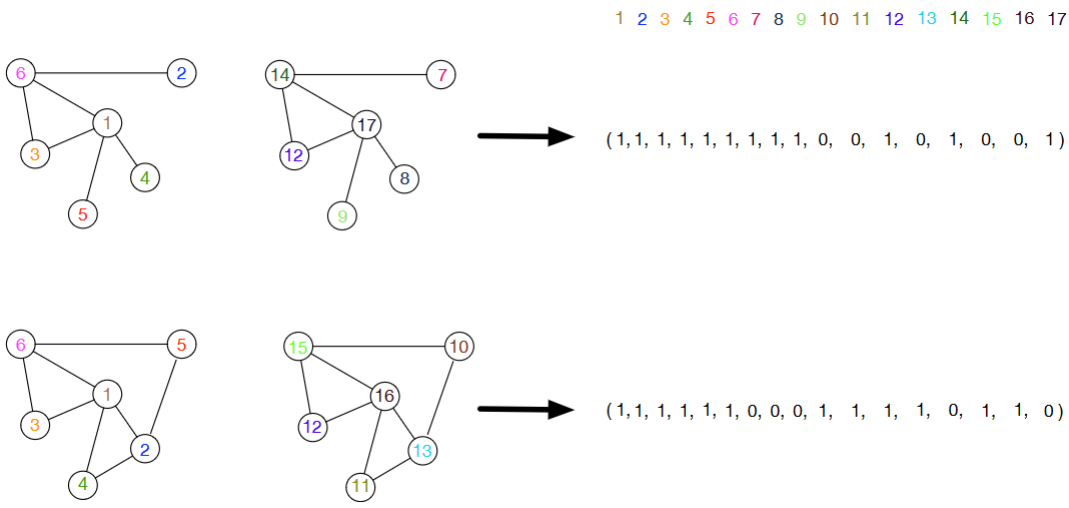


Figure 15.4: Execution of the graph kernel. This kernel will use a string in which it will store 1 or 0 respectively if it has found the i-th label in the graph or not.

Chapter 16

Deep learning

A crucial limitation of the perceptron is that it can only model linear functions. Hence, as we have discussed, we cannot for example provide a linear separation for the XOR. We introduced non-linearity by means of feature mapping. Moreover, we studied how to implicitly provide non-linearity via kernels. To do this we need to design appropriate kernels (possibly selecting from a set, i.e. kernel learning) able to produce reasonable similarity measures. The most common solution (i.e. the decision function) is *linear combination of kernels*.

$$f(\boldsymbol{x}) = \sum_i \alpha_i y_i K(\boldsymbol{x_i}, \boldsymbol{x})$$

Remark: this formula is meaningful for the classification case. In regression the coefficient for $K(\boldsymbol{x_i}, \boldsymbol{x})$ is different.

There are two main drawback regarding kernels:

- We need to design appropriate kernels, perhaps selecting from a set.
- Once the appropriate kernels are selected, the decision function is a linear combination of them. The kernel structure is constant and it is not learned or progressively refined.

The *multilayer perceptron* (MLP) is the simplest network of interconnected neurons. It is a layered architecture: neurons from one layer send outputs to the following layer (this structure is named *feed forward neural network*). The input layer is at the bottom of the architecture (input features). One or more hidden layers are placed in the middle (learned features). Finally on the top there is the output layer (where the prediction takes place). A graphical representation of the multilayer perceptron is illustrated in Figure 16.1.

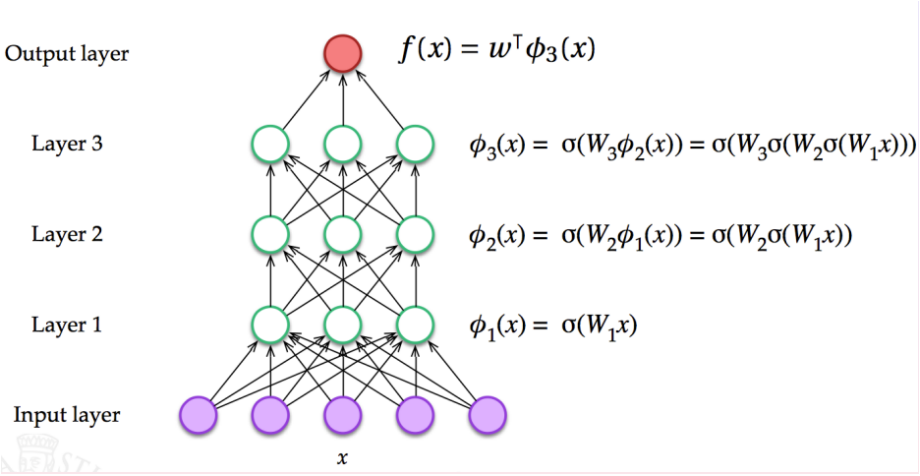


Figure 16.1: Multilayer Perceptron (MLP)

Remark: the neuron in the input layer are not exactly neuron. Indeed, they represent the features of the given input.

Remark: the connections are always from one layer to the following layer. Each neuron has a connection with each of the neurons of the following layer. This structure is called *densely connected architecture*. This is in contrast to the case of convolutional neural networks for example, where the structure of connections is template based.

Remark: on the right of the structure there are the formulas which describe the behaviour of the neural network.

- Note that each W_i is a matrix of weights. For example W_1 represents the weights of the first layer. Each row of the matrix represents the weights of one neuron in the layer. The weights values are the weights of each input of the given neuron. The product $W_1\boldsymbol{x}$ returns a vector which is a value for each of the hidden neurons.
- σ is a non linear function called *activation function* which is applied to the weighted sum of the inputs. Overall, the operation in the first layer is represented by $\phi_1(x)$.

Remark: a beautiful aspect of this structure is the following:

- In non-linear kernel machines the decision function is described by:

$$f(\boldsymbol{x}) = w^T \phi(\boldsymbol{x})$$

In the multilayer perceptron, the final output layer is described by

$$f(\boldsymbol{x}) = w^T \phi_3(\boldsymbol{x})$$

However in the case of multilayer perceptron the function $\phi_3(\boldsymbol{x})$ is not fixed but it is learned. Indeed all the weights of the structure are learned. We have not to design a kernel, we delegate to the neural network the task of defining a proper data representation (more flexibility). The downside is that, in order to learn this representation, a lot of data is needed.

Remark: without σ the decision function would look like a linear function:

$$f(\boldsymbol{x}) = \boldsymbol{w}^T W_3 W_2 W_1 \boldsymbol{x} = W_{\text{all}}^T \boldsymbol{x}$$

16.1 Activation Function

In Figure 16.2 it is illustrated a single perceptron which is composed of a summation and an activation function. The activation function is the *threshold activation*.

$$f(\boldsymbol{x}) = sign(\boldsymbol{w}^T \boldsymbol{x})$$

The derivative of the function is zero everywhere apart from zero, where it is not differentiable. As a result, it is impossible to run gradient-based optimization. Hence, we cannot minimize the error function, there is not chance to propagate a gradient. In conclusion, the threshold activation is not a valid activation function for a deep neural network.

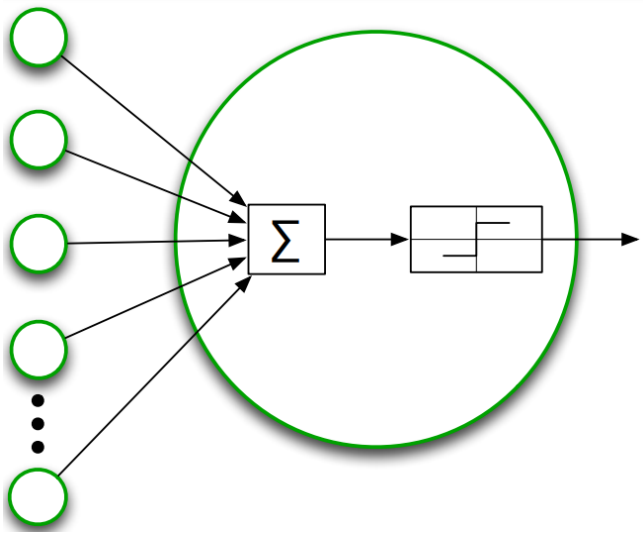


Figure 16.2: Single perceptron.

Remark: $sign(x)$ is equal to +1 if x is positive and $sign(x)$ is equal to -1 if x is negative. This function is clearly non-linear

We need a smoother function. The literature proposed the *sigmoid* function as a reasonable alternative.

$$f(\boldsymbol{x}) = \sigma(\boldsymbol{w}^T\boldsymbol{x}) = \frac{1}{1+\exp(-\boldsymbol{w}^T\boldsymbol{x})} \tag{16.1}$$

The function is illustrated in Figure 16.3. Actually, the curve is a smooth version of threshold. The behaviour is approximately linear around zero and it saturates for large and small values.

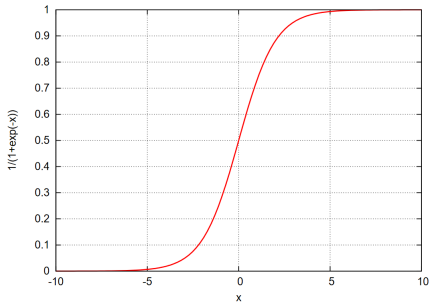


Figure 16.3: Sigmoid activation function.

Remark: this function is derivable and can be plugged into the deep network architecture.

Remark: a problem of this function is that the gradient is almost zero on the left and on the right of its domain. This result in the problem of vanishing gradient.

16.2 Output layer

The deep network architecture can be adopted for many kinds of machine learning problems.

16.2.1 Binary classification

In the case of binary classification we use a single output neuron $o(\boldsymbol{x})$. The sigmoid activation (which allows us to get values between 0 and 1) function becomes:

$$f(\boldsymbol{x}) = \sigma(o(\boldsymbol{x})) = \frac{1}{1+\exp(-o(\boldsymbol{x}))}$$

Then we put a decision threshold at 0.5.

$$y^* = \text{sign}(f(\boldsymbol{x}) - 0.5)$$

16.2.2 Multiclass classification

In the case of multiclass classification we use one output neuron per class (called *logits* layer):

$$[o_1(\boldsymbol{x}), \ldots, o_c(\boldsymbol{x})]$$

We adopt an activation function which ensures that each output neuron outputs a value between zero and one and that the output of the neurons sum to one. Basically, they represent a probability distribution among the possible classes. The way to do that is to use an activation function:

$$f_i(x) = \frac{\exp(o_i(\boldsymbol{x}))}{\sum_{j=1}^{c} \exp(o_j(\boldsymbol{x}))} \tag{16.2}$$

Remark: This is an activation function which is not computed independently for each neuron. It is a layer wise activation function (it requires a normalization step).

Remark: The exponential makes the terms non-negative. Moreover, the normalization ensures that results are between 0 and 1.

The decision is the highest scoring class:

$$y^* = \text{argmax}_{i \in [1,c]} f_i(\boldsymbol{x}) \tag{16.3}$$

16.2.3 Regression

Typically, in the regression case, we take the output of the last layer as it is, using the identity as activation function.
We use one output neuron $o(\boldsymbol{x})$. The activation function is linear, **we delegate to the previous hidden layers the non-linearity of the prediction**. Overall, the decision is the value of output neuron:

$$f(\boldsymbol{x}) = o(\boldsymbol{x}) \tag{16.4}$$

Remark: we do not constraint the values to be in a certain range.

16.3 Representational power of MLP

The purpose of this section is to reason about the representational power of multi-layer perceptrons.

- **boolean functions** Any boolean function can be represented by some network with two layers of units. Indeed, each boolean formula can be written in DNF or CNF form (Figure 16.4). For example we can represent a boolean formula expressed in CNF with one neuron for each clause (OR gate), with negative weights for negated literals and one neuron at the top (AND gate). The problem is that these representations can need an

exponentially big number of terms. Some functions require an exponential number of gates (e.g. parity function). On the other hand, we can express these functions with linear number of gates with a deep network (e.g. combination of XOR gates).

- **continuous functions** Every bounded continuous function can be approximated with arbitrary small error by a network with two layers of units (sigmoid hidden activation, linear output activation).
- **arbitrary functions** any function can be approximated to arbitrary accuracy by a network with three layers of units (sigmoid hidden activation, linear output activation).

At the end of the day we theoretically need few layers in a deep neural network. However if we adopt few layers, the weights become very difficult to learn. Moreover, we would get an exponentially big number of nodes in each of the layers. In practice, we prefer deep structures rather than shallow networks with exponential number of nodes.

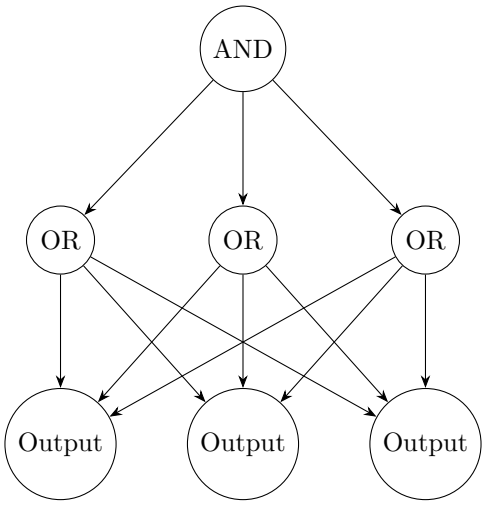


Figure 16.4: Neural network with input layer (AND), hidden layer (OR) with 3 neurons and output layer with 3 neurons.

16.4 Training MLP

Similarly to other machine learning models we need to define a loss function and apply gradient based minimization. Some common choices for loss functions are the followings:

- **Cross entropy** for binary classification ($y \in \{0, 1\}$)

$$\ell(y, f(\boldsymbol{x})) = -(y \log f(\boldsymbol{x}) + (1 - y) \log (1 - f(\boldsymbol{x}))) \tag{16.5}$$

Note that the formula compares the expected output with the actual output of the network. If $y = 1$ then we take into account the term $\log f(\boldsymbol{x})$ otherwise if $y = 0$ we take into account the term $\log(1 - f(\boldsymbol{x}))$. In some ways, the cross entropy tells us the confidence of the given prediction. There is a minus in the front because the higher the value the better the confidence is (we want to minimize). We can extend cross entropy also to the case of multiclass classification as explained in the following point.

- **Cross entropy** for multiclass classification ($y \in [1, c]$):

$$\ell(y, f(\boldsymbol{x})) = -\log f_y(\boldsymbol{x}) \tag{16.6}$$

The minus is to express the cross entropy as an error function instead of a scoring function.

- **Mean squared error** for regression:

$$\ell(y, f(\boldsymbol{x})) = (y - f(\boldsymbol{x}))^2 \tag{16.7}$$

Remark: minimizing cross entropy corresponds to maximizing likelihood.

So, in order to train the neural neural network we identify a proper training error for example (x, y) (e.g. regression):

$$E(W) = \frac{1}{2}(y - f(x))^2 \tag{16.8}$$

and use stochastic gradient descent. Given a learning rate ∇, the gradient update is:

$$w_{lj} = w_{lj} - \eta \frac{\partial E(W)}{\partial w_{lj}} \tag{16.9}$$

We represent a generic weight in the network with w_{lj} representing the weight of the edge which connects node j of one layer with node l of the following layer (Figure 16.5).

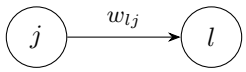


Figure 16.5: $w_{lj} = w_{lj} - \eta \frac{\partial E(W)}{\partial w_{lj}}$

The difficult thing is to compute the partial derivative because the weights could be really far from the output layer where we actually compute the error. The solution of this problem is faced with *backpropagation*. The idea is to use the chain rule for derivation. Each weight of the network is updated according to the error computed at the output layer. The notation used in the following formula refers to Figure 16.6.

$$\frac{\partial E(W)}{\partial w_{li}} = \frac{\partial E(W)}{\partial a_l}\frac{\partial a_l}{w_{lj}} = \delta_l \phi_j \tag{16.10}$$

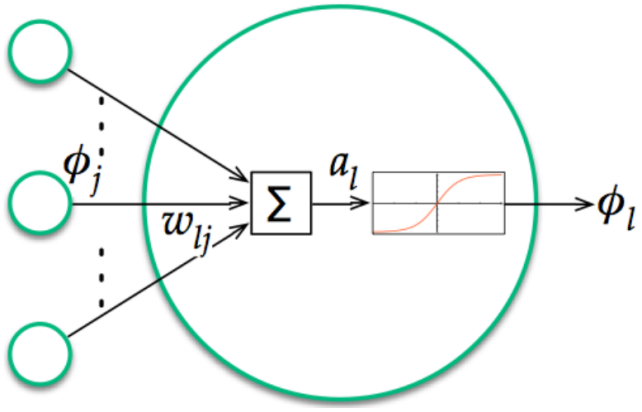


Figure 16.6: The idea of backpropagation is to use the chain rule for derivation.

Remark: the big circle in Figure 16.6 is the node l and ϕ_l is the output of node l.

Looking inside node l we find the weighted sum of the inputs of the node including $\phi_j w_{lj}$. The result of the summation a_l is the input of an arbitrary activation function. The output of node l is ϕ_l.
In Equation 16.10 the aim is to compute the derivative of the error with respect to w_{ij}. According to the chain rule this is equale to $\frac{\partial E(W)}{\partial a_l}\frac{\partial a_l}{w_{lj}}$. a_l is the sum of each input of the layer times its weight. As a result the derivative with respect to the weight w_{lj} is the input ϕ_j.
We represent the fraction $\frac{\partial E(W)}{\partial a_i}$ with δ_l. The computation of δ_l is different if l is a hidden neuron or an output neuron. In what follows, we examine both these cases:

- **Output units** Delta is easy to compute on output units. E.g. for regression with linear outputs:

$$\delta_o = \frac{\partial E(W)}{\partial a_0} = \frac{\partial \frac{1}{2}(y - f(x))^2}{\partial a_0}$$

  Actually $f(x)$ is the result of the application of the non linearity on a_o. However, as we said above, in regression we typically do not apply non-linearity at the output layer. As a consequence in this case $f(x) = a_o$.

$$\delta_o = \frac{\partial \frac{1}{2}(y - a_o)^2}{\partial a_0} = -(y - a_0)$$

This value allows us to compute the derivative of the error for the weights of the last layer.

$$\frac{\partial E(W)}{\partial w_{oj}} = \delta_o \phi_j$$

- **Hidden units** here we consider the contribution to error through all outer connections (sigmoid activation). Here l is not an output node but an internal node. The error E is computed at the very top of the graph illustrated in Figure 16.7, in the red node. As a result we have to backpropagate this error through the network structure.
The derivative of the error $\frac{\partial E(W)}{\partial a_l}$ is decomposed into the derivative of the error with respect to each of the nodes which follow l.

$$\delta_l = \frac{\partial E(W)}{\partial a_l} = \sum_{k \in \text{ch}[l]} \frac{\partial E(W)}{\partial a_k} \frac{\partial a_k}{\partial a_l}$$

Intuitively we are calculating the contribution of l to the error at the end of the structure.
Now we notice that the $\frac{\partial E(W)}{\partial a_k}$ is actually δ_k (recursion). Moreover, remember that the output of l is represented with ϕ_l.

$$\delta_l = \sum_{k \in \text{ch}[l]} \frac{\partial E(W)}{\partial a_k} \frac{\partial a_k}{\partial a_l} = \sum_{k \in \text{ch}[l]} \delta_k \frac{\partial a_k}{\partial \phi_l} \frac{\partial \phi_l}{\partial a_l}$$

We assign weight w_{kl} to the edge (k, l). As a consequence $\frac{\partial a_k}{\partial \phi_l} = w_{kl}$. Moreover, $\frac{\partial \phi_l}{\partial a_l}$ is essentially the derivation of the activation function with respect to its input.

$$\delta_l = \sum_{k \in \text{ch}[l]} \delta_k \frac{\partial a_k}{\partial \phi_l} \frac{\partial \phi_l}{\partial a_l} = \sum_{k \in \text{ch}[l]} \delta_k w_{kl} \frac{\partial \sigma(a_l)}{\partial a_l}$$

In this case we take into account the sigmoid activation function.

$$\delta_l = \sum_{k \in \text{ch}[l]} \delta_k w_{kl} \frac{\partial \sigma(a_l)}{\partial a_l} = \sum_{k \in \text{ch}[l]} \delta_k w_{kl} \sigma(a_l)(1 - \sigma(a_l))$$

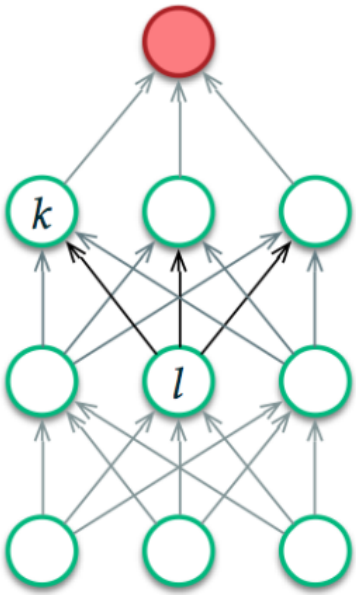


Figure 16.7: Consider contribution to error through all outer connection.

In deep neural network, the neural architectures are really seen as combinations of modules (Figure 16.8). Each of these modules has an interface to interact with the outside. The aim is to combine modules in order to construct a deep architecture. In Figure 16.8 we understand how this architecture looks like and how it is updated.

- We have a first layer $F_1(x, W_1)$ which takes input x and weight matrix W_1 and computes ϕ_1.
- The second layer $F_2(\phi_1, W_2)$ takes as input ϕ_1 and weight matrix W_2 and computes the new representation of the input ϕ_2.
- In the output layer we apply a loss function $\text{Loss}(\phi_3, y)$. With this module we compute the error for having predicted a certain value instead of y. With this error value we can compute $\frac{\partial E}{\partial \sigma_3}$.
- At this point, the $\frac{\partial E}{\partial W_j}$ can be decomposed as:

$$\frac{\partial E}{\partial W_j} = \frac{\partial E}{\partial \phi_j}\frac{\partial \phi_j}{\partial W_j} = \frac{\partial E}{\partial \phi_j}\frac{\partial F_j(\phi_{j-1}, W_j)}{\partial W_j}$$

  At the very end of the chain the output layer produces $\frac{\partial E}{\partial \phi_j}$. However, each arbitrary layer needs this information from the following layer. Each layer should be able to propagate this information to the layer below. So,

each layer can compute $\frac{\partial E}{\partial \phi_{j-1}}$. For example layer three in Figure 16.8 is able to compute $\frac{\partial E}{\partial \phi_2}$.

$$\frac{\partial E}{\partial \phi_{j-1}} = \frac{\partial E}{\partial \phi_j}\frac{\partial \phi_j}{\partial \phi_{j-1}} = \frac{\partial E}{\partial \phi_j}\frac{\partial F_j(\phi_{j-1}, W_j)}{\partial \phi_{j-1}}$$

- Each module should be able to compute the derivative of its output with respect to its weights ($\frac{\partial F_j(\phi_{j-1}, W_j)}{\partial W_j}$). Moreover, each module should be able to compute the derivative of its output with respect to its inputs ($\frac{\partial F_j(\phi_{j-1}, W_j)}{\partial \phi_{j-1}}$). The former is used by the module to update its own weights and the latter is used by the module to backpropagate the error. This allows to following layers down in the hierarchy to update their own weights.

Overall, this is a modular architecture: we do not care about what F_j does internally.

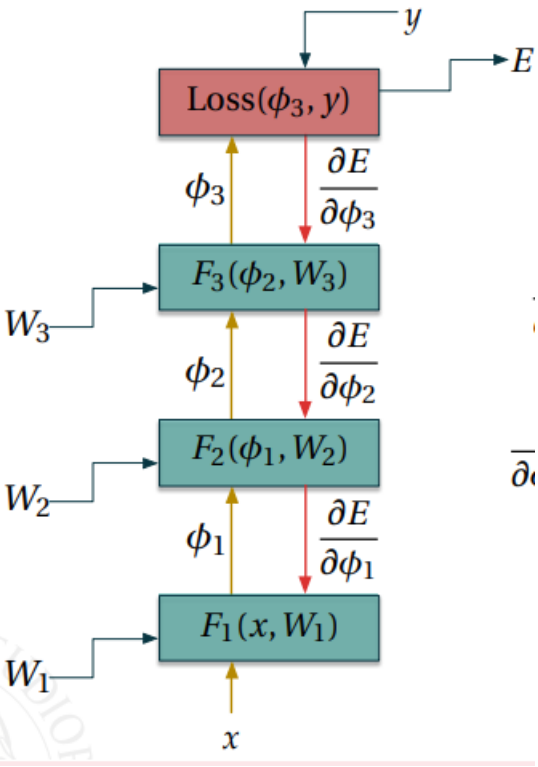


Figure 16.8: In deep neural network, the neural architectures are really seen as combinations of modules.

At the end of the day with backpropagation we perform gradient descent. Hence, backpropagation is only guaranteed to converge to a local minimum. Moreover, the error surface of a multilayer neural network can contain several minima. The literature proposes heuristic attempts to address the problem (the problem in question is that backpropagation is only guaranteed to converge to a local minimum):

- use stochastic instead of true gradient descent
- train multiple networks with different random weights and average or choose best
- random restart of the weights
- many more...

Remark: training kernel machines requires solving quadratic optimization problems. As a result, finding the global optimum is guaranteed (convex problem). On the other hand deep networks are more expressive in principle, but harder to train.

16.4.1 Stopping criterion and generalization

Commonly the validation set is used to choose the best configuration of the hyperparameters. In deep learning we can still use the validation set to configure the hyperparameters. However, in deep learning we can use the validation set to choose the training termination condition. Actually, overtraining the network increases possibility of overfitting training data (we could progressively fit noise penalizing generalization). Neural network models are really expressive, which means that they are complex, which means that they are prone to overfitting. In particular, overfitting occurs at later iterations, when increasingly complex surfaces are being generated. Note that at the beginning, the network is initialized with small random weights and so the decision surface is very simple. Overall, the aim is to stop learning before starting learning the noise. To do this, we use a separate validation set to estimate performance of the network and choose when to stop training. As we can seen in Figure 16.9, the error on the validation set at some point (when the model begins to overfit) starts to increase. In practice, we keep track of the validation error; we periodically save the best weights seen in the validation set; we stop learning when we notice that the validation error starts to increase. Probably in the case of Figure 16.9 we keep learning until epoch 9 and then we keep the weights learned at epoch 5. This technique is referred to as *early stopping*.

Remark: in backpropagation we typically do not take into account all the examples of the training set in order to compute the error in the output layer and update the weights. Typically, we perform as an alternative, mini-batch gradient descent.

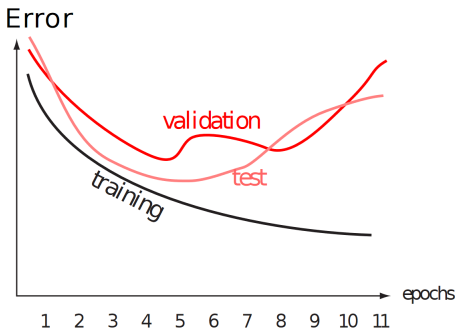


Figure 16.9: Training error, validation error, test error in deep learning.

16.4.2 The problem of vanishing gradient

Training a neural network is not a convex problem, hence there are more things that might go wrong than SVMs. The problem of vanishing gradient together with the lack of training data and the limited amount of computation power are perhaps the most common reasons why deep learning has become a hot research topic only in the last ten years. As we discussed above, the main idea of back-propagation is that the error gradient (somehow a signal) is backpropagated through layers. At each step gradient is multiplied by derivative of sigmoid. A problem of the sigmoid function is that it saturates for small values and large values of its input. The gradient is informative where the sigmoid behaves almost linearly, but it is zero or close to zero in the saturation areas. At some point of the training process we might start propagating values of the gradient which are almost zero. This problem is known as *vanishing gradient*: the gradient vanishes in lower layers.

In the following we present few simple suggestions:

- Do not initialize weights to zero, but to small random values around zero. In this way we somehow inject diversity inside the network avoiding that all the neurons learn in the same way.
- It is possible to have to deal with datasets characterized by inputs with very different range. Some numbers between 0 and 100, some other numbers between -10000 and 100000 for instance. This results in a high probability to fall in the saturation area of the activation functions or in very high (or very small) values of the weights to balance the structure. A valid trick is to standardize inputs to avoid saturating hidden units:

$$x' = \frac{(x - \mu_x)}{\sigma_x}$$

In order to standardize an input, for each possible feature, we have to subtract the mean and divide by the standard deviation. In this way each feature is 0 mean and 1 standard deviation.

- Randomly shuffle training examples before each training epoch. In this way we ensure that at each epoch examples are presented in a different order.

Perhaps, one of the main reasons why this architecture started to be trainable is the realization of the community that the saturated areas of the sigmoid were the main caveats for which backpropagation did not work properly. As a result the literature came out with an alternative (still non linear) activation function. This last is called *ReLU* (rectified linear unit):

$$f(x) = \max(0, \boldsymbol{w}^T \boldsymbol{x}) \tag{16.11}$$

Remark: Essentially, the ReLU returns the maximum between zero and the input of the activation function ($\boldsymbol{w}^T \boldsymbol{x}$) where $\boldsymbol{x}$ is the input of the neuron.

Remark: somehow this function is similar to the hinge loss.

As we can see in Figure 16.10 this function brings everything to zero if the input is negative, while on the right is linear. The function saturates only on the left. As a consequence we have more chance to avoid vanishing gradient problem.

To sum up:

- Linearity is nice for learning
- Saturation (as in sigmoid) is bad for learning (gradient vanishes implies no weight update)
- Neuron employing rectifier activation called rectified linear unit (ReLU)

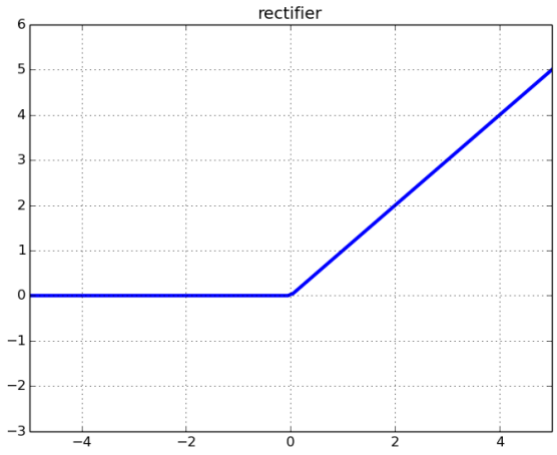


Figure 16.10: Rectified Linear Unit (ReLU)

Another trick of the trade to face the problems related to the training of a deep neural network is regularization. As discussed above in the text, the aim of regularization theory is to minimize both model complexity and training error. So the objective function ($J(W)$) is composed by two terms: the error ($E(W)$) on the training set and the complexity of the model ($\lambda||W||_2$) with the tradeoff parameter λ:

$$J(W) = E(W) + \lambda||W||_2$$

For the sake of simplicity, in Figure 16.11 we assume to have two weights (W_1 and W_2). The Euclidean norm (or 2-norm) of the weights is something that is minimal at zero and grows in circles. This means that points on the same circle have the same norm, they are indistinguishable with respect to the minimization of the regularization term $||W||_2$. Then, suppose the error function is an ellipsoid which is minimal in the center. Depending on the value of λ we decide a tradeoff between the two objectives to minimize. With this procedure we end up choosing a point where the regularization circle touches the error ellipsoid.

In essence *2-norm regularization* penalizes weights by Euclidean norm. Weights with less influence on error get smaller values.

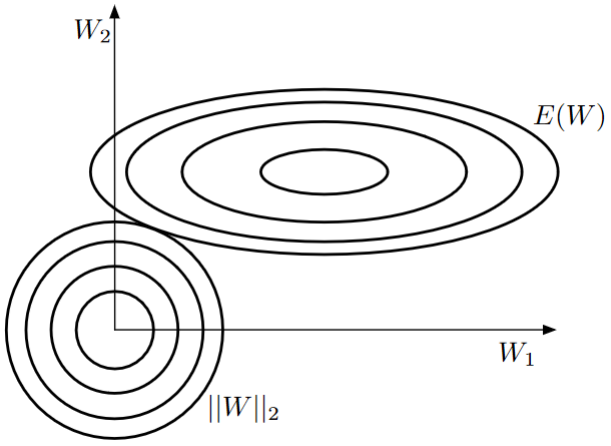


Figure 16.11: Regularization allows us to penalize weights by Euclidean norm: weights with less influence on error get smaller values.

There is an alternative form of regularization that can be used which is *1-norm regularization* (Figure 16.12):

$$J(W) = E(W) + \lambda|W|$$

In this case the regularization penalizes weights by sum of absolute values ($|w| = \sum_i |w_i|$). In this case the ball around zero is no more a circle but

something like a square. Of course, points on the same square have the same norm. This encourages less relevant weights to be exactly zero (sparsity inducing norm). Indeed, in the previous case the circle touches the ellipsoid at a random point, on the other hand in this case the square is more likely to touch the ellipsoid at the vertices. Note that if we touch the square at a vertex some of the weights are zero. This is called *sparsifying norm*: a norm which encourages sparse solutions. This kind of regularization can be useful when we want to minimize the number of weights in the solution for instance for interpretability or for feature selection.

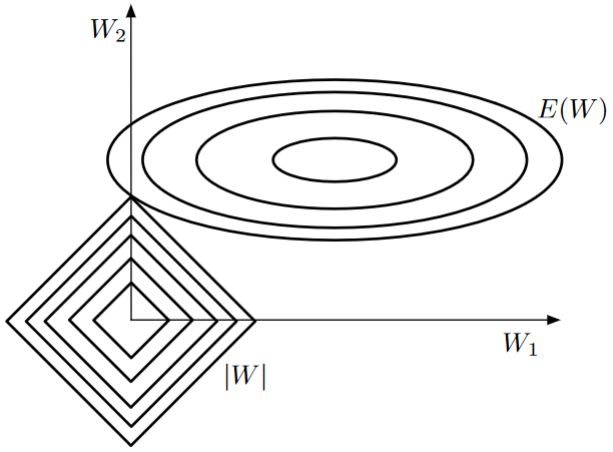


Figure 16.12: Regularization penalizes weights by sum of absolute values. This encourages less relevant weights to be exactly zero (sparsity inducing norm).

Another trick of the trade to face the problems related to the training of a deep neural network is initialization. First of all it is important to randomly initialize weights. In this way we break symmetries between neurons. Indeed, if we initialize all weights to zero they might all learn the same thing becoming redundant. This would reduce the overall expressiveness of the network. In order to carefully set the initialization range to preserve forward and backward variance we use the following formula:

$$W_{ij} \sim U(-\frac{\sqrt{6}}{\sqrt{n+m}}, \frac{\sqrt{6}}{\sqrt{n+m}})$$

where n and m are number of inputs and outputs. Overall, the aim is to ensure sparse initialization: enforce a fraction of weights to be non-zero (this encourages diversity between neurons).

Another trick of the trade to face the problems related to the training of a deep neural network is minibatch gradient descent. Batch gradient descent updates parameters after seeing all examples (of the training set). This procedure is too slow for large datasets. On the other hand, full stochastic gradient descent updates parameters after seeing each example. In a large network, the error that we compute for a single example can be quite different from the error that is averaged over many examples. Overall, the objective would be too different from the true one. A valid solution is an intermediate technique between batch and fully stochastic gradient descent. *Minibatch* gradient descent divides the training set into minibatches and updates parameters after seeing a minibatch of m examples. The proper value of m depends on many factor, e.g. size of GPU memory. Indeed, our architectural consideration is that we do not want a minibatch too large which does not fit the GPU memory.

Another trick of the trade to face the problems related to the training of a deep neural network is a proper configuration of the learning rate. With this aim an popular approach is the usage of the *momentum* concept. The idea is to update the weights not only depending on the current direction of the gradient but also partially according to the directions of the previous iterations. The tradeoff is regulated by a coefficient α called momentum.

$$v_{ji} = \alpha v_{ji} - \eta \frac{\partial E(W)}{\partial w_l j}$$

$$\boldsymbol{w}_{ji} = w_{ji} + v_{ji}$$

where $0 \leq \alpha < 1$. The adoption of the momentum tends to keep the updating of the weights in the same direction. Think of a ball rolling on an error surface. The possible effects could be:

- roll through small local minima without stopping
- traverse flat surfaces instead of stopping there
- increase step size of search in regions of constant gradient

Another trick of the trade to face the problems related to the training of a deep neural network is the usage an adaptive (decreasing) learning rate. The idea is to use larger learning rate at the beginning (when we are far from the optimum) for faster convergence towards attraction basin. Then, smaller learning rate at the end to avoid oscillation close to the minimum. Indeed with too big oscillation, we could miss the optimum.

$$\eta_t = \begin{cases} (1 - \frac{t}{\tau})\eta_0 + \frac{t}{\tau}\eta_\tau & \text{if } t < \tau \\ \eta_\tau & \text{otherwise} \end{cases}$$

In the formula, after a certain number of iterations τ, η_t becomes η_τ. Before, the learning rate is a combination of the two. While the procedure progressively

reaches the optimum, the learning rate progressively goes from η_0 to η_τ.

Another trick of the trade to face the problems related to the training of a deep neural network is related to the *Adagrad* approach.

$$r_{ji} = r_{ji} + (\frac{\partial E(W)}{\partial w_{lj}})^2$$

$$\boldsymbol{w}_{ji} = w_{ji} - \frac{\eta}{\sqrt{r_{ji}}} \frac{\partial E(W)}{\partial w_{lj}}$$

Using momentum we vary the learning rate only with respect to time, but the learning rate value is the same for all the directions. In high-dimensional problems, the behaviour of the function can be very different in each direction. From this, we understand that using the same learning rate for all directions is clearly suboptimal. The idea of the adagrad approach is to develop an adaptive gradient such that:

- Reduce learning rate in steep directions
- Increase learning rate in gentler directions

Intuitively r_{ji} accumulates the square of the partial derivatives for a certain dimension. This means that if a certain dimension has a high value of the derivative for a lot of iterations, the function is steep along that dimension. So, the learning rate is divided by this size of the gradient ($\frac{\eta}{\sqrt{r_{ji}}}$). (The square root is needed to scale the value in the order of the weights, avoiding the square term).
Overall, adagrad is actually a good solution to speed up convergence in high dimensional spaces. However, we can highlight a problem of this procedure:

- The square gradient is accumulated over all iterations. This could lead to non-local decisions. This means that the reduction of the learning rate could become too large at some point. For non-convex problems, learning rate reduction can be excessive/premature.

To solve this problem, the literature propose an alternative called *RMSProp*.

$$r_{ji} = \rho r_{ji} + (1-\rho)(\frac{\partial E(W)}{\partial w_{ij}})^2$$

$$\boldsymbol{w}_{ji} = w_{ji} - \frac{\eta}{\sqrt{r_{ji}}} \frac{\partial E(W)}{\partial w_{lj}}$$

The most relevant difference with respect to the previous approach is that we calculate a linear combination ($\rho r_{ji} + (1-\rho)(\frac{\partial E(W)}{\partial w_{ij}})^2$) of the current value and the new squared gradient. In this way, we obtain an exponentially decaying accumulation of squared gradient ($0 < \rho < 1$). The squared gradient is exponentially decaying with respect to how far we are from the current position. So,

old values of the gradient do not count anymore in order to decide whether to slow down or not. In this way the update of the learning rate becomes more local. Doing this, RMSProp avoids the premature reduction of Adagrad and the respective slowing down. Poi sulle slide c'è scritto Adgrad-like behaviour when reaching convex bowl, ma non so bene cosa si intenda.

The update rule for neural networks which is the defacto standard is a variant of the methods we have discussed, which is called ADAM.

Another trick of the trade to face the problems related to the training of a deep neural network is *batch normalization*. This concept is related to the so called *covariate shift problem*. Covariate shift problem is when the input distribution to your model changes over time and the model does not adapt to the change. The fact that the input distribution changes over time is problematic. Indeed, when we train a model we typically assume that examples are independent and identically distributed (i.i.d.). If we consider examples whose distribution changes over time, they are no more identically distributed. The learning model should adapt continuously to the change of the data. In (very) deep networks, internal covariate shift takes place among layers when they get updated by backpropagation. Basically, the update of a layer refers to a representation of the data which is old with respect to the current collection of examples. This leads to a slower convergence: we need to keep adjusting the weights according to the changes in the network.

Remark: according to the web the term covariate refers to an independent variable that can influence the outcome of a given statistical trial, but which is not of direct interest. Other definitions more related to the topic at hand are welcome.

Batch normalization is a solution proposed by the literature to solve this problem. The idea is to normalize each node activation (input to activation function) by its batch statistics in order to keep it always in the same range. This prevents the covariate shift since for each layer the activation to the activation function stays in the same range.

$$\hat{x_i} = \frac{x_i - \mu_B}{\sigma_B}$$

where:

- x is the activation of an arbitrary node in an arbitrary layer
- $\mathcal{B} = \{x_1, \ldots, x_m\}$, is a batch of values for that activation
- μ_β, σ_B^2 are batch mean and variance (for each neuron)

Keeping the activation always in the same range is in contrast with respect to the aim of learning a suitable configuration of the deep neural network. Perhaps,

we could end up to concentrate the activation in the same range too much. To avoid this we combine the normalization with scale and shift parameters (γ, β).

$$y_i = \gamma \hat{x}_i + \beta$$

In this manner, we scale and shift each activation with adjustable parameters. In this way the standardized activation is adjusted before being processed by the activation function.

Remark: the scale γ and shift β coefficients become part of the network parameters and they have to be learnt.

Some advantages of batch normalization are:

- More robustness of parameter initialization
- Allows for faster learning rates without divergence
- Keeps activations in non-saturated region even for saturating activation functions. In this sense, in some cases, we could use the sigmoid activation function if we adopt batch normalization
- Regularizes the model

Another trick of the trade to face the problems related to the training of a deep neural network is *layerwise pre-training*. Modern solution for pre-training are:

- Supervised pre-training: layerwise training with actual labels
- Transfer learning: train network on similar task, discard last layers and retrain on target task
- Multi-level supervision: auxiliary output nodes at intermediate layers to speed up learning

Note: these methods were useful when the vanishing of the gradient was a major problem (no relu, no best weight initialization strategies, etc.)

16.5 Many different architectures

At this point we overview the main deep neural network architectures as an alternative of the multilayer perceptron. In this context, the literature proposes many different architectures:

- *convolutional networks* for exploiting local correlations (e.g. for images).
- *recurrent* and *recursive* networks for collective predictions (e.g. sequential labelling). This model are useful for inputs which are not vectors but sequences (sequential data) or other structures (e.g. graph neural network).

- *deep Boltzmann machines* as probabilistic generative models (can also generate new instances of a certain class). Another example of deep generative model are variatonal autoencoders. These kind of generative models do not need to model directly the probabilistic relationships between single variables in terms of conditional probability distributions as we studied in the case of Bayesian networks.
- *generative adversarial networks* to generate new instances as a game between discriminator and generator. In general, the aim is to learn a model which is able to generate objects which are similar to the ones which are seen in input.

16.5.1 Autoencoder

The *autoencoder* is a deep neural architecure whose purpose is to train shallow network to reproduce input in the output. In other words, the autoencoder is trained to recunstruct its intput. As we can see in Figure 16.13 the structure has the same number of inputs and outputs. Moreover, we can notice that there is only one internal layer, called *coding layer*.
Once the autoencoder is trained, its purpose is to process x as input and provide x as output. This process can be done with unlabelled examples in a *unsupervised learning* fashion.
As an example, one can structure the code layer in such a way that handwritten digits are encoded in the hidden layer with a binary representation. In general, the aim of the autoencoder is to learn to map inputs into a sensible hidden representation (*representation learning*). From this representation it should be possible to reconstruct the original input. As a consequence, the autoencoder is often used to map data in a different dimensional space. Another typical application of autoencoders is to generate data from the representation that has been learned. Autoencoders were also adopted to perform layerwise pre-training (Figure 16.14, Figure 16.15):

1. train the autoencoder
2. discard the output layer
3. freeze hidden layer weights
4. add another hidden layer and another output layer
5. train network to reproduce input

In this way we learn a representation of the input which is more abstract than the first one, but that can still reconstruct the original input.
We can use the output of this approach (as deep as we want) in supervised learning task:

1. discard autoencoder output layer

2. add appropriate output layer for supervised task (e.g. one-hot encoding for multiclass classification)
3. learn output layer weights and refine all internal weights by backpropagation algorithm

Intuitively, the hidden layers of the final network are designed to conserve as much information of the input as possible.

Remark: studying autoencoders we can highlight a crucial difference with respect to kernel machines. With autoencoders the model learns a suitable data representation, on the other hand kernel machines propose an engineered representation $\phi(x)$ which is given and not learnt. In general, in neural network the architecture is engineered while the feature mapping is learnt.

Nowadays, autoencoders are no more used to perform layerwise pre-training.

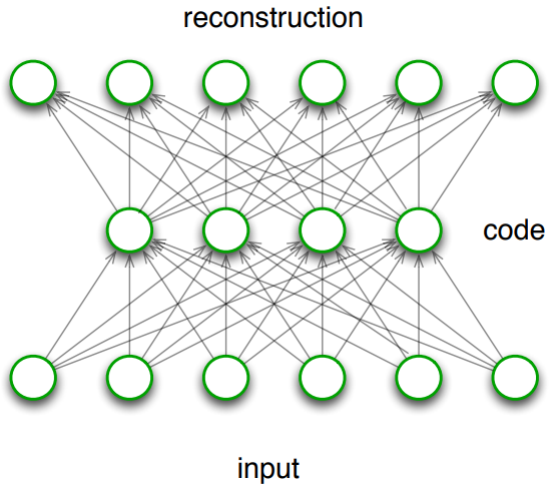


Figure 16.13: Autoencoder

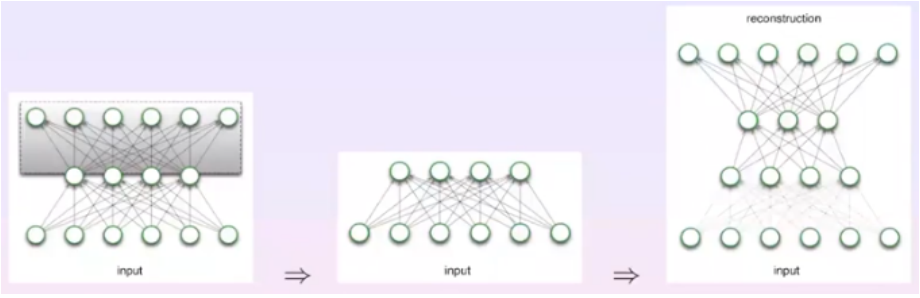


Figure 16.14: Layerwise pre-training with autoencoder 1.

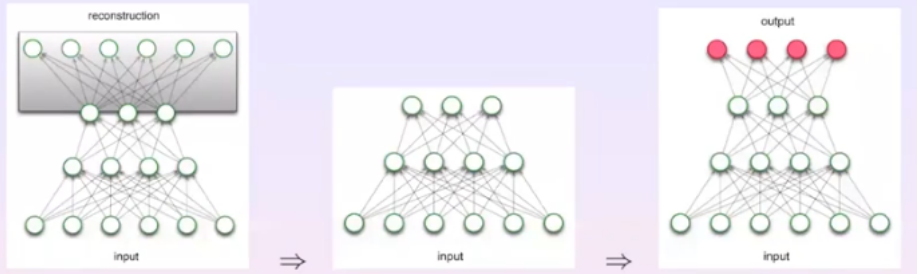


Figure 16.15: Layerwise pre-training with autoencoder 2.

16.5.2 Convolutional network (CNN)

Convolutional neural networks (Figure 16.16) are a consolidated architecture in the field of image processing and image classification. Images are seen as two dimensional matrices of pixels or as multidimensional objects. For example, we can imagine an image as a structure composed by an additional channel representing image color (we often use a matrix for each channel). CNNs allow to understand correlations among spatially closed pixels. Moreover, convolutional neural network has to implement location invariance (e.g. translation inside the image): if we detect a boat, its position inside the image does not affect the classification output. In order to achieve these purposes CNNs use *convolution filters*. These are matrices of weights which are used to scan the image from the very top down to the end. Each input portion of the image inside the matrix is combined (aggregated) into one value. A convolutional neural network can adopt more than one filter. Every filter produces its own representation for the given image. The purpose of convolution filters is to extract local features.
In addition to convolution, CNNs perform also pooling to provide invariance to local variations. Pooling allow us to combine the local prediction in a small area, and this leads to more robustness against translations for example. A common pooling implementation is *max pooling* where we take the maximum value of the matrix in a certain area. The maximum value is used to produce a new smaller representation of the matrix.
Overall, convolution filters and pooling are the two main components of a CNN such that:

- filters should learn to identify patterns inside the input image
- the pooling process tries to understand if in the given area it has been found something

A combination of filters and convolution layers is repeated along the network in order to progressively find higher level patterns. The first layers understand simple patterns like straight lines; subsequently, further layers can combine patterns to construct more complex and higher level patterns. In this way we obtain a hierarchy of filters to compose complex features from simpler ones (e.g. pixels

to edges to shapes). All these matrices are learnt in an end to end fashion. At some point the convolutional neural network ends with a multi layer perceptron architecture: all the features are placed in a vector and processed with densely fully connected layers. Finally, at the output level we reach a representation which allows us to perform the (e.g. classification) task.

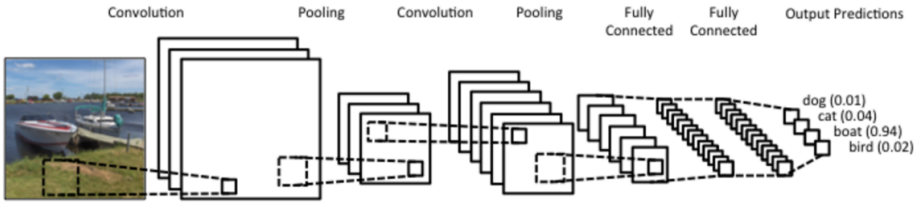


Figure 16.16: Convolutional neural network.

16.5.3 Long Short-Term Memory Networks (LSMT)

Recurrent neural networks allow to make predictions about objects which are not fixed size vectors but for example sequences. We discussed about kernel techniques which are used to process sequences. In this subsection we introduce a template based model which is used to process sequences to obtain representations which are suitable for learning. As illustrated in Figure 16.17 the architecture is composed by the same unit repeated a number of times equal to the number of inputs. Each unit processes an input obtaining an output which is sent to the following unit. The result is a propagation of information from input to output. Looking inside an unit:

- σ are parametric layers which end with the sigmoid. The sigmoid function restricts the input in a range between zero and one. We can see this value as a probabilistic value.
- Somewhere inside the unit products are computed. In Figure 16.17 they are are represented with a X. These are element wise products.
- These operations allow to learn a state inside the unit and propagate it along the network.
- During the multiplications, if the value (I think of the sigmoid) for a certain dimension is zero, we reset that dimension (I think the state associated with that dimension). On the other hand, if the value (I think of the sigmoid) is one we keep that dimension as it is. Then, if the value (I think of the sigmoid) is between zero and one, the dimension is down-weighted.

At the end of the day we obtain a recurrent computation with selective memory such that:

- a cell state is propagated along the chain

- a forget gate selectively forgets parts of the cell state. Implicitly we also select which part of the state to remember to go to the next state.
- an input gate selectively chooses parts of the candidate (i.e. input) for cell update (i.e. update the state, actually, after the multiplication there is a plus operation). In essence, the same concept of the forget gate is applied to the input itself. Indeed, in Figure 16.17 we can notice a sigma layer which is multiplied with the input.
- Finally output gate selectively chooses parts of the cell state for output

Inside a cell, the overall state is a combination of the part of the previous state which has been not forgotten (this is represented by the first X inside the second network unit in Figure 16.17) and a selected part of the information that we get from the input (this is represented by the plus after the first X in Figure 16.17). This combination produces the updated state which is propagated to the next unit. Before being propagated, this state is combined with the output gate (that depends also on the input, look at the arrow from σ in the top right of the figure to output in Figure 16.17) to decide what parts of the cell state have to be sent as the output of the given unit. Moreover, the output of the current unit will become part of the combination of the local input and the propagated state of the following unit.
In Figure 16.17, letter h at each unit is the representation at each state that has been learnt. According to the task at hand, we can add (for example) a classification or a regression layer on top of h (basically everything we need to perform the final prediction).

The most relevant thing of this gate architecture is that all the gates are parametric, are learnt. In this way, the network decides what to remember, what to use to update, what to output, at each unit. This kind of architecture is commonly referred to as *long short-term memory* because it adaptively decides what to remember. We can use this structure for example to avoid forgetting a signal which is propagated too long in time. The selective memory can decide what to remember, simplifying the task of remembering something for a long time. In this sense, this architecture works well in sequential prediction tasks.

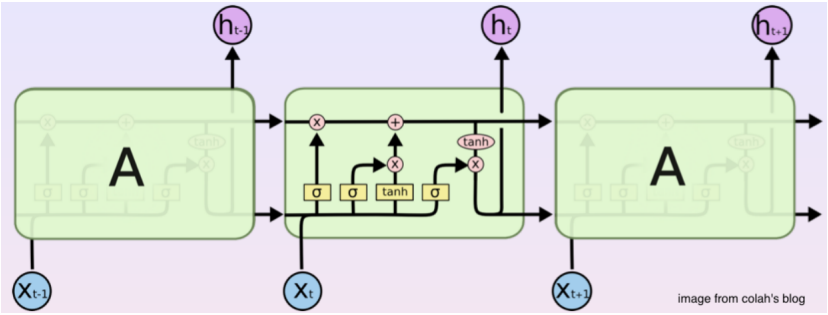


Figure 16.17: Long Short-Term Memory Networks (LSMT).

16.5.4 Generative Adversarial Networks (GAN)

Another popular deep architecture is the *generative adversarial network* (Figure 16.18). The goal of GANs is to generate new objects. The task of generating new data can become extremely complicated when dealing with high dimensions. The architecure is composed by:

- a *generator* network which learns to generate items (e.g. images) from random noise. The generator is trained in order to fool the discriminator. Sometimes we refer to the generator as the *forger*.
- a *discriminator* network which learns to distinguish between real items and generated ones. The discriminator is trained taking as input a dataset of real images (positive examples) and images generated by the generator neural network (negative examples). Sometimes we refer to the generator as the *detective*.

The two networks are jointly learnt as in an adversarial game. Along the process no human supervision is needed.

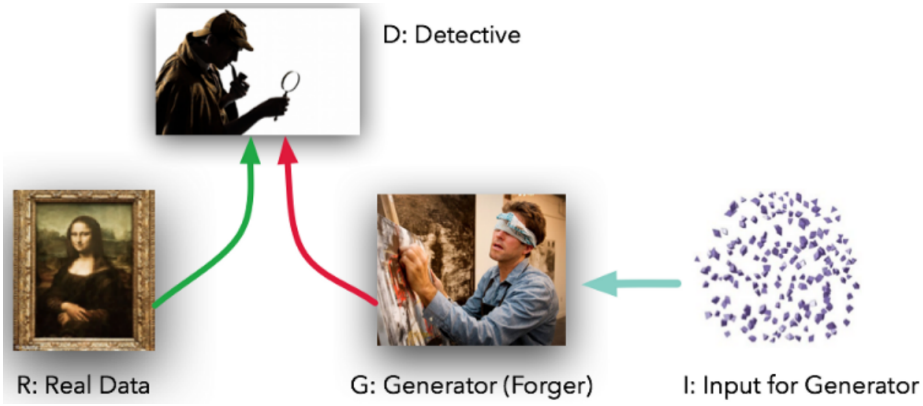


Figure 16.18: Generative Adversarial Networks (GAN).

16.5.5 Transformers

In this subsection we write about a particular neural network model which has been proven to be especially effective for common natural language processing tasks. The model is called *Transformer* and it makes use of several methods and mechanisms that we introduce here (Figure 16.19).

16.5.5.1 Sequence to Sequence Learning Attention

The paper *Attention is all you Need* describes transformers and what is called a sequence-to-sequence architecture. Sequence-to-Sequence (or Seq2Seq) is a neural network that transforms a given sequence of elements, such as the sequence

of words in a sentence, into another sequence. Seq2Seq models are particularly good at translation, where the sequence of words from one language is transformed into a sequence of different words in another language. A popular choice for this type of model is Long-Short-Term-Memory (LSTM)-based models. With sequence-dependent data, the LSTM modules can give meaning to the sequence while remembering (or forgetting the parts it finds important (or unimportant). Sentences, for example, are sequence-dependent since the order of the words is crucial for understanding the sentence. LSTM are a natural choice for this type of data.

Seq2Seq models consist of an Encoder and a Decoder. The Encoder takes the input sequence and maps it into a higher dimensional space (n-dimensional vector). That abstract vector is fed into the Decoder which turns it into an output sequence. The output sequence can be in another language, symbols, a copy of the input, etc.

Imagine the Encoder and Decoder as human translators who can speak only two languages. Their first language is their mother tongue, which differs between both of them (e.g. German and French) and their second language an imaginary one they have in common. To translate German into French, the Encoder converts the German sentence into the other language it knows, namely the imaginary language. Since the Decoder is able to read the imaginary language, it can now translates from that language into French. Together, the model (consisting of Encoder and Decoder) can translate German into French!

Suppose that, initially, neither the Encoder nor the Decoder of the Seq2Seq model is very fluent in the imaginary language. To learn it, we train them (the model) on a lot of examples.

A very basic choice for Encoder and the Decoder of the Seq2Seq model is a single LSTM for each of them.

We need one more technical detail to make the Transformers easier to understand: *Attention*. The attention-mechanism looks at an input sequence and decides at each step which other parts of the sequence are important. It sounds abstract, but let us clarify with an easy example: when reading this text, you always focus on the word you read but at the same time your mind still holds the important keywords of the text in memory in order to provide context. An attention-mechanism works similarly for a given sequence. For our example with the human Encoder and Decoder, imagine that instead of only writing down the translation of the sentence in the imaginary language, the Encoder also writes down keywords that are important to the semantics of the sentence, and gives them to the Decoder in addition to the regular translation. Those new keywords make the translation much easier for the Decoder because it knows what parts of the sentence are important and which key terms give the sentence context.

In other words, for each input that the LSTM (Encored) reads, the attention-mechanism takes into account several other inputs at the same time and decides which ones are important by attributing different weights to those inputs. The Decoder will them take as input the encoded sentence and the weighs provided by the attention-mechanism.

16.5.5.2 The Transformer

The paper *Attention is All You Need* introduces a novel architecture called Transformer. As the title indicates, it uses the attention-mechanism we saw earlier. Like LSTM, Transformer is an architecture for transforming one sequence into another one with the help of two parts (Encoder and Decoder), but it differs from the previously described/exisiting sequence-to-sequence models because it does not imply any Recurrent Networks (GRU, LSTM, etc.). *Recurrent Networks* were, until now, one of the best ways to capture the timely dependencies in sequences. However, the team presenting the paper proved that an architecture with only attention-mechanisms without any RNN (Recurrent Neural Network) can improve on the results in translation task and other tasks. The purpose of this subsection is to describe what a Transformer is. For the sake of clarity, look at the structure depicted in Figure 16.20.

The Encored is on the left and the Decoder is on the right. Both Encoder and Decoder are composed of modules that can be stacked on top of each other multiple times, which is described by Nx in the figure. We see that the modules consist mainly of Multi-Head Attention and Feed Forward layers. The inputs and outputs (target sentences) are first embedded into an n-dimensional space since we cannot use strings directly.
One slight but important part of the model is the positional encoding of the different words. Since we have no recurrent networks that can remember how sequences are fed into a model, we need to somehow give every word/part in our sequence a relative position since a sequence depends on the order of its elements. These positions are added to the embedded representation (n-dimensional vector)of each word.

In Figure 16.21 we take a closer look at the Multi-Head Attention bricks. Let's start with the left description of the attention-mechanism. It's not very complicated and can be described by the following equation:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{16.12}$$

Q is a matrix that contains the query (vector representation of one word in the sequence), K are all the keys (vector representations of all the words in the sequence) and V are the values, which are again the vector representations of all the words in the sequence. For the encoder and decoder, multi-head attention modules, V consists of the same word sequence than Q. However, for the attention module that is taking into account the encoder and the decoder

sequence, V is different from the sequence represented by Q.
To simplify this a little bit, we could say that the values in V are multiplied and summed with some attention-weights a, where our weights are defined by:

$$a = \text{softmax}(\frac{QK^T}{\sqrt{d_k}}) \tag{16.13}$$

This means that the weights a are defined by how each word of the sequence (represented by Q) is influenced by all the other words in the sequence (represented by K). Additionally, the SoftMax function is applied to the weights a to have a distribution between 0 and 1. Those weights are then applied to all the words in the sequence that are introduced in V (same vectors than Q for encoder and decoder but different for the module that has encoder and decoder inputs).

The righthand picture describes how this attention-mechanism can be parallelized into multiple mechanisms that can be used side by side. The attention mechanism is repeated multiple times with linear projections of Q, K and V. This allows the system to learn from different representations of Q, K and V, which is beneficial to the model. These linear representations are done by multiplying Q, K and V by weight matrices W that are learned during the training.

Those matrices Q, K and V are different for each position of the attention modules in the structure depending on whether they are in the encoder, decoder or in-between encoder and decoder. The reason is that we want to attend on either the whole encoder input sequence or a part of the decoder input sequence. The multi-head attention module that connects the encoder and decoder will make sure that the encoder input-sequence is taken into account together with the decoder input-sequence up to a given position.

After the multi-attention heads in both the encoder and decoder, we have a pointwise feed-forward layer. This little feed-forward network has identical parameters for each position, which can be described as a separate, identical linear transformation of each element from the given sequence.

16.5.5.3 Training

The purpose of this section is to understand how to train such a structure. We know that to train a model for translation tasks we need two sentences in different languages that are translations of each other. Once we have a lot of sentence pairs, we can start training our model. Let's say we want to translate French to German. Our encoded input will be a French sentence and the input for the decoder will be a German sentence. However, the decoder input will be shifted to the right by one position. One reason is that we do not want our model to learn how to copy our decoder input during training, but we want to learn that given the encoder sequence and a particular decoder sequence, which has been already seen by the model, we predict the next word/character. If we

don't shift the decoder sequence, the model learns to simply 'copy' the decoder input, since the target word/character for position i would be the word/character i in the decoder input. Thus, by shifting the decoder input by one position, our model needs to predict the target word/character for position i having only seen the word/characters $1, \ldots, i-1$ in the decoder sequence. This prevents our model from learning the copy/paste task. We fill the first position of the decoder input with a start-of-sentence token, since that place would otherwise be empty because of the right-shift. Similarly, we append an end-of-sentence token to the decoder input sequence to mark the end of that sequence and it is also appended to the target output sentence. In a moment, we'll see how that is useful for inferring the results.

This is true for Seq2Seq models and for the Transformer. In addition to the right-shifting, the Transformer applies a mask to the input in the first multi-head attention module to avoid seeing potential 'future' sequence elements. This is specific to the Transformer architecture because we do not have RNNs where we can input our sequence sequentially. Here, we input everything together and if there were no mask, the multi-head attention would consider the whole decoder input sequence at each position.

The target sequence we want for our loss calculations is simply the decoder input (German sentence) without shifting it and with an end-of-sequence token at the end.

Figure 16.19: Transformers.

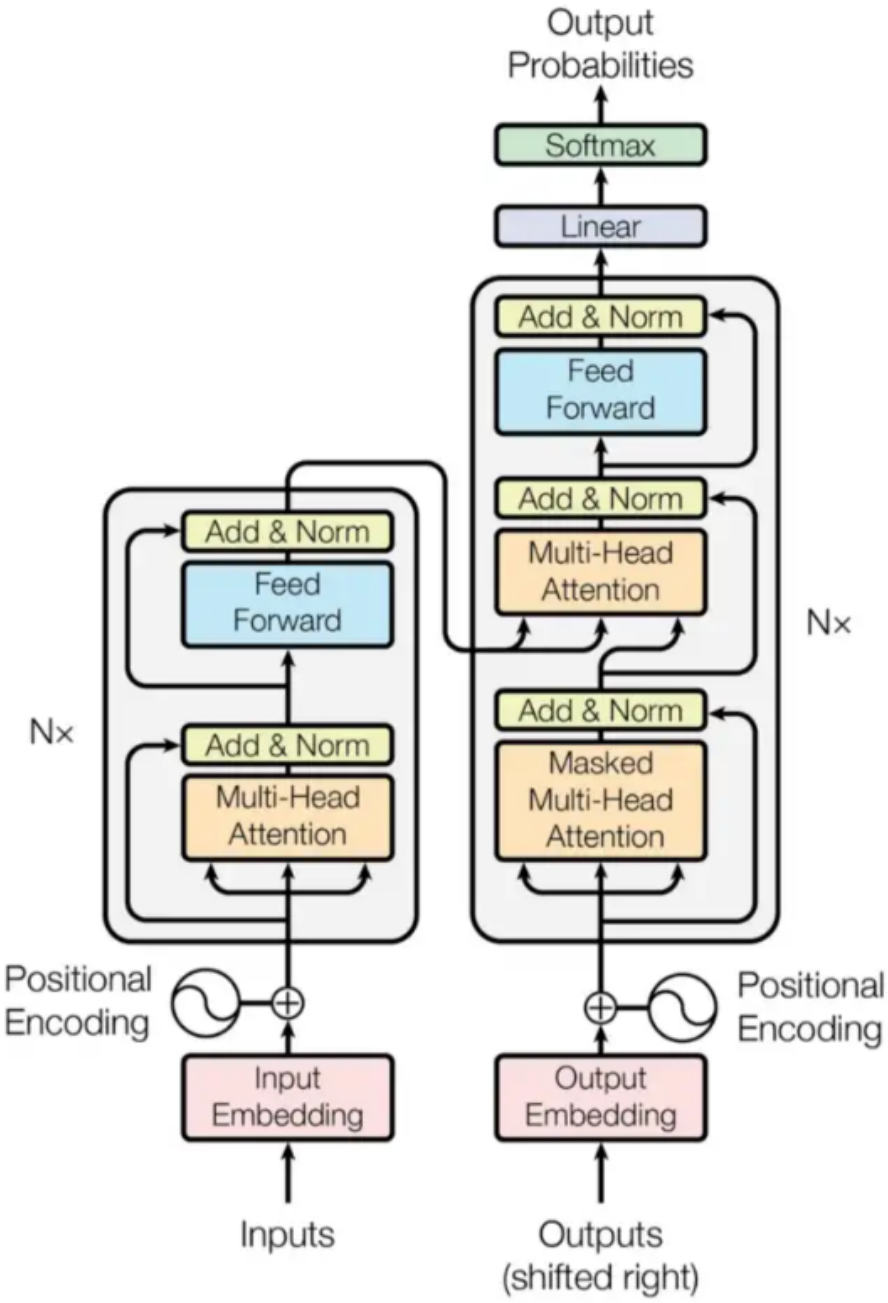


Figure 16.20: The Transformer model architecture.

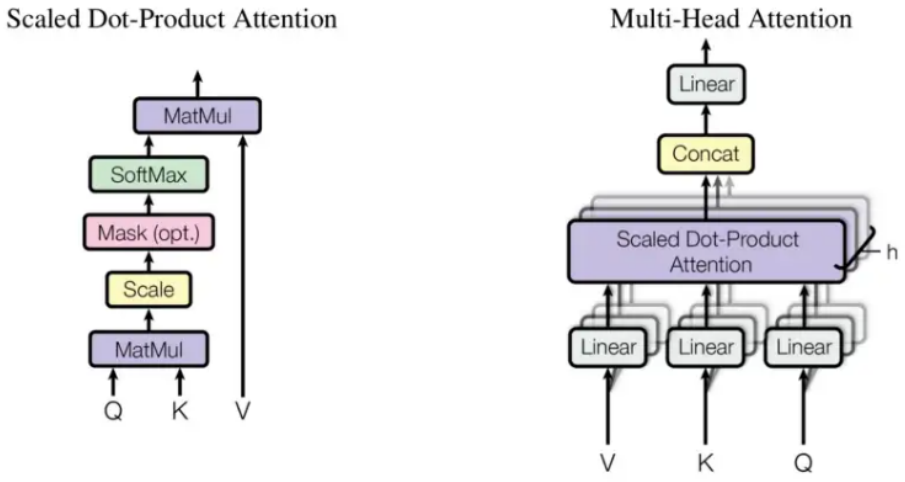


Figure 16.21: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

16.5.5.4 Inference

Inferring with those models is different from the training, which makes sense because in the end we want to translate a French sentence without having the German sentence. The trick here is to re-feed our model for each position of the output sequence until we come across an end-of-sentence token.

A more step by step method would be:

- Input the full encoder sequence (French sentence) and as decoder input, we take an empty sequence with only a start-of-sentence token on the first position. This will output a sequence where we will only take the first element.
- That element will be filled into second position of our decoder input sequence, which now has a start-of-sentence token and a first word/character in it.
- Input both the encoder sequence and the new decoder sequence into the model. Take the second element of the output and put it into the decoder input sequence.
- Repeat this until you predict an end-of-sentence token, which marks the end of the translation.

We see that we need multiple runs through our model to translate our sentence.

16.5.5.5 Professor slide

- Use attention mechanism to learn input word encodings that depend on other words in the sentence
- Use attention mechanism to learn output word encodings that depend on input word encodings and previously generated output words
- Predict output words sequentially stopping when the "word" end-of-sentence is predicted

16.5.6 Graph Neural Networks

At this point we understood that deep learning is good at capturing hidden patterns of Euclidean data (images, text, videos). But what about applications where data is generated from non-Euclidean domains, represented as graphs with complex relationships and interdependencies between objects? That's where *Graph Neural Networks* (GNN) come in, which we'll explore in this subsection (Figure 16.22).

Sometimes the nodes of the represented graph have a set of features (for example, a user profile). If the node has f numbers of features, then the node feature matrix X has a dimension of $(n \times f)$.

Graph data is so complex that it's created a lot of challenges for existing machine learning algorithms (Figure 16.23).

The reason is that conventional Machine Learning and Deep Learning tools are specialized in simple data types. Like images with the same structure and size, which we can think of as fixed-size grid graphs. Text and speech are sequences, so we can think of them as line graphs.

But there are more complex graphs, without a fixed form, with a variable size of unordered nodes, where nodes can have different amounts of neighbors.

It also doesn't help that existing machine learning algorithms have a core assumption that instances are independent of each other. This is false for graph data, because each node is related to others by links of various types.

Graph Neural Networks (GNNs) are a class of deep learning methods designed to perform inference on data described by graphs.

GNNs are neural networks that can be directly applied to graphs, and provide an easy way to do node-level, edge-level, and graph-level prediction tasks.

GNNs can do what Convolutional Neural Networks (CNNs) failed to do.

CNNs can be used to make machines visualize things, and perform tasks like image classification, image recognition, or object detection. This is where CNNs are the most popular.

The core concept behind CNNs introduces hidden convolution and pooling layers to identify spatially localized features via a set of receptive fields in kernel form (Figure 16.24).

How does convolution operate on images that are regular grids? We slide the convolutional operator window across a two-dimensional image, and we compute some function over that sliding window. Then, we pass it through many layers.

Our goal is to generalize the notion of convolution beyond these simple two-dimensional lattices.

The insight allowing us to reach our goal is that convolution takes a little sub-patch of the image (a little rectangular part of the image), applies a function to it, and produces a new part (a new pixel).

What happens is that the center node of that center pixel aggregates information from its neighbors, as well as from itself, to produce a new value.

It's very difficult to perform CNN on graphs because of the arbitrary size of the graph, and the complex topology, which means there is no spatial locality.

There's also unfixed node ordering. If we first labeled the nodes A, B, C, D, E, and the second time we labeled them B, D, A, E, C, then the inputs of the matrix in the network will change. Graphs are invariant to node ordering, so we want to get the same result regardless of how we order the nodes.

In graph theory, we implement the concept of Node Embedding. It means mapping nodes to a d- dimensional embedding space (low dimensional space rather than the actual dimension of the graph), so that similar nodes in the graph are embedded close to each other.

Our goal is to map nodes so that similarity in the embedding space approximates similarity in the network.

Let's define u and v as two nodes in a graph.

x_u and x_v are two feature vectors.

Now we'll define the encoder function Enc(u) and Enc(v), which converts the feature vectors to z_u and z_v (Figure 16.25).

Note: the similarity function could be Euclidean distance.

So the challenge now is how to come up with the encoder function?

The encoder function should be able to perform :

- Locality (local network neighborhoods)
- Aggregate information
- Stacking multiple layers (computation)

Locality information can be achieved by using a *computational graph*. As shown in the graph of Figure 16.26, i is the red node where we see how this node is connected to its neighbors and those neighbors' neighbors. We'll see all the possible connections, and form a computation graph.

By doing this, we're capturing the structure, and also borrowing feature information at the same time.

Once the locality information preserves the computational graph, we start aggregating. This is basically done using neural networks (Figure 16.27).

Neural Networks are presented in grey boxes. They require aggregations to be order-invariant, like sum, average, maximum, because they are permutation-invariant functions. This property enables the aggregations to be performed.

Let's move on to the forward propagation rule in GNNs. It determines how the information from the input will go to the output side of the neural network (Figure 16.28).

Every node has a feature vector.

For example, (X_A) is a feature vector of node A.

The inputs are those feature vectors, and the box will take the two feature vectors (X_A and X_c), aggregate them, and then pass on to the next layer.

Notice that, for example, the input at node C are the features of node C, but the representation of node C in layer 1 will be a hidden, latent representation of the node, and in layer 2 it'll be another latent representation.

Now, to train the model we need to define a loss function on the embeddings. We can feed the embeddings into any loss function and run stochastic gradient descent to train the weight parameters. Training can be unsupervised or supervised.

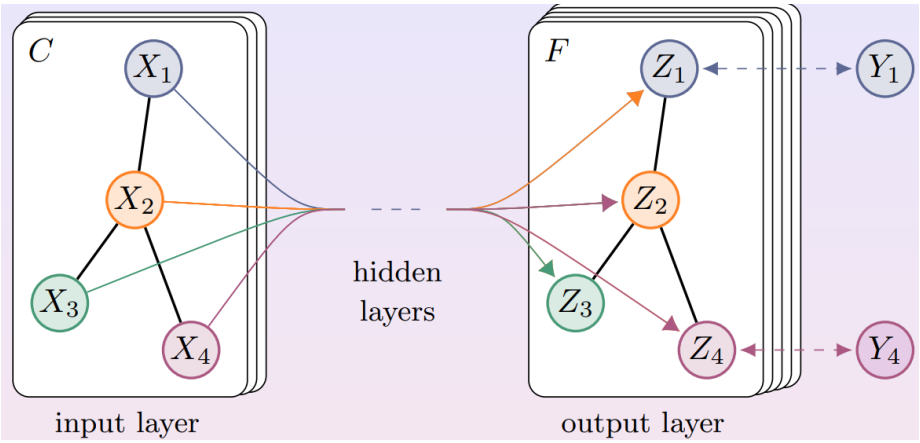


Figure 16.22: Graph Neural Networks.

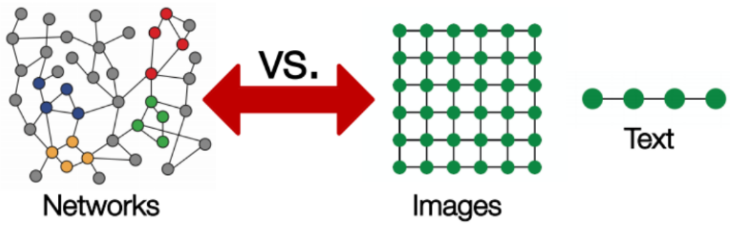


Figure 16.23: Conventional Machine Learning and Deep Learning tools are specialized in simple data types. Like images with the same structure and size, which we can think of as fixed-size grid graphs. Text and speech are sequences, so we can think of them as line graphs.

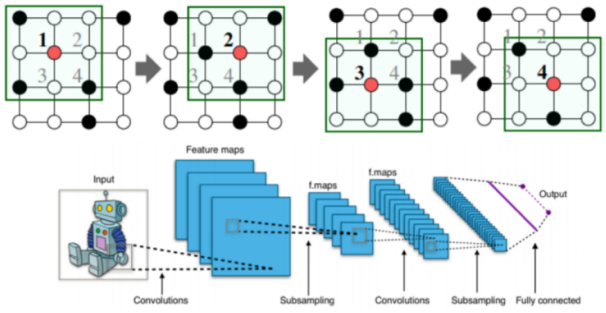


Figure 16.24: The core concept behind CNNs introduces hidden convolution and pooling layers to identify spatially localized features via a set of receptive fields in kernel form.

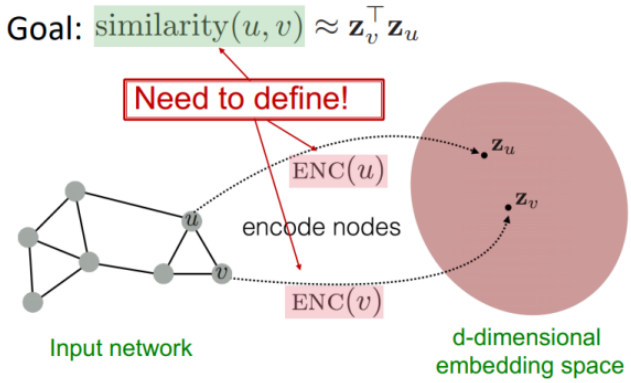


Figure 16.25: The encoder function Enc(u) and Enc(v), converts the feature vectors to z_u and z_v.

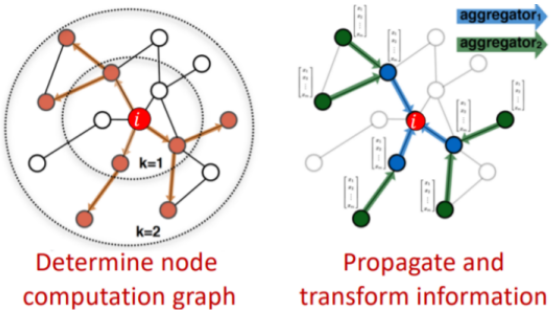


Figure 16.26: Locality information can be achieved by using a *computational graph*.

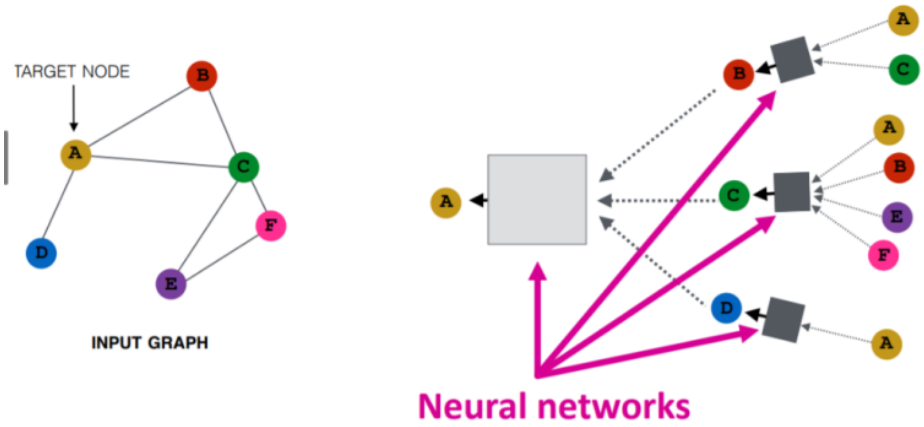


Figure 16.27: Once the locality information preserves the computational graph, we start aggregating. This is basically done using neural networks.

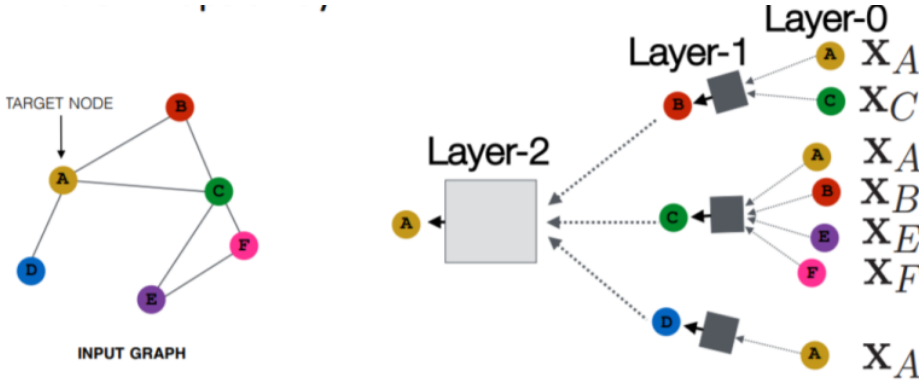


Figure 16.28: The forward propagation rule in GNNs. It determines how the information from the input will go to the output side of the neural network.

16.5.6.1 Graph Convolutional Networks

GCNs were first introduced in "Spectral Networks and Deep Locally Connected Networks on Graphs" (Bruna et al, 2014), as a method for applying neural networks to graph-structured data.

The simplest GCN has only three different operators:

- Graph convolution
- Linear layer
- Nonlinear activation

The operations are usually done in this order. Together, they make up one network layer. We can combine one or more layers to form a complete GCN.

16.5.6.2 Applications of GNNs

Graph-structured data is present everywhere. The problems that GNNs resolve can be classified into these categories:

1. Node Classification: the task here is to determine the labeling of samples (represented as nodes) by looking at the labels of their neighbors. Usually, problems of this type are trained in a semi-supervised way, with only a part of the graph being labeled.
2. Graph Classification: the task here is to classify the whole graph into different categories. It's like image classification, but the target changes into the graph domain. The applications of graph classification are numerous and range from determining whether a protein is an enzyme or not in bioinformatics, to categorizing documents in NLP, or social network analysis.

3. Graph visualization: is an area of mathematics and computer science, at the intersection of geometric graph theory and information visualization. It is concerned with the visual representation of graphs that reveals structures and anomalies that may be present in the data and helps the user to understand the graphs.

4. Link prediction: here, the algorithm has to understand the relationship between entities in graphs, and it also tries to predict whether there's a connection between two entities. It's essential in social networks to infer social interactions or to suggest possible friends to the users. It has also been used in recommender system problems and in predicting criminal associations.

5. Graph clustering: refers to the clustering of data in the form of graphs. There are two distinct forms of clustering performed on graph data. Vertex clustering seeks to cluster the nodes of the graph into groups of densely connected regions based on either edge weights or edge distances. The second form of graph clustering treats the graphs as the objects to be clustered and clusters these objects based on similarity.

16.5.6.3 Professor slide

Task: learning with graph convolution (Figure 16.22):

- Allow to learn feature representations for nodes
- Allow to propagate information between neighbouring nodes
- Allow for efficient training (with respect to e.g. graph kernels)

Chapter 17

Ensemble Methods

17.1 Ensemble Methods

The rationale:

- Groups of individuals can make better decisions than individuals (if they don't all think the same)
- Combining multiple ML models can produce better predictions than those of any single model
- Training ensembles can (often) be parallelized, with substantial computational savings

Models should be diverse enough for the ensemble to work. Ensembling strategies:

- **bagging**: diversify the training sets
- **stacking**: diversify the models being trained
- **boosting**: (progressively) diversify the importance of examples

17.2 Bagging: Bootstrap Aggregation

17.2.1 In a nutshell

1. Take a learning algorithm $\mathcal{A}$ (e.g. decision tree learning)
2. Extract m different datasets $\mathcal{D}^{(i)}$ from the original training set $\mathcal{D}$
3. Train one base model per dataset

$$f^{(i)} = \mathcal{A}\left(\mathcal{D}^{(i)}\right)$$

4. Combine predictions of different base models

$$\hat{y} = \text{Combine}\left(f^{(1)}(x), \ldots, f^{(m)}(x)\right)$$

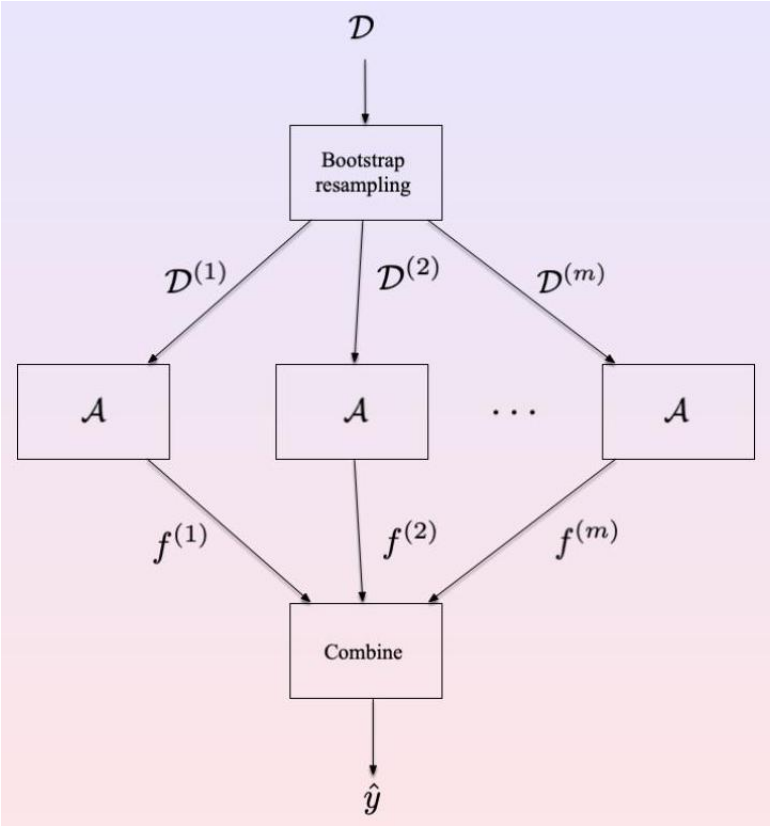


Figure 17.1: Bagging: Bootstrap aggregation

17.2.2 Extracting Datasets: Boostrap Resampling

Simply partitioning $\mathcal{D}$ into m subsets produces very small training sets. Bootstrap resampling extracts $N = |\mathcal{D}|$ samples from $\mathcal{D}$ with replacement (i.e., same example can be selected multiple times). Repeating the procedure m times to get diverse datasets of the same size as $\mathcal{D}$.

Diversity of Datasets:

- Each example has probability $(1 - 1/N)$ of not being selected at each draw
- Each example has probability $(1 - 1/N)^N$ of not being selected after N draws

- For large enough N, this is 37%, i.e., 37% of the dataset are not part of a given training set (out-of-bag instances, oob)
- oob instances can be used to estimate test performance of the base model

17.2.3 Combination Strategies

Majority voting predict the class with most votes (classification)

$$\hat{y} = \text{argmax}_y \sum_{i=1}^{m} \delta\left(y, \hat{y}^{(i)}\right)$$

Soft voting predict the class with largest sum of predicted probability (classification). Assumes base classifier outputs a confidence/probability

$$\hat{y} = \text{argmax}_y \sum_{i=1}^{m} f_y^{(i)}(x)$$

Mean predict the mean (or median) of the base model outputs (regression)

$$\hat{y} = \sum_{i=1}^{m} f^{(i)}(x)$$

17.2.4 Bagging example: Random forests

Bagging and Model Decorrelation:

- Use decision trees as the base models
- Even if $\mathcal{D}^{(i)} \neq \mathcal{D}^{(j)}$, the learned DTs can be too much correlated
- Introduce additional stochasticity in the DT training process
- At each node, choose the best feature to split from a random selection of the set of features (instead of using them all)

17.3 Stacking: Stacked Generalization

17.3.1 In a nutshell

1. Train m base models $f^{(i)}$ on $\mathcal{D}$ with m different learning algorithms $\mathcal{A}^{(i)}$

$$f^{(i)} = \mathcal{A}^{(i)}(\mathcal{D})$$

2. Use a meta-learner to learn to combine base models (e.g., linear combination)

$$g = \mathcal{A}_{\text{META}}\left(\left[f^{(1)}, \ldots, f^{(m)}\right], \mathcal{D}'\right)$$

3. Use the meta-model to output ensemble predictions

$$\hat{y} = g\left(\left[f^{(1)}(x), \ldots, f^{(m)}(x)\right]\right)$$

The meta-model should be trained on a dataset $\mathcal{D}' \neq \mathcal{D}$, or it will learn to focus on the best performing model on $\mathcal{D}$.

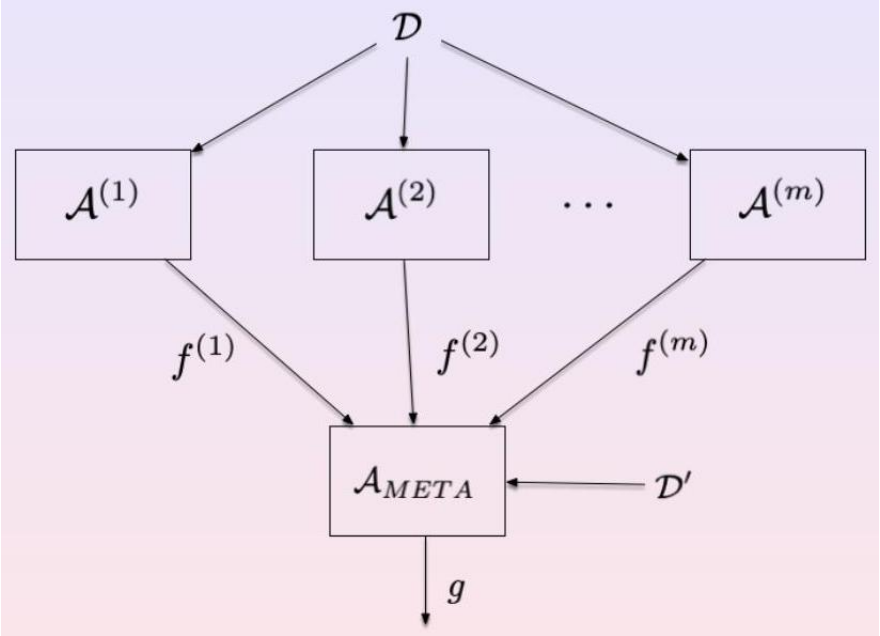


Figure 17.2: Bagging: Bootstrap aggregation

17.4 Boosting

17.4.1 In a nutshell

1. Take a learner $\mathcal{A}$ and train it on $\mathcal{D}$
2. Reweight examples in $\mathcal{D}$ based on their accuracy according to the trained model (harder examples get larger weights)
3. Train $\mathcal{A}$ again on the reweighted dataset
4. Repeat the procedure m times
5. Combine the learned models into the final model

A weak learner learns models with accuracy slightly better than random and is easy to implement and train. Applying boosting with weak learners as the base learning algorithm allows to turn them into strong learners. This can be easier than designing a strong learner directly.

17.4.2 Boosting example: AdaBoost

1: procedure ADABOOST
2: $\mathbf{d}^{(0)} \leftarrow \left\langle \frac{1}{N}, \ldots, \frac{1}{N} \right\rangle$ ▷ initialize importance weights uniformly
3: for $i = 1, \ldots, m$ do
4: $f^{(i)} \leftarrow \mathcal{A}\left(\mathcal{D}, \mathbf{d}^{(i-1)}\right)$ ▷ train i^{th} model on weighted data
5: $\hat{y}_n \leftarrow f^{(i)}\left(x_n\right), \forall n$ ▷ collect model predictions
6: $\hat{\epsilon}^{(i)} \leftarrow \sum_n d_n^{(i-1)} \mathbb{1}\left[y_n \neq \hat{y}_n\right]$ ▷ compute weighted training error
7: $\alpha^{(i)} \leftarrow \frac{1}{2} \log\left(\frac{1-\hat{\epsilon}^{(i)}}{\hat{\epsilon}^{(i)}}\right)$ ▷ compute adaptive parameter
8: $d_n^{(i)} \leftarrow \frac{1}{Z} d_n^{(i-1)} \exp\left(-\alpha^{(i)} y_n \hat{y}_n\right), \forall n$ ▷ re-weight examples
9: end for
10: return $f(\mathbf{x}) = \operatorname{sgn}\left(\sum_i \alpha^{(i)} f^{(i)}(\mathbf{x})\right)$ ▷ return ensemble model
11: end procedure

$$d_n^{(i)} \leftarrow \frac{1}{Z} d_n^{(i-1)} \exp\left(-\alpha^{(i)} y_n \hat{y}_n\right)$$

Example re-weighting:

- correctly classified examples $y_n \hat{y}_n = +1$ are multiplicatively downweighted
- incorrectly classified examples $y_n \hat{y}_n = -1$ are multiplicatively upweighted
- Z is a normalization constant making weights sum to one (importance as probability distribution over examples)

Role of Adaptive Parameter (AdaBoost):

- Let $\mathcal{D}$ have 80 positive and 20 negative examples
- Assume the first classifier $f^{(1)}$ simply returns the majority class (i.e., $f^{(1)}\left(\mathbf{x}_n\right) = +1$ for all n)
- It's weighted error rate is 0.2 (all negative examples are incorrectly predicted)
- The adaptive parameter is

$$\alpha^{(1)} = \frac{1}{2} \log\left(\frac{1-\hat{\epsilon}^{(i)}}{\hat{\epsilon}^{(i)}}\right) = 1/2 \log(4)$$

- The multiplicative weight for positive (correct) examples is $\exp\left(-\alpha^{(i)} y_n \hat{y}_n\right) = \exp(-1/2 \log(4)) = 1/2$.
- The multiplicative weight for negative (incorrect) examples is $\exp\left(-\alpha^{(i)} y_n \hat{y}_n\right) = \exp(1/2 \log(4)) = 2$.

- The normalization Z (ignoring $d_n^{(i-1)}$ for simplicity) is $80*1/2+20*2 = 80$
- The normalized weights for positive and negative examples are $1/2*1/80 = 1/160$ and $2 * 1/80 = 1/40$
- The overall weight of positive and negative examples is now $1/160 * 80 = 1/2$ and $1/40 * 20 = 1/2$
- The reweighed dataset is thus fully balanced, and a majority class predictor is not a weak learner any more (accuracy exactly 50%).

Chapter 18

Unsupervised learning

While for supervised models we use labelled examples, for unsupervised we do not have any labels, but only data.

18.1 Clustering

18.1.1 K-means clustering

This is a quite popular naive approach for clustering. This method assumes that you decide a priori the number of clusters. Each cluster will be represented by its mean μ_i. The idea is to assign example to the clusters with the closer mean.

The algorithm works within this line:

1. initialize cluster means $\mu_1, \ldots, \mu_n$ (a way of doing it is at random)
2. iterate until no mean changes:
   (a) assign each example to cluster with nearest mean
   (b) update cluster means according to assigned examples

This is a very simple approach, it is still used especially when we have some form of processing (dimensionality reduction ...) that maps example in low dimensional space, where k-means is sufficient.
Combined with other approached is still something that is currently used frequently.

18.1.2 Distance metrics

In order to compute some clustering, we need to define a metric for the concept of "distance" (aka notion of *similarity*), here we report the main metrics that can be used:

1. **standard euclidean distance in** $\mathbb{R}^d$: is the most natural choice, it is a instance of a more general metric (Minkowski, see next point)

$$d(\boldsymbol{x}, \boldsymbol{x}') = \sqrt{\sum_{i=1}^{d} (\boldsymbol{x}_i - \boldsymbol{x}'_i)^2}$$

2. **generic Minkowski metric for** $p \geq 1$:

$$d(\boldsymbol{x}, \boldsymbol{x}') = \left(\sum_{i=1}^{d} |\boldsymbol{x}_i - \boldsymbol{x}'_i|^p \right)^{1/p}$$

   If we replace $p = 2$ we get the euclidean distance while if $p = 1$ we get the Manhattan distance.

3. **Cosine similarity (cosine of the angle between two vectors)**: this is just the dot product divided by the two norms.

$$s(\boldsymbol{x}, \boldsymbol{x}') = \frac{\boldsymbol{x}^T \boldsymbol{x}'}{||\boldsymbol{x}||||\boldsymbol{x}'||}$$

It is also possible to use metric learning: instead of assuming a pre-defined metric, we can learn metric from data.

18.1.3 Quality of clustering

There are a number of criteria for defining a quality of clusters, here we present just one, quite intuitive: **Sum-of-squared error criterion**. This method tells me how bad the approximation of the cluster is using the means.

For each cluster, we take the mean μ_i (as general measure, not only k-means) within each cluster. I compare all examples (n_i number of samples) in the cluster with the mean of the cluster they are in. The error will be computed as the squared error (sample and mean):

$$\mu_i = \frac{1}{n_i} \sum_{\boldsymbol{X} \in \mathcal{D}_i} \boldsymbol{x}$$

The sum-of-squared errors is defined as the sum over the totality of clusters

$$E = \sum_{i=1}^{k} \sum_{\boldsymbol{X} \in \mathcal{D}_i} ||\boldsymbol{x} - \mu_i||^2$$

This basically tells us how bad the approximation of a cluster is using the means.

18.2 Gaussian Mixture Model (GMM)

The idea is again assuming a priori the number of clusters I want to get, then I suppose that each cluster is represented as a Gaussian distribution. I need to estimate the mean and possibly the variance (or co-variance) of the Gaussian distribution.

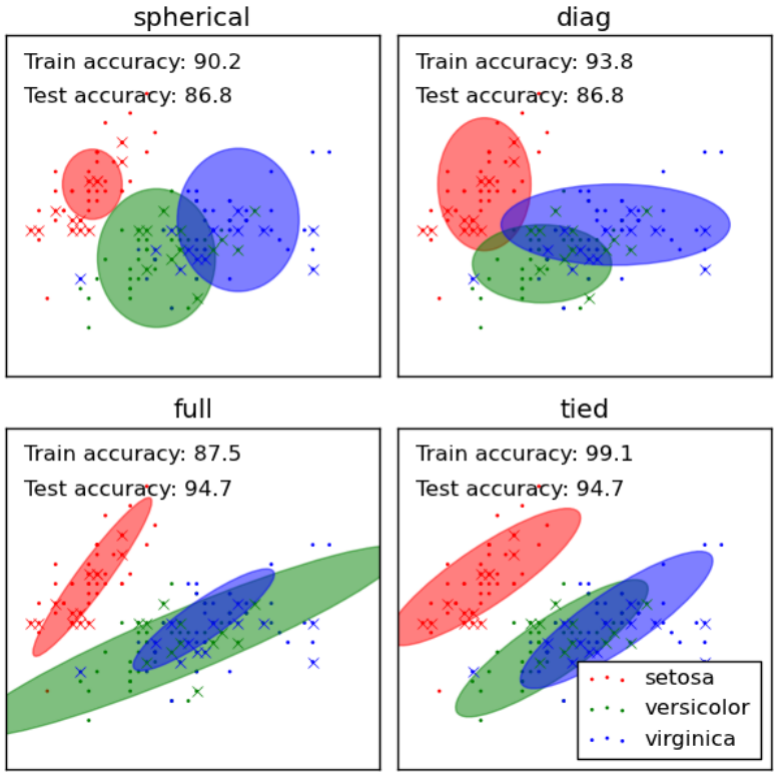


Figure 18.1: GMM clustering for the same data with different parameters

In this Figure 18.1, we represent ways in which we can use variances in different dimensions:

1. **Spherical**: we use same variance for each dimension and for each cluster
2. **Diagonal**: we use diagonal variance (axis aligned gaussians), we can still have different spreads in different dimensions
3. **Tied**: in this case we also introduce covariance (estimate relationships between features), shared (tied) between different clusters.
4. **Full**: in this example we see a full solution, in which we can have different co-variances matrices for different clusters.

This is an example of different ways of fitting the same data, so based on the approach we select we can get different solutions.

18.2.1 Spherical GMM

This is the easier way of using GMM, therefore mean estimation. Of course the problem of applying GMM, we need to estimate parameter, but we can no longer use maximum likelihood estimation because we do not have any label for the data. Here is an example of the so called *Latent Variables* (which consists in a probability of assigning the sample to each cluster). Based on this, we can do Expectation Maximization.

1. start with initialization of parameters,
2. we compute the expected value for the latent variables (expected value for clusters assignment)
3. using this expected value, we maximize the likelihood through the parameters
4. this gives me a new value of the parameter and then iterate until convergence

Settings:

- A dataset of n examples
- for each example x_i, a cluster assignment (the latent variable) is modelled as $z_{i1}, \ldots, z_{ik}$ (for each sample, a value for each cluster) modelled with a one-hot encoding (vector of binary variables, we have 1 only for the correct class, 0 otherwise)
- we assume for simplicity that we are only estimating the mean of the gaussian, the variance is assumed to be known

General Algorithm:

1. initialize $h = \mu_1, \ldots, \mu_k$ as hypothesis, initialization randomly
2. iterate until the difference in maximum likelihood is below a certain threshold:
   (a) **E-step**: calculate the expected value $E[z_{ij}]$ of each latent variable assuming that the current hypotheses holds.
   (b) **M-step**: calculate new maximum likelihood values (new hypothesis) $h' = \mu_1, \ldots, \mu_k$ assuming values of latent variables are their expected values. Replace h with h'.

Compute the expected value z_{ij}

1. **E-step**: computing the probability that example x_i comes from Gaussian$_j$, assuming that we known the means and the variances of the Gaussian

$$E[z_{ij}] = \frac{p(x_i|\mu_j)}{\sum_{l=1}^{k} p(x_i|\mu_l)} = \frac{e^{-\frac{1}{2\sigma^2}(x_i-\mu_j)^2}}{\sum_{l=1}^{k} e^{-\frac{1}{2\sigma^2}(x_i-\mu_l)^2}}$$

   The coefficients of the Gaussian distributions is omitted because can be simplified (is the variance is shared). This is a soft assignment.

2. **M-step**: perform maximum likelihood of the means μ'_j assuming that the latent variables are given by the expectations. The result is:

$$\mu'_j = \frac{\sum_{i=1}^{n} E[z_{ij}]x_i}{\sum_{i=1}^{n} E[z_{ij}]}$$

   If we know which example comes from which Gaussian, therefore we can be satisfied with the sample mean, but we do not know the belongings of the sample to the class. Here we know an *expectation* of belonging to classes (in terms of probability), therefore we do not have a zero-one value, but as continuous probability, which can be seen as an assignment to a class weighted to a certain probability.

Expectation maximization:
This is a general strategy used to deal with optimization and maximization of the parameters of the hypothesis, in a setting in which I have unobserved data. Generally I have a dataset made of observed part X and unobserved part Z. We would like to estimate the hypothesis maximizing the (log) likelihood of the data, where the latter include both X, Z.

$$h^* = \text{argmax}_h E_Z[ln \quad p(X, Z|h)]$$

We want to maximize given the expectation of Z. Unobserved data Z should be treated as random variables governed by the distribution depending on both X, h. The fact is that we do not know h, which is indeed the variable we want to estimate.
The iterative procedure is described as follows:

1. initialize hypothesis h
2. iterate until convergence
   (a) **E-step**: compute the expected (log) likelihood of an hypothesis h' for the full data, where the unobserved data distribution is modelled according to the current hypothesis h (from previous iterations) and the observed data as

$$Q(h', h) = E_Z[ln(p(X, Z|h'))|h, X]$$

   The expectation for the missing data Z (in GMM is the cluster assigment vector) is computed with respect to h, Z which are given by the last iteration and the known data.

(b) **M-step**: replace the current hypothesis with the one maximizing $Q(h', h)$

$$h \leftarrow \text{argmax}_{h'} Q(h', h)$$

This procedure is guaranteed to converge but it is likely to get to a local optimum. We can start with a good initialization coming from different methods.

The results for a GMM are:

$$p(x_i, z_{i1}, \dots, z_{ik}|h') = \frac{1}{\sqrt{2\pi}\sigma} e^{\left[-\sum_{j=1}^{k} z_{ij} \frac{(x_i - \mu'_j)^2}{2\sigma^2}\right]}$$

In which any generic x is x_i and z is a vector for each x and each of the possible clusters (one-hot encoding). We can model the right end side with a Gaussian distribution, but instead of having a single mean, we get all differences $(x_i - \mu'_j)^2$ and then we sum them up in some way that is weighted with the cluster assignment probability z_{ij}. If z_{ij} is one-hot, we will recover a single Gaussian, otherwise we will have a mixture.

If I compute the log likelihood on the full dataset:

$$ln(p(X, Z|h)) = \sum_{i=1}^{n} \left(ln \frac{1}{\sqrt{2\pi}\sigma} - \sum_{j=1}^{k} z_{ij} \frac{(x_i - \mu'_j)^2}{2\sigma^2} \right)$$

This is just log of product over the example of the likelihood (properties of logarithm used on the previous formula).

What we still need is the expectation over $E_Z[\dots]$. This is computed as follows:

$$\begin{aligned} E_Z[ln(p(Z, X|h'))] &= E_Z \left[\sum_{i=1}^{n} \left(ln \frac{1}{\sqrt{2\pi}\sigma} - \sum_{j=1}^{k} z_{ij} \frac{(x_i - \mu'_j)^2}{2\sigma^2} \right) \right] \\ &= \sum_{i=1}^{n} \left(ln \frac{1}{\sqrt{2\pi}\sigma} - \sum_{j=1}^{k} E[z_{ij}] \frac{(x_i - \mu'_j)^2}{2\sigma^2} \right) \end{aligned}$$

While the expectation of the x_i data comes from the j-th cluster, given current hypothesis h and observed data X is computed as:

$$\begin{aligned} E[z_{ij}] &= \frac{p(x_i|\mu_j)}{\sum_{l=1}^{k} p(x_i|\mu_l)} \\ &= \frac{e^{-\frac{1}{2\sigma^2}(x_i - \mu_j)^2}}{\sum_{l=1}^{k} e^{-\frac{1}{2\sigma^2}(x_i - \mu_l)^2}} \end{aligned}$$

The expectation of the log-likelihood is computed with h' while the second is computed starting from h (the old one, therefore this quantity is computable).

Given that I have the formula of the likelihood, then I can maximize it.
The next step is taking this expectation and then I perform maximization over the new hypothesis.
The likelihood maximization gives this first result (since there is a minus sign, we can minimize instead of maximize), then we can set the derivation equal to zero with respect to μ'_j in order to find the maximum (in the example we are considering just one specific μ_j instead summing for every component).

$$\begin{aligned}
\text{argmax}_{h'} Q(h'; h) &= \text{argmax}_{h'} \sum_{i=1}^{n} \left(\ln \frac{1}{\sqrt{2\pi}\sigma} - \sum_{j=1}^{k} E[z_{ij}] \frac{(x_i - \mu'_j)^2}{2\sigma^2} \right) \\
&= \text{argmin}_{h'} \sum_{i=1}^{n} \sum_{j=1}^{k} E[z_{i_j}] (x_i - \mu'_j)^2 \\
\frac{\partial}{\partial \mu_j} &= -2 \sum_{i=1}^{n} E[z_{ij}] (x_i - \mu'_j) = 0 \\
\mu'_j &= \frac{\sum_{i=1}^{n} E[z_{ij}] x_i}{\sum_{i=1}^{n} E[z_{ij}]}
\end{aligned}$$

18.2.2 How to choose the number of clusters

Up to now, we only assumed that we are given in advanced the number of clusters, but of course it is not always the case. In some cases, having background helps us to understand the number of clusters should use, in general we can try to find out the number of cluster automatically by trying different values and look for the best one. There are some methods that can be used for this purpose:

- **Elbow method**: based on the squared errors, if we increase the number of clusters, this quality measure drops (the edge case in which every example is by its own a cluster has an error equal to zero). We need to find a solution for the trade-off between the number of clusters and the means squared error: this method checks whether increasing the number of clusters is worth it or not. We run the clustering (any kind) with an increasing number of clusters and we plot (Figure 18.2) the error measure: in the curve we get we look for an "elbow", therefore a threshold for which we should stop increasing the number of clusters (drop in error starts to be less important). This method is a bit ambiguous, there is no formal procedure to do decide a point over another.

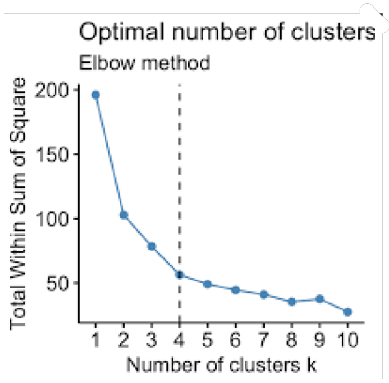


Figure 18.2: Plot of the error rate within the increase of the number of clusters

- **Average silhouette**: this method gives us a precise definition for the number of clusters. This is based on the fact that by increasing the number of clusters, each of them becomes more homogeneous. But on the other hand also different clusters also become similar to each other (splitting data further). We need to look for inter-clusters dissimilarity (not only intra-clusters similarity).
  The procedure is pretty simple:

  1. compute the average similarity between i and examples of its cluster (defined in terms of distances d), this is the **intra-cluster** similarity:

  $$a_i = d(i, C) = \frac{1}{|C|} \sum_{j \in C} d(i, j)$$

  2. compute the average dissimilarity between i and examples of each cluster C' different from its own cluster C and take the minimum (**inter-cluster** dissimilarity). We take the minimum to get the lower dissimilarity:

  $$b_i = \min_{C' \neq C} d(i, C')$$

  3. the silhouette coefficient is:

  $$s_i = \frac{b_i - a_i}{\max(a_i, b_i)}$$

  s_i will be averaged over all samples and clusters. We plot this coefficient within the increase of the number of cluster (Figure 18.3), that represent the trade-off between the inter e intra similarity. Eventually, we get the higher coefficient as final choice.

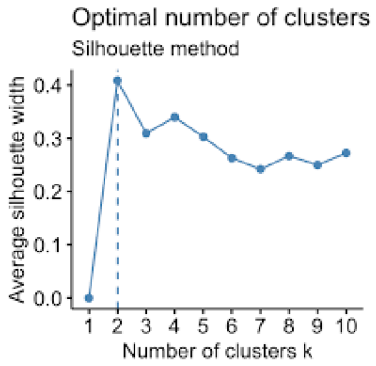


Figure 18.3: Plot the average silhouette coefficient according to an increasing number of clusters

18.3 Hierarchical clustering

This approach gives a lot of information about the structure of the data. The underline idea is that the groups often are hierarchical (as taxonomy). There are two main greedy procedure to build hierarchical clustering:

1. **Top-down approach**: starts with a single cluster and recursively splits the cluster in two groups according to a certain measure and then it repeats the procedure until we reach a maximum numbers of clusters or we reach clusters represented as single example.

2. **Bottom-up approach**: starts with a number of clusters equal to the numbers of examples, and tries to merge the most similar clusters. Again, this procedure is recursive.

This is something very common in computational biology, we end up with some kind of tree. The procedure for a bottom-up approach is pretty simple:

1. initialize:
   (a) final cluster number k (the minimum final number of clusters)
   (b) initial cluster number $\tilde{k} = n$
   (c) initial clusters $\mathcal{D}_i = \{x_i\}, i \in 1, \ldots, n$
2. while $\tilde{k} > k$:
   (a) find pairwise nearest clusters $\mathcal{D}_i, \mathcal{D}_j$ based on certain similarity measure
   (b) merge $\mathcal{D}_i, \mathcal{D}_j$
   (c) update $\tilde{k} = \tilde{k} - 1$

The similarity measure is computed between clusters, there are several methods that we can use:

- nearest neighbours, computes the similarity between sets as the minimal distance between the elements in the two sets (sensible to outliers and expensive)
- furthest neighbors, computes the similarity between sets as the maximum distance between the elements in the two sets (sensible to outliers, expensive)
- average distance, computes the average distances between the elements in the two sets (expensive)
- distance between means of clusters (more efficient)

Each of this approaches has pros and cons, depending on the problem one way could be better than another. We can also use some external and specific evaluation metric in order to find the clusters to merge (or to divide). This approach is **greedy** but not necessarily optimal.

Chapter 19

Reinforcement learning

The main character of a reinforcement learning setting is an *agent*, usually referred to in the slide as the *learner*. The learner interacts with an environment by means of actions. At any given point of time t, the learner is in a state s_t which belongs to a set of possible states $\mathcal{S}$. For each state $s_t \in \mathcal{S}$ the learner can perform a set of possible actions $a_t \in \mathcal{A}$ in order to move to a next state s_{t+1}. In performing action a from state s, the learner is provided an immediate *reward* $r(s, a)$ (possibly negative). The behaviour of the agent is regulated by a *policy function* $\pi : \mathcal{S} \to \mathcal{A}$. The task of a reinforcement learning algorithm is to learn a policy allowing to choose for each state s the action a maximizing the overall reward. A high level graphical representation of this learning setting is depicted in Figure 19.1.

The typical challenges faced by the leaner during this process are:

- *delayed reward* coming from future moves
- trade-off between *exploitation* and *exploration*

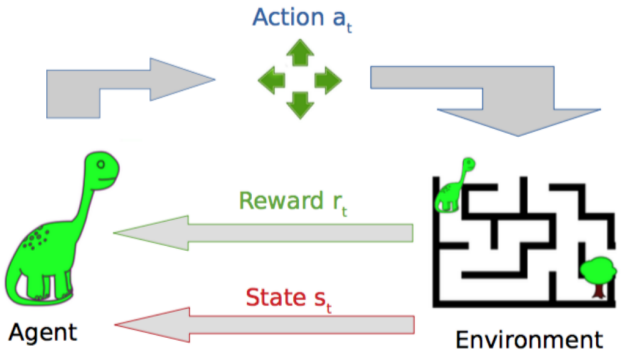


Figure 19.1: High level idea of a reinforcement learning setting

19.1 Applications

Reinforcement learning has several real world applications. The main research fields where reinforcement learning solutions are adopted are:

- videogames:
  - objective = complete the game with the highest score
  - state = raw pixel inputs of the game state
  - action = game controls (left, right, up, down)
  - reward = score increase/decrease
- robotics
- board games (e.g. Go):
  - objective = win the game
  - state = position of all pieces
  - action = where to put the next piece down
  - reward = 1 if win at the end of the game, 0 otherwise

19.2 Markov Decision Process MDP

Markov Decision Process (MDP) is a framework used to help to make decision on a *stochastic environment*. In a stochastic environment there is uncertainty in the result of a decision.
Formally a MDP is defined by a tuple of five elements: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \gamma)$:

- a set of states $\mathcal{S}$ in which the agent can be at each time instant. At the beginning the learner is at the initial state s_0. In S there is a (possibly empty) set of *teminal states* $\mathcal{S}_G \subset \mathcal{S}$
- a set of actions $\mathcal{A}$ the agent can make
- a reward function $R(s, a, s')$ for making action a in state s and reaching state s'
- a *transition* model providing the probability of going to a state s' with action a from state s. *Markov assumption* is that the probability of going to s' from s depends only on s and not on any other past actions or states.

$$P(s'|s, a)\ s, s' \in \mathcal{S}, a \in \mathcal{A} \tag{19.1}$$

- *discount factor* γ: a scalar (optional)

In Figure 19.2 there is illustrated an example of MDP. In this case, we have an agent moving in a room. The elements of the MDP are:

- state = occupied cell
- terminal states (row, column) = (4,2), (4,3)
- actions = UP, DOWN, LEFT, RIGHT
- transitions probabilities = 0.8 in direction of action, 0.1 in each orthogonal direction
- rewards = $R((4,2)) = -1$, $R((4,3)) = +1$

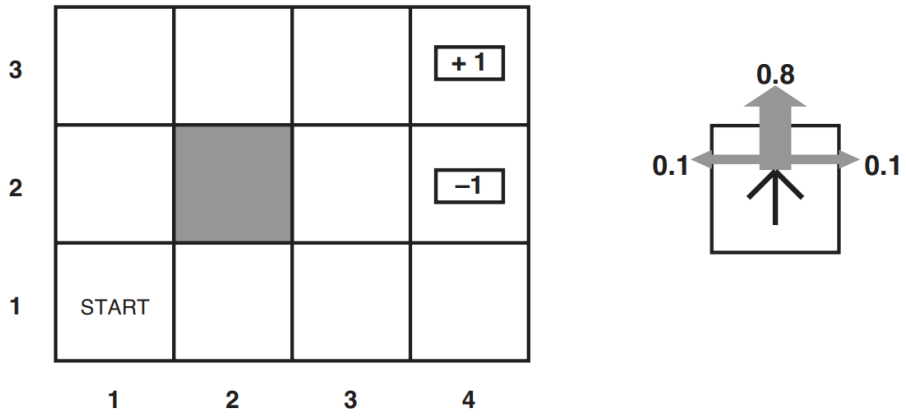


Figure 19.2: MDP Example

An *environment history* is a sequence of states. For example, a certain policy π starting in state s_0 could lead to a sequence of states $s_0, s_1, s_2, \ldots$. *Utilities* are defined over environment histories. In the following of the section we are going to take into account the following assumptions:

- we assume an *infinite horizon* (no constraints on the number of time steps)
- we assume *stationary* preferences, i.e. if one history is preferred to another at time t, the same should hold at time t' provided they start from the same state

Given an environment history $s_0, s_1, s_2, \ldots$, the utility can be computed in two main ways:

- *additive rewards*

$$U([s_0, s_1, s_2, \ldots]) = R(s_0) + R(s_1) + R(s_2) + \ldots \tag{19.2}$$

- *discounted rewards*

$$U([s_0, s_1, s_2, \ldots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots \tag{19.3}$$

Remark: in the more general case each reward should be written as $R(s_t, a_t, s_{t+1})$

Remark: the lower the discount factor γ is, the less important future rewards are, and the agent will tend to focus on actions which will yield immediate rewards only.

19.2.1 Taking decisions

A *policy* π is a full specification of what action to take at each state.

The *expected utility* of a policy is the utility of an environment history, taken in expectation over all possible histories generated with that policy.

An *optimal policy* $\pi*$ is a policy maximizing expected utility. In Figure 19.3 we propose an example of optimal policy:

- utility is made with additive rewards
- r is the reward of non-terminal states
- arrows indicate the best action to take
- star indicates all actions are equally optimal

From the proposed structure we can make the following observations:

- if moving is very expensive, optimal policy is to reach any terminal state as soon as possible
- if moving is very cheap, optimal policy is avoiding the bad terminal state at all costs
- if moving gives positive reward, optimal policy is to stay away of terminal states

Remark: for infinite horizons, optimal policies are *stationary*, i.e. they only depend on the current state.

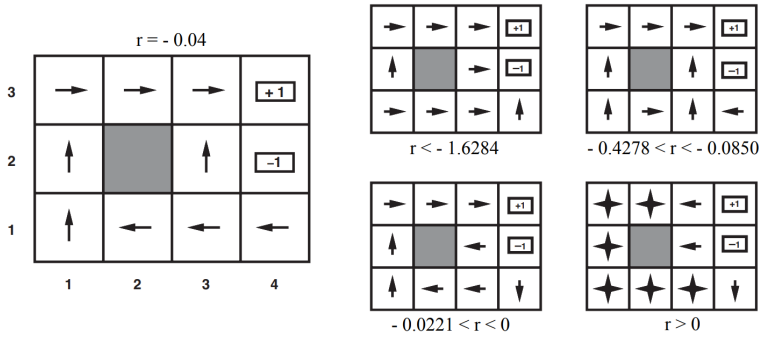


Figure 19.3: Optimal policy example

19.3 Learning the optimal policy

As opposed to supervised learning methods, in reinforcement learning, the learning loop does not allow us to define a loss function directly. As a result the learning procedure is more complicated. In general the solution is to let the agent exploring the environment and update the policy according to the achieved reward. More in depth, the literature proposes two main learning approaches:

- **value-based methods:**
  - the goal of the agent is to optimize the utility of a state $U(s)$ which quantifies "how good" is a state
  - the value of each state s is the total amount of the reward an agent can expect to collect over the future from s
  - maximizing $U(s)$ for each state, we indirectly optimize the policy
- **policy base methods**: we define a policy which we need to optimize directly. To do this, we typically learn a neural network which outputs a *stochastic policy*, i.e. a probability distribution over different actions.

$$\pi_\theta(s, a) \approx P(a|s)$$

19.3.1 Value based methods

The utility function gives the total amount of reward the agent can expect from a particular state to all possible state from that state. With the utility function we are going to find a policy. More formally, the utility U of a state s given a policy π is the expected cumulative reward we can get following policy π starting from s:

$$U^\pi(s) = E[\sum_{t=0}^{\infty} \gamma^t r(s_t)|s_0 = s, \pi] \tag{19.4}$$

with: $a_t = \pi(s_t)$; $s_{t+1} \sim P(\cdot|s_t, a_t)$

The true utility of a state is its utility under an optimal policy:

$$U(s) = U^{\pi^*}(s) = \max_\pi E[\sum_{t=0}^{\infty} \gamma^t r(s_t)|s_0 = s, \pi] \tag{19.5}$$

Given the true utility, an optimal policy is as follows:

$$\pi^*(s) = \text{argmax}_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} p(s'|s, a)U(s') \tag{19.6}$$

For each state $s \in S$ we define a **Bellman equation**: *the utility of a state is its immediate reward plus the expected discounted utility of the next state, assuming that the agent chooses an optimal action.*

$$U(s) = R(s) + \gamma * \max_{a \in \mathcal{A}} \sum_{s' \in S} p(s'|s, a)U(s') \tag{19.7}$$

Utilities of states are solutions of the set of Bellman equations. The solutions to the set of Bellman equations are unique. However, directly solving the set of equations is hard. As a consequence, we rely on a dynamic programming algorithmic approach (*value iteration*):

- **step 1:** initialize $U_0(s)$ to zero for all s
- **step 2:** repeat until max utility difference is below a threshold
  - **step 2.1:** do Bellman update for each state s:
    $$U_{i+1}(s) \leftarrow R(s) + \gamma * \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} p(s'|s,a) U_i(s')$$
  - **step 2.2:** $i \leftarrow i + 1$
- **step 3:** return U

This latter algorithm is used as a subroutine of the following algorithm (*policy iteration*) which is used to learn the optimal policy:

- **step 1:** initialize π_0 randomly
- **step 2:** repeat until no policy improvement
  - **step 2.1: policy evaluation** solve set of linear equations (as explained in the previous algorithm):
    $$U_i(s) = R(s) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, \pi_i(s)) U_i(s') \; \forall s \in \mathcal{S}$$
    where $\pi_i(s)$ is the action that policy π_i prescribes for state s
  - **step 2.2: policy improvement**
    $$\pi_{i+1}(s) \leftarrow \text{argmax}_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} p(s'|s,a) U_i(s') \; \forall s \in \mathcal{S}$$
  - **step 2.3:** $i \leftarrow i + 1$
- **step 3:** return π

19.4 Reinforcement learning in an unknown environment

Value iteration and policy iteration assume perfect knowledge (environment, transition model, rewards). However, in most cases, some of these aspects are not known. In order to face this limitation, reinforcement learning aims at learning policies by space exploration.

- **policy evaluation:** the policy is given and the passive agent explores the environment
- **policy improvement:** the policy is learnt while the active agent explores the environment

19.4.1 Adaptive Dynamic Programming (ADP) algorithm

In this case we assume that the environment is unknown and that the policy is given. The fact that we don't know the environment means that we don't know the states, the transitions probabilities and so on. The idea in this case is to explore the space collecting rewards r and saving them as a reward function $R(s) = r$. Then we take the action given by the policy (that is given) bringing the agent to state s'. Now we update the counts that allows to compute the transition probability function p incrementing by one the number of times we are in state s and we take action a, i.e.

$$N_{sa} \leftarrow N_{sa} + 1 \tag{19.8}$$

In addition we keep track of the number of times we are in state s and take an action a brings the agent to state s':

$$N_{s'|sa} \leftarrow N_{s'|sa} + 1 \tag{19.9}$$

At this point we have all the information that we need to update the transition model, by using maximum likelihood estimation. The probability of reaching state s'' being in state s taking action a is given by the fraction of times in which the agent is in state s and take an action a that brings the agent to state s'' divided by the number of times we are in state s and we take action a:

$$p(s''|s, a) = \frac{N_{s''|sa}}{N_{sa}} \; \forall s'' \in \mathcal{S} \tag{19.10}$$

The last step is to run *policy-evaluation* to obtain the utility estimate U (U is initially empty). Each step is expensive as it runs policy evaluation.

- **step 1:** initialize s
- **step 2:** repeat until s is terminal
  - **step 2.1:** receive reward r, set $R(s) = r$
  - **step 2.2:** choose next action $a \leftarrow \pi(s)$
  - **step 2.3:** take action a, reach step s'
  - **step 2.4:** update counts

$$N_{sa} \leftarrow N_{sa} + 1$$

$$N_{s'|sa} \leftarrow N_{s'|sa} + 1$$

  - **step 2.5:** update transition model $\forall s'' \in \mathcal{S}$

$$p(s''|s, a) \leftarrow \frac{N_{s''|sa}}{N_{sa}}$$

  - **step 2.6:** update utility estimate

$$U \leftarrow \text{policyEvaluation}(\pi, U, p, R, \gamma)$$

19.4.2 Temporal-difference (TD) policy evaluation

This is an approximate method that avoids to run policy evaluation at each iteration to compute the correct utility (expensive). The algorithms instead approximates the utility of the state which is based on the intuition that if we are in state s and the policy suggests to take action a and reach s', then if s' was *always* the successor of s the utility of s should be computed as:

$$U(s) = R(s) + \gamma U(s') \tag{19.11}$$

Note that this is a rough approximation, not the exact utility of the state s. Now the idea is to try to make the utility closer to the above approximation. Making the current $U(s)$ closer to $R(s)+\gamma U(s')$ can be possible by updating the current $U(s)$ in such a way that the difference between what we want ($R(s) + \gamma U(s')$) and what we have (the current $U(s)$)

$$U(s) = U(s) + \alpha(R(s) + \gamma U(s') - U(s)) \tag{19.12}$$

where α is a learning rate (possibly decreasing over time).

Remark: Note that we do not need a perfect estimate of the utility U of the states, the only think that we want is a good utility of the states that allows to find the best policy.

Remark: One of the main benefits of this algorithm is that it does not need a transition model to update the utility. As a consequence, each step is much faster than ADP. However, depending on the learning rate α, TD policy evaluation algorithm takes longer to converge. In a sense, this algorithm can be seen as a rough efficient approximation of ADP.

- **step 1:** initialize s
- **step 2:** repeat until s is terminal
  - **step 2.1:** receive reward r
  - **step 2.2:** choose next action $a \leftarrow \pi(s)$
  - **step 2.3:** take action a, reach step s'
  - **step 2.4:** update local utility estimate

$$U(s) \leftarrow U(s) + \alpha(r + \gamma U(s') - U(s))$$

19.4.3 Exploration vs exploitation trade-off

Up to this point we know that policy learning requires combining learning the environment and learning the optimal policy for the environment. Learning the environment and performing policy evaluation (which requires solving system

of linear equations) are expensive tasks to accommodate. A simple trick is the one followed by *greedy agents*. The idea is to replace the policy evaluation in ADP with optimal policy computation given the current knowledge of the environment:

$$U(s) = R(s) + \gamma \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} p(s'|s,a)U(s') \tag{19.13}$$

The problem is that the knowledge of the environment is incomplete. As a consequence, a greedy agent usually learns a suboptimal policy. More technically, this is a result of lack of *exploration*.

In Figure 19.4 we propose an example of greedy agent behaviour. The algorithm finds a policy reaching the +1 terminal state along the lower route (2,1), (3,1), (3,2) and (3,3). It never learns the utilities of the other states. As a result, it fails to discover the optimal route (1,2), (1,3) and (2,3).

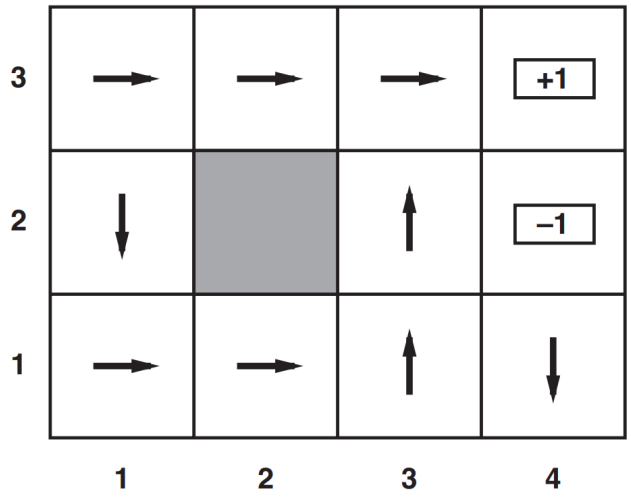


Figure 19.4: Greedy agent behaviour.

The solution to face this problem is a good trade-off between *exploitation* and *exploration*.

- **Exploitation:** consists in following promising directions given current knowledge.
- **Exploration:** consists in trying novel directions looking for better (unknown) alternatives.

In order to find a good trade-off between the twos there are two approaches:

1. ϵ-greedy strategy: with probability ϵ the agent moves randomly (exploration) and with probability $1 - \epsilon$ goes greedy.

2. another approach is to assign high utility estimates to unexplored state-action pairs and decrease the utility on an action if you already did it a lot of times, i.e. a high value of N_{sa} causes the utility on that state to reduce.

$$U^+(s) = R(s) + \gamma \text{max}_{a \in \mathcal{A}} f(\sum_{s' \in \mathcal{S}} p(s'|s,a) U^+(s'), N_{sa}) \tag{19.14}$$

with f increasing over the first argument and decreasing over the second.

19.4.4 Q-learning

As an alternative to what we have learnt so far, it is convenient to define the utility of a state-action (s, a) pair and not only of a state s. The value of a state-action pair is written $Q(s, a)$. Given policy π, the utility of a state action pair (s, a) is computed as follows:

$$Q^\pi(s,a) = E[\sum_{t=0}^{\infty} \gamma^t r(s_t) | s_0 = s, a_0 = a, \pi] \tag{19.15}$$

The true utility of a state-action pair (s, a) is its utility under an optimal policy:

$$Q(s,a) = Q^{\pi *}(s,a) = max_\pi E[\sum_{t=0}^{\infty} \gamma^t r(s_t) | s_0 = s, a_0 = a, \pi] \tag{19.16}$$

Once $Q(s, a)$ has been computed for each state action pair, the optimal policy corresponds to:

$$\pi *(s) = \text{argmax}_{a \in \mathcal{A}} Q(s,a) \tag{19.17}$$

The task is to progressively learn a table where we have the maximum expected future reward, for each action at each state. As we are going to see in the next algorithmic procedure, the update of the utility of a state action pair (s, a) is again computed by means of the Bellman equation:

$$Q(s,a) = R(s) + \gamma \text{max}_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} p(s'|s,a) U(s') \tag{19.18}$$

The procedure is recursive:

- **step 1:** initialize the Q matrix with zeros
- **step 2:** select a random initial state and a discount factor
- **step 3:** for each *episode* (set of actions that starts on the initial state and ends in the goal state)
  - **step 3.1:** while state is not a goal state

    - **step 3.1.1** select a random possible action for the current state
    - **step 3.1.2** using this possible action consider going to this next state s
    - **step 3.1.3** get maximum Q value for this next state s (taking into account all actions from this next state). This is computed solving Bellmann equation.

After some episode, my *Q-table* encodes the optimal policy:

- **step 1:** set current state = initial state
- **step 2:** repeat until current state = goal state
    - **step 2.1:** from current state find the action with the highest Q value
    - **step 2.2:** set current state = next state (state from action chosen on step 2)

This procedure assumes perfect knowledge of the environment, transition model, rewards. However, this is not the case in most of the application. As a consequence we can use ADP and TD algorithms also in this scenario. In particular TD SARSA algorithm is implemented in this context as follows:

- **step 1:** initialize s
- **step 2:** repeat until s is terminal
    - **step 2.1:** receive reward r
    - **step 2.2:** choose next action $a \leftarrow \pi^{\epsilon}(s)$
    - **step 2.3:** take action a, reach step s'
    - **step 2.4:** choose action $a' \leftarrow \pi^{\epsilon}(s')$
    - **step 2.5:** update local utility estimate

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma Q(s',a') - Q(s,a))$$

This is an **on-policy** solution. On the other hand we introduce here an alternative, referred to as *Q-learning* which implements an **off-policy** solution.

- **step 1:** initialize s
- **step 2:** repeat until s is terminal
    - **step 2.1:** receive reward r
    - **step 2.2:** choose next action $a \leftarrow \pi^{\epsilon}(s)$
    - **step 2.3:** take action a, reach step s'
    - **step 2.5:** update local utility estimate

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a' \in \mathcal{A}} Q(s',a') - Q(s,a))$$

The main difference between SARSA and Q-learning is that SARSA is *on-policy*, meaning that it updates Q using the current policy's action. On the other hand Q-learning is *off-policy*, meaning that it updates Q using the greedy policy's action (which is NOT the policy it uses to search).

Off-policy methods are more flexible. On the other hand, on-policy methods tend to converge faster, and are easier to use for continuous-state spaces and linear function approximators about which we discuss in the following of this chapter.

19.5 Scaling to large state spaces

All techniques seen so far assume a tabular representation of utility functions (e.g. state-action pairs). However, tabular representations do not scale to large state spaces. For example Backgammon has an order of 10^{20} states.

19.5.1 Function approximation

The first solution to deal with this limitation is to rely on *function approximation* (Figure 19.5). In essence, our aim is to approximate $U(s)$ or $Q(s, a)$ with a parametrized function. The function takes a state representation as input. For example (x, y) coordinates of a maze. With this formulation, the function allows to generalize to unseen states.

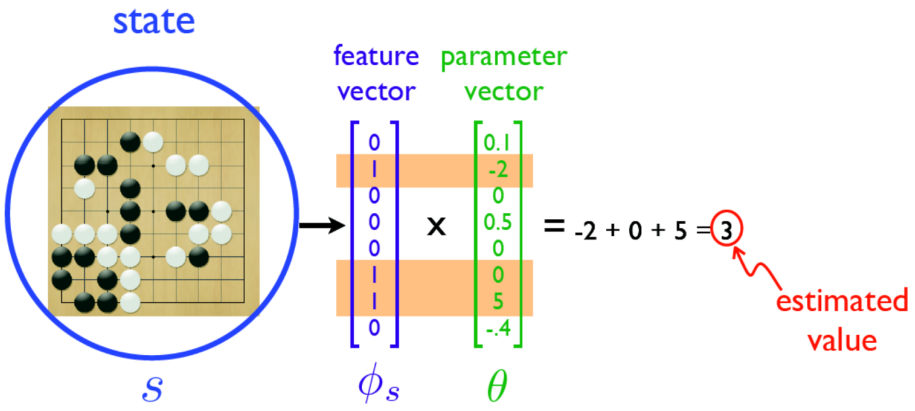


Figure 19.5: State utility function approximation.

19.5.2 Deep Q-learning

A refinement of the previous idea is to approximate $U(s)$ and $Q(s, a)$ with a parametric function and introduce a **deep neural network** to learn the parameters (Figure 19.6).

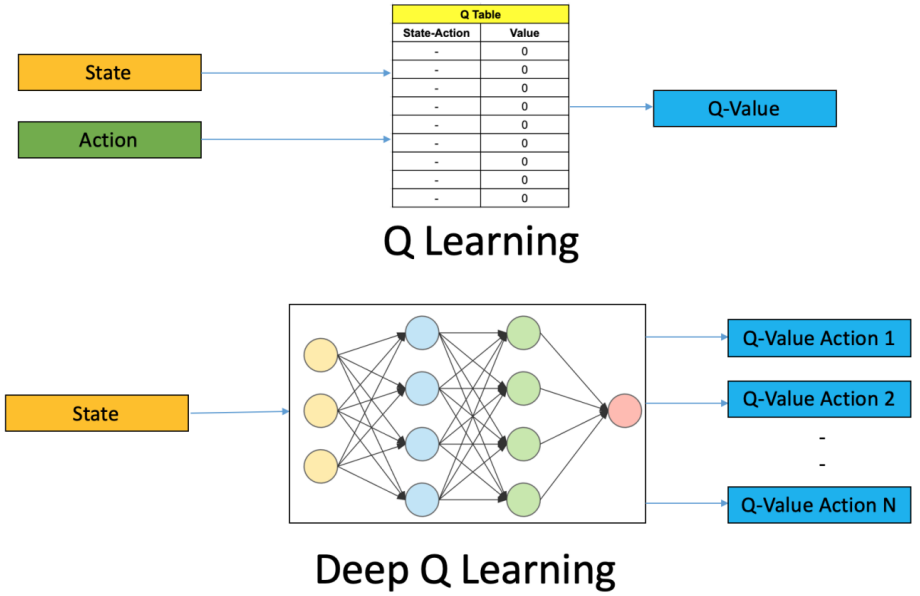


Figure 19.6: Q learning and deep Q learning comparison.

19.5.2.1 state utility

$$U(s) \approx U_\theta(s) \tag{19.19}$$

It is not straightforward to define a proper loss function for this problem. The professor in the slide writes this formula:

$$E(s, s') = \frac{1}{2}(R(s) + \gamma U_\theta(s') - U_\theta(s))^2 \tag{19.20}$$

This formulation allows us to learn by means of stochastic gradient descent procedure.

$$\nabla_\theta E(s, s') = (R(s) + \gamma U_\theta(s') - U_\theta(s))(-\nabla_\theta U_\theta(s)) \tag{19.21}$$

In this implementation of stochastic gradient descent, we sample *experiences* (s, a, s') instead of training set examples.

The stochastic gradient update rule is:

$$\theta = \theta - \alpha \nabla_\theta E(s, s') \tag{19.22}$$

19.5.2.2 action utility

$$Q(s, a) \approx Q(s, a)_\theta(s) \tag{19.23}$$

It is not straightforward to define a proper loss function for this problem. The professor in the slide writes this formula:

$$E((s,a),s') = \frac{1}{2}(R(s) + \gamma \text{max}_{a' \in \mathcal{A}} Q_\theta(s',a') - Q_\theta(s,a))^2 \tag{19.24}$$

This formulation allows us to learn by means of stochastic gradient descent procedure.

$$\nabla_\theta E((s,a),s') = (R(s) + \gamma \text{max}_{a' \in \mathcal{A}} Q_\theta(s',a') - Q_\theta(s,a))(-\nabla_\theta Q_\theta(s,a)) \tag{19.25}$$

In this implementation of stochastic gradient descent, we sample *experiences* (s,a,s') instead of training set examples.

The stochastic gradient update rule is:

$$\theta = \theta - \alpha \nabla_\theta E((s,a),s') \tag{19.26}$$