SCHOOL OF ENGINEERING

MACHINE LEARNING TECHNIQUES

Aude G. Billard

Illustrations based on the
MLDemos Software by Basilio Noris

I. Introduction

1 Forewords

These lecture notes have been written with the goal of providing a memento of the main concepts covered in the *Applied* and *Advanced Machine Learning* courses given at the EPFL. They do not constitute a textbook. We hence encourage students to refer to existing textbooks for complementary information on these techniques. A list of existing textbooks is given in Chapter **10**.

The course and these lecture notes are intended to students, who have completed their Bachelor in an engineering field. It hence assumes that the reader has a solid background in Linear Algebra, Calculus and Statistics. The annexes to these lecture notes give a brief summary of basic concepts from these fields used in the technical development covered in the lecture notes.

1.1 What is Machine Learning? - Definitions

Machine Learning (ML) encompasses a variety of techniques to enable machines (computers, robots, etc) to learn from a set of data. While it started as a topical curiosity of statistics and computer science in the 80's, it has matured over the past decades and is now a field of scientific study on its own.

It is broad in scope, as ML algorithms differ in both the type of computation performed and the way they perform these computations. ML entails algorithms for classification, clustering, feature extraction and regression, to name a few.

There is, hence, not a single definition of machine learning that is representative of this variety of applications and use of machine learning techniques. We can however list three definitions that encapsulate some of the key concepts behind the field:

Machine Learning is the field of scientific study that concentrates on induction algorithms and on other algorithms that can be said to "learn".

Machine Learning Journal, Kluwer Acadnemic

Machine Learning is an area of artificial intelligence involving developing techniques to allow computers to "learn". More specifically, machine learning is a method for creating computer programs by the analysis of data sets, rather than the intuition of engineers. Machine learning overlaps heavily with statistics, since both fields study the analysis of data.

Webster Dictionary

Machine learning is a branch of statistics and computer science, which studies algorithms and architectures that learn from data sets.

WordIQ

These three definitions are similar in that they emphasize the idea of *learning from data.* This implies that, on the one hand, we have data, and on the other hand, we process these data so as to extract some useful information from which we can "learn". In any learning process, including the one you undergo when taking courses at EPFL, you need to be provided with relevant material to be able to learn from it. Hence, gathering an *informative* dataset is a key issue in machine learning. In Section 1.3, we will discuss issues related to choosing and using the dataset to obtain optimal performance. Additionally, the practice sessions that accompany the courses will be

devoted to help you to acquire a practical understanding of the effect of choosing a poor or good dataset on the algorithms' performance. Let us first start by giving you're an intuition of what is meant for a machine to "learn".

1.2 What is Learning?

We start by citing two definitions provided by the machine learning community to situate how "learning" is understood in the context of ML.

> *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 tasks in T, as measured by P, improves with experience E.*
>
> *Mitchell, Tom. M. 1997. Machine Learning. New York: McGraw-Hill.*

> *Things learn when they change their behavior in a way that makes them perform better in the future.*
>
> *Witten, Ian H., and Eibe Frank. 2000. Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations. San Diego, CA: Morgan Kaufmann.*

Both definitions emphasize the notion of learning *from experience*. Learning hence depends on gathering data and is inhenrently an incremental process. The second definition implies also that this process leads to an improvement. Note that this raises one important issue. To measure improvement in performance, one must have a metric to measure performance. The metric of performance is key to guide how the system may change its behavior when incorporating new data, and it must be chosen with care.

1.2.1 Taxonomy of learning algorithms

Machine learning algorithms are often organized according to taxonomy, based on the desired outcome of the algorithm. Common algorithm types include:

- ***Supervised learning*** – where the algorithm learns a function or model that maps best a set of inputs to a set of desired outputs.
- ***Reinforcement learning –*** where the algorithm learns a mechanism that generates a set of outputs from one input in order to maximize a reward value (external and delayed feedback)
- ***Unsupervised learning*** – where the algorithm learns a model that best represents a set of inputs without any feedback (no desired output, no external reinforcement)
- **Learning to learn** – where the algorithm learns its own inductive bias based on previous experiences

In all types of learning, it is important that the training examples contain enough information to enable the system to find the solution, and that they are representative of the complexity of the whole dataset in order to avoid an overgeneralization.

Machine learning algorithms can also be classified according to the type of computation they can perform on a given dataset. Common types of computation include:

- **Features extraction**: finding relationship within the data

- **Classification:** learn to put instances into pre-defined classes
- **Clustering:** discover classes of instances that belong together
- **Regression:** Predict a continuous output given a set of input. This can be used for predicting time series or trends in data.
- **Association**: learn relationships between the attributes

1.2.2 Other important terms in machine learning

On-line learning: an algorithm is said to work on-line or in **real-time**, when it can perform its computation on the data as they stream through. Typical example is an algorithm for vision that could be performed without slowing down the 30 frames per second flow of images.

Connectionist models, more commonly called **Artificial Neural Networks (ANN),** form a large part of the algorithms of machine learning. ANNs encompass a diverse set of algorithms that share the same principle of computation, *pseudo-parallel* computation.

In connectionist models, unsupervised learning is also known as **self-organization**, a principle by which the network self-organizes to best represent the data. Examples of self-organized networks are Kohonen, Hopfield and other Associative Memories. Examples of supervised networks are feed-forward and recurrent neural networks with Backpropagation.

1.2.3 Key features for a good learning system

There are a number of desiderata one could formulate for any given learning system, such as:

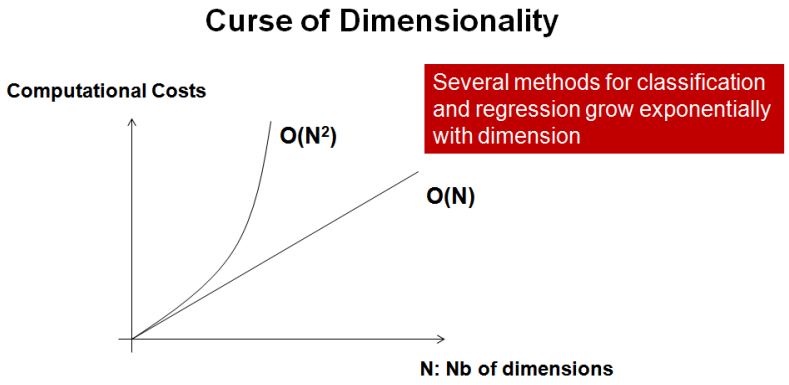
Denoising: Working in a real world application implies noise. Noise can mean several things. It can be imprecision in the measurement (e.g. light affecting infra-red sensors). It can be side effects from the experimental setups (e.g. systematic bias in the initial conditions). Eliminating noise is probably the most crucial processing one might want to perform on the data, prior to any other processing. However, it has its costs. Denoising requires noise model. It is seldom the case that the model is known. There are methods to learn these models, but the process is slow. An incorrect noise model is bound to eliminate good data with the noise, or get rid of too little noise.

Decorrelating: Decorrelating the data is often a first step before denoising or proceeding to any feature extraction. The process of decorrelation aims at making sure that, prior to analyzing the data, you have found a way to represent the data that explicitly encapsulate the correlations. Data with low correlations can often be considered as noise, or of little relevance to modeling the process.

Generalization versus memorization: A general and important feature of a learning system that differentiates it from a pure “memory” is its ability to generalize. Generalizing consists of extracting key features from the data, matching those across data (to find resemblances) and storing a generalized representation of the data features that account best (according to a given metric) for all the small differences across data. Classification and clustering techniques are examples of methods that generalize through a categorization of the data. Generalizing is the opposite of memorizing and often one might want to find a tradeoff between over-generalizing, hence losing information on the data, and over fitting, i.e. keeping more information than required. Generalization is particularly important in order to reduce the influence of noise, introduced in the variability of the data.

Feature extraction Part of the process of learning consists in extracting what is relevant in a set of data. This is often done by finding what is common across all the examples used for learning. The *common* **feature** to all data may consist of complex patterns (e.g. when training an algorithm to recognize faces, you may expect the algorithm to learn to extract features such as nose, eyes, mouth, etc). The process of finding what is common relies usually on a measure of correlation across data points. Extracting correlation can often reduce the dimensionality of the dataset at hand, by saving only a subpart of all the information.

Curse of dimensionality A problem which one often encounters in machine learning is what one calls the curse of dimensionality, whereby the number of samples used for training should grow exponentially with the dimensionality of your data set to remain representative. This exponential increase is reflected back into an exponential increase of the computation steps for training. Both of these are usually not practical and, as we will see in this class, many algorithms have been developed to reduce the dimensionality of the data prior to further processing. Alternatively, efforts are made when designing new algorithms to ensure that the growth of the computation cost is reduced by finding simplification in the computation. The ideal is a linear growth. Algorithms that achieve this are highly praised.



1.2.4 Exercise

For the following scenarios: 1) learning to drive a bicycle; 2) learning how to open a box with a lever; 3) learning sign language (if lost on a Island with only deaf people), determine:

a) The variables at hand (input and output to the system),
b) The unknown parameters of the model,
c) A good measure of performance,
d) A criteria of “good enough” optimality,
e) A threshold of sub-optimality (“too poor”)
f) The minimal time lag

1.3 Best Practices in ML

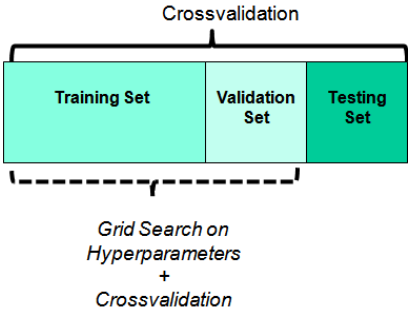
ML algorithms are sensitive to the choice of data you use for training them. Ideally, you would like your training set to be sufficiently large to be *representative* of all possible data you may encounter (In a probabilistic view, this means that you want your sample to be representative of the distribution of the data you try to estimate.) In practice, this is not feasible. For instance, imagine that you want to train an algorithm to recognize human faces. When training the algorithm, you can use only a subset of all faces you may encounter in life. Yet, you would still like your algorithm to generalize a

model of “human faces”. If you provide the algorithm with too many examples of the same type of faces (e.g. by training the algorithm only on faces of people with long hair), the algorithm may **overfit,** i.e. it may learn that faces all have long hair and fail to latter recognize faces of people with short hair. Each time an algorithm cannot detect correctly instances of a given class (e.g. human faces with short hair) is called a **false negative**.

If the algorithm was not provided with representative examples of what is not a face, it may incorrectly classify images of other objects as faces. For instance, in the above example, an algorithm that retained that the feature "long hair" is representative of human faces may incorrectly classify a picture of a horse as a human face. Such mistakes are called **false positives.** In ML, you try to balance the number of false negatives and false positives and one uses the ROC curve to measure this, see Section 3.5.1. In some applications, you may want to ensure to have no **false positives,** even at the cost of increasing the **false negatives**, or conversely**.** For instance, if you develop an application for identity verification, you want to make sure that no identity is mistakenly associated with the wrong person.

Finally, since ML algorithms are essentially looking at correlations across data, they may fit spurious correlations, if provided with a poorly chosen set of training data. To ensure that the algorithm has generalized correctly over the subsets of training examples you used, a number of good practices have been developed, which we review briefly below.

1.3.1 Training, validation and testing sets and crossvalidation.



Common practice to assess the validity of a Machine Learning algorithm is to measure its performance against three data sets, the *training, validation* and *testing sets.* These three sets are disjoint partitions of all the data at hand.

Training and validation sets are used to determine the sensitivity of the learning to the choice of *hyperparameters (i.e. parameters not learned during training)*. Values for the hyperparameters are set through a *grid search* (i.e. by sampling uniformly all values that each parameter can take within some fixed interval). The effect of each set of parameters on the performance of the algorithm is measured through *crossvalidation* (see below).

Once the optimal hyperparameters have been picked, the model is trained with the complete set (training + validation sets) and tested on the testing set.

The testing set consists of a subset of the data which you would normally encounter once training is completed. It allows to measure the sensitivity of the algorithm specifically for this application. Let us consider again the example we used before, where we trained an algorithm to recognize human faces. To proceed to the first stage of training, one would typically choose a set of different faces representative of all gender, ethnicities and other fashionable additions (hair cuts, glasses, moustache, etc), collected from a variety of type of cameras, so as to provide a generic set of examples of pictures of face independent of the material used to collect these. The testing set could consist of faces recorded by the camera of the client to whom you plan to sell the algorithm after training it in the laboratory and is hence used to validate the generalization properties of your algorithm to this specific type of camera.

In practice, one often uses solely training and testing sets and performs crossvalidation directly on these, while relying on grid search without crossvalidation (on training + validation sets) to determine the optimal set of hyperparameters.

There is no fixed rule to determine how to split the dataset. In practice, people tend to give more datapoints to the training set than to the validation set (and similarly to the testing set), using the $2/3^{rd}$ – $1/3^{rd}$ rule, i.e. with twice more datapoints for training than for validation or testing. But you can try to reduce this ratio. The *smaller the training/testing ratio required for good generalization, the better the model.*

Increasing the training/testing ratio provides the algorithm with more information and usually leads to better performance at training. However, this can be detrimental at testing and lead to **overfitting**. The training/testing ratio required to achieve good performance is indicative of the complexity of the dataset. If one needs to use at least 80% or more of the datapoints to achieve good generalization, this indicates that the dataset is very complex, since almost all datapoints are needed to build a representative model. In this case, one should collect more data to provide more statistics.

1.3.2 Crossvalidation

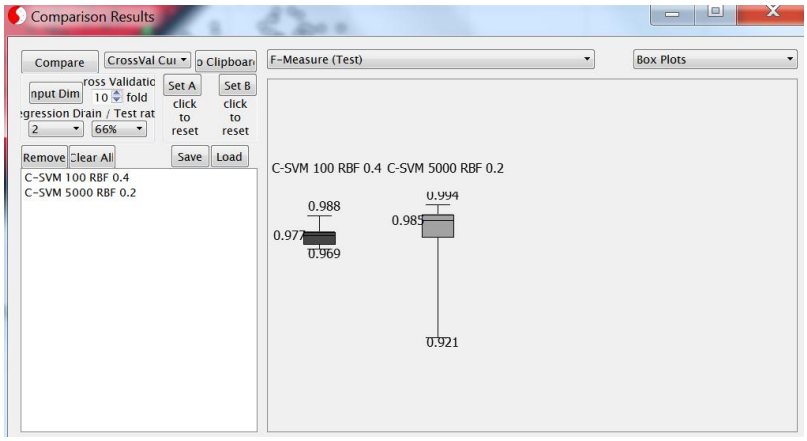
To ensure that the splitting of data into training and validation sets was not a lucky split leading to an optimal (but not representative) performance, one must apply a *crossvalidation* step. *"Crossvalidation refers to the practice of confirming an experimental finding by repeating the experiment using an independent assay technique*" (Wikipedia). In ML, this consists of splitting the dataset several times at random into training and validation sets. The number of repetitions is referred to as *the number of folds.* When one proceeds to 10 such random draws of training and validation sets, one says that one performs 10-fold crossvalidation.

K-fold cross validation: The data set is divided into *k* subsets and the training/validation steps is repeated *k times.* Each time, one of the *k* subsets is used as the test set and the other *k-1* subsets are put together to form a training set. Then the average error across all *k* trials is computed. The advantage of this method is that it matters less how the data gets divided. Every data point gets to be in a test set exactly once, and gets to be in a training set *k-1* times. The variance of the resulting estimate is reduced as *k* is increased. The disadvantage of this method is that the training algorithm has to be rerun from scratch *k* times, which means it takes *k* times as much computation to make an evaluation. A variant of this method is to randomly divide the data into a test and training set *k* different times. The advantage of doing this is that you can independently choose how large each test set is and how many trials you average over. In this class, we use the random variant of K-fold crossvalidation.

Leave-one-out cross validation is K-fold cross validation taken to its logical extreme, with K equal to N, the number of data points in the set. The evaluation given by leave-one-out cross validation error is very informative and comprehensive, but it is overly expensive if you use algorithms that are very expensive to compute at training time

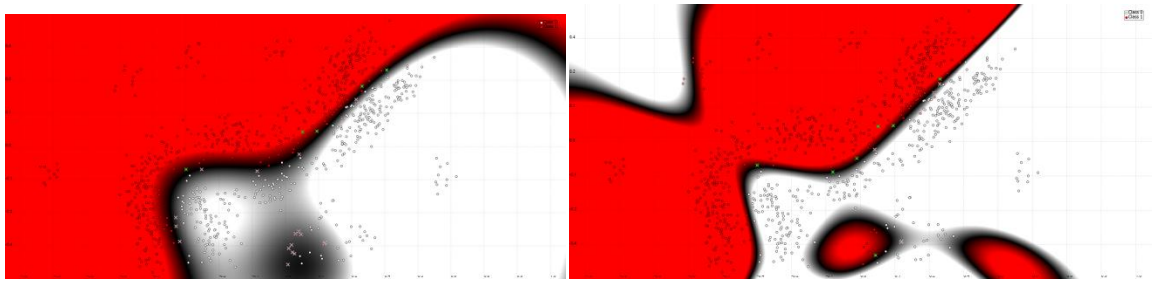
As for the choice of training/validation sets ratio, there is no fixed rule to determine what would be the optimal number of folds. People tend to use 10-fold crossvalidation. However, this makes sense only if the number of datapoints is large enough for such a large set of repetitions to draw groups of datapoints that are statistically significantly different at each round.

The performance of the algorithms is then measured on the whole set of experiments, e.g. by taking the mean and standard deviation of the error when applying the algorithm on each set. The standard deviation is important as it gives a measure of potential discrepancies in the structure of your data set. A large variance is indicative of *overfitting,* as shown in the example below. The crossvalidation phase is used to fine tune parameters of the algorithm (e.g. thresholds, priors, learning rates, etc) to ensure optimal performance.



The plot on the left shows the result of 10-fold crossvalidation with a 66% training/testing ratio for two different sets of hyperparameters in a binary classification problem, using SVM as classifier. The secon model that uses a large penalty (C=5000) and very small kernel width (0.2) (see Section 5.7 on SVM) leads to overfitting.

This overfitting is visible through the large standard deviation on the error.



The two plots above display the result of the classification for the first and second models (left and right figures respectively). On the right, we see that the model fitted perfectly the small set of datapoints of the red class located in the middle of the white class, leading to overgeneralization on the bottom right of the distribution.

1.3.2.1 Ground Truth

Comparing the performance of a novel method to existing ones or trivial baselines is crucial. Whenever possible, one will use the *ground truth* as a means to compare performance of the system during training.

In classification, the ground truthcan be obtained for instance from manually labeling each data with the appropriate class label. If the choice of class label may be subjective (e.g. labeling facial expressions of happiness, sadness, etc), one may ask several raters to label the picture and one may take the average score across raters for the “true” label.

1.3.3 Performance measures

Performance measures depend on the method you use. We will present performance measure for clustering, semi-supervised clustering and classification in their respective chapters.

1.3.4 Other practical considerations

This class will allow you to program some ML algorithms. Here are a few caveats to bear in mind when programming ML algorithms.

A large number of algorithms we will see in class require knowing the mean and covariance of the probability distribution function of the data. In practice, the class means and covariances are not known. They can, however, be estimated from the training set. Either the maximum likelihood estimate or the maximum a posteriori estimate may be used in place of the exact value. Further,

several of these algorithms assume that the underlying distribution follow a normal distribution. This again is usually not true. Thus, one should keep in mind that, although the estimates of the covariance may be considered optimal, this does not mean that the resulting computation obtained by substituting these values is optimal, even if the assumption of normally distributed classes is correct.

Another complication that you will often encounter when dealing with algorithms that require computing the inverse of the covariance matrix of the data is that, with real data, the number of observations of each sample exceeds the number of samples. In this case, the covariance estimates do not have full rank, and so cannot be inverted. There are a number of ways to deal with this. One is to use the pseudoinverse of the covariance matrix. Another way is to proceed to *singular value decomposition*.

2 Methods for Dimensionality Reduction

This chapter will introduce you to basic techniques used to perform *reduction of dimensionality*. As discussed in the introduction of these lecture notes, reducing the dimension of dataset is a main goal in Machine Learning. Indeed, most machine learning algorithms suffer from the *curse of dimensionality.* This refers to the fact that the number of model's parameters to estimate grow usually exponentially with the dimension of the data. Techniques for reducing the dimensionality of the problem aim, hence, at reducing the dimension of the space in which each datapoint is encoded. To reduce the dimension of the space, these techniques try to determine the minimum number of dimensions required to encode each datapoint while incurring minimum loss. This minimum loss can be measured in different ways. In Principle Component Analysis (PCA) it is measured in terms of mean-square error when reconstructing the dataset. In other probabilistic techniques (such as Gaussian Mixture Models), which we will see in later chapters of these lecture notes, loss is measured in terms of minimum discrepancy in likelihood term when estimating the density of datapoints.

Data are usually encoded in thousands of dimensions. Take, for example the problem of separating a set of pictures of faces from two different persons, as shown in Figure 2-1. Pictures are high-dimensional, as they are composed of millions of pixels. Standard cameras usually take pictures in a 320x240 format, which results in 76800 pixels. If the image is in color, this number must be multiplied by 3 as each pixel has an associated 3 color channel code RGB (Red-Green-Blue channel), yielding a dimension of 230400. If we apply Principal Component Analysis (PCA), a method for reducing the dimension through linear projection (which we will cover in Section 2.2.1), and look at the projections of each data picture along the first two projections (see Figure 2-1 right) we see that the picture s of the first person (in red) can be easily separated from the picture s of the second person (in green) by drawing a line. When sole a line is sufficient to distinguish between two groups of datapoints, one calls this a *linearly separable* problem. As we will see when tackling Classification methods in Chapter 3, being able to separate *linearly* is preferred over non-linear separation as it requires less parameters to code for a line than it does for coding a complex curve.

The example of Figure 2-1 shows that the true dimension of the problem was orders of magnitude smaller than the original dimension of the problem. Projecting the data through PCA reduced the dimension from 230400 to 2, hence a gain of more than 99% of memory required to store the projected data.

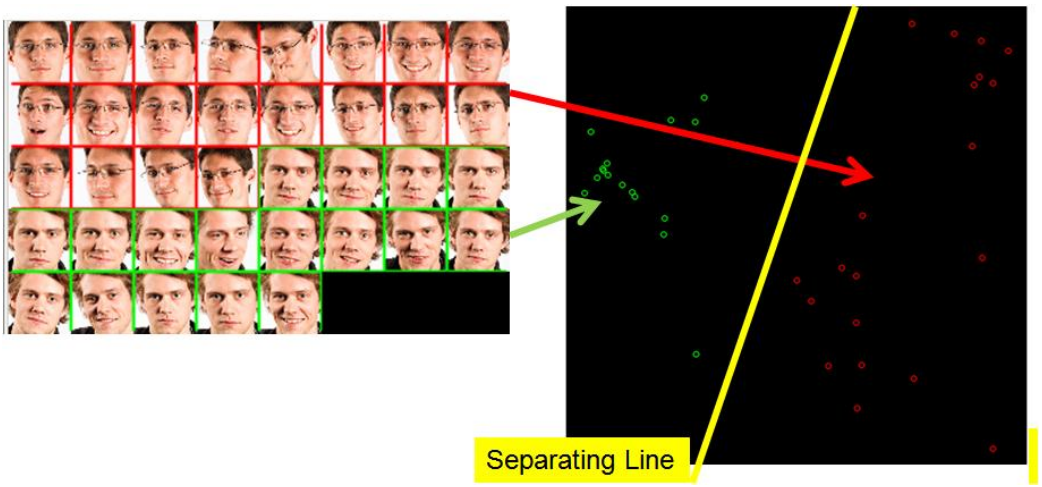


Figure 2-1*: Example of two-class classification problem which can be approached by first projecting the high-dimensional points from camera pictures (left) into a low-dimensional space (2-dimensional, right), where they can be separated through a line.*

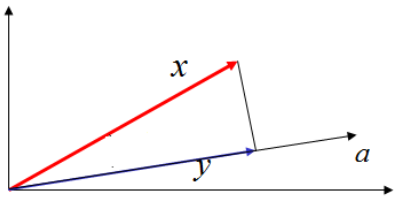
In this chapter, we present one technique for reducing the dimensionality of the dataset which is based on *linear* projections only. Extensions of these techniques for non-linear projection will be presented in long version of these lecture notes, dedicated for advanced course in ML. Next, we introduce the general formalism of linear projection techniques.

2.1 Formalisation of Reduction of Dimensionality through Linear Projection

Projection in 1-dimension: Recall that if you project a vector point *x* onto a projection vector ***a***, the image *y* of *x* on ***a*** is given by:

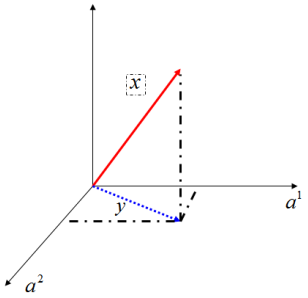
$$y = \frac{a^T \cdot x}{\|a\|^2} a \ .$$

The figure on the right illustrates this projection.



Projection in 2-dimensions: The projection onto a plane defined by the pair of linearly independent and *orthogonal* vectors $\left(a^1, a^2\right)$ with $\left(a^1\right)^T a^2 = 0$ is given by:

$$y = \underbrace{\frac{\left(a^1\right)^T \cdot x}{\|a^1\|^2}}_{\substack{\text{coordinate} \\ \text{of y onto } a^1}} a^1 + \underbrace{\frac{\left(a^2\right)^T \cdot x}{\|a^2\|^2}}_{\substack{\text{coordinate} \\ \text{of y onto } a^2}} a^2$$



If the axes of projection $\left(a^1, a^2\right)$ are **orthonormal**, the numerators disappear and Equation **Error! Reference source not found.** simplifies to: $y = \left(a^1\right)^T \cdot x \cdot a^1 + \left(a^2\right)^T \cdot x \cdot a^2$. As we will PCA uses orthonormal axes of projection.

Projection in larger dimensions:

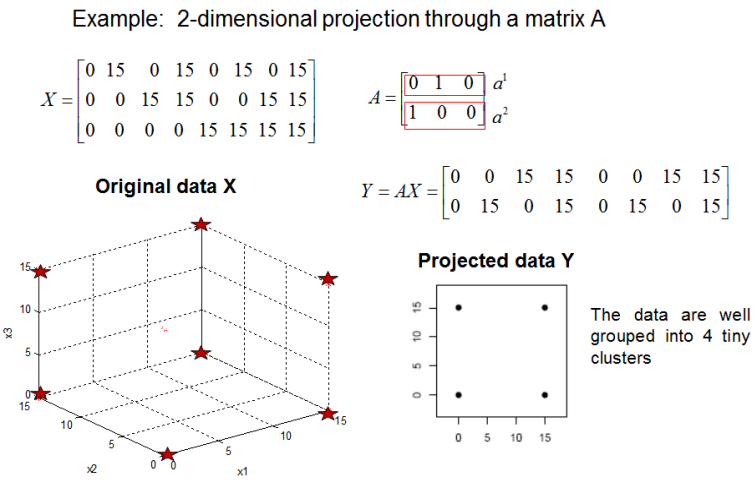
The result in 2-dimensions generalizes to higher dimensions as long as the axes of projections are orthogonal. It is hence interesting to project onto a basis formed of orthogonal axes. As we will PCA and its variants choose an orthogonal basis (other techniques, such as Independent Component Analysis, relaxes this constraint and requests only that the axes of projections be linearly independent). We now formalize the principle of linear projection onto an orthonormal basis:

Consider a dataset *X* of *M* datapoints, i.e. $X = \{x^i\}_{i=1}^{M}$. Each datapoint is *N-dimensional*, i.e. $x^i \in \mathbb{R}^N$, $i = 1...M$. We order each datapoint in a matrix *X* whose columns are composed of the each datapoint x^i. A linear projection of the dataset onto a low-dimensional space is performed by applying a projection matrix $A: p \times N,\ p < N$ onto X. The *image* Y of X is given by:

$$Y = AX \tag{2.1}$$

Y is a matrix with M column vectors $y^i \in \mathbb{R}^P$, $i = 1...\mathrm{M}$, each of which corresponds to the image of each original vector x^i.

Each row of the matrix *A* corresponds to the projection axes onto which the data is projected. Below is an example of such projection:



If we apply this formalism to the example of Figure 2-1, we have *M=37* original datapoints x^i of dimension *N*=230400. The images of these datapoints when projected onto a low dimensional space, *p=2*, correspond to a set of 2-dimensional points.

2.2 Principal Component Analysis

Principal Component Analysis (PCA) is a technique for reducing the dimensionality of a multidimensional data set by identifying a suitable *projection* of the data in which the data are *de-correlated*. To do this, PCA will seek to determine directions along which the data is correlated!

Extracting correlations across dimensions has two advantages a) it allows to reduce the dimensionality, as if one dimension is correlated with another dimension, you need to know only

one of the two dimensions (e.g. assume you have a dataset made of blue balls and red boxes. Since all balls in your dataset are blue and all boxes are red, then, detecting the color of the object is sufficient to infer that the type of object and you can discard information about the objects' shape); b) in high-dimensional datasets, correlations in subset of the dimensions correspond to features. Extracting these features is useful to group datapoints that share these features. PCA can then be used as pre-processing method to project data onto dimensions where their features are easily extractale. This make it easier then to cluster or classify the data.

2.2.1 Principal Component Analysis - Formalism

Consider a data set of M N-dimensional data points

$$\mathrm{X}=\left\{x_j^i\right\}_{j=1,\ldots N}^{i=1\ldots M} \text{ and } \mathrm{x}^i \in \mathbb{R}^N, \; i=1,\ldots,M.$$

PCA aims at finding a matrix A, such that:

$$A\colon \mathbb{R}^N \xrightarrow{A} \mathbb{R}^q, \text{ with } \mathrm{q} \le N$$
$$\mathrm{X} \xrightarrow{A} Y = AX, \quad \text{with } Y = \left\{y^1,\ldots,y^M\right\} \text{ and each } y^i \in \mathbb{R}^q$$

Since there exists an infinite number of possible projection matrices A, we need a criterion to limit our choice. PCA sets as criterion to determine the projection that *incurs minimum loss, by minimizing reconstruction error (error measured in norm-2)*. As we will see later, this is equivalent to looking for the projection that *maximizes the variance of the projected data*.

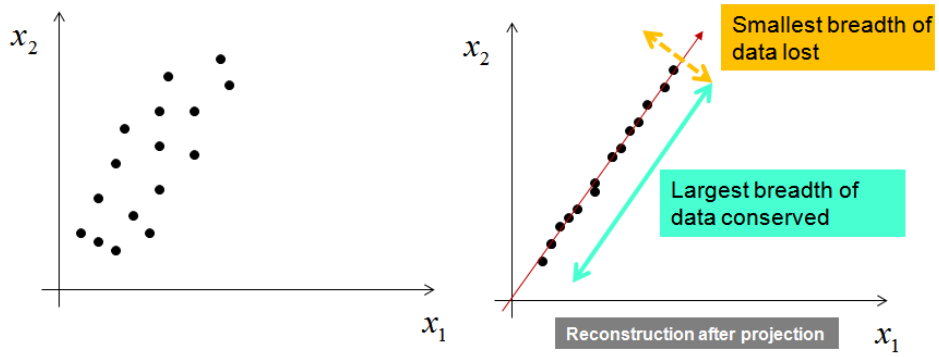


Figure 2-2: (left) original data; (right) data reconctructed in $\mathbb{R}^2$ from the projection on the red axis.

Figure 2-2 above illustrate the principle of minimum loss in PCA. The data (shown in the right figure) are projected along the red axes (left figure) in a projection $A:\mathbb{R}^2 \to \mathbb{R}$. There is a loss of information since the distribution of the original data along the direction perpendicular to the red axis has become zero. This is however the minimum loss one can incur as any other projection would result in a larger fraction of the data' spread lost.

The criterion for minimum loss can be expressed mathematically by computing the mean square error between the original data and the reconstructed one after projection.

If x is an original datapoint and y its image after projection through A, then we write $y^* = \begin{bmatrix} y \\ 0_{N-p} \end{bmatrix}$ the least square reconstruction vector. The loss incurred by projecting the data through *A* and removing the N-p dimensions is then given by:

$$\left\| A^{-1} y^* - x \right\| \tag{2.2}$$

PCA solves then the following constrained minimization:

$$\min_A \left\| A^{-1} y^* - x \right\| \tag{2.3}$$

$$\text{with } A = \begin{bmatrix} \left(e^1\right)^T \\ \left(e^2\right)^T \\ . \\ . \end{bmatrix} \text{ and } \begin{cases} \left\| e^i \right\| = 1, \ \forall i \\ \left(e^i\right)^T e_j = 0, \ \forall i \neq j \end{cases}$$

As mentioned previously, the rows of *A* consist of orthonormal vectors that form a basis of $\mathbb{R}^p$. The image of the datapoint can hence be expressed in this basis as:

$$y = \sum_{i=1}^{p} \left(\left(e^i\right)^T x \right) e^i \tag{2.4}$$

Replacing (2.4) into (2.3), and asking that all projection axes be orthogonal, $\left(e^i\right)^T e^j = 0, \ i \neq j$ we get:

$$\Rightarrow \min_{e^p,\ldots,e^N} \left\| \sum_{i=p}^{N} \left(\left(e^i\right)^T x \right) e^i \right\| = \min_{e^p,\ldots,e^N} \sum_{i=p}^{N} \left\| \left(\left(e^i\right)^T x \right) e^i \right\|$$

$$\Rightarrow \min_{e^p,\ldots,e^N} \sum_{i=p}^{N} \left(\left(e^i\right)^T x e^i \right)^T \left(\left(e^i\right)^T x e^i \right) = \min_{e^p,\ldots,e^N} \sum_{i=p}^{N} \left(e^i\right)^T x x^T e^i$$

The same procedure can be applied to compute the error when projecting the set *X* of *M* datapoints:

$$\min_{e^p,\ldots,e^N} \frac{1}{M} \sum_{i=p}^{N} \sum_{j=1}^{M} \left(\left(e^i\right)^T x^j e^i \right)^T \left(\left(e^i\right)^T x^j e^i \right) = \min_{e^p,\ldots,e^N} \sum_{i=p}^{N} \left(e^i\right)^T \frac{1}{M} \sum_{j=1}^{M} x^j \left(x^j\right)^T e^i$$

Observe that $C = \frac{1}{M} \sum_{j=1}^{M} x^j \left(x^j\right)^T = \frac{1}{M} X X^T$ is the covariance matrix of the dataset, when the dataset has zero mean. Hence one first preprocessing step in PCA will be to *center* the data, i.e. to substract the mean of the dataset, so that it is zero-mean.

The constrained based optimization problem expressed in (2.3) becomes:

$$\min_{e^j} \left(e^j\right)^T Ce^j$$
under the constraints that:
$$\left\|e^i\right\| = 1, \ \forall i \qquad (2.5)$$
$$\left(e^i\right)^T e_j = 0, \ \forall i \neq j$$

We can solve this iteratively (for each projection vector) using *Lagrange*. We start with the first vector e^1 and ask that the first projection vector satisfies only the normality condition (i.e. that its norm is 1). This gives us:

$$L\left(e^1, \lambda\right) = \left(e^1\right)^T Ce^1 - \lambda\left(\left(e^1\right)^T e^1 - 1\right)$$
$$\frac{\partial L\left(e^1, \lambda\right)}{\partial e^1} = Ce^1 - \lambda e^1 = 0$$

The solution is given by:

$$Ce^1 = \lambda e^1 \qquad (2.6)$$

The first projection vector is hence an eigenvector of the Covariance matrix!

Observe now that, when we proceed to an eigenvalue decomposition of the covariance matrix, we end up with *N* eigenvectors. All these eigenvectors are orthogonal to one another. As a result, performing an eigendecomposition of the covariance matrix *C* is the solution to the constrained optimization problem expressed in (2.5). All eigenvectors of the covariance matrix are projection vectors that satisfy our objective function (minimal reconstruction error) under the constraint that they be orthonormal. We now have all at our disposal to proceed to PCA. We give the steps of the algorithm next.

Algorithm:

Classical batch algorithm for PCA goes as follows:

Step 1: Center the data and then compute the covariance matrix

The mean of the dataset is denoted by

$$\mu = \left(\mu_1, \mu_1, ..., \mu_N\right) = E\left(X\right) = \left(\frac{x_1^1 + .. + x_1^M}{M}, ..., \frac{x_N^1 + .. + x_N^M}{M}\right) \qquad (2.7)$$

We substract the mean from the data and we get the new matrix of centered datapoints:
$X' = X - \mu, \ \text{where } X' \text{ is zero mean.}$

The covariance matrix of the centered dataset is:

$$C = E\left\{X'\left(X'\right)^T\right\} = \frac{B^T B}{M}, \qquad \mathrm{B} = \begin{pmatrix} x_1^1 - \mu_1, ..., x_N^1 - \mu_N \\ ..... \\ x_1^M - \mu_1, ..., x_1^M - \mu_N \end{pmatrix} \qquad (2.8)$$

Each diagonal entry of the covariance matrix *C*, denoted c_{ii}, corresponds to the variance of the data points along the dimension *i.*

The off-diagonal components of *C*, denoted by $c_{ij} = \frac{1}{M}\sum_{k=1}^{M}(x_i^k - \mu_i)(x_j^k - \mu_j)$, are a measure of *the correlation of the datapoints across the dimensions i and j.* If they are uncorrelated, their covariance is zero, i.e. $c_{ij} = c_{ji} = 0$.

The covariance matrix is symmetric and positive semi-definite. It has, thus, an orthogonal basis, defined by *N Eigenvectors* e^i, $i = 1,...N$ with associated eigenvalues λ_i:

$$Ce^i = \lambda_i e^i \tag{2.9}$$

The Eigenvalues λ_i are calculated by solving the equation:

$$\left|C - \lambda I\right| = 0 \tag{2.10}$$

When *I* is the $N \times N$ identity matrix and $\left|\ \right|$ the determinant of the matrix.

If the data vector has *N* components, the characteristic equation becomes of order *N*. This is easy to solve only if *N* is small. Note that, nowadays, you have numerous code at your disposal to perform this eigenvalue decomposition automatically. When *N* is really large and if we know that we care only about a subset of the eigenvectors, one can proceed directly to a singular value decomposition, where we search for a subset p of projections.

Singular value decomposition is also needed when the covariance matrix is not full rank. Recall that to compute the covariance matrix, we substract the mean. For the matrix to be full rank, we need to have at least N-1 independent observations. When N is high, it may happen that the number of observations (datapoints) is small compared to N, such as when using PCA for projecting high-dimensional images.

2.2.2 Dimensionality Reduction

To reduce the dimensionality of the data, we must discard a subset of dimensions. PCA chooses the dimensions that incur a minimum reconstruction loss as explained previously.

By discarding the *N-p* eigenvectors with smallest eigenvalues $\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_p \geq \lambda_{p+1} \geq \lambda_N$ and projecting onto the eigenvectors $e^1,...,e^p$ one can reduce the dimensionality of the dataset to a subspace that contains only the major directions of variation of the data. This can be useful in order to reduce noise, as it is likely that the noise is encapsulated in the lower dimensions of the dataset.

We can measure how much information is lost, by looking at the eigenvalues.

Each projection Y_i of X on e^i is given by $Y_i = \left(\left(e^i\right)^T X\right)e^i$. The percentage of the dataset covered by each projection is $\dfrac{\left\|X^T e^i\right\|}{\left\|\sum_j X^T e^j\right\|}$.

If we expand the numerator, we have $\left(X^T e^i\right)^T X^T e^i = \left(e^i\right)^T XX^T e^i$.

Using the fact that each eigenvector $XX^T e^i = M\lambda_i e^i$ and $\forall\ (i,j),\ i \neq j\ \left(e^i\right)^T e^j = 0$, we have

$$\frac{\left\|X^T e^i\right\|}{\left\|\sum_j X^T e^j\right\|} = \frac{\lambda_i}{\sum_j \lambda_j}.$$

Each eigenvalue gives us the amount of information contained in the projection. The loss of information when projecting onto the first *p* eigenvectors is hence given by the ratio: $\sum_{i=1}^{p} \lambda_i \Big/ \sum_{i=1}^{N} \lambda_i$

Hence, by ordering the eigenvectors in the order of descending eigenvalues (largest first), one can create an ordered orthogonal basis with the first eigenvector having the direction of largest variance of the data. The eigenvector corresponding to the largest eigenvalue is aligned with the direction along which the variance of the data is maximal, see Figure 2-1. In this figure, the directions of the two eigenvectors are indicated by the arrows in the right plot; the first eigenvector having the largest eigenvalue points to the direction of largest variance (the longest axis of the ellipse) whereas the second eigenvector is orthogonal to the first one (pointing to the second axis of the ellipse). The right plot shows the data after projection onto the first eigenvector, one we have discarded the projections onto the second eigenvector.

Example of PCA for data compression: We describe this procedure for dimensionality reduction when applied to one single data vector *x* and show how PCA, in this case, can be used to compress data (a typical example of use of PCA for *data compression)*:

Let *A* be a matrix, whose row vectors are composed of the eigenvectors of the covariance matrix as, i.e.:

$$A = \begin{pmatrix} e^1 \\ e^2 \\ \dots \\ e^N \end{pmatrix} \tag{2.11}$$

Let us transform the data vector x into y in the orthogonal coordinate system defined by the eigenvectors:

$$y = A\left(x - \mu\right) \tag{2.12}$$

The components $y_i,\ i=1...N$ of y are the coordinates of x in the orthogonal basis formed by the eigenvectors.

In order to reconstruct the original data vector x from y, one must compute:

$$x = A^T y + \mu$$

using the property of an orthogonal matrix $A^{-1} = A^T$.

Now, instead of using all the eigenvectors of the covariance matrix, one may represent the data in terms of only a few basis vectors of the orthogonal basis. If we denote the reduced transfer matrix A_p, that contains only the *p* first eigenvectors. The reduced transformation is, thus: $y = A_p\left(x - \mu\right)$.

If *x* is a vector composed of the pixel of an image, after the above transformation, the image *y* of *x* lives in a coordinates system of dimension *p.* This provides a way to compress data without losing much information. Below is an example of PCA compression for one image, where only the first 10% of the eigenvectors were retained, leading to a 90% compression gain.

Original Image **Image compressed 90%**

Example dimensionality reduction for group of images: PCA can also be used to reduce the dimensionality of a dataset. Below, we show two examples of PCA projection of a dataset of images of faces and of colored ball. Looking at the eigenvectors provides information on the features extracted by each PCA projection. Recall, however, that the first projection is the one that minimizes reconstruction error and maximizes variance. It is hence representative of all datapoints and usually represents a mix between all the faces. If you want to use PCA to separate groups of images, you must often use the 2nd, 3rd, and other projections that will embed features that are not common to all the dataset.

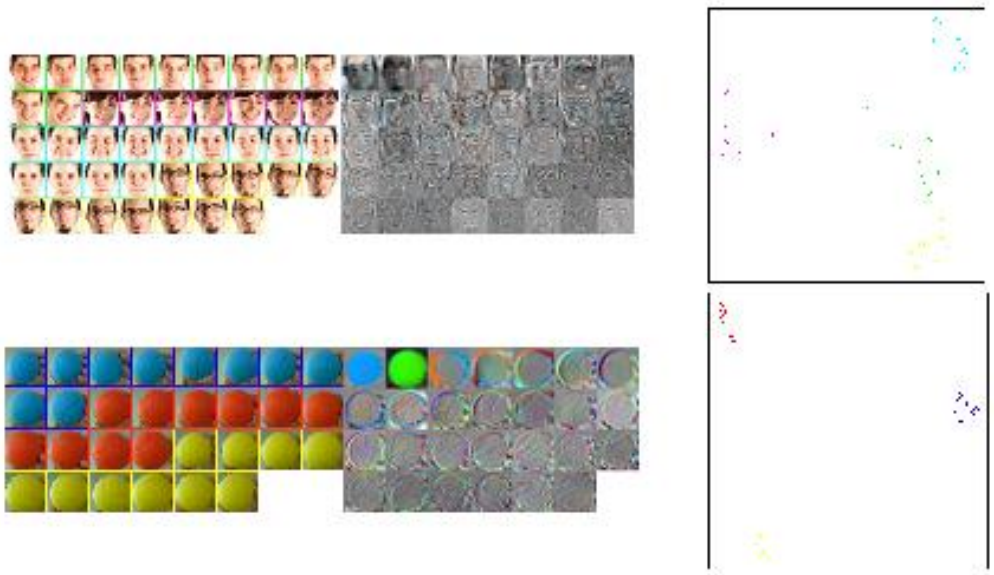


Figure 2-3: An example of dimensionality reduction using PCA. The source images (left) are 32x32 color pixels. Each image corresponds to a 3072 dimensional vector. (center) the principal components shown in decreasing order of eigenvalue, notice how the first components contain the main features of the data (e.g. color of the balls) while the components further down only contain fine details. (right) projection of the source images onto the first two principal components.

When applying techniques for reducting the dimension of the data, we are faced with contradictory goals: On the one hand, we should simplify the problem by reducing the dimension of the representation. On the other hand, we want to preserve as much as possible of the original information content. PCA offers a convenient way to control the trade-off between loosing information and simplifying the problem at hand.

Reconstructing an approximation of the original datapoint after projection:

If we want to reconstruct x from the projection y, we proceed as follows:

First, recall that:

$$A = \begin{bmatrix} \left(e^1\right)^T \\ \left(e^2\right)^T \\ . \\ . \end{bmatrix} \quad \text{with} \begin{cases} \left\|e^i\right\| = 1, \ \forall i \\ \left(e^i\right)^T e_j = 0, \ \forall i \neq j \end{cases}$$

A is $N \times N$. As A is square and orthogonal, $A^{-1} = A^T$.

When reducing the dimensionality, we have pruned the $N - p$ eigenvectors. We obtained $y \in \mathbb{R}^p$, a p-dim. vector. A_p is $N \times p$, and is composed of the p eigenvectors: ${A_p}^{-1} = {A_p}^T$.

We can reconctruct $x \in \mathbb{R}^N$ through least-square approximation as $x = {A_p}^T y$.

2.2.2.1 Data onto PCA projections is uncorrelated

As stated in introduction to this chapter, projecting onto the vectors generated by PCA de-correlates the data.

Proof: let us start by decomposing the covariance matrix $C = V\Lambda V^T$ with V the matrix of eigenvectors and Λ the matrix of eigenvalues.

The image Y of a dataset X is given by: $Y = AX$ with $A=V^T$.

The covariance matrix of Y is given by:

$$C_Y = YY^T = V^T XXV = V^T V\Lambda V^T V = \Lambda.$$

The covariance matrix in the projected space is hence diagonal. This means that all off-diagonal elements that correspond to correlation across dimensions are zero.

The projection vectors (eigenvectors) embed correlations across the original dimensions of the dataset (this is true only if there are correlations). Looking at the coordinate of these vectors onto the original dimensions gives us an idea of what relations it found across the original dimensions, see the exercise sessions of the course for examples.

2.2.2.2 Solving PCA as a variance maximization through constrained optimization

The PCA procedure delineated in the previous paragraphs was derived from solving an optimization under constraint problem, where we were minimizing the reconstruction error. As we explained earlier on, by projecting the zero-mean dataset onto the eigenvectors of its covariance matrix, PCA ensures that the first projection is along the direction of maximal variance of the data.

One can then also formulate PCA as an optimization problem that maximizes variance along the projections:

$$\arg\max_j J\left(e^1,...,e^N\right) = \arg\max_{j\in\{1,...N\}} \frac{1}{M}\sum_{i=1}^{M}\left\|\left(e^j\right)^T x^i\right\| = \left(e^j\right)^T Ce^j \tag{2.13}$$

with C the covariance of the dataset.

Adding the constraint that the e^j should form an orthonormal basis

$\left\|e^j\right\| = 1,\ \forall j = 1...q$ and $\left(e^k\right)^T e^j = 0\ \ \forall k \neq j.$

PCA becomes an optimization under constraint problem, which can be solved using Lagrange multipliers. PCA proceeds iteratively by first solving for the first eigenvector, using:

$$L\left(e^1\right) = \left(e^1\right)^T Ce^1 - \lambda_1\left(1 - \left(e^1\right)^T e^1\right) \tag{2.14}$$

where λ_1 is the first Lagrange multiplier. We end up with the same solution as found previously with the minimization of reconstruction error.

2.2.3 PCA limitations

PCA is a simple and straightforward means of determining the major dimensions of a dataset. It suffers, however, from a number of drawbacks. The principal components found by projecting the

dataset onto the perpendicular basis vectors (eigenvectors) are uncorrelated, and their directions orthogonal. The assumption that the referential is orthogonal is often too constraining, see Figure 2-4 for an illustration.

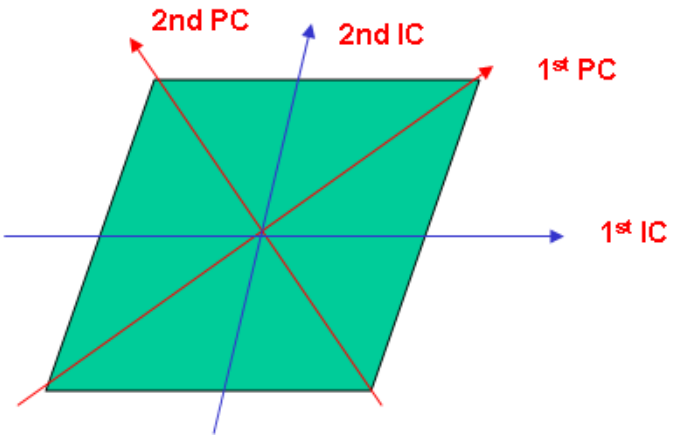


Figure 2-4: Assume a set of data points whose joint distribution forms a parallelogram. The first PC is the direction with the greatest spread, along the longest axis of the parallelogram. The second PC is orthogonal to the first one, by necessity. The independent component directions are, however, parallel to the sides of the parallelogram.

PCA ensures only uncorrelatedness. This is a less constraining condition than statistical independence, which makes standard PCA ill suited for dealing with non-Gaussian data. ICA is a method that specifically ensures statistical independence.

It is clear that PCA can be used as the basis of a clustering method, by grouping the points according to how they clump along each projection. Each cluster can then be treated separately. illustrates a situation in which PCA is useful in projecting the dataset onto a new frame of reference, from which the cluster of datapoints can be easily inferred. Figure 2-5 on the other hand shows an example in which PCA would fail to segment two clusters. By projecting the data along the principal direction, i.e. the vertical axis, PCA would merge the two clusters. ICA is one alternative method for separating such clusters.

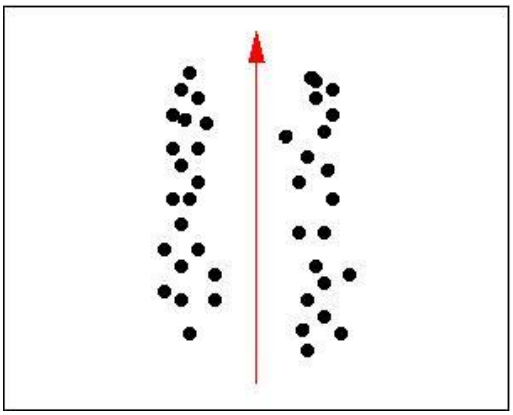


Figure 2-5: A classical illustration of problems occurring with variance-based methods such as PCA.

2.2.4 Projection Pursuit

Projection–Pursuit (PP) methods aim at finding structures in multivariate data by projecting them on a lower-dimensional subspace. Since there are infinitely many projections from a higher dimension to a lower dimension, PP aims at finding a technique to *pursue* a finite sequence of projections that can reveal the most “interesting” structures of the data. The interesting projections are often those that depart from that of normal/Gaussian distributions. For instance, these structures can take the form of trends, clusters, hypersurfaces, or anomalies. Figure 2-5 illustrates the ``interestingness'' of non-gaussian projections. The data in this figure is clearly divided into two clusters. However, the principal component, i.e. the direction of maximum variance, would be vertical, providing no separation between the clusters. In contrast, the strongly non-gaussian projection pursuit direction would be horizontal, providing optimal separation of the clusters. Independent Component Analysis (see Section 2.4) is a method for discovering a linear decomposition into directions that maximize non-Gaussianity. In this respect, it provides a first step for PP decomposition.

Typically, PP uses a *projection index*, a functional computed on a projected density (or data set), to measure the “interestingness” of the current projection and then uses a numerical optimizer to move the projection direction to a more interesting position. Interesting projections usually refer to non-linear structures in the data, i.e. structure than cannot be extracted by linear projections such as PCA and ICA. PP can be formalized as follows:

Given a dataset $X \subset \mathbb{R}^P$ and a unit vector $a \subset \mathbb{R}^P$ one defines an index $I_a : \mathbb{R}^P \to \mathbb{R}^P$ that measures the *interest* of the associated projection $P_a\left(X\right)$. PP finds the parameters a that maximize I_a.

PCA is an example of PP approach that takes the variance as the projection index. Further, if we consider a linear projection of each datapoint x through a of the form $z = a^T x$ and take the negative Shannon entropy as a measure for the index:

$$I_a = -\int f_z\left(z\right) \log f_z\left(z\right) dz \tag{2.15}$$

Where f_z is the probability density function of the projected data z, then PP is equivalent to Independent Component Analysis.

2.2.5 Probabilistic PCA

Until now, all variants of PCA we have seen had a deterministic nature. However, implicitly through the computation of the mean and covariance matrix of the data, one assumed that the data followed a distribution that could be parameterized through these two parameters. A distribution parameterized with this is the Gaussian distribution.

Here we will see how the standard PCA procedure can be extended with a probabilistic model. This rewriting will provide a first step toward introducing a series of methods based on so-called *latent variables*, which will see later on.

Latent variables correspond to unobserved variables. They offer a *lower dimensional* representation of the data and their *dependencies.* Fewer dimensions result in more parsimonious models. Probabilistic PCA (PPCA) is then PCA through projection on a latent space.

Formalism:

Asssume a N-dimenionsal data set $X \in \mathbb{R}^N$. X corresponds to the observations. Probabilistic PCA starts with the assumption that the data X were generated by a *Gaussian latent variable model* of the form:

$$x = Wz + \mu + \varepsilon \tag{2.16}$$

where $z \in \mathbb{R}^q$ are the q-dimensional Latent Variable,

W is a $N \times q$ matrix

$\mu \in \mathbb{R}^N$ is a vector of parameters

$\varepsilon \in \mathbb{R}^N$ is the noise and follows a zero mean Gaussian distribution $\varepsilon = \mathrm{N}\left(0, \Sigma_\varepsilon\right)$

Probabilistic PCA hence differs from PCA by assuming that a) the linear transformation through the matrix W goes from the latent variables to the observables; b) the transformation is no longer deterministic and is affected by random noise.

Note that the noise is a random variable with zero mean and fixed covariance. If one further assumes that the covariance matrix of the noise Σ_ε is diagonal, i.e. that the noise along each dimension is uncorrelated to the noise along the other dimensions, this leads to a *conditional independence* on the observables *given* the latent variables. In other words, the latent variables z encapsulate the correlations across the variables. Such conditional independence on the observables is advantageous for further processing, e.g. to proceed to an estimation of the likelihood of the model given the data. In this case, one can then simply take the product of the likelihood of the data for each dimension separately.

Probabilistic PCA consists then in estimating the density of the latent variable z. PPCA does so through *maximum likelihood*.

Algorithm:

If one further assumes that the noise follows an isotropic Gaussian distribution of the form $\mathrm{N}(0,\sigma_\varepsilon^2\mathrm{I})$, i.e. that its variance σ_ε^2 is constant along all dimensions. The conditional probability of the observables *X* given the latent variables $p(x\,|\,z)$ is given by:

$$p(x\,|\,z)=\mathrm{N}\left(Wz+\mu,\sigma_\varepsilon^2 I\right) \tag{2.17}$$

The marginal distribution can then be computed by integrating out the latent variable and one obtains:

$$p_z\left(x\right)=\mathrm{N}\left(\mu,WW^T+\sigma_\varepsilon^2 I\right) \tag{2.18}$$

If we set $\mathrm{B}=\mathrm{WW}^T+\sigma_\varepsilon^2 I,$ one can then compute the log-likelihood:

$$\mathrm{L}\left(B,\sigma_\varepsilon,\mu\right)=-\frac{M}{2}\left\{N\ln\left(2\pi\right)+\ln\left|B\right|+tr\left(B^{-1}C\right)\right\} \tag{2.19}$$

where $C=\frac{1}{M}\sum_{i=1}^{M}\left(x^i-\mu\right)\left(x^i-\mu\right)^T$ is the covariance matrix of the complete set of M datapoints $\mathrm{X}=\left\{x^1,...,x^M\right\}$.

The parameters $B,\ \mu$ and σ_ε can then be computed through *maximum likelihood,* i.e. by maximizing the quantity $\mathrm{L}\left(B,\sigma_\varepsilon,\mu\right)$ using expectation-maximization. Unsurprisingly, the maximum estimate of μ turns out to be the mean of the dataset.

The maximum-likelihood estimates of B and σ^2 are then:

$$B^*=W_q\left(\Lambda_q-\sigma_\varepsilon^2 I\right)^{\frac{1}{2}}R$$

$$\left(\sigma^*\right)^2=\frac{1}{N-q}\sum_{j=q+1}^{N}\lambda_j \qquad \text{(this is also called the residual)}$$

where W_q is the matrix of eigenvectors of C and the λ_j are the associated eigenvalues.

As in PCA, the dimension *N* of the original dataset, i.e. the observable *X,* is reduced by fixing the dimension $q<N$ of the latent variable. The conditional distribution of the latent variable given the observable is:

$$p\left(z\,|\,x\right)=\mathrm{N}\left(B^{-1}W\left(x-\mu\right),B^{-1}\sigma_\varepsilon^2\right) \tag{2.20}$$

where $B=\left(W\right)^T W+\sigma_\varepsilon^2 I.$

Finally note that, in the absence of noise, one recovers standard PCA. Simply observe that: $\left(\left(W\right)^T W\right)^{-1}W\left(x-\mu\right)$ is an orthonormal projection of the zero mean dataset, and hence if one sets $A=\left(\left(W\right)^T W\right)^{-1}W$, one recovers the standard PCA transformation.

2.3 Canonical Correlation Analysis

Adapted from Hardoon, Szedmak & Shawe-Taylor, *Neural Computation*, 16:12, p. 2639-2664, 2004

Canonical Correlation Analysis (CCA) is a technique for dimensionality reduction designed for data that are described in different modalities. An example of such multi-modal dataset is illustrated in Figure 2-6. Each datapoint is described by an image and an audio file, each of which lives in a high-dimensional space.

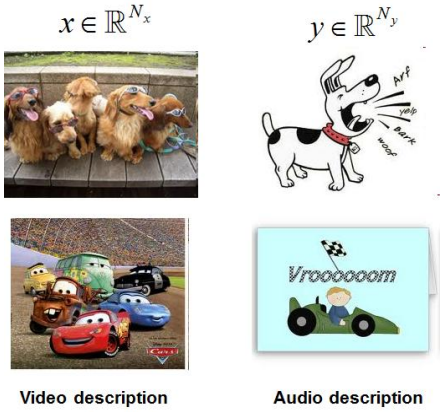


Figure 2-6 Example of a bimodal 2 datapoints dataset. Each datapoint is described by an image and an audio file, each of which lives in a high-dimensional space.

CCA can be used to find a projection in image and audio space separately. CCA uses the data labels to enforce that the data in each projected space correlate with one another. This preserves consistency across the projections. Data with similar labels will grouped similarly in each space.

Imagine that you want to develop a biometric system that can identify different people based on audio recordings of the persons' voices and sets of video recordings of each person's face when talking. Audio and video recordings are recorded simultaneously and hence when taken as a pair they may reveal some more information than taking each of these individually. CCA will aim at extracting the features in each dataset (audio and visual) that correlate best for each person individually. CCA can be seen as a generalized version of PCA for two or more multi-dimensional datasets. In each space (audio and video), CCA would find one or more projections that maximize correlation across pair of first, second, third eigenvectors in each basis. In other words, CCA tries to find a linear combination of audio features and a linear combination of video features that correlate best. Data once projected in each of these separate sets of eigenvectors would then be most representative of a person and hence can be used to best discriminate afterwards (e.g. by using the projected data in a classifier afterwards).

While PCA works with a single dataset and maximizes the variance of the projections of the data onto a set of eigenvectors forming a basis of this dataset, CCA works with a pair of random vectors determined in each basis separately and maximizes correlation between sets of projections. In other words, while PCA leads to an eigenvector problem, CCA leads to a generalized eigenvector problem.

2.3.1.1 Formalism:

Consider a pair of multivariate datasets $X=\left\{x^i \in \mathbb{R}^{N_x}\right\}_{i=1}^{M}, Y=\in\left\{y^i \in \mathbb{R}^{N_y}\right\}_{i=1}^{M}$ of which we measure a sample of *M* instances pairs $\left\{\left(x^i, y^i\right)\right\}_{i=1}^{M}$. CCA consists in determining a set of projection vectors w_x and w_y for X and Y such that the correlation ρ between the *projections* $z_x = w_x^T X$ and $z_y = w_y^T Y$ (the *canonical variates*) is maximized.

$$\max \rho = \max_{w_x, w_y} \operatorname{corr}\left(\mathrm{z}_x, \mathrm{z}_y\right) == \max_{w_x, w_y} \frac{\mathrm{w}_\mathrm{x}^\mathrm{T} E\left\{XY^T\right\} \mathrm{w}_\mathrm{y}}{\left\|\mathrm{w}_\mathrm{x}^\mathrm{T} X\right\|\left\|\mathrm{w}_\mathrm{y}^T Y\right\|} = \max_{w_x, w_y} \frac{\mathrm{w}_\mathrm{x}^\mathrm{T} C_{xy} \mathrm{w}_\mathrm{y}}{\sqrt{\mathrm{w}_\mathrm{x}^\mathrm{T} C_{xx} w_x \mathrm{w}_\mathrm{y}^\mathrm{T} C_{yy} w_y}} \tag{2.21}$$

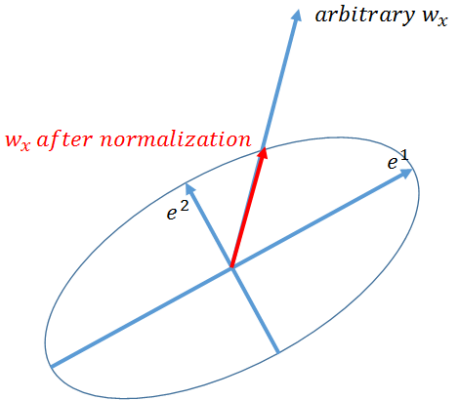
Where C_{xy}, C_{xx}, C_{xy} are respectively the inter-set and within sets covariance matrices. C_{xy} is $N_x \times N_y$, C_{xx} is $N_x \times N_x$ and C_{xy} $N_y \times N_y$.

Note that, for the above computation, we have assumed that both *X and Y* are zero-mean. This is a pre-processing step, which we already did for PCA and that simplifies the computation of the correlation, by removing the expectation terms on *X* and *Y.*

Given that the correlation is not affected by rescaling the norm of the vectors w_x, w_y, we can set that:

$$\left\|\mathrm{w}_\mathrm{x}^\mathrm{T} C_{xx} w_x\right\| = \left\|\mathrm{w}_\mathrm{y}^\mathrm{T} C_{yy} w_y\right\| = 1 \tag{2.22}$$

The above constraints bound the norm of w_x and w_y. This rescaling is equivalent to forcing the vectors to point on an ellipse with axes aligned with the eigenvectors of the covariance matrices C_{xx} and C_{yy}. The lengths of the ellipse's axes are given by the inverse of the square root of the eigenvalue of the associated eigenvectors. Applying this constraint amounts to scaling the norm of the vectors w_x and w_y so that they touch the ellipsoid isoline of value 1. It does not change the direction of the vectors w_x and w_y and hence does not change the relative correlation across pairs of vectors, as illustrated on the figure on the right.



The CCA algorithm consists thus in finding the optimum of ρ under the above two constraints. This again can be resolved through Lagrange and gives:

$$L\left(w_x, w_y, \lambda_x, \lambda_y\right) = w_x^T C_{xy} w_y - \lambda_x\left(w_x^T C_{xx} w_x - 1\right) - \lambda_y\left(w_y^T C_{yy} w_y - 1\right)$$

with λ_x, λ_y the Lagrange multiplier of the constraint.

Taking the partial derivatives over w_x, w_y and setting $\lambda_x = \lambda_y := \lambda / 2$, we get:

$$\begin{pmatrix} 0 & C_{xy} \\ C_{yx} & 0 \end{pmatrix} \begin{pmatrix} w_x \\ w_y \end{pmatrix} = \lambda \begin{pmatrix} C_{xx} & 0 \\ 0 & C_{yy} \end{pmatrix} \begin{pmatrix} w_x \\ w_y \end{pmatrix}$$

$\Rightarrow$ Which can be rewritten as $C_{xy} C_{yy}^{-1} C_{yx} w_x = \lambda^2 C_{xx} w_x$

(2.23)

This is a generalized eigenproblem of the form $Ax = \lambda Bx$. If C_{xx} is invertible (which is the case if all the columns of *X* are independent), then the above problem reduces to a single symmetric eigenvector problem of the form $C_{yx} C_{xx}^{-1} C_{xy} w_y = \lambda^2 C_{yy} w_y$, and conversely for w_x.

2.3.2 CCA for more than two variables

Given sets of *K* random variables $X^k \ \left(k = 1,...,K\right)$ with dimension $M \times n^k$ (note that while each variate can have different dimensions n^k, they must all have the same number of observed instances *M*). The generalized CCA problem consists then in determining the set of projection vectors $W^k = \left\{ w_i^k \right\}_{i=1...p}$, with $p = \min_{k=1..K} n_k$, such that:

$$\begin{aligned} &\min_{W} \ \sum_{l \neq k, k, l = 1,...,K} \left\| X^k W^k - X^l W^l \right\|_F \\ &\text{s.t.} \qquad \left(W^k\right)^T C_{kk} W^k = I, \ \ \forall k = 1,..,K, \\ &\qquad\qquad \mathrm{w}_i^{\mathrm{k}} C_{kl} \mathrm{w}_j^{\mathrm{l}} = 0, \ \ \forall i, j = 1,..., p, i \neq j. \end{aligned} \tag{2.24}$$

Where $\| \ \|_F$ is the *Frobenius norm (Euclidean distance for matrices)*. The above consists of K minimization under constraint problems. Unfolding the objective function, we have the sum of the squared Euclidean distances between all of the pairs of the column vectors of the matrices $X^k W^k, k = 1,..., K$. This problem can be solved by using *singular value decomposition* for arbitrary *K*.

2.3.3 Limitations

CCA is dependent on the coordinate system in which the variables are described, so even if there is a very strong linear relationship between two sets of multidimensional variables, depending on the coordinate system used, this relationship might not be visible as a correlation. Kernel CCA has been proposed as an extension to CCA, where data are first projected into a higher dimensional representation. KCCA is covered in Section 5.4 of these Lecture Notes.

2.4 Independent Component Analysis

Adapted from ***Independent Component Analysis,*** **A. Hyvarinen, J. Karhunen and E. Oja**, Wiley Inter-Sciences. 2001

Similarly to PCA, ICA is a linear transformation that projects the dataset on a sub-manifold, that best represent the data at hand. While PCA looks for the principal components, *ICA looks for the directions along which statistical dependence of the data is minimal.*

ICA is particularly useful in order to *decorrelate* and *denoise* data. ICA is very closely related to the method called *blind source separation* (BSS) or blind signal separation. The ``source'' is the original signal, i.e. the independent components, (e.g. the speaker in a cocktail party). ``Blind'' means that neither the mixing matrix nor the independent components are known to start with. ICA is one method, perhaps the most widely used, for performing blind source separation. It has been used successfully in a wide range of signal processing applications, e.g. for de-correlating multiple sound sources or extracting invariant features in image processing.

2.4.1 Illustration of ICA

,

To illustrate the idea of ICA, consider that you have observed four instances of a two-dimensional distribution *S* of independent components s_1 and s_2 and that the four instances form the four corners of a rectangle, as shown in Figure 2-7:

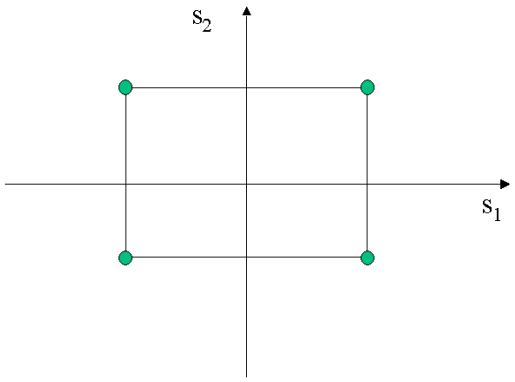


Figure 2-7; Distribution of two independent components

Note that the range of values for these data was chosen so as to make the mean zero and the variance equal to one, an important constraint of ICA. Now let us mix these two independent components, using the mixing matrix $A = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$. This gives us two mixed variables, x_1 and x_2.

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = A \cdot \begin{pmatrix} s_1 \\ s_2 \end{pmatrix}$$

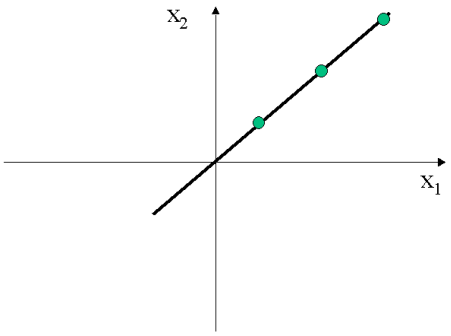


Figure 2-8: Mixture of variables

Note that the random variables x_1 and x_2 are not independent any more, see Figure 2-8; an easy way to see this is to consider, whether it is possible to predict the value of one of them, say x_2, from the value of the other. Clearly it is the case, as they line up along a line of slope 1.

The problem of estimating the data model of ICA is now to estimate the mixing matrix A using only information contained in the mixtures x_1 and x_2. In the above example, it is easy to estimate A by simply solving the inverse for each of the four data points, assuming that we have at least four points. The problem is less trivial, once we consider a mixture of two arbitrary continuous distributions.

Let us now assume that s_1 and s_2 were generated by the following uniform distribution:

$$p(s_i) = \begin{cases} \dfrac{1}{2\sqrt{3}} & \text{if } |s_i| \le \sqrt{3} \\ 0 & \text{otherwise} \end{cases} \tag{2.25}$$

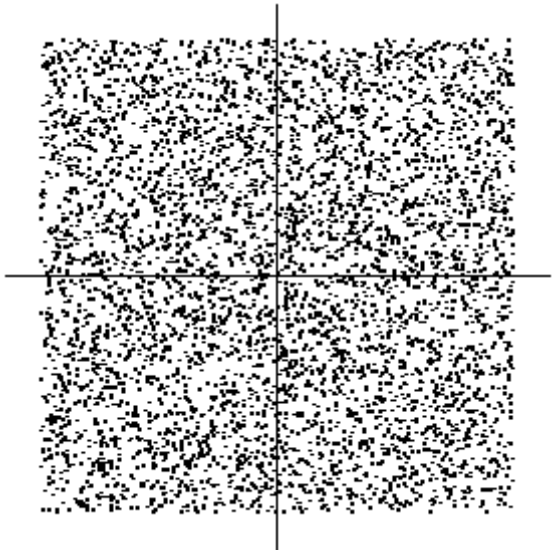


Figure 2-9: joint distributions of S1 and S2

If we now mix those two distributions according to $A = \begin{pmatrix} 2 & 3 \\ 2 & 1 \end{pmatrix}$, we have:

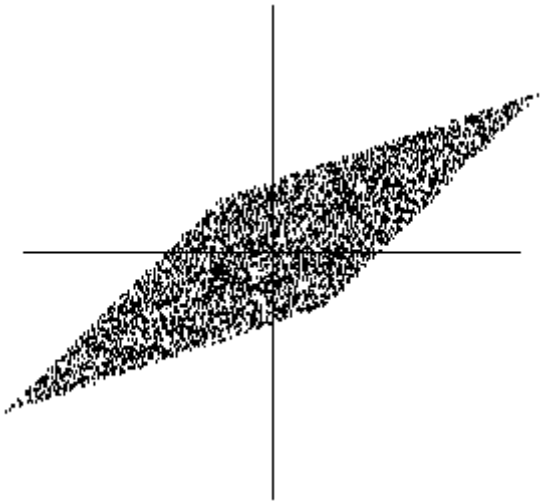


Figure 2-10: Joint distribution of the two observables X1 and X2

The edges of the parallelogram are in the directions of the columns of A . This means that we could, in principle, estimate the ICA model by first estimating the joint density of x_1 and x_2, and then locating the edges. So, the problem seems to have a solution.

In practice, however, this can serve solely as an intuition. Determining the edges explicitly works with variables that either follow uniform distributions or are sufficiently low dimensional that one can sample and observe the entire distribution. What we need is a method that works for any distribution of the independent components and works fast and reliably. Ideally, this should be a method that can be expressed as an optimization to automatically identify the sources and mixing matix.

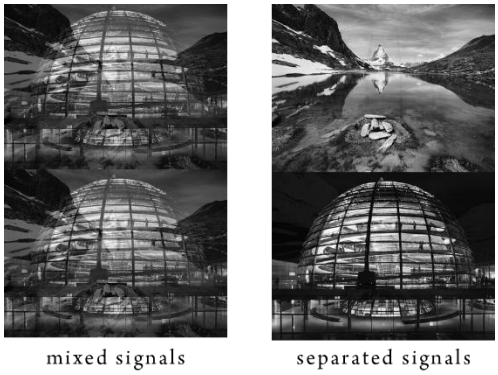


Figure 2-11: ICA can be applied to separate signals that have been mixed together (e.g. one image bleeding into another during wireless transmission of a video signal). [DEMOS\ICA\ICA_IMAGE_MIX.M]

2.4.2 Why Gaussian variables are forbidden

As mentioned above, a fundamental restriction of ICA is that the independent components must be non-Gaussian for ICA to be possible. To see why Gaussian variables make ICA impossible,

assume that the mixing matrix is orthogonal and the s_i are Gaussian. Then x_1 and x_2 are Gaussian, uncorrelated, and of unit variance. Their joint density is given by:

$$p(x_1, x_2) = \frac{1}{2\pi} e^{\left(-\frac{x_1+x_2}{2}\right)} \tag{2.26}$$

This distribution is illustrated in Figure 2-12. The Figure shows that the density is completely symmetric. Therefore, it does not contain any information on the directions of the columns of the mixing matrix *A*. This is why *A* cannot be estimated.

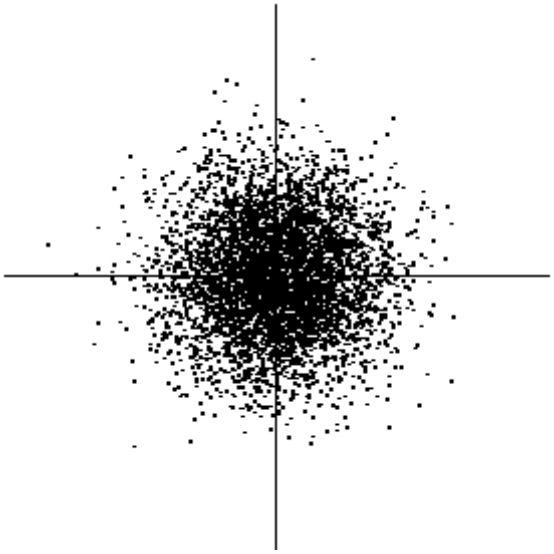


Figure 2-12: The multivariate distribution of two independent Gaussian variables

More rigorously, one can prove that the distribution of any orthogonal transformation of the Gaussian (x_1,x_2) has exactly the same distribution as (x_1,x_2), and that x_1 and x_2 are independent. Thus, in the case of Gaussian variables, we can only estimate the ICA model up to an orthogonal transformation. In other words, the matrix A is not identifiable for Gaussian independent components. Note that if just one of the independent components is Gaussian, the ICA model can still be estimated.

2.4.3 Definition of ICA

Let $x = \{x_1, ..., x_N\}$ be a N-dimensional random vector of observables.

ICA consists of finding a linear transform $s = Wx$ *so that the projections* $s_1, ..., s_q$ *of* x through *W are linearly independent.*

To proceed, one builds a general linear model of the form:

$$x = A \cdot s + \varepsilon$$

Where *A* is a $q \times N$ 'mixing' matrix, $s_1,...,s_q$ the set of independent components and ε a *N*-dimensional random noise vector[1]. The independent components are latent variables, meaning that they cannot be directly observed.

This formulation reduces the ICA problem to ordinary estimation of a latent variable model. Because the estimation of the latent variable in the noisy case can be very tricky, the majority of ICA research has concentrated on the noise-free ICA model, where:

$$x = A \cdot s \tag{2.27}$$

$$x_i = \sum_{j=1}^{q} a_{ij} s_j \tag{2.28}$$

The matrix *W* is then the inverse of *A,* i.e. $W = A^{-1}$ (this is possible only if *A* is invertible; to this end, one can ensure that *A* is full rank by reducing its dimension to that of the independent component, see below)..

Hypotheses of ICA:

The starting point for ICA is the following assumptions:

- Without loss of generality, we can assume that, both the mixture variables x and the independent components s have zero mean. Observe that the observable variables *x* can always be centered by subtracting the sample mean, i.e. $x = x' - E\{x'\}$. Consequently, the independent component have also zero mean, since $E\{s\} = A^{-1}E\{x\}$.
- The components s_i are statistically *independent*. Statistical independence is rigorously defined in Section 9.2.8. It is a stronger constraint than uncorrelatedness (which is ensured through PCA). Hence, ICA decomposition results usually in different estimates from that found through PCA decomposition.

- As discussed before, we must also assume that the independent components have *non-Gaussian* distributions. Usually, assuming that the data follow a Gaussian distribution is handy and is done in many other techniques we will see in this course (see e.g. PPCA or GMM). Gaussian distributions are so-called parametric distribution that is the distribution is fully determined once its parameters have been defined. Hence, assuming a Gaussian distribution simplifies the estimation of the density as one must solely estimates the parameters of the Gaussian (or mixture of Gaussians, as in GMM). Since ICA does not make any assumption regarding the form of the distribution of the independent component, it looks as if ICA would estimate the full density of *s*. This is a very difficult problem. ICA will overcome this difficulty by not estimating explicitly the density of *s.* Rather; the density of *s* can be recovered by sampling through *x* and using the inverse transformation.

- In our general definition of the ICA model given previously, we have assumed that *A* was a $q \times N$ matrix. Here, we will focus on a simplified version of ICA whereby one assumes that the unknown mixing matrix *A* is square, i.e. that the number of independent components is equal to the number of observed mixtures and thus *A* is. $q \times q$ (note that this assumption can be sometimes relaxed (see extensions proposed in Hyvarinen et al

[1] Note the similarity between this model and that introduced in PPCA, see Section 2.2.5**2.2.5**; the difference here lies in the optimization method whereby ICA optimizes for statistical independence and PCA optimizes for maximal variance.

2001). This can be done for instance by performing first PCA on the original dataset and then use a reduced set of *q* dimensions obtained by PCA for ICA. If *A* is square, then, after estimating the matrix A, one can compute its inverse, $W = A^{-1}$, and obtain the independent component simply by:

$$s = Wx = A^{-1}x \tag{2.29}$$

- The data is white, i.e. each datapoint is uncorrelated and the variance of the dataset is equal to unity. This will be a basic preprocessing step in ICA, as we will discuss next.

2.4.4 Whitening

A useful preprocessing strategy in ICA is to first whiten the observed variables. This means that before the application of the ICA algorithm (and after centering), we transform the observed vector x *linearly* so that we obtain a new vector $\tilde{x}$, which is white, i.e. its components are uncorrelated and its variance equal unity. In other words, the covariance matrix of $\tilde{x}$ equals the identity matrix:

$$E\left\{\tilde{x}\tilde{x}^T\right\} = I \tag{2.30}$$

The whitening transformation is always possible. One popular method for whitening is to use the eigen-value decomposition of the covariance matrix $E\left\{\tilde{x}\tilde{x}^T\right\} = UDU^T$, where *U* is the orthogonal matrix of eigenvectors of the basis of *x* and *D* is the diagonal matrix of its eigenvalues, $D = diag\left(\lambda_1,...,\lambda_n\right)$. Note that $E\left\{\tilde{x}\tilde{x}^T\right\}$ is the empirical means, i.e. it is estimated from the available data samples. Whitening can now be done by computing:

$$\tilde{x} = UD^{-\frac{1}{2}}U^T x \tag{2.31}$$

The matrix $D^{-\frac{1}{2}}$ is computed by a simple component-wise operation, such that:

$$D^{-\frac{1}{2}} = diag\left(d_1^{-\frac{1}{2}},...,d_n^{-\frac{1}{2}}\right).$$

It is easy to check that now $E\{\tilde{x}\tilde{x}^T\} = I$.

Whitening transforms the mixing matrix into a new one, $\tilde{A}$:

$$\tilde{x} = UD^{-\frac{1}{2}}U^T As = \tilde{A}s \tag{2.32}$$

The utility of whitening resides in the fact that the new mixing matrix $\tilde{A}$ is orthogonal. This can be seen from

$$E\left\{\tilde{x}\tilde{x}^T\right\} = \tilde{A}E\left\{ss^T\right\}\tilde{A}^T = \tilde{A}\tilde{A}^T = I \tag{2.33}$$

Here we see that whitening reduces the number of parameters to be estimated. Instead of having to estimate the N^2 parameters that are the elements of the original matrix A, we only need to

estimate the new, orthogonal mixing matrix $\tilde{A}$. An orthogonal matrix has $N(N-1)/2$ degrees of freedom. For example, in two dimensions, an orthogonal transformation is determined by a single angle parameter. In larger dimensions, an orthogonal matrix contains only about half of the number of parameters of an arbitrary matrix. Thus one can say that whitening solves half of the problem of ICA. Because whitening is a very simple and standard procedure, much simpler than any ICA algorithms, it is a good idea to reduce the complexity of the problem this way.

It may also be quite useful to reduce the dimension of the data at the same time as we do the whitening. For instance, one can look at the eigenvalues of $E\{xx^T\}$ and discard those that are too small, as is often done in the statistical technique of principal component analysis. This has often the effect of reducing noise. Moreover, dimension reduction prevents overlearning, which can sometimes be observed in ICA

A graphical illustration of the effect of whitening can be seen in Figure 2-13, in which the data in Figure 2-10 has been whitened. The square defining the distribution is now clearly a rotated version of the original square in Figure 2-9. All that is left is the estimation of a single angle that gives the rotation.

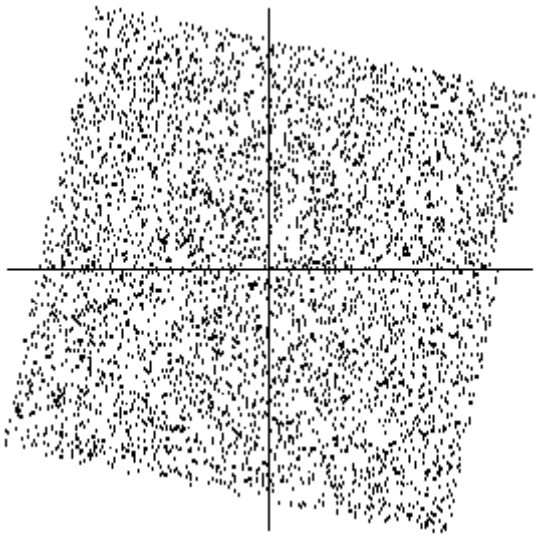


Figure 2-13: The joint distribution of the whitened mixtures

In the rest of this chapter, we assume that the data has been preprocessed by centering and whitening. For simplicity of notation, we denote the preprocessed data just by x, and the transformed mixing matrix by A, omitting the tildes.

2.4.5 ICA Ambiguities

We cannot determine the variances of the independent components. The reason is that, both s and *A* being unknown, any scalar multiplier in one of the sources s_i could always be cancelled by dividing the corresponding column a_i of *A* by the same scalar, say α_i.

$$x = \sum_i \left(\frac{1}{\alpha_i} a_i \right) \left(s_i \cdot \alpha_i \right) \tag{2.34}$$

As a consequence, we may as well fix the magnitudes of the independent components; as they are random variables. The most natural way to do this is to assume that each component has unit variance, i.e. $E\left\{ s_i^2 \right\} = 1$. Then the matrix *A* will be adapted in the ICA solution methods to take into account this restriction. Note that this still leaves the ambiguity of the sign: we could multiply any of the independent components by -1 without affecting the model. This ambiguity is, fortunately, insignificant in most applications.

We cannot determine the order of the independent components.
The reason is that, again both s and *A* being unknown, we can freely change the order of the terms in the sum in (2.28) and call any of the independent components the first one. Formally, a permutation matrix P and its inverse can be substituted in the model to give $x = AP^{-1}Ps$. The elements of Ps are the original independent variables s_j, but in another order. The matrix AP^{-1} is just a new unknown mixing matrix, to be solved by the ICA algorithms.

ICA properties are:

- *Redundancy reduction:* reducing redundancy in a dataset has numerous advantages. When the data show no redundancy any longer, the correlations across the data is null. In other words, each data point is significant and encapsulates a relevant characteristic of the dataset. Moreover, such a dataset is noise-free, because noise affects all data similarly. The brain seems to rely particularly on redundancy reduction. As we will see in later on in these lecture notes, the capacity of Neural Networks based on Hebbian Learning is maximal with non-redundant datasets.

- *Project pursuit:* In a noise-free or non-redundant dataset, there might still be a number of features irrelevant to the task. Project pursuit is a method for determining the relevant directions along which lie the data.

2.4.6 ICA by maximizing non-gaussianity

A simple and intuitive way of estimating ICA is done through the FastICA method. FastICA is based on a fixed-point iteration scheme for finding a maximum of a measure of nongaussianity. We present this solution here. Note that other objective functions have been proposed in the literature to solve ICA, such as minimizing mutual information or maximizing the likelihood. We will not review these here. In Section 7.7.2, we will offer another solution to ICA using Artificial Neural Networks.

2.4.6.1 Negentropy

To obtain a measure of non-Gaussianity that is zero for a Gaussian variable and always nonnegative, one often uses a slightly modified version of the definition of differential entropy, called the negentropy. The negentropy *J* is defined as follows

$$J\left(y\right) = H\left(y_{gauss}\right) - H\left(y\right) \tag{2.35}$$

where y_{gauss} is a Gaussian random variable of the same covariance matrix as *y* to the above-mentioned properties. The negentropy is always non-negative, and it is zero if and only if *y* has a Gaussian distribution. Negentropy is invariant for invertible linear transformations.

The advantage of using negentropy, or, equivalently, differential entropy, as a measure of nongaussianity is that it is well justified by statistical theory. In fact, negentropy is in some sense the optimal estimator of nongaussianity, as far as statistical properties are concerned. The problem in using negentropy is, however, that it is computationally very difficult. Estimating negentropy using the above definition would require an estimate (possibly nonparametric) of the pdf of the independent component. Therefore, simpler approximations of negentropy are very useful. For instance, Hyvarinen et al 2001 propose to use the following approximation. For a given non-linear quadratic function *G*, then:

$$\begin{aligned} J(y) &\propto \left[E\left(G(y)\right) - E\left(G(y_{Gauss})\right)\right]^2 \\ J(y) &\propto \left[E\left(G(w^T x)\right) - E\left(G(y_{Gauss})\right)\right]^2 \end{aligned} \tag{2.36}$$

2.4.6.2 FastICA for one unit

To begin with, we shall show the one-unit version of FastICA. By a "unit" we refer to a computational unit. As we will see in Section 7.7.2, this can also be considered as the output of a neuron.

The FastICA learning rule determines a direction, i.e. a unit vector w such that the projection $w^T x$ maximizes nongaussianity. Nongaussianity is here measured by the approximation of the empirical measure of *Negentropy* on the projection, i.e. $J\left(w^T x\right)$.

Recall that the variance of $w^T x$ must here be constrained to unity; for whitened data this is equivalent to constraining the norm of w to be unity.

The FastICA is based on a fixed-point iteration scheme for finding a maximum of the nongaussianity of $w^T x$. It can be also derived as an approximative Newton iteration, minimizing for the derivative of the Negentropy, i.e. $\frac{dJ\left(y = w^T x\right)}{dw}$. Hyvarinen et al 2001 propose to use either of the two following quadratic functions:

$$G_1(y) = \frac{1}{a}\log\left(\cosh\left(a \cdot y\right)\right), \qquad 1 \le \mathrm{a} \le 2$$

$$G_2(y) = -e^{-\frac{1}{2}y^2}$$

Denote by *g* the derivative of the above two functions, we have:

$$g_1(u) = \tanh\left(a_1 u\right)$$

$$g_2(u) = ue^{\left(-\frac{u^2}{2}\right)}$$

where $1 \le a_1 \le 2$ is some suitable constant, often taken as a_1=1. These function are monotonic and hence particularly well suited for performing gradient descent.

The basic form of the FastICA algorithm is as follows:

1. Choose an initial (e.g. random) weight vector *W*.

2. Compute the quantity $w^+ = E\left\{xg\left(w^T x\right)\right\} - E\left\{g\left(w^T x\right)\right\} w$

3. Proceed to a normalization of the weight vector: $w = \frac{w^+}{\left\|w^+\right\|}$

4. If the weights have not converged, i.e. $\vec{w}(t-1)\cdot\vec{w}(t) \neq 1$, go back to step 2.

Note that it is not necessary that the vector converge to a single point, since *w* and -*w* define the same direction. Recall also that it is here assumed that the data have been whitened.

2.4.6.3 FastICA for several units

The one-unit algorithm of the preceding subsection estimates just one of the independent components, or one projection pursuit direction. To estimate several independent components, we need to run the one-unit FastICA algorithm using several units (e.g. neurons) with weight vectors $w_1,...,w_q$.

To prevent different vectors from converging to the same maxima we must *decorrelate* the outputs $w_1^T,...,w_q^T$ at each iteration. We present here three methods for achieving this.

A simple way of achieving decorrelation is a deflation scheme based on a Gram-Schmidt-like decorrelation. This means that we estimate the independent components one by one. When we have estimated *p* independent components, or *p* vectors $w_1,...,w_p$, we run the one-unit fixed-point algorithm for w_{p+1}, and after every iteration step subtract from w_{p+1} the ``projections" $w_{p+1}^T w_j w_j$, $j = 1,...p$ of the previously estimated *p* vectors, and then renormalize w_{p+1}:

$$\begin{aligned} &1.\ \text{Let } w_{p+1} = w_{p+1} - \sum_{j=1}^{p} w_{p+1}^T w_j w_j \\ &2.\ \text{Let } w_{p+1} = w_{p+1} / \sqrt{w_{p+1}^T w_{p+1}} \end{aligned} \tag{2.37}$$

In certain applications, however, it may be desired to use a symmetric decorrelation, in which no vectors are ``privileged" over others. This can be accomplished, e.g., by the classical method involving matrix square roots,

$$\text{Let } W = \left(WW^T\right)^{-1/2} W \tag{2.38}$$

where W is the matrix $\left(w_1,...,w_q\right)^T$ of the vectors, and the inverse square root $\left(WW^T\right)^{-\frac{1}{2}}$ is obtained from the eigenvalue decomposition of $WW^T = F\Lambda F^T$ as $\left(WW^T\right)^{-\frac{1}{2}} = F\Lambda^{-\frac{1}{2}}F^T$. A simpler alternative is the following iterative algorithm:

$$\begin{aligned} &1.\ \text{Let}\ \ W = W / \sqrt{\|WW^T\|} \\ &\text{Repeat 2. until convergence} \\ &2.\ \text{Let}\ \ W = \frac{3}{2}W - \frac{3}{2}WW^TW \end{aligned} \tag{2.39}$$

The norm in step 1 can be almost any ordinary matrix norm, e.g., the 2-norm or the largest absolute row (or column) sum (but not the Frobenius norm).

2.4.7 Further Readings

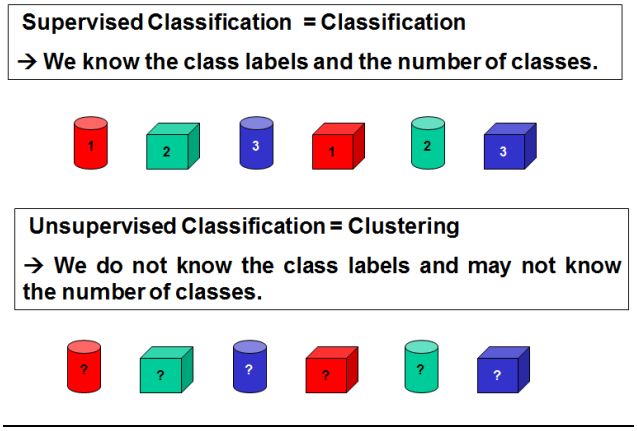
In this chapter, we have focused only on the linear version of ICA, and, on one method for solving ICA, namely fast ICA. Note that there exist also methods for non-linear ICA and for time-dependent ICA. The reader can refer to [Hyvarien et al, 2003] for further readings on ICA and its applications.

3 Clustering and Classification

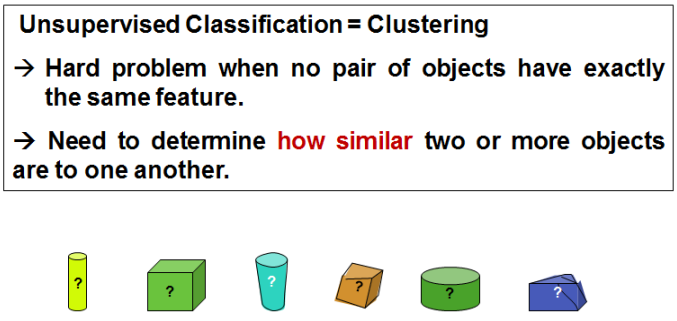
This chapter will present a series of methods to perform clustering and classification.

> *Clustering is a process of partitioning a set of data (or objects) in a set of meaningful sub-classes, called clusters. A cluster is a collection of data objects that are "similar" to one another and thus can be treated collectively as one group.*
>
> *Classification consists of modeling and categorizing data belonging to the same class.*

Clustering is oftern referred to as *unsupervised classification.* In contrast, classification relates to *supervised classification.* The notion of supervision/unsupervision in clustering and classification relates to the fact that in classification, we are provided with the *class labels,* whereas in clustering we do not have the class labels. This is illustrated in the schematic example below. A dataset of 6 objects are in one case already categorized by the user who attributed them class labels, grouping the objects according to their color. In the second case, the objects do not have a class label and one is left to infer what would be an adequate way of grouping the data. There are at least two different ways in which one could group these data. One could either group them according to their color (ending up with the same grouping as in the classification case) or one could group them according to shapes (ending up with only two groups with a mix of color).



Clustering is hard as we do not know the true class label and that there are many ways in which one could group data. Also, in real life, datapoints are never identical and they all resemble each other somehow, as illustrated in the schematic below:



The type of clusters you will create depends on the *metric of similarity* you choose for grouping your datapoints. If you choose to pre-process your data by applying PCA, for instance, then the choice of projection will highlight some features (correlations across the datapoints). This implicitely

will extract some measure of similarity across datapoints, as in some projections some datapoints will look closer to one another than in other projections, see example below:

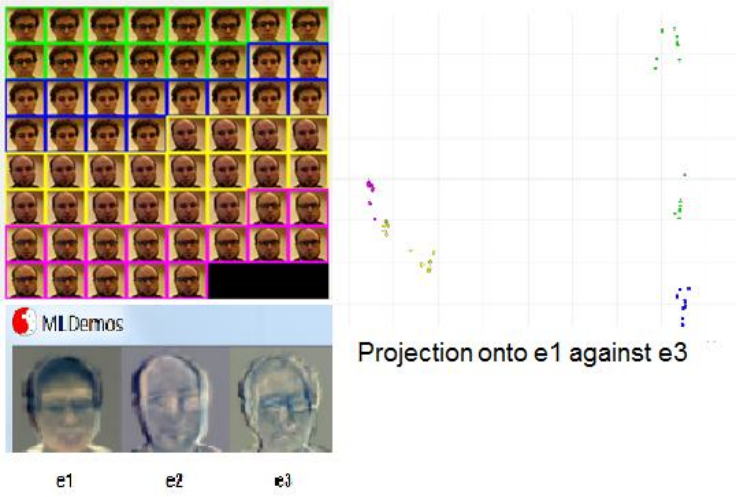
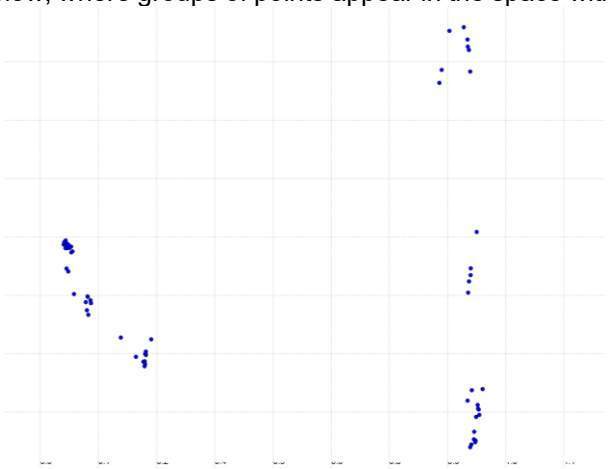


Figure 3-1: A database of images from two persons composed of 4 classes highlighted in colors is projected onto eigenvectors 1 and 3. When projecting onto e1 and e3, we can separate the image of the first person with and without glasses, as the eigenvector e3 embeds features distinctive of person1 primarily.

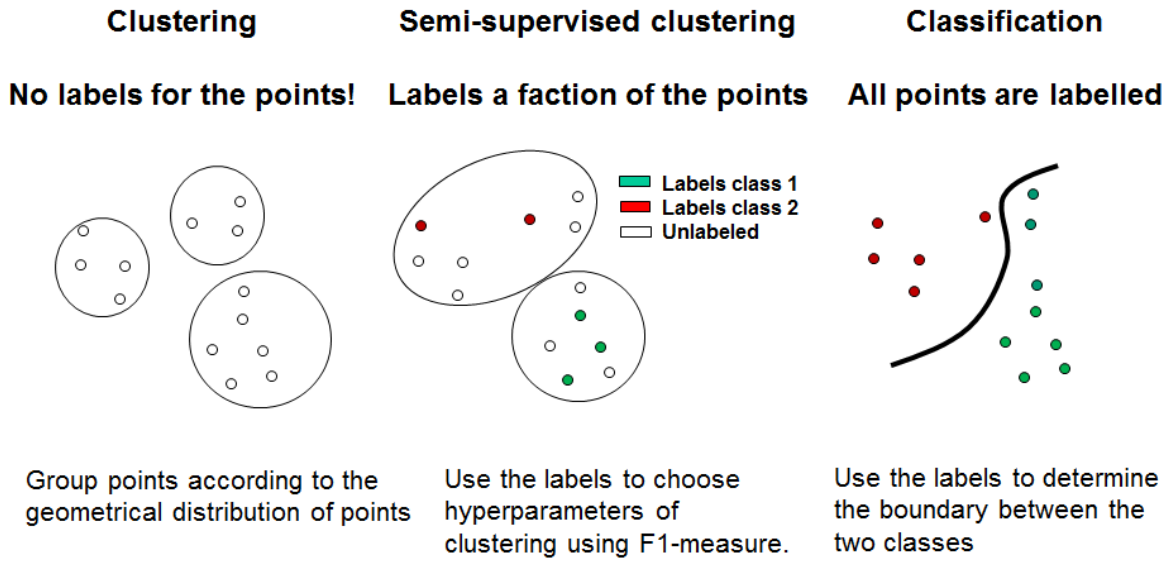
In this projection, it becomes easier to find clusters denoting the glasses versus non-glasses images of the first person. Do not forget that in clustering you do not have the class labels. Hence, the colors in the plot above would not be an information at your disposal and the real challenge would look like the figure below, where groups of points appear in the space without labels (colors):



3.1.1 Semi-supervised clustering

Since clustering is very hard job for lack of labels, whenever possible, one will try to gather labels. However, gathering labels may be much more time consuming than gathering the data. For instance, imagine that you are a company selling on-line a set of products such as movies, books and musics. To increase your sale, you may want to propose to your customer a set of items which you think “they may like”. To select which product you will propose, you need to get a representation of how items group with one another according to types of customers’ tastes. However, you do not know how many groups of customers and of products exists. To get a better idea of how many potential groups exists, you may pay a few customers to labels the items they find “similar”. While you may have thousands of customers on a yearly basis visit your website, only a small subset of them will be paid to provide their labels. You hence end up with a partial labeling of the data. To cluster the rest of your data, you will use this partial set of labeled data. Clustering methods that

use partially labeled dataset are referred to as semi-supervised clustering. The schematic below illustrates the difference between clustering, semi-supervised clustering and classification in a binary classification problem.



3.2 Types of Clustering Techniques

There exist many types of clustering techniques. In these lecture notes, we cover some examples that illustrate distinctive features across clustering techniques.

One important feature of a clustering technique is the type of clusters it creates.

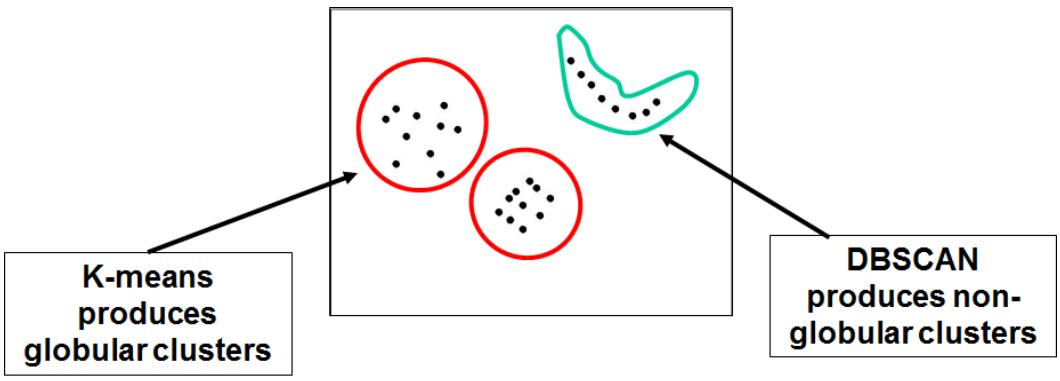


Figure 3-2: Example of a dataset that form 3 clusters

There are many ways to shape clusters. Clusters can be either *globular* (*convex)* such that any line you can draw between two cluster members stays inside the boundaries of the cluster, or *non-globular* (*concave),* i.e. taking any shape. Globular clusters, such as those produced by the K-means clustering technique we will see next, are advantageous in that they require few parameters to describe the cluster. A ball shape is described by its centroid and the radius to its centroid. However, a convex envelope is limiting and may not encapsulate well groups of points that are distributed along a non-convex shape. Non-globular clusters, such as those generated by the

DBSCAN technique, are advantageous in that they can produce tight boundaries of arbitrary shapes around the data. However, this comes at a price, with a large increase in computational cost, as we will discuss later on.

Clusters can contain an equal number of data, or be of equal size (distance between the data). Central to all of the goals of cluster analysis is the notion of degree of similarity (or dissimilarity) between the individual objects being clustered. This notion determines the type of clusters that will be formed.

In this chapter, we will see different methods of clustering that belong to two major classes of clustering algorithms, namely **hierarchical clustering** and **partitioning algorithms.** We will discuss the limitation of such methods. In particular, we will look at a generalization of K-means clustering, namely **soft k-means** and **mixture of Gaussians** that enable to overcome some but no all limitations of simple K-means.

While hierarchical algorithms build clusters gradually (as crystals are grown), partitioning algorithms learn clusters directly. In doing so, they either try to discover clusters by iteratively relocating points between subsets, or try to identify clusters as areas highly populated with data. Algorithms of the first kind are surveyed in the section Partitioning Relocation Methods. They are further categorized into **k-means methods** (different schemes, initialization, optimization, harmonic means, extensions) and **probabilistic clustering or density-Based Partitioning** (E.g. soft-K-means and Mixture of Gaussians). Such methods concentrate on how well points fit into their clusters and tend to build clusters of proper convex shapes.

When reading the following section, keep in mind that the major properties one is concerned with when designing a clustering methods include:

- Type of attributes algorithm can handle
- Scalability to large datasets
- Ability to work with high dimensional data
- Ability to find clusters of irregular shape
- Handling outliers
- Time complexity (when there is no confusion, we use the term complexity)
- Data order dependency
- Labeling or assignment (hard or strict vs. soft of fuzzy)
- Reliance on a priori knowledge and user defined parameters
- Interpretability of results

3.2.1 Hierarchical Clustering

In hierarchical clustering, the data is partitioned iteratively, by either *agglomerating* the data or by *dividing* the data. The result of such an algorithm can be best represented by a *dendrogram.*

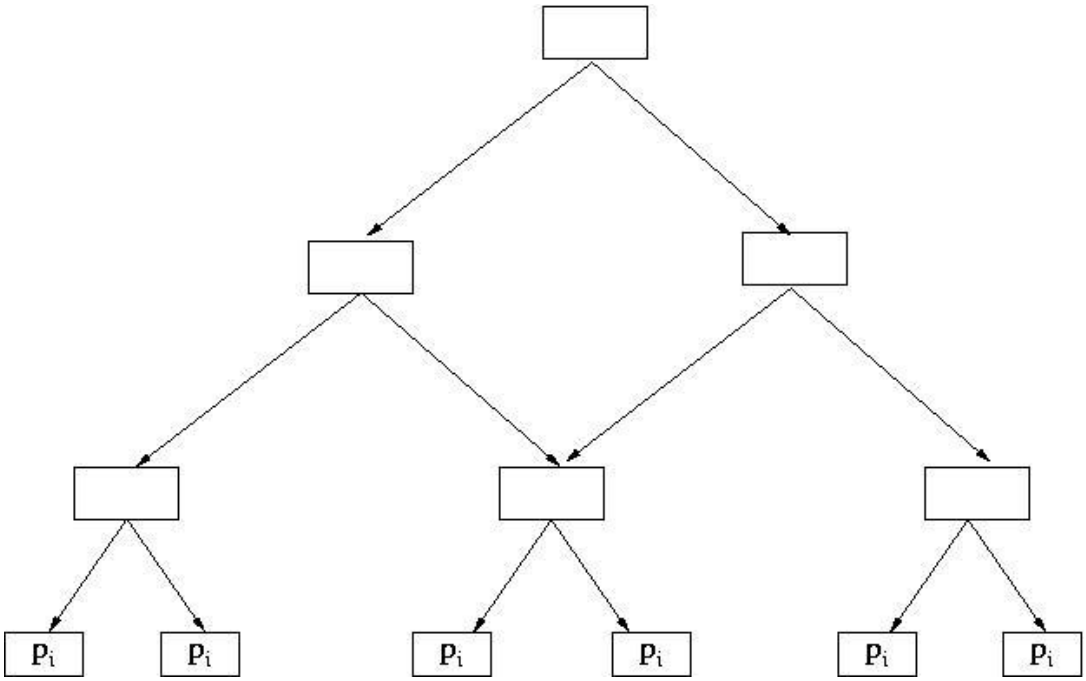


Figure 3-3: Dendogram

An agglomerative clustering starts with one-point (singleton) clusters and recursively merges two or more most appropriate clusters. A divisive clustering starts with one cluster of all data points and recursively splits the most appropriate cluster. The process continues until a stopping criterion (frequently, the requested number k of clusters) is achieved.

To merge or split subsets of points rather than individual points, the distance between individual points has to be generalized to the distance between subsets. Such derived proximity measure is called a linkage metric. The type of the linkage metric used significantly affects hierarchical algorithms, since it reflects the particular concept of closeness and connectivity. Major inter-cluster linkage metrics include ***single link***, ***average link***, and ***complete link***. The underlying dissimilarity measure (usually, distance) is computed for every pair of points with one point in the first set and another point in the second set. A specific operation such as minimum (single link), average (average link), or maximum (complete link) is applied to pair-wise dissimilarity measures:

$$d\left(c_1, c_2\right) = operation\left\{d\left(x, y\right) \mid x \in c_1, y \in c_2\right\}$$

An agglomerative method proceeds as follows:

1. **Initialization:** To each of the *N* data points p_i, $\{i=1,..,N\}$ associate one cluster c_i. You, thus, start with *N* clusters.

2. Find the closest clusters according to a *distancemetric* $d(c_i, c_j)$. The distance between groups can either be:

   - **Single Linkage Clustering**: The distance between the closest pair of data points $d(c_i, c_j)$ = Min { $d(p_i, p_j)$: Where data point $p_i$is in cluster c_i and data point p_j,is cluster c_j}

   - **Complete Linkage Clustering**: The distance between the farthest pair of data points

d(**c_i**,** c_j**) = Max { $d(p_i, p_j)$: Where data point $p_i$is in cluster c_i and data point p_j,is cluster c_j}

- **Average Linkage Clustering**: The average distance between all pairs of data points

d(**c_i**,** c_j**) = Mean { $d(p_i, p_j)$: Where data point $p_i$is in cluster c_i and data point p_j,is cluster c_j}

3. Merge the two clusters into one single cluster taking, e.g., either the mean or the median across the two clusters.

4. Repeat step 2 and 3 until some criterion is achieved, such as, for instance, when the minimal number of clusters or the maximal distance between two clusters has been reached.

The *divisive* hierarchical clustering methods work the other way around. They start by considering all data points as belonging to one cluster and divide iteratively the clusters, according to the points furthest apart.

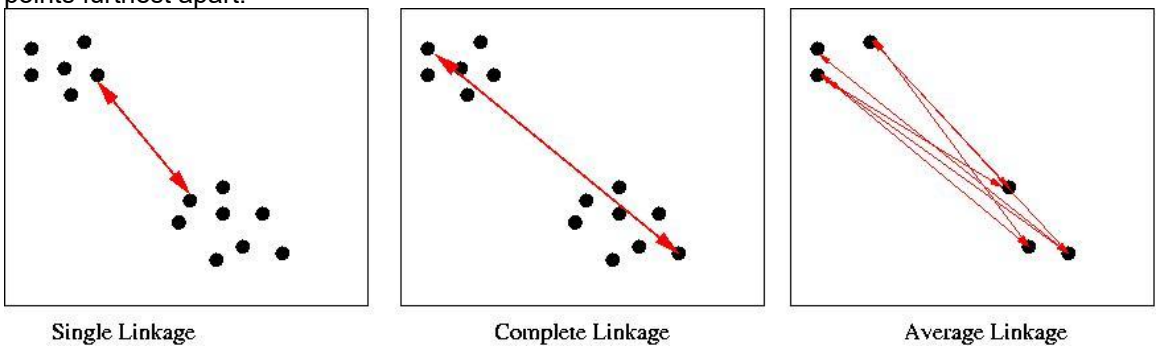


Figure 3-4: Example of distance measurement in hierarchical clustering methods

It is clear that the number and type of clusters will strongly depend on the choice of the distance metric and on the method used to merge the clusters. A typical measure of distance between two N-dimensional data points x, y takes the general form:

$$d^p(x,y) = \sqrt[p]{\sum_{i=1}^{N} |x_i - y_i|^p} \quad (3.1)$$

The 1-norm distance, i.e. *p=1,* sometimes referred to as the *Manhattan distance,* because it is the distance a car would drive in a city laid out in square blocks (if there are no one-way streets). The 2-norm distance is the classical *Euclidean distance.*

Figure 3-5 shows examples of data sets in which such *nearest neighbor* technique would fail. Failure to converge to a correct solution might occur, for instance, when the data points within a dataset are further apart than the two clusters. An even worse situation occurs when the clusters contain one another, as shown in Figure 3-5 right. In other words such a simple clustering technique works well only when *the clusters are linearly separable*. A solution to such a situation is to change coordinate system, e.g. using polar coordinates. However, determining the appropriate coordinate system remains a challenge in itself.

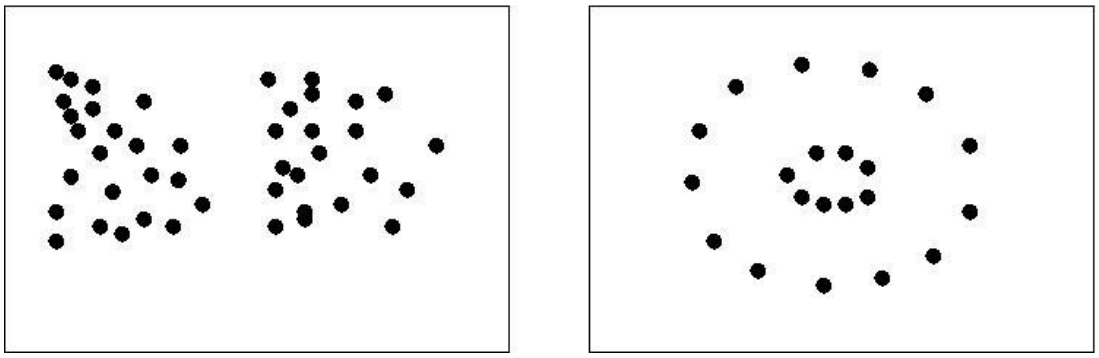


Figure 3-5: Example of pairs of clusters, easy to see but awkward to extract for standard clustering algorithms

3.2.2 K-means clustering

K-Means clustering generates a number *K* of disjoint, flat (non-hierarchical) clusters C^k, $k = 1,...K$, so as to minimize the sum-of-squares criterion

$$J\left(\mu_1,...\mu_K\right) = \sum_{k=1}^{K}\sum_{i\in C^k} d^p(x,y) \qquad \text{and} \qquad d^p(x,y) = \sqrt[p]{\sum_{i=1}^{N}\left|x_i - y_i\right|^p} \tag{3.2}$$

where x_i is a vector representing the *i*th data point, usually p=2 (norm-2) and μ_k is the *geometric centroid* of the data points associated tocluster C^k.

The objective function minimizes the distance of all datapoints to the centroid of each cluster. In other words, it asks that the points to be well grouped around the centroids. By using norm-2, it builds a ball around each centroid and groups points within this ball.

The K-Means method is numerical, unsupervised, non-deterministic and iterative. We summarize the steps of K-means next.

K-Means Algorithm:

1 **Initialization:** Pick *K* arbitrary centroids and set their geometric means $\mu_1,...,\mu_K$ to random values.

2 Calculate the distance $d(x_i,\mu_k)$ from each data point i to each centroid k.

3 **Assignment Step:** Assign the responsibility r_k^i of each data point i to its "closest" centroid k_i (**E-Step).** If a tie happens (i.e. two centroids are equidistant to a data point, one assigns the data point to the smallest winning centroid).

$$k_i = \arg\min_k \left\{ d\left(x_i,\mu_k\right)\right\} \tag{3.3}$$

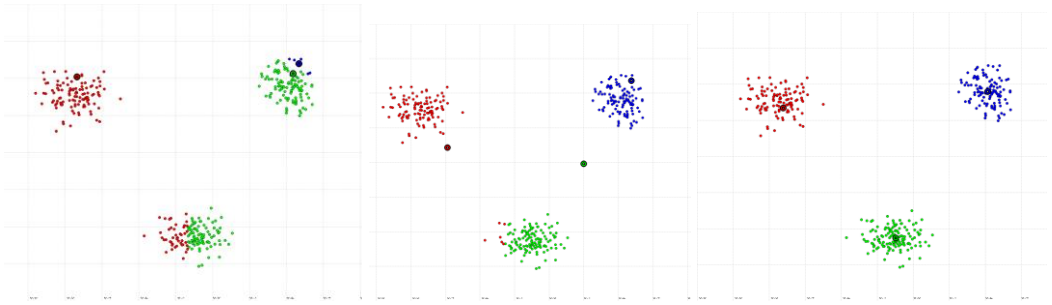
$$r_i^k = \begin{cases} 1 & \text{if } \mathrm{k}^{\mathrm{i}} = k \\ 0 & \text{otherwise} \end{cases}$$

4 **Update Step:** Adjust the centroids to be the means of all data points assigned to them (**M-Step**)

$$\mu_k = \frac{\sum_i r_i^k x_i}{\sum_i r_i^k} \tag{3.4}$$

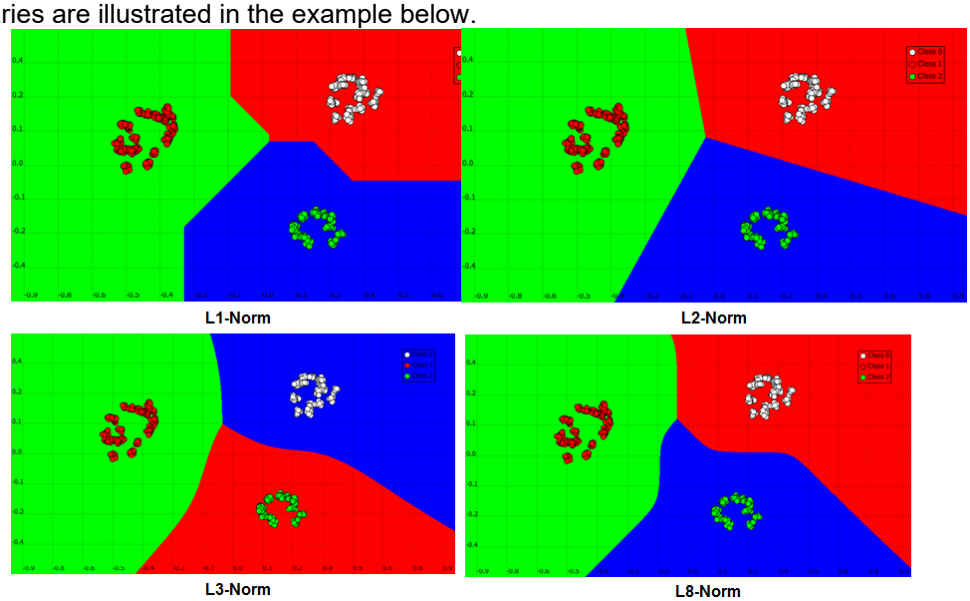
5 Go back to step 2 and repeat the process until a complete pass through all the data points results in no data point moving from one cluster to another. At this point the clusters are stable and the clustering process ends.

We illustrate below an example of K-means clustering for 3 clusters. At initialization the clusters are not well split, as two of the centroids are initialized in the same cluster (top-right). After one iteration, we see how the centroid move toward each of their clusters. The algorithms converge in two iterations in the correct solution.



Initialization iteration 1 iteration 2 - convergence

K-means generates *globular clusters* with isotropic distances in all dimensions. The boundaries between the clusters are linear if we use norm-2 as distance metric (classical K-means version). One can however use other metrics, such as norm-1, norm-p or infinite norm. The effect on the boundaries are illustrated in the example below.



Using other norms than the Euclidean norms allows K-means to produce non-linear boundaries and hence to move away from the globular clustering it is known for. Yet, these non-linear

boundaries remain quite limited in their power of structure and the equidistance across clusters remains a strong limitation which forces clusters to be balanced in both their spread and their relative distance to one another.

K-means clustering is a particular case of *Gaussian Mixture Model* estimation with **EM (expectation maximization)** where the covariance matrices are fixed (these are actually diagonal and isotropic), see Section 3.2.4.

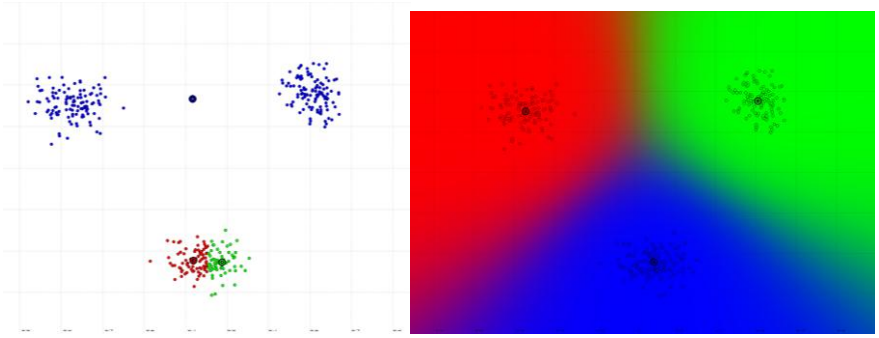
The properties of K-means algorithm can be summarized as follows: There are always K clusters. There is always at least one item in each cluster. The clusters are non-hierarchical and they do not overlap. Every member of a cluster is closer to its cluster than any other cluster because closeness does not always involve the 'center' of clusters.

The algorithm is guaranteed to converge in a finite number of iterations.
But it converges to a local optimum

Advantages of K-means:
With a large number of variables, K-Means may be computationally faster than other clustering techniques, such as hierarchical clustering or non-parametric clustering. Its computational costs grow linearly with both the number of datapoints and of the number of clusters, i.e. *O(MK).* K-Means may produce tighter clusters, especially if the clusters are globular. K-mean is guaranteed to converge

Drawbacks of K-means:
K-mean clustering suffers from several drawbacks. The choice of initial partition can greatly affect the final clusters that result, in terms of inter-cluster and intra-cluster distances and cohesion. Hence, one might find it difficult to compare the quality of the clusters produced (e.g. for different initial partitions or values of K affect outcome).

Below we show the solution found by K-means for the same 3 cluster clustering problem shown previously but with another initialization. There, the algorithm stopped even thought it had split one cluster in half and grouped two very distant clusters. This is an illustration of the sensitivity of the clustering technique to initialization and of the drawback of the hard assignment of the datapoints to one one cluster. A softer assignment as in soft-K-means would not get stuck in this local minimum as it allows all points to influence the update step, as we will see next and as illustrated below.



Left: Solution with K-means; Right: Solution with soft K-means

Moreover, K-means assumes a *fixed* number *K* of clusters, which is difficult to estimate off-hands. It does not work well with *non-globular* clusters. Different initial partitions can result in different final clusters see below:

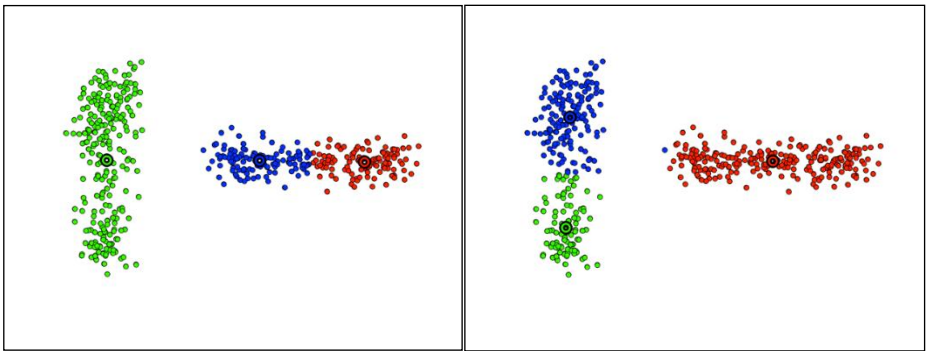


Figure 3-6: The random initialization of K-Means can lead to different clustering results, (in this case using 3 clusters) [DEMOS\CLUSTERING\KMEANS-RANDOM_INITIALIZATIONS.ML]

It is, therefore, good practice to run the algorithm several times using different K values, to determine the optimal number of clusters.

Cases where K-means might be viewed as failing:

Unbalanced clusters:
The K-means algorithm takes into account only the distance between the means and data points; it has no representation of the weight and breadth of each cluster. Consequently, data points belonging to unbalanced clusters (clusters with unbalanced number of points, spread with a smaller breadth) will be incorrectly assigned.

Elongated clusters:
The K-means algorithm has no way to represent the shape of a cluster. Hence, a simple measure of distance would be unable to separate two elongated clusters as shown in Figure 3-7.

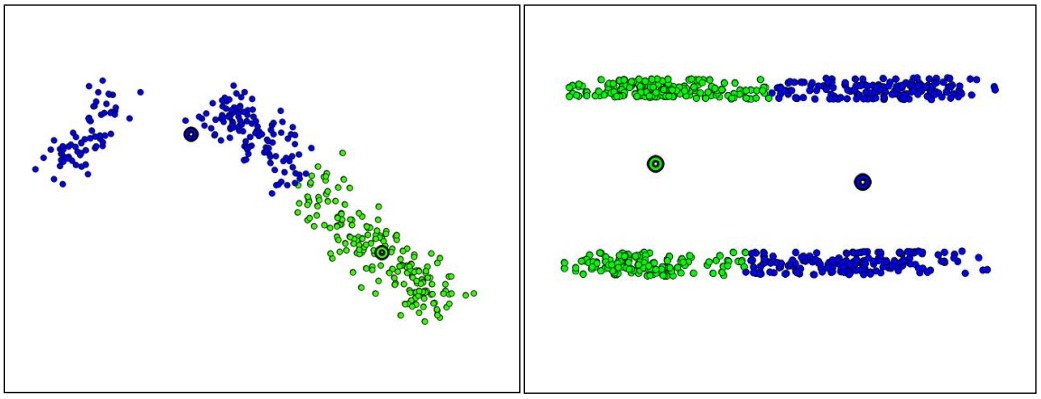


Figure 3-7: Typical examples for which K-means is ill-suited: unbalanced clusters (left) and elongated clusters (right). In both cases the algorithm fails to separate the two clusters. [DEMOS\CLUSTERING\KMEANS-UNBALANCED.ML] [DEMOS\CLUSTERING\KMEANS-ELONGATED.ML]

3.2.3 Soft K-means

`Soft K-means algorithm' was offered as an alternative to K-means in order to “soften” the assignment of variables. Each data point $x_i$ is given a soft `degree of assignment' to each of the

means. The degree to which x_i is assigned to cluster k is the responsibility r_k^i of cluster k for point i and is a real value comprised between 0 and 1.

$$r_i^k = \frac{e^{\left(-\beta \cdot d\left(\mu_k, x_i\right)\right)}}{\sum_{k'} e^{\left(-\beta \cdot d\left(\mu_{k'}, x_i\right)\right)}} \qquad (3.5)$$

β is the stiffness and determines the binding between clusters. The larger β, the bigger the distance across two clusters. A measure of the disparity across clusters is given by $\sigma \equiv \frac{1}{\sqrt{\beta}}$, the radius of the circle that surrounds the means of the *K* clusters.

The sum of the K responsibilities for the i[th] point is 1, i.e. $\sum_k r_i^k = 1$

Update step. The model parameters, i.e. the means, are adjusted to match the sample means of the data points that they are responsible for.

$$\mu^k = \frac{\sum_i r_i^k \cdot x_i}{\sum_i r_i^k}$$

The update algorithm of the soft K-means is identical to that of the hard K-means, apart for the fact that the responsibilities to a particular cluster are now real numbers varying between 0 and 1.

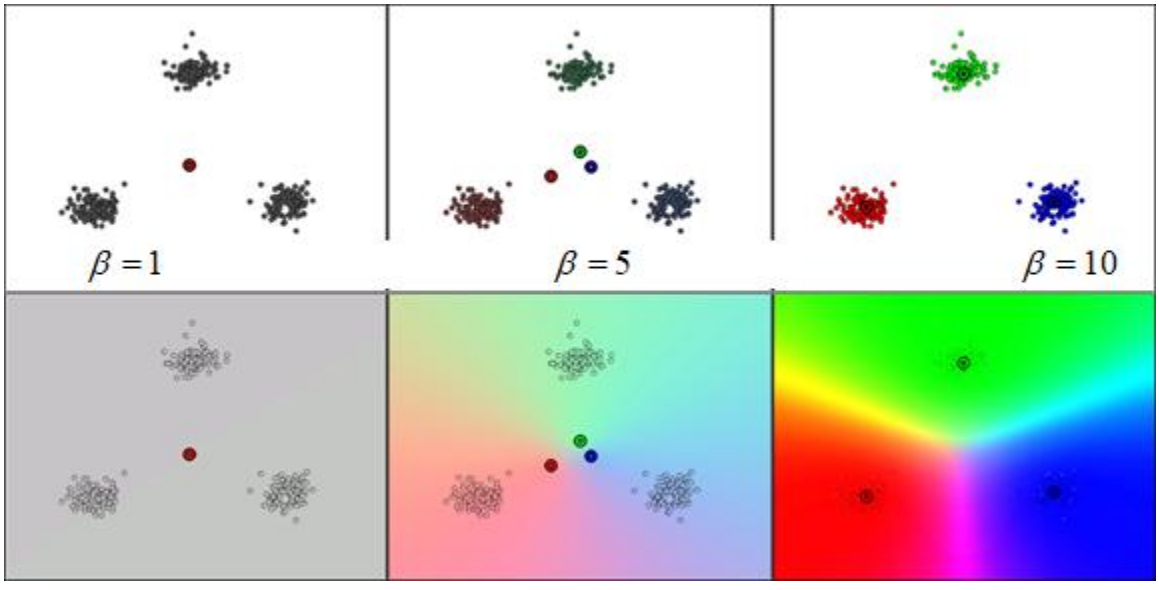


Figure 3-8: Soft K-means algorithm with a small (left), medium (center) and large(right) β [DEMOS\CLUSTERING\SOFT-KMEANS-SIGMA.ML]

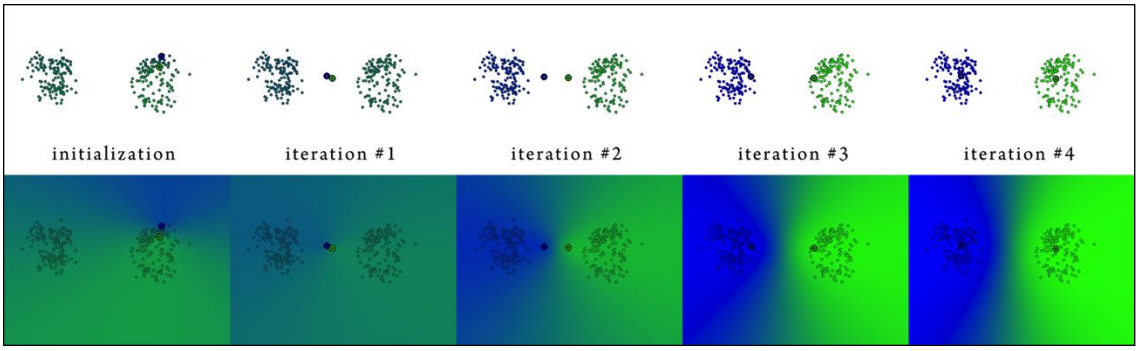


Figure 3-9: Iterations of the Soft K-means algorithm from the random initialization (left) to convergence (right). Computed with β= 10. [DEMOS\CLUSTERING\SOFT-KMEANS-ITERATIONS.ML]

3.2.4 Density Based Spatial Clustering of Applications with Noise (DBSCAN)

DBSCAN is a simple clustering algorithm which offers an interesting alternative to K-means. It provided non-globular clusters (concave). Most importantly, it can detect outliers and remove these from clustering.
Like K-means, it proceeds iteratively. It has two hyperparameters (parameters fixed by the user). These are the minimal size of a cluster ε and the minimum number of datapoints in a cluster m_{data}. These two parameters determine minimal bounds on noise/outlier determination. They must be chosen with care and as a function of the dataset.

The algorithms is summarized below:

1. Pick a first datapoint at random
2. Compute number of datapoints within ε around the datapoint
3. If this number is < m_{data}, set this datapoint as an outlier
   otherwise, for each point found within ε around the first datapoint, assign it to the same cluster as the first datapoint.
4. If there is a cluster within ε, merge with this cluster.

Below, we illustrate the main steps of the algorithm in a schematic:

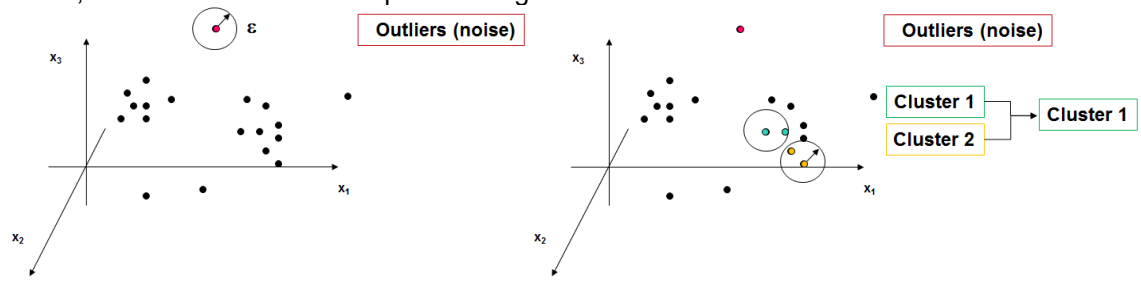


Figure 3-10: Illustration of the step of DBSCAN. Left: a point is identified as an outlier as it has no datapoint within an ε neighbourhood. Right: two newly created cluster of datapoints are merged as the two clusters are within ε from one another.

DBSCAN can cluster correctly elongated clusters (which K-means cannot cluster correctly), see example below:

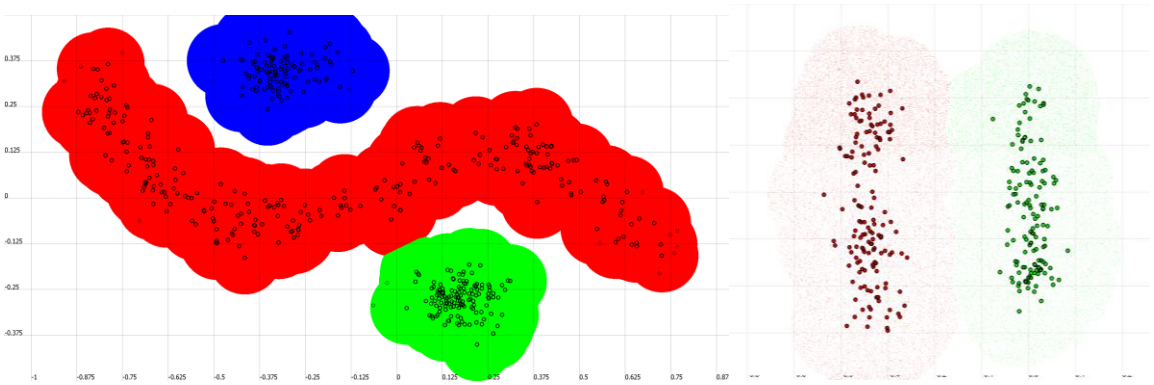


Figure 3-11: Clustering obtained with DBSCAN illustrating the ability of the algorithm to generate non-globular clusters (left) and that it can separate the double elongated clusters which K-means cannot (right).

DBSCAN is computationally intensive, as one must swipe the data several times to test the neighbourhood and then to merge the clusters. Its computational costs grow quadratically with *M.* There exists methods to speed up computation, but at best it grows with *O(M log(M)).*

3.2.5 Clustering with Mixtures of Gaussians

An extension of the soft K-means algorithm consists of fitting the data with a *Mixture of Gaussians* (not to be confused with *Gaussian Mixture Model (GMM)*)) which we will review later on. Instead of simply attaching a responsibility factor to each cluster, one attaches a density of probability measuring how well each cluster represents the distribution of the data. The method is bound to converge to a state that maximizes the likelihood of each point to belong to each distribution.

Soft-Clustering methods are part of *model-based approaches* to clustering. In clustering with mixture of Gaussians, the model is naturally a Gaussian. Other model-based methods use, for instance, the Poisson or the Normal distributions.

The main advantages of model-based clustering are:

- It can make use of well-studied statistical inference techniques;
- Its flexibility in choosing the component distribution;
- It obtains a density estimation for each cluster;
- It is a “soft” means of classification.

Clusters with mixtures of Gaussian places *K* distributions, whose barycentres are located on the cluster means, as in Figure 3-13.

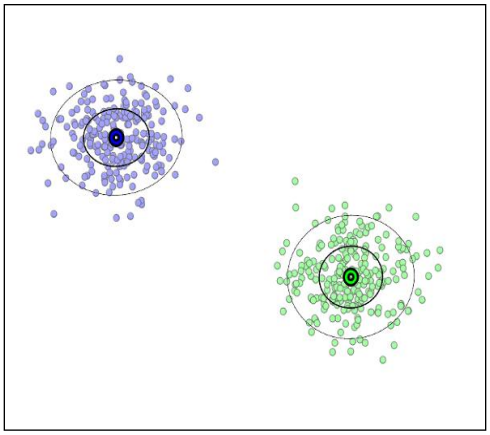


Figure 3-12: Examples of clustering with Mixtures of Gaussians (the grey circles represent the first and second variances of the distributions).[DEMOS\CLUSTERING\GMM-CLUSTERING-SIMPLE.ML]

Algorithm

Assignment Step (E-step): The responsibilities are

$$r_i^k = \frac{\alpha_k \frac{1}{\left(\sqrt{2\pi\sigma_k}\right)^N} e^{\left(-\frac{1}{\sigma_k^2}\cdot d(\mu_k, x_i)\right)}}{\sum_{k'} \alpha_{k'} \frac{1}{\left(\sqrt{2\pi\sigma_{k'}}\right)^N} e^{\left(-\frac{1}{\sigma_{k'}^2}\cdot d(\mu_{k'}, x_i)\right)}} \tag{3.6}$$

where *M* is the number ofdatapoints, i.e. $x=\{x_1, x_2,..., x_M\}$, $d(x, y)$ is a distance measure, e.g. the Euclidean distance.

Update Step (M-step): Each cluster's parameters μ_k, α_k and σ_k are adjusted to match the data points for which the cluster is responsible:

$$\mu_k = \frac{\sum_i r_i^k x_i}{\sum_i r_i^k} \tag{3.7}$$

$$\sigma_k^2 = \frac{\sum_i r_i^k (x_i - \mu_k)^2}{N \cdot \sum_i r_i^k} \tag{3.8}$$

$$\alpha_k = \frac{\sum_i r_i^k}{\sum_k \sum_i r_i^k} \tag{3.9}$$

The α_k represents a measure of the likelihood that the Gaussian *k* (or cluster *k*)generated the whole dataset.

This fits a mixture of *spherical* Gaussians. “Spherical” means that the variance of the Gaussians is the same in all directions. In other words, in the case of multidimensional dataset, the covariance matrix of the Gaussian is diagonal and isotropic (i.e. all the elements on the diagonal are equal and all elements on the off-diagonal are zero). This algorithm is still not good at modeling datasets spread along two elongated clusters, as shown in Figure 3-7. If we wish to model the clusters by axis-aligned Gaussians, we replace the assignment rule given by Equations (3.6) and (3.8) with the following:

$$r_i^k = \frac{\alpha_k \dfrac{1}{\prod_{i=1}^{N}\left(\sqrt{2\pi}\sigma_i^k\right)} e^{\left(-\sum_{i=1}^{N}\frac{d(\mu_k, x_i)^2}{2\left(\sigma_i^k\right)^2}\right)}}{\sum_{k'} \alpha_k \dfrac{1}{\prod_{i=1}^{N}\left(\sqrt{2\pi}\sigma_i^{k'}\right)} e^{\left(-\sum_{i=1}^{N}\frac{d(\mu_{k'}, x_i)^2}{2\left(\sigma_i^k\right)^2}\right)}} \tag{3.10}$$

$$\mu_j^k = \frac{\sum_i r_i^k x_{i,j}}{\sum_i r_i^k} \tag{3.11}$$

$$\left(\sigma_j^k\right)^2 = \frac{\sum_i r_i^k \left(x_{i,j} - \mu_j^k\right)^2}{N \cdot \sum_i r_i^k} \tag{3.12}$$

$$\alpha_k = \frac{\sum_i r_i^k}{\sum_k \sum_i r_i^k} \tag{3.13}$$

Soft K-means and the mixture of Gaussians are two examples of *maximum likelihood* algorithm.

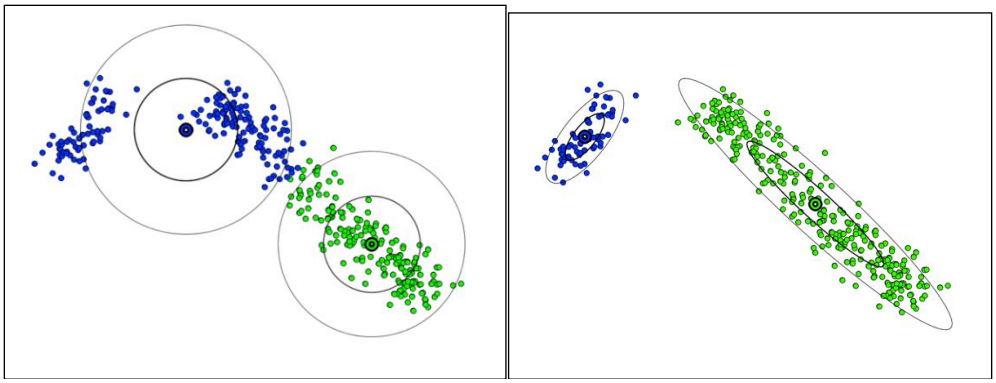


Figure 3-13: Examples of clustering with Mixtures of Gaussians using spherical Gaussians (left) and non-spherical Gaussians (i.e. with full covariance matrix) (right). Notice how the clusters become elongated along the direction of the clusters (the grey circles represent the first and second variances of the distributions).

[DEMOS\CLUSTERING\GMM-ELONGATED.ML]

A fatal flaw of maximum likelihood (KABOOM!):
The major drawback of maximizing the likelihood (people tend to actually minimize the –log of the likelihood) is that there is no actual maximum. Indeed, the pdf is a positive but unbounded function. As a result, one may observe the so-called KABOOM effect.

When the mean of one of the Gaussian is located on a particular data point, and, if the variance is already very small $\sigma^2 << 1$, then σ^2 will become even smaller. This is a fatal flaw of soft k-means algorithms: very small clusters can be formed that satisfy only a few data points. If the variance goes to zero, you will obtain an arbitrary large likelihood (kaboom). This is a general flaw of maximum likelihood methods that can find highly tuned models that fit only part of the data, but perfectly. This phenomenon is known as over fitting.

3.2.6 Gaussian Mixture Models

While Gaussian Mixture Models (GMM) are often presented alongside other parametric or non-parametric density estimation methods (e.g. such as Gaussian Process which we here present in the regression method section), we felt that in the framework of these lecture notes it was preferable to introduceGMM here as its notation follows naturally from that introduced in the soft-K-means methods.

The estimation of Mixture of Gaussians we reviewed in the previous section assumed a diagonal covariance matrix for each Gaussian. This is rather restrictive as it forces the Gaussians to have their axes aligned with the original axes of the data, seeFigure 3-14. As a result, they are ill-suited for estimating highly non-linear densities that have local correlations.

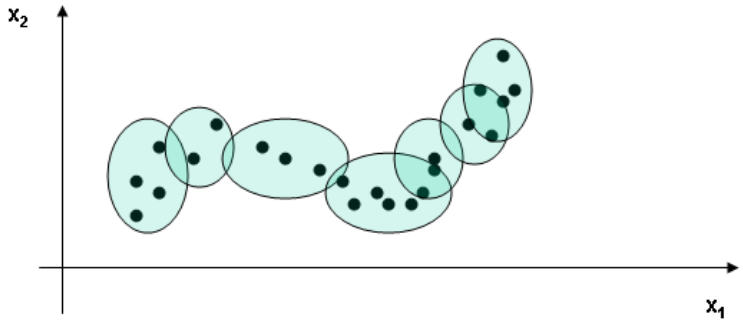


Figure 3-14: A spherical Mixture of Gaussian Model can only fit Gaussians whose axes are aligned with the data axes. It may require far more local models to account for local correlations across the dimensions of the data *X*.

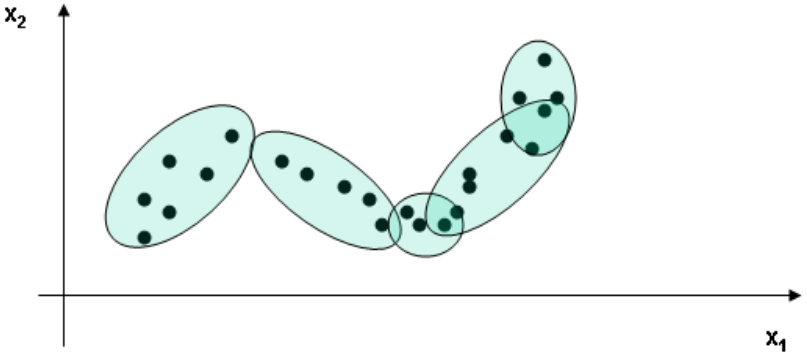


Figure 3-15: In contrast, a Gaussian Mixture Model can exploit local correlations and adapt the covariance matrix of each Gaussian so that it aligns with the local direction of correlation.

Gaussian Mixture Model is a generic method that fits a mixture of Gaussians with full covariance matrices. Each Gaussian serves as a local linear model and in effect performs a local PCA on the data. By mixing the effect of all these local models, one can obtain a very complex density estimate, see as an illustration the schematics in Figure 3-14 and Figure 3-15.

Let suppose that we have at our disposal $X = \left\{ x_i^j \right\}_{i=1,...N}^{j=1,...M}$ the set of*MN*-dimensional data points whose density we wish to estimate. We consider that these are observed instances of the variable x. The probability distribution function or density of x is further assumed to be composed of a mixture of $\mathrm{k} = 1,..,K$ Gaussians with mean μ^k and covariance matrix $\sum^k$, i.e.

$$p(x) = \sum_{k=1}^{K} \alpha_k \cdot p_k\left(x \mid \mu^k, \Sigma^k\right), \qquad \text{with } p_k\left(x \mid \mu^k, \Sigma^k\right) = N\left(\mu^k, \Sigma^k\right) \tag{3.14}$$

The α_k are the so-called mixing coefficients. Their sum is 1, i.e. $\sum_{k=1}^{K} \alpha_k = 1$. These coefficients are usually estimated together with the estimation of the parameters of the Gaussians (i.e. the means and covariance matrices). In some cases, they can however be set to a constant,for instance to give equal weight to each Gaussian (in this case, $\alpha_k = 1/K \quad \forall k = 1,...K$). When the coefficients

are estimated, they end up representing a measure of the proportion of data points that belong most to that particular Gaussian (this is similar to the definition of the α seen in the case of Mixture of Gaussians in the previous section). In a probabilistic sense, these coefficientsrepresent the *prior* probability with which each Gaussian may have generated the whole dataset and can hence be written $\alpha_k = p(k) = \sum_{j=1}^{M} p(k \mid x^j)$.

Learning of GMM requiresdetermining the means and covariance matrices and prior probabilities of the K Gaussians. The most popular method relies on Expectation-Maximization (E-M). We advise the reader to take a detour here and read the tutorial by Gilmes provided in the annexes of these lectures notes for a full derivation of GMM parameters estimation through E-M.We briefly summarize the principle next.

We want to maximize the likelihood of the model's parameters $\Theta = \{\alpha^1, ...\alpha^K, \mu^1, ...\mu^K, \Sigma^1, ... \Sigma^K, \}$ given the data, that is:

$$\max_{\Theta} \; L(\Theta \mid X) = \max_{\Theta} \; p(X \mid \Theta) \tag{3.15}$$

Assuming that the set of *M* datapoints $X = \{x^j\}_{j=1}^{M}$ is identically and independently distributed (iid), we get:

$$\max_{\Theta} \; p(X \mid \Theta) = \max_{\Theta} \; \prod_{j=1}^{M} \sum_{k=1}^{K} \alpha_k \cdot p(x^j \mid \mu^k, \Sigma^k) \tag{3.16}$$

Taking the log of the likelihood is often a good approach as it simplifies the computation. Using the fact that the optimum x^* of a function $f(x)$ is also an optimum of $\log f(x)$, one can compute:

$$\begin{aligned} &\max_{\Theta} \; p(X \mid \Theta) = \max_{\Theta} \; \log p(X \mid \Theta) \\ &\max_{\Theta} \; \log \prod_{j=1}^{M} \sum_{k=1}^{K} \alpha_k \cdot p(x^j \mid \mu^k, \Sigma^k) = \max_{\Theta} \; \sum_{j=1}^{M} \log \left(\sum_{k=1}^{K} \alpha_k \cdot p(x^j \mid \mu^k, \Sigma^k) \right) \end{aligned} \tag{3.17}$$

The log of a sum is difficult to compute and one is led to proceed iteratively by calculating an approximation at each step (EM), see Sections 9.4.2.The final update procedure runs as follows:

Initialization of the parameters:

Initialize all parameters to a value to start with. The $p_1, .., p_K$ can for instance be initialized with a uniform prior, while the means can be initialized by running K-Means first. The complete set of parameters is then given by:

$$\Theta_0 = \{\alpha_1^{(0)}, ..., \alpha_K^{(0)}, \mu_1^{(0)}, ..., \mu_K^{(0)}, \Sigma_1^{(0)}, ..., \Sigma_K^{(0)}\}$$

E-step:

At each step *t, estimate*, for each Gaussian*k*, the probability that this Gaussian is being responsible for generating each point of the dataset by computing:

$$\alpha_k^{(t)} = p\left(k \mid \Theta^{(t)}\right) = \frac{p_k\left(x^i \mid \mu^{k(t)}, \Sigma^{k(t)}\right) \cdot \alpha_k^{(t)}}{\sum_j p_k\left(x^i \mid j, \mu^{k(t)}\right) \cdot \alpha_j^{(t)}} \tag{3.18}$$

M-step:

Recompute the means, Covariances and prior probabilities so as to maximize the log-likelihood of the current estimate: $\log\left(L\left(\Theta^{(t)} \mid X\right)\right)$ and using current estimate of the probabilities $p\left(k \mid \Theta^{(t)}\right)$:

$$\mu^{k(t+1)} = \frac{\sum_j p\left(k \mid x^j, \Theta^{(t)}\right) \cdot x_j}{\sum_j p\left(k \mid x^j, \Theta^{(t)}\right)} \tag{3.19}$$

$$\Sigma^{k(t+1)} = \frac{\sum_j p\left(k \mid x^j, \Theta^{(t)}\right)\left(x^j - \mu^{k(t+1)}\right)\left(x^j - \mu^{k(t+1)}\right)^T}{\sum_i p\left(k \mid x^i, \Theta^{(t)}\right)} \tag{3.20}$$

$$\alpha_k^{(t+1)} = \frac{\sum_j P\left(k \mid x^j, \Theta^{(t)}\right)}{M} \tag{3.21}$$

Note finally that we have not discussed how to choose the optimal number of states *K*. There are various techniques based on determining a tradeoff between increasing the number of states and hence the number of parameters (increasing greatly the computation required to estimate such a large set of parameters) and the improvement that such an increase brings to the computation of the likelihood. Such a tradeoff can be measured by either of the BIC, DIC or AIC criteria, see the slides of the class.

Figure 3-16 shows an example of clustering using a 3 GMM model with full covariance matrix and illustrates how a poor initialization can result in poor clustering.

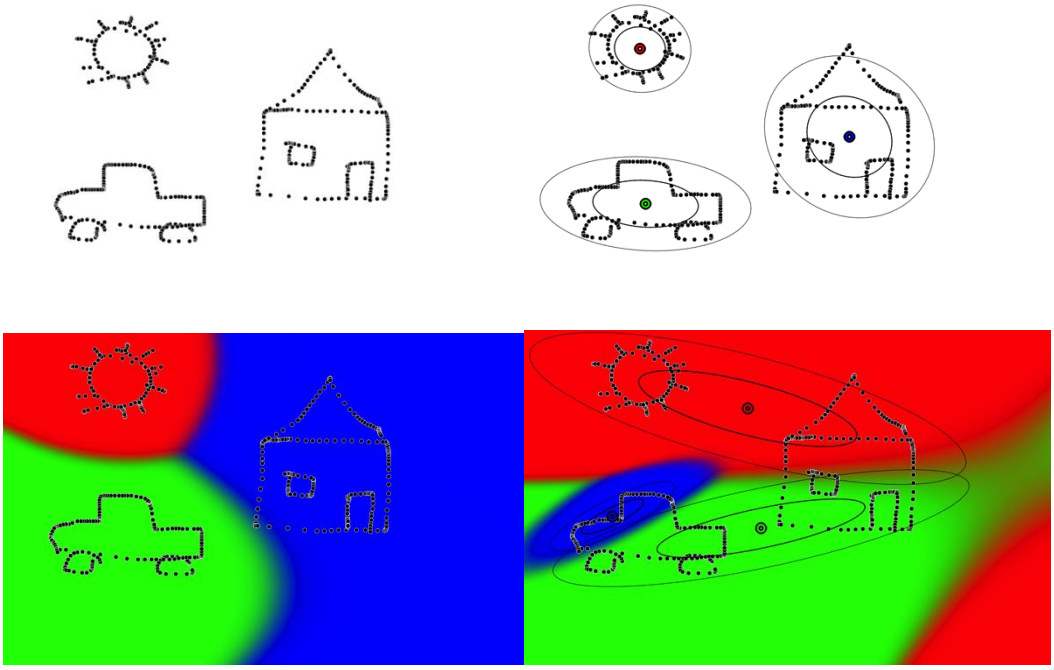


Figure 3-16: Clustering with 3 Gaussians using full covariance matrix. Original data (top left), superposed Gaussians (top right), regions associated to each Gaussian using Bayes rule for determining the separation (bottom right); effect of a poor initialization on the clustering (bottom right).

3.2.7 Metrics for evaluating clustering techniques

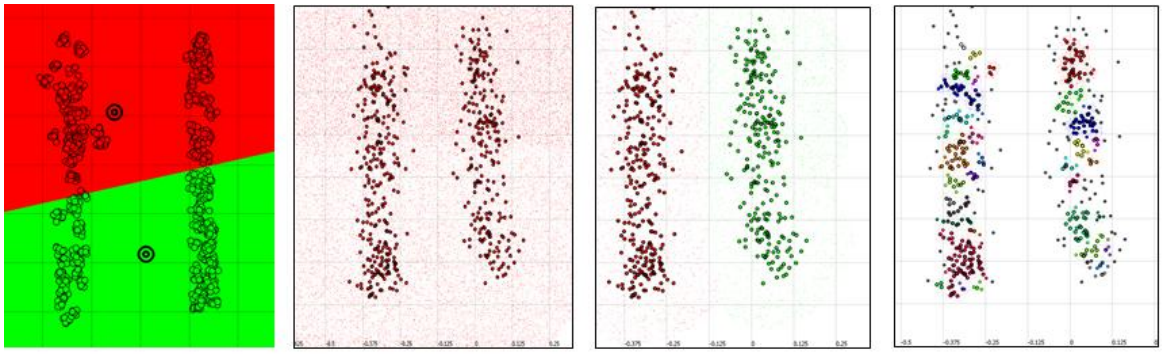
One major issue with clustering is the fact that we do not have access to the ground truth, as we do not have the true class labels. Moreover, we do not even know how many classes (clusters) there are. One must hence rely on other criteria to evaluate how good the clustering is. We here present three metrics to help you assess how good your clustering is and help you to derermine the optimal number of clusters. These metrics are usually referred to as *internal measures,* by opposition to *external measures,* which we will see with semi-supervised clustering.

3.2.7.1 The RSS measure

The residual sum of square (RSS) measure computes the distance (in norm-2) of each datapoint from its centroid for all clusters.

$$\text{RSS}=\sum_{k=1}^{K}\sum_{x\in C_k}\left|x-\mu^k\right|^2$$

It hence measures how close the datapoints are to the center of the cluster. The closer, the smaller the measure. This measure is identical to the objective function that is minimized in K-means. K-means solution is hence optimal according to RSS. However, since K-means is quite sensitive to initialization, different initialization may lead to different clustering and the goodness of one solution over another solution can be assessed with RSS. DBSCAN tends also to group datapoints close to one another and would hence fare well on the RSS measure.



K-means DBSCAN large ε DBSCAN medium ε DBSCAN small ε

	K-means	DBSCAN large ε	DBSCAN medium ε	DBSCAN small ε
RSS	51	95	59	0.6
BIC	65	118	88	331
AIC	55	102	67	93

Figure 3-17: (From left to right:) Clustering of two elongated clusters with K-means, DBSCAN with large/medium/small ε, respectively. The RSS, AIC and BIC measures give different results, with optimum for different solutions. If one looks for the model for which all 3 metrics give a low value, then DBSCAN with medium ε is the best solution.

In Figure 2-1, we show an example of clustering on the difficult case of clustering two elongated clusters. We contrast the value obtained on the RSS measure for K-means, and DBSCAN with different values of ε. The RSS measures does not allow one to determine that K-means is doing poorly nor to determine that DBSCAN with a tiny value of ε is not doing a good job. This is normal since the RSS measure is optimal for K-means and it values tight clusters. In fact, the RSS measure is optimal when one places one cluster on any single datapoint! This is illustrated below for K-means and this is what we observed already above for small values of ε in DBSCAN.

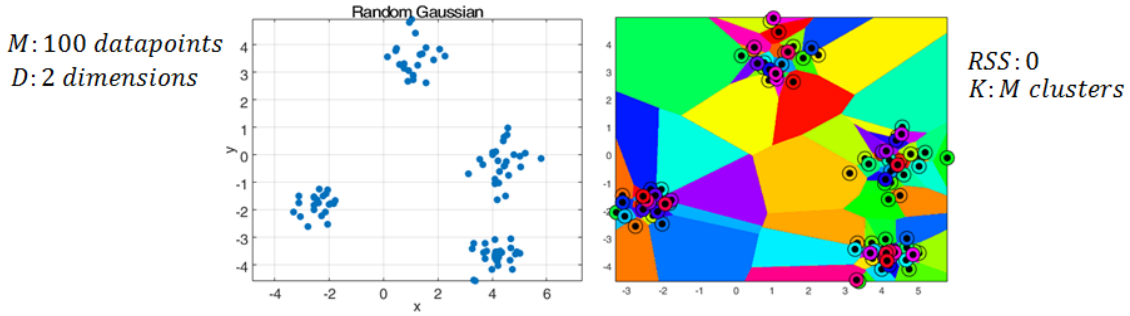
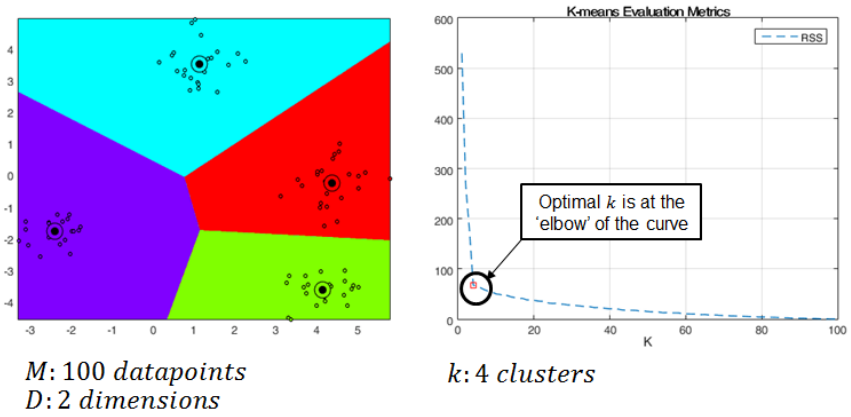


Figure 3-18: Example of application of K-means on a dataset with *as many clusters as datapoints.* This yields an optimum on RSS with RSS=0.

Although the RSS measure is optimal is the degenerated case where one places one cluster on everysingle datapoint, it can still be used to help one to determine the correct number of clusters *K* in K-means. In the previous example, we see that when incrementing the number of cluster and computing the RSS measure, we can determine correctly that the optimal number of clusters is 4, see below.



The same can be done also for determining the correct value of the two hyperpameters of DBSCAN, although the RSS measure may sometimes not show the desired plateau.

3.2.7.2 AIC and BIC criteria

The Aikaike Information Criterion (AIC) and the Bayesian Information Criterion (BIC) determine how good the model fits the dataset in a probabilistic sense. It takes a measure of how likely it is that the model fits well the underlying (unknown) distribution of the data. The measure is balanced by how many parameters are needed to get a good fit.

$$\mathrm{AIC} = -2\ln L + 2B$$

$$BIC = -2\ln L + B\ln(M)$$

where L is the maximum likelihood of the model, B the number of free parameters and M the number of datapoints.

While AIC penalizes only for an increase in the number of parameters of the model, BIC penalizes takes into account the number of datapoints. The larger the number of datapoints, the larger the necessary increment in likelihood needed to counterbalance the penalty. In other words, for large dataset, BIC will tend to penalize harder models with low gains in likelihood. This avoids overfitting and tends to favor sparse but highly representative models. It is particularly suited to fit GMM and would lead to models with few Gaussians but well tunes means and covariance matrices so as to maximize coverage of the data with the model. This is illustrated below:

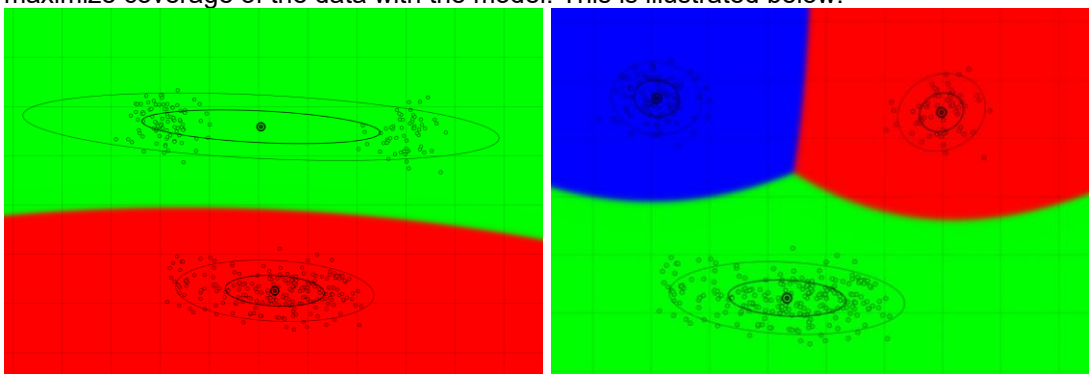


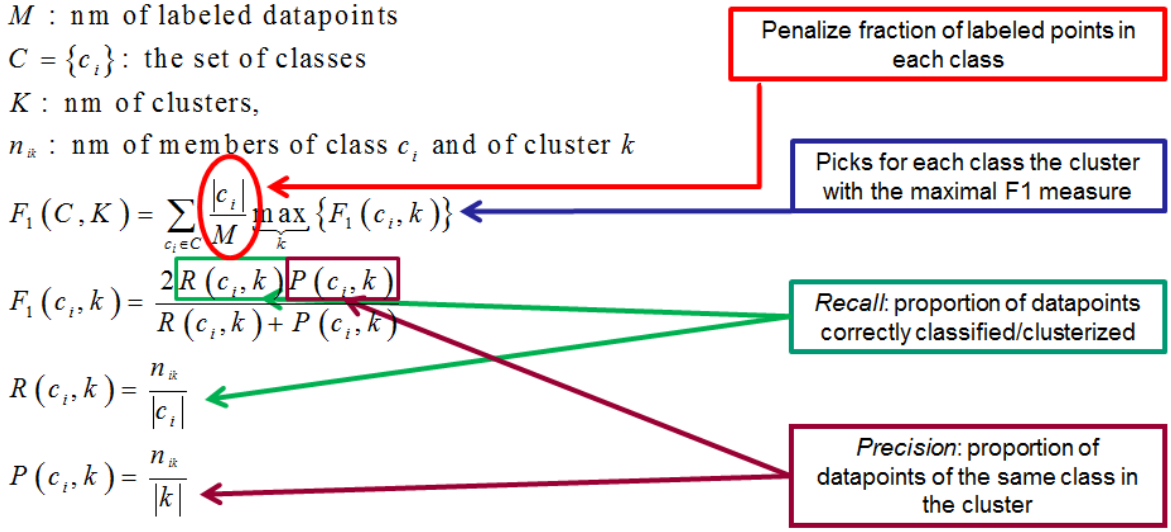
Figure 3-19: Fit with a GMM with 2 (left) and 3 (right) full covariance models. BIC yields a value of 29 and 22 respectively, hence favoring the model with more parameters but better statistical coverage of the data.

As we see in Figure 3-17, AIC and BIC yield both very different values when comparing the solutions found with K-means and DBSCAN on the two elongated clusters. While each metric cannot be used to determine the correct solution, when we compare the solutions we see that the solution that finds a compromise across all metrics is the optimal choice, namely DBSCAN with medium ε.

3.2.7.3 F1-measure for semi-supervised clustering

As we have seen previously, one of the main difficulties in evaluating clustering is the fact that we do not have the ground truth and know neither the true class labels nor the true number of classes. Whenever possible, one may help clustering by providing a subset of the trues class labels. This is known as semi-supervised clustering. In this case a fraction of the data receives a class label. The higher the fraction, the more information at our disposal. In real life, this fraction of labeled data can be very low, ranging from 10% to 1% of all data. The reason for such a low number is that it is very time consuming to label data and it may in some cases simply be impossible to label all possible data. Typically, semi-supervised clustering may be used when one wants to group products sold on on-line shopping websites (amazon, itunes), one will pay a few willing customers to provide their grouping and use their labeling to help clustering.

To evaluate the clustering, we can use the F1-measure, given below:



This metric looks at each class and measures how well each class is clustered. It then balances two factors, the *recall* and the *precision.*

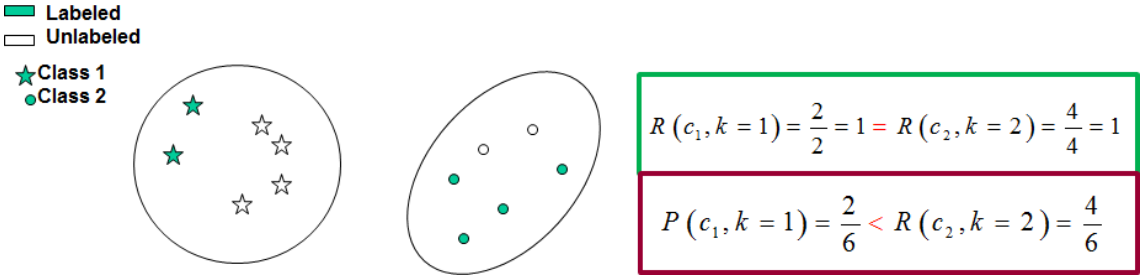
- Recall measures if all datapoints for which we have the class label have been clustered in the same cluster. It is 1 when all points of the same class are grouped together and zero when the cluster contains no labelled datapoints.
- Since the number of points with class labels is low, recall is balanced by precision. Precision measures how many labeled points are contained in the cluster compared to the total number of datapoints. A cluster with a large number of unlabeled points compared to the number of datapoints correctly classified would receive a low value of precision.

Finally, the F1-measure sums the F1-values of each class and balances their contribution according to the fraction of labelled datapoints from this class. Classes with few datapoints will have less impact on the total F1-measure.

The F1-measure is comprised between 0 and 1. It is 1 when all datapoints are perfectly classified and the clusters contain only correctly classified datapoints. In semi-supervised clustering, the F1-

measure can never be 1, since only part of the data are labelled, and henceforth the precision will always be lower than 1.

Below is a schematic example of the result yielded by the F1-measure. The two clusters have perfect recall, as they both grouped correctly all labelled datapoints of class 1 and 2 within the same cluster. Their respective F-measure will however be penalized by having a low value for the precision. Precision measures the fraction of labelled points over all points within the cluster. The first cluster is more penalized than the second one, as only two datapoints over a total of 6 are labelled.



The total F1-measure sums the result of the best F1- measure for each cluster and weights their influence by the fraction of points labelled over all points with labels. In other words, clusters that would have a perfect F1-measure but include only a low number of labelled point would be penalized for this. In the schematic example above, this would yield the following F1-measure:

$$F_1(C,K) = \frac{2}{6}\left(F_1(c_1, k=1)\right) + \frac{4}{6}\left(F_1(c_2, k=2)\right) = 0.7$$

Penalize fraction of labeled points in each class

The cluster one is penalized further by the fact that it represents only 2 out of the 6 labelled datapoints. The total F1-measure is 0.7.

3.3 Classification

3.3.1 Bayes Classifier

The simplest means to perform binary classification in probabilistic models is to use the so-called *Bayes Classifier*. Assume a binary labeling $y^i \in [-1, +1]$ of a set of $i = 1....M$ datapoints x^i. Assume that you have built twoprobabilistic models $p_+(y \mid x)$ and $p_-(y \mid x)$ that predict the probability that the data point x had associated label +1 and -1 respectively.A Bayes classifier will decide on the correct labeling simply by comparing the relative probabilities of each model, i.e.:

$$\begin{aligned} &\text{If } p_+(y \mid x) \geq p_-(y \mid x), \text{ then y=+1.} \\ &\text{Otherwise y=-1.} \end{aligned} \tag{3.22}$$

Take for example a two classification problem where you have modeled each class with a single Gauss distribution. The likelihood of each class is given by:

$$p(x \mid y=+1) \sim N\left(\mu^1, \Sigma^1\right) = \frac{1}{(2\pi)^{N/2}\left|\Sigma^1\right|^{1/2}} e^{-\left(x-\mu^1\right)^T\left(\Sigma^1\right)^{-1}\left(x-\mu^1\right)}$$

$$p(x \mid y=-1) \sim N\left(\mu^2, \Sigma^2\right) = \frac{1}{(2\pi)^{N/2}\left|\Sigma^2\right|^{1/2}} e^{-\left(x-\mu^2\right)^T\left(\Sigma^2\right)^{-1}\left(x-\mu^2\right)}$$

To determine to which of the two classes one point belongs, one applies the Bayes' rule to compute for each class label *i*, $p(y=i \mid x)$,

$$p(y=i \mid x) = \frac{p(x \mid y=i)\, p(y=i)}{p(x)}, \quad i = \pm 1.$$

This numerator of this equation simplifies if one assumes that the likelihood of observing class +1 is equal to the likelihood of observing class label -1. To determine if a point belongs to either class + 1 or class -1 is equivalent to requesting that the ratio between the two likelihood be greated than 1, i.e.:

$$\frac{p(x \mid y=1)}{p(x \mid y=-1)} > 1$$

Given that the distribution follow two Gauss distributions, we can take the log and we obtain an explicitly rule:

$$\frac{p(x \mid y=1)}{p(x \mid y=-1)} > 1 \Rightarrow \ln\left(\frac{p(x \mid y=1)}{p(x \mid y=-1)}\right) > 0$$

$$\Rightarrow \left(x-\mu^1\right)^T\left(\Sigma^1\right)^{-1}\left(x-\mu^1\right) + \log\left|\Sigma^1\right| < \left(x-\mu^2\right)^T\left(\Sigma^2\right)^{-1}\left(x-\mu^2\right) + \log\left|\Sigma^2\right|$$

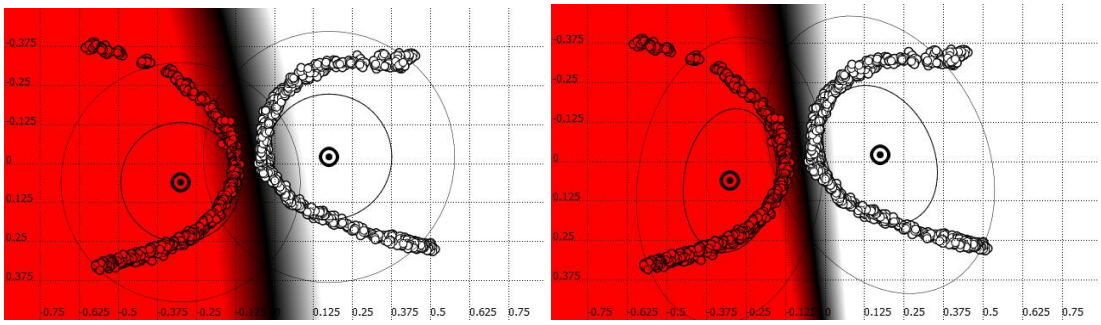


Figure 3-20: Example of binary classification using Bayes rule on two classes modelled by (left) isotropic Gaussian distributions and (right) full covariance Gaussian distributions.

Note that Bayes classification can easily be extended to multiclass classification. Assume that you wish to classify a set of datapoints $X = \left\{x^i\right\}_{i=1}^{M}$ into *K* classes $C^1,....,C^K$. Each label y^i associated to each datapoint x^i, $i = 1....M$, can then take any value $C^1,....,C^K$. One can then compute the posterior probability of each class, using:

$$p\left(y=C^i \mid x\right)=\frac{p\left(y=C^i\right)p\left(x \mid y=C^i\right)}{\sum_{k=1}^{K} p\left(y=C^k\right)p\left(x \mid y=C^k\right)} \tag{3.23}$$

For a multi-class classification problem with K Gaussian functions, one for each class, to determine the class label the discriminant rule becomes:

$$C^k(x)=\arg\min_k \left\{\left(x-\mu^k\right)^T\left(\Sigma^k\right)^{-1}\left(x-\mu^k\right)+\log\left|\Sigma^k\right|\right\}$$

While powerful, the Bayes classifier remains a very simplistic model and is bound to make erroneous predictions far from the data, as it does not take into account the absolute value of the likelihood associated with each mode (only their relative importance). If both classifiers are predicting the labelof x with a very low likelihood (which happens when the data point x is very far from the training points or when the two classes overlap heavily in that region) then, deciding on one class label over the other is usually no better than random. Besides, when $p_+ \text{ and } p_-$ are two arbitrary densities, comparing these may be dangerous if one did not make sure that the quantities are comparable. Since a pdf is a positive, but unbounded (above), function, its likelihood may grow arbitrarily. It usually grows with the numer of parameters. A good precaution is then to make sure that the two classifiers have the same number of parameters and are trained on the same number of datapoints. The latter means that one must have at its disposal as many examples of the positive class as that of the negative class. If this can not be ensured, then one may apply some normalization term on the likelihood of each classifier or one may use a threshold on the likelihood of both models to determine when inference is warranted. Despite these caveats, Bayes classifiers remain quite popular, in part because of their extreme simplicity, but also because they yield very good performance in practice.

We see next how this can be used when using Gaussian Mixture Models.

3.3.2 Bayes classification with Gaussian Mixture Models

The Gaussian Mixture Model, introduced in Section 3.2.6, can be used in conjunction with the Bayes classifier to perform binary classification. For two knows classes of datapoints, one can train two separate GMM-s, one for each class of datapoints and use the Bayes classifier to determine the labeling of any given datapoint. In other word, each GMM yields a density $p_+ \text{ and } p_-$ which can then be used to determine the label of a new training point using (3.22).

To ensure that the two models are comparable, one should train two GMM with same number of Gaussians, using the same modeling (with full, diagonal or isotropic covariance matrix in each case), and using similar number of training points for each GMM. Figure 3-21 shows one example of such classification using two GMM-s with 8 Gaussians each.

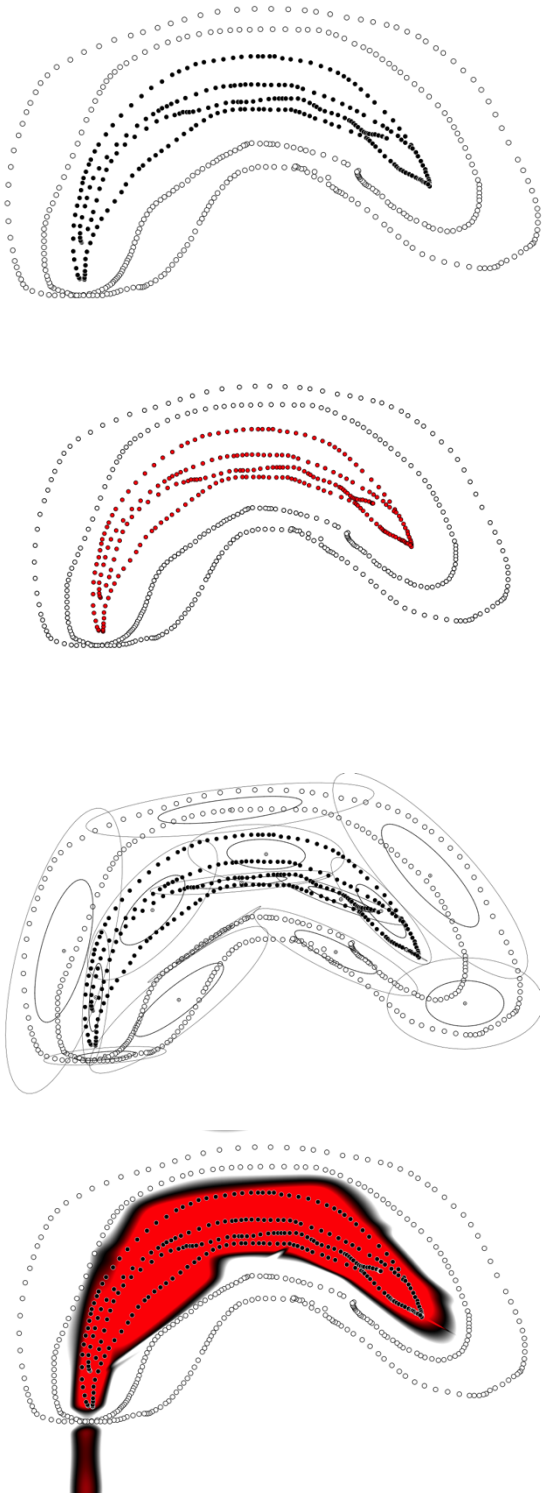


Figure 3-21: Bayes classification using a two GMMs with 8 Gaussians each and full covariance matrix. From left to right top to bottom: i) original datapoint (in dark, datapoints belonging to class +1, empty circles denote datapoints belonging to class -1), ii) result of the classification; all datapoints are correctly classified in their respective class; iii) the 16 Gaussians superimposed on the datapoints; iv) the region associated to each class using Bayes classification.

3.3.3 Linear Classifiers

In this section, we will consider solely ways to provide a linear classification of data. Non-linear methods for classification such as ANN with backpropagation and Support Vector Machine will be covered later on, in these lecture notes.

Linear Discriminant Analysis and the related Fisher's linear discriminant are methods to find the linear combination of features (projections of the data) that best *separate two or more classes of objects*. The resulting combination may be used as a *linear classifier.* We describe these next.

3.3.3.1 Linear Discriminant Analysis

Linear Discriminant Analysis (LDA) combines concepts of PCA and clustering to determine projections along which a dataset can be best separated into two distinct classes.

Consider a data matrix composed of *M N*-dimensional data points, i.e. $X \in \mathbb{R}^{M\times N}$. LDA aims to find a *linear* transformation $A \in \mathbb{R}^{M\times P}$ that maps each column x_i of *X*, for $i = 1,...M$ (these are N-dimensional vectors) into a corresponding *q*-dimensional vector y_i. That is,

$$A : x_i \in \mathbb{R}^N \to y_i = Ax_i \in \mathbb{R}^q \left(q \le N\right) \tag{3.24}$$

Let us further assume that the data in *X* is partitioned into *K* classes $\left\{C^k\right\}_{k=1}^K$, where each class C^k contains n^k data points and $\sum_{k=1}^K n^k = M$. LDA aims to find the optimal transformation *A* such that the class structure of the original high-dimensional space is preserved in the low-dimensional space.

In general, if each class is tightly grouped, but well separated from the other classes, the quality of the cluster is considered to be high. In discriminant analysis, two scatter matrices, called *within-class* (S_w) and *between-class* (S_b) matrices, are defined to quantify the quality of the clusters as follows:

$$\begin{aligned} S_w &= \sum_{k=1}^K \sum_{x\in C^k} \left(x-\mu^k\right)\left(x-\mu^k\right)^T \\ S_b &= \sum_{k=1}^K n^k \left(\mu^k-\bar{\mu}\right)\left(\mu^k-\bar{\mu}\right)^T \end{aligned} \tag{3.25}$$

Where $\mu^k = \frac{1}{n^k}\sum_{x\in C^k}$, $\bar{\mu} = \frac{1}{N}\sum_{k=1}^K \sum_{x\in C^k}$ are, respectively, the mean of the k-th class and the global mean. An implicit assumption of LDA is that all classes have equal class covariance (otherwise, the elements of the within-class matrix should be normalized by the covariance on the set of data points of that class).

When the transformation *A* is linear, as given by 2.40, solving the above problem consists of finding the optimum of $J\left(A\right) = \frac{\left|A^T S_b A\right|}{\left|A^T S_w A\right|}$, where $\left|\ \right|$ represents the matrix determinant. Equivalently, this

reduces to maximizing trace(S_b) and minimizing trace(S_w). It is easy to see that trace(S_w) measures the closeness of the vectors within the classes, while trace(S_w) measures the separation between classes.

Such an optimization problem is equivalent to an eigenvalue problem of the form $S_b x = \lambda S_w x, \quad \lambda \neq 0$. The solution can be obtained by performing an eigenvalue decomposition on the matrix $S_b^{-1} S_w$ if S_b is non singular or on $S_w^{-1} S_b$ if S_w is non singular.

There are at most *K*-1 eigenvectors corresponding to nonzero eigenvalues, since the rank of the matrix S_b is bounded from above by *K*-1. In effect, LDA requires that at least one of the two matrices S_b, S_w be non-singular. The above problem is an intrinsic limitation of LDA and referred to as the *singularity problem* that is, it fails when all scatter matrices are singular. If both matrices are singular, then, one can use extension of LDA using pseudo-inverse transformation. Another approach to deal with the singularity problem is to apply an intermediate dimension reduction stage using Principal Component Analysis (PCA) before LDA.

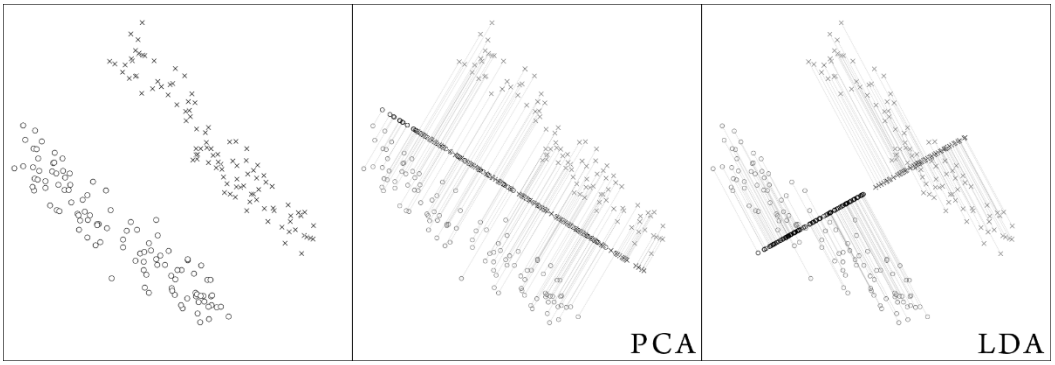


Figure 3-22: If data is elongated, the principal components (center) will not help separating the two clusters. However, LDA is able to find a projection that helps the separation of the data.

3.3.3.2 Fisher Linear Discriminant

The *Fisher's linear discriminant* is very similar to LDA in that it aims at determining a criterion for optimizing classification by increasing the between class similarity and decreasing the within class similarity. It however differs from LDA in that it does not assume that the classes are as normally distributed classes or equal class covariances. For two classes whose probability distribution functions have associated means μ_1, μ_2 and covariance matrices $\sum_1, \sum_2$, then the between and within-classes matrices are given by:

$$\begin{aligned} S_w &= \left(\mu^1 - \mu^2\right)\left(\mu^1 - \mu^2\right)^T \\ S_b &= \sum^1 + \sum^2 \end{aligned} \tag{3.26}$$

In the case where there are more than two classes (see example for LDA), the analysis used in the derivation of the Fisher discriminant can be extended to find a subspace which appears to contain all of the class variability. Then the within and between class matrices are given by:

$$S_w = \sum_{k=1}^{K} n^k \left(\mu^k - \bar{\mu}\right)\left(\mu^k - \bar{\mu}\right)^T$$
$$S_b = \sum_{k=1}^{K} \Sigma^k \tag{3.27}$$

When classification results from a linear projection through a map *A,* as in LDA, then again the problem corresponds to maximizing $J(A) = \frac{\left|A^T S_b A\right|}{\left|A^T S_w A\right|}$ with S_w, S_b defined as above.

.

3.3.4 Mixture of linear classifiers (boosting and bagging)

Adapted from Bagging Predictors by Leo Breiman, Machine Learning, 24, 123–140 (1996)

Linear classifiers as described previously rely on regression and can perform only binary classification. To extend the domain of application of classifiers to allow classification of more than two classes, linear classifiers can be combined. A number of combining techniques can be used, among which *bagging* and *boosting* have appeared in recent years as the most popular methods due to their simplicity.

All of these methods modify the training data set, build classifiers on these modified training sets, and then combine them into a final decision rule by simple or weighted majority voting. However, they perform in a different way. We briefly review *bagging* and *boosting* next.

3.3.4.1 Bagging

Let us assume, once more, that the complete training set is composed of *M* data points $\left\{x^i\right\}_{i=1,..M}$ .Bagging consists in selecting at random *K* subsets of training data $X^k \; k = 1,...K$. Each of these subsets will be used to create a classifier $\left\{C^k\left(X^k\right)\right\}_{k=1}^{K}$. The final classifier is composed of a linear combination of the classifiers, so that each data point *x* is then classified according to the function:

$$c(x) = \sum_{k=1}^{K} C^k(x) \tag{3.28}$$

The hope is, thus, that the *aggregation* of all the classifiers will give better classification results than training a single classifier on the whole dataset. Intuitively, it is easy to see that if the classification method bases its estimation on the average of the data, for a very heterogeneous dataset with local structures in the data, such an approach would fail to properly represent the local structure. A set of classifiers based on smaller subsets of the data may have more chances to catch these local structures.

3.3.4.2 Boosting / Adaboost

Boosting starts also from a linear combination of classifiers trained on subset or on the complete dataset. A weight is associated to each instance of the training set as well to each classifier. Weights associated to the instances are adapted at each iteration step of the procedure to reflect how well the instance is predicted by the global classifier. The less well classified the data point, the bigger its associated weight. This way, poorly classified data will be given more influence on the measure of the error and thus when retraining the classifiers, they should be better estimated.

Similarly the classifiers are weighted when combined to form the final classifier, so as to reflect their classification power. The poorer the classification power of a given classifier on the training set associated with this classifier, the less influence the classifier is given for final classification.

Note:

Boosting and Bagging are methods based on a supervised process, where the “real” class of the data is known a priori. In contrast, the clustering techniques seen in the first part of this chapter work in an unsupervised manner whereby the true labeling of the data is unknown.

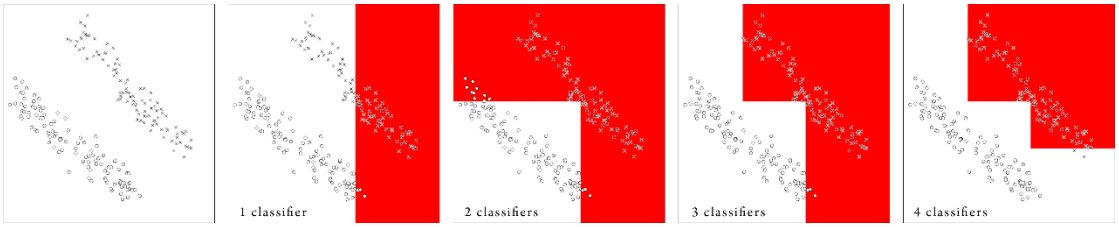


Figure 3-23: Classification using a linear combination of very simple classifiers trained by boosting. Each classifier on its own would not be able to separate the data. The linear combination is able to separate complex data.

3.4 Further Readings

There exist numerous methods to cluster or classify data. This chapter dealt only with some of the major algorithms for clustering and classification from which most state-of-the-art methods are derived. In this manuscript, we will also see two other techniques for classification as part of the Artificial Neural Networks chapter and the Kernel Methods chapters.

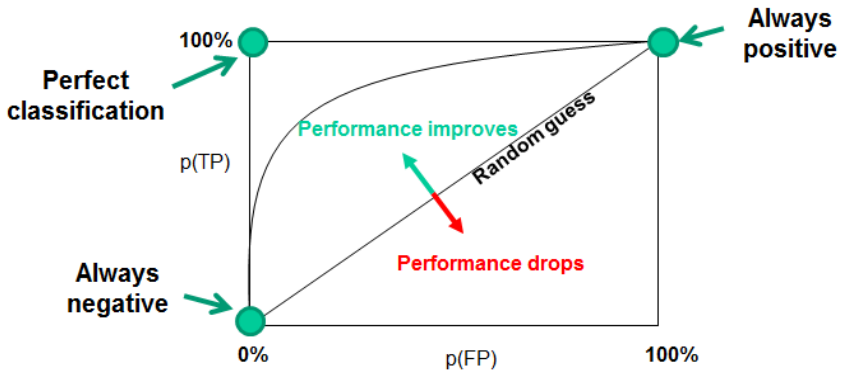
3.5 Performance Measures for Clasification

3.5.1 ROC

Graphical representations such as the Receiver Operating Characteristic, so called ROC curve provide an interesting way to quickly compare algorithms independently of the choice of parameters.

The ROC curve is used in binary classification problems. It balances the number of *true positives (TP),* i.e. number of instances when the algorithm classified correctly a datapoint of the 1st class to the 1st class again the numbe of *false positives (FP),* i.e. number of instances when the algorithm classified incorrectly datapoints of the 2nd class to the 1st class.

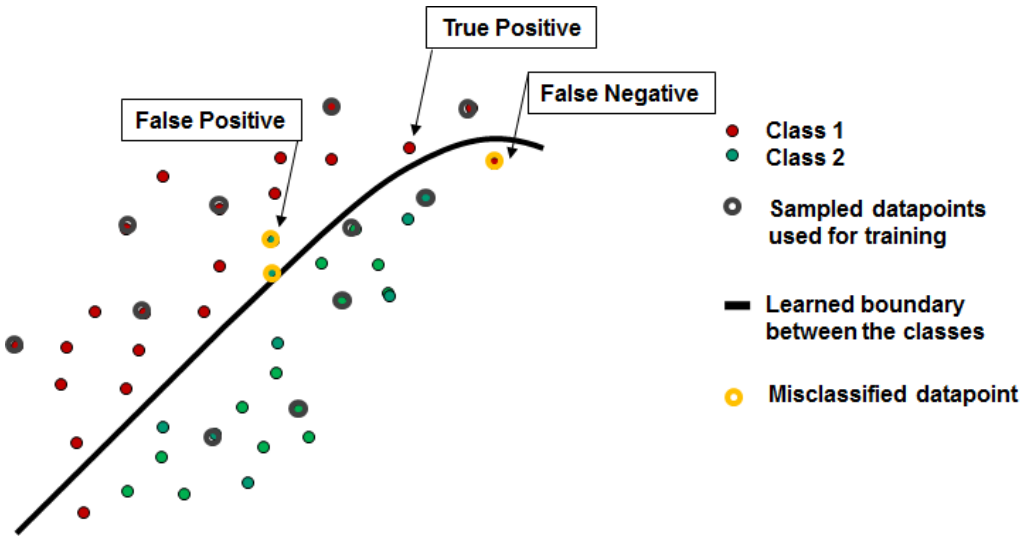
The ROC curve plots the fraction of true positives (TP) and false positives (FP) over the total number of samples of the 1st class in the dataset. Each point on the curve corresponds to a different value of the classifier's parameter (usually a threshold; e.g. a threshold on Bayes' classification). A typical ROC curve is drawn below.



As you may have noticed, the ROC curve tends to measure only classification performance with respect to one class, the 1st class. This is due to the fact that one often cares about achieving excellent classification on one class against the rest (the non-class), e.g. recognizing human faces versus any other image which is not a human face.

If, however, one cares about computing performance on both classes, one will also compute the number of *false negatives (FN)*, namely the number of instances when the algorithm classified a datapoint of the 2nd class to the 1st class.

Below you can find a schematic illustrating the notion of FP, TP and FN in a binary classification example.



3.6 Further Readings

There exist numerous methods to cluster or classify data. This chapter dealt only with some of the major algorithms for clustering and classification from which most of state-of-the-art methods are derived. In this manuscript, we will also see two other techniques for classification as part of the Artificial Neural Networks chapter. These are the perceptron and the backpropagation algorithm.

In part-II of this manuscript, we will also consider other techniques for classification, such as Support Vector Machine and Support Vector Clustering that exploit non-linear transformation of the data through *kernel* projection.

4 Regression Techniques

There is a growing interest in machine learning to design powerful algorithms for performing non-linear regression. We will here consider a few of these. The principle behind the technique we will present here is the same as the one used in most other variants we find in the literature and hence this offers a good background for an interested reader.

Consider a multi-dimensional (zero-mean) variable $x \in \mathbb{R}^N$ and a *one-dimensional* variable $y \in \mathbb{R}$, regression techniques aim at approximating a relationship *f* between y and x by building a model of the form:

$$y = f(x) \tag{4.1}$$

4.1 Linear Regression

The most classical technique that the reader will probably be familiar with is the linear regressive model, whereby one assumes that f is a linear function parametrized by $w \in \mathbb{R}^N$, that is:

$$y = f(x, w) = x^T w \tag{4.2}$$

For a given instance of the pair x and y one can solve explicitly for w. Consider now the case where we are provided with a set of *M observed* instances $X = \{x^i\}_{i=1}^M$ and $Y = \{y^i\}_{i=1}^M$ of the variables x and y such that the observation of y has been corrupted by some noise which may or not be a function of x, i.e. $\varepsilon(x)$:

$$y = x^T w + \varepsilon(x) \tag{4.3}$$

Classical means to estimate the parameters w is through mean-square, which we review next.

4.2 Partial Least Square Methods

Adapted from R. Rosipal and N. Kramer, Overview and Recent Advances in Partial Least Squares, C. Saunders et al. (Eds.): SLSFS 2005, LNCS 3940, pp. 34–51, 2006.

Partial Least Squares (PLS) refer to a wide class of methods for modeling relations between sets of observed variables by means of latent variables. It comprises *regression* and *classification* tasks as well as *dimension reduction* techniques and modeling tools. As the name implies, PLS is based on least-square regression. Consider a set of *M* pairs of variables $X = \{x^i\}_{i=1}^M$ and $Y = \{y^i\}_{i=1}^M$, least-square regression looks for a mapping *w* that sends *X* onto *Y*, such that:

.

$$\min_w \left(\sum_{i=1}^N \frac{1}{2} \left(x^i w - y^i \right)^2 \right) \tag{4.4}$$

Note that each variable pair must have the same dimension. This is hence a particular case of regression.

PCA and CCA by extension can be viewed as regression problems, whereby one set of variable *Y* can be expressed in terms of a linear combination of the second set of variable *X*.

The primary problem with PCA Regression is that PCA does not take into account the response variable when constructing the principal components or latent variables. Thus even for easy classification problems such as that shown in Figure 4-1, the method may select poor latent variables. PLS incorporate information about the response in the model by using latent variables.

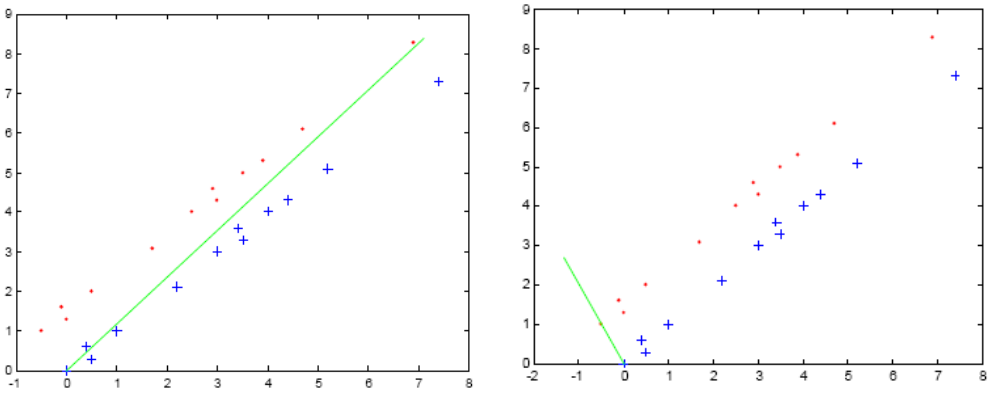


Figure 4-1: **LEFT:** Line is direction of maximum variance (*w*) constructed by PCA. **RIGHT:** Line is direction *w* constructed by PLS.

The underlying assumption of all PLS methods is that the observed data is generated by a system or process which is driven by a small number of latent (not directly observed or measured) variables.[illegible] its principle. It however differs in the algorithm.

The connections between PCA, CCA and PLS can be seen through the optimization criterion they use to define projection directions. PCA projects the original variables onto a direction of maximal variance called principal direction. Similarly, CCA finds the direction of maximal correlation across two sets of variables. PLS represents a form of CCA, where the criterion of maximal correlation is balanced with the requirement to explain as much variance as possible in both X and Y spaces. Formally, CCA and PLS can be viewed as finding the solution to the following optimization problem. For two sets of variables *X* and *Y*, find the set of vectors w_x, w_y that maximizes the following quantity:

$$\max_{|w_x|=|w_y|=1} \frac{\operatorname{cov}\left(Xw_x, Yw_y\right)^2}{\left(\left[1-\gamma X\right]\operatorname{var}\left(Xw_x\right)+\gamma X\right)\left(\left[1-\gamma Y\right]\operatorname{var}\left(Yw_y\right)+\gamma Y\right)} \tag{4.5}$$

For $\gamma X = \gamma Y = 0$, the above optimization leads to CCA. While for $\gamma X = \gamma Y = 1$, the solution to PLS is found.

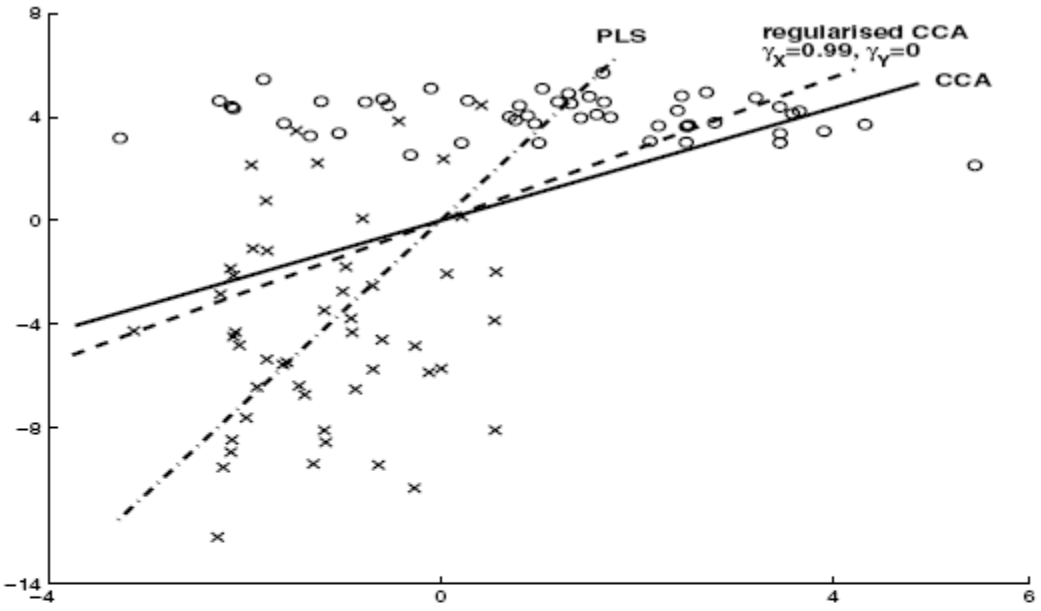


Fig. 1. An example of the weight vector w directions as found by CCA (solid line), PLS (dash-dotted line) and regularised CCA (dashed line) given by (7) with $\gamma_X = 0.99$ and $\gamma_Y = 0$. Circle and cross samples represent two Gaussian distributed classes with different sample means and covariances.

4.3 Probabilistic Regression

Probabilistic regression is a statistical approach to classical linear regression and assumes that the observed instances of x and y have been generated by an underlying probabilistic process of which it will try to build an estimate. The rationale goes as follows:

If one knows the joint distribution $p(x, y)$ of x and y, then an estimate $\hat{y}$ of the output *y* can be built from knowing the input *x* by computing the expectation of y given x,

$$\hat{y} = E\{p(y \mid x)\} \tag{4.6}$$

Note that in many cases, if one is solely interested in constructing a regressive model, one does not need to build a model of the joint density but needs solely to estimate the conditional $p(y \mid x)$.

Probabilistic regression extends the concept of linear regression by assuming that the observed values of y differ from $f(x)$ by an additive random noise ε (noise is usually assumed to be independent of the observable x):

$$y = f(x, w) + \varepsilon \tag{4.7}$$

Usually, to simplify computation, one further assumes that the noise follows a zero-mean Gaussian distribution with uncorrelated isotropic variance σ^2. The covariance matrix is diagonal with all

elements equal to σ^2; so we write simply $\varepsilon = N\left(0,\sigma^2\right)$. Such an assumption is called *putting a prior distribution over the noise*.

Let us first consider the probabilistic solution to the linear regression problem described before, that is:

$$y = xw^T + N\left(0,\sigma^2\right) \tag{4.8}$$

We have now one more variable to estimate, namely the variance of the noise σ.

Assuming that all pairs of observable are i.i.d (identically independently distributed), we can construct an estimate of the conditional probability of y given x and a choice of parameters $w,\ \sigma$ as:

$$p\left(y \mid x, w, \sigma\right) = \prod_{i=1}^{M} p\left(y \mid x^i, w, \sigma\right) \tag{4.9}$$

Solving for the fact that sole the noise model is probabilistic and that it follows a Gaussian distribution with zero mean, we obtain:

$$\begin{aligned} \Rightarrow p\left(y \mid x, w, \sigma\right) &= \prod_{i=1}^{M} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{\left(y_i - x_i^T w\right)^2}{2\sigma^2} \right) \\ &= \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2\sigma^2} \left(y - X^T w\right)^T \left(y - X^T w\right) \right) \end{aligned} \tag{4.10}$$

In other words, the conditional is a Gaussian distribution with mean $X^T w$ and covariance matrix $\sigma^2 I$, i.e.

$$p\left(y \mid x, w, \sigma\right) = N\left(X^T w, \sigma^2 I\right) \tag{4.11}$$

.

In *Bayesian formalism*, one would also specify a *prior* over the parameter *w*. Typically, one would assume a *zero mean Gaussian* prior with fixed covariance matrix $\sum_w$:

$$p\left(w\right) = N\left(0, \sum_w\right) = \exp\left(-\frac{1}{2} w^T \sum_w^{-1} w \right) \tag{4.12}$$

To determine the optimal set of parameters One can then compute the *posterior distribution* over the parameter *w* using Bayes' Theorem:

$$\text{posterior} = \frac{\text{likelihood} \times \text{prior}}{\text{marginal likelihood}}, \qquad p\left(w \mid X, y\right) = \frac{p\left(y \mid X, w\right) p\left(w\right)}{p\left(y \mid X\right)} \tag{4.13}$$

The marginal likelihood is independent of the weight and can be computed from our current estimate of the likelihood and given our prior on the weight distribution:

$$p(y \mid X) = \int p(y \mid X, w)\, p(w)\, dw \tag{4.14}$$

The quantity we are interested in is the posterior over the weight, which can now solve for:

$$\begin{aligned} p(w \mid X, y) &\propto \exp\left(-\frac{1}{2}\left(y - X^T w\right)^T \left(y - X^T w\right)\right) \cdot \exp\left(-\frac{1}{2} w^T \sum_w^{-1} w \right) \\ &\propto \exp\left(-\frac{1}{2}(w-v)^T \left(\frac{1}{\sigma^2} XX^T + \sum_w^{-1} \right)(w-v) \right) \\ &\propto \exp\left(-\frac{1}{2}(w-v)^T \sum_v^{-1} (w-v) \right) \\ \text{where } v = \sigma^{-2} &\left(\sigma^{-2} XX^T + \sum_w^{-1} \right)^{-1} Xy \quad \text{and} \quad \sum_v = \left(\sigma^{-2} XX^T + \sum_w^{-1} \right)^{-1} \end{aligned} \tag{4.15}$$

The posterior distribution of the weight is thus a Gaussian distribution with mean v and covariance matrix $\sum_v$. Notice that the first term on the right handside of both terms is the covariance of the input modulated by the variance of the noise. In a deterministic model where the variance of the weight is zero and the variance of the noise is the identity, we recover the classical linear regressive model.

We can compute our best estimate of the weight by computing the expectation over the posterior distribution, i.e. $E\left\{ p(w \mid y, x) \right\} = \sigma^{-2} \left(\sigma^{-2} XX^T + \sum_w^{-1} \right)^{-1} Xy$. This is called the *maximum a posteriori* (MAP) estimate of w. Further, notice that the posterior of the weight and by extension its MAP estimate depend not only on the input variable X but also on the output variable y. X is a $N \times M$ matrix. In practice, computation of MAP grows quadratically with the number of examples *M* and is a major drawback of such methods. Current efforts in research related to the development of such regression techniques are all devoted to reducing the dimension of the training set to make this computation bearable. These are so-called *sparse* methods. We will discuss these when we covere kernel methods later in these notes.

Once the parameters w have been so estimated, we can use our probabilistic regressive model to make predictions given new data points. Hence, given a so-called *query* point x^* (such point is usually a point not used for training the model; in other words, this would be a point belonging to the test set or the validation set), we can compute our estimate of its associated $y^* = f\left(x^*\right)$. This is usually done by averaging over all possible values for the weight, i.e.:

$$\begin{aligned} p\left(y^* \mid x^*, X, y\right) &= \int p\left(y^* \mid x^*, w\right) p(w \mid X, y)\, dw \\ &= N\left(\frac{1}{\sigma^2} x^{*T} \sum_w Xy, x^{*T} \sum_w x^* \right) \end{aligned} \tag{4.16}$$

The predictive distribution is again Gaussian. Notice that the uncertainty (variance) on the predictive model grows quadratically with the query point and with the variance of the weight

distribution. This is expected from linear regressive model and hence is a limitation. This effect is illustrated in Figure 4-2.

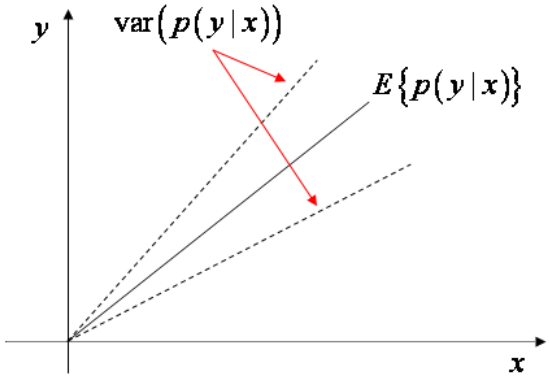


Figure 4-2: Illustration of the fact that in a probabilistic regressive model the uncertainty on the predictive model increases quadratically with the amplifude of the input.

This could be overcome if one was to build several local models and perform regression locally, as we will see next through Gaussian Mixture Regression. An alternative method which will be covered in the chapter on kernel Methods is Gaussian Process Regression.

4.4 Gaussian Mixture Regression

Adapted from Hsi Guang Sung, Gaussian Mixture Regression and Classification, PhD thesis, Rice University, 2004.

In Section 3.2.6, we introduced Gaussian Mixture Models (GMM). Assume $z \in \mathbb{R}^N$ a multi-dimensional variable GMM builds a model of $p(z) = p(z_1,...,z_N)$ of the joint distribution of x using a mixture of $\mathrm{k} = 1,..,K$ Gaussians $p_k\left(z \mid \mu^k, \Sigma^k\right) = N\left(\left[\mu^k, \Sigma^k\right]\right)$ with mean $\mu^k \in \mathbb{R}^N$ and covariance matrix $\sum^k \in \mathbb{R}^{N \times N}$, i.e.

$$p(z) = \sum_{k=1}^{K} \alpha_k \cdot p_k\left(z \mid \mu^k, \Sigma^k\right) = \sum_{k=1}^{K} \alpha_k \cdot \frac{1}{\sqrt{2\pi}\left|\Sigma^k\right|^{\frac{N}{2}}} \exp\left(\left(z-\mu^k\right)^T \left(\Sigma^k\right)^{-1}\left(z-\mu^k\right)\right) \tag{4.17}$$

The mixing coefficients α_k satisfy $\sum_{k=1}^{K} \alpha_k = 1$.

Gaussian Mixture Regression (GMR) exploits the joint density to construct its estimate using the expectation on the conditional of the variable onto which we wish to make a prediction. If for instance, we wish to predict z_1 from knowing all other entries $z_2,...z_N$, then we can compute:

$$\hat{z}_1 = E\left\{p\left(z_1 \mid z_2,...,z_N\right)\right\} = \int z_1 \cdot p\left(z_1 \mid z_2,...,z_N\right) \cdot dz_1 \tag{4.18}$$

When applying Bayes' theorem, we get:

$$z_1 = \frac{\int z_1 \cdot p\left(z_1, z_2, ..., z_N\right) \cdot dz_1}{\int p\left(z_1, z_2, ..., z_N\right) \cdot dz_1} \tag{4.19}$$

In contrast to most regression techniques, GMR also allows to make prediction on multi-dimensional output in one swipe. So, if we set $y = \left\{z_1, z_2, z_3\right\}$ and $x = \left\{z_4, ..., z_N\right\}$, then we can compute:

$$\hat{y} = E\left\{p\left(y \mid x\right)\right\} = E\left\{p\left(z_1, z_2, z_3 \mid z_4, z_5, ..., z_N\right)\right\} \tag{4.20}$$

4.4.1 One Gaussian Case

Let us first consider first the case where the joint density $p\left(z\right) = p\left(y, x\right)$ follows a single Gaussian distribution (and not a mixture). Using the fact that, if the joint distribution of two variables is Gaussian then each conditional distribution is also Gaussian, see Figure 9-3 for an illustration,we can compute the conditional density $p\left(y \mid x\right)$ which is given by:

$$p\left(y \mid x\right) = N\left(\mu_y + \Sigma_{YX}\left(\Sigma_{XX}\right)^{-1}\left(x - \mu_x\right), \Sigma_{YX} - \Sigma_{YX}\left(\Sigma_{XX}\right)^{-1}\Sigma_{XY}\right) \tag{4.21}$$

where μ_Y and μ_X are the means of the variable y and x respectively and have thus the dimensions of y and x, which we shall call N^Y, N^X. Σ_{XX} the covariance matrix of the input variable x has dimension $N^X \times N^X$ and the cross-covariances matrices Σ_{YX}, Σ_{XY} have dimension $N^Y \times N^X, N^X \times N^Y$ respectively.

In predictive regression, we are interested in computing the expectation and uncertainty of the predictive regressive model at a given query point. When the conditional density is Gaussian, this is easy to obtain and we thus have for a query point x^*, the estimated output:

$$\hat{y}^* = E\left\{p\left(y \mid x^*\right)\right\} = \mu_Y + \Sigma_{YX}\left(\Sigma_{XX}\right)^{-1}\left(x - \mu_X\right) \tag{4.22}$$

with associated uncertainty:

$$\mathrm{var}\left(p\left(y \mid x\right)\right) = \Sigma_{YX} - \Sigma_{YX}\left(\Sigma_{XX}\right)^{-1}\Sigma_{XY} \tag{4.23}$$

4.4.2 Multi-Gaussian Case

It is easy to extend the previous result to a mixture of Gaussians, where $p(x,y)$ is given by Equation(4.17). Recall that each joint density $p_k(y,x),\ k=1,..K$ can be decomposed as $p_k(y,x)=p_k(y\,|\,x)\cdot p_k(x)$ (Bayes' Theorem), then the joint density of the complete mixture is given by:

$$p(y,x)=\sum_{k=1}^{K}\alpha_k\cdot p_k\left(y\,|\,x,\mu^k(x),\Sigma^k\right)\cdot p_k\left(x\,|\,\mu_X^k,\Sigma_{XX}^k\right) \tag{4.24}$$

with $\mu^k(x)=\mu_Y^k+\Sigma_{YX}^k\left(\Sigma_{XX}^k\right)^{-1}\left(x-\mu_X^k\right)$ and $\Sigma^k=\Sigma_{YX}^k-\Sigma_{YX}^k\left(\Sigma_{XX}^k\right)^{-1}\Sigma_{XY}^k$.

The marginal density of x is given by:

$$p_k\left(x\,|\,\mu^k,\Sigma^k\right)=\sum_{k=1}^{K}p_k\left(x\,|\,\mu^k,\Sigma^k\right) \tag{4.25}$$

Replacing (4.25) in (4.24), we can express the conditional probability of y given x:

$$p(y\,|\,x)=\sum_{k=1}^{K}w_k(x)\cdot p_k(y\,|\,x) \tag{4.26}$$

where $w_k=\dfrac{\alpha_k p_k\left(x\,|\,\mu^k,\Sigma^k\right)}{\sum_{j=1}^{K}\alpha_j p_j\left(x\,|\,\mu^j,\Sigma^j\right)}$.

Equation (4.26) forms the core of the GMR model. One can then compute explicitly the expectation and variance on the above conditional, similarly to what we did for the probabilistic regression.

$$E\left\{p(y\,|\,x)\right\}=\sum_{k=1}^{K}w_k(x)\cdot\mu^k(x)=\sum_{k=1}^{K}w_k(x)\cdot\left(\mu_Y^k+\Sigma_{YX}^k\left(\Sigma_{XX}^k\right)^{-1}\left(x-\mu_X^k\right)\right) \tag{4.27}$$

The expectation is thus a non-linear combination of the expectation of each local component. In effect, the regression signal from GMR is the result of a non-linear weighting of local linear regressions.

$$\begin{aligned}&\mathrm{var}\left\{p(y\,|\,x)\right\}=\sum_{k=1}^{K}w_k(x)\cdot\left(\left(\mu^k(x)\right)^2+\left(\tilde{\Sigma}^k\right)\right)-\sum_{k=1}^{K}\left(w_k(x)\cdot\mu^k(x)\right)^2\\&\tilde{\Sigma}^k=\Sigma_{yy}^k-\Sigma_{yx}^k\left(\Sigma_{xx}^k\right)^{-1}\tilde{\Sigma}_{xy}^k\end{aligned} \tag{4.28}$$

The variance of GMR is no longer a simple function increasing with the amplitude of the input x (as in probabilistic regression). Rather it is modulated by the variance of each component locally and hence carries across a notion of local variance.

A schematic of these variables is shown in Figure 4-3.

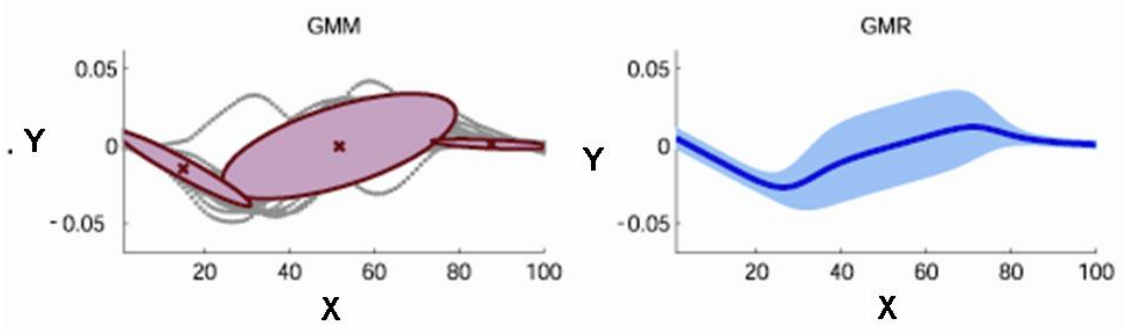


Figure 4-3: Two-dimensional illustration of a) left: a Gaussian Mixture Model with 3 Gaussians (in violet) superposed to the original set of pairs (x,y) of datapoints (in grey), b) the Gaussian Mixture Regression model resulting from the GMM. The dark line is the regression signal, i.e. $E\{p(Y \mid X)\}$. The blue envelope around it is one standard devitation computed from $\mathrm{var}\{p(Y \mid X)\}$.

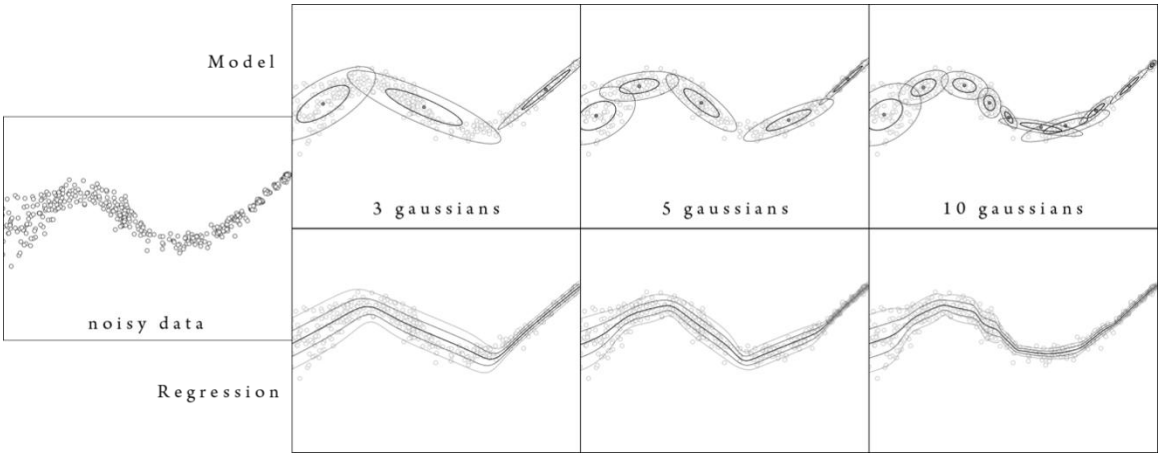


Figure 4-4: Gaussian Mixture Model (top row) and Regression (bottom row) with different amounts of Gaussians in the mixture. The center line shows the regression mean, while the two thinner lines display respectively one and two sigmas. The regression function becomes more detailed when the data is modeled by more Gaussians, and represents better the modulation of its variance.[DEMOS\REGRESSION\GMR-THINNING.ML]

5 Kernel Methods

These lecture notes have sofar covered *linear* methods for performing a variety of computation, ranging from dimensionality reduction, clustering and classification to regression. The term *linear* refers to the assumptions that all of these transformations could be expressed as *linear* transformation of the form $y = Ax$ (where y and x are given and *A* is the unknown transformation).

Kernel methods relax the assumption of a linear transformation, so as to perform non-linear regression, classification, etc. Kernel methods proceed by first projecting the data through a non-linear transformation into a *feature* space and then perform the same type of computation (e.g. classification or regression), as in the linear case, in the feature space. This principle is illustrated in Figure 5-1.

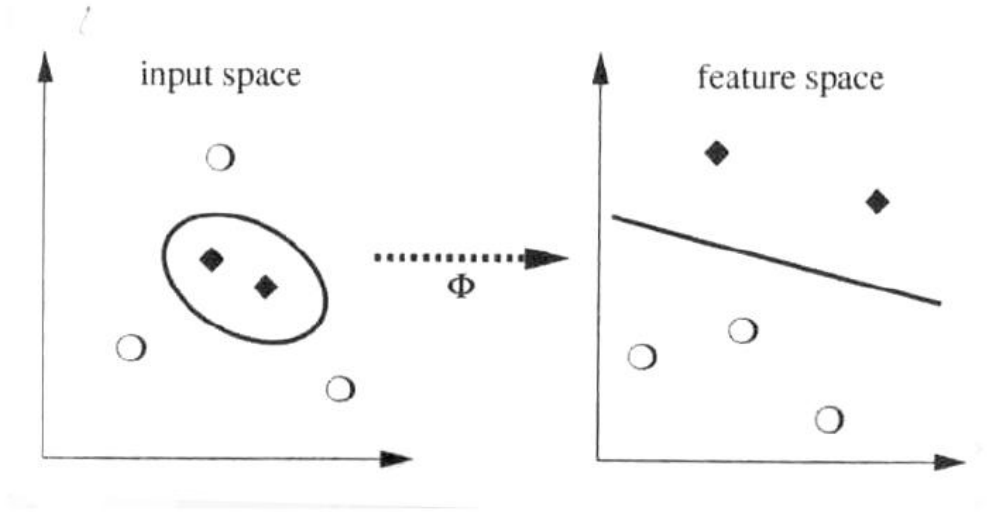


Figure 5-1: Illustration of the principle of non-linear classification through Support Vector Machine. The data are first projected from input space to a feature space via a non-linear map ϕ. Linear separation is then performed in the feature space. Figure from "Learning with Kernels" by B. Scholkopf and A. Smola, MIT Press 2002.

In these lecture notes, we cover a subset of existing Kernel methods. We focus on kernel methods that extend the algorithms we presented previously for dimensionality reduction (from PCA, CCA, ICA to kernel PCA, kernel ICA, kernel CCA), for clustering (from K-Means to kernel K-means), for classification (from linear classifier to support vector machine) and for regression (from probabilistic regression to Gaussian Process Regression).

5.1 The kernel trick

All kernel methods we will see here are based on the so-called *kernel trick*. The kernel trick stems from a variety of fundamental mathematical principles which we will skip here, focusing solely on explaining its basic principle.

If $X = \left\{x^i\right\}_{i=1}^{M}$ is the set of our *M* observations, we can construct a projection of X onto the feature space H through a non-linear mapping ϕ:

$$\begin{aligned} &\phi : X \to H \\ &x \to \phi(x). \end{aligned} \tag{5.1}$$

Note that if the dimension of X is N , then the dimension of the feature space may be greater than N .

Finding the appropriate non-linear transformation ϕ is difficult in practice. The kernel trick relates to the observation that most methods rely on computing an *inner* product $\langle x^i, x^j \rangle$ across pairs of observations $x^i, x^j \in X$ (e.g., recall that in order to perform PCA, we needed to compute the covariance matrix on X , which is given by the dot product on all the data points, i.e. XX^T ; similarly when performing linear regression, we had to multiply by the inverse of the covariance on X).

Hence, determining an expression for this inner product in feature space may save us from having to compute explicitly the mapping ϕ . This inner product in feature space is expressed through the *kernel function* (or simply the *kernel*) $k(.,.)$.

All the kernel methods we will see in these lecture notes assume that the feature space *H* into which the data are sent through the non-linear function ϕ is a *Reproducing Kernel Hilbert Space (RKHS).H* is hence a space $H := \{h_1(.), h_2(.), ...\}$ of functions from $\mathbb{R}^N$ to $\mathbb{R}$. In this space, $\phi(., x^i)$, $\phi(., x^j)$ $\forall x^i, x^j \in X$ defines a set of functions indexed by x . The "reproducing property" of the kernel ensures that taking the inner product between a pair of functions in *H* yields a new function in *H* . Hence setting $k(., x) := \phi(., x)$ yields the following kernel:

$$k(x^i, x^j) = \langle \phi(., x^i), \phi(., x^j) \rangle = \int_{-\infty}^{\infty} \phi(z, x^i), \phi(z, x^j) dz \tag{5.2}$$

The kernel in (5.2) is the dot product in feature space. Most techniques covered in these lecture notes will be based on this kernel.

Note that, in the literature, people tend to omit the open parameter on $\phi(., x^i)$ and write simply $\phi(x^i)$. We will follow this notation in the remainder of this document.

For proper use in the algorithms we will see next, the kernel must satisfy a number of properties that follow the *Mercer's theorem.* Among these, we will retain that *k*is a symmetric continuous positive function that maps:

$$\begin{aligned} &k : X \times X \to \mathbb{R} \\ &k(x^i, x^j) \to \langle \phi(x^i), \phi(x^j) \rangle \end{aligned} \tag{5.3}$$

The kernel *k*provides a metric of similarity across datapoints. Using *k* may allow extracting features common to all training datapoints. These features are some non-linear correlations not visible in the original space.

Classical kernels one finds in the literature are:

- Homogeneous Polynomial Kernels: $k(x,x') = \langle x,x' \rangle^p, \quad p \in \mathbb{N};$
- Inhomogeneous Polynomial Kernels: $k(x,x') = \left(\langle x,x' \rangle^p + c \right), \quad p \in \mathbb{N},\ c \geq 0;$
- Hyperbolic Tangent Kernel (similar to the sigmoid function): $k(x,x') = \tanh\left(\theta + \langle x,x' \rangle \right), \quad \theta \in \mathbb{R};$
- Gaussian Kernel (translation-invariant): $k(x,x') = e^{-\frac{\|x-x'\|}{2\sigma^2}}, \quad \sigma \in \mathbb{R}.$

The most popular kernel is the Gaussian kernel.

5.1.1 Stationary and non-stationary kernels

A kernel is stationary if the kernel depends only on the distance between two points. The Gaussian kernel is stationary. The other three kernels listed above are non-stationary as they depend on the coordinatre of the points.

5.2 Which kernel, when?

A recurrent question we often hear in class is “how does one choose the kernel”. A number of recent works have addressed this problem by offering techniques to *learn* the kernel. This often amounts to building esimators that rely on mixture of kernels. One then learns how to combine these kernels to find the optimal mixture. There is however unfortunately no good recipe to determine which kernel to use when. One may either try different kernel in an iterative manner and look at which provides the best result for a given technique. For instance, if you perform classification, you may want to compare the performance obtained on both training and testing sets after crossvalidation when doing so with either Gaussian kernel or Polynomial kernel. You may then pick the kernel that yields to best performance.

While choosing the kernel is already an issue, once a kernel is chosen, one is left with the problem of chosing the hyperparameter of the kernel. These are, for instance, the variance σ in the Gaussian kernel or the order of the polynome p in the polynomial kernels. There again, there is no good recipe. When using a Gaussian kernel, a number of approaches, known under the term of kernel polarization, have been proposed whereby one learns the optimal covariance parameters. Most of these approaches however are iterative by nature and rely on a discrete sampling of values taken by the parameters, using some heuristics or boosting techniques to optimize the search through these parameter values.

5.3 Kernel PCA

We start our review of kernel methods with kernel PCA, a non-linear extension to Principal Component Analysis (PCA).

To recall, PCA is a powerful technique for extracting structure from high-dimensional data. In Section 2.2.1, of these lecture notes, we had reviewed two ways of computing PCA. We can either do it through an analytical decomposition into eigenvalues and eigenvectors of the data space, or through an iterative decomposition using Hebbian Learning in multiple input-output networks.

A key assumption of PCA was that the transformation applied on the data was linear. Kernel PCA is a generalization of the standard PCA, in that it considers any linear or non-linear transformation of the data.

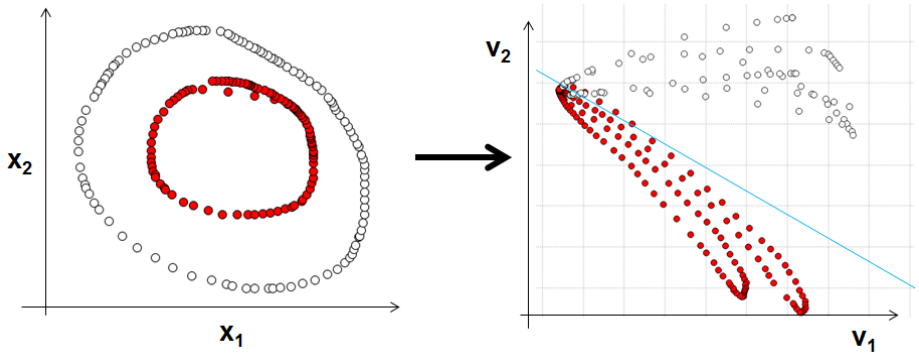


Figure 5-2: (RIGHT:) Example of a two-class dataset (red and white classes) in original space. (LEFT☺ Data becomes *linearly* separable when using a rbf kernel and projecting onto first 2 PC-s of kernel PCA.

We first start by reintroducing the PCA notation and then show how we can go from linear PCA to non-linear PCA.

Linear PCA:

Assume that the dataset is composed of a set of vectors $\mathrm{x}^i \in \mathbb{R}^N, i=1...M$. Assume further that the dataset is zero mean, i.e. $\sum_{i=1}^{M} x^i = 0$.

PCA finds an orthonormal basis such that the projections along each axis maximize the variance of the data. To do so, PCA proceeds by diagonalizing the covariance matrix of the dataset $\mathrm{C}=\mathrm{XX}^T$. C is positive definite and can thus be diagonalized with non-negative eigenvalues.
The principal components are then all the vectors $v^i, i=1...,N,$ solution of:

$$\mathrm{C}v^i = \lambda_i v^i \tag{5.4}$$

where λ_i is a scalar and corresponds to the eigenvalue associated to the eigenvector v^i.

Non-Linear Case:

Observe first that one can rewrite the eigenvalue decomposition problem of linear PCA in terms of dot product across pairs of the training datapoints. Indeed, each principal component v^i can be expressed as a linear combination of the datapoints. Using $\mathrm{C}=\frac{1}{M}\sum_{j=1}^{M} x^j \left(x^j\right)^T$ and replacing in (5.4), we obtain:

$$Cv^i = \frac{1}{M}\sum_{j=1}^{M} x^j \left(x^j\right)^T v^i = \lambda_i v^i \tag{5.5}$$

which allows us to express each eigenvector as follows:

$$\begin{aligned} v^i &= \frac{1}{M\lambda_i}\sum_{j=1}^{M} x^j \left(x^j\right)^T v^i \\ &= \frac{1}{M\lambda_i}\sum_{j=1}^{M}\left(x^j \left(v^i\right)^T\right) x^j \end{aligned} \tag{5.6}$$

Introducing a set of scalars $\alpha_j^i = \left(x^j \left(v^i\right)^T\right)$ for each data point x^j (these correspond to the value of the projection of x^j onto the eigenvector vector v^i.
Then each eigenvector v^i with non-zero eigenvalue λ^i lies in the space spanned by the input vectors $x^1, ..., x^M$:

$$v^i = \frac{1}{M\lambda^i}\sum_{j=1}^{M} \alpha_j^i x^j. \tag{5.7}$$

Assuming now that the data are projected into a feature space through a non-linear map ϕ and that these are centered in feature space, i.e.

$$\sum_{i=1}^{M} \phi\left(x^i\right) = 0, \tag{5.8}$$

the Correlation matrix in the feature space is then given by:

$$C_\phi = \frac{1}{M} FF^T \tag{5.9}$$

where each $i = 1, ..., M$ columns of F are composed of the projections $\phi\left(x^i\right)$.

As in the original space, in feature space, the correlation matrix C_ϕ can be diagonalized and we have now to find the eigenvalues $\lambda_i \geq 0,\ i = 1...M$, satisfying:

$$\begin{aligned} & C_\phi v^i = \lambda_i v^i \\ & \Rightarrow \left\langle \phi\left(x^j\right), C_\phi v^i \right\rangle = \lambda_i \left\langle \phi\left(x^j\right), v^i \right\rangle,\ \forall i, j = 1,...M \end{aligned} \tag{5.10}$$

Each eigenvector $v^1,...,v^M$ can be expressed as a linear combination of the images of the datapoints.

Rewriting PCA in terms of dot products:

Using $C_\phi v^i = \frac{1}{M}\sum_{j=1}^{M} \phi\left(x^j\right)\phi\left(x^j\right)^T v^i$ with $C_\phi v^i = \lambda_i v^i$

we obtain, $v^i = \frac{1}{\lambda_i M}\sum_{j=1}^{M} \phi\left(x^j\right)\underbrace{\phi\left(x^j\right)^T v^i}_{\alpha_j^i}$

All solutions v *with* $\lambda \neq 0$ lie in the span of the $\phi(\mathrm{x}^1),\ldots,\phi(\mathrm{x}^M)$ and we can thus write:

$$v^i = \frac{1}{\lambda_i M}\sum_{j=1}^{M} \alpha_j^i \phi\left(x^j\right). \tag{5.11}$$

Developping the left hand-side

$$\left\langle \phi\left(x^j\right), C_\phi v^i \right\rangle = \left\langle \phi\left(x^j\right), C_\phi \frac{1}{\lambda_i M}\sum_{j=1}^{M} \alpha_j^i \phi\left(x^j\right) \right\rangle$$

$$= \frac{1}{\lambda_i M}\sum_{l=1}^{M} \alpha_l^i \left\langle \phi\left(x^j\right), C_\phi \phi\left(x^l\right) \right\rangle$$

$$= \frac{1}{\lambda_i M^2}\sum_{l=1}^{M} \alpha_l^i \left\langle \phi\left(x^j\right), FF^T \phi\left(x^l\right) \right\rangle$$

$$= \frac{1}{\lambda_i M^2}\sum_{l=1}^{M} \alpha_l^i \left\langle \phi\left(x^j\right), \sum_{j=1}^{M} \phi\left(x^j\right)\left\langle \phi\left(x^j\right), \phi\left(x^l\right) \right\rangle \right\rangle$$

Replacing the latter expression in the definition of the correlation matrix, one gets:

$$\frac{1}{\lambda_i M^2}\sum_{k=1}^{M}\left\langle \phi\left(x^j\right), \phi\left(x^k\right)\sum_{l=1}^{M} \alpha_l^i \left\langle \phi\left(x^k\right), \phi\left(x^l\right) \right\rangle \right\rangle = \frac{1}{M}\sum_{l=1}^{M} \alpha_l^i \left\langle \phi\left(x^j\right), \phi\left(x^l\right) \right\rangle \tag{5.12}$$

Using the kernel trick, one can define the Gram Matrix K, whose elements are composed of the dot product between each pair of datapoints projected in feature space, i.e. $K_{ij} = \left\langle \phi\left(x^i\right), \phi\left(x^j\right)\right\rangle$. Beware that *K* is $M \times M$, where M is the number of data points.

We can finally rewrite the expression given in (5.12) as an eigenvalue problem of the form:

$$\begin{aligned} K^2\alpha^i &= M\lambda_i K\alpha^i, \qquad i = 1...M \\ K\alpha^i &= M\lambda_i\alpha^i \end{aligned} \tag{5.13}$$

This is the dual eigenvalue problem of finding the eigenvectors v^i of C.

The solutions to the dual eigenvalue problem are given by all the eigenvectors $\alpha^1,...,\alpha^M$ with non-zero eigenvalues $\lambda_1,...,\lambda_M$.

Asking that the eigenvectors v of C_ϕ be normalized, i.e. $\left\langle v^i, v^i\right\rangle = 1 \quad \forall i = 1,...,M$ is equivalent to asking that the dual eigenvectors $\alpha^1,...,\alpha^M$ be such that $1/\lambda_i = \left\|\alpha^i\right\|$.

One can now compute the projections of a given query point x onto the eigenvectors v^i using:

$$\left\langle v^i, \phi(x)\right\rangle = \sum_{j=1}^{M} \alpha_j^i \left\langle \phi\left(x^j\right), \phi(x)\right\rangle = \sum_{j=1}^{M} \alpha_j^i k\left(x^j, x\right) \tag{5.14}$$

Note that the solution of kernel PCA yields M eigenvectors, whereas the solution to linear PCA yielded N eigenvectors, where N is the dimensionality of the dataset and M is the number of datapoints. Usually, M is much larger than N, hence kernel PCA corresponds to a lifting into a higher-dimensional space, whereas linear PCA was a projection into a space of lower dimension than the original space. By lifting the data, kernel PCA aims at extracting features that are common to subsets of datapoints. In some way, this is close to a clustering technique. Datapoints that bear some similarity will be close to one another along some particular projection.If the data points have really no regularity, then they will be distributed homogeneously along all projections.

Figure 5-3 illustrates the principle of PCA on a dataset that forms approximatively three clusters. By looking at the regions with equal projection value on the first or second eigenvector, we see that some subgroups of datapoints tend to group in the same region. When the kernel width is large, the two clusters on the right-handside of the figure are encapsulated onto a single contour line in the first projection. The datapoints of the cluster on the far left are closely grouped together. As a result, the contour lines form ellipsoids that match the dispersion of the data. When the datapoints are more loosely grouped, as it is the case for the two groups at the center and far right, one observes some non-linear deformations of the contour lines that reflect the deflections due to the absence of the datapoints. Using a smaller kernel width allows for encapsulating finer features and allows us to separate the groups. A very small kernel width will however lead to one datapoint per projection.

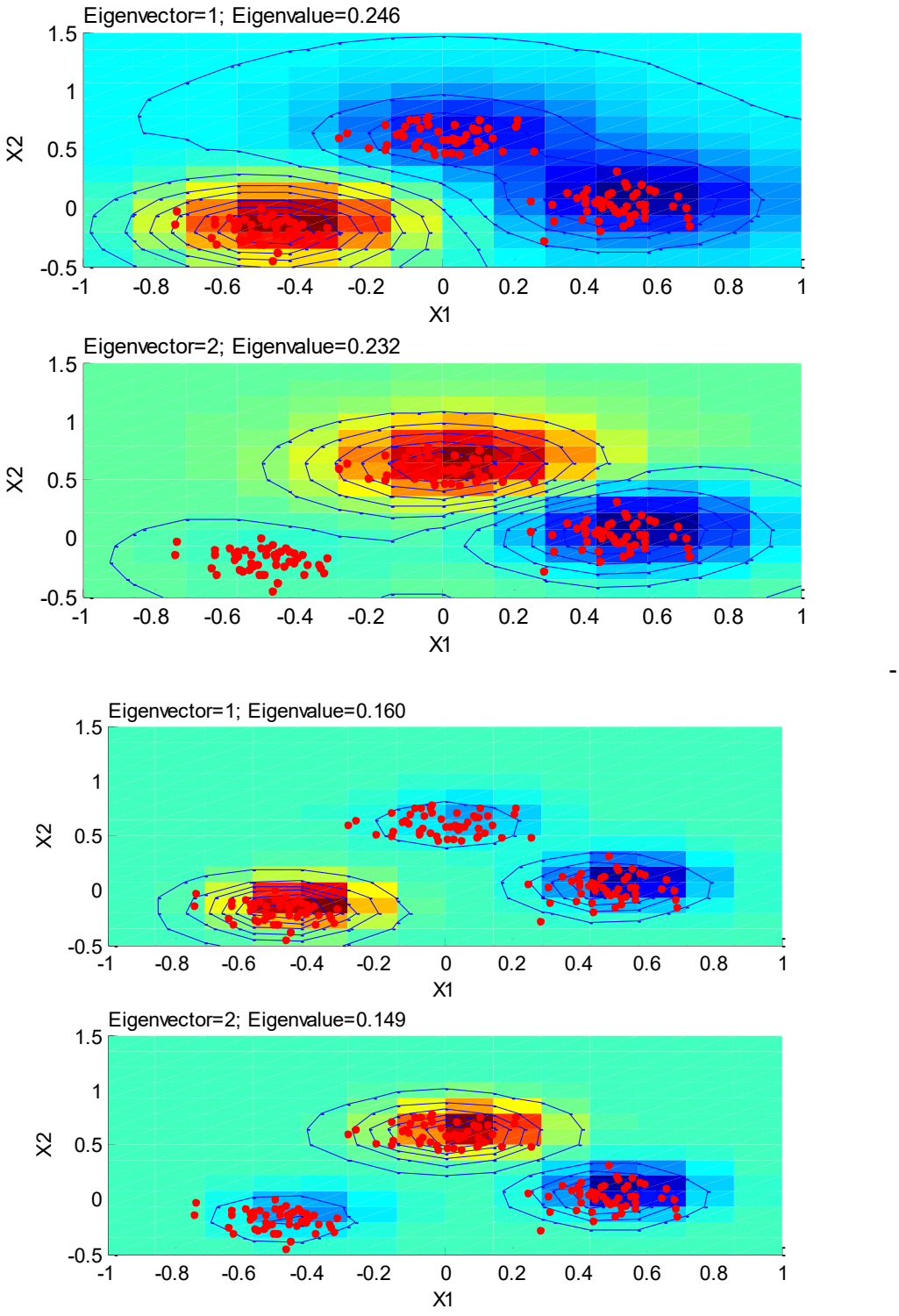


Figure 5-3: Example of clustering done with kernel PCA with a Gaussian Kernel and kernel width $\sigma = 0.1$ (top) and $\sigma = 0.04$ (bottom). Reconstruction in the original dataspace of the projections onto the first two eigenvectors. The contour lines represent regions with equal projection value.

5.4 Kernel CCA

The linear version of CCA was treated in Section 2.3. Here we consider its extension to non-linear projections to find each pair of eigenvectors. We start with a brief recall of CCA.

Consider a pair of zero-mean multivariate datasets $X = \left\{ x^i \in \mathbb{R}^{N_x} \right\}_{i=1}^{M}, Y = \in \left\{ y^i \in \mathbb{R}^{N_y} \right\}_{i=1}^{M}$ of which we measure a sample of *M* instances $\left\{ \left(x^i, y^i \right) \right\}_{i=1}^{M}$. CCA consists in determining a set of projection vectors $w_x \text{ and } w_y$ for *X* and *Y* such that the correlation ρ between the *projections* $z_x = w_x^T X \quad \text{and} \quad z_y = w_y^T Y$ (the *canonical variates*) is maximized:

$$\max \rho = \max_{w_x, w_y} \ \text{corr}\left(\text{z}_x, \text{z}_y \right) == \max_{w_x, w_y} \ \frac{\text{w}_\text{x}^\text{T} E\left\{ XY^T \right\} \text{w}_\text{y}}{\left\| \text{w}_\text{x}^\text{T} X \right\| \left\| \text{w}_\text{y}^T Y \right\|} = \max_{w_x, w_y} \ \frac{\text{w}_\text{x}^\text{T} C_{xy} \text{w}_\text{y}}{\sqrt{\text{w}_\text{x}^\text{T} C_{xx} w_x \text{w}_\text{y}^\text{T} C_{yy} w_y}} \tag{5.15}$$

Where C_{xy}, C_{xx}, C_{xy} are respectively the inter-set and within sets covariance matrices. $C_{xy} \text{ is } N_x \times N_y, C_{xx} = \text{E}\left\{ XX^T \right\} : N_x \times N_x, \ C_{yy} = \text{E}\left\{ YY^T \right\} : \ N_y \times N_y$

Non-linear Case:

Kernel CCA extends this notion to non-linear projections. As for kernel PCA, let us assume that both sets of data have been projected into a feature space through a non-linear map $\phi_x, \ \phi_y$, such that we have now the two sets $\left\{ \phi_x \left(x^i \right) \right\}_{i=1}^{M} \quad \text{and} \quad \left\{ \phi_y \left(y^i \right) \right\}_{i=1}^{M}$. Let us further assume that the data are centered in feature space, i.e. $\sum_{i=1}^{M} \phi_x \left(x^i \right) = 0 \text{ and } \sum_{i=1}^{M} \phi_y \left(y^i \right) = 0$ (if the data are not centered in feature space, one can find a Gram matrix that ensures that these are centered, as done for kernel PCA, see exercise session). Kernel canonical correlation analysis aims at maximizing the correlation between the data in their corresponding projection space. Similarly to kernel PCA, we can construct the two kernel matrices $K_x = F_x^T F_x, \ K_y = F_y^T F_y$, where $F_x \text{ and } F_y$ are two $M \times M$ matrices, whose columns are composed of the projections $\left\{ \phi_x \left(x^i \right) \right\}_{i=1}^{M} \text{ and } \left\{ \phi_y \left(y^i \right) \right\}_{i=1}^{M}$, respectively.

The weights $w_x, \ w_y$ can be expressed as a linear combination of the training examples in feature space, i.e. $w_x = F_x \alpha_x \text{ and } w_y = F_y \alpha_y$. Substituting into the equation for linear CCA yields the following optimization:

In kernel CCA, we solve for:

$$\max_{w_x, w_y} \alpha_x^{\mathrm{T}} F_x^T F_x F_y^T F_y \alpha_y$$

$$\text{u.c.} \quad \alpha_x^{\mathrm{T}} F_x^T F_x F_x^T F_x \alpha_x = \alpha_y^{\mathrm{T}} F_y^T F_y F_y^T F_y \alpha_y = 1$$

Replacing $F_x^T F_x = K_x, \quad F_y^T F_y = K_y,$ we obtain:

$$\max_{\alpha_x, \alpha_y} \rho = \max_{\alpha_x, \alpha_y} \frac{\alpha_x K_x K_y \alpha_y}{\left(\alpha_x K_x^2 \alpha_x\right)^{1/2} \left(\alpha_y K_y^2 \alpha_y\right)^{1/2}} \tag{5.16}$$

Then, as done for linear CCA, see Section 2.3, one can express this optimization as a generalized eigenvalue problem of the form:

$$\begin{pmatrix} 0 & K_x K_y \\ K_y K_x & 0 \end{pmatrix} \begin{pmatrix} \alpha_x \\ \alpha_y \end{pmatrix} = \rho \begin{pmatrix} K_x^2 & 0 \\ 0 & K_y^2 \end{pmatrix} \begin{pmatrix} \alpha_x \\ \alpha_y \end{pmatrix} \tag{5.17}$$

A first difficulty that arises when solving the above problem is the fact that its solution is usually always $\rho = 1$, irrespective of the kernel. This is due to the fact that the intersection between the spaces spanned by the columns of K_x and K_y is usually non-zero[2]. In this case, there exist two vectors α_x and α_y such that $K_x \alpha_x = K_y \alpha_y$, that are solution of (5.17). Finding a solution to (5.17) will hence yield different projections depending on the kernel, but all of these will have maximal correlation. This hence cannot serve as a means to determine which non-linear transformation *K* is most appropriate.

A second difficulty when solving (5.17) is that it requires inverting the Gram matrices. These may not always be invertible, especially as these are not full rank (because of the constraint of zero mean in feature space).

To counter this effect, one may use a regularization parameter (also called ridge parameter) κ on the norm of the transformation yielded by K_x, K_y and end up with the following regularized problem:

$$\underbrace{\begin{pmatrix} 0 & K_x K_y \\ K_y K_x & 0 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} \alpha_x \\ \alpha_y \end{pmatrix}}_{\alpha} = \rho \underbrace{\begin{pmatrix} \left(K_x + \frac{M\kappa}{2}\mathrm{I}\right)^2 & 0 \\ 0 & \left(K_y + \frac{M\kappa}{2}\mathrm{I}\right)^2 \end{pmatrix}}_{B} \begin{pmatrix} \alpha_x \\ \alpha_y \end{pmatrix} \tag{5.18}$$

$$\Leftrightarrow A\alpha = \rho B\alpha$$

[2] K_x and K_y are centered and are hence at most of dimensions $M-1$. The dimension of the intersection between the spaces spanned by the column vectors of these matrices is $\dim\left(C(K_x), C(K_y)\right) \geq \dim\left(C(K_x)\right) + \dim\left(C(K_y)\right) - M$. If *M*>2, i.e. if we have more than two datapoints, then the space spanned by the two canonical basis intersect.

Thanks to the regularization term, when *M* is very large, the right-handside matrix becomes PSD (which is usually the case as the number of datapoints *M>>N,* the dimension of the dataset; if this is not the case, then one makes κ very large). In this case, the matrix *B* can be decomposed into $B = C^T C$. Replacing, this yields a classical eigenvalue problem of the form: $\left(C^T\right)^{-1} A C^{-1} \beta = \rho\beta, \text{ with } \beta = C\alpha.$

Note that the solution to this eigenvalue problem neither shows the geometry of the kernel canonical vectors nor gives an optimal correlation of the variates. On the other hand, the additional ridge parameter induces a beneficial control of over-fitting and enhances the numerical stability of the solutions. (Bach & Jordan, 2002) report that, in many experiments, the solution of this regularized problem show better generalization abilities than the kernel canonical vectors, in the sense of giving higher correlated scores for new objects.

Generalizing to multiple multi-dimensional datasets:

As in linear CCA, one can extend this to comparison across several multidimensional datasets. If $X_1, ..., X_L$ are the *L* datasets, each of which are composed of M observations. Each dataset may however have different dimensions $N_1, ...N_L$ and hence $X_i : N_i \times M$. As in the two-dimensional case, one can construct a set of *L* Gram matrices $K_1, ..., K_L$ and the solution to this kernel CCA problem is given by solving:

$$\begin{pmatrix} 0 & K_1K_2 & ..... & K_1K_L \\ K_2K_1 & 0 & ..... & K_2K_L \\ . & & & \\ . & & & \\ K_LK_1 & K_LK_2 & ..... & K_LK_L \end{pmatrix} \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ . \\ . \\ \alpha_L \end{pmatrix} = \rho \begin{pmatrix} \left(K_1^2 + \frac{M\kappa}{2}\mathrm{I}\right)^2 & & 0 \\ & ..... & \\ 0 & & \left(K_L^2 + \frac{M\kappa}{2}\mathrm{I}\right)^2 \end{pmatrix} \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ . \\ . \\ \alpha_L \end{pmatrix}. \tag{5.19}$$

When using a Gaussian kernel, one can see that as the kernel width increases, the correlation will also increase as an effect of the overlap created across the mapping between the two Gaussian functions.

Further Readings:

For a good overview of kernel CCA, see *M. Kuss and T, Graepel ,The Geometry Of Kernel Canonical Correlation Analysis Tech. Report*

5.5 Kernel ICA

Adapted from Kernel Independent Component Analysis, F. Bach and M.I. Jordan, Journal of Machine Learning Research 3 (2002) 1-48

In Section 2.4, we covered the linear version of Independent Component Analysis. We first here revisit ICA, presenting an alternative approach based on minimization of mutual information. We then extend this approach to non-linear ICA, also known as kernel ICA.

Linear ICA

ICA assumes that the set of M observations $X = \left\{x_j^i\right\}_{j=1,...N}^{i=1...M}$ was generated by a set of *statistically independent* sources $S = \left\{s_1,....s_q\right\},$ with $q \le N$ through a linear transformation A :

$$A : \mathbb{R}^q \to \mathbb{R}^N$$
$$s \to x = As$$
$$\text{where } A\text{: is an unknown } N \times q \text{ mixing matrix.}$$

ICA consists then in estimating both A and S knowing only X .

ICA bears important similarity with Probabilistic PCA, see Section **2.2.5**. The sources S are *latent random* variables, i.e. each source $s_1,...,s_q$ was generated by an independent random process and hence as an associated distribution p_{s_i} . ICA differs from PPCA in that it requires that the sources be statistically independent. PCA in contrast requires solely that the projections be uncorrelated. Statistical independence is a stronger constraint than un-correlatedness, see definitions in Annexes, Sections 9.2.8 and 9.2.9.

To solve for ICA would require doing a multi-dimensional density estimation to estimate each of the densities $p_{s_1},\cdots p_{s_q}$. This is impractical and hence ICA is usually solved by approximation techniques. In Section 2.4, we saw one method to solve ICA using a measure of *non-gaussianity.* The intuition behind this approach was that the distribution of the mixture of independent sources becomes closer to a Gaussian distribution than the distribution of each source independently. This, of course, assumed that the distribution of each source is non-gaussian. This idea is illustrated in Figure 5-4. A measure of non-gaussianity was proposed based on the Negentropy $J(y) = H(y_{Gauss}) - H(y)$. The negentropy measures by how much the entropy $H(y)$ of the current estimate $y \sim s$ of the distribution of the sources differs from the entropy $H(y_{Gauss})$ of a Gaussian distribution with the same mean and covariance as that of the distribution of y . In information theory, the entropy is a measure of the uncertainty attached to the information contained in the observation of a given variable. The more entropy, the more uncertain the event is, see Section **Error! Reference source not found.**. The notion of entropy can be extended to j oint and conditional distributions. When observing two variables, the joint entropy is a measure of the information conveyed by the observation of one variable onto the other variable. It is hence tightly linked to the notion of *information.* Unsurprisingly, ICA can hence also be formulated in terms of mutual information, as we will see next.

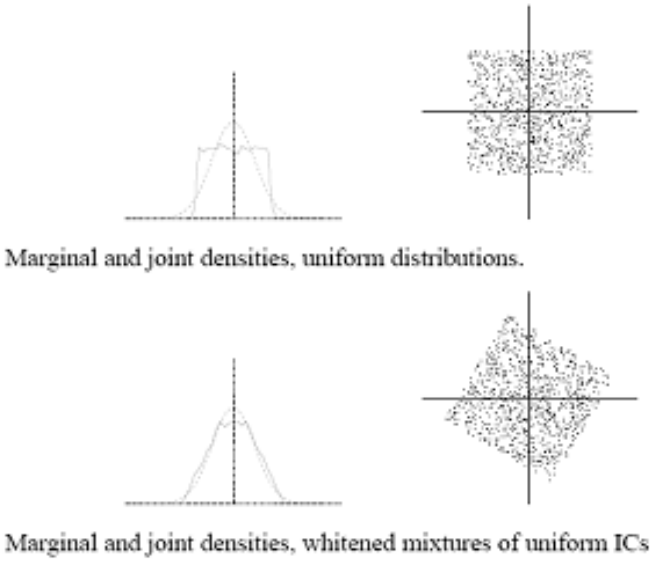


Figure 5-4: TOP: Marginal (left) and joint (right) distributions of two statistically independent sources, a Gaussian distribution and a uniform distribution. BOTTOM: The marginal distributions after whitening are closer to that of a Gaussian distribution.

ICA by minimization of mutual information

ICA searches statistically independent sources. These sources must therefore have minimal mutual information. A measure of the mutual information across the *q* sources is given by:

$$I\left(\mathrm{s}_1,\ldots s_q\right)=\sum_{i=1}^{q}h\left(s_i\right)-h\left(x\right)-\log\left|\det\left(A^{-1}\right)\right| \tag{5.20}$$

where $h\left(x\right)$ is the entropy of the distribution of the observations.

To recall, ICA started with the assumption that the data was centered and white, i.e. $x\sim N\left(0,I\right)$. In practice, this requires to first substract the mean of the data and then to proceed to a decorrelation through PCA, followed by a normalization, see Section **Error! Reference source not found.**. By extension if $x\sim N\left(0,I\right)$ then the sources are also centered and white and hence:

$$I=E\left\{ss^T\right\}=A^{-1}E\left\{xx^T\right\}\left(A^{-1}\right)^T \tag{5.21}$$

Given that $\det\left(I\right)=1$, we have:

$$\begin{aligned}&\det\left(A^{-1}E\left\{xx^T\right\}\left(A^{-1}\right)^T\right)=1\\&\Leftrightarrow\det\left(A^{-1}\right)\det\left(E\left\{xx^T\right\}\right)\det\left(\left(A^{-1}\right)^T\right)=1.\end{aligned} \tag{5.22}$$

This implies that $\det\left(A^{-1}\right)$ is a constant (and is equal to ± 1 in the particular case of whitenend mixtures).

Assuming then that the data is zero-mean and white, that the number of sources equal the dimension of the data, i.e. N, and replacing in (5.20), one can see that maximizing the negentropy is equivalent to minimizing mutual information, as the two quantities differ only by a constant factor, i.e.

$$I\left(\mathrm{s}_1,...s_N\right) = cst. - \sum_{i=1}^{N} J\left(x_i\right). \tag{5.23}$$

Given that the mutual information of a random vector is nonnegative, and zero if and only if the components of the vector are independent, minimization of mutual information for ICA is interesting in that it has a single global minimum. Unfortunately, the mutual information is difficult to approximate and optimize on the basis of a finite sample of datapoints, and much research on ICA has focused on alternative contrast functions. The earliest approach that optimizes for the negentropy using the fast-ICA algorithm, see Section **Error! Reference source not found.**, relies o n an adhoc set of non-linear functions. While not very generic, this approach remains advantageous in that it allows estimating each component separately and in that it does the estimation only on uncorrelated patterns.

Non-linear Case

We will here assume that the sources and the original datapoints have the same dimension and these are zero-mean and white. In practice this can be achieved by performing first PCA and then reducing to the smallest set of eigenvectors with best representation. Hence, given a set of observation vectors $X = \left\{x_i^j\right\}_{i=1,...N}^{j=1,...M}$, we will look for the sources $s_1,....s_N$, such that: $s_i = Wx_i$ and $W = A^{-1}$. As in linear ICA, we do not compute *A* and then its inverse, but rather estimate directly its inverse by computing the columns of *W.*

While the fast-ICA method in linear ICA relied on chosing one or two predefined non-linear functions for the transformation, Kernel ICA solves ICA by considering not just a single non-linear function, but an entire function space $\Phi := \left\{\phi_1(.), \phi_2(.), ...\right\}$ of candidate nonlinearities. The hope is that using a function space makes it possible to adapt to a variety of sources and thus would make the algorithms more robust to varying source distributions. Each function $\phi_1, \phi_2, \cdots$ goes from $\mathbb{R}^N \to \mathbb{R}$. We can hence determine a set of *N* transformations $\phi_i : \left\{x_i^j\right\}^{j=1...M} \to \Phi,\ i = 1,...N$ with associated kernels $K_i = \left\langle \phi_i(.), \phi_i(.) \right\rangle,\ i = 1,...N$. Beware that we are here comparing the projections of the datapoints along the dimensions $i = 1,...N$ of the original dataset, which is thought to be the same dimension as that of the sources. Each matrix K_i is then $M \times M$.

In Section 5.4, when considering kernel CCA, we saw that one can determine the projections that maximize the correlation across the variables using different sets of projections $\phi_1, \phi_2, \ldots$. This is measured by $\rho\left(K_1,....K_N\right)$. This quantity is comprised between zero and one. While CCA aimed at finding the maximal correlation, kernel ICA will try to find the projections that maximize statistical independence. We define a measure of this independence through

$\lambda\left(K_1,....K_N\right)=1-\rho\left(K_1,....K_N\right)$. This quantity is also comprised between 0 and 1 and is equal to 1 if the variables $X_1,...,X_N$ are pairwise independent. Optimizing for independence can be done by maximizing $\lambda\left(K_1,....K_N\right)$. This is equivalent to minimizing the following objective function:

$$J\left(K_1,....K_N\right)=-\log\lambda\left(K_1,....K_N\right) \tag{5.24}$$

This optimization problem is solved in an iterative method. One starts with an initial guess for *W* and then iterates by gradient descent on *J.* Details can be found in Bach & Jordan 2002. Interestingly, an implementation of Kernel-ICA, whose computational complexity is *linear* in the number of data points, is proposed there. This reduces importantly the computational costs.

5.6 Kernel K-Means

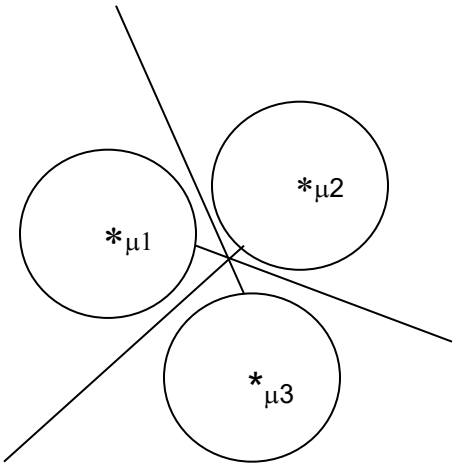
Kernel K-Means is one attempt at using the kernel trick to improve the properties of one of the simplest clustering techiques to date, the so-called K-means clustering technique, see Section 3.2.2.

K-means builds partition the data into a finite set of K clusters $C^k,\ k=1..K,$ (here, do not confuse the scalar K with the Gram matrix seen previously). K-means relies on a measure of distance across datapoint, usually the Euclidean distance. It proceeds iteratively by updating, at each iteration, the centers $\mu_1,....,\mu_K$ of the clusters until no update is required. Given a set of *M* datapoints $X=\{x_i\}_{i=1}^{M}$, the K-means processus consists in minimizing the following objective function:

$$J\left(\mu^1,....,\mu^K\right)=\sum_{k=1}^{K}\sum_{x^j\in C^k}\left\|x^j-\mu^k\right\|^2 \qquad \text{with } \mu_i=\frac{\sum_{x^j\in C^i}x^j}{m_i} \tag{5.25}$$

Where m_i is the number of datapoints in cluster C^i.

Since each cluster relies on a common distance measure, each cluster is separated from the other by a linear hyperplane, as illustrated below:



To counter this disadvantage, kernel K-means first maps the datapoints onto a higher-dimensional feature space through a non-linear map ϕ. It then proceeds as classical K-means and search for hyperplanes in the feature space. To do this, kernel K-means exploits once more the kernel trick and sets the kernel $k\left(x,x'\right)=\left\langle\phi\left(x\right),\phi\left(x'\right)\right\rangle$ as the dot product in feature space. Using the observation that kernel K-means objective function can be expanded into a sum of inner product across datapoints, i.e. $\phi\left(\mu^k\right)=\frac{\sum_{x^j\in C^k}\phi\left(x^j\right)}{m_k}$ yields:

$$J\left(\mu^1,....,\mu^K\right)=\sum_{k=1}^{K}\sum_{x^j\in C^k}\left\|\phi\left(x^j\right)-\phi\left(\mu^k\right)\right\|^2$$

$$=\sum_{i=1}^{K}\sum_{x^j\in C^i}\left(\phi\left(x^j\right)\phi\left(x^j\right)-\frac{2\sum_{x^l\in C^i}\phi\left(x^l\right)\phi\left(x^j\right)}{m_i}+\frac{\sum_{x^j,x^l\in C^i}\phi\left(x^j\right)\phi\left(x^l\right)}{\left(m_i\right)^2}\right) \quad (5.26)$$

$$=\sum_{i=1}^{K}\sum_{x^j\in C^i}\left(k\left(x^i,x^i\right)-\frac{2\sum_{x^j\in C^i}k\left(x^i,x^j\right)}{m_i}+\frac{\sum_{x^j,x^l\in C^i}k\left(x^j,x^l\right)}{\left(m_i\right)^2}\right)$$

In kernel K-Means, the iteration stops when the change in J is sufficiently small. The kernel K-means algorithm reads then as follows:

Kernel K-means algorithm is also an iterative procedure:

1. Initialization: pick K clusters

2. Assignment Step: Assign each data point to its "closest" centroid (E-step).

$$\arg\min_k d\left(x^i,C^k\right)=\arg\min_k\left(\mathrm{k}\left(x^i,x^i\right)-\frac{2\sum_{x^j\in C^k}\mathrm{k}\left(x^i,x^j\right)}{m_k}+\frac{\sum_{x^j,x^l\in C^k}\mathrm{k}\left(x^j,x^l\right)}{\left(m_k\right)^2}\right)$$

3. **Update Step:** Update the list of points belonging to each centroid (M-step)

4. Go back to step 2 and repeat the process until the clusters are stable.

As for all kernel methods, a clear bottleneck of kernel K-Means is that each iteration steps requires $O\left(M^2\right)$ computation step. Smilarly to classical K-means, kernel K-Means is also very sensitive to the initialization of the centers and may lead to poor partiotioning with a poor initialization, as shown in Figure 5-5.

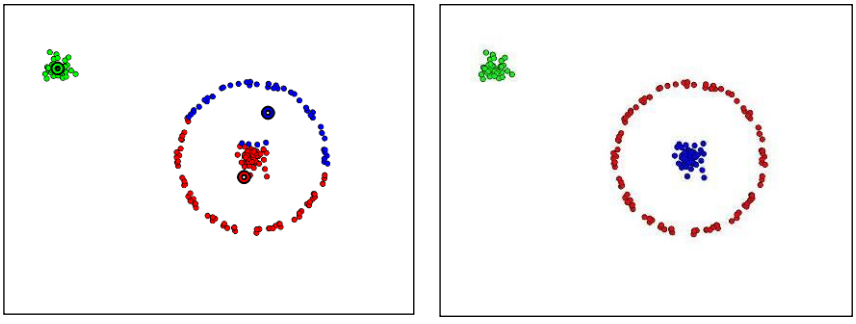


Figure 5-5: A difficult clustering problem. Standard K-Means (left) is unable to correctly separate the three clusters. Using a Gaussian kernel (gamma = 0.01) Kernel K-Means (right) successfully separates the three clusters. [DEMOS\CLUSTERING\KERNEL-KMEANS.ML]

5.7 Support Vector Machines

Adapted from "Learning with Kernels" by B. Scholkopf and A. Smola, MIT Press 2002, and "A Tutorial on Support Vector Machines for Pattern Recognition", by C.J.C. Burges, Data Mining and Knowledge Discovery, 2, 1998.

Support vector machine (SVM) is probably one of the most popular applications of kernel methods. SVM exploits the kernel trick to extend classical linear classification to non-linear classification. It has been shown to be very powerful at separating highly intertwined data. Its simplicity of use and the large number of available software makes it easy to implement. We will here very briefly review the principle and the derivation of the algorithm. We will highlight on the sensivity to some hyperparameter so as to guide the potential user and ensure optimal use of the algorithm.

Linear Case

Let us first start by considering a very simple classification problem which illustrates well the reasoning behind using Kernel methods for classification. Suppose we are given two classes of objects. We are then faced with a new object, and we have to assign it to one of the two classes. This problem can be formalized as follows:

Consider a training set composed of M input-output pairs where each input $\left\{x^i\right\}^{i=1...M} \in \mathbb{R}^N$ is associated with a label $\left\{y^i\right\}^{i=1...M}$. The label y^i denotes the class to which the pattern x^i belongs. In SVM, we consider solely binary classification problems, i.e.:

$$\left\{x^i, y^i\right\}^{i=1...M} \in X \times \left\{\pm 1\right\} \tag{5.27}$$

Note that there exist extensions to *multiclass* SVM. We will here focus first on the binary classification case.

Given this training set, we wish to build a model of the relationships across the input points and their associated class label that would be a good predictor of the class to which each pattern belongs and would allow us to do *inference:* that is, given a new pattern x, we could estimate the class to which this new pattern belongs. In some sense, for a given new pattern x, we would choose a corresponding y, so that the pair $\left\{x, y\right\}$ is somewhat similar to the training examples. To this end,we need a notion of similarity in X and in $\left\{\pm 1\right\}$.

Similarity Measures: Characterizing the similarity of the outputs $\left\{\pm 1\right\}$ is easy. In binary classification, only two situations occur: two labels can either be identical or different. The choice of the similarity measure for the inputs, on the other hand, is more complex and is tighly linked to the idea of kernel. As we have seen previously in these lecture notes, the kernel $k\left(x, x'\right)$ gives a measure of similarity across two datapoints x and x'. A natural choice for the kernel when considering the simple linear classification problem outlined above is to take the dot product, i.e.:

$$k\left(x, x'\right) = \left\langle x, x'\right\rangle = \sum_{i=1}^{N} x_i x'_i \tag{5.28}$$

The geometrical interpretation of the canonical dot product is that it computes the cosine of the angle between the vectors x and x', provided they are normalized with length 1.

A simple pattern recognition algorithm

We now have all the tools to derive a simple classification algorithm.
It is now possible, using the canonical dot product as a similarity measure in our dot product space H to design a simple pattern recognition algorithm that will separate our input space into two classes. The algorithm proceeds as follows:

1. Assign each point randomly to one of the two classes
2. Compute the means of the two classes in the feature space:

$$c_+ = \frac{1}{m_+} \sum_{i|y^i=+1} x^i$$
$$c_- = \frac{1}{m_-} \sum_{i|y^i=-1} x^i \tag{5.29}$$

   Where m_+ and m_- are the number of examples with positive and negative labels, respectively. We assume that both classes are non-empty.

3. Reassign each point to the class whose mean is closest.

This geometric construction can be formulated in terms of the dot product $\langle x, x' \rangle$. Halfway between c_+ and c_- lies the point $c = (c_+ + c_-)/2$. We compute the class to which the point x belongs by checking whether the vector connecting $x - c$ encloses an angle smaller than $\pi/2$ with the vector $w = c_+ - c_-$ connectingthe class means. This leads to:

$$\begin{aligned} y &= \operatorname{sgn}\left(\langle x - c, w \rangle\right) \\ &= \operatorname{sgn}\left(\langle x - (c_+ + c_-)/2, c_+ + c_- \rangle\right) \\ &= \operatorname{sgn}\left(\langle x, c_+ \rangle - \langle x, c_- \rangle + b\right). \end{aligned} \tag{5.30}$$

with offset $b := \frac{1}{2}\left(\|c_-\|^2 - \|c_+\|^2\right)$.
The decision boundary indicated by the dotted line is a hyperplane, orthogonal to w, seeFigure 5-6.

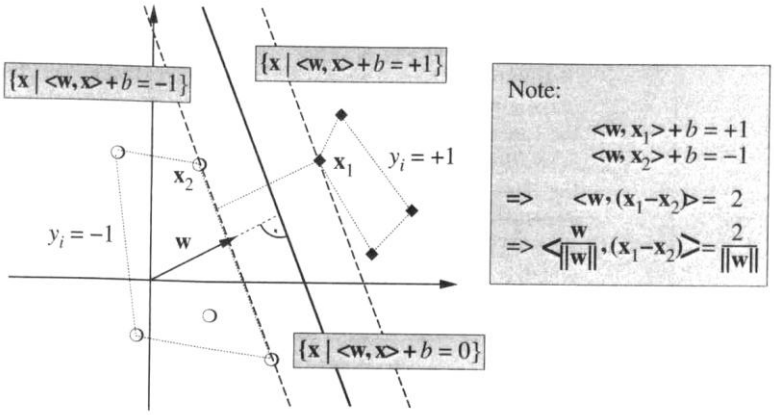


Figure 5-6: A simple geometrical classification algorithm. The decision boundary indicated by the dotted line is a hyperplane orthogonal to w. (From Scholkopf & Smola 2002)

Let us now rewrite this result in terms of the input patterns x^i , using thekernel k(.,.) to compute the dot product. We get the *decision function*:

$$y = \operatorname{sgn}\left(\frac{1}{m_+} \sum_{i=|y^i=+1} \left\langle x, x^i \right\rangle - \frac{1}{m_-} \sum_{i=|y^i=+1} \left\langle x, x^i \right\rangle + b \right) \tag{5.31}$$

$$= \operatorname{sgn}\left(\frac{1}{m_+} \sum_{i=|y^i=+1} k\left(x, x^i\right) - \frac{1}{m_-} \sum_{i=|y^i=+1} k\left(x, x^i\right) + b \right) \tag{5.32}$$

If $b = 0$, i.e. the two classes' means are equidistant to the origin, then *k* can beviewed as a probability density when one of its arguments is fixed. By this, wemean that it is positive and has unit integral,

$$\int_X k\left(x, x'\right) dx = 1 \quad \forall x' \in X$$

In this case, *y* takes the form of the so-called *Bayes classifier* separating the twoclasses, subject to the assumption that the two classes of patterns were generatedby sampling from two probability distributions that are correctly estimated bythe *Parzen Windows* estimator of the two class densities,

$$p_+ := \frac{1}{m_+} \sum_{i|y^i=+1} k\left(x, x^i\right)$$

and

$$p_- := \frac{1}{m_-} \sum_{i|y^i=-1} k\left(x, x^i\right)$$

where $x \in X$.

Thus, given some point x, the class label is computed by checking which of thetwo values p_+ or p_- is larger. This is the best decision one can take if one hasno prior information on the data distribution.

5.7.1 Support Vector Machine for Linearly Separable Datasets

SVM consists of determining a hyperplane that determines a decision boundaryin a binary classification problem. We consider first the linear case and then move to the non-linear case.

Linear Support Vector Machines

Let us assume for a moment that our datapoints Xlive in a feature space *H*. The class of hyperplanesin the dot product space*H* is given by:

$$\langle w, x\rangle + b = 0 \text{, where } w \in H, b \in \mathbb{R} \tag{5.33}$$

with the corresponding decision functions

$$f(x) = sign\left(\langle w, x\rangle + b\right) \tag{5.34}$$

We can now define a learning algorithm for linearly separable problems. First, observe that amongall hyperplanes separating the data, there exists a unique *optimal hyperplane*,distinguished by the maximum *margin* of separation between any training pointand the hyperplane, defined by:

$$\text{maximize}_{w\in H, b\in\Re} \min\left\{\left\|x - x^i\right\|, x \in H, \langle w, x\rangle + b = 0,\ i = 1,...,M\right\} \tag{5.35}$$

While in the simple classification problem we had presented earlier on, it wassufficient to simply compute the distance between the two cluster' means todefine the normal vector and so the hyperplane, here, the problem of findingthe normal vector that leads to the largest margin is slightly more complex.

To construct the optimal hyperplane, we have to solve for the objective function $\tau(w)$:

$$\text{minimize}_{w\in H, b\in\Re} \tau(w) = \frac{1}{2}\|w\|^2 \tag{5.36}$$

subject to the inequality constraints:

$$y^i\left(\langle w, x^i\rangle + b\right) \geq 1, \quad \forall \text{i=1,...M} \tag{5.37}$$

Consider the points for which the equality in (5.37)holds (requiringthat there exists such a point is equivalent to choosing a scale for w and b).These points lie on two hyperplanes $H_1\left(\langle w, x^i\rangle + b = 1\right)$ and $H_2\left(\langle w, x^i\rangle + b = -1\right)$ with normal w and perpendicular distance from the origin $|1-b|/\|w\|$. Hence $d_+ = d_- = 1/\|w\|$ and the margin is simply $2/\|w\|$. Note that H_1 and H_2 areparallel (they have the same normal) and that no training points fall betweenthem. Thus we can find the pair of hyperplanes which gives the maximummargin by minimizing $\|w\|^2$, subject to constraints (5.37) that ensures that the class label for a given x^i will be +1, if $y^i = +1$, and-1,for $y_i = -1$.

Let us now rephrase the minimization under constraint problem given by (5.36) and (5.37) in terms ofthe Lagrange multipliers $\alpha_i,\ i=1,...,l$, one for each of the inequality constraints in (5.37). Recall that the rule is that for constraints of the form $c_i \geq 0$,the constraint equations are multiplied by positive Lagrange multipliers andsubtracted from the objective function, in (5.36), to form the Lagrangian. Forequality constraints, the Lagrange multipliers are unconstrained. This gives theLagrangian:

$$L_P\left(w,b,\alpha\right) \equiv \frac{1}{2}\|w\|^2 - \sum_{i=1}^{l}\alpha_i y^i\left(\left\langle w,x^i\right\rangle + b\right) + \sum_{i=1}^{l}\alpha_i. \tag{5.38}$$

We must now minimize L_P with respect tow, b, and simultaneously requirethat the derivatives of L_P with respect to all the α_i vanish, all subject to theconstraints $\alpha_i \geq 0$. This is a convex quadratic programming problem, since theobjective function is itself convex, and those points which satisfy the constraintsalso form a convex set (any linear constraint defines a convex set, and a set of N simultaneous linear constraints defines the intersection of N convex sets, whichis also a convex set). This means that we can equivalently solve the followingdual problem: maximize L_P , subject to the constraints that the gradient of L_P with respect to w and b vanish, and subject also to the constraints that the $\alpha_i \geq 0$.

Requesting that the gradient of L_P with respect to w and b vanish

$$\frac{\partial}{\partial w_j}L_P = w_j - \sum_i \alpha_i y^i x^i_j, \quad \text{j=1,....N} \tag{5.39}$$

$$\frac{\partial}{\partial b}L_P = -\sum_i \alpha_i y^i = 0 \tag{5.40}$$

gives theconditions:

$$w = \sum_{i=1}^{M}\alpha_i y^i x^i \tag{5.41}$$

$$\sum_{i=1}^{M}\alpha_i y^i = 0. \tag{5.42}$$

Since these are equality constraints in the dual formulation, we can substitutethem into(5.38) to give:

$$L_D\left(\alpha\right) = \sum_i \alpha_i y^i - \frac{1}{2}\sum_{i,j}\alpha_i\alpha_j y^i y^j \left\langle x^i, x^j\right\rangle. \tag{5.43}$$

Note that we have now given the Lagrangian different labels (P for primal,D for dual) to emphasize that the two formulations are different: L_p and L_D arise from the same objective function but with diferent constraints; and thesolution is found either by minimizing L_p or by maximizing L_D . Note also that if weformulate the problem with b=0, which amounts to requiring that all hyper-planes contain the origin (this is a mild restriction for high dimensional spaces,since it amounts to reducing the number of degrees of freedom by one), supportvector training (for the separable, linear case)

then amounts to maximizing L_D with respect to the α_i, subject to constraints (5.42) and positivity of the α_i, withsolution given by (5.41). Notice that there is a Lagrange multiplier α_i for everytraining point. In the solution, those points for which $\alpha_i > 0$ are called the*supportvectors* and lie on one of the hyperplanes H_1, H_2. All other training points have $\alpha_i = 0$, and lie on either side of H_1 or H_2, such that thestrict inequality in(5.37) is satisfied.

For these machines, the support vectors are the critical elements of the training set. They lie closest to the decision boundary; if all other training pointswere removed (or moved around, but so as not to cross H_1 or H_2), and trainingwas repeated, the same separating hyperplane would be found.

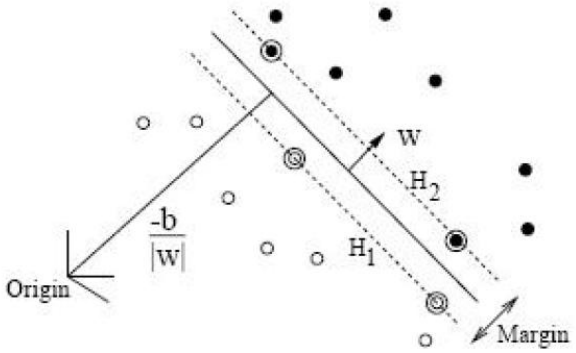


Figure 5-7: A binary classification toy problem: separate dark balls from white balls. The *optimal hyperplane* is shown as the solid line. The problem being separable, there exists a weight vector w and an offset b such that $y^i\left(\left\langle w, x^i\right\rangle + b\right) > 0 \ (i = 1,...,m)$: Rescaling w and b such that the points closest to the hyperplane satisfy $\left|\left\langle w, x^i\right\rangle + b\right| = 1$, we obtain a canonical form (w, b) of the hyperplane, satisfying $y^i\left(\left\langle w, x^i\right\rangle + b\right) \geq 1$. Note that, in this case, the *margin* (the distance to the closest point to the hyperplane) equals $1/\|w\|$. This can be seen by considering two points x^1, x^2 on opposite sides of the margin, that is, $\left\langle w, x^1\right\rangle + b = 1$, $\left\langle w, x^2\right\rangle + b = -1$, and projecting them onto the hyperplane normal vector $w/\|w\|$. The support vectors are the points on the margin (encircled in the drawing).

The generic minimization problem when one does not assume that the hyperplanegoes through the origin (i.e. $b \neq 0$) can be found by solving the so-called KKT(Karush-Kuhn-Tucker) conditions. These state that, at each point of the solution, the product between the dual variables and constraints has to vanish, i.e.:

$$y^i\left(\left\langle w, x^i\right\rangle + b\right) - 1 \geq 0, \quad i = 1,..,M \tag{5.44}$$

$$\alpha_i \geq 0, \quad \forall i = 1,...M \tag{5.45}$$

$$\alpha_i \left(y^i \left(\left\langle w, x^i \right\rangle + b \right) - 1 \right) = 0, \quad \forall i = 1, \ldots M \tag{5.46}$$

The KKT conditions are *necessary* for w, b, α to be a solution.

With all of the above, we now can determine the variables of the SVM problem. w is explicitly determined by(5.41). The threshold b is determinedby solving the KKT "complementarity" condition given by(5.46): by choosing any ifor which $\alpha_i \neq 0$, one can computeb (note that it is numerically safer to take themean value of b resulting from all such equations).

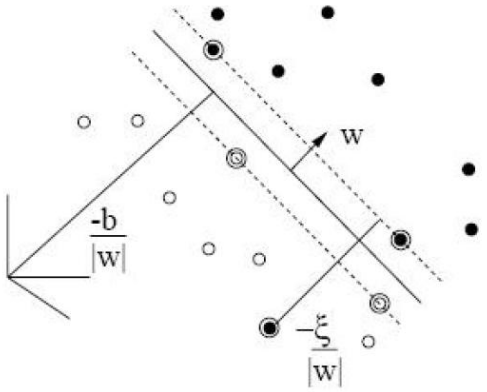


Figure 5-8:Linear separating hyperplanes for the non-separable case.

5.7.2 Support Vector Machine for Non-linearly Separable Datasets

The above algorithm for separable data, when applied to non-separable data,will find no feasible solution: this will be evidenced by the objective function (i.e.the dual Lagrangian) growing arbitrarily large. In order to handle non-separabledata, one must relax the constraints(5.37). This can be done by introducing positiveslack variables $\xi_i, i = 1, \ldots, M$, in the constraints, which then become:

$$w^T x^i + b \geq +1 - \xi_i \quad \text{for } y_i = +1 \tag{5.47}$$

$$w^T x^i + b \leq -1 + \xi_i \quad \text{for } y_i = -1 \tag{5.48}$$

$$\xi_i \geq 0 \ \forall i \tag{5.49}$$

Thus, for an error to occur the corresponding ξ_i must exceed unity, so $\sum_i \xi_i$ is an upper bound on the number of training errors. Hence a natural way toassign an extra cost for errors is to change the objective function to be minimizedto include a cost function, such as

$$\min_{w,\xi} \left(\|w\|^2 + \frac{C}{M} \sum_{i=1}^{M} \xi_i \right),$$

Where *C* is a parameter to be chosen by the user, a larger *C* corresponding toassigning a higher penalty to errors. As it stands, this is a convex programmingproblem and the Wolfe dual problembecomes:

$$\max_{\alpha} L_D(\alpha) \equiv \sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y^i y^j \left\langle x^i, x^j \right\rangle \tag{5.50}$$

subject to:

$$0 \le \alpha_i \le \frac{C}{M} \tag{5.51}$$

$$\sum_i \alpha_i y^i = 0 \tag{5.52}$$

The solution is again given by:

$$w = \sum_{i=1}^{N_s} \alpha_i y^i x^i \tag{5.53}$$

where N_s is the number of support vectors (to recall, the support vectors areall the points x_i for which the corresponding Lagrange multiplier $\alpha_i > 0$; thesepoints lie exactly on the margin). Thus the only diference from the optimalhyperplane case is that the α_i nowhave an upper bound with $\frac{C}{M}$.

Once again, we must use the KKT conditions to solve the primal minimization of L_P and the KKT complementary conditions to find b.

5.7.3 Non-Linear Support Vector Machines

Let us now consider an extension of the linear type of classifier we consideredbefore to tackle non-linear classification problem, such as the one highlighted inFigure 5-9.

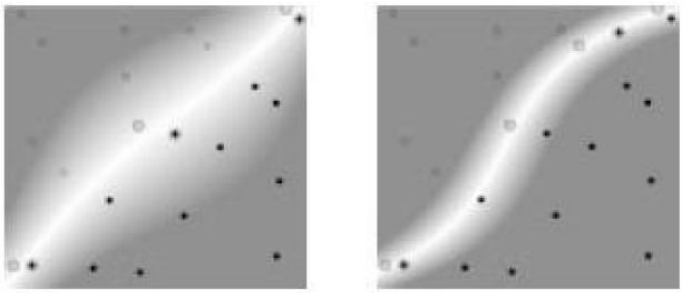


Figure 5-9: Degree 3 polynomial kernel. The background colour shows the shape of the decision surface.

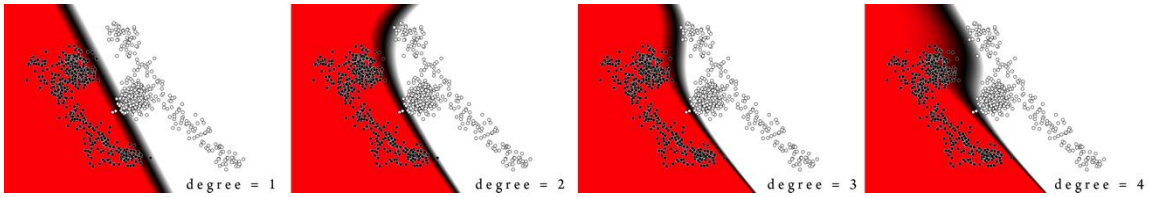


Figure 5-10: Classification using a polynomial kernel with different degrees (SVM). The data is not linearly separable (left). By increasing the degree of the polynomial, the separation plane becomes non-linear and is able to correctly separate the data.[DEMOS\CLASSIFICATION\SVM-POLY.ML]

Following the same rational as presented earlier on, let us first map the dataonto an Euclidean space H, using a mapping ϕ :

$$\begin{aligned} \phi : X &\mapsto H \\ x &\mapsto \phi(x) \end{aligned} \tag{5.54}$$

Assume that the training problem is in the form of dot products $\left\langle x^i, x^j \right\rangle = \left(x^i\right)^T x^j$. Inthis case, the training algorithm in the mapped space would only depend on thedata through dot products in*H*, i.e. on functions of the form $\left\langle \phi\left(x^i\right), \phi\left(x^j\right) \right\rangle$. Ifwe can define a "kernel function" *k* suchthat $k\left(x^i, x^j\right) = \left\langle \phi\left(x^i\right), \phi\left(x^j\right) \right\rangle$, thentraining depends only on knowing *k* and would not require to know ϕ .

The optimization problem consists then to maximize the following quantity:

$$-\frac{1}{2}\sum_{i,j=1}^{M}\left(\alpha_i - \alpha_i'\right)\left(\alpha_i - \alpha_i\right)k\left(x^i, x^j\right) - \varepsilon\sum_{i=1}^{M}\left(\alpha_i - \alpha_i'\right) + \sum_{i=1}^{M} y^i\left(\alpha_i - \alpha_i'\right) \tag{5.55}$$

subject to $\sum_{i=1}^{l}\left(\alpha_i - \alpha_i'\right) = 0$ and $\alpha_i, \alpha_i' \in [0, A]$, $\varepsilon \in \mathbb{R}$. In this case, the classlabel is computed as follows:

$$y = \operatorname{sgn}\left(\sum_{i}^{M} \alpha_i k\left(x, x^j\right) + b\right) \tag{5.56}$$

Each expansion corresponds to a separating hyperplane in a feature space. Inthis sense, the α_i can be considered a *dual representation* of the hyperplane'snormal vector. A test point isclassified by comparing it to all the trainingpoints with non-zero weight.

5.7.4 Nu-SVM

Chosing the right parameter C may be difficult in practice. nu-SVM is an alternative that optimizes for the best tradeoff between model complexity (the largest margin) and penalty on the error automatically. To this end, it introduces two other parameters ν and ρ . $\nu \geq 0$ is an open parameter while ρ will be optimized for. The objective function becomes:

$$\begin{aligned} &\min_{w,\xi}\left(\|w\|^2 - \nu\rho + \frac{1}{M}\sum_{i=1}^{M}\xi_i\right), \\ &\text{subject to} \quad \mathrm{y}^i\left(\left\langle w, x^i \right\rangle + b\right) \geq \rho - \xi_i \\ &\text{and} \quad \xi_i \geq 0,\ \rho \geq 0. \end{aligned} \tag{5.57}$$

To understand the role of ρ , observe first that when the points are well classified, the margin has now been changed to $2\rho / \|w\|$. The larger ρ the larger the margin. Since points within the margin may be misclassified, one can compute the margin error, i.e. the number of points that are within the margin while misclassified. ν varies the effect of this increase in the margin error while optimizing for a large value of ρ . One can show that:

- ν is an upper bound on the fraction of margin error (i.e. the number of datapoints misclassified in the margin)
- ν is a lower bound on the number of support vectors

5.8 Support Vector Regression

In SVM, we have consider a mapping from the input data X onto a binary output $y = \pm 1$. Support Vector regression (SVR) extends the principle of SVM to allow a mapping f to a real value output:

$$\begin{aligned} f &: X \to \mathbb{R} \\ x &\to y = f(x) \end{aligned} \tag{5.58}$$

SVR starts from the standard linear regression model and applies it in feature space. Assume a projection of the data into a feature space $X \to \phi(X) \in H$. Then, SVR seeks a linear mapping in feature space of the form:

$$f(x) = \langle w, \phi(x) \rangle + b, \quad w \in H,\ b \in \mathbb{R} \tag{5.59}$$

To recall, SVM approximates the classification problem by choosing a subset of data points, *the support vectors*, to support the decision function. This *sparsification* of the training dataset is the key strength of the algorithm. When considering non-separable datasets, SVM introduced a slack variable to give room for imprecise classification. SVR proceeds similarly and tries to find the optimal number of support vector while allowing for imprecise fitting. The allowed imprecision of the fitting through f is measured by a parameter $\varepsilon \geq 0$ and is measured through an ε-loss function:

$$|y - f(x)|_\varepsilon = \max\{0, |y - f(x)| - \varepsilon\}, \tag{5.60}$$

Points with a non-zero ε-loss function lie outside the ε*-insensitive tube* that surrounds the function f, seeFigure 5-11.

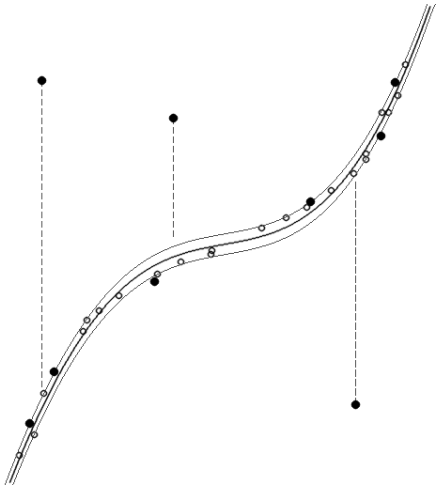


Figure 5-11: Effect of a non-linear regression through SVR. The tightness of the ε*-insensitive tube* around the regression signal varies along the state space as an effect of the distance in feature space between the support vectors (the support vector are plain circles). Datapoints within the ε*-insensitive tube* do not influence the regression model and are indicated with un-filled circles.

In SVM, we found that minimizing $\|w\|$ was interesting as it allowed a better separation of the two classes. Scholkopf and Smola argue that, in SVR, minimizing $\|w\|$ is also interesting albeit for a different geometrical reason. They start by computing the □-margin given by:

$$m_\varepsilon(f) := \inf\left\{\|\phi(x)-\phi(x')\|,\ \forall x,x',\ \text{s.t. } \|f(x)-f(x')\| \geq 2\varepsilon\right\} \tag{5.61}$$

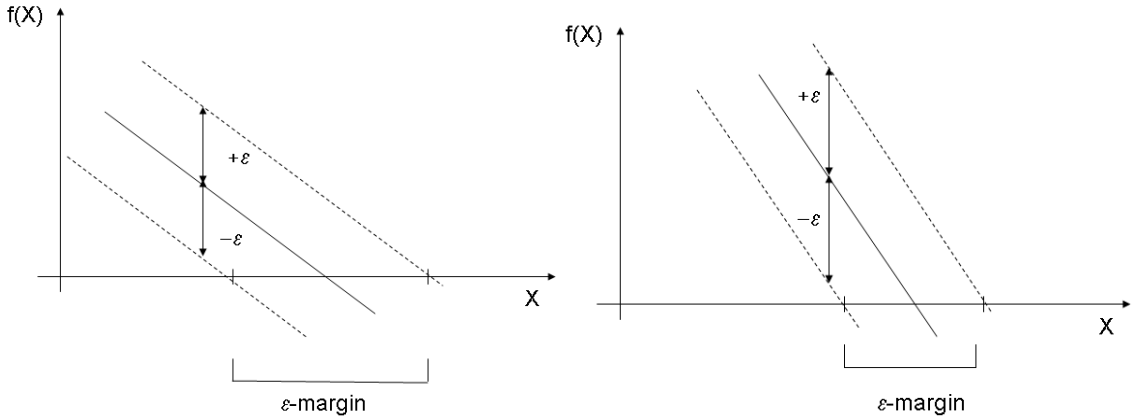


Figure 5-12: Minimizing the ε -margin in SVR is equivalent to maximizing the slope of the function f.

Similarly to the margin in SVM, the □-margin is a function of the projection vector w. As illustrated inFigure 5-12, the flatter the slope w of the function *f,* the larger the margin. Conversely, the steeper the slope is, the larger the width of the □-insensitive tube. Hence to maximize the margin, we must minimize $|w|$ (minimizing each projection of w will flatten the slope). The linear illustration in Figure 5-12 holds in feature space as we will proceed to a linear fit in feature space.

Finding the optimal estimate of f can now be formulated as an optimization problem of the form:

$$\min_w\left(\frac{1}{2}\|w\|^2 + C\cdot R^\varepsilon[f]\right) \tag{5.62}$$

Where $R^\varepsilon[f]$ is a regularized risk function that gives a measure of the e-insensitive error:

$$R^\varepsilon[f] = \frac{1}{M}\sum_{i=1}^{M}\left|y^i - f\left(x^i\right)\right|_\varepsilon \tag{5.63}$$

C in (5.62)is a constant and hence a free parameter that determines a tradeoff between minimizing the increase in the error and optimizing for the complexity of the fit. This procedure is called ***□-SVR***.

Formalism:

The optimization problem given by(5.62) can be rephrased as an optimization under constraint problem of the form:

$$\begin{aligned} &\text{minimize } \frac{1}{2}\|w\|^2 \\ &\text{subject to } \begin{cases} \left\langle w, \phi\left(x^i\right)\right\rangle + b - y^i \le \varepsilon \\ y^i - \left\langle w, \phi\left(x^i\right)\right\rangle - b \le \varepsilon \end{cases} \qquad \forall i = 1,...M \end{aligned} \tag{5.64}$$

Minimizing for the norm of w, we ensure a) that the optimization problem at stake is convex (hence easier to solve) and b) that the conditions given in (5.64) are easier to satisfy (for a small value of $\|w\|$, the left handside of each condition is more likely to be bounded above by ε).

Note that satisfying the conditions of(5.64)may be difficult (or impossible)for an arbitrary small □, one can once more introduce the set of slack variables ξ_i and ξ_i^* $(i = 1...M)$ for each datapoint, to denote whether the datapoint is on the left or righthandside of the function f [3]. The slack variables measure by how much each datapoint is incorrectly fitted. This yields the following optimization problem:

$$\begin{aligned} &\text{minimize } \frac{1}{2}\|w\|^2 + \frac{\mathrm{C}}{M}\sum_{i=1}^{M}\left(\xi_i + \xi_i^*\right) \\ &\text{subject to } \begin{cases} \left\langle w, \phi\left(x^i\right)\right\rangle + b - y^i \le \varepsilon + \xi_i \\ y^i - \left\langle w, \phi\left(x^i\right)\right\rangle - b \le \varepsilon + \xi_i^* \\ \xi_i \ge 0, \quad \xi_i^* \ge 0 \end{cases} \end{aligned} \tag{5.65}$$

Again, one can solve this quadratic problem through Lagrange. Introducing a set of Lagrange multipliers $\alpha_i, \eta_i \ge 0$ for each of our inequalities constraints and writing the Lagrangian:

$$\begin{aligned} \mathrm{L}\left(w, \xi, \xi^*, b\right) = &\frac{1}{2}\|w\|^2 + \frac{\mathrm{C}}{M}\sum_{i=1}^{M}\left(\xi_i + \xi_i^*\right) - \frac{\mathrm{C}}{M}\sum_{i=1}^{M}\left(\eta_i\xi_i + \eta_i^*\xi_i^*\right) \\ &- \sum_{i=1}^{M}\alpha_i\left(\varepsilon + \xi_i + y^i - \left\langle w, \phi\left(x^i\right)\right\rangle - b\right) \\ &- \sum_{i=1}^{M}\alpha_i^*\left(\varepsilon + \xi_i^* - y^i + \left\langle w, \phi\left(x^i\right)\right\rangle + b\right) \end{aligned} \tag{5.66}$$

[3] We will use the shorthand $\xi_i^{(*)}$ when the notation applies to both ξ_i and ξ_i^*.

Solving for each of the partial derivatives: $\frac{\partial \mathrm{L}}{\partial b}=0;\ \frac{\partial \mathrm{L}}{\partial w}=0;\ \frac{\partial \mathrm{L}}{\partial \xi^{(*)}}=0$

$$\frac{\partial \mathrm{L}}{\partial b}=\sum_{i=1}^{M}\left(\alpha_i-\alpha_i^*\right)=0; \tag{5.67}$$

$$\frac{\partial \mathrm{L}}{\partial w}=w-\sum_{i=1}^{M}\left(\alpha_i^*-\alpha_i\right)\phi\left(x^i\right)=0; \tag{5.68}$$

$$\frac{\partial \mathrm{L}}{\partial \xi^{i(*)}}=\frac{C}{M}-\alpha_i^{(*)}-\eta_i^{(*)}=0 \tag{5.69}$$

Substituting in (5.66) yields the dual optimization under constraint problem:

$$\max_{\alpha,\alpha^*}\begin{cases}-\frac{1}{2}\sum_{i,j=1}^{M}\left(\alpha_i^*-\alpha_i\right)\left(\alpha_j^*-\alpha_j\right)\cdot k\left(x^i,x^j\right)\\ -\varepsilon\sum_{i=1}^{M}\left(\alpha_i^*+\alpha_i\right)+\sum_{i=1}^{M}y^i\left(\alpha_i^*+\alpha_i\right)\end{cases} \tag{5.70}$$

$$\text{subject to } \sum_{i=1}^{M}\left(\alpha_i^*-\alpha_i\right)=0 \text{ and } \alpha_i^*,\alpha_i{}^i\in\left[0,\frac{C}{M}\right]$$

Solving for the above optimization problem will yield solutions for the Lagrange multipliers α^*,α. These can then be used to construct our estimate of the best projection vector w by replacing in(5.68):

$$w=\sum_{i=1}^{M}\left(\alpha_i^*-\alpha_i\right)\phi\left(x^i\right) \tag{5.71}$$

To determine the value of the offset b, one must further solve for the Karush-Kuhn-Tucker (KKT) conditions:

$$\begin{aligned}&\alpha_i\left(\varepsilon+\xi_i+y^i-\left\langle w,\phi\left(x^i\right)\right\rangle-b\right)=0,\\ &\alpha_i^*\left(\varepsilon+\xi_i^*-y^i+\left\langle w,\phi\left(x^i\right)\right\rangle+b\right)=0,\\ &\text{and}\\ &\left(\frac{C}{M}-\alpha_i\right)\xi_i=0,\\ &\left(\frac{C}{M}-\alpha_i^*\right)\xi_i^*=0.\end{aligned} \tag{5.72}$$

The above optimization problem can be solved using interior-point optimization, see (Scholkopf & Smola, 2002).

It is worth spending a little bit of time looking at the geometrical implication of conditions (5.72). The last two conditions show that, when the Lagrange multipliers take value $\alpha_i^{(*)} = \frac{C}{M}$, the corresponding slack variable $\xi_i^{(*)}$ can take arbitrary value and hence the corresponding points can be anywhere, including outside the ε-insensitive tube. Vice-versa, when $\alpha_i^{(*)} \in \left]0, \frac{C}{M}\right[$, we have $\xi_i^{(*)} = 0$. These become the support vectors.

Note that since we never have the projections $\phi\left(x^i\right)$, we cannot compute explicitly w in (5.71). We can however once more exploit the kernel trick to compute f. Using

$$\begin{aligned}\left\langle w, \phi(x)\right\rangle &= \sum_{i=1}^{M}\left(\alpha_i^* - \alpha_i\right)\phi\left(x^i\right)\phi(x)\\ &= \sum_{i=1}^{M}\left(\alpha_i^* - \alpha_i\right)k\left(x^i, x\right),\end{aligned}$$

we can compute an estimate of our regression function through:

$$f(x) = \sum_{i=1}^{M}\left(\alpha_i^* - \alpha_i\right)k\left(x^i, x\right) + b \tag{5.73}$$

Observe that the computation costs for the regression function grow linearly with the number M of datapoints. As for SVM, computation can be reduced importantly if one reduces the number of datapoints for which the elements in the sum are non-zero, the so-called *support vectors.*This is the case when $\left(\alpha_i^* - \alpha_i\right) = 0$. Note that for all points within the ε*-insensitive tube* $\alpha_i^* = \alpha_i = 0$ (so as to satisfy the KKT conditions given in (5.72).

Examples:

Figure 5-13 illustrates the application of SVR on the same dataset (these data were generated using MLdemos). When using a linear kernel with a large □, one can completely encapsulate the datapoints, while still yielding a very poor fit. Using a Gaussian kernel allows to fit better the non-linearities. This is however very sensitive to the choice of penalty given to poor fit through parameter C. A high penalty will result in a smoother fit.

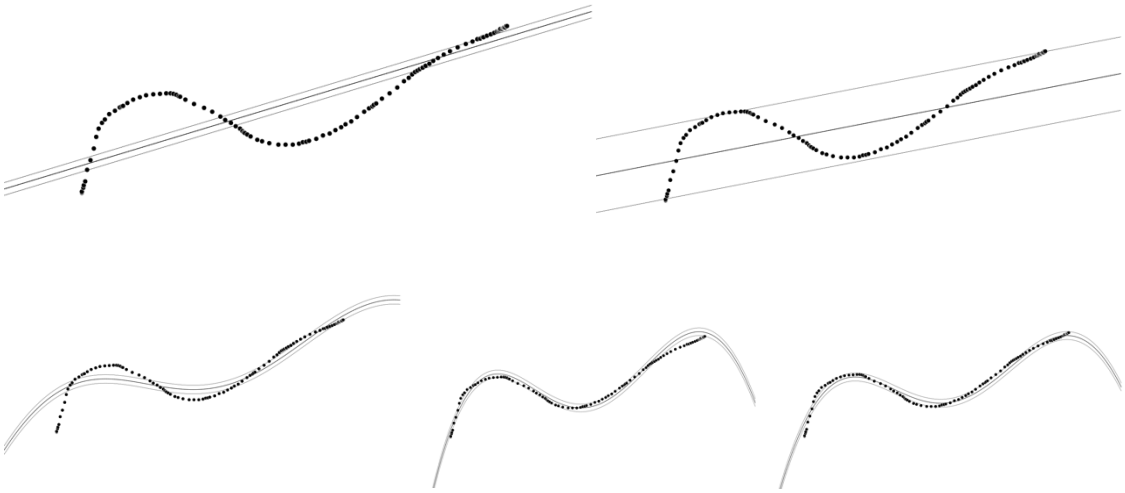


Figure 5-13: Example of □-SVR on a two-dimensional datasets. Datapoints are shown in plain dot. The regression signal is in solid line. The boundaries around the □-insensitite tube are drawn in light grey lines. using a linear kernel (top) and a Gaussian kernel (bottom). TOP: effect of the increasing the □-tube with □=0.02 (left) and □=0.1 (right). BOTTOM: effect of increasing parameter C (from left to right, C=10, 100 and 1000) that penalies for poorly fit datapoints.

SVR as all other regression techniques remain very sensitive also the choice of hyperparameter for the kernel. This is illustrated in Figure 5-14. Too small a kernel width will lead to overfitting of the data. Too large may lead to very poor performance. The latter effect can be compensated by choosing appropriate value for the hyperparameters of the optimization function, namely C and □, see Figure 5-15. The small kernel width that had led to a poor fit in Figure 5-14 is compensated by relaxing the constraints using a large □□and smaller C. We also see that reducing the constraints and widening □ decreases the number of support vectors. While the very tight fit in Figure 5-14 used almost all datapoints as support vector, the looser fit in Figure 5-15 used much less.

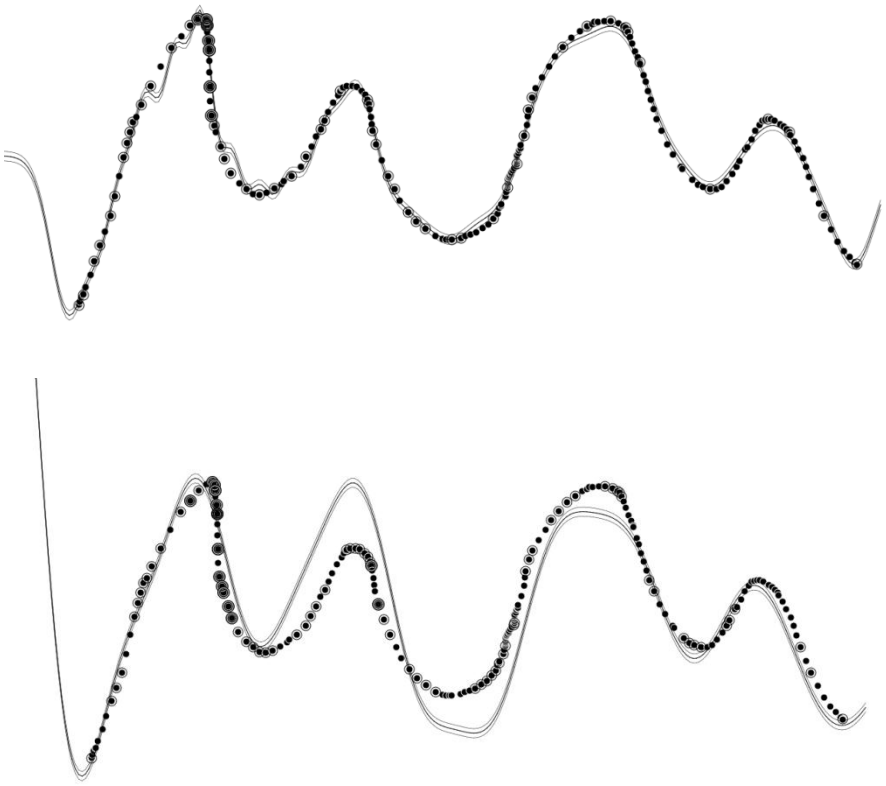


Figure 5-14: Effect of the kernel width on the fit. Here fit using C=1000, □=0.01, kernel width=0.01 (top), 0.1 (bottom). The points encircled represent the support vectors.

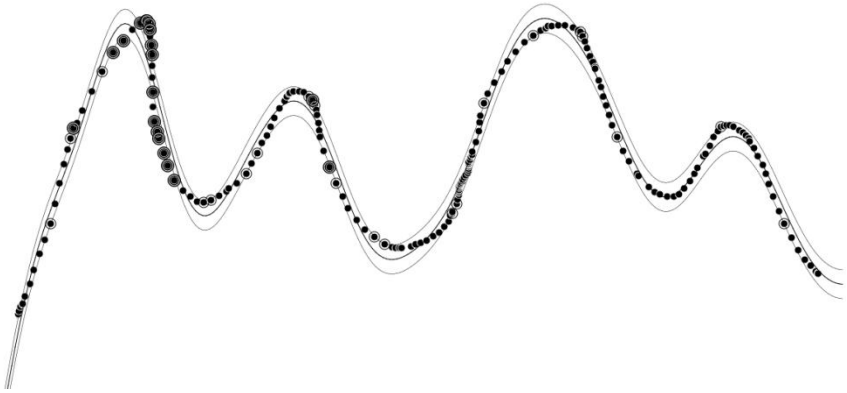


Figure 5-15: Reduction of the effect of the kernel width on the fit by choosing appropriate hyperparameters. Gaussian SVR fit using C=100, □=0.03, kernel width=0.1.

5.8.1 Nu-SVR

The □parameter in □-SVR determines the desired accuracy of the approximation. Determining a good □ may be difficult in practice.□□SVR is an alternative method that suppresses the need topre-specifying the free parameter C in (5.65), by introducing (yet another!) parameter □□□The idea if that this new parameter will make it easier to estimate □as one finds a tradeoff between model complexity and slack variables.

By introducing a penalty through $\nu \geq 0$, the optimization of (5.65) becomes:

$$\begin{aligned} &\text{minimize } \mathrm{L}\left(w, \xi^{(*)}, \varepsilon\right) \frac{1}{2}\|w\|^2 + \mathrm{C}\left(\nu\varepsilon + \frac{1}{M}\sum_{i=1}^{M}\left(\xi_i + \xi_i^*\right)\right) \\ &\text{subject to } \begin{cases} \left\langle w, \phi\left(x^i\right)\right\rangle + b - y^i \leq \varepsilon + \xi_i \\ y^i - \left\langle w, \phi\left(x^i\right)\right\rangle - b \leq \varepsilon + \xi_i^* \\ \xi_i \geq 0, \quad \xi_i^* \geq 0, \varepsilon \geq 0 \end{cases} \end{aligned} \tag{5.74}$$

Notice that we now optimize also for the value of □□□The term $\nu\varepsilon$ in the objective function is now a weighting term that balances a growth of ε and the effect this has on the increase of the poorly fit datapoints (last term of the objective function).

This last new equality constraint on □ yields a new Lagrange solution with associated Lagrange multiplier β . One can then proceed to the same steps as done when solving □-SVM, i.e. write the Lagrangian, take the partial derivatives, write the dual and the solution to KKT conditions (see Scholkopf & Smola 2002 for details). This yields the following □□SVR optimization problem:

$$\text{For } \nu \geq 0, \mathrm{C}>0$$

$$\max_{\alpha^{(*)} \in \mathbb{R}^M} \left(\sum_{i=1}^{M}\left(\alpha_i^* - \alpha_i\right) y^i - \frac{1}{2}\sum_{i,j=1}^{M}\left(\alpha_i^* - \alpha_i\right)\left(\alpha_j^* - \alpha_j\right) k\left(x^i, x^j\right) \right), \tag{5.75}$$

$$\begin{aligned} &\text{subject to } \sum_{i,j=1}^{M}\left(\alpha_i^* - \alpha_i\right) = 0, \\ &\alpha_i^{(*)} \in \left[0, \frac{C}{M}\right], \\ &\sum_{i,j=1}^{M}\left(\alpha_i^* + \alpha_i\right) \leq C\nu. \end{aligned}$$

The regression estimate then takes the same form as before and is expressed as a linear combination of the kernel estimated on each of the data point with non zero $\alpha_i^{(*)}$ (the support vectors), i.e.:

$$f\left(x\right) = \sum_{i,j=1}^{M}\left(\alpha_i^* - \alpha_i\right) k\left(x^i, x\right) + b. \tag{5.76}$$

As we did before solely for, we can now also find b and ε by solving the KKT conditions.

To better understand the effect that □ has on the optimization observe first that if $\nu \geq 1$ then the second and third conditions in (5.75) are redundant and hence all values of □ greater than 1 will have the same effect on the optimization. One should then work solely with values ranging between $0 \leq \nu \leq 1$.

Interestingly, a number of properties can be derived:

- □□is an upper bound on the fraction of error (i.e. the proportion of datapoints with $\xi^{(*)} > 0$),
- □□is a lower bound on the fraction of support vectors.

These two properties are illustrated in Figure 5-17.

In summary, □-SVR is advantageous over □-SVM in that it allows one to automatically adjust the sensitivity of the algorithm through the automatic computation of □. To some extent, this is equivalent to fitting a model of the noise on the data (assuming a uniform noise model). This is illustrated in Figure 5-16.

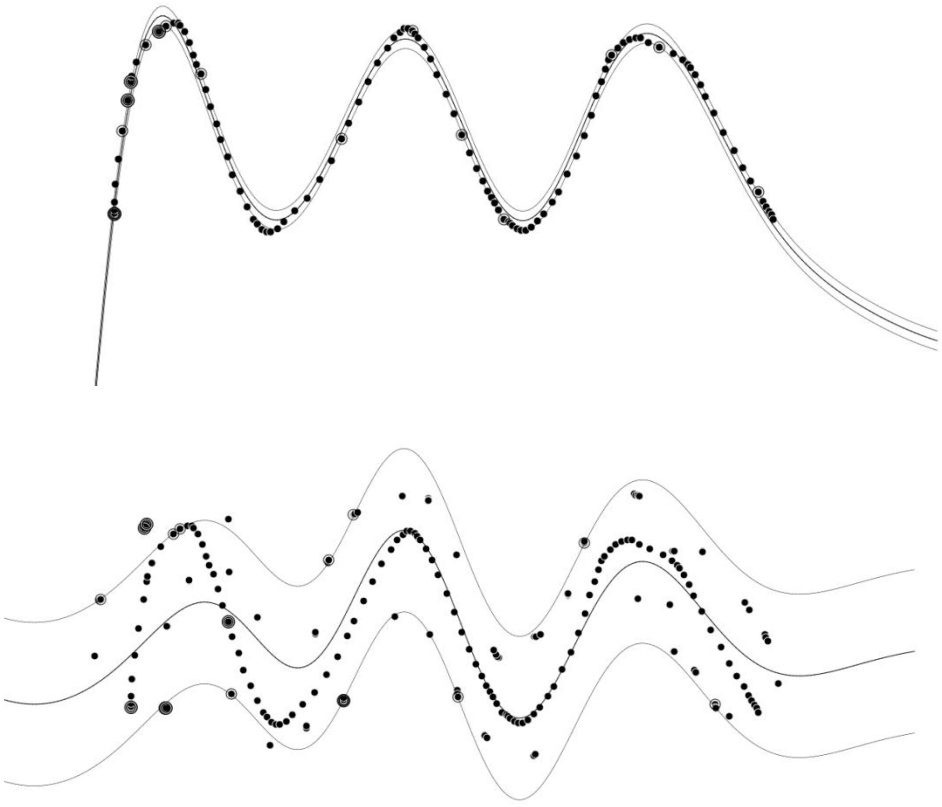


Figure 5-16: Effect of the automatic adaptation of ε using ν-SVR. (Top) Data with no noise. (Bottom) Same dataset with a white noise. In both plots, ν-SVR was fitted with C=100, ν=0.05, and a Gaussian kernel with kernel width=0.021.

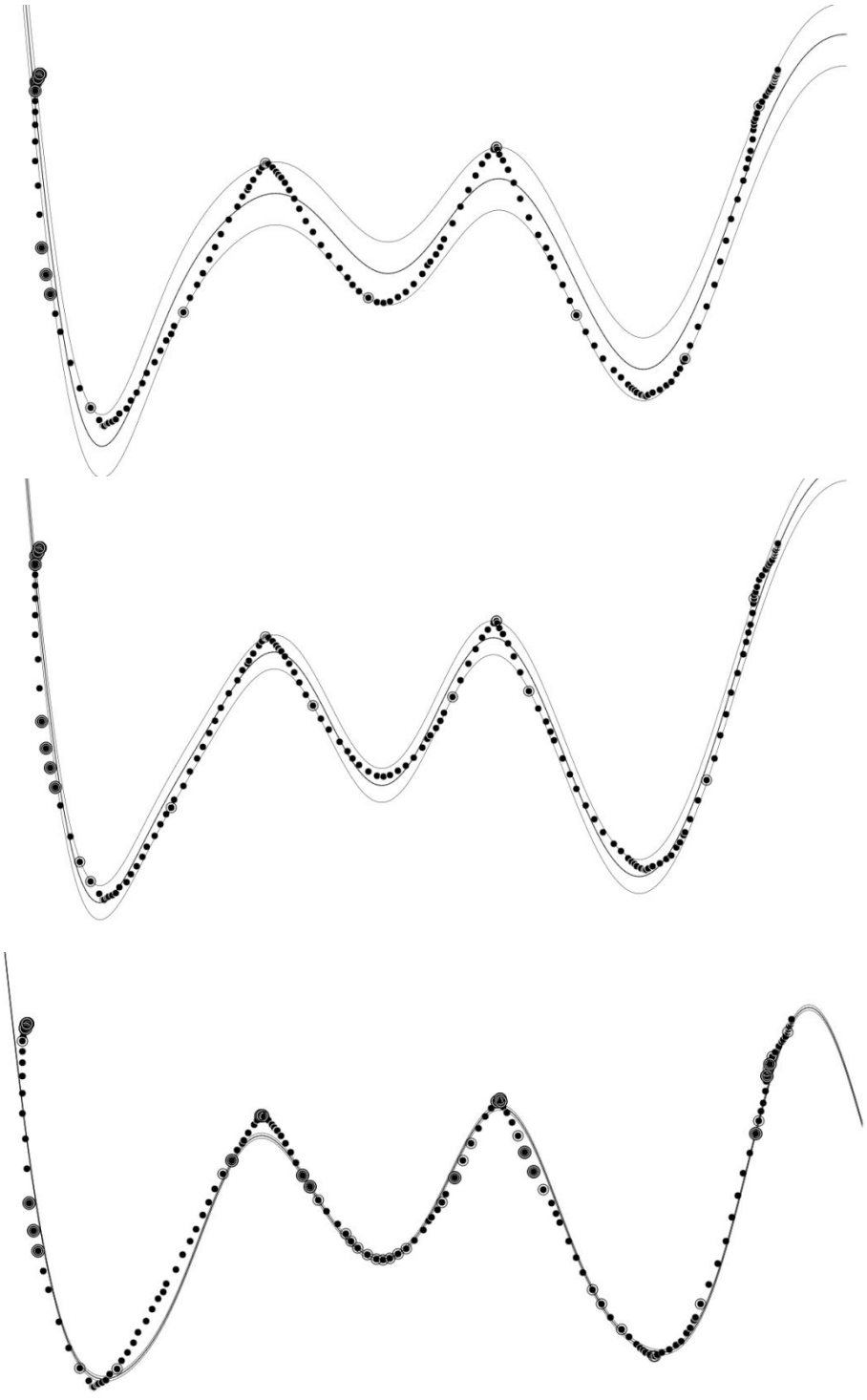


Figure 5-17: Increase of the number of support vectors and of the proportion of datapoints outside the ε-sensitive tube when increasing ν. From top to bottom, νtakes value 0.08, 0.1 and 0.9. ν-SVR was fitted with C=50 and a Gaussian kernel with kernel width=0.021.

5.9 Gaussian Process Regression

Adapted from C. E. Rasmussen & C. K. I. Williams, Gaussian Processes for Machine Learning, the MIT Press, 2006.

In Section **Error! Reference source not found.**, we introduced probabilistic regression, a method b y which the standard linear regressive model $y = w^T x + N\left(0, \sigma^2\right)$ was extended to building a probabilistic estimate of the conditional distribution $p\left(y \mid x\right)$. Then for a new query point *x*,* one could compute an estimate *y** by taking the expectation of y given x, $\hat{y} = E\left\{p\left(y \mid x\right)\right\}$. Further, using the assumption that all training points are i.i.d. and using a Gaussian prior with variance $\sum_w$ on the distribution of the parameters w of the model, we found that the predictive distribution is also Gaussian and is given by:

$$p\left(y^* \mid x^*, X, y\right) = N\left(\frac{1}{\sigma^2} x^{*T} \sum_w Xy, x^{*T} \sum_w x^*\right) \tag{5.77}$$

where X is $N \times M$ and y is $1 \times N$ are the matrix and vector of input-output training points. Next, we see how this probabilistic linear regressive model can be extended to allow non-linear regression, exploiting once more the kernel trick.

Non-linear Case

Assuming a non-linear transformation into feature space through the function $\phi\left(x\right)$, that maps each *N-dimensional* datapoint *x* into a *D-dimensional feature space*, and substituting everywhere in the linear model, the predictive distribution for the non-linear model becomes:

$$\begin{aligned} &p\left(y^* \mid x^*, X, y\right) = N\left(\frac{1}{\sigma^2} \phi\left(x^*\right)^T A^{-1} \Phi\left(X\right) y, \phi\left(x^*\right)^T A^{-1} \phi\left(x^*\right)^*\right) \\ &\text{with } A = \sigma^{-2} \Phi\left(X\right) \Phi\left(X\right)^T + \Sigma_w^{-1} \end{aligned} \tag{5.78}$$

$\Phi\left(X\right)$ is a matrix whose columns are composed of each projection $\phi\left(x\right)$ of each training point $x \in X$. While the expression of this density is quite simple, in practice computing the inverse of the matrix *A* may be very difficult as its dimension is proportional to the dimension of the feature space that may be quite large.

5.9.1 What is a Gaussian Process

The baysian regression model given by (5.78) is one example of Gaussian Process.

In its generic definition, a *"Gaussian process is a collection of random variables, any finite number of which have a joint Gaussian distribution".*

Assume that the real process you wish to describe is regulated by the function $f\left(x\right)$, where x spans the data space. Then, a Gaussian Process (GP) estimate of the function *f* is entirely defined

by its mean $m(x)$ and covariance function $k(x,x')$ (*k* is defined for each pair of point *x, x'* that span the data space):

$$\begin{aligned} m(x) &= \mathrm{E}\{f(x)\} \\ k(x,x') &= \mathrm{E}\{(f(x)-m(x))(f(x)-m(x))\} \end{aligned} \tag{5.79}$$

For simplicity, most GP assume a zero-mean process, i.e. $m(x)=0$. While the above description may be multidimensional, most regression techniques on GP assume that the output $f(x)$ is unidimensional. This is a limitation in the application of the process for regression as it allows making inference solely on a single dimension, say $y=f(x),\ y \in \mathbb{R}$. For multi-dimensional inference, one may run one GP per output variable.

Using (5.79) and (5.78) and assuming zero-mean distribution, the baysian regression model can be rewritten as a Gaussian process defined by:

$$\begin{aligned} E\{f(x)\} &= \phi(x)^T E\{w\} = 0, \\ k(x,x') &= E\{f(x)f(x')\} = \phi(x)^T E\{ww^T\}\phi(x') = \phi(x)^T \Sigma_w \phi(x') \end{aligned} \tag{5.80}$$

We are now endowed with a probabilistic representation of our real process *f*. The value taken by *f* at any given *pair* of input *x,* x' is *jointly* Gaussian with zero mean and covariance given by $k(x,x')$. This means that the estimate of *f* can be drawn only from looking conjointly at the distribution of *f* across two or more input variables. In practice, to visualize the process, one may sample a set X^* of *M** of data points $X^* = \{x^{*i}\}_{i=1}^{M^*}$ and compute *f** a M*-dimensional vector of estimates of *f*, such that:

$$f^* \sim N(0, K(X^*, X^*)) \tag{5.81}$$

Where $K(X^*,X^*)$ is a $M^* \times M^*$ covariance matrx, whose elements are computed using $K(X^*,X^*)_{ij} = k(x^i, x^j),\ \forall i,j = 1....M^*$. Note that if $M^* > D$, i.e. the number of datapoints exceed the dimension of the feature space, the matrix is singular as the rank of *K* is *D.*

Generating such a vector is called drawing from the *prior* distribution of *f* as it uses solely information on the query datapoints themselves and the *prior* assumption that the underlying process if jointly Gaussian and zero mean, as given by (5.81). A better inference can be made if one can make use of prior information in terms of a set of training points. Consider the set $X = \{x^i\}_{i=1}^{M}$ as the training datapoints, one can then express the joint distribution of *the estimates f and f** associated with the training and testing points respectively as:

$$\begin{bmatrix} f \\ f* \end{bmatrix} \sim N\left(0, \begin{bmatrix} K(X,X) & K(X*,X) \\ K(X,X*) & K(X*,X*) \end{bmatrix}\right) \tag{5.82}$$

One can then use the above expression of the joint distribution over *f* and *f** to compute the *posterior* distribution of *f** given the training and testing sets *X*, $X*$ and our prior on $f \sim N\left(0, K(X,X)\right)$ which yields:

$$f*|X*,X,f \sim N(K(X*,X)K(X,X)^{-1} f, \\ K(X*,X*) - K(X*,X)K(X,X)^{-1} K(X,X*)) \tag{5.83}$$

One can then simply sample from this posterior distribution by evaluating the mean and covariance matrix from (5.83) and generating samples as done previously for the prior distribution on *f.*

Figure 5-18 shows three examples of such sampling. In all three plots the shaded area represents the pointwise mean plus and plus/minus the standard deviation for each input value (corresponding to the ~97% confidence region), for the posterior distribution. We used the square exponential covariance function given by $k(x,x') = \exp\left(-\frac{1}{2}|x-x'|^2\right)$. We plot in light grey the area around the regression signal that corresponds to +/- one standard deviation (using the covariance given by (5.83)). This gives a measure of the uncertainty of the inference of the model. Areas with large uncertainty are due to lack of training points covering that space. From left to right, we see the effect of adding one new point (in red) in areas where previously there were no training points on decreasing locally the variance.

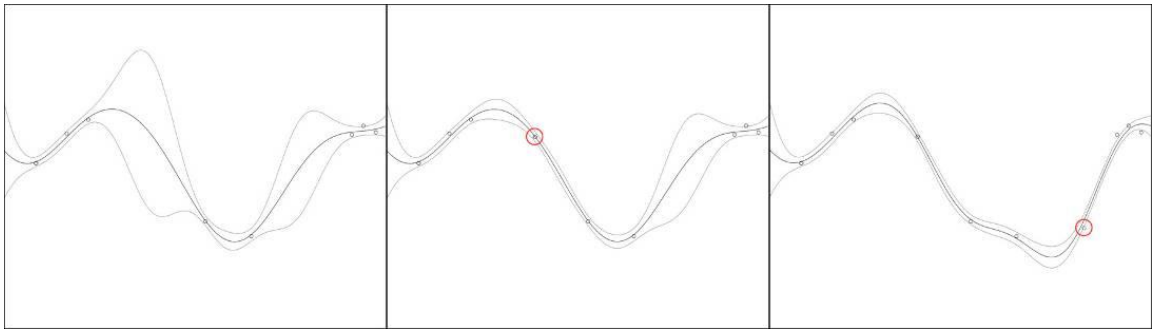


Figure 5-18: The confidence of a Gaussian Process is dependent on the amount of data present in a specific region of space (left). Regions of low data density have lower confidence. By adding points in those regions, the confidence increases (center), but the regression function will change to adapt to the new data (right). [DEMOS\REGRESSION\GPR-CONFIDENCE.ML]

The previous model assumed that the process *f* was noise free. However, when modeling real data, it is usual to assume some noise superposition. As in the case of the linear regression mode, we can assume that the noisy version of our non-linear regression model follows:

$$y = f(x) + \varepsilon$$
$$\varepsilon \sim N(0, \sigma) \tag{5.84}$$

Where the noise ε follows a zero mean Gaussian distribution.

In this case the covariance of the prior becomes: $yy^T = \mathrm{cov}\left(f(X) + \varepsilon\right) = K(X, X) + \sigma^2 I$, where y is a matrix whose columns y^i, i=1….M, correspond to the projection of the associated training point x^i through (5.84).

As done previously, we can express the joint distribution of the prior *y* (now including noise in the estimate of prior on the training datapoints) and the testing points given through *f*:*

$$\begin{bmatrix} y \\ f* \end{bmatrix} \sim N\left(0, \begin{bmatrix} K(X,X) + \sigma^2 \mathrm{I} & K(X*,X) \\ K(X,X*) & K(X*,X*) \end{bmatrix}\right) \tag{5.85}$$

Again, one can compute the conditional distribution of *f** given the pair of training datapoints *X,* the testing datapoints *X** and the noisy prior *y*.

$$\begin{aligned} & f* \mid X*, X, y \sim N\left(\bar{f}*, \mathrm{cov}(f*)\right) \\ & \bar{f}* = E\left\{f* \mid X*, X, y\right\} = K(X*, X)\left[K(X,X) + \sigma^2 I\right]^{-1} y \\ & \mathrm{cov}(f*) = K(X*, X*) - K(X*, X)\left[K(X,X) + \sigma^2 I\right]^{-1} K(X, X*) \end{aligned} \tag{5.86}$$

We are usually interested in computing solely the response of the model to one query point $x*$. In this case, the estimate of the associated output $y*$ is given by the following:

$$\bar{y}* \sim \bar{f}* = E\left\{f* \mid x*, X, y\right\} = k(x*, X)^T \left[K(X,X) + \sigma^2 I\right]^{-1} y \tag{5.87}$$

$k(x*, X)$ is the vector of covariance $k(x*, x^i)$ between the query point and the M training data points x^i, $i = 1...M$.

Since all the training pairs (x^i, y^i), $i = 1...M$ are given, these can be treated as parameters to the system and hence the prediction on $y*$ from Equation (5.87) can be expressed as a linear combination of kernel functions $k(x*, x^i)$:

$$\begin{aligned} &\bar{y}^* \sim \bar{f}^* = E\left\{ f^* \mid x^*, X, y \right\} = \sum_{i=1}^{M} \alpha_i k\left(x^*, x^i \right) \\ &\text{with } \alpha = \left[K\left(X, X \right) + \sigma^2 I \right]^{-1} y \end{aligned} \tag{5.88}$$

We have *M* kernel functions for each of the M training points x^i .

5.9.2 Equivalence of Gaussian Process Regression and Gaussian Mixture Regression

The expression found in Equation (5.88) is somewhat very similar to that found for Gaussian Mixture Regression, see Section 4.4.2 and Equation(4.27). The non-linear term $k\left(x^*, x^i \right)$ is equivalent to the non-linear weights $w_i\left(x^* \right)$ in Equation (4.27), whereas the parameters α_i somewhat correspond to the linear terms stemming from local PCA projections through the Cross-covariance matrices of each Gaussian in GMM (given by $\mu_Y^i + \Sigma_{YX}^i \left(\Sigma_{XX}^i \right)^{-1} \left(x - \mu_X^i \right)$ in Equation (4.27)).

The difference between GPR and GMR lies primarily in the fact that GP uses all the datapoints to do inference, whereas GMM performs some local clustering and uses a much smaller number of points (the centers of the Gaussians) to do inference. However, the two methods may become equivalent under certain conditions.

Assume a normalized Gaussian kernel for the function $k\left(x^*, x^i \right)$ and a noise free model, i.e. $\sigma = 0$. Let us further assume that for a well chosen kernel width, $k\left(x^j, x^i \right)$ is non-zero only for data points deemed close to one another according to the metric *k(.,.)* and is (almost) zero for all other pairs. As a result, the matrix K is sparse. We can hence define a partitioning of the datapoints through a set of $m << M$ (not necessarily disjoint) clusters $C^l : \left\{ x^i \in X,\ s.t.\ k\left(x^i, x^l \right) > \delta \right\},\ \ 1 = 1...m$, centered on *m* datapoints $x^l \in X$. $\delta > 0$ is an arbitrary threshold that determines the closenest of the datapoints. How to choose the *m* datapoints is core to the problematic of most clustering techniques and we refer to reader to Chapter 3.2 for a discussion of these techniques.

Rearranging the ordering of the datapoints so that points belonging to each cluster are located on adjacent columns and duplicating each column of datapoints for points that belong to more than one cluster one can create the following block diagonal Gram matrix:

$$K = \begin{bmatrix} \left[K^1 \right] & 0 & ..... & 0 \\ ..... & & & \\ 0 & ..... & & \left[K^m \right] \end{bmatrix} \tag{5.89}$$

where the elements $K_{ij}^l = k\left(x^{l,i}, x^{l,j} \right)$ of the K^l matrix are composed of the kernel function applied on each pair of datapoints $\left(x^{l,i}, x^{l,j} \right)$ belonging to the associated cluster.

Using properties for the inverse of a block diagonal matrix, we obtain a simplified expression for:

$$\alpha \sim \frac{1}{\prod_{i=1}^{m}\left|K^i\right|}\begin{bmatrix}\left[\left(K^1\right)^{-1}\right] & 0 & \ldots\ldots\ldots & 0 \\ \ldots\ldots\ldots\ldots\ldots\ldots & & & \\ 0 & \ldots\ldots\ldots\ldots & & \left[\left(K^m\right)^{-1}\right]\end{bmatrix}\begin{bmatrix}y^1 \\ . \\ . \\ y^m\end{bmatrix} \qquad (5.90)$$

$$\alpha^l = \frac{1}{\prod_{i=1}^{m}\left|K^i\right|}\left[K^l\right]^{-1} y^l, \quad l = 1...m$$

Where y^l is composed of the output value associated to the datapoints X^l.

For each cluster C^l, one can then compute the centroid $\mu^l = \left\{\mu_x^l, \mu_y^l\right\}$ of the cluster and a measure of the dispersion of the datapoints associated to this cluster around the centroid, given by the covariance matrix $\Sigma_{xx}^j = E\left\{\left(X^l - \mu_x^l I\right)\left(X^l - \mu_x^l I\right)^T\right\}$ of the matrix X^l of datapoints associated to the cluster. Further, for each set of datapoints X^l, one can use the associated set of output value y^l nd compute the crosscovariance matrix $\Sigma_{yx}^l = \left(\left(y^l\right)^T - \mu_y^l I\right)\left(X^l - \mu_x^l I\right)^T$.

If we further assume that each of local kernel Matrix is approximatively a measure of the covariance, i.e. $\left[K^l\right]^{-1} \sim \left[\left(X^l - \mu_x^l I\right)^T\left(X^l - \mu_x^l I\right)\right]^{-1}$ and $k\left(x^*, X^l\right)^T \sim \left(X^l - \mu_x^l I\right)^T x^*$.
Replacing in Equation (5.88) yields:

$$\overline{f}^* = \frac{1}{\sum_{i=1}^{m}\left|K^i\right|}\sum_{l=1}^{m}\left[K^l\right]^{-1} y^l k(x^*, X^l) \qquad (5.91)$$

Observe that our prediction is now a non-linear combination of *m* linear projections of the datapoints through $\left[K^l\right]^{-1} y^l$. If the number of cluster *m* is composed of a single datapoint, we obtain a degenerate Gaussian Mixture Model with a unitary covariance matrix for each Gaussian.

Similarly when the clusters are disjoints, the prediction $\overline{f}^*$ can be expressed as the product of the conditional on each cluster separately and the equivalence with GMM is immediate. In the more general case, when the clusters contain an arbitrary number of datapoints and are not disjoints, the full Gaussian Process takes into account influence from all datapoints. In a GMM the interactions across all datapoints are conveyed in part through the weighting of the effect of each Gaussian. Note also that the centers of the Gaussians are chosen so as to best balance the effect of the different datapoints through E-M.

5.9.3 Curse of dimensionality, choice of hyperparameters

The above showed that when considering particular condition on the form of the kernel matrix for GP, GPR and GMR become equivalent. In its generic form, however, GP is not equivalent to GMM and offers a more powerful tool for inference.

Unfortunately, this is done at the expense of being prohibitive in its computation steps. Indeed, Gaussian Process is a very expensive method as inference grows with the number of datapoints with $O\left(M^3\right)$. Advances in the field investigate sparsifying techniques to decrease in a clever manner the number of training points. Unfortunately, most methods are based on some heuristics to determine which points are deemed better than others. As a result the gain of doing a full GP inference with heuristic-driven sparsification over GMM is no longer obvious.

Another drawback lies in the choice of the kernel function and in particular of the kernel width. This is illustrated in Figure 5-19. Too small a kernel width may lead to poor generalization as it allows solely points very close to one another to influence inference. On the other hand too large a kernel width may smooth out local irregularities. In this respect, GMM is a more powerful tool to provide generalization outside areas not covered by the datapoints, while still encapsulating the local non-linearities. I

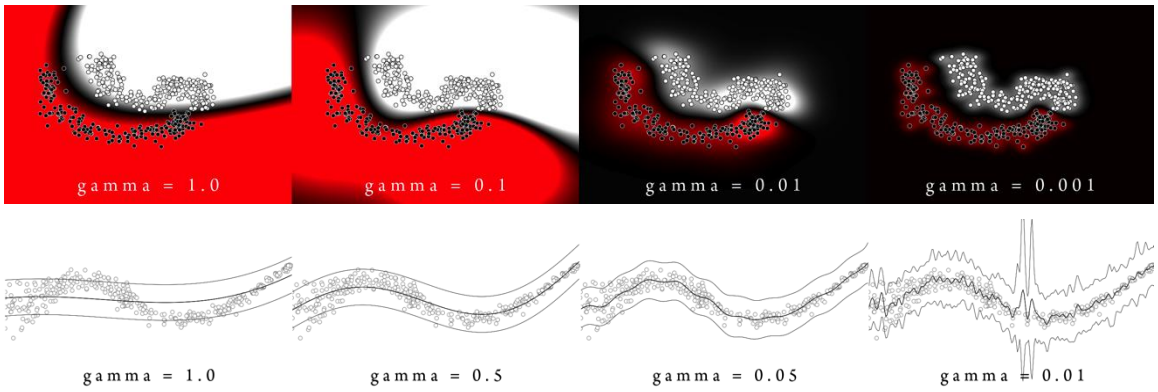


Figure 5-19: Effect of the width of a gaussian kernel on a classification task (SVM, top) and regression (GPR, bottom). When the kernel width (gamma) is very small, the generalization becomes very poor (i.e. the system overfits the data and is unable to estimate correctly on samples that have never been seen). Choosing appropriate parameters for the kernel depends on the data and is one of the main challenges in kernel methods.
[DEMOS\CLASSIFICATION\SVM-KERNEL-WIDTH.ML] [DEMOS\REGRESSION\GPR-KERNEL-WIDTH.ML]

5.10 Gaussian Process Classification

Generative versus Discriminative Approaches

To recall, there are two approaches to classification. One can either follow a so-called *generative approach* whereby one first learns the joint distribution $p(X,Y)$ that associates the set of datapoints $X=\{x^1,....x^M\}$ to a set of labels $Y=\{y^1,...,y^M\}$, where each class label denotes the associated class numbering. Using the joint distribution, one can then compute the conditional distribution $p(Y\mid X)$ to predict the class label for each datapoint. Alternatively, one can follow a *discriminative approach* in which one estimates directly the conditional distribution $p(Y\mid X)$. In the previous section, we showed how to perform regression using a Gaussian Process, it seems hence intuitive to extend this to classification and to this end to take a discriminative approach.

> *Given that there are the generative and discriminative approaches, which one should we prefer? This is perhaps the biggest question in classification, and we do not believe that there is a right answer, as both ways of writing the joint p(y, x) are correct. However, it is possible to identify some strengths and weaknesses of the two approaches. The discriminative approach is appealing in that it is directly modelling what we want, p(y|x). Also, density estimation for the class-conditional distributions is a hard problem, particularly when x is high dimensional, so if we are just interested in classification then the generative approach may mean that we are trying to solve a harder problem than we need to. However, to deal with missing input values, outliers and unlabelled data missing values points in a principled fashion it is very helpful to have access to p(x), and this can be obtained from marginalizing out the class label y from the joint as* $p(x)=\sum_y p(y)p(x\mid y)$ *in the generative approach. A further factor in the choice of a generative or discriminative approach could also be which one is most conducive to the incorporation of any prior information which is available. (Quote from C. E. Rasmussen & C. K. I. Williams, Gaussian Processes for Machine Learning, the MIT Press, 2006.)*

Linear Case

As for all other kernel methods send in this class, we will first start with the linear case and then extend it to the non-linear case by exploiting the kernel trick.

One problem when performing multi-class classification is to compare the predictions of each class. It would be easier if the output of the density could have a direct probabilistic interpretation. In the simply binary classification problem where the label y takes either value +1 or value -1, i.e. $y^i\in[-1;+1]$, $i=1...M$, a probabilistic readout of the the conditional $p(y^i\mid x^i)$ can be obtained easily by using, for instance, the logistic regressive model and compute:

$$p(y^i=+1\mid x^i)=\frac{1}{1+e^{-(x^i)^T w}} \tag{5.92}$$

By extension, the probability of the label -1 is $p(y^i=-1\mid x^i)=1-p(y^i=+1\mid x^i)$.

The weight w determines the slope of the sigmoid function. This is an open parameter which should be chosen so as to ensure optimal classification, or conversely minimal penalty due misclassification. We now have all the tools to proceed to Gaussian Process classification.

As in Gaussian Process Regression, we can start by putting a Gaussian prior on the distribution of the parameter w , i.e. $w \sim N(0, \Sigma_w)$ and compute the log of the posterior distribution on the weight given our dataset $X = \{x^1, ...x^M\}$ with associated labels $Y = \{y^1, ..., y^M\}$:

$$\log p(w \mid X, Y) = -\frac{1}{2} w^T \Sigma_w w + \sum_{i=1}^{M} \log \frac{1}{1 + e^{-(x^i)^T w}} \tag{5.93}$$

Unfortunately the posterior does not have an analytical solution, a contrario to the regression case. One can however observe that the posterior is concave and hence its maximum can be found using classical optimization method, such as Newton's methods or conjugate gradient descent. Figure *Figure 5-20* illustrates nicely how the original weight distribution is tilted as an effect of fitting the data.

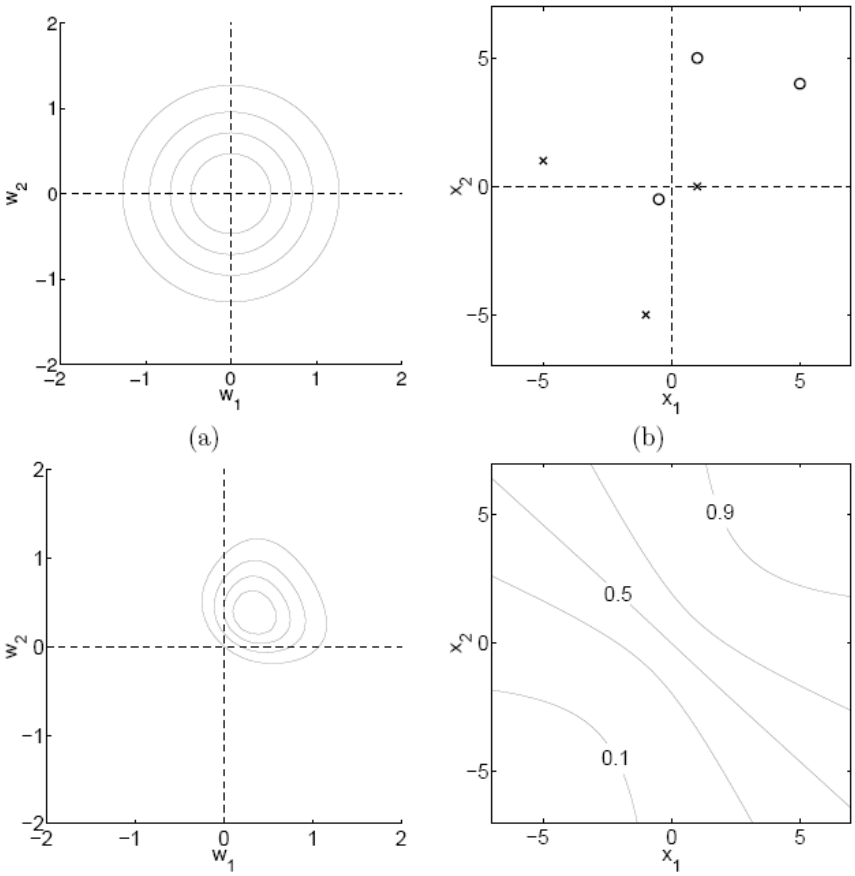


Figure 5-20: a) Contours of the prior distribution p(w) = N(0, I). (b) dataset, with circles indicating class +1 and crosses denoting class −1. (c) contours of the posterior distribution p(w|D). (d) contours of the predictive distribution p(y_=+1|x_), adapted from C. E. Rasmussen & C. K. I. Williams, Gaussian Processes for Machine Learning, the MIT Press, 2006.

Non-linear case

The non-linear case of classification using Gaussian Process proceeds similarly to Gaussian Process Regression. Instead of putting a prior on the weight as in the linear case, we put a prior on the *latent* function $f(x)$, such that we have

$$p(y=+1 \mid x) = \frac{1}{1+e^{-f(x)}} \tag{5.94}$$

This has for effect to ensure that the output is bounded between 0 and +1 and can hence be interpreted as a probability (as in the linear case), see Figure Figure 5-21

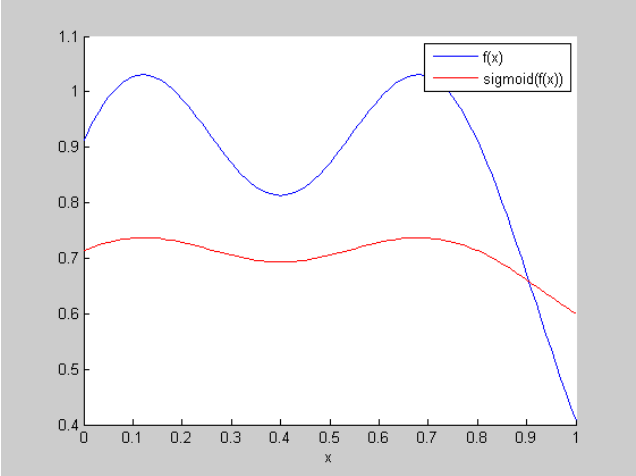


Figure 5-21: Example of an arbitrary prior function f(x) (here composed of the superposition of two Gaussian). Applyinhg the sigmoid function on f(x) flattens the function, while normalizing between 0 and +1.

One now can build an estimate of the class label y^* for a query point x^* by computing the posterior distribution of the function $f(x)$ applied on our query point. If we make this a distribution that is a function of the training datapoints, we have $p\left(f\left(x^*\right) \mid x^*, X, Y\right)$ and the posterior distribution we want to compute is given by:

$$p\left(y^* \mid x^*, X, Y\right) = \int sigmoid\left(f\left(x^*\right)\right) p\left(f\left(x^*\right) \mid x^*, X, Y\right) df \tag{5.95}$$

The integral on the righthandside compute all values on our prior on $f(x)$. While in GPR there was an analytical solution, in the classification case, the integral is usually analytically intractable. To solve this, one must use either an analytic approximations of integrals, or solutions based on Monte Carlo sampling.

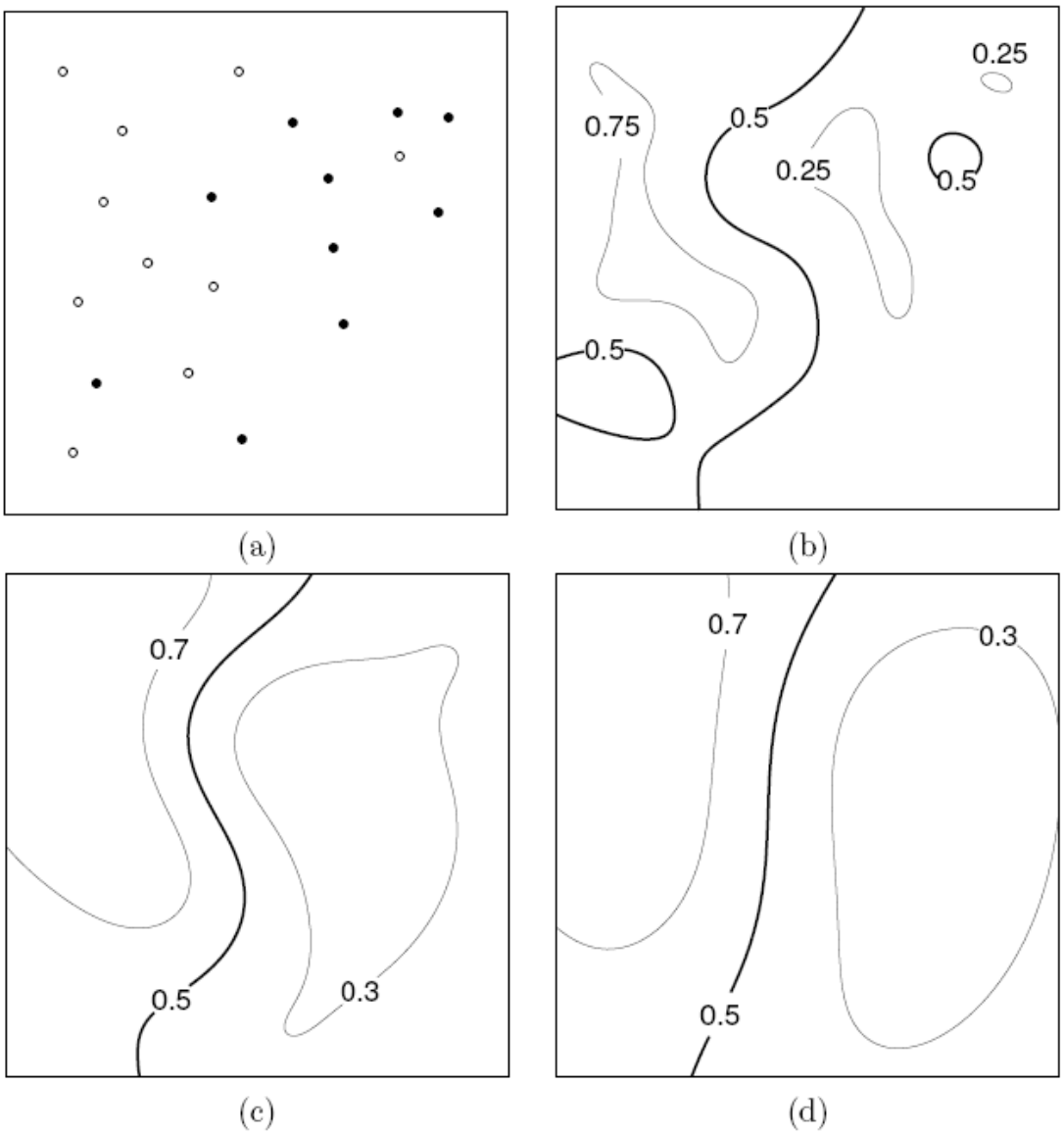


Figure 5-22: Example of successful classification with Gaussian Process classification, adapted *from C. E. Rasmussen & C. K. I. Williams, Gaussian Processes for Machine Learning, the MIT Press, 2006.Panel a* shows the location of the data points in the two-dimensional space [0, 1]. The two classes are labelled as open circles (+1) and closed circles (- 1). Panels (b)-(d) show contour plots of the predictive probability Eq[_(x_)|y] for signal variance f = 9 and length-scales ` of 0.1, 0.2 and 0.3 respectively. The decision boundaries between the two classes are shown by the thicker black lines. The maximum value attained is 0.84, and the minimum is 0.19.

6 Markov-Based Models

Except for ICA, recurrent neural networks and continuous Hopfield network, these lectures notes have covered primarily methods to encode *static* datasets, i.e. data that do not vary in time. Being able to predict patterns that vary in time is fundamental to many applications. Such patterns are often referred to as *time-series.*

In this chapter, we cover two models to encode time-series based on the hypothesis of a first-order Markov Process, which we describe next.

6.1 Markov Process

A *Markov Process* is by definition discrete and assumes that the evolution of a system can be described by a *finite* set of *states.* If x_t is the state of the system at time, then x can take only N particular values $\{s_1,...s_N\}$, i.e. $x \in \{s_1,...s_N\}$.

A *first-order* markov process is a process by which the future state of the system can be determined solely by knowing the *current* state of the system. In other words, if one has made *T* consecutive observation of the state x of the system: $X = \{x_t\}_{t=1}^{T}$, the probability that the state of the system at the next time step x_{T+1} take any of the *N* possible value is determined solely by the current state, i.e.:

$$p\left(x_{t+1} = s_j \mid x_t, x_{t-1}, ..., x_0\right) = p\left(x_{t+1} = s_j \mid x_t\right) \tag{5.96}$$

When modeling time-series, to assume that a process description is first-order markov is advantageous in that it simplifies greatly computation. Instead of computing the complete conditional on all previously observed states $x_t, x_{t-1}, ..., x_0$, one must solely compute the conditional on the last state. We will see how this is exploited in two different techniques widely used in machine learning next.

6.2 Hidden Markov Models

Hidden Markov Models (HMMs) are used to model the temporal evolution of a complex problem of which one can have only a *partial* description. The goal is to use this model to predict the evolution of the system in the future. One may be provided solely with the current state of the system or with several of the previous states of the system.

Take for example the problem of predicting traffic jams. This seems to be a very intricate problem, especially as one can usually only observe part of the problem. One may have video cameras and other type of sensors monitoring the traffic at every single junction. However, one cannot know what individual path each car will follow. While one may suddenly measure an increase in traffic at all points and one may hence infer that there are good chances that a traffic jam may be created. It is very difficult to determine when and on which particular road the traffic jam will happen, if any. The underlying process leading to traffic jam is stochastic and highly complex. HMM-s aim at encapsulating such stochasticity and complexity in a *partially observable* problem.

HMM are used widely in speech recognition and gesture recognition. In speech recognition, they are used to model the temporal evolution of speech pattern and to recognize either complete words or parts of the words, such as phonemes. In gesture recognition, HMM are used to recognize patterns of motions (from either video data or from recording joint motion through e.g. exoskeleton or motion sensors). One usually builds one HMM per gesture and one uses a measure of the likelihood that a new observed gesture was generated by a particular HMM to *recognize* this newspecific gesture (e.g. stop motion, waving motion, pointing motion, etc). These motions can either be computed from searching for body features in an image or through the use of an exoskeleton to measure directly displacement of each body segment.

6.2.1 Formalism

HMM extends the principle of first-order Markov Process and assumes that the observed time series was generated by an underlying *hidden* process. This hidden process is assumed to consist of stochastic finite state automata where the states sequence is not observed directly. Each state has an underlying probabilistic function describing the distribution of observable outputs. Two concurrent stochastic processes are involved, one modeling the sequential structure of the data, and one modeling the local properties of the data. A typical graphical representation of this process is given in Figure 6-1.

Let $O = \{o_t\}_{t=1}^{T}$ be the set of *observations.* These correspond to the set of values for all parameters describing the system which one recorded over a time period *T.* Each observation is usually multidimensional, e.g. $o_t \in \mathbb{R}^q$. The set $S = \{s_i\}_{i=1}^{N}$ of *N* hidden states is finite.

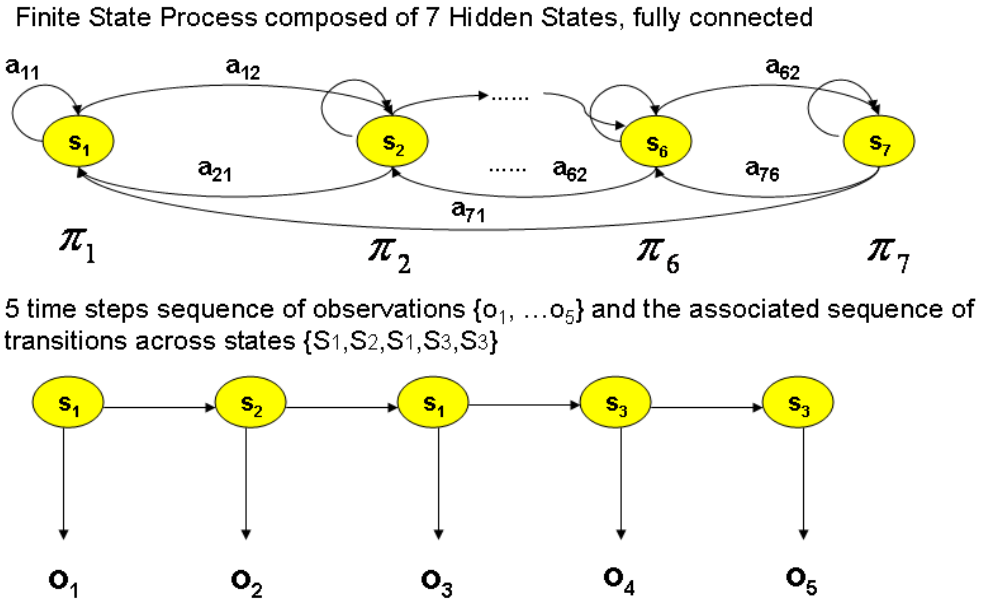


Figure 6-1: Schematic illustrating the concepts of a HMM. The process is assumed to be composed of 7 hidden states. All transitions across all states are possible (fully connected). Note that for simplicity, we show only arrows across adjacent states. The schematic at the bottom shows a particular instance of transition across states over five time steps. Transitiosn across each state leads to an observation of a particular set of values for the system's variables. As we see, the system rotates first across states 1 and 2 and then stays for two time steps on state 3.

Transitions across states are described as a *stochastic* finite automata process. The stochasticity of the process is represented by computing a set of *transition probabilities* that determine the likelihood to stay in a state or to jump to another state. Transition probabilities are encapsulated in a $N \times N$ matrix A, whose elements $\left\{a_{ij}\right\}_{i,j=1...N}$ represent the probability of transiting from state i to state j, i.e. $a_{ij} = p\left(s_j \mid s_i\right)$. The sum of all elements in each row of *A* equals 1. Each state is associated an initial probability π_i, $i = 1,..N$ that represents the likelihood to be in that state at any given point in time. In addition, one assigns to each state i a density $b_i\left(o\right)$, so-called the *emission probability* that determines the probability of the observation to take a particular value when in state S_i. Depending on whether the observables take *discrete* versus *continuous* values, we talk about a discrete versus continuous HMM. When continuous, the density may be estimated through a Gaussian Mixture Model. In this case, one associates a GMM per state. HMM are used widely in speech processing. There, often, one uses very few hidden states (at maximum 3!). The complexity of the speech is then embedded in the GMM density modeling associated at each state.

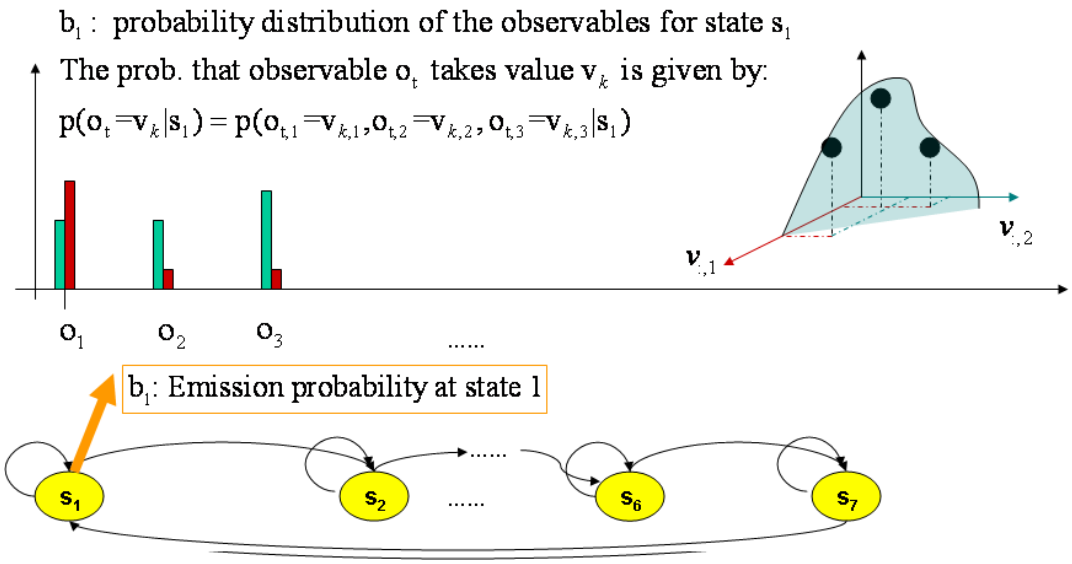


Figure 6-2: Schematic illustrating the concept of emission probabilities associated to each state in a HMM. o_t is the observable at time t. It can take any value $\mathrm{v} \in \mathbb{R}^3$.

6.2.2 Estimating a HMM

Designing an HMM consists in determining the hidden process that explains at best the observations. The unkown variables in a HMM are the number of states, the transitions and initial probabilities and the emission probabilities. Since the matrix *A* is quite large, sometimes, often people chose a sparse matrix, i.e. they set to zero most of the probabilities hence allowing only some transitions across some states. The most popular model is the so-called *left-right* model that allows transitions solely from state 1, to state 2, and so forth until reaching the final state.

Estimating the parameters of the HMM is done through a variant of Expectation-Maximization called the Baum-Welch procedure. For a fixed topology (i.e. number of states), one estimates the set of parameters $\lambda = (A, B, \pi)$ by maximizing the likelihood of the observations given the model:

$$P(O \mid \lambda) = \sum_{q} P(O \mid q, \lambda) P(q \mid \lambda) \tag{5.97}$$

where $q = \{q_1, ..., q_T\}$ is one particular set of expected state transitions during the T observation steps. In the example of Figure 6-1, the set *q* is $\{q_1 = s_1, q_2 = s_2, q_3 = s_1, q_4 = s_3, q_5 = s_3\}$.

Computing all possible combinations of states in Equation (5.97) is prohibitive. To simplify computation, one uses *dynamic* programming through the so-called *forward-backward* computation. The principle is illustrated in Figure 6-3 left. It consists of propagating forward in time the estimate of the probability of being in a particular state given the set of observations. At each time step, the estimate of being in state *i* is given by:

$$\alpha_t(i) = P(o_1 ... o_t, q_t = s_i \mid \lambda) \tag{5.98}$$

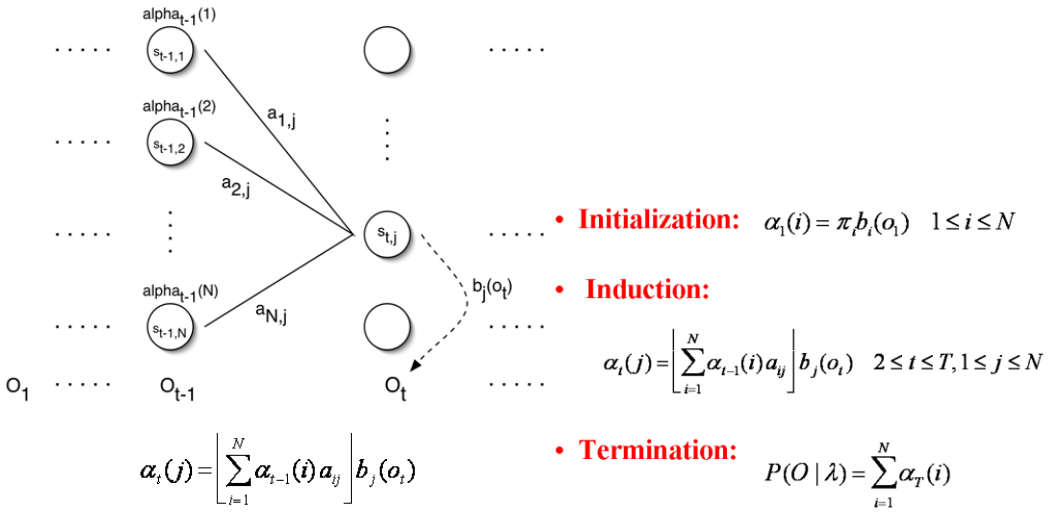


Figure 6-3: The Forward Procedure used in the Baum-Welch algorithm for estimating the HMM parameters

The forward procedure is thus iterative with an initialization step, an induction step (typical of dynamic programming) and a termination step, see Figure 6-3 right.

It is easy to see that the same principle can be extended *backwards* and, thus, that one can infer the probability of being in a particular state by starting from the last state and building one's way back in time. This is referred to as the *backward procedure* and is illustrated in Figure 6-4.

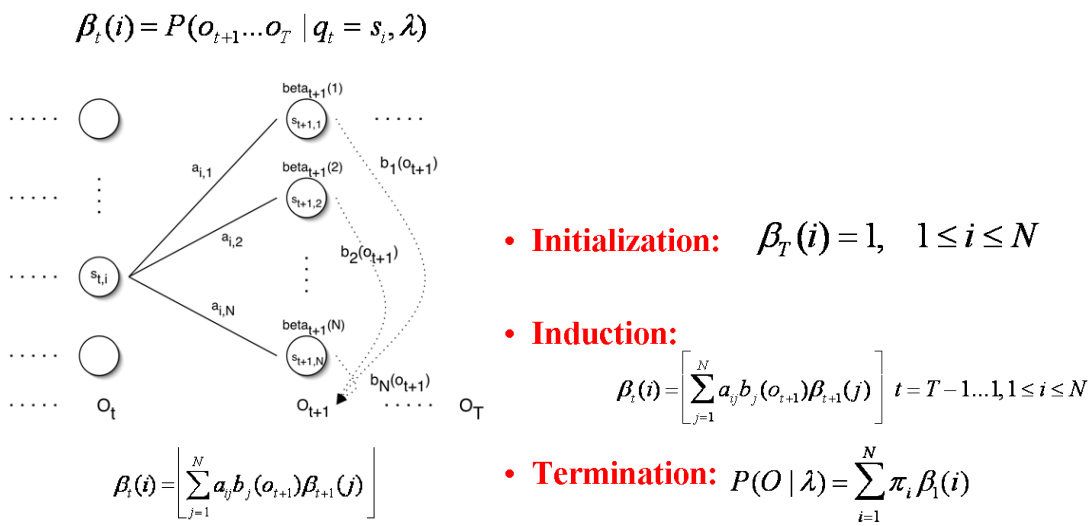


Figure 6-4: The backward procedure in the Baum-Welch algorithm for estimating the HMM parameters

While solving with Equation (5.97) would take on the order of $2T*N^T$ computation steps, the forward or backward procedures reduce this computation greatly and take instead on the order of N^2T computation steps each. One may wonder what the advantage would be to do the computation either forwards or backwards. There is no direct advantage to do either. These two procedures are however crucial to the estimation of the parameters of the HMM. It is easy to see that, if one can compute the probability to be in state *i* at time *t* with the forward procedure and the probability to

be in state *j* at time *t+1* with the backward procedure, then one can combine these two quantities to compute the probability $\xi_t(i,j)$ of transiting from *i* to *j* at time *t.* This is illustrated in Figure 6-5.

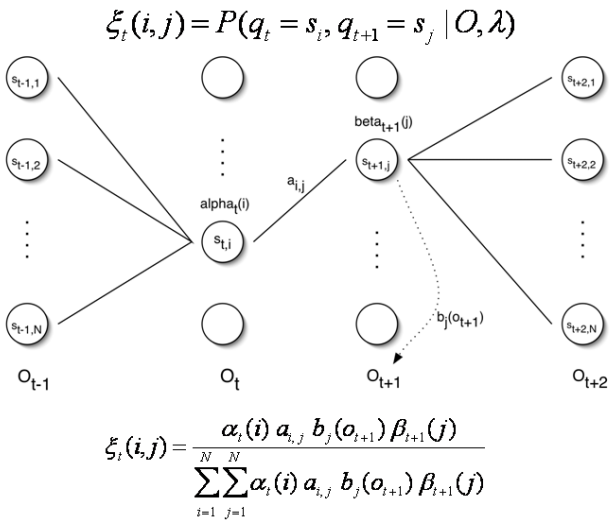


Figure 6-5: Combining the forward and backward procedure to estimate the transition probability across the arc joining state *i* and state *j.*

The probability of being in the state *i at time t* can then be inferred by simply summing over all the transition probabilities and is given by:

$$\gamma_t(i) = \sum_{j=1\ldots N} \xi_t(i,j) \tag{5.99}$$

From the computation of both $\xi_t(i,j)$ and $\gamma_t(i)$ for each state, i.e. $i, j = 1\ldots N$, one can now derive the update rule for all open parameters. HMM re-estimation of the parameter hence starts from an estimate of the parameters $\lambda = (A, B, \pi)$ and updates these iteratively acoording to:

The initial probabilities follow immediately from the computation of the probability to be in any given state at time t=1:

$$\hat{\pi}_i = \gamma_1(i) \tag{5.100}$$

The transition probabilities are then given by computing the sum of probabilities of transiting to this particular state over the course of the whole sequence. This is normalized by the probability of being in any state (the reader will here recognize the Bayes rule, with $\hat{a}_{ij}$ being the conditional probability of being in *j* when being in *j*):

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)} \tag{5.101}$$

Finally, the emission probabilities in the discrete case are given by computing the expected number of times one will be in each particular state and that the observation took a particular value v_k , i.e. $o_t = v_k$:

$$\hat{b}_i(\mathrm{v}_k) = \frac{\sum_{\{t:o_t=\mathrm{v}_k\}} \gamma_t(i)}{\sum_{t=1}^{T} \gamma_t(i)} \tag{5.102}$$

In the continuous case, one builds a continuous estimate of the $\hat{b}_i$, e.g. by building a GMM for each state.

The ^ symbol in each of the above designates the new estimate of the values for each of the HMM parameter.

6.2.3 Determining the number of states

A crucial parameter to HMM (and similarly to GMM) is the number of states *K*. Increasing K increases the number of parameters and so computation. Of course, increasing the number of states allows a finer description of the density and hence increases the likelihood of the model given the data. This increase may however be negligeable in contrast to the increase in computation time due to the increase of parameters. One must then find a tradeoff between increasing the number of parameters and improving the likelihood of the model. This is encapsulated in the Aikaike Information Criterion (AIC):

$$\mathrm{AIC} = -2\ln L + 2K \tag{5.103}$$

where *L* is the likelihood of the model.

This criterion was however lately replaced by the Bayesian Information Criterion that also weights the fact that the estimate of the number of parameters depends on the number of data points. Sparsity in the data may require a lot of parameters to encapsulate non-linearities in these. As the number of datapoints *M* increases, so does the influence of the number of parameters.

$$BIC = -2\ln L + K\ln(M) \tag{5.104}$$

An alternative to the above two criteria is the Deviation Information Criterion (DIC)

$$\mathrm{DIC} = E\{D(K)\} - D(E\{K\}) \qquad \text{with} \qquad D(K) = -2\ln \mathrm{p}(X \mid K) \tag{5.105}$$

Again the smaller the DIC, the better the model fits the data. DIC favors a good fit, while penalizing models by measuring the effective number of parameters required for a similar level of fit.

6.2.4 Decoding an HMM

There are two ways to use a HMM once built. It can be used either for classification or for prediction.

Classification: assume that you have trained two different HMM-s on the basis of two classes of time series (say you trained a HMM to recognize a waving gesture and another HMM to recognize a pointing gesture on the basis of ten examples each). One can use these two HMM-s classify the observation of a new times series (a new movement) to determine whether it belonged more to the first or second type of time series (i.e. whether the person was waving or rather pointing). This can be done by computing the likelihood of the new times series given the model and determining simply which model is most likely to explain this time series. If the new times series is *X'*, the set of parameters of each model are $\lambda^1 = (A^1, B^1, \pi^1)$, $\lambda^2 = (A^2, B^2, \pi^2)$ respectively, then the likelihoods of *X'* to have been generated by each of these two models are $p\left(X' \mid \lambda^1\right)$, $p\left(X' \mid \lambda^2\right)$. These likelihoods can be computed using either the forward or backward procedure.

One can then decide which of model 1 and 2 is most likely to have explained the time series by directly comparing the two likelihoods. That is, model 1 is more likely to have generated *X'* if:

$$p\left(X' \mid \lambda^1\right) > p\left(X' \mid \lambda^2\right) \tag{5.106}$$

Such a comparison is however difficult as the value of the likelihood is unbounded (recall that pdf are positive functions with no upper bound). It can take very large values just as an effect of one model having much more parameters than the other. Hence, in practice, one sometimes normalizes the likelihood by the number of parameters of each model. Since one only compares two values, one has no idea how well any of the two models actually explain the data. It could be that both likelihoods are very low and that hence the observed time series belongs to neither of these two models. In speech processing, people avoid this problem by creating a "garbage" model that ensures that, if a times series is not well represented by any of the meaningful model, then it will be automatically classified in the garbage model. Building a garbage model requires training the model on a large variety of things that cannot be the pattern one is looking for (e.g. noise from cars, doors shutting down, etc). Generating such a large set of non-class samples is not very practical. An alternative is to retain a value of the average likelihood obtained with the training examples for each model, and use this to ensure that the likelihood of the new time series is at least as big as that seen during training (or within some bounds).

Prediction: Once built a HMM can be used to make prediction about the evolution of a given time series if provided with part of the time series. Clearly the farther back in time one can go, the better the prediction.

Say that you have built a HMM to represent the temporal evolution of the weather day after day taking into account the seasons and the particular geographical area, one can then use the model to predict the weather for the following couple of days. The farther in the future the prediction, the less reliable it may be.

To perform prediction one uses the *Viterbi algorithm*. The viterbi algorithm aims at generating the *most likely path*. Given a HMM model with parameters $\lambda = (A, B, \pi)$, one looks for the most likely state sequence $Q = \left\{q_1, \square, q_T\right\}$ for *T* time steps in the future, i.e. one optimizes for:

$$Q = \arg\max_{Q'} P(Q' \mid O, \lambda) \tag{5.107}$$

To do this, one proceeds iteratively and tries to find the optimal state at each time step. Such an iterative procedure is advantageous in that, if one is provided with part of the observations as in

the above weather prediction example, one can use the observations $\{o_1,...,o_t\}$ made over the first *t* time steps to guide the inference.

The optimal state at each time step is obtained by combining inferences made with the forward and backward procedures and is given by $\gamma_t(j) = \frac{\alpha_j(t)\beta_j(t)}{\sum_{i=1}^{N} \alpha_i(t)\beta_i(t)}$, see also Equation(5.99).

The most likely sequence is then obtained by computing:

$$q(t) = \arg\max_{1\le i\le N}\left(\gamma_t(i)\right) \tag{5.108}$$

The state sequence maximizing the probability of a path, which accounts for the first *t* observations and ends in state j is given by:

$$\delta_t(j) = \max_{q_1\cdots q_{t-1}} p(q_1...q_{t-1}, q_t = j, o_1...o_t) \tag{5.109}$$

Computing the above quantity requires taking into account the emission probabilities and the transition probabilites. Again one proceeds iteratively through induction. This forms the core of the Viterbi algorithm and is summarized in the table below:

Viterbi Algorithm

- **Initialization:** $\delta_1(i) = \pi_i b_i(o_1), \quad 1 \le i \le N$
- **Induction:** $\delta_t(j) = \left[\max_{1\le i\le N} \delta_{t-1}(i)\, a_{ij}\right] b_j(o_t)$

  $\psi_t(j) = \left\lfloor \arg\max_{1\le i\le N} \delta_{t-1}(i)\, a_{ij} \right\rfloor \quad 2 \le t \le T, 1 \le j \le N$
- **Termination:** $p^* = \max_{1\le i\le N} \delta_T(i) \qquad q_T^* = \arg\max_{1\le i\le N} \delta_T(i)$
- **Path backtracking:** $q_t^* = \psi_{t+1}(q_{t+1}^*) \quad t = T-1,...,1$

Hence, when inferring the weather over the next five days, given information on the weather for the last ten days, one would first compute the first 10 states sequence $q_1,.....q_{10}$ using (5.109) and then one would use (5.108) to infer the next five states $q_{11},.....q_{15}$. Given $q_{11},.....q_{15}$, one would then draw from the associated emission probabilities to predict the particular weather (i.e. the particular observation one should make) for the next 15 time slots.

:

6.2.5 Further Readings

Rabiner, L.R. (1989) “A tutorial on hidden Markov models and selected applications in speech recognition”, Proceedings of the IEEE, 77:2

Shai Fine, Yoram Singer and Naftali Tishby (1998), “The Hierarchical Hidden Markov Model”, Machine Learning, Volume 32, Number 1, 41-62.

6.3 Reinforcement Learning

Adapted from [Sutton & Barto, 1998; Kaelblin, Littman & Moore, 1996].

Reinforcement learning is the problem faced by an agent that must learn behavior through trial-and-error interactions with a dynamic environment.

Reinforcement learning is learning what to do--how to map situations to actions--so as to maximize a numerical reward signal. The learner is not told which actions to take, as in most forms of machine learning, but instead must discover which actions yield the most reward by trying them.

In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics--trial-and-error search and delayed reward--are the two most important distinguishing features of reinforcement learning.

A good way to understand reinforcement learning is to consider some of the examples and possible applications that have guided its development.

Consider a master chess player making a move. The choice is informed both by planning--anticipating possible replies and counter replies--and by immediate, intuitive judgments of the desirability of particular positions and moves. However, a chess player was not born with this knowledge. The player acquired it. A chess player becomes a master only after numerous defeats. It is through these unsuccessful games that the player manages, through a complex process of deduction, to determine which of the numerous steps were misguided and led to the ultimate defeat.

Consider a baby managing her first steps. Before walking steadily, the baby is likely to have fallen down numerous times. It is as likely that none of the falls were similar to one another. Thus, a simple associative process would not be sufficient to infer the correct set of actions to undertake in order to achieve a steady walking pattern.

In the first example, the *reward* is a binary value, the success of the game, while in the second; the reward is more continuous (duration of the walk before falling, complete fall versus sitting). A challenge in reinforcement learning is to define the reward. It is clear that the goal of the agent is to maximize the reward. However, how can the agent know when the maximum is reached?

Reinforcement learning is different from *supervised learning*. Supervised learning is learning from examples provided by a knowledgeable external supervisor. This is an important kind of learning, but alone it is not adequate for learning from simply interacting with the world, as in the second example above. In interactive problems it is often impractical to obtain examples of desired behavior that are both correct and representative of all the situations in which the agent has to act. In uncharted territory--where one would expect learning to be most beneficial--an agent must be able to learn from its own experience.

One of the challenges that arise in reinforcement learning and not in other kinds of learning is the trade-off between exploration and exploitation. To obtain a lot of reward, a reinforcement learning agent must prefer actions that it has tried in the past and found to be effective in producing reward.

But to discover such actions, it has to try actions that it has not selected before. The agent has to *exploit* what it already knows in order to obtain reward, but it also has to *explore* in order to make better action selections in the future. The dilemma is that neither exploration nor exploitation can be pursued exclusively without failing at the task. The agent must try a variety of actions *and* progressively favor those that appear to be best. On a stochastic task, each action must be tried many times to gain a reliable estimate of its expected reward.

6.3.1 Principle

In reinforcement learning, the purpose or goal of the agent is formalized in terms of a special reward signal passing from the environment to the agent. At each time step, the reward is a simple number, $r_t \in \Re$. Informally, the agent's goal is to maximize the total amount of reward it receives. This means maximizing not immediate reward, but cumulative reward in the long run.

The reinforcement-learning problem is meant to be a straightforward framing of the problem of learning from interaction to achieve a goal. The learner and decision-maker is called the *agent*. The thing it interacts with, comprising everything outside the agent, is called the *environment*. These interact continually, the agent selecting actions and the environment responding to those actions and presenting new situations to the agent. The environment also gives rise to rewards, special numerical values that the agent tries to maximize over time. A complete specification of an environment defines a *task*, one instance of the reinforcement-learning problem.

More specifically, the agent and environment interact at each of a sequence of discrete time steps, $t = 0, 1, 2, 3, \ldots$. At each time step t, the agent receives some representation of the environment's *state*, $s_t \in S$, where S is the set of possible states, and on that basis selects an *action*, $a_t \in A$, where $A(s_t)$ is the set of actions available in state s_t. One time step later, in part as a consequence of its action, the agent receives a numerical *reward*, $r_t \in \Re$, and finds itself in a new state s_{t+1}. Figure 6-6 diagrams the agent-environment interaction.

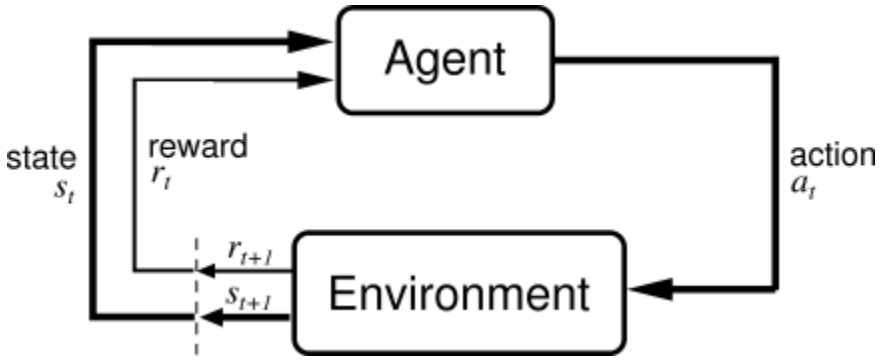


Figure 6-6: The agent-environment interaction in reinforcement learning.

An intuitive way to understand the relation between the agent and its environment is with the following example dialogue.

Environment: You are in state 65. You have 4 possible actions.

Agent: I'll take action 2.

Environment: You received a reinforcement of 7 units. You are now in state 15. You have 2 possible actions.

Agent: I'll take action 1.

Environment: You received a reinforcement of -4 units. You are now in state 65. You have 4 possible actions.

Agent: I'll take action 2.

Environment: You received a reinforcement of 5 units. You are now in state 44. You have 5 possible actions.

⋮ ⋮

6.3.2 Defining the Reward

If the sequence of rewards received after time step *t* is denoted $r_{t+1}, r_{t+2}, r_{t+3}, \ldots$, then what precise aspect of this sequence do we wish to maximize? In general, we seek to maximize the *expected return*, where the return, R_t, is defined as some specific function of the reward sequence. In the simplest case the return is the sum of the rewards:

$$R = r_{t+1} + r_{t+2} + r_{t+3} + \ldots + r_T \tag{5.110}$$

where *T* is a final time step. This approach makes sense in applications in which there is a natural notion of final time step, that is, when the agent-environment interaction breaks naturally into subsequences, i.e. *episodes*, such as plays of a game, trips through a maze, or any sort of repeated interactions. Each episode ends in a special state called the *terminal state*, followed by a reset to a standard starting state or to a sample from a standard distribution of starting states. Tasks with episodes of this kind are called *episodic tasks*.

On the other hand, in many cases the agent-environment interaction does not break naturally into identifiable episodes, but goes on continually without limit. For example, this would be the natural way to formulate a continual process-control task, or an application to a robot with a long life span. We call these *continuing tasks*. The return formulation is problematic for continuing tasks because the final time step would be $T = \infty$, and the return, which is what we are trying to maximize, could itself easily be infinite. (For example, suppose the agent receives a reward of +1 at each time step.)

The additional concept that we need is that of *discounting*. According to this approach, the agent tries to select actions so that the sum of the discounted rewards it receives over the future is maximized. In particular, it chooses a_t to maximize the expected *discounted return*:

$$R = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{5.111}$$

where γ is a parameter, $0 \le \gamma \le 1$, called the *discount rate*.

The discount rate determines the present value of future rewards: a reward received *k* time steps in the future is worth only γ^{k+1} times what it would be worth if it were received immediately. If $\gamma < 1$

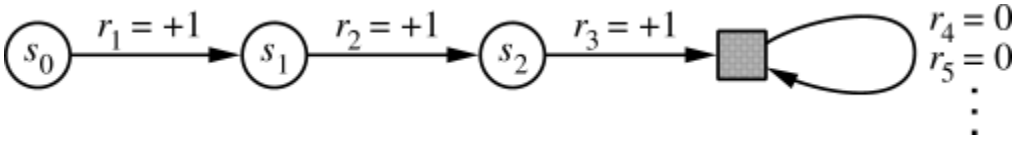
, the infinite sum has a finite value as long as the reward sequence $\{r_k\}$ is bounded. If $\gamma = 0$, the agent is "myopic" in being concerned only with maximizing immediate rewards: its objective in this case is to learn how to choose a_t so as to maximize only r_{t+1}. If each of the agent's actions happened to influence only the immediate reward, not future rewards as well, then a myopic agent could maximize by separately maximizing each immediate reward. But in general, acting to maximize immediate reward can reduce access to future rewards so that the return may actually be reduced. As γ approaches 1, the objective takes future rewards into account more strongly: the agent becomes more farsighted. We can interpret γ in several ways. It can be seen as an interest rate, a probability of living another step, or as a mathematical trick to bound the infinite sum.

Another optimality criterion is the *average-reward model*, in which the agent is supposed to take actions that optimize its long-run average reward:

$$\lim_{h\to\infty} E\left(\frac{1}{h}\sum_{t=0}^{h} r_t\right) \tag{5.112}$$

Such a policy is referred to as a *gain optimal* policy; it can be seen as the limiting case of the infinite-horizon discounted model as the discount factor approaches One problem with this criterion is that there is no way to distinguish between two policies, one of which gains a large amount of reward in the initial phases and the other of which does not. Reward gained on any initial prefix of the agent's life is overshadowed by the long-run average performance. It is possible to generalize this model so that it takes into account both the long run average and the amount of initial reward than can be gained. In the generalized, *bias optimal* model, a policy is preferred if it maximizes the long-run average and ties are broken by the initial extra reward.

6.3.3 Markov World



A reinforcement-learning task is described as a *Markov decision process*, or *MDP*. If the state and action spaces are finite, then it is called a *finite Markov decision process (finite MDP)*.

A particular finite MDP is defined by its state and action sets and by the one-step dynamics of the environment. Given any state and action, *s* and *a*, the probability of each possible next state, *s'*, is

$$P^a_{ss'} = \mathrm{P}\{s_{t+1} = s' \mid s_t = s, a_t = a\} \tag{5.113}$$

These quantities are called *transition probabilities*. Similarly, given any current state and action, s and a, together with any next state, s', the expected value of the next reward is

$$R^a_{ss'} = E\{r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\} \tag{5.114}$$

These quantities, $P^a_{ss'}$ and $R^a_{ss'}$, completely specify the most important aspects of the dynamics of a finite MDP (only information about the distribution of rewards around the expected value is lost).

Almost all reinforcement learning algorithms are based on estimating *value functions*--functions of states (or of state-action pairs) that estimate *how good* it is for the agent to be in a given state (or how good it is to perform a given action in a given state). The notion of "how good" here is defined in terms of future rewards that can be expected, or, to be precise, in terms of expected return. Of course the rewards the agent can expect to receive in the future depend on what actions it will take. Accordingly, value functions are defined with respect to particular policies.

A policy, π, is a mapping from each state, $s \in S$, and action, $a \in A(s)$, to the probability $\pi(s,a)$ of taking action *a* when in state *s*. Informally, the *value* of a state *s* under a policy π, denoted $V^\pi(s)$, is the expected return when starting in *s* and following π thereafter. For MDPs, we can define $V^\pi(s)$ formally as:

$$V^\pi(s) = E_\pi\{R_t \mid s_t = s\} = E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right\} \tag{5.115}$$

where E_π denotes the expected value given that the agent follows policy π, and *t* is any time step. Note that the value of the terminal state, if any, is always zero. We call the function V^π the *state-value function for policy* π.

Similarly, we define the value of taking action *a* in state *s* under a policy π, denoted $Q^\pi(s,a)$, as the expected return starting from *s*, taking the action *a*, and thereafter following policy π:

$$Q^\pi(s,a) = E_\pi\{R_t \mid s_t = s, a_t = a\} = E_\pi\left\{\sum_{k=0}^{\infty} \gamma^t r_{t+k+1} \mid s_t = s, a_t = a\right\} \tag{5.116}$$

We call Q^π the *action-value function for policy* π. The value functions V^π and Q^π can be estimated from experience.

6.3.4 Estimating the policy

We have defined optimal value functions and optimal policies. Clearly, an agent that learns an optimal policy has done very well, but in practice this rarely happens. For the kinds of tasks in which we are interested, optimal policies can be generated only with extreme computational cost. Even if we have a complete and accurate model of the environment's dynamics, it is usually not possible to simply compute an optimal policy. For example, board games such as chess are a tiny fraction of human experience, yet large, custom-designed computers still cannot compute the optimal moves. A critical aspect of the problem facing the agent is always the computational power available to it, in particular, the amount of computation it can perform in a single time step.

The memory available is also an important constraint. A large amount of memory is often required to build up approximations of value functions, policies, and models. In tasks with small, finite state sets, it is possible to form these approximations using arrays or tables with one entry for each state (or state-action pair). This, we call the *tabular* case, and the corresponding methods we call tabular

methods. In many cases of practical interest, however, there are far more states than could possibly be entries in a table. In these cases the functions must be approximated, using some sort of more compact parameterized function representation.

One must, thus, turn towards approximation techniques. In approximating optimal behavior, there may be many states that the agent faces with such a low probability that selecting sub optimal actions for them has little impact on the amount of reward the agent receives. The on-line nature of reinforcement learning makes it possible to approximate optimal policies in ways that put more effort into learning to make good decisions for frequently encountered states, at the expense of less effort for infrequently encountered states.

There exist three main methods for estimating the optimal policy. These are: *dynamic programming, Monte Carlo methods, and temporal-difference learning*. Covering these methods in details goes beyond the scope of this class. A brief summary of the principles can be found in the slides of the class. The interested reader might refer to [Sutton & Barto, 1998]. Next is a quick summary of the key notion of RL, which should be retained from this very introductory course.

6.3.5 Summary

Reinforcement learning is about learning from interaction how to behave in order to achieve a goal. The reinforcement learning *agent* and its *environment* interact over a sequence of discrete time steps. The specification of their interface defines a particular task: the *actions* are the choices made by the agent; the *states* are the basis for making the choices; and the *rewards* are the basis for evaluating the choices. Everything inside the agent is completely known and controllable by the agent; everything outside is incompletely controllable but may or may not be completely known. A *policy* is a stochastic rule by which the agent selects actions as a function of states. The agent's objective is to maximize the amount of reward it receives over time.

The *return* is the function of future rewards that the agent seeks to maximize. It has several different definitions depending upon the nature of the task and whether one wishes to *discount* delayed reward. The undiscounted formulation is appropriate for *episodic tasks*, in which the agent-environment interaction breaks naturally into *episodes*; the discounted formulation is appropriate for *continuing tasks*, in which the interaction does not naturally break into episodes but continues without limit.

A policy's *value functions* assign to each state, or state-action pair, the expected return from that state, or state-action pair, given that the agent uses the policy. The *optimal value functions* assign to each state, or state-action pair, the largest expected return achievable by any policy. A policy whose value functions are optimal is an *optimal policy*. Whereas the optimal value functions for states and state-action pairs are unique for a given MDP, there can be many optimal policies. Any policy that is *greedy* with respect to the optimal value functions must be an optimal policy. The *Bellman optimality equations* are special consistency condition that the optimal value functions must satisfy and that can, in principle, be solved for the optimal value functions, from which an optimal policy can be determined with relative ease.

A reinforcement-learning problem can be posed in a variety of different ways depending on assumptions about the level of knowledge initially available to the agent. In problems of *complete knowledge*, the agent has a complete and accurate model of the environment's dynamics. If the environment is an MDP, then such a model consists of the one-step *transition probabilities* and *expected rewards* for all states and their allowable actions. In problems of *incomplete knowledge*, a complete and perfect model of the environment is not available.

Even if the agent has a complete and accurate environment model, the agent is typically unable to perform enough computation per time step to fully use it. The memory available is also an important constraint. Memory may be required to build up accurate approximations of value functions,

policies, and models. In most cases of practical interest there are far more states than could possibly be entries in a table, and approximations must be made.

A well-defined notion of optimality organizes the approach to learning we describe in this book and provides a way to understand the theoretical properties of various learning algorithms, but it is an ideal that reinforcement learning agents can only approximate to varying degrees. In reinforcement learning we are very much concerned with cases in which optimal solutions cannot be found but must be approximated in some way.

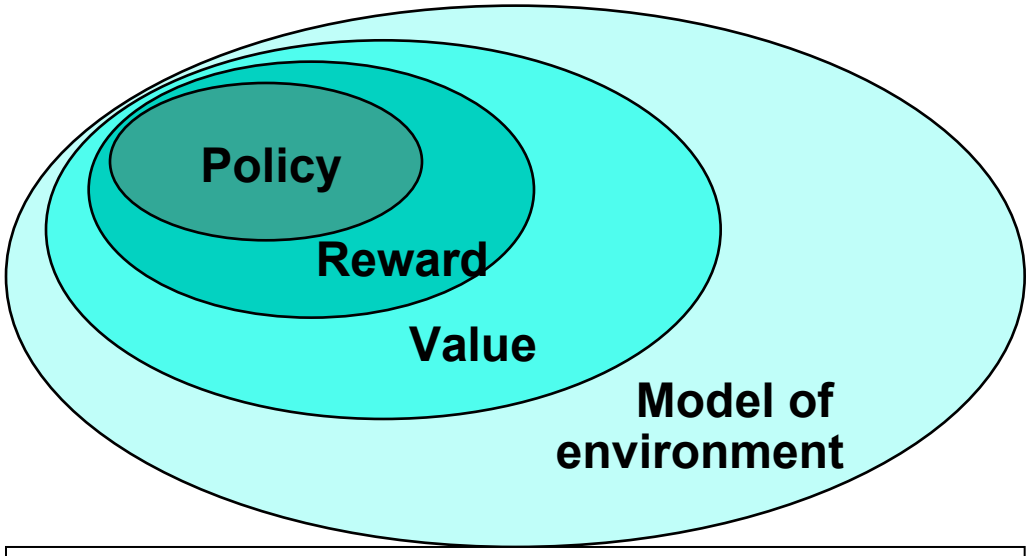


Figure 6-7: Reinforcement learning is characterized by:
- **Policy**: what to do
- **Reward**: what is good
- **Value function**: what is good because it *predicts* reward
- **Model**: what follows what

7 Artificial Neural Networks

Artificial neural network (ANN) has become one of those buzzwords that is either popular or unpopular, but leaves no one indifferent. Unfortunately, for most people, the word relates only to one type of neural network, namely feed-forward neural networks. ANNs are, however, far more than this. We will see, in this chapter and the next one, different ANN architectures that greatly differ from one another, both in terms of computation they can perform as well as in their algorithms. We will see how the architecture and the learning rule determine the type of computation an ANN can perform.

7.1 Applications of ANN

ANNs can perform diverse types of computation. Here is a non-exhaustive list:

Type of Computation	ANNs
Pattern recognition	Feed-forward ANN and Perceptron
Associative memory	Hebbian Network, Willshaw Net, Hopfield Net
PCA	Hebbian Network
ICA	Anti-hebbian learning
Dimensionality Reduction	SOM, Kohonen
Learning Time series	TDNN, RNN

ANNs are powerful tools that can deal with large, multidimensional, non-linear, datasets. They have found application in numerous domains, as statistical tools for data mining in finance and particle physics or as optimization tools in industrial control systems and robotics.

There are two modes of functioning for an ANN:

1. **Activation transfer mode** when activation is transmitted throughout the network.
2. **Learning mode** when the network organizes itself, usually on the basis of the most recent activation transfer.

7.2 Biological motivation

ANNs were developed with the goal to mimic the highly parallel and distributed computation performed in the human brain.

7.2.1 The Brain as an Information Processing System

The human brain contains about 10 billion nerve cells, or **neurons**. On average, each neuron is connected to other neurons through about 10 000 **synapses**. (The actual figures vary greatly, depending on the local neuroanatomy.) The brain's network of neurons forms a massively parallel information processing system. This contrasts with conventional computers, in which a single processor executes a single series of instructions.

Against this, consider the time taken for each elementary operation: neurons typically operate at a maximum rate of about 100 Hz, while a conventional CPU carries out several hundred million machine level operations per second. Despite of being built with very slow hardware, the brain has quite remarkable capabilities:

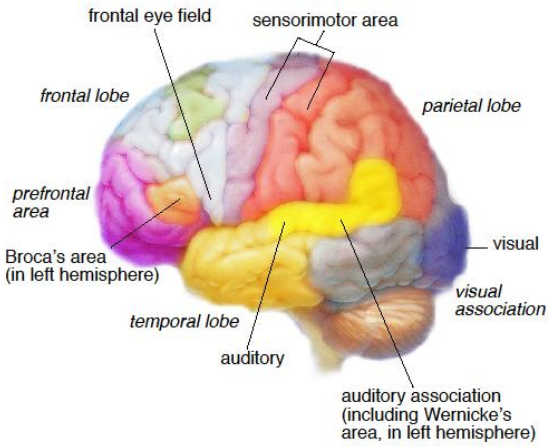
- its performance tends to degrade gracefully under partial damage. In contrast, most programs and engineered systems are brittle: if you remove some arbitrary parts, very likely the whole will cease to function.

- it can learn (reorganize itself) from experience.
- this means that partial recovery from damage is possible if healthy units can learn to take over the functions previously carried out by the damaged areas.
- it performs massively parallel computations extremely efficiently. For example, complex visual perception occurs within less than 100 ms, that is, 10 processing steps!
- it supports our intelligence and self-awareness. (Nobody knows yet how this occurs.)

	processing elements	element size	energy use	processing speed	style of computation	fault tolerant	learns
	10^{14} synapses	10^{-6} m	30 W	100 Hz	parallel, distributed	yes	yes
	10^{8} transistors	10^{-6} m	30 W (CPU)	10^{9} Hz	serial, centralized	no	a little

As a discipline of Artificial Intelligence, Neural Networks attempt to bring computers a little closer to the brain's capabilities by imitating certain aspects of information processing in the brain, in a highly simplified way.

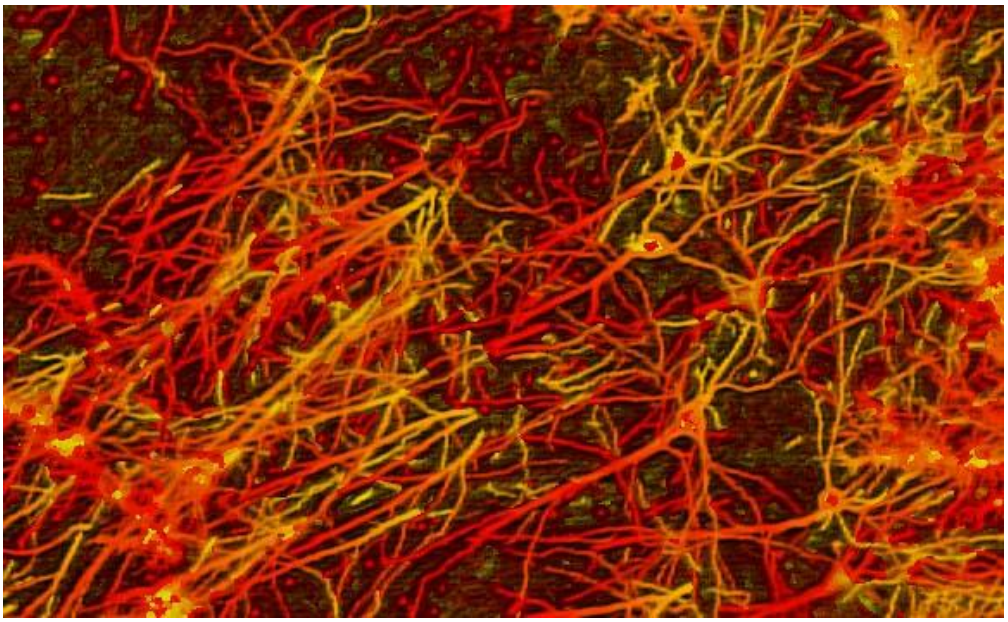
7.2.2 Neural Networks in the Brain



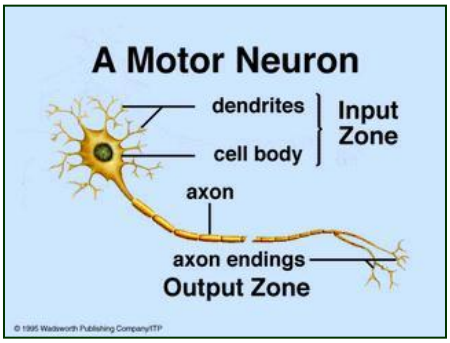
The brain is not homogeneous. At the largest anatomical scale, we distinguish **cortex**, **midbrain**, **brainstem**, and **cerebellum**. Each of these can be hierarchically subdivided into many **regions**, and **areas** within each region, either according to the anatomical structure of the neural networks within it, or according to the function performed by them.

The overall pattern of **projections** (bundles of neural connections) between areas is extremely complex, and only partially known. The best mapped (and largest) system in the human brain is the visual system, where the first 10 or 11 processing stages have been identified. We distinguish **feedforward** projections that go from earlier processing stages (near the sensory input) to later ones (near the motor output), from **feedback** connections that go in the opposite direction.

In addition to these long-range connections, neurons also link up with many thousands of their neighbours. In this way they form very dense, complex local networks.



7.2.3 Neurons and Synapses



The basic computational unit in the nervous system is the nerve cell, or **neuron**. A neuron has:

- Dendrites (inputs)
- Cell body
- Axon (output)

A neuron receives input from other neurons (typically many thousands). Once the input exceeds a critical level, the neuron discharges a **spike** - an electrical pulse that travels from the body, down the axon, to the next neuron(s) (or other receptors). This spiking event is also called **depolarization**, and is followed by a **refractory period**, during which the neuron is unable to fire.

The axon endings (Output Zone) almost touch the dendrites or cell body of the next neuron. Transmission of an electrical signal from one neuron to the next is effected by **neurotransmitters**, chemicals which are released from the first neuron and which bind to receptors in the second. This link is called a **synapse**. The extent to which the signal from one neuron is passed on to the next depends on many factors, e.g. the amount of neurotransmittor available, the number and arrangement of receptors, amount of neurotransmittor reabsorbed, etc.

7.2.4 Synaptic Learning

Brains learn. From what we know of neuronal structures, one way brains learn is by altering the strengths of connections between neurons, and by adding or deleting connections between neurons. Furthermore, they learn "on-line", based on experience, and typically without the benefit of a benevolent teacher.

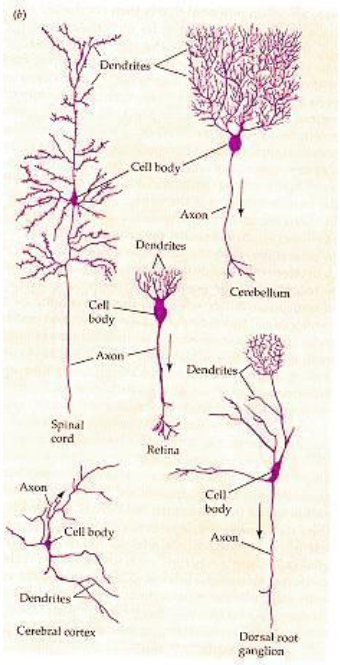
The efficacy of a synapse can change as a result of experience, providing both memory and learning through **long-term potentiation**. One way this happens is through release of more neurotransmitter. Many other changes may also be involved.

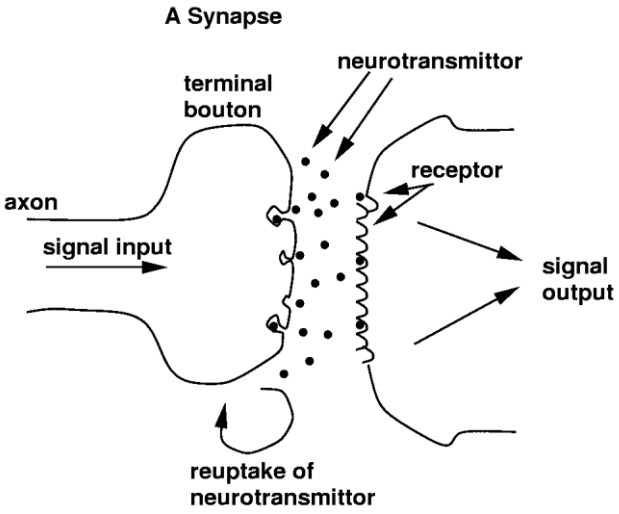
Long-term Potentiation:
An enduring (>1 hour) increase in synaptic efficacy that results from high-frequency stimulation of an afferent (input) pathway → Hebbian Learning, see Section 7.4.

7.2.5 Summary

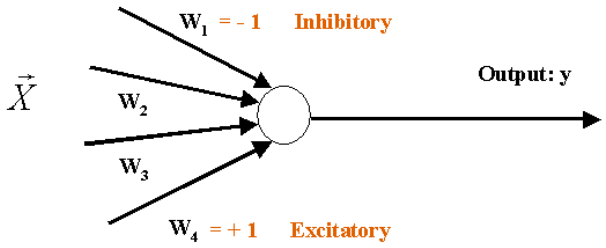
The following properties of nervous systems are of particular interest in neurally-inspired models:

- parallel, distributed information processing
- high degree of connectivity among basic units
- connections are modifiable based on experience
- learning is a constant process, and usually unsupervised
- learning is based only on local information
- performance degrades gracefully if some units are removed





7.3 Perceptron



The earliest neural network models go back to 1940 with the McCulloch & Pitts *perceptron.* The perceptron consists of a simple threshold unit taking binary inputs $\vec{X} = \{x_1,...,x_n\}$. Its output *y* is the product of its entries multiplied by a (1-dim) weight matrix *W*, modulated by a *transfer function* or an *activity function* f.

$$y = f\left(\sum_{i=0}^{n} w_i \cdot x_i\right) = f\left(W \cdot X\right) \tag{6.1}$$

where $w_0 x_0$ is the *bias* and is generally negative.

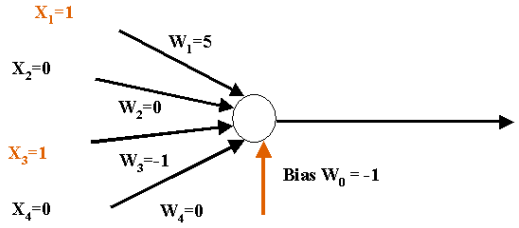


Figure 7-1: Perceptron with constant bias

Classical transfer functions are:

The linear function $f(x) = x$

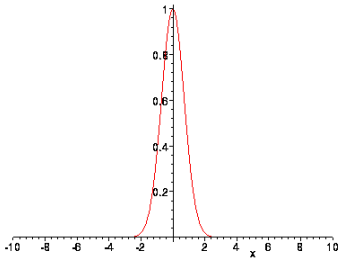
The step function $f(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$

The sigmoid $f(x) = \tanh(x) \quad \in [-1,1]$

Or its positive form: $f(x) = \frac{1}{1+e^{-D \cdot x}} \in [0,1]$***, where D defines the slope of the function***

Radial Basis Functions $f(x) = e^{(-ax^2)}$

A radial basis function is simply a Gaussian.



The sigmoid is probably the most popular activation function, as it is differentiable; this is very important once one attempts to prove the convergence of the system.

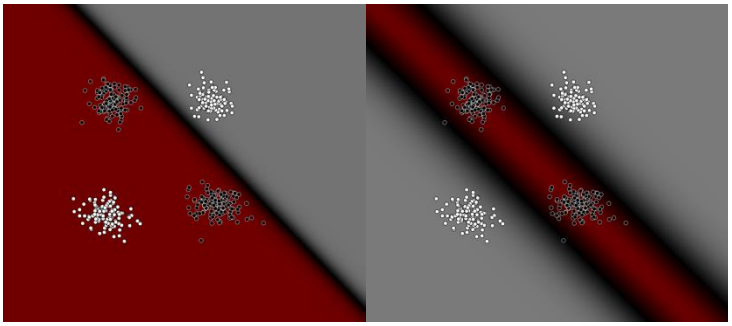
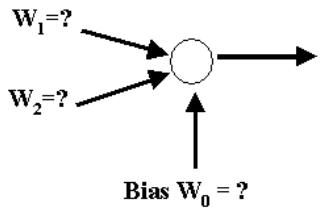


Figure 7-2: The XOR problem is unsolvable using a perceptron and a sigmoid activation function (left). However, by chosing a Radial Basis Function (RBF) it is possible to obtain a working solution (right). Note that the parameter of the RBF kernel needs to be suitably adapted to the size of the data.
[DEMOS\CLASSIFICATION\MLP-XOR.ML]

Exercise: Can you find suitable weights that would make the perceptron behave like an AND-gate, NOR-gate and NAND-gate?



7.3.1 Learning rule for the Perceptron

Let $X^{(p)} = \left\{ x_1^{(p)}, ..., x_n^{(p)} \right\}$ be the set of input patterns and $\hat{y}^{(p)} = \left\{ \hat{y}^{(1)}, ..., \hat{y}^{(P)} \right\}$ the associated set of values for the output.

Learning Algorithm:

1. Initialize the weights to small random values $w_i(0) \quad (0 \le i \le n)$ with w_0 fixed (the bias) and $x_0 = 1$.
2. Present the next pattern $X^{(p)} = \left\{ x_1^{(p)}, ..., x_n^{(p)} \right\}$
3. Calculate $y(t) = f\left(\sum_{i=0}^{n} w_i(t) \cdot x_i^{(p)} \right)$
4. Adjust the weights following $w_i(t) = w_i(t-1) + \eta \cdot \left(\hat{y}^{(p)} - y(t) \right) \cdot x_i^{(p)}(t)$ (6.2)

   The learning rate η modulates the speed at which the weight will vary.

5. Move on to the next pattern and go back to 2.

The process stops if all patterns have been learned. However, it is not clear that all patterns can be learned. The Perceptron acts as a classifier that separates data into two classes. Note that the dataset must be linearly separable for the learning to converge, see illustration in Figure 7-3.

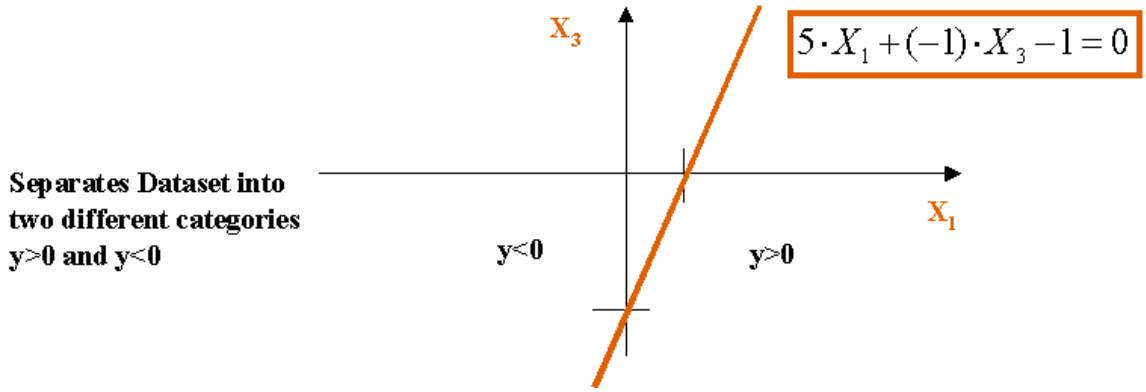


Figure 7-3: Learning in one neuron.

Exercise: Which of the AND, OR and XOR gates satisfy the condition of linear separability?

7.3.2 Information Theory and the Neuron

A good start to understand the properties of ANN is to look at those from the point of view of information theory. Linsker applied information theory to the analysis of the processing of a single neuron. He started by defining the *infomax principle,* which states that: the learning rule should be such that the joint information between the inputs and output of the neuron is maximal.

As mentioned in the introduction, a fundamental principle of learning systems is their robustness to noise. In a single neuron model, one way to measure the neuron's robustness to noise is to determine the joint information between its inputs and output.

Let us consider a neuron with noise on the output:

$$y = \left(\sum_j w_j x_j \right) + \nu \tag{6.3}$$

Let us assume that the output, *y*, and the noise, ν follow a *Gaussian* distribution with zero-mean and variance σ_y^2 and σ_ν^2 respectively. Let us further assume that the noise is uncorrelated with any of the inputs, i.e.

$$E(\nu x_i) = 0 \quad \forall \mathrm{i} \tag{6.4}$$

We can now compute the mutual information between the neuron's input and output:

$$I(x,y) = h(y) - h(y \mid x) \tag{6.5}$$

where $h(z)$ is the entropy of z , see Section 9.3.1.

In other words, *I(x,y)* measures the information contained in the output about the input and is equal to the information in the output minus the uncertainty in the output, when knowing the value of the input.

Since the neuron's activation function is deterministic, the uncertainty on the output is solely due to the noise. Hence, we have: $h(y \mid x) = h(\nu)$. Taking into account the *Gaussian* properties of the output and the noise, we have;

$$\begin{aligned} I(x,y) &= h(y) - h(\nu) \\ &= \frac{1}{2} \log \frac{\sigma_y^2}{\sigma_\nu^2} \end{aligned} \tag{6.6}$$

$\frac{\sigma_y^2}{\sigma_\nu^2}$ stands for the signal/noise ratio of the neuron. Since the experimental set-up fixes the amount and variance of noise, one can modulate only the variance of the output. In order to improve the signal/noise ratio, and, thus, the amount of information, one can, thus, increase the variance of the

output, e.g. by increasing the weights. Note that, by doing this, you increase the unpredictability of the neuron's output, which can have disastrous consequences in some applications.

Noise on the Inputs

In most applications, you will have to face noise in the input, with different type of noise depending on the input. In such a scenario, the output of a neuron would be described by the following:

$$y = \sum_j w_j \left(x_j + \nu_j \right) \tag{6.7}$$

One can show that the mutual information between input and output in this case becomes:

$$I\left(x, y\right) = \frac{1}{2} \log \frac{\sigma_y^2}{\sigma_\nu^2 \left(\sum_j w_j \right)^2} \tag{6.8}$$

In this case, it is not sufficient to just increase the weights, since, by doing so, one will also increase the amount on the denominator. More sophisticated techniques must be used on a neuron-by-neuron basis.

More than one output neuron

Imagine a two input-output scenario in which the two outputs attempt to jointly convey as much information as possible on the two inputs. In this case, each output's neuron's activation is given by:

$$y_i = \sum_j \left(w_{ij} x_j \right) + \nu_i \tag{6.9}$$

Similarly to the 1-output case, we can assume that the noise terms are uncorrelated and Gaussian and we can write:

$$h\left(\nu\right) = h\left(\nu_1, \nu_2\right) = h\left(\nu_1\right) + h\left(\nu_2\right) = 1 + 2\log\left(2\pi\sigma_\nu^2\right)$$

Since the output neurons are both dependent on the same two inputs, they are correlated. One can calculate the correlation matrix *R* as:

$$R = E\left(yy^T\right) = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \left(y_1 \; y_2 \right) = \begin{pmatrix} r_{11} & r_{12} \\ r_{21} & r_{22} \end{pmatrix} \tag{6.10}$$

One can show that the mutual information is equal to

$$I(x,y) = \log\left(\frac{\det(R)}{\sigma_\nu^2}\right) \tag{6.11}$$

Again, the variance of the noise σ_ν^2 is fixed and, so, to maximize mutual information, we must maximize

$$\det(R) = r_{11}r_{22} - r_{12}r_{21} = \sigma_\nu^4 + \sigma_\nu^2\left(\sigma_1^2 + \sigma_2^2\right) + \sigma_1^2 2\sigma_2^2\left(1 - \rho_{12}^2\right)$$

$\sigma_1^2, i = 1,2$ is the variance of each output neuron *in the absence of noise* and ρ_{12} is the correlation coefficient of the output signals also in absence of noise.

One can thus consider the two following situations:

Large noise variance :

If σ_ν is large, one can ignore the 3rd term of the equation. It remains, thus, to maximize the sum $\left(\sigma_1^2 + \sigma_2^2\right)$, which can be done by maximizing the variance of either neuron independently.

Low noise variance:

If, on the contrary, the variance is very small, the 3rd term becomes more important than the two first ones. In that case, one must find a tradeoff between maximizing the variance on each output neuron while keeping the correlation factor sufficiently small.

In other words, in a low noise situation, it is best to use a network in which each neuron's output is de-correlated from one another, i.e. where each output neuron conveys a different information about the inputs; while in a high noise situation, it is best to have a high redundancy in the output. This way one gives more chances for the information conveyed in the input to be appropriately transferred to the output.

7.4 The Backpropagation Learning Rule

In Section 7.3.1, we have seen the perceptron learning rule. Here, we will see a general supervised learning rule for a multi-layer perceptron neural network, called *Backpropagation.* Backpropagation is part of *objective functions,* a class of functions that minimize an error as the criterion for optimization. Such functions are said to perform a *gradient descent* type of optimization, see Section 9.4.3.

Error descent methods are usually associated with supervised learning methods, in which we must provide the network with a set of example data *and* the answer we expect the network to give is presented together with the data.

7.4.1 The Adaline

The Adaline is a simple one-layer feed-forward neural network, composed of perceptron units.

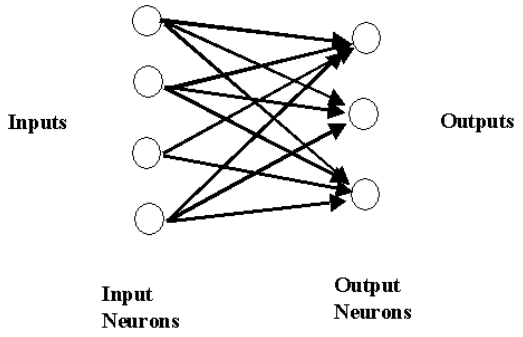


Figure 7-4: one-layer neural network

Let P input patterns $X^p, p = 1,...,P$ be the patterns you want to train the network with. y^p is the *real* output of the network when presented with each X^p, and z^p the *desired* output for the network. One can compute an error measure over all patterns:

$$E = \sum_{p=1}^{P} E^p = \frac{1}{2} \sum_{p=1}^{P} \left(z^p - y^p \right)^2 \tag{6.12}$$

In order to minimize the error, one can determine the gradient of the error with respect to the weights and move the weights in opposite direction.

$$\Delta w_j = -\gamma \frac{\partial E}{\partial w_j} \tag{6.13}$$

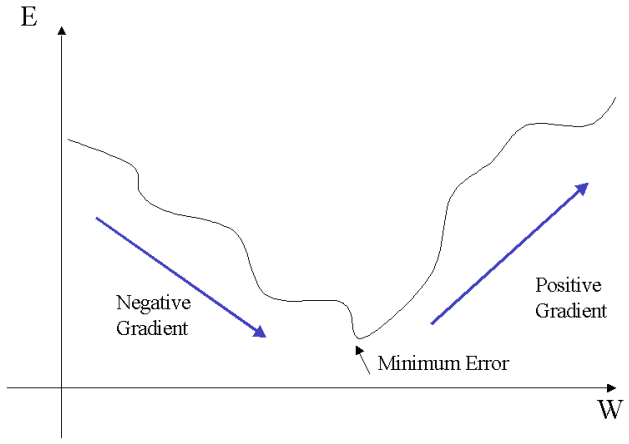


Figure 7-5: A schematic diagram showing the principle of error descent.

If the gradient is positive, changing the weights in a positive direction would increase the error. Therefore, we change the weights in a negative direction. Conversely, if the gradient is negative, one must change the weight in a positive direction to decrease the error.

Because of the linearity of the activation function of the Perceptron neuron (i.e. $y = \sum_j w_j x_j$) of the Adaline network, we have:

$$\frac{\partial y^p}{\partial w_j} = x_j^p \tag{6.14}$$

$$\frac{\partial E^p}{\partial y^p} = -\left(z^p - y^p\right) \tag{6.15}$$

$$\Delta_p w_j = \gamma\left(z^p - y^p\right) \cdot x^p \tag{6.16}$$

This has proven to be a most powerful rule and is at the core of almost all current supervised learning methods for ANN. But, it should be emphasized that nothing we have written guarantees that the method will cause the weight to converge. It can be proved that the method will give an optimal (in a least square error sense) approximation of the function being modeled. However, the method does not ensure convergence to a global optimum.

7.4.2 The Backpropagation Network

An example of multi-layered Perceptron, or *feed-forward neural network,* is shown in Figure 7-6 Activity in the network is propagated forwards via a first set of weights from the input layer to the hidden layer, and, then, via a second set of weights from hidden layer to output layer. The error is calculated by Equation (6.12), similarly to what was done for the Adaline network. Now two sets of weights must be calculated.

Here, however, one does not have access to the desired output of the hidden units. This is referred to, as the ***Credit assignment problem*** – in that we must assign how much effect each weight in the first layer of weights has on the final output of the network. In order to compute the weight change, we need to propagate backwards, to *backpropagate,* the error across the two layers. The algorithm is quite general and applies to any number of hidden layers.

An example of multilayered perceptron is shown in

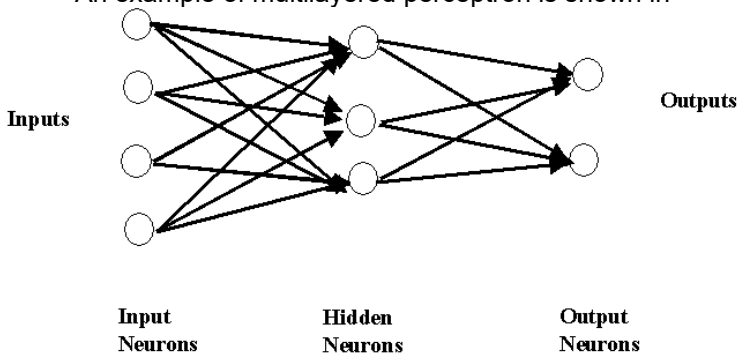


Figure 7-6: Multi-layered feed-forward NN

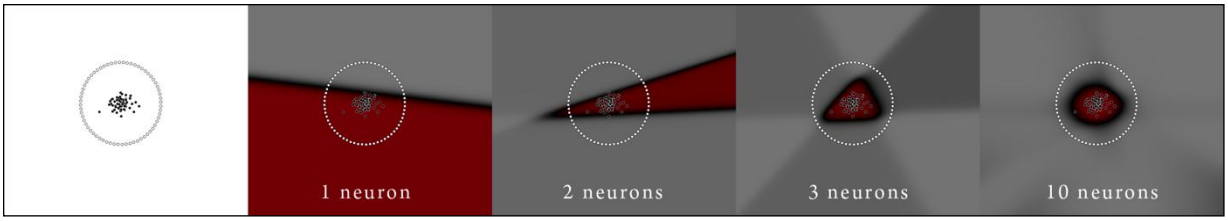


Figure 7-7: A difficult classification problem using a multi-layered feed-forward NN with increasing numbers of neurons in the hidden layer. 3 neurons are sufficient to solve this problem.
[DEMOS\CLASSIFICATION\MLP-NEURONS.ML]

7.4.3 The Backpropagation Algorithm

1. Initialize the weights to small random numbers
2. Present an input patter X^p to the network
3. Compute $a_i^p = f\left(\sum_j w_{ij}^1 x_j\right)$ and $y^p = f\left(\sum_j w_{ij}^2 a_j\right)$ the outputs of the hidden and output units respectively, where f is the activation function (usually the sigmoid), and w_{ij}^1, w_{ij}^2 are the weights from 1st layer, input to hidden units, and 2nd layer, from hidden units to output units, respectively.
4. Compute the error, according to (6.12)
5. Compute the gradient of the error along each weight direction $\frac{\partial E^p}{\partial w_{ij}^2}$ for the second layer.
6. Compute the gradient of the error $\frac{\partial E^p}{\partial w_{ij}^1} = \frac{\partial E^p}{\partial s_i}\frac{\partial s_i}{\partial w_{ij}^1}$ along the hidden units, where $s_i = \sum_j w_{ij}^1 x_j^p$. In order to do this, you need to backpropagate the error to compute $\frac{\partial E^p}{\partial s_i}$.
7. Update all weights according to (6.16)
8. Repeat all steps from 2 for all patterns and until the network has converged (reached a minimal error).

Exercise: Show that a 1-layer feed-forward NN can be used to compute the XOR problem.

7.5 Willshaw net

David Willshaw developed one of the simplest associative memory in 1967.

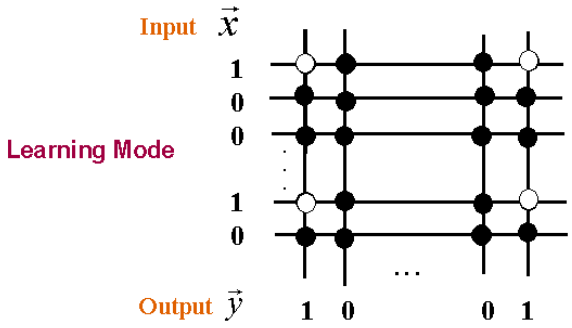


Figure 7-8: The Willshaw Network - Learning phase

The update rule follows:

$$\Delta w_{ij} = \delta\left(x_i \cdot y_i\right) \qquad \delta(h) = \begin{cases} 1 & \text{if h} = 1 \\ 0 & \text{otherwise} \end{cases} \tag{6.17}$$

The retrieval rule follows:

$$y_j = \vartheta\left(\sum_i w_{ij} \cdot x_i - w_0\right) \qquad w_0 = 1 \qquad \vartheta(h) = \begin{cases} 1 & \text{if h} \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{6.18}$$

Such a network suffers from two major drawbacks: the memory quickly fills up; the network cannot learn patterns with overlapping inputs.

One can show that the total number of patterns that can be stored in such a network is:

$$\rho = 1 - \left(1 - \frac{M_{IN}}{N_{IN}} \cdot \frac{M_{OUT}}{N_{OUT}}\right) \tag{6.19}$$

with N_{IN} input lines , M_{IN} the nm of input bits set to 1 , N_{OUT} output lines and M_{OUT} the nm of output bits set to 1.

A more generic learning rule for such associative net is the Hebbian rule, which we will see next.

7.6 Hebbian Learning

Hebbian learning is the core of unsupervised learning techniques in neural networks. It takes its name from the original postulate of the neurobiologist Donald Hebb (Hebb, 1949) stating that:

When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.

The Hebbian learning rule lets the weights across two units grow as a function of the coactivation of the input and output units. If we consider the classical perceptron with no noise:

$$y_i = \sum_j \left(w_{ij} x_j \right) \tag{6.20}$$

Then, the weights increase following:

$$\Delta w_{ji} = \alpha \cdot x_j \cdot y_i \tag{6.21}$$

α is the *learning rate* and is usually comprised between $\left[0,1\right]$. It determines the speed at which the weights grow. If *x* and *y* are binary inputs, the weights increase only when both *x* and *y* are 1. Note that, in the discrete case, the co-activation must be simultaneous. This is often too strong a constraint in a real-time system which displays large variation in the temporality of concurrent events.

A continuous time neural network would best represent such a system. In the continuous case, we would have:

$$\partial w_{ji}\left(t\right) = \alpha\left(t\right) \cdot x_j\left(t\right) \cdot y_i\left(t\right)$$

$$\Delta w_{ji}\left(t\right) = \int_{t_1}^{t_2} \alpha\left(t\right) \cdot x_j\left(t\right) \cdot y_i\left(t\right) dt$$

which corresponds to the area of superposed coactivation of the two neurons in the time interval $[t_1\ t_2]$.

One can show that $\frac{\Delta w_{ij}}{\Delta t} = \sum_k w_{ik} x_k x_j$ and in the limit $\Delta t \longrightarrow 0$ is equivalent to

$$\frac{d}{dt} W\left(t\right) \propto C \cdot W\left(t\right) \tag{6.22}$$

where C_{ij} is the correlation coefficient calculated over all input patterns between the i^{th} and j^{th} term of the inputs and **W(t)** is the matrix of weights at time t.

The major drawback of the Hebbian learning rule, as stated in Equation (6.21), is that weights grow continuously and without bounds. This can quickly get out of hand. If learning is to be continuous, the values taken by the weights can quickly go over the floating-point margin of your system. We will next review two major ways of limiting the growth of the weights.

7.6.1 Weights bounds

One of the easiest way to bound weights is to fix a margin for the weights after which they are no longer incremented, see Figure 7-9. This has the effect of stopping the learning once the upper bound is reached. Another option is to renormalize the weights whenever they reach the upper bound, i.e.

If $w_{ij} = w_{\max}$, then $w_{ij} = \frac{w_{ij}}{w_{\max}} \quad \forall i, j$

Such a renormalization would give more importance to events occurring after the normalization, unless one rescales the increment factor α by the same amount, i.e. $\frac{1}{w_{\max}}$.

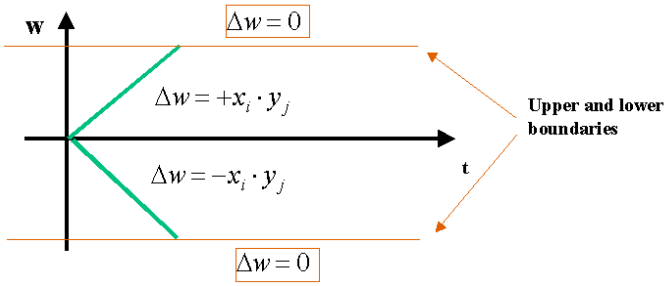


Figure 7-9: Weight clipping

This, however, will quickly lead the network to reach the limit of its floating point encoding for α.

Another option is to renormalize at each time step, while conserving the length of the complete weight vector. The algorithm is performed in two steps:

$$w_j = w_j + \Delta w_j$$

$$w_j = \frac{w_j}{\|w_j\|}$$

This has the advantage to preserve one of the directions along which the weight matrix codes. However, it makes the whole computation heavier. Moreover, it has a tendency of not conserving the relative importance across weight vectors.

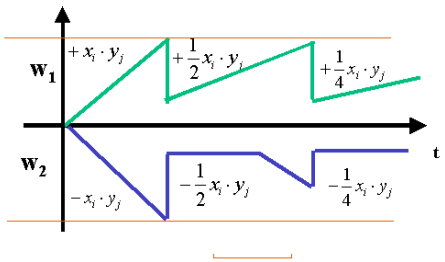


Figure 7-10: Weight renormalization

7.6.2 Weight decay

A more interesting option is to prune weights that seem to have little influence on the network operation. This is an operation that requires global knowledge of all the weights in the network, and, thus, is less supported biologically. The idea is that no single weight should grow too large, while keeping the total weight connections into a particular output neuron fairly constant. One of the simplest rule for weight decay was developed by Grossberg in 1968 and was of the form:

$$\frac{dw_{ij}}{dt} = \alpha y_i x_j - w_{ij} \tag{6.23}$$

It is clear that the weights will be stable when $\frac{dw_{ij}}{dt} = 0$ at the points where $w_{ij} = \alpha E\left(x_i y_j\right)$

One can show that, at stability, we must have $\alpha C w = w$ and, thus, that *w* must be an eigenvector of the correlation matrix *C.*

We will next consider two more sophisticated learning equations, which use weight decay.

The *instar rule* where the decay term is gated by the input term x_j:

$$\frac{dw_{ij}}{dt} = \alpha\left\{y_i - w_{ij}\right\} x_j \tag{6.24}$$

In the discrete case, we have:

$$\Delta w_{ij} = \alpha \cdot x_i \cdot y_j - \gamma \cdot w_{ij} \cdot y_j \tag{6.25}$$

$$\alpha = \gamma \quad \Rightarrow \quad \Delta w_{ij} = \alpha \cdot y_i \cdot \left(x_i - w_{ij}\right) \tag{6.26}$$

$$\begin{aligned} y_i = 1 \quad &\Rightarrow \quad \Delta w_{ij} = \alpha \cdot \left(x_i - w_{ij}\right) \\ &\Rightarrow \quad w_{ij}\left(t\right) = (1-\alpha) \cdot w_{ij}\left(t-1\right) + \alpha \cdot x_i\left(t\right) \end{aligned} \tag{6.27}$$

The weight moves in the direction of the input.

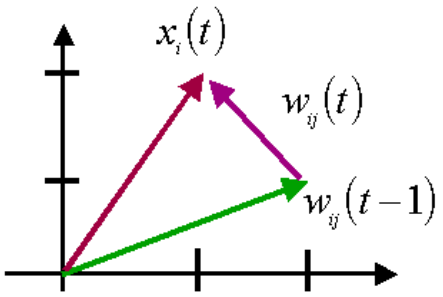


Figure 7-11: The weight vector moves toward the input vector

The *outstar rule*, where the decay term is gated by the output term y_i.

$$\frac{dw_{ij}}{dt} = \alpha \left\{ x_j - w_{ij} \right\} y_i \tag{6.28}$$

In the discrete case, we have:

$$\Delta w_{ij} = \alpha \cdot x_i \cdot \left(y_i - w_{ij} \right) \tag{6.29}$$

$$\begin{aligned} y_i = 1 \quad &\Rightarrow \quad \Delta w_{ij} = \alpha \cdot \left(y_j - w_{ij} \right) \\ &\Rightarrow \quad w_{ij}(t) = (1-\alpha) \cdot w_{ij}(t-1) + \alpha \cdot y_j \end{aligned} \tag{6.30}$$

In this case, the weight vector moves toward the output vector.

7.6.3 Principal Components

Let us consider again the activation function of the Perceptron and rewrite as follows:

$$y_i = \sum_i w_{ij} x_j = \vec{\mathrm{w}}_{\mathrm{i}} \cdot \vec{\mathrm{x}}$$

where $\vec{\mathrm{w}}_{\mathrm{i}}$ is the weight vector along i and $\vec{\mathrm{x}}$ the input vector. We have:

$$\vec{\mathrm{w}}_{\mathrm{i}} \cdot \vec{\mathrm{x}} = \left\| \vec{\mathrm{w}}_{\mathrm{i}} \right\| \left\| \vec{\mathrm{x}} \right\| \cos(\theta)$$

where θ is the angle across the two vector.

This quantity is maximal when the angle θ is zero, i.e. when the two vectors are aligned. Thus, if the weight converge towards the principal components of the input space (i.e. $\vec{\mathrm{w}}_1$ is the 1st eigenvector, $\vec{\mathrm{w}}_2$ the second, etc), then the first output vector y_1 will transmit maximally the input information along the direction with largest variance. In other words, if we define a learning rule,

such that it projects the weights along the principal components of the input space, such a network would in effect produce a PCA analysis.

There are two major advantages of using an ANN to process PCA as opposed to follow the algorithm given in Section **Error! Reference source not found.**. It allows **on-line** as well as **long-life** learning Classical PCA requires having at hand a sample of data point and runs in batch mode. ANNs can be run on-line. They will discover the PCA components as data are fed onto them. This provides more flexibility towards the data.

Let us now show that the classical Hebbian rule converges to the principal components of the input vector:

Recall that $\Delta\vec{w}_i = \alpha y_i \vec{x}$ and that $y_i = \vec{w}_i^T \vec{x}$. Thus, we have:

$$\Delta\vec{w} = \alpha \cdot y \cdot \vec{x} = \alpha \cdot \left(\vec{w}^T \cdot \vec{x}\right) \cdot \vec{x} \tag{6.31}$$

$$\Delta\vec{w} = \alpha \cdot \left|\vec{w}^T\right| \left|\vec{x}\right| \cdot \cos\left(\theta\right) \cdot \vec{x}$$

where θ is the angle between the weight vector and the input vector.

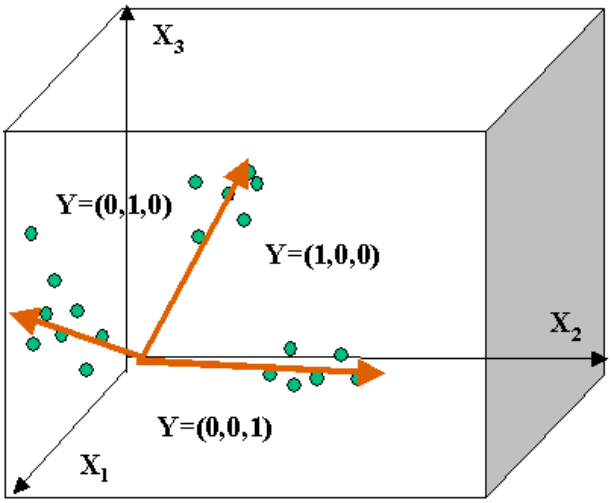
The weight vector moves proportionally to the distance across the weight vector and the input vector: the bigger the distance, the bigger the move. Hebbian learning will, thus, minimize the angle. If the data has zero mean, Hebbian learning will adjust the weight vector $\vec{w}$ so as to maximize the variance in the output y. In other words, when projected onto this direction given by $\vec{w}$, the data will exhibit variance greater than in any other direction. This direction is, thus, the largest *principal component* of the data and corresponds to the eigenvector of the correlation matrix **C** with the largest corresponding eigenvalue.

If we decompose the current $\vec{w}$ in terms of the eigenvectors of **C**, $\vec{w} = \sum_i a_i e_i$, then the

expected weight update rule becomes:

$$\begin{aligned}\Delta\vec{w} &= \alpha \cdot \mathrm{C} \cdot \vec{w} \\ &= \alpha \cdot \mathrm{C} \cdot \sum_{\mathrm{i}} a_i \vec{e}_i \\ &= \alpha \cdot \sum_{\mathrm{i}} a_i \lambda_i \vec{e}_i\end{aligned}$$

This will move the weight vector $\vec{w}$ towards eigenvector $\vec{e}_i$ by a factor proportional to $a_i\lambda_i$. Over many updates, the eigenvector with the largest λ_i will drown out the contributions from the others.



7.6.4 Oja's one Neuron Model

In the 1980's Oja proposed a model that extracts the largest principal components from the input data. The model uses a single output neuron with the classical activation rule:

$$y = \sum_i w_i x_i \tag{6.32}$$

Oja's variation of the Hebbian rule is as follows:

$$\Delta w_i = \alpha\left(x_i y - y^2 w_i\right) \tag{6.33}$$

Note that this rule is defined by a multiplicative constraint of the form $y^2 = \gamma\left(w\right)$ and so will converge to the principal eigenvector of the input covariance matrix. The weight decay term has the simultaneous effect of making $\sum_i \left(w_i\right)^2$ tend towards 1, i.e. the weights are normalized.

This rule will allow the network to find only the first eigenvector. In order to determine the other principal components, one must let the neurons interact with one another.

7.6.5 Convergence of the Weights Decay rule

The Hebbian rule with decay solves the problem of large weights. It eliminates very small and irregular noise. However, it does so at a price. The environment must continuously present all stimuli that have associations. Without reinforcement, associations will decay away. Moreover, it takes longer to eliminate the effect of noise.

As mentioned earlier on, an important and distinctive function of the Hebbian learning rule is its stability. Work by Miller & MacKay made an important contribution in showing the convergence of the Hebbian learning rule with weight decay.

Let us consider the continuous time dependent learning rule with weight decay.

$$\frac{d}{dt}w(t) = Cw(t) - g\left(w(t)\right) \tag{6.34}$$

where *g* is an arbitrary function dependent on the weight *w*.

One can consider two cases:

1) Multiplicative constraints

$$g\left(w(t)\right) = \gamma\left(w(t)\right) \cdot w(t) \tag{6.35}$$

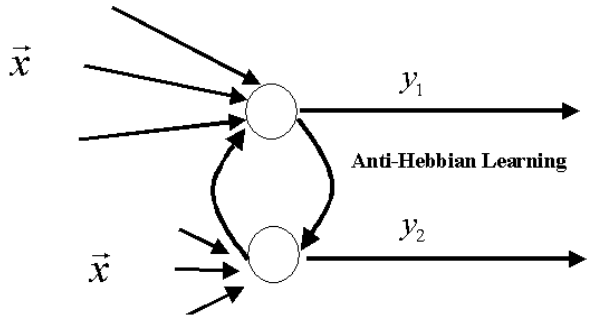
where the decay term is multiplied by the weight. In this case, the decay term can be viewed as a feedback term, which limits the rate of growth of each weight. The bigger the weight, the bigger the decay.

2) Substractive constraint

$$\frac{d}{dt}w(t) = Cw(t) - \gamma\left(w(t)\right)w(t) \tag{6.36}$$

where the weight decay is proportional to the weight and multiplies the weight.

7.7 Anti-Hebbian learning



If inputs to a neural network are correlated, then each contains information about the other. In other words, there is redundancy in the information conveyed by the input and $I\left(x;y\right) > 0$.

Anti-Hebbian learning is designed to *decorrelate* inputs. The ultimate goal is to maximize the information that can be processed by the network. The less redundancy, the more information and the less number of output nodes required for transferring this information.

Anti-Hebbian learning is also known as lateral inhibition, as the anti-learning occurs within members of the same layer. The basic model is defined by:

$$\Delta w_{ij} = -\alpha < y_i \cdot y_j > \tag{6.37}$$

where the angle brackets indicate the ensemble average of values taken over all training patterns. Note that this is a tremendous limitation of the system, as it forces global and off-line learning.

If y_i and y_j are highly correlated, then, the weights between them will grow to a large negative value and each will tend to turn the other off. Indeed, we have:

$$\left(\Delta w_{ij} \to 0\right) \;\Rightarrow\; \left(<y_i, y_j> \to 0\right)$$

The weight change stops when the two outputs are decorrelated. At this stage, the algorithm converges. Note that there is no need for weight decay or renormalizing on anti-Hebbian weights, as they are automatically self-limiting.

7.7.1 Foldiak's models

Foldiak has suggested several models combining anti-Hebbian learning and weight decay. Here, we will consider the first 2 models as examples of solely anti-Hebbian learning. The first model is shown in Figure 7-12 and has anti-Hebbian connections between the output neurons.

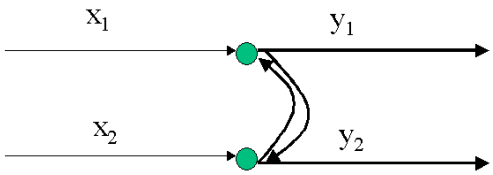


Figure 7-12: Foldiak's 1st model

The equations, which define its dynamical behavior, are

$$y_i = x_i + \sum_{j=1}^{n} w_{ij} y_j \tag{6.38}$$

with learning rule

$$\Delta w_{ij} = -\alpha \cdot y_i \cdot y_j \quad \text{for } i \neq j \tag{6.39}$$

In matrix terms, we have

$$\begin{aligned} y &= x + W \cdot y \\ \text{And so, } \; y &= \left(I - W\right)^{-1} \cdot x \end{aligned} \tag{6.40}$$

Therefore, we can view the system as a transformation, T, from the input vector x to the output y given by:

$$y = T \cdot x = \left(I - W\right)^{-1} \cdot x \tag{6.41}$$

Now, the matrix W must be symmetric. It has only non-zero non-diagonal terms, i.e. if we consider only a two input, two output net as in the diagram.

$$W = \begin{pmatrix} 0 & w \\ w & 0 \end{pmatrix}$$

so that T is given by:

$$T = \left(I - W\right)^{-1} = \begin{pmatrix} 1 & -w \\ -w & 1 \end{pmatrix} = \frac{1}{1-w^2}\begin{pmatrix} 1 & w \\ w & 1 \end{pmatrix} \tag{6.42}$$

Now, let the two dimensional input vector have correlation matrix

$$C_{xx} = \begin{pmatrix} \sigma_1^2 & \rho\sigma_1\sigma_2 \\ \rho\sigma_1\sigma_2 & \sigma_2^2 \end{pmatrix}$$

where ρ is the correlation coefficient and σ_1, σ_2 the variance of the elements of x. Now the correlation matrix for y can be calculated. Since $y = T \cdot x$, we have:

$$C_{yy} = E\left\{yy^T\right\} = E\left\{T \cdot x \cdot \left(T \cdot x\right)^T\right\} = T \cdot C_{xx} \cdot T^T.$$

Then,

$$C_{yy} = \frac{1}{\left(w^2-1\right)^2}\begin{pmatrix} \sigma_1^2 + 2w\rho\sigma_1\sigma_2 + w^2\sigma_2^2 & \rho\sigma_1\sigma_2\left(w^2+1\right) + \left(\sigma_1^2+\sigma_2^2\right)w \\ \rho\sigma_1\sigma_2\left(w^2+1\right) + \left(\sigma_1^2+\sigma_2^2\right)w & \sigma_2^2 + 2w\rho\sigma_1\sigma_2 + w^2\sigma_1^2 \end{pmatrix} \tag{6.43}$$

The anti-Hebbs rule reaches equilibrium when the units are decorrelated and so the terms $w_{12} = w_{21} = 0$. Notice that this gives us a quadratic equation in w (which naturally we can sove).

Let us consider the special case that the elements of x have the same variance so that $\sigma_1 = \sigma_2 = \sigma$. Then, the cross correlation terms become $\rho\sigma^2\left(w^2+1\right) + \left(2\sigma^2\right)w + \rho\sigma^2$ and so we must solve the quadratic equation:

$$\rho w^2 + 2w + \rho = 0 \tag{6.44}$$

which has a zero at

$$w_f = \frac{-1 + \sqrt{1-\rho^2}}{\rho} \tag{6.45}$$

One can further show that this is a stable point in the weight space.

Foldiak's second model allows all neurons to receive their own outputs with weight 1.

$$\Delta w_{ii} = \alpha\left(1 - y_i y_i\right) \tag{6.46}$$

which can be written in matrix form as

$$\Delta W = \alpha\left(I - YY^T\right) \tag{6.47}$$

where I is the identity matrix.

This network will converge when the outputs are decorrelated (due to the off-diagonal anti-Hebbian learning) and when the expected variance of the outputs is equal to 1. i.e. this learning rule forces each network output to take responsibility for the same amount of information since the entropy of each output is the same.

This is generalizable to

$$\Delta w_{ij} = \alpha\left(\theta_{ij} - y_i y_j\right) \tag{6.48}$$

Where $\theta_{ij} = 0 \quad \text{for} \quad i \neq j$. The value of θ_{ii} for all i, will determine the variance on that output and so we can manage the information output of each neuron.

7.7.2 CCA Revisited

Adapted from Peiling Lai and Colin Fyfe, Kernel and Nonlinear Canonical Correlation Analysis, Computing and Information Systems, 7 (2000) p. 43-49.

The Canonical Correlation Network

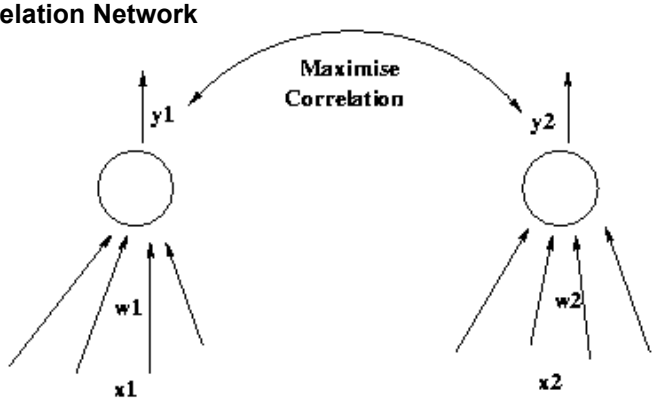


Figure 1 The CCA Network. By adjusting weights, w_1 and w_2, we maximize correlation between y_1 and y_2.

Let us consider CCA in artificial neural network terms. The input data comprises two vectors $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$. Activation is fed forward from each input to the corresponding output through the respective weights, $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ (see Figure 1 and equations (1) and (2)) to give outputs y_1 and y_2.

One can derive an objective function for the maximization of this correlation under the constraint that the variance of y_1 and y_2 should be 1 as:

$$J = E\{(y_1 y_2) + \frac{1}{2}\lambda_1(1 - y_1^2) + \frac{1}{2}\lambda_2(1 - y_2^2)\} \tag{6.49}$$

where the constraints have been implemented using Lagrange multipliers, λ_1 and λ_2. Using gradient ascent with respect to $\boldsymbol{w}_i$ and gradient descent with respect to λ_i gives the learning rules:

$$\begin{aligned}
\Delta w_{1j} &\sim x_{1j}\left(y_2 - \lambda_1 y_1\right) \\
\Delta\lambda_1 &\sim -\left(1 - \left(y_1\right)^2\right) \\
\Delta w_{2j} &\sim x_{2j}\left(y_1 - \lambda_2 y_2\right) \\
\Delta\lambda_2 &\sim -\left(1 - \left(y_2\right)^2\right)
\end{aligned} \tag{6.50}$$

where w_{1j} is the j^{th} element of weight vector $\boldsymbol{w}_1$ etc.

However, just as a neural implementation of Principal Component Analysis (PCA) may be very interesting but not a generally useful method of finding Principal Components, so a neural implementation of CCA may be only a curiosity. It becomes interesting only in the light of performing *nonlinear* CCA, which is described next.

7.7.2.1 Non-linear Canonical Correlation Analysis

To determine whether a neural network as described in the previous section can extract nonlinear correlations between two data sets, $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$, and further to test whether such correlations are greater than the maximum linear correlations, one must introduce a nonlinearity term to the network. This takes the usual form of a tanh() function, which we also find in classical approach to non-linear ANN approximation (e.g. as in feedforward NN with backpropagation).

The outputs y_1 and y_2 of the network become:

$$y_1 = \sum_j w_{1j} \tanh\left(v_{1j} x_{1j}\right) = w_1 f_1$$

$$y_2 = \sum_j w_{2j} \tanh\left(v_{2j} x_{2j}\right) = w_2 f_2$$

To maximise the correlation between y_1 and y_2, we again use the objective function

$$J_1 = E\{(y_1 y_2) + \frac{1}{2}\lambda_1(1 - y_1^2) + \frac{1}{2}\lambda_2(1 - y_2^2)\}$$

whose derivatives give us

$$\begin{aligned}\frac{\partial J_1}{\partial w_1} &= f_1 y_2 - \lambda_1 y_1 f_1 \\ &= f_1\left(y_2 - \lambda_1 y_1\right) \\ \frac{\partial J_1}{\partial v_1} &= w_1 y_2\left(1 - f_1^2\right)x_1 - \lambda_1 w_1\left(1 - f_1^2\right)x_1 y_1 \\ &= w_1\left(1 - f_1^2\right)x_1\left(y_2 - \lambda_1 y_1\right)\end{aligned} \tag{6.51}$$

where in this last equation, all the vector multiplications are to be understood as being on an element by element basis. Similarly with $\boldsymbol{w}_2$, $\boldsymbol{v}_2$, and λ_2. This gives us a method for changing the weights and the Lagrange multipliers on an online basis. This leads to the joint learning rules:

$$\begin{aligned}\Delta w_1 &= \eta f_1\left(y_2 - \lambda_1 y_1\right) \\ \Delta v_{1i} &= \eta x_{1i} w_{1i}\left(y_2 - \lambda_1 y_1\right)\left(1 - f_1^2\right)\end{aligned} \tag{6.52}$$

and similarly with the second set of weights.

7.7.3 ICA Revisited

An important aspect of anti-Hebbian learning is its ability to decorrelate inputs. As discussed in Chapter **Error! Reference source not found.**, decorrelating the dataset if often too soft a constraint, and, it is often more desirable to find independent components of the dataset, so as to reduce maximally its dimensionaly, and, in the case of ANN, so as to maximize the information transmitted by the network.

Jutten and Herault proposed a neural network architecture, based on anti-Hebbian learning, that can perform Independent Component Analysis. The activation function is as follows:

$$y_i = x_i - \sum_{j=1}^{n} w_{ij} y_j \tag{6.53}$$

Which is equivalent to:

$$y = x - Wy \tag{6.54}$$

$$y = \left(I + W\right)^{-1} x \tag{6.55}$$

While this is similar to the Foldiak's model, a crucial difference lies in the non-linearity of the learning rule, defined as follows:

$$\Delta w_{ij} = -\alpha f\left(y_i\right) g\left(y_j\right) \quad \text{for} \quad i \neq j \tag{6.56}$$

Notice that, if we use the identity function for *f* and *g*, we find again the classical Hebbian rule. Recall that, if the two variables are uncorrelated, we have $E\{y_1, y_2\} = 0$, and that if they are independent we have $E(f(y_1)f(y_2)) = E(f(y_1))E(f(y_2))$ for any given function *f*. The network must, thus, converge to a solution that satisfies the later condition.

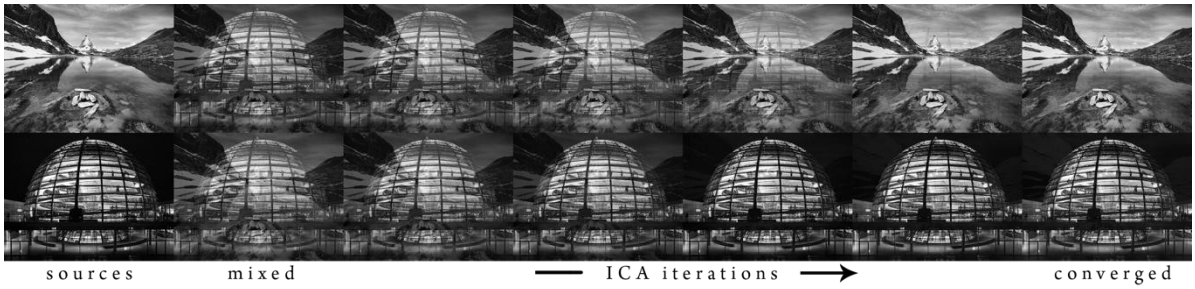


Figure 7-13: ICA with anti-Hebbian learning applied to two images that have been mixed together. After a number of iterations the network converges to a correct separation of the two source images.
[DEMOS\ICA\ICA_IMAGE_MIX.M]

7.8 The Self-Organizing Map (SOM)

The SOM is an algorithm used to visualize and interpret large high-dimensional data sets. Typical applications are visualization of process states or financial results by representing the central dependencies within the data on the map. It is a way of reducing the dimensionality of a dataset, by producing a map of usually 1 or 2 dimensions, which plot the similarities of the data by grouping similar data items together.

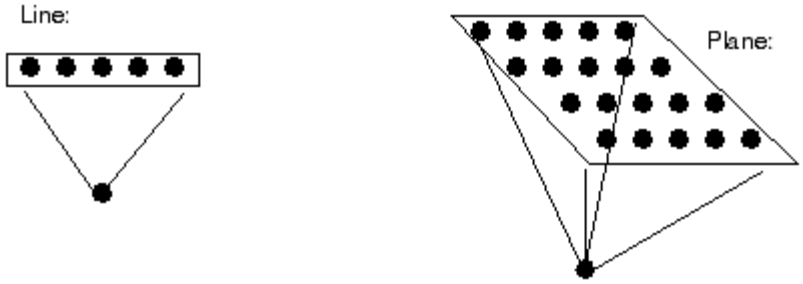
The map consists of a regular grid of processing units, "neurons". A model of some multidimensional observation, eventually a vector consisting of features, is associated with each unit. The map attempts to represent all the available observations with optimal accuracy using a restricted set of models. At the same time the models become ordered on the grid so that similar models are close to each other and dissimilar models far from each other.

7.8.1 Kohonen Network

Kohonen's SOM is called a topology-preserving map because there is a topological structure imposed on the nodes in the network. A topological map is simply a mapping that preserves neighborhood relations.

In the networks we have considered so far, we have ignored the geometrical arrangements of output nodes. Each node in a given layer has been identical in that each is connected with all of the nodes in the upper and/or lower layer. We are now going to take into consideration that physical arrangement of these nodes. Nodes that are "close" together are going to interact differently than nodes that are "far" apart.

What do we mean by "close" and "far"? We can think of organizing the output nodes in a line or in a planar configuration.



The goal is to train the net so that nearby outputs correspond to nearby inputs. E.g. if x_1 and x_2 are two input vectors and z_1 and z_2 are the locations of the corresponding winning output nodes, then z_1 and z_2 should be close if x_1 and x_2 are similar. A network that performs this kind of mapping is called a feature map.

In the brain, neurons tend to cluster in groups. The connections within the group are much greater than the connections with the neurons outside of the group. Kohonen's network tries to mimic this in a simple way.

7.8.1.1 Algorithm for Kohonen's Self Organizing Map

Assume output nodes are connected in an array (usually 1 or 2 dimensional). Assume that the network is fully connected - all nodes in input layer are connected to all nodes in output layer.

Use the competitive learning algorithm as follows:

1. Initialize the weights – the initial weights may be random values or values assigned to cover the space of inputs.

2. Present an input $\vec{x}$ to the net

3. For each output node j, calculate the distance between the input $\vec{x}$ and its weight vector $\vec{w}^j$, e.g. using the Euclidean distance

$$d\left(\vec{x},\vec{w}^j\right)=\left|\vec{x}-\vec{w}^j\right|=\sum_{i=1}^{n}\left|x_i-w_i^j\right| \tag{6.57}$$

4. Determine the "winning" output node i, for which *d* is minimal.

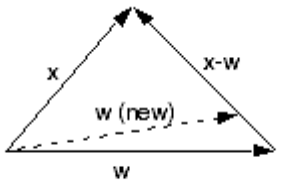
$$\left|\vec{w}^i-\vec{x}\right|\le\left|\vec{w}^k-\vec{x}\right| \qquad \forall k \tag{6.58}$$

Note: the above equation is equivalent to w_i x >= w_k x only if the weights are normalized.

5. Update the weights of the winner node and of nodes in a *neighborhood* using the rule:

$$\Delta\vec{w}^k\left(t\right)=\eta\left(t\right)\cdot h_{k,i}\left(t\right)\cdot\left(\vec{x}-\vec{w}^i\left(t\right)\right) \tag{6.59}$$

where $\eta\left(t\right)\in[0,1]$ is a learning rate or gain and $h_{k,i}\left(t\right)=\frac{1}{r_{k,i}\left(t\right)}$ is the neighborhood function. It is equal to 1 when $i=k$ and falls off with the distance r_{ki} between output nodes i and k. Thus, the closer are the units to the winner, the larger the update. Note that both the learning rate and the neighborhood vary with time. It is here that the topological information is supplied. Nearby units receive similar updates and thus end up responding to nearby input patterns. The above rule drags the weight vector w^i and the weights of nearby units towards the input x.



Note that if the output nodes are 1, the Kohonen rule is equivalent to the Outstar rule, see Section 7.6.2.

$$\Delta w_{ij}=\alpha\cdot x_i\cdot y_j-\gamma\cdot w_{ij} \tag{6.60}$$

6. Decrease the size of the neighborhood and the learning rate

Example of neighborhood function could be $h_{k,i}(t) = e^{\frac{\left(-r_{k,i}(t)^2\right)}{2\sigma^2}}$ where σ^2 is the width parameter that can gradually be decreased over time.

7. Unless the net is adequately trained (for example η has become very small), go back to step 2.

7.8.1.2 Normalized weight vectors

The weights in Kohonen net are often normalized to unit length. This allows a somewhat cheaper computation when determining the winning unit. In effect, the network tries to adjust the *direction* of the weight vectors to line up with that of the input. The basic idea is that the distance between the two vectors is minimal, when the two vectors are aligned. So, minimizing the Euclidean distance is equivalent to maximizing the vector dot product $\vec{x} \cdot \vec{w}^i$.

7.8.1.3 Caveats for the Kohonen Net

The use of Kohonen nets is not as straightforward as it might appear. Although the net does much of the work in sorting out the organization implicit in the input space, problems can arise.

The first difficulty is in choosing the initial weights. These are often set at random, but if the distribution of the weights actually selected does not match well with the organization of the input space, the net will converge very slowly. One way round to this is to set the initial weights deliberately to reflect the structure of the input space if that is known. For example, thee unit weight vectors could be smoothly and uniformly distributed across the input space.

A second problem is in deciding how to vary the neighborhood and learning rate to achieve the best organization. The answer really depends on the application. Note, though, that the standard Kohonen training algorithm assumes that the training and performance are separate: first you train the network and, when that is complete, then you use it to classify inputs. For some applications (for instance, in robot control) it is more appropriate that the net continue learning and that it generates classifications even though partially trained.

A third problem concerns the presentation of inputs. The training procedure above presupposes that inputs are presented fairly and at random – in other words – there is no systematic bias in favor of one part of the input space against others. Without this condition, the net may not produce a topographic organization, but rather a single clump of units all organized around the over represented part of the input space. This problem too arises in robotic applications, where the world is experienced sequentially and consecutive inputs are not randomly chosen from the input space. Batching up inputs and presenting them randomly from those batches can solve the problem.

7.8.2 Bayesian self-organizing map

Adapted from Yin, H.; Allinson, N.M. Self-organizing mixture networks for probability density estimation IEEE Transactions on Neural Networks, Volume 12, Issue 2, March 2001 Page(s):405 – 411.

The *Bayesian self-organizing map* (BSOM) is a method for estimating a probability distribution generating data points on the basis of a Bayesian stochastic model. BSOM can be used to estimate the parameters of Gaussian Mixture Models (GMM). In this case, BSOM estimates GMM parameters by minimizing the Kullback–Leibler information metric and as such provides an alternative to the classical Expectation-Maximization (EM) method for estimating the GMM parameters which performs better in terms of both convergence speed and escape of local minima. Since BSOM makes no assumption on the form of the distribution (thanks to the KL metric), it can be applied to estimate other mixture of distribution than purely Gaussian distribution.

The term SOM in BSOM is due to the fact that the update rule uses a concept similar to the neighborhood of the update rule in self-organizing map (SOM) neural network. BSOM creates a set *K probability density distribution* (PDF). At the instar of EM that updates the parameters of the pdf globally so as to maximize the total probability that the mixture explains well the data, BSOM updates each pdf separately by using only a subset of the data located in a neightbourhood around the data point for which the pdf is maximal. The advantage is that the update and estimation steps are faster. However, the drawback is that like Kohonen network, the algorithm depends very much on the choice of the hyperparameters (nm of pdf, size of the neighbourhood, learning rate).

7.8.2.1 Stochastic Model and Parameter Estimation

BSOM proceeds as follows. Suppose that the distribution of data points is given by $p(x)$. BSOM will build an estimate $\hat{p}(x)$ by constructing a mixture distribution of *K* probability distribution functions (pdf) $p_i(x)$ with associated parameters θ_i , $i = 1...K$, on a d-dimensional input space x. If $P_1,...P_K$ are the prior probabilities of each pdf, then the joint probability density for each data sample is given by:

$$\hat{p}(x) = \sum_{i=1}^{K} p_i(x \mid \theta_i) P_i \tag{6.61}$$

BSOM will update incrementally the mixture so as to minimize the divergence between the two distributions measured by the Kullback-Leibler metric:

$$I = -\int \log\left(\frac{\hat{p}(x)}{p(x)}\right) p(x)\, dx \tag{6.62}$$

The update step (or M-step by analogy to EM) consists of re-estimating the parameters of each pdf by minimizing *I* via its partial derivatives over each parameter $\frac{\partial I}{\partial \theta_{ij}}$ under the constraint that

the resulting distribution is a valid probability, i.e. $\sum_{i=1}^{K} P_i = 1$. Such an optimization problem can be resolved through Lagrange and gives the following update step:

$$P_i(n+1) = P_i(n) - \alpha(n)\left[P(i \mid x) - P_i(n)\right] \tag{6.63}$$

where *n* is the update step and α is the learning rate.

To speed up the computation, the updating of the above parameters is then limited to a small neighborhood around the "*winning node"*, i.e. the pdf *i* which has the largest posterior probability $p(i \mid x)$. The distribution for a given point *x* is then approximated by a mixture of a small number of nodes at one time, i.e.

$$\hat{p}(x) \approx \sum_{i \in \eta} p_i(x \mid \theta_i) P_i \tag{6.64}$$

with η a neighborhood around the winner.

Learning thus proceeds as in Kohonen. At each iteration step *n,* one picks a data point at random, selects the winning pdf for this data points and then update the parameters for all the pdf located in a neighborhood of the winning distribution (i.e. the pdf for which the maximum is located on a point in a neighborhood of the data point chosen here).

7.9 Static Hopfield Network

In his 1982 seminal paper, John J. Hopfield [Hopfield, 1982] presented a novel type of associative memory, consisting in a fully connected artificial neural network, later called the *Hopfield Network*. This work was one of the first demonstrations that models of physical systems, such as neural networks, could be used to solve computational problems. Later, follow-up on this research showed that such systems could be implemented in hardware by combining standard components such as capacitors and resistors.

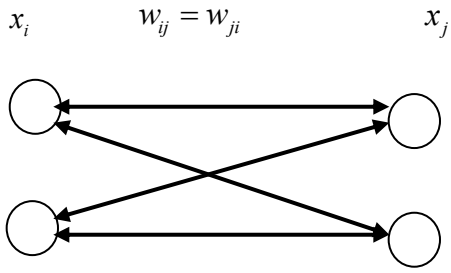
The original Hopfield network, which we will refer to as the static Hopfield Network, consists of a one-layer fully recurrent neural network. Subsequent development led Hopfield and coworkers to develop different extensions. The major ones are *the Bolzmann Machine,* in which a parameter (the temperature) can be varied so as to increase the capacity of the model, and the *continuous time Hopfield,* whose neurons dynamics is continuous over time. In this class, we will consider only the static and continuous time Hopfiel model. The interested reader can refer to [Barna & Kaski, 1989; Ackley et al, 1985] for an introduction to the Boltzmann Machine.

The Hopfield network is created by supplying input data vectors, or pattern vectors, corresponding to the different classes. These patterns are called *class patterns*. In an *n*-dimensional data space the class patterns should have *n* binary components {1,-1}; that is, each class pattern corresponds to a corner of a cube in an *n*-dimensional space. The network is then used to classify distorted patterns into these classes. When a distorted pattern is presented to the network, then it is associated with another pattern. If the network works properly, this associated pattern is one of the class patterns. In some cases (when the different class patterns are correlated), spurious minima

can also appear. This means that some patterns are associated with patterns that are not among the pattern vectors.

Hopfield networks are sometimes called *associative networks* since they associate a class pattern to each input pattern. The importance of the different Hopfield networks in practical application is limited due to theoretical limitations of the network structure (capacity sensitive to correlated data) but, in situations where data can be decorrelated, they may form interesting models. Hopfield networks are typically used for classification problems with binary pattern vectors.

7.9.1 Hopfield Network Structure



The Hopfield network is composed of K neurons and K^2 connection weights $w_{ij}, i, j = 1,..,K$. It is fully connected through symmetric, bi-directional weights, i.e. $w_{ij} = w_{ji}$. It has no self-connections, i.e. $w_{ii} = 0 \ \ \forall i$.

In the Hopfield network, learning takes one time step and retrieval takes several time steps!

7.9.2 Learning Phase

The learning rule is intended to make a set of desired *patterns* $\vec{x}^n = \{x_1^n,...,x_K^n\}, \quad n = 1,...,N$ *stable states* of the Hopfield network's activity rule.

Initialization:
At time t=0, set all the weights to $w_{ij}(0) = w_{ji}(0) = 0 \quad \forall i, j$.

Update:

The network weights are updated only once to represent the correlations across all bits from all patterns, following:

$$w_{ji} = \eta \sum_n x_i^n x_j^n \tag{6.65}$$

$$x_j^n \in \{-1,1\}$$

To simplify the description of the information retrieval, one, often, uses:

$$\eta = \frac{1}{K} \tag{6.66}$$

7.9.3 Retrieval Phase

Each neuron updates its state as if it were a single neuron with the threshold activation function

$$x_j = \theta\left(\sum_i w_{ij} x_i\right), \qquad \theta(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{if } z < 0 \end{cases} \tag{6.67}$$

$$x_j^n \in \{-1, 1\}$$

The activity of the neurons can be updated either *synchronously* or *asynchronously.* The retrieval procedure is iterative and consists of the following:

Initialization:

At time t=0, inject a pattern to the net $\vec{x}^{\mu}$ (possibly a noisy version of a stored pattern or a random pattern), i.e. set the state of all the network's neuron to

$$x_1(0) = x_1^{\mu}, \ldots, x_K(0) = x_K^{\mu}$$

Update:

Synchronous Update:

$$x_j(t) = \theta\left(a_j(t-1)\right) \quad \forall j \quad a_j(t-1) = \left(\sum_i w_{ij} x_i(t-1)\right) \tag{6.68}$$

All neurons compute their activation and update their state simultaneously following:

Asynchronous Update:

One neuron at a time computes its activations and updates its state. The sequence of selected neuron may be fixed or random:

$$x_j(t) = \theta\left(a_j(t-1)\right) \qquad a_j(t-1) = \left(\sum_i w_{ij} x_i(t-1)\right) \tag{6.69}$$

$$x_i(t) = x_i(t-1) \quad \forall \ i \neq j$$

$$x_i(t+1) = \theta\left(a_i(t)\right) \qquad a_i(t) = \left(\sum_j w_{ji} x_j(t)\right)$$
$$x_j(t+1) = x_j(t) \quad \forall \ \mathrm{j} \neq i$$

If $x_j(t) = x_j(t-1)$ for all units j, then the network has reached a stable state, otherwise we keep changing the state of the units until it converges.

Note that the properties of the Hopfield network may be sensitive to the choice of synchronous versus asynchronous activation.

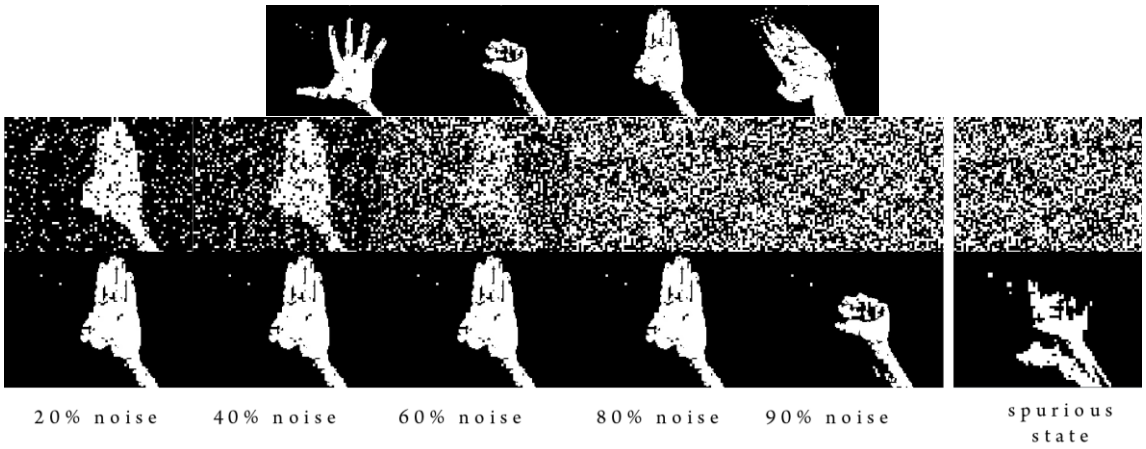


Figure 7-14: Hopfield Network trained on 4 patterns (top row). (middle row) increasing amounts of noise are added to one of the patterns. (bottom row) Hopfield network after convergence from the noisy state. Notice how the network is able to retrieve the original pattern even when most of the image is noise. The rightmost image is pure random noise, the network converges to a spurious state due to a local minimum in the network energy. [DEMOS\HOPFIELD\HOPFIELD.EXE]

7.9.4 Capacity of the static Hopfield Network

One can show that the maximal number of patterns *N* that can be stored in a network of size *K* is*:*

$$N_{\max} \simeq 0.138 \cdot K \tag{6.70}$$

The capacity of the Hopfield network decreases importantly in the face of correlated patterns. One way of suppressing the effect of correlated patterns is to modify the learning rule in order to *decorrelate* the patterns. The learning rule becomes:

$$w_{ji} = \frac{1}{K} \sum_{\mu=1}^{N} \sum_{\nu=1}^{N} x_i^{\mu} \left(C^{-1}\right)_{\mu\nu} x_j^{\nu} \tag{6.71}$$

where $\left(C^{-1}\right)_{\mu\nu}$ is the $\mu\nu$ th element of the inverse matrix C^{-1}. *C* measures the degree of overlap of each pattern on each bit and is given by:

$$\left(C\right)_{\mu\nu} = \sum_{i=1}^{N} x_i^{\mu} x_i^{\nu} \tag{6.72}$$

The price paid for the improvement on the network capacity is that the learning rule becomes non-local, which makes the model less attractive from a biological point of view.

7.9.5 Convergence of the Static Hopfield Network

One can show that the Hopfield Network with an asynchronous update rule is bound to converge. The idea is that the dynamics of the system can be described by an *Energy function:*

$$E = -\frac{1}{2}\sum_{i,j} W_{ij} s_i s_j \tag{6.73}$$

It remains to show that *E* is a Lyapunov function; i.e., it is bound to converge to a stable fixed point x^*. A Lyapunov function *E* is, by definition, a *continuous differentiable, real value* function with the following two properties:

$$\text{Positive Definite: } E\left(x\right) > 0 \quad \forall x \neq x^* \quad \text{and} \quad E\left(x^*\right) = 0 \tag{6.74}$$

$$\text{All trajectory flow downhill: } \frac{d}{dx} E\left(x\right) < 0 \quad \forall x \neq x^* \tag{6.75}$$

Hint: Once the network has converged, all units have reached a stable state, i.e. $s_i\left(t\right) = s_i\left(t+1\right) \quad \forall i$.

7.10 Continuous Time Hopfield Network and Leaky-Integrator Neurons

As mentioned previously, the convergence and capacity of the Hopfield network can be majorly affected when changing from asynchronous update to synchronous update. It turns out that, once we move to a continuous-time version of the Hopfield network, this issue melts away.

Let us assume that each neuron's activity x_j is a continuous function of time $x_j\left(t\right)$. The neuron's response to the activation of other neurons in the network is then given by a first-order differential equation:

$$\tau_i \frac{dm_i}{dt} = -m_i + \sum_j w_{ij} \cdot x_j(t)$$
$$x_i = \frac{1}{1+e^{-D\cdot(m_i+b)}} \tag{6.76}$$

where m_i is the membrane potential, x_i the firing rate, τ_i the time constant, b the bias and D a scalar that defines the slope of the sigmoid function.

Neurons following the activation given in (6.76) are called *leaky-integrator neurons.* $f\left(\sum_j w_{ij} \cdot x_j(t)\right)$ corresponds to the *integrative* term. As an effect of this term, the activity of the neuron increases over time due to the external activation of the other neurons to reach a plateau. The speed of convergence depends on τ , see

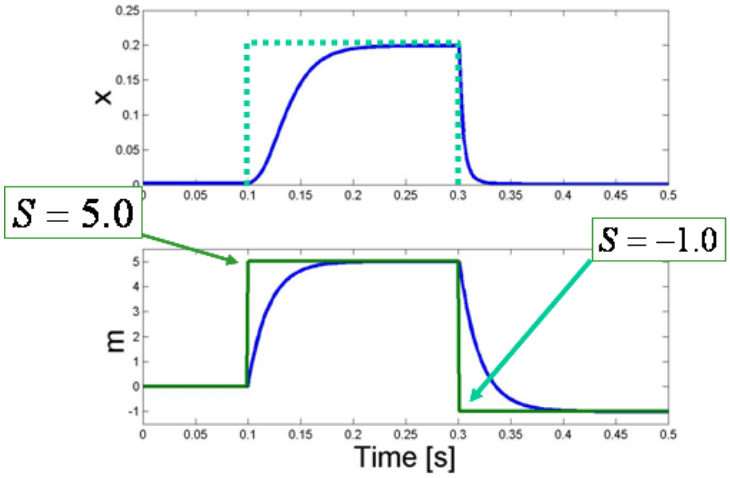


Figure 7-15: Typical activity pattern of a leaky-integrator neuron in response to an incoming input (synaptic activity) $S(t) = \sum_j w_{ij} \cdot x_j(t)$. Unpon receiving a steady input, the membrane potential *m* increases or decreases until it reaches the input value.

For a given steady or null external activation, the activity of the neuron decays or *leaks* exponentially according to the time constant τ following:

$$\frac{d}{dt} x_i(t) = -\frac{1}{\tau} x_i(t) \tag{6.77}$$

It is important to notice that this term corresponds to adding a self-connection to each neuron, and, as such provides a memory of the neuron's activity over time.

Similarly to the static case, one can show that the network is bound to converge under some conditions. But, in most cases, the complexity of the dynamics that results from combining several neurons whose dynamics follows a first order differential equation, as given in (6.77), becomes quickly intractable analytically.

The continuous time Hopfield network is a powerful to tool to store sequences. It has been used in numerous applications in robotics, e.g. learning to recognize and reproduce complex patterns of motion. The strength of this modeling comes from the fact that the complexity of the pattern that emerges from mixing different first order differential equations as given in (6.76) becomes very rapidly untractable.

Consider the case whereby the neuron received a single steady input *S* with no self-connection, i.e. with no decay term. The behavior of such a neuron is a linear differential equation that can be solved analytically and is given by:

$$\begin{aligned} m(t) &= (m_0 - S)\, e^{-t/\tau} + S \\ x(t) &= \frac{1}{1 + e^{-D\cdot(m(t)+b)}} \end{aligned} \tag{6.78}$$

The system grows until reaching a maximum as shown in Figure 7-16.

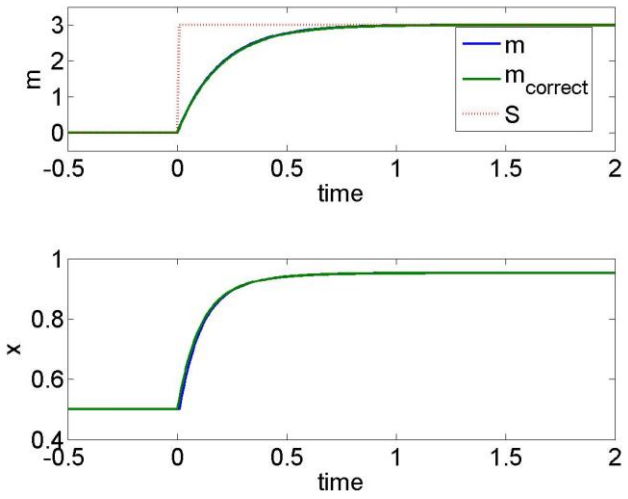


Figure 7-16: Dynamics of a single Leaky-Integrator neuron with no self-connection

The behavior of a single leaky-integrator neuron with a self-connection is much more complex. It follows a *nonlinear* differential equation that cannot be solved analytically. To study convergence of the neuron, one must then rely on finding the equilibrium points of the activation function of its membrane potential. This is given by:

$$\begin{aligned} &\frac{dm}{dt} = 0 \quad \Rightarrow \quad m - w_{11}\sigma(m+b) = S \\ &\text{where} \quad \sigma(z) = \frac{1}{1+e^{-Dz}} \ \text{ is the sigmoid function} \end{aligned} \tag{6.79}$$

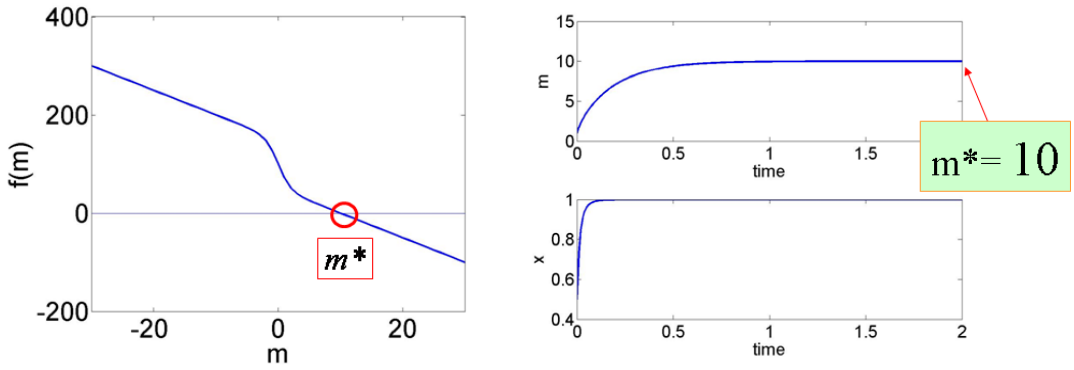


Figure 7-17: To determine the convergence of a leaky-integrator neuron with a self-connection, one must find numerically the value *m** for which the derivative of the membrane potential *m* is zero. Here we have a single stable point $m^* = 10$ for a steady input S=30 with a *negative* self-connection $w_{11} = -20$.

The zeros of the derivative correspond to equilibrium points. These may however be stable or unstable. A stable point is such that if it slightly pushed away from the stable point, it will eventually come back to its equilibrium. In contrast, an unstable point is such that a small perturbation in the input may send the system away from its equilibrium. In Leak-Integrator Neurons, this is easily achieved and depends on the value of the different parameters, but especially that of the self-connection. Figure 7-17 and Figure 7-18 illustrate these two cases.

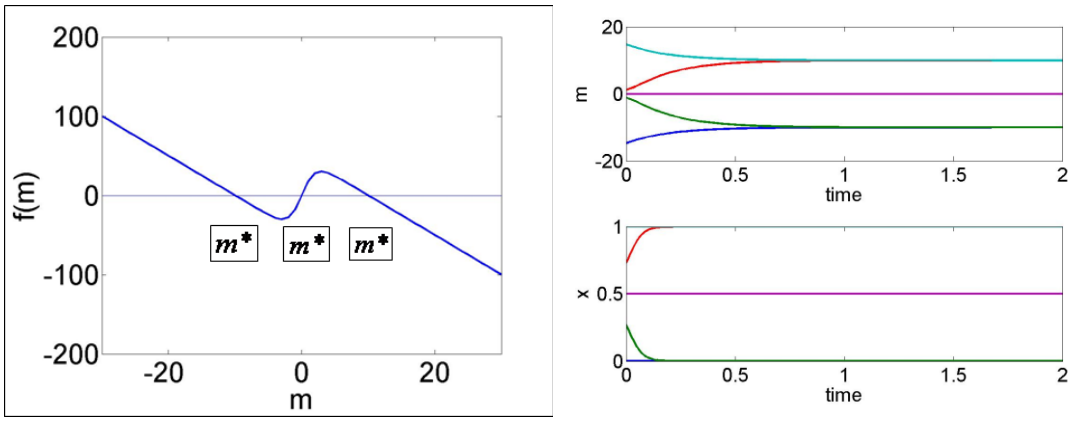


Figure 7-18: Example of a leaky-integrator neuron for a steady input S=30 with a *positive* self-connection $w_{11} = 20$. The system has three equilibrium points at m=0, m=-10 and m=10. m=0 is an *unstable* point, whereas m=-10 and m=10 are two stable points. This can be seen by observing that the slope of m(s) around m=0 is positive, whereas it is negative for m=-10 and m=10 (see left figure).

Stability of the equilibrium points can be determined by looking at the direction of the slope of the derivative of the membrane potential around the equilibrium point, i.e.:

$$f(m) = \frac{dm}{dt} = \frac{1}{\tau}(-m + S + w_{11}x) \tag{6.80}$$

If the slope is negative, i.e. $\frac{\partial f(m)}{\partial m} < 0$(note that here we are exploring the function in input space and not in time), then the equilibrium point is stable. Intuitively, this is easy to see. The slope gives you the direction in which you are pulled. With a negative slope you are pulled back to where you were. With a positive slope you are pulled away from where you were, usually in the direction of another equilibrium point.

Given that the dynamics of a single neuron can already become very complex, the dynamics of groups of such neurons becomes rapidly intractable. In a seminal paper, Randall Beer (Beer Adaptive Behavior, Vol 3 No 4, 1995) showed how the dynamics of a group of only two interconnected leaky-integrator neurons could lead to four complex dynamics. Figure 7-19 shows the possible trajectories of values taken by the outputs of the two neurons. Such a plot is called a *phase plot.*

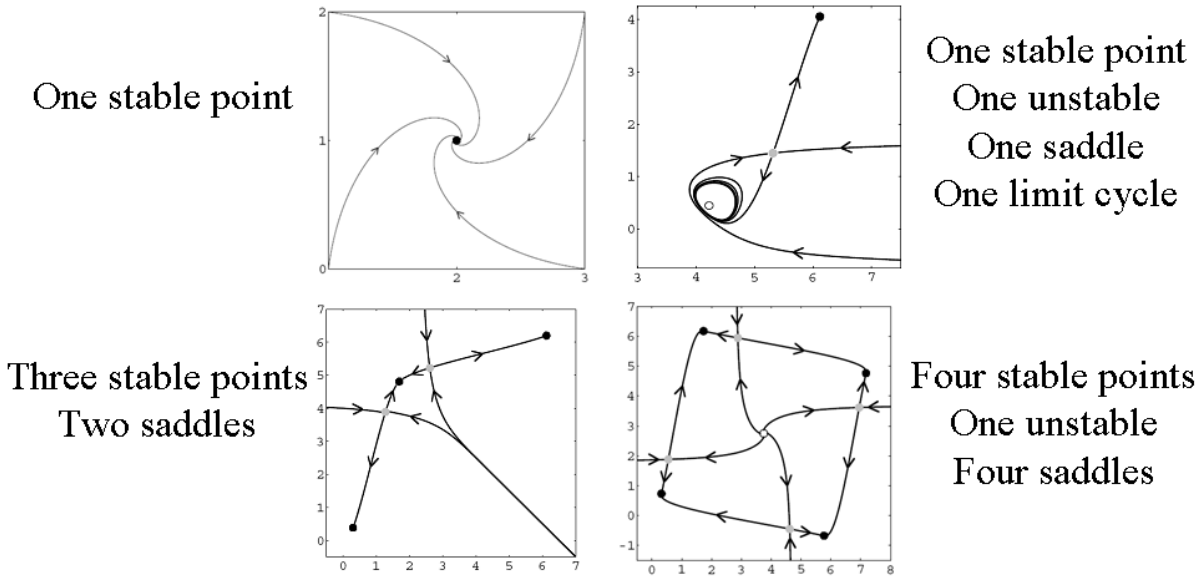
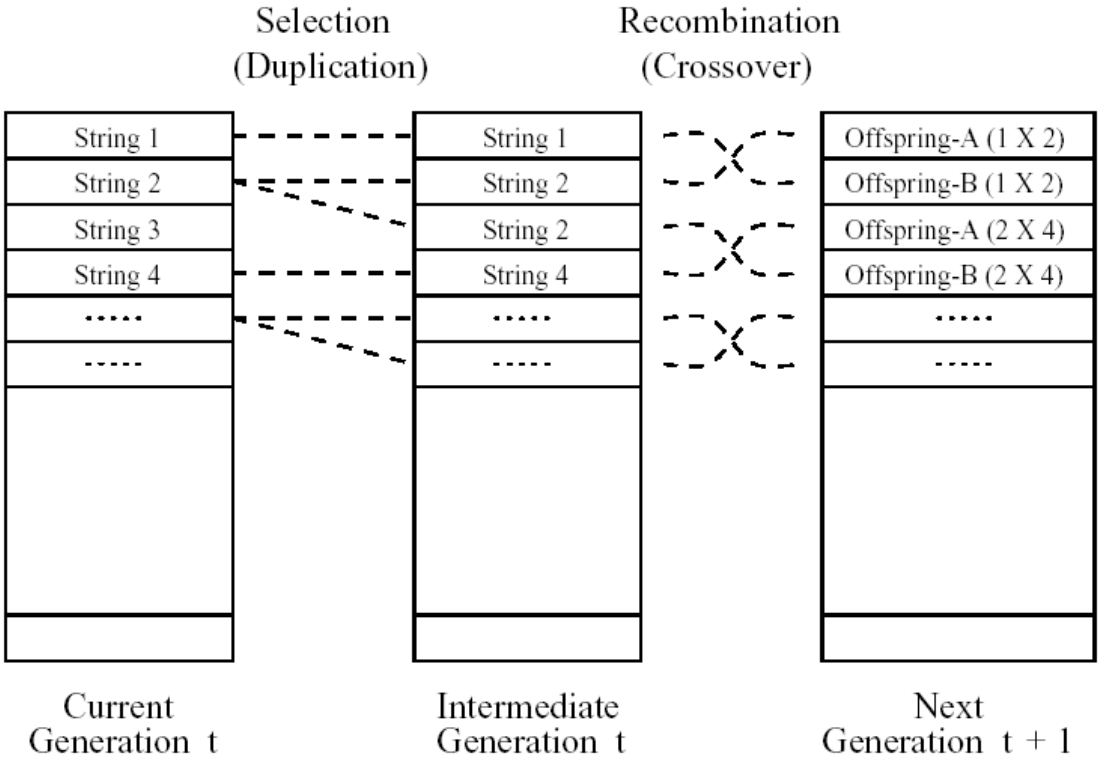


Figure 7-19: Possible dynamics that can be generated by a network composed of two leaky-integrator neurons (R. Beer, Adaptive Behavior, 1995).

8 Genetic Algorithms

The term *genetic algorithm* refers to a model introduced by John Holland in 1975 [Holland, 1975]. Genetic Algorithms are a family of computational models inspired by Darwin's principle of selective evolution. The algorithms encode a potential solution to a specific problem on a simple chromosome-like data structure and apply recombination operators to these structures so as to preserve critical information.

An implementation of a genetic algorithm begins with a population of typically random chromosomes. One, then, evaluates these structures and allocates reproductive opportunities in such a way that those chromosomes, which represent a better solution to the target problem, are given more chances to reproduce than those chromosomes, which are poorer solutions. The "goodness" of a solution is typically defined with respect to the current population.



Genetic algorithms are often viewed as optimization tool, although the range of problems to which genetic algorithms have been applied is quite broad and goes beyond simple function optimization. They are used to find solutions to complex systems in domains such as:

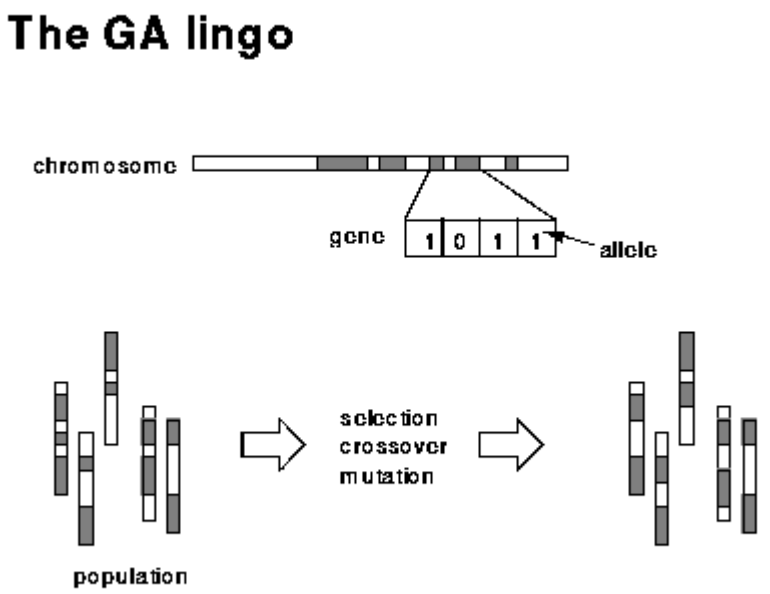
- Finances: Market predictors, risk evaluators
- Business: Scheduling, time and storage optimization
- Engineering: Dynamics of particles, fluids, etc.
- Biological Sciences: Dynamics of groups

In Robotics, GA have been used successfully to optimize parameters of controllers in order to evolve the Artificial Neural Networks Controllers. The GA is used to determine the weights or the topology of the network.

8.1 Principle

Genetic Algorithms provide a form of simulated evolution. The basic idea goes along the following steps:
- Start from a basic random population of parents
- Apply a process of selection
- Breed pairs of parents
- Produce pairs of children with only the good properties of the parents
- Reject bad children

The process leads to the evolution of a population of children that are better than the parents.



8.2 Encoding

To use a genetic algorithm, you must encode solutions to your problem in a structure that can be stored in the computer. This object is a genome (or chromosome). Thus, the first step before using a GA consists in a) defining a representation of the problem (this is often the most difficult step, as it requires some a priori knowledge on where the important features of the problem lie); b) defining the parameters to optimize (e.g. the weights in a neural networks); c) encoding the parameters into a *chromosome*; and d) creating a whole *population,* i.e. generate several random chromosomes.

You can use any representation for the individual genomes in the genetic algorithm. Holland worked primarily with strings of bits, but you can use arrays, trees, lists, or any other object. However, you must define genetic operators (initialization, mutation, crossover, comparison) for any representation that you decide to use

Each chromosome must represent a complete solution to the problem you are trying to optimize.

8.3 Breeding and Selection

In order to evaluate the goodness of each solution, you must define a selection criterion. This criterion is often referred to as the ***fitness function.*** The rest of the algorithm will breed the chromosomes, so as to maximize the fitness function. Applying iteratively ***crossover*** and ***mutation*** to the chromosomes in the population to generate new solutions does this.

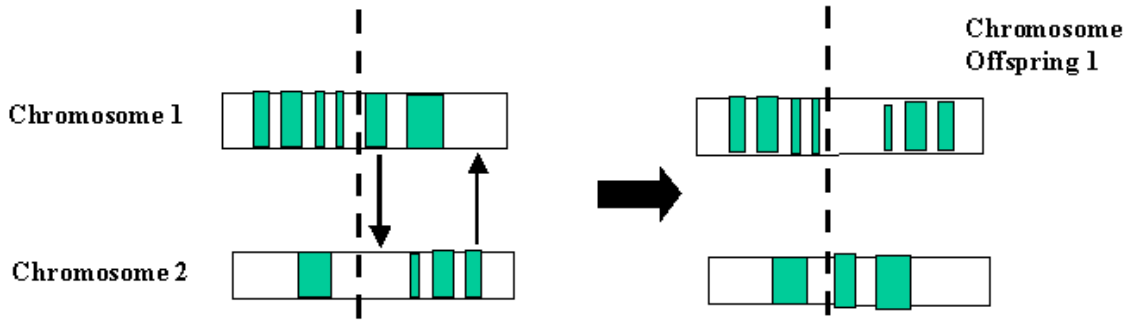


Figure 8-1: **Crossover:** Given 2 selected chromosomes; genetic material is swapped between them around a selected point.

Typically crossover is defined so that two individuals (the parents) combine to produce two more individuals (the children). But you can define asexual crossover or single-child crossover as well. The primary purpose of the crossover operator is to get genetic material from the previous generation to the subsequent generation.

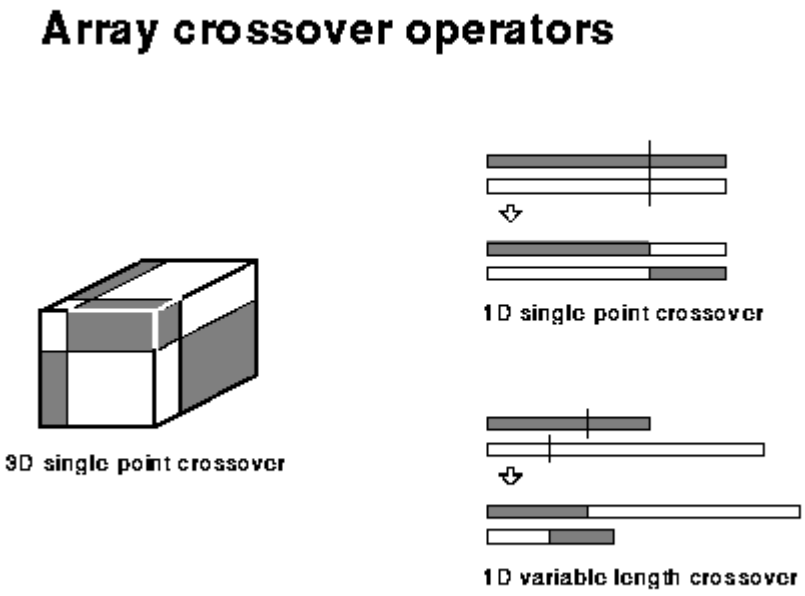


Figure 8-2: Mutation: Each location is mutated with a very small probability. When using a binary encoding, mutation let the bit go from 0 to 1 or vice-versa. In real number encoding, mutation adds or substracts a small random number.

The mutation operator introduces a certain amount of randomness to the search. It can help the search find solutions that crossover alone might not encounter.

Subsequently, GA uses various selection criteria so as to pick the best individuals for mating (and subsequent crossover), based on the value returned by the *fitness function*. For instance, parents can be chosen with some probability depending on:

- their fitness (Roulette-wheel selection)
- their rank according to the fitness (Rank-based selection)

The selection method determines how individuals are chosen for mating. If you use a selection method that picks only the best individual, then the population will quickly converge to that individual. So the selector should be biased toward better individuals, but should also pick some that are not quite as good (but hopefully have some good genetic material in them).

Some of the more common methods include roulette wheel selection (the likelihood of picking an individual is proportional to the individual's score), tournament selection (a number of individuals are picked using roulette wheel selection, then the best of these is (are) chosen for mating), and rank selection (pick the best individual every time). Threshold selection can also be effective.

Often the crossover operator and selection method are too effective and they end up driving the genetic algorithm to create a population of individuals that are almost exactly the same. When the population consists of similar individuals, the likelihood of finding new solutions typically decreases.

On one hand, you want the genetic algorithm to find good individuals, but on the other you want it to maintain diversity. A common problem in optimization (not just genetic algorithms) is how to define an objective function that accurately (and consistently) captures the effects of multiple objectives.

In general, genetic algorithms are better than gradient search methods if your search space has many local optima. Since the genetic algorithm traverses the search space using the genotype rather than the phenotype, it is less likely to get stuck on a local high or low.

8.4 The Algorithm

In summary, the Genetic Algorithm goes as follows:

1. Define a basic population of chromosomes.
2. Evaluate the fitness of each chromosome.
3. Select parents with a probability (rank-based, roulette-wheel)
4. Breed pairs of parents (cross-over).
5. Apply mutation on children.
6. Evaluate fitness of children.
7. Population size is kept constant through a rejection process. Throw away the worst solutions of the old population or of the old population + the new children.

8. Start again at point 3.
9. The GA stops when either satisfactory level of fitness is attained, or when the population is uniform (local minima).

8.5 Convergence

There is, to this day, no theoretical proof that a GA will eventually converge, nor that it will converge to the global optimum of the fitness function. There are, however, a number of “rules of the thumb” that one might follow to ensure good performance of the algorithm.

Convergence of a GA depends on choosing correctly the parameters. One should keep in mind the following:
• If the mutation is too small (not frequent enough and making small steps), there will not be enough exploration. Hence, the algorithm might get stuck in some local optimum.
• If, conversely, the mutation is too strong (too frequent and too important steps), the algorithm will be very slow to converge.
• Cross-over can slow down convergence, by destroying good solutions.
• Convergence depends importantly on the encoding and on the location of the genes on the chromosomes.
• The number of “children” produced at each generation determines the speed at which one explores the environment.

9 Annexes

9.1 Brief recall of basic transformations from linear algebra

9.1.1 Eigenvalue Decomposition

The decomposition of a *square* matrix *A* into eigenvalues and eigenvectors is known as eigen decomposition. The fact that this decomposition is always possible as long as the matrix consisting of the eigenvectors of *A* is square is known as the *eigen decomposition theorem.*

Let *A* be a linear transformation represented by a matrix A . If there is a vector $v \in \mathbb{R}^N \neq 0$ such that

$$Av = \lambda v \tag{7.1}$$

for some scalar λ , then λ is called the eigenvalue of A with corresponding (right) eigenvector v .
Letting A be a $N \times N$ square matrix:

$$\begin{bmatrix} a_{11}\ a_{12}\ ...\ a_{1N} \\ a_{21}\ a_{22}\ ...\ aN \\ ... \\ a_{N1}\ a_{N2}\ ...\ a_{NN} \end{bmatrix} \tag{7.2}$$

with eigenvalue λ , then the corresponding eigenvectors satisfy

$$\begin{bmatrix} a_{11}\ a_{12}\ ...\ a_{1N} \\ a_{21}\ a_{22}\ ...\ a_{2N} \\ ... \\ a_{N1}\ a_{N2}\ ...\ a_{NN} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ ... \\ v_N \end{bmatrix} = \lambda \begin{bmatrix} v_1 \\ v_2 \\ ... \\ v_N \end{bmatrix} \tag{7.3}$$

which is equivalent to the homogeneous system

$$\begin{bmatrix} a_{11}\text{-}\lambda & a_{12} & ... & a_{1k} \\ a_{21} & a_{22}\text{-}\lambda & ... & a_{2k} \\ ... & & & \\ a_{N1} & a_{N2} & ... & a_{NN}\text{-}\lambda \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ ... \\ v_N \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ ... \\ 0 \end{bmatrix} \tag{7.4}$$

Equation (7.4) can be written compactly as

$$\left(A-\lambda I\right)v=0 \tag{7.5}$$

where I is the identity matrix. As shown in Cramer's rule, a linear system of equations has nontrivial solutions iff the determinant vanishes, so the solutions of equation (7.5) are given by

$$\det\left(A-\lambda I\right)=0 \tag{7.6}$$

This equation is known as the *characteristic equation* of A , and the left-hand side is known as the *characteristic polynomial*.

For example, for a 2×2 matrix, the eigenvalues are

$$\lambda_{\pm}=\frac{1}{2}\left[\left(a_{11}+a_{22}\right)\pm\sqrt{4a_{12}a_{21}+\left(a_{11}-a_{22}\right)^{2}}\right] \tag{7.7}$$

If all *N* eigenvalues are different, then plugging these back in 6.5 gives $N-1$ independent equations for the *N* components of each corresponding eigenvector, and the system is said to be nondegenerate. If all or some of the eigenvalues are identical, then the system is said to be degenerate and the eigenvectors are not linearly independent. In such cases, the additional constraint that the eigenvectors be orthogonal,

$$\left(e^{i}\right)^{T}e^{j}=0 \tag{7.8}$$

can be applied to yield additional constraints, thus allowing solution for the eigenvectors.

If all the eigenvectors are linearly independent, then the eigenvalue decomposition

$$A=V\Lambda V^{-1} \tag{7.9}$$

Where the columns of *V* are composed of the eigenvectors of *A* and Λ is a diagonal matrix composed of the associated eigenvalues.

9.1.2 Singular Value Decomposition (SVD)

When the eigenvectors are not linearly independent, then *V* does not have an inverse (it is thus *singular)* and such decomposition does not exist. The eigenvalue decomposition consists then in finding a similarity transformation such that:

$$A=U\Lambda V \tag{7.10}$$

With U,V two orthogonal (if real) or unitary (if complex) matrices and Λ a diagonal matrix. Such decomposition is called *singular value decomposition* (SVD). SVD are useful in sofar

that *A* represents the mapping of a *N*-dimensional space onto itself, where *N* is the dimension of *A.*

An alternative to SVD is to compute the *Moore-Penrose Pseudoinverse* $A^{\#}$ of the non invertible matrix *A* and then exploit the fact that, for a pair of vectors z and c, $z = A^{\#}c$ is the shortest length least-square solution to the problem $Az = c$ .Methods such as PCA that find the optimal (in a least-square sense) projection of a dataset can be approximated using the pseudoinverse when the transformation matrix is singular.

9.1.3 Frobenius Norm

The Frobenius norm of an $m \times n$ matrix *A* is given by:

$$\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2} \tag{7.11}$$

9.2 Recall of basic notions of statistics and probabilities

9.2.1 Probabilities

Consider two variables *x* and *y* taking discrete values over the intervals [x_1,…, x_M] and [y_1,…, y_N] respectively, then

$P(x = x_i)$ is the probability that the variable x takes value x_i, with:

i) $0 \le P(x = x_i) \le 1, \quad \forall i = 1,...,M,$

ii) $\sum_{i=1}^{M} P(x = x_i) = 1.$

The same two above properties applies to the probabilities $P(y = y_j), \ \forall j = 1,...N.$

Some properties follow from the above:
Let $P(x = a)$ be the probability that the variable *x* will take the value *a.* If $P(x = a) = 1$, *x* is a constant with value *a.* If *x* is an *integer* and can take any value *a* between $[1, N] \in \mathbb{N}$ with equal probability, then the probability that *x* takes value *a* is $P(x = a) = \frac{1}{N}$

Joint probability:
The *joint probability* that the two events A (variable *x* takes value *a*) *and* B(variable *y* takes value *b*) occuris expressed as:

$$P(A,B) = P(A \cap B) = P\big((x = a) \cap (y = b)\big) \tag{7.12}$$

Conditional probability:

P(A| B) is the *conditional probability* that *event A will take place if event B already took place*

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)} \tag{7.13}$$

It follows that:

$$P(A \cap B) = P(A \mid B)P(B) \tag{7.14}$$

By the same reasoning, we have:

$$P(A \cap B) = P(B \mid A)P(A) \tag{7.15}$$

Hence,

$$P(A \mid B)P(B) = P(B \mid A)P(A) \tag{7.16}$$

Bayes' theorem:

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)} \tag{7.17}$$

Marginal Probability:

The so-called *marginal probability* that variable x will take value x_i is given by:

$$P_x(x = x_i) = \sum_{j=1}^{N} P(x = x_i, y = y_j) \tag{7.18}$$

To compute the marginal, one needs to know the joint distribution of variables x and y. Often, one does not know it and one can only estimate it.Note that if x is a multidimensional random variable, then the marginal is a joint distribution over the random variable spanned by x.

The joint distribution is far richer than the marginals. The marginals of N variables taking K values corresponds to N(K-1) probabilities, while their joint distribution corresponds to K^{N-1} probabilities.

9.2.2 Probability Distributions, Probability Density Function

p(x) is the probability density function of the variable $x \in \mathbb{R}$. p(x) is continuous, Lebesgue integrable, non-negative and normalized:

$$\begin{aligned} p(x) &\geq 0, \quad \forall x \\ \int_{-\infty}^{\infty} p(x)dx &= 1 \end{aligned} \tag{7.19}$$

9.2.3 Expectation

If *x* an take a set if discrete value in *X* and follow with probability *P(x),* then:

$$\mu = E\{x\} = \sum_{x \in X} xP(x) \tag{7.20}$$

The *expectation* of *x, E{x},* is the mean μ of its distribution:

$$\mu = E\{x\} = \int_X x \cdot p(x) \cdot dx \tag{7.21}$$

9.2.4 Variance and Covariance

The *variance* σ^2 of a distribution measures the amount of spread of the distribution around its mean:

$$\sigma^2 = Var\{x\} = E\left\{(x-\mu)^2\right\} = E\left\{x^2\right\} - \left[E\{x\}\right]^2 \tag{7.22}$$

σ is the *standard deviation* of *x.*

For two random variables *x* and *y*, one can measure the level of correlation between the two variables through the *co-variance:*

$$\text{cov}(x,y) = E\{x,y\} - E\{x\}E\{y\} \tag{7.23}$$

$E\{x,y\} = \int xyp(x,y)dxdy$ is the expectation of the joint distribution.

9.2.5 Distribution Function or Cumulative Distribution Function

The distribution function $D_x\left(x^*\right) = P\left(x \leq x^*\right)$ also called the cumulative distribution function (CDF) describes the probability that a random variable *x* takes on value less than or equal to a number *x*.*

If *p(x)* is the probability density function of *x* then:

$$P\left(x \leq x^*\right) = D_x\left(x^*\right) = \int p(x)dx \tag{7.24}$$

$p(x)\,dx$ ~ probability of x to fall within an infinitesimal interval $[x, x + dx]$.

The probability that the variable x takes a value in the subinterval [a,b] is equal to:

$$P(a \le x \le b) = \int_a^b p(x)dx$$

9.2.6 Joint and Conditional Probability Distribution

For two random variables x and y with joint distribution p(x,y), the *conditional probability* of *y* given x is given by:

$$p(y \mid x) = \frac{p(x,y)}{p(x)} \Leftrightarrow p(y \mid x) = \frac{p(x \mid y)p(y)}{p(x)} \tag{7.25}$$

9.2.7 Marginal Probability Distribution or Marginal Density

For two random variables x and y with joint distribution p(x,y), the *marginal density* of x is given by:

$$p_x(x) = \int p(x,y)dy$$

9.2.8 Statistical Independence

If x and y are two random variables, generated by the distribution $p(x)$ and p(y), then x and y are said to be *mutually independent*, if their joint density $p(x,y)$ is equal to the product of their density functions:

$$p(x,y) = p(x)p(y) \tag{7.26}$$

9.2.9 Uncorrelatedness

x and y are said to be *uncorrelated*, if their covariance is zero, i.e. $\operatorname{cov}(x,y) = 0$. If the variables are generated by a distribution with zero means, then uncorrelatedness is equal to product of their separate expectations:

$$\begin{aligned} &\operatorname{cov}(x,y) = E\{x,y\} - E\{x\}E\{y\} = 0 \\ &\Leftrightarrow E\{x,y\} = E\{x\}E\{y\} \end{aligned} \tag{7.27}$$

Independence is in general a much stronger requirement than uncorrelatedness. Indeed, if the *variables* are independent, for any transformation f_1 and f_2

$$E\{f_1(x)f_2(y)\} = E\{f_1(x)\}E\{f_2(x)\} \tag{7.28}$$

9.2.10 Uniform and Gaussian PDF

We here recap the definition of the uniform and Gaussian distribution, the two most classical *parametric* distribution probabilities. These distributions are *parametric* because their equation is known and depends only on a finite set of parameters. Using such distribution to model arbitrary distribution is advantageous as it amounts to determine only

the parameters of the distribution. In contrast, non-parametric distribution require to keep in memory all datapoints to perform a precise estimate of the distribution.

- **The Uniform distribution** is usually defined over an interval where the random variable takes the same *uniform* density of probability.

$$p(x)=\begin{cases}0 & \text{if} \quad a<x \\ \dfrac{1}{b-a} & \text{if} \quad a\le x\le b, \quad \text{with } x,\ a,\ b\in\mathbb{R} \\ 0 & \text{otherwise}\end{cases} \tag{7.29}$$

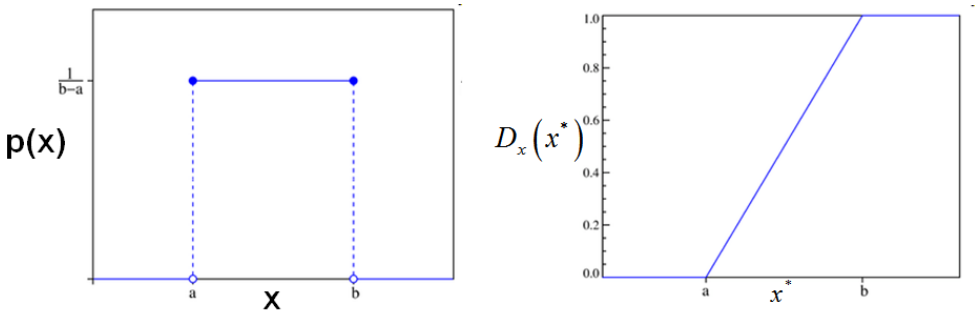


Figure 9-1: Pdf and Cumulative Distribution for the Uniform Distribution

The uniform pdf is unbounded above and can reach infinity as the b-a interval goes to zero.

- In one dimension, **the Gaussian distribution, usually called Normal distribution**

$$p(x)=\frac{1}{\sigma\sqrt{2\pi}}e^{\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)} \tag{7.30}$$

To emphasize that the Gaussian pdf depends on two parameters, the mean and variance, one sometimes writes $p(x;\mu,\sigma)$, or equivalently $p(x\,|\,\mu,\sigma)$.

- **The multi-dimensional Gaussian or Normal distribution** has a pdf given by:

$$p(x;\mu,\Sigma)=\frac{1}{(2\pi)^{\frac{N}{2}}|\Sigma|^{\frac{1}{2}}}e^{\left(-(x-\mu)^T\Sigma^{-1}(x-\mu)\right)} \tag{7.31}$$

if x is N-dimensional, then
μ is a $N-$dimensional mean vector
Σ is a $N\times N$ covariance matrix

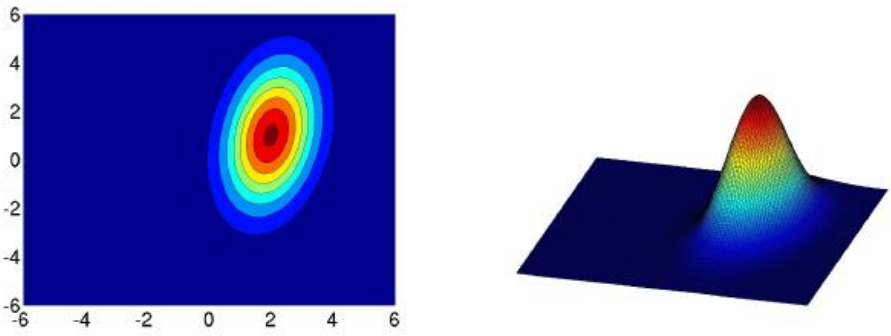


Figure 9-2: A 2-dimensional Gaussian distribution. The 2D oval-shape representation (left) is obtained by projecting the density onto the 2D axes of the random variable *X*. The color code represent surfaces of 1 standard deviation (std) in red, 2 std in light blue, 3 std in darker blue, etc.

The marginal and conditional distribution in a multivariate Gaussian distribution are all Gaussian distributions, see Figure 9-3. This result is quite important and is used extensivly in machine learning to exploit the fact that the marginal and conditional can be parametrized and their parameters can be estimated through techniques applied to Gaussian distribution, such as expectation-maximization.

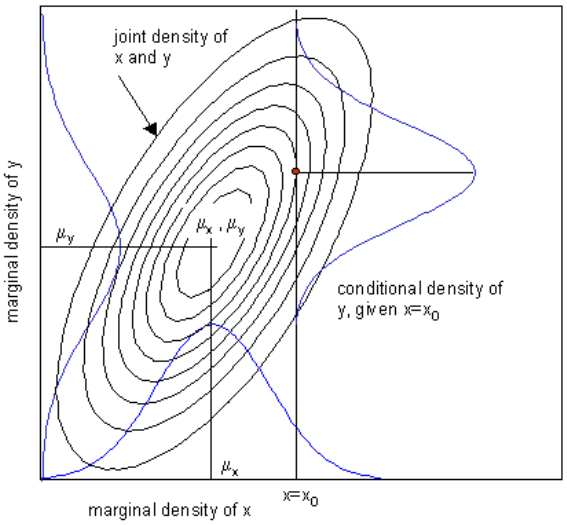


Figure 9-3: How the Joint, Marginal, and Conditional distributions are related in a bi-dimensional Gaussian distribution over the random variable x and y.

9.2.11 Likelihood Function

The likelihood function L (usually simply called the *likelihood*) is an important quantity in statistics and in machine learning. It determines the total probability of observing the set,

$X = \{x^i\}_{i=1}^{M}$, of *M* observation of the random variable x given that x follows a distribution $p(x)$.

$$L(X) = p(x^1, x^2, ..., x^M) \tag{7.32}$$

For a set *X* of M datapoints $X = \{x^i\}_{i=1}^{M}$, where each datapoint is *identically* and *independently* distributed **(i.i.d)** according to the density $p(x)$, the likelihood is given by:

$$L(X) = \prod_{i=1}^{M} p(x^i) \tag{7.33}$$

The likelihood is used to help to determine how to improve the parameter of the model of the pdf to increase the likelihood that the particular choice of parameter (and of density) represent the set of data.

For a parametrized density $p(x;\mu,\sigma)$, with parameters μ,σ, the Likelihood Function is the likelihood of the data X given the parameters: $L(\mu,\sigma \mid X) = p(X \mid \mu,\sigma)$.

9.2.12 Kullback-Leibler Distance

The most common method to measure the difference between two probability distributions *p* and *q* is to us the Kullback-Leibler distance (DKL), sometimes known as *relative entropy*:

$$D(p \,||\, q) = \int p(x) \log \frac{p(x)}{q(x)} dx \tag{7.34}$$

DKL is always positive $D(p \,||\, q) \geq 0$. It is zero if and only if *p*=*q*, i.e. ifthe variables are statistically independent. Note that in general the relative entropy or DKL is not symmetric under interchange of the distributions *p* and q: in general $D(p \,||\, q) \neq D(q \,||\, p)$. Hence, **DKL is not strictly a distance**. The measure of relative entropy is very important in pattern recognition and neural networks, as well as in information theory, as will be shown in other parts of this class.

9.3 Information Theory

The mathematical formalism for measuring the *information content*, *I*, of an arbitrary measurement I is due largely to a seminal 1948 paper by Claude E. Shannon. Within the context of sending and receiving messages in a communication system, Shannon was interested in finding a measure of the information content of a received message. Shannon's measure of information aimed at conveying the impression that an unlikely event contains more information than a likely event. The information content of an outcome *x* [illegible]

$$I(x) = \log_2 \frac{1}{P(x)} = -\log_2 P(x) \tag{7.35}$$

where P(x) is the probability of x.

Shannon's information measure is binary coded. To get an intuitive feeling of this concept, consider the case where the probability of x is uniform over the interval $[1,2] \in \mathbb{N}$. Then, the information conveyed in the observed event x=1 is equal to:

$$I(x=1) = -\log_2 \frac{1}{2} = \log_2 2 = 1$$

If, on the other hand, x is uniformly distributed over the interval $[1,8] \in \mathbb{N}$, making each of the occurrences of x 3 times less likely. Then, the information conveyed in observing *x=1,* is 3 times more important:

$$I(x=1) = \log_2 8 = \log_2 2^3 = 3$$

9.3.1 Entropy

The notion of information is tightly linked to the physics notion of entropy. The entropy of the information on *x* is a measure of *the uncertainty* or of *the information content* of a set of *N* observations of *x*:

$$H(x) = -\sum_{i=1}^{N} P(x) \log P(x) \tag{7.36}$$

If we take again our earlier example, the entropy of *x*, when x is uniform over the interval [1,2] is equal to $H(x) = -\sum_{i=1}^{2} \frac{1}{2} \log_2 \frac{1}{2} = 1$

[illegible] $H(x) = -\left(\frac{1}{4}\log_2 \frac{1}{4} + \frac{3}{4}\log_2 \frac{3}{4}\right) = 0.8$ [illegible]

[illegible]is more uncertainty in a situation where every outcome is equally probable.

In the continuous case, one defines the *differential entropy* of a distribution *f* of a random variable *x* as:

$$h(x) = -\int_S f(x) \log f(x) dx \tag{7.37}$$

where *S* is the support set of *x.* For instance, consider the case where *x* is uniformly distribution between [0,a], so that its density function is

$$f(x) = \begin{cases} \frac{1}{a} & \text{if } 0 \le x \le a \\ \frac{1}{b} & \text{if } a < x \le a+b \end{cases},$$

Then, its differential entropy is:

$$h(x) = -\int_0^a \frac{1}{a} \log \frac{1}{a} - \int_a^{a+b} \frac{1}{b} \log \frac{1}{b} = \log a + \log b = \log(a \cdot b)$$

This definition is appealing, as, if *a* or *b* have a large value, the spread of the distribution will be large.

One can show that the *Gaussian distribution* is the distribution with maximal entropy for a given variance. In other words, this means that the gaussian distribution is the ``most random'' or the least structured of all distributions. Entropy is small for distributions that are clearly concentrated on certain values, i.e., when the variable is clearly clustered, or has a probability distribution function that is very ``spiky''.

9.3.2 Joint and conditional entropy

If *x* and *y* are two random variables taking values between $[0, I] \in \mathbb{N}$ and $[0, J] \in \mathbb{N}$ respectively, and $p(i, j)$ the joint probability that *x* takes value *I* and *y* takes value *j*, then the entropy of the joint distribution is equal to:

$$H(x, y) = -\sum_{i,j} P(i, j) \log P(i, j) \tag{7.38}$$

We can then derive the equation for the conditional entropy.

$$\begin{aligned} H(x \mid y) &= -\sum_j P(j) H(x \mid y = j) \\ &= -\sum_j P(j) \sum_i P(i \mid j) \log P(i \mid j) \\ &= -\sum_{i,j} P(i, j) \log P(i \mid j) \end{aligned}$$

Finally, conditional and joint entropy can be related as follows:

$$H(x,y) = H\left(x\right) + H\left(y \mid x\right) \tag{7.39}$$

[illegible]

$$H(x,y) = H(x) + H(y) \quad \text{iff } P(x,y) = P(x)P(y)$$

9.3.3 Mutual Information

The mutual information between two random variables *x* and *y* is denoted by

$$\begin{aligned} I(x,y) &= H\left(x\right) - H\left(y \mid x\right) \\ &= H\left(y\right) - H\left(x \mid y\right) \\ &= H\left(x\right) + H\left(y\right) - H\left(x,y\right) \end{aligned} \tag{7.40}$$

Mutual information is a natural measure of the dependence between random variables. **If the variables are independent, they provide no information on each other**. An important property of mutual information is that one can define an invertible linear transformation

$$y = Wx \text{ , s.t. } I\left(y\right) = H\left(y\right) - H\left(x\right) - \log\left|\det W\right|$$

9.3.4 Relation to mutual information

Mutual information can be related to the notion of correlation through canonical correlation analysis. Since information is additive for statistically independent variables and since canonical variates are uncorrelated, the mutual information between two independent variate *x* and *y* is the sum of mutual information between all the canonical variates $x_i \text{ and } y_i$ (i.e. the canonical projections of x and y). For Gaussian variables this means:

$$I\left(x;y\right) = \frac{1}{2}\log\left(\frac{1}{\prod_i\left(1-\rho_i^2\right)}\right) = \frac{1}{2}\sum_i \log\left(\frac{1}{\left(1-\rho_i^2\right)}\right) \tag{7.41}$$

9.3.5 Kullback-Leibler Distance

The most common method to measure the difference between two probability distributions *p* and *q* is to us the Kullback-Leibler distance (DKL), sometimes known as *relative entropy*:

$$D(p\,||\,q) = \sum_i p(i) \log \frac{p(i)}{q(i)} = E\left(\log \frac{p(i)}{q(i)} \right) \tag{7.42}$$

DKL is always positive $D(p\,||\,q) \geq 0$. It is zero if and only if *p*=*q*, i.e. if the variables are statistically independent. Note that in general the relative entropy or DKL is not symmetric under interchange of the distributions *p* and q: in general $D(p\,||\,q) \neq D(q\,||\,p)$. Hence, **DKL is not strictly a distance**. The measure of relative entropy is very important in pattern recognition and neural networks, as well as in information theory, as will be shown in other parts of this class.

9.4 Estimators

9.4.1 Maximum Likelihood

Machine learning techniques often assume that the form of the distribution function is known and that sole its parameters must be optimized to fit at best a set of observed datapoints. It then proceeds to determine these parameters through maximum likelihood optimization.

The principle of *maximum likelihood* consists of finding the optimal parameters of a given distribution by maximizing the likelihood function of these parameters, equivalently by maximizing the probability of the data given the model and its parameters, e.g.:

The method of ***maximum likelihood*** is a method of point estimation that uses as an estimate of the parameters of the distribution by maximizing the likelihood function. If μ *is* the parameter of the density and $X = \left\{ x^i \right\}_{i=1}^{M}$ the set of observed instances of the random variable *x*, the probability of having observed the set X given the underlying distribution *p(x*| μ) is:

$$L\left(\mu \mid X\right) = P\left(X \mid \mu\right) \tag{7.43}$$

the value of *X* that maximizes $L\left(\mu\right)$ is the **maximum-likelihood estimate** of μ . In order to find the maximum, one will compute the derivative of *L*: $\frac{\partial L}{\partial \mu} = 0$. Often, it is simpler to compute the derivative of the *logarithm* of the likelihood function, the *log-likelihood.*

If $p\left(x\right)$ is a multi-dimensional Gaussian function, with parameters μ, Σ . The mean and covariance matrices can be estimated in closed-form by Maximum-Likelihood:

$$\begin{aligned} &\max_{\mu,\Sigma} L\left(\mu,\Sigma \mid X\right) = \max_{\mu,\Sigma} p\left(X \mid \mu,\Sigma\right) \\ &\frac{\partial}{\partial \mu} p\left(X \mid \mu,\Sigma\right) = 0 \quad \text{and} \quad \frac{\partial}{\partial \Sigma} p\left(X \mid \mu,\Sigma\right) = 0 \end{aligned} \tag{7.44}$$

In most problems, it is not possible to find an analytical expression for $\hat{\mu}$. One, then, has to proceed to an estimation of the parameter through an iterative procedure called Expectation-Maximization (EM), see next.

9.4.2 EM-Algorithm

EM is an algorithm for finding *maximum likelihood* estimates of parameters in *probabilistic* models, where the model depends on unobserved latent variables. EM alternates between performing an expectation (E) step, which computes the expected value of the latent variables, and a maximization (M) step, which computes the maximum likelihood estimates of the parameters given the data and setting the latent variables to their expectation. It is

used in a vast variety of Machine Learning techniques, such as K-means, Gaussian Mixture Models and Hidden Markov Models.

If $X = \left\{x^i\right\}_{i=1}^{M}$ is the set of *M* observed data, $Z = \left\{z^i\right\}_{i=1}^{K}$, the set of *K* discrete variables and Θ the set of parameters of the pdf function, the likelihood is:

$$L\left(\Theta \mid X, Z\right) = P\left(X, Z \mid \Theta\right)$$

At first, it may not be very clear why one should distinguish between the parameters Θ and the latent variables Z. The notion of latent variable offers a mean to express the fact that the distribution of the observables can be explained by a finite set of distributions. Each observable x^i may have been generated by one or a combination of the *K distributions* associated to each of the K latent variables. Introducing Latent variables allows to proceed to a reduction of the original problem. The expression of the density $p\left(x\right)$ can be reduced to a finite set of densities. For instance, in Gaussian Mixture Models, the density $p\left(x\right)$ is expressed as a weighted linear combination of K Gaussian functions:

$$p\left(x;\Theta\right) = \sum_{k=1}^{K} \alpha_k p_k\left(x;\mu^k,\Sigma^k\right) \text{ with } \Theta = \{\mu^1,\Sigma^1,...\mu^K,\Sigma^K\}, \alpha_k \in [0,1]..$$

Each Gaussian function represents a latent variable with associated parameters (mean and Covariance matrix). In Hidden Markov Models, the latent variables correspond to the hidden states. Each hidden state is associated two pdf to explain the probablity of the observation and the probability of transiting across states.

In E-M, the number of latent variables Z is fixed during the entire E-M process. Only the parameters are updated. However, the latent variables play a role in the computation of the likelihood.

The EM-algorithm is an iterative process that aims at updating the:

- Step 0: Make an initial estimate of the parameters $\hat{\Theta}$ and of the latent variables Z. (***Initialization***)
- Using current $\hat{\Theta}$**,** compute the expectation of the complete data likelihood $L\left(\hat{\Theta} \mid X, Z\right)$. (***E-step***)
- Find (and update) $\hat{\Theta}$ to maximize the expectation (***M-step***).

EM stops when there is no improvement in Likelihood above some threshold. Of course, this all depends on the threshold. Since the likelihood function is unbounded, fixing this threshold requires some care. Even when taking the log of the likelihood function, variation in the likelihood are hard to quantify and depend on the number of datapoints and type of density used. Good practice is to plot the decrease in the log of the likelihood and determine visually a plateau, which one can then translate in a numerical value.

EM is a description of a class of related algorithm, not of a particular algorithm. EM is a recipe or meta-algorithm, which is used to devise particular algorithms. The *Baum-Welch algorithm* is an example of an EM algorithm applied to *Hidden Markov Models*. Another example is the K-means clustering algorithm.

It can be shown that an EM iteration does not decrease the observed data likelihood function, and that the only *stationary points* of the iteration are the stationary points of the

observed data likelihood function. In practice, this means that an EM algorithm will converge to a *local maximum* of the observed data likelihood function.

EM proceeds thus iteratively and is particularly suited for parameter estimation under incomplete data or missing data situations. By using the EM procedure, the so-called **marginal** (or incomplete-data) **likelihood** $L(\Theta \mid X) = P(X \mid \Theta)$ is obtained by computing the average or expectation of the complete-data likelihood with respect to the missing data using the current parameter estimates (E-step), then the new parameter estimates are obtained by maximizing the marginal likelihood (M-step).

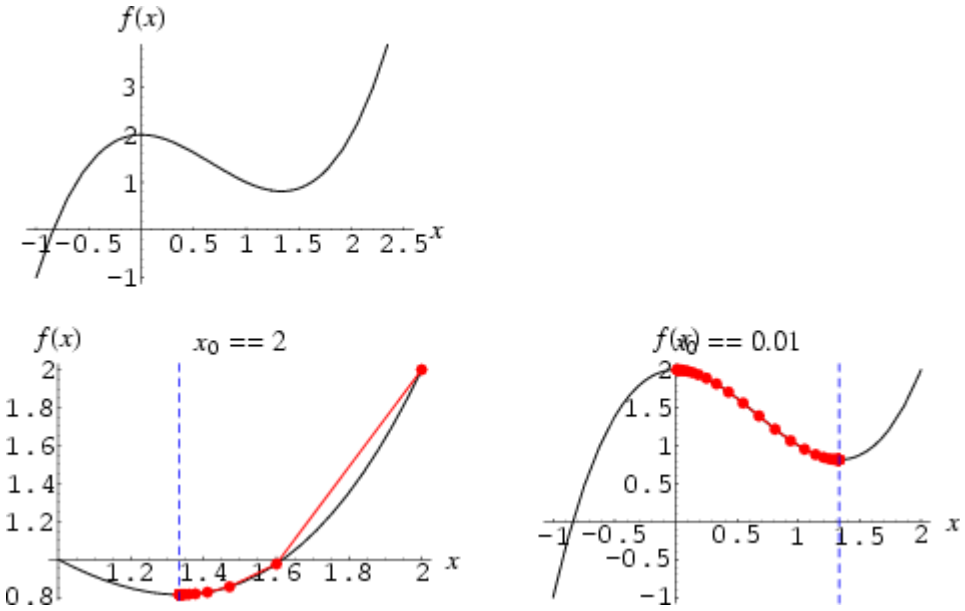
For a complete description of EM, we refer the reader to: Moon, Todd K. "The expectation-maximization algorithm." *Signal processing magazine, IEEE* 13.6 (1996): 47-60.

9.4.3 Gradient descent

(from *Mathworld)*

Gradient descent is an incremental *hill-climbing* algorithm that approaches a minimum or maximum of a function by taking steps proportional to the gradient at the current point.

An algorithm for finding the nearest *local minimum* of a function, which presupposes that the *gradient* of the function can be computed. The method of steepest descent, also called the gradient descent method, starts at a point P_{i+1} and, as many times as needed, moves from P_i to P_{i+1} by minimizing along the line extending from $-\nabla f(P_i)$ the local downhill gradient.



When applied to a 1-dimensional function *f*(*x*), the method takes the form of iterating

$$x_i = x_{i-1} - \varepsilon f(x_{i-1}) \tag{7.45}$$

from a starting point x_0 for small $\varepsilon > 0$ until a fixed point is reached. The results are illustrated above for the functions $f(x) = x^3 - 2x^2 + 2$ with $\varepsilon = 0.1$ and starting point $x_0 = 2$ and 0.01, respectively.

This method has the severe drawback of requiring a great many iterations for functions, which have long, narrow valley structures. In such cases, a *conjugate gradient method* is preferable.

9.4.4 Conjugate Gradient descent

The conjugate gradient method is an algorithm for finding the nearest local minimum of a function of *n* variables, which presupposes that the gradient of the function can be computed. It uses conjugate directions instead of the local gradient for going downhill. If the vicinity of the minimum has the shape of a long, narrow valley, the minimum is reached in far fewer steps than would be the case using the method of steepest descent.

The conjugate gradient method is an effective method for symmetric positive definite systems. It is the oldest and best-known non-stationary method. The method proceeds by generating vector sequences of iterates (i.e., successive approximations to the solution), residuals corresponding to the iterates, and searches directions used in updating the iterates and residuals. Although the length of these sequences can become large, only a small number of vectors need to be kept in memory. At each iteration step of the method, two inner products are performed in order to compute update scalars that are defined to make the sequences satisfy specific orthogonality conditions. On a symmetric positive definite linear system these conditions imply that the distance to the true solution is minimized in some norm.

Conjugate gradient technique:

1. Let P_0 (with the position vector $\vec{x}_0$) be the starting point of the search; the local gradient at P_0 is

$$\vec{g}_0 \equiv -\nabla f(\vec{x}_0) = -\mathbf{A}^T \cdot \left[\mathbf{A} \cdot \vec{x}_0 - \vec{b}\right] \tag{2.1}$$

The next "low point" P_1 is then situated at

$$\vec{x}_1 = \vec{x}_0 + \lambda_1 \vec{g}_0 \tag{2.2}$$

with

$$\lambda_1 = \frac{|\vec{g}_0|^2}{|\mathbf{A} \cdot \vec{g}_0|^2} \,. \tag{2.3}$$

2. From P_1 we proceed *not* along the local gradient

$$\vec{g}_1 = -\mathbf{A}^T \cdot \left[\mathbf{A} \cdot \vec{x}_1 - \vec{b}\right] \tag{2.4}$$

but along the gradient conjugate to $\vec{g}_0$, i.e.

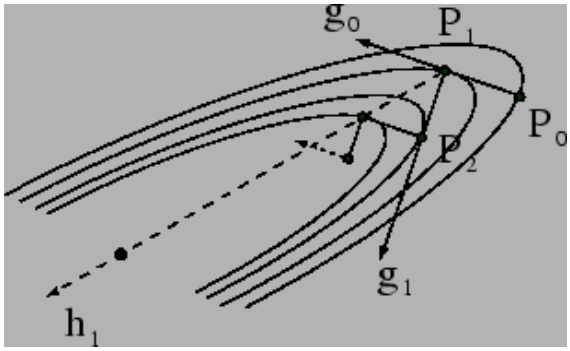
$$\vec{h}_1 = \vec{g}_1 - \frac{(\mathbf{A} \cdot \vec{g}_1) \cdot (\mathbf{A} \cdot \vec{g}_0)}{|\mathbf{A} \cdot \vec{g}_0|^2} \,. \tag{2.5}$$

The low point along this path is at

$$\vec{x}_2 = \vec{x}_1 + \lambda_2 \vec{h}_1 \,, \tag{2.6}$$

with

$$\lambda_2 = \frac{\left|\vec{g}_1 \cdot \vec{h}_1\right|}{\left|\mathbf{A} \cdot \vec{h}_1\right|^2} \tag{2.7}$$



10 References

General Textbooks on Machine Learning:

- ***Machine Learning: a Probabilistic Perspective,*** K. P. Murphy, *MIT Press, 2013.*
- ***Machine Learning,*** Tom Mitchell, *McGraw Hill, 1997.*
- ***Pattern Classification*** Richard O. Duda, Peter E. Hart, David G. Stork, 2nd Edition, *Willey* 2007
- ***Elements of Machine Learning,*** Pat Langley, Morgan Kaufmann, 1996.

General Textbooks on Neural Networks:

- ***Information Theory, Inference and Learning Algorithms***, David J.C Mackay, *Cambridge University Press, 2003*.
- ***Artificial Neural Networks and Information Theory*****,** Colin Fyfe, *Tech. Report, Dept. of Computing and Infortion Science, The University of Paisley, 2000.*
- ***Neural Networks,*** Simon Haykin, *Prentice Hall International Editions, 1994.*

Textbooks on Topics of Machine Learning

- ***Learning with Kernels,*** *B. Scholkopf and A. Smola, MIT Press 2002*
- ***A Tutorial on Support Vector Machines for Pattern Recognition,*** *C.J.C. Burges, Data Mining and Knowledge Discovery, 2, 121167 1998.*
- **Reinforcement Learning: An Introduction,**. R. Sutton & A. Barto, A Bradford Book. MIT Press, 1998.
- ***Cluster Analysis,*** Copyright StatSoft, Inc., 1984-2004

Textbooks on Topics of Neural Networks

- ***Independent Component Analysis,*** A. Hyvarinen, J. Karhunen and E. Oja, *Wiley Inter-Sciences. 2001.*
- ***Self-Organizing Maps,***Teuvo Kohonen, *Springer Series in Information Sciences, 30, Springer. 2001.*
- ***Neural Networks for Pattern Recognition*****,** Bishop, C.M. New York: Oxford University Press, 1996.
- **Adaptation In Natural and Articial Systems**. University of Michigan Press. Holland, J (1975)

Other recommended readings on some basic neural networks:

Hopfield Network:

J.J. Hopfield**, "*Neural networks as physical systems with emergent collective computational abilities*",** Proc. National Academy of Sciences of USA, vol.72, pp.2554-2558, 1982.

J.J. Hopfield**. *Pattern recognition computation using action potential timing for stimulus representation*.** Nature, 376:33--36, 1995

Boltzmann Machine:

G Barna and K Kaski 1989 **Variations on the Boltzmann machine** *J. Phys. A: Math. Gen.***22** 5143-5151

Ackley D H, Hinton G E and Sejnowski T J 1985 ***Cognitive Sci. 9*** 147.

Reinforcement Learning:

Leslie Pack Kaelbling **&**Michael L. Littman and Andrew W. Moore, **Reinforcement Learning: A Survey,** *Journal of Artificial Intelligence Research*, Volume 4, 1996., 1996.

State-of-the-art

State-of-the-art research in Machine Learning can be found in the proceedings of the International Conference in Machine Learning (ICML) and the Neural Information Processing Symposium (NIPS), but also in publications in major journals such as the Machine Learning Journal, the IEEE Transactions on Signal processing, IEEE Transactions on Pattern Analysis, IEEE Transactions on Pattern Recognition, the Journal of Machine Learning Research.

State-of-the-art research in Neural Networks can be found in the proceedings of the International Conference on Artificial Neural Networks (ICANN), the Internation Joint Conference on Neural Networks (IJCNN), the European Conference on Artificial Neural Networks (ECANN) and in major journals such as Neural Networks, Neural Computation, IEEE transactions on Neural Networks, Neurocomputing. We list major sites for on-line resources in ML next.

10.1.1 ML Resources:

One most prominent resource for machine learning is found at the official website of the *machine learning society* http://www.machinelearning.org/index.html
This website keeps a repository of all papers published at International Machine Learning Conference (ICML), one of the two major venues for research papers in ML.

Below is a list of other repositories which the interested reader is invited to visit to keep up to date on research advances in ML and for getting access to various databases for benchmarking the development of new algorithms.

ML List http://www.ics.uci.edu/~mlearn/MLList.html	"Archives of the Machine Learning List"
ML Repository http://www.ics.uci.edu/~mlearn/MLRepository.html	"This is a repository of databases, domain theories and data generators that are used by the machine learning community for the empirical analysis of machine learning algorithms."
MLnet http://www.mlnet.org/	Machine Learning network online information service. "This site is dedicated to the field of machine learning, knowledge discovery, case-based reasoning, knowledge acquisition, and data mining."
KDnet http://www.kdnet.org/control/index	"The KDNet (= Knowledge Discovery Network of Excellence) is an open Network of participants from science, industry and the public sector. The major purpose of this international project is to integrate real-life business problems into research discussions and to collaborate in shaping the future of Knowledge Discovery and Data Mining."
NIPS http://nips.cc/	Neural Information Processing Conference – on-line repository of all research papers on theoretical ML
Pascal http://www.pascal-network.org/	Network of excellence on Pattern Recognition, Statistical Modelling and Computational Learning

Major venues for publishing research on machine learning are:

Journals:

- Machine Learning
- IEEE Transactions on Signal processing
- IEEE Transactions on Pattern Analysis
- IEEE Transactions on Pattern Recognition
- The Journal of Machine Learning Research
- Data Mining and Knowledge Discovery
- ACM Transactions on Information Systems

- Neural Networks
- Neural Computation
- Biological Cybernetics

Conferences:

- ICML: Int. Conf. on Machine Learning
- NIPS: Neural Information Processing Conference

- COLT: Annual Conference on Learning Theory
- IJCAI: Int. Joint Conf. on Artificial Intelligence

- ICANN: Int. Conf. on Artificial Neural Networks
- ESANN: European Conf. on Artificial Neural Networks
- IJCNN: Int. Joint Conf. on Neural Networks